Minimizing Cost by Reducing Scaling Operations in Distributed Stream Processing

Michael Borkowski, Christoph Hochreiner, Stefan Schulte
Distributed Systems Group
TU Wien, Vienna, Austria
{m.borkowski, c.hochreiner, s.schulte}@infosys.tuwien.ac.at

ABSTRACT

Elastic distributed stream processing systems are able to dynamically adapt to changes in the workload. Often, these systems react to the rate of incoming data, or to the level of resource utilization, by scaling up or down. The goal is to optimize the system's resource usage, thereby reducing its operational cost. However, such scaling operations consume resources on their own, introducing a certain overhead of resource usage, and therefore cost, for every scaling operation. In addition, migrations caused by scaling operations inevitably lead to brief processing gaps. Therefore, an excessive number of scaling operations should be avoided.

We approach this problem by preventing unnecessary scaling operations and over-compensating reactions to short-term changes in the workload. This allows to maintain elasticity, while also minimizing the incurred overhead cost of scaling operations. To achieve this, we use advanced filtering techniques from the field of signal processing to pre-process raw system measurements, thus mitigating superfluous scaling operations. We perform a real-world testbed evaluation verifying the effects, and provide a break-even cost analysis to show the economic feasibility of our approach.

PVLDB Reference Format:
Michael Borkowski, Christoph Hochreiner, Stefan Schulte. Minimizing Cost by Reducing Scaling Operations in Distributed Stream Processing. *PVLDB*, 12(7): 724-737, 2019.
DOI: https://doi.org/10.14778/3317315.3317316

1. INTRODUCTION

Elasticity is a major concern in modern data stream processing (DSP) systems [19]. In short, an elastic system is capable of rapidly scaling up during times of increased load, and scaling down during times of reduced load, instead of constantly under-provisioning computational resources, i.e., operating with less capacity than required to handle the input data rate, or over-provisioning resources, i.e., operating with more capacity than required [1, 25]. Using the capability of elasticity, stream processing engines (SPEs) dynamically adapt to changes in the input data rate during runtime, reducing cost while aiming at meeting a predefined Quality of Service (QoS) level [16].

Certain properties of DSP systems can be monitored in order to reach scaling decisions. On the one hand, the observed properties may be extrinsic to the system, e.g., the rate or type of incoming data or queries [3, 20, 46]. On the other hand, these properties may be intrinsic to the system, e.g., CPU load [16], memory utilization [7], network traffic [49], or its overall performance [4]. Generally, every scaling operation requires resources by itself, i.e., it incurs a delay, during which the DSP system must wait for the computational resource to be reachable, consumes energy, leads to computational overhead [14, 35], and therefore increased cost. Additional DSP operators, i.e., software responsible for the processing of data within the SPE, must be deployed on computational resources, such as Virtual Machines (VMs). Therefore, while being crucial for elasticity, intensive operations such as scaling and migration should be kept at a minimum [7, 9, 22, 35].

Current approaches to scaling in DSP assume thresholds of resource utilization, such as CPU load, as the foundation for scaling decisions [3, 7, 21, 36]. The DSP system is maintained between those thresholds. For any metric exceeding the upper threshold, additional resources are necessary, and new DSP operator instances are activated (scale up). However, a threshold-based scaling approach can result in frequent scaling operations. This, in turn, incurs an overhead of resource usage and cost [3, 7]. In certain cases, this overhead is necessary in order to benefit from the additional computing power, to avoid under-provisioning, or to save energy by scaling down. However, excessive scaling also increases the risk of unnecessary cost [7, 35].

Metrics such as the rate of incoming data can be considered as time series, containing both long-term trends in data rate, as well as short-term variances (spikes and valleys) [3]. The long-term trend can be the development of input data depending, for instance, on the time of day (e.g., peak hours) or time of year (e.g., summer holidays), while short-term spikes might stem from spontaneous and short-lived events like bursts in network communication. The latter represent noise that we aim to ignore for scaling decisions.

Following this, we propose to improve classic threshold-based scaling in DSP by improving the scaling mechanism's reaction to load changes. Instead of relying on one metric and employing simple threshold-based scaling, we observe both multiple intrinsic metrics of an SPE (e.g., CPU and memory utilization) and extrinsic metrics, representing the

Proceedings of the VLDB Endowment, Vol. 12, No. 7
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3317315.3317316

environment of an SPE (e.g., rate of incoming data). From these values, we derive an estimated *true* inner state, neglecting noise (i.e., the short-term variance) by separating it from the long-term trend. Based on this estimated state, more stable and robust scaling decisions can be made. The intuition is to reduce the amount of scaling decisions while keeping the DSP system highly adaptive to load changes.

Overall, the contributions of this work are as follows:

- We specify a control loop, describing the relationship between the controlled system, its environment, measurable feedback, and a scaling controller.
- We define a formal model for pre-processing measured resource utilization values in order to reduce the number of scaling operations performed by DSP operators.
- We propose a concrete scaling mechanism applying an Extended Kalman Filter (EKF) [27, 29], in order to use knowledge about state changes in the operator's environment to improve scaling decisions.
- We evaluate our approach in detail using a testbed with an image processing workload taken from the research field of biomedical engineering [23].

The remainder of this paper is structured as follows: In Section 2, we present our approach to minimize the number of scaling operations in DSP systems. We then evaluate the approach in Section 3, and discuss the results in Section 4. In Section 5, we review related work. Finally, we conclude and give an overview of future work in Section 6.

2. APPROACH

The goal of our work is to minimize the amount of scaling operations, while maintaining rapid elasticity, to avoid the cost of overly frequent scaling operations [3, 7, 14, 35]. We regard each operator within a DSP separately, and measure the amount of incoming data (*data rate*) and the system state, with the goal of reaching a scaling decision. This decision can be to either (a) scale up by starting more operator instances, (b) to scale down by shutting down instances, or (c) to remain in the same state. Each operator can be executed using one or more operator instances, determined by the scaling decision. We denote the set of operator instances for any given operator as B, with instances $b_1, b_2, b_3, \ldots, b_n$, where $n = |B|$.

This work extends traditional signal filtering using EKF in order to reach better scaling decisions. The system state measured by the EKF-based filter is represented in a vector. Multiple state variables (metrics) can be included in this vector, and our general filtering approach poses no limits to this number. Possible metrics include the system's CPU load [16], its memory utilization [7], network traffic [49], throughput and queue sizes [12], or other metrics determining the system's overall performance [4].

In the following sections, we discuss our approach. First, in Section 2.1, we define a control loop, describing the relationship between the controlled system (the stream processing operator), its environment (incoming data), measurable feedback (the system state), and a scaling controller. Second, in Section 2.2, we show our formal model for time series filtering. Section 2.3 gives a detailed description of the EKF used, with Section 2.4 showing the bootstrapping process.

Table 1: Notation

Model	
B	Set of operator instances
$b_1, b_2, \ldots$	Individual operator instances in B
Θ^-	Scale-down threshold
Θ^+	Scale-up threshold
Time Series and Filtering	
T	Set of all measurement times
$t_1, t_2, \ldots$	Individual timestamps in T
Z	Time series of system state measurements
$Z_{t_1}, Z_{t_2}, \ldots$	Individual state measurements at time t in Z
$\lambda(\cdot)$	Filtering (smoothing) function
$\lambda_Z(t)$	λ at time t based on history Z
Z'	Filtered system state time series
$Z'_{t_1}, Z'_{t_2}, \ldots$	Individual filtered measurements at time t in Z'
Measurements and EKF Model	
D_t	Data rate at time t
ΔD_t	Date rate change from $t-1$ to t
$u_t = (D_t, \Delta D_t)$	System input at time t
x_t	System state at time t
$\hat{x}_t^*$	System state estimation for time t, a priori
$\hat{x}_t$	System state estimation for time t, a posteriori
z_t	Measurement (observation) at time t
$\hat{z}_t$	Measurement estimation for time t
w_t	System noise at time t
v_t	Measurement noise at time t
Q	System noise covariance
R	Measurement noise covariance
$f(x_t, u_t)$	State transition function
$h(x_t)$	Measurement function (in our case $h(x) = x$)
EKF-Internal State Matrices	
P_t	Estimation error at time t
F_t	Jacobian matrix of $f(\hat{x}_t, u_t)$
H_t	Jacobian matrix of $h(\hat{x}_t)$
G_t	Kalman gain at time t
Miscellaneous	
$\mathcal{N}(\mu, \sigma^2)$	Normal distribution, variance σ^2 around μ
X^T	Transpose of matrix X
X^{-1}	Inverse of matrix X

Section 2.5 discusses the parameters and complexity of our approach. Table 1 gives an overview of the notation used.

2.1 Control Loop

We treat the observed operator, the rate of incoming data, and the scaling mechanism which controls the scaling decisions of the operator, as a control system. Our approach involves creating a *closed control loop* [15], shown in Figure 1. During operation, the *system* (the stream processing operator) is under constant supply of input data, measured by its rate. Since we cannot control the amount of incoming data, we define this as the operator's *environment*[1]. The stream processing operator is controlled by the scaling mechanism, being the *controller* in our control loop. The controller is responsible for making scaling decisions, i.e., defining whether the operator must be scaled up, scaled down, or can remain unchanged. During the processing of data, the operator is changing the system state (e.g., CPU load or memory utilization), constituting the *feedback*, which is measured by the controller. The controller therefore has two sources of information for performing the scaling decisions: the rate of incoming data, and the system state of the operator.

2.2 Filtering Model

As discussed above, we measure the system state over time, and base our scaling decisions on the measured values.

[1]Known as *disturbance* in other literature [29].

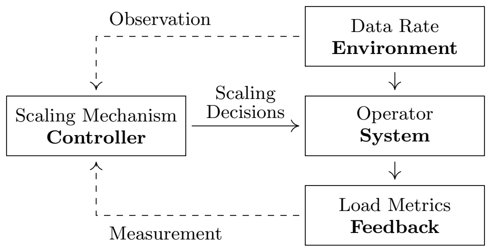

Figure 1: Overview of the proposed approach, modeled as a control loop.

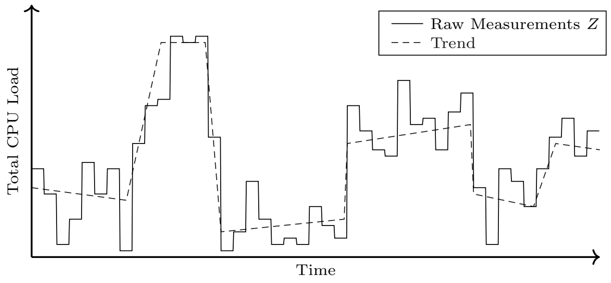

Figure 2: Long-term load trend (dashed) and actual, measured values (solid).

However, we employ a filtering of the raw measurements to create a smoother version of the measurement curve.

The time series of recorded measurements of the system state is denoted as Z. In practice, those measurements are offset from a trend by a certain noise. This noise can have numerous causes, ranging from disturbances at the operating system level, hypervisor strategies at the VM level, or workload shared with other applications. All of these aspects cause the system state (e.g., CPU load or memory utilization) to exhibit high variance. Nevertheless, a certain trend is always present, e.g., a highly demanding processing node will have a given baseline (trend) of load throughout its operation. Figure 2 demonstrates a scenario, using the CPU load as an example of fluctuating system state.

Naturally, if a stream processing system bases scaling decisions purely on the raw data, an excessive amount of scaling operations occurs [7, 14, 35]. This is shown in Figure 3, where the fluctuating CPU load measurements (top graph) lead to a high amount of scaling operations (bottom graph). Our approach applies filters to this process to reduce the number of scaling operations, i.e., to reduce the number of steps in the *operators* line in Figure 3.

Therefore, we formally define our approach as follows. We regard a history of raw system state measurements, Z, at various points in time t out of all measured times T, where Z_t is the measured state at time t:

$$T = \{t_0, t_1, \ldots, t_n\} \tag{1}$$

$$Z = \bigcup_{t \in T} Z_t = \{Z_{t_0}, Z_{t_1}, \ldots, Z_{t_n}\} \tag{2}$$

Based on the raw measurements $Z_t \in Z$, we employ a filter, which we denote as $\lambda(\cdot)$, and apply this filter to each Z_t. This application is performed at every given measurement time t and has access to all other measurement values in Z, with the practical limitation that it can only access

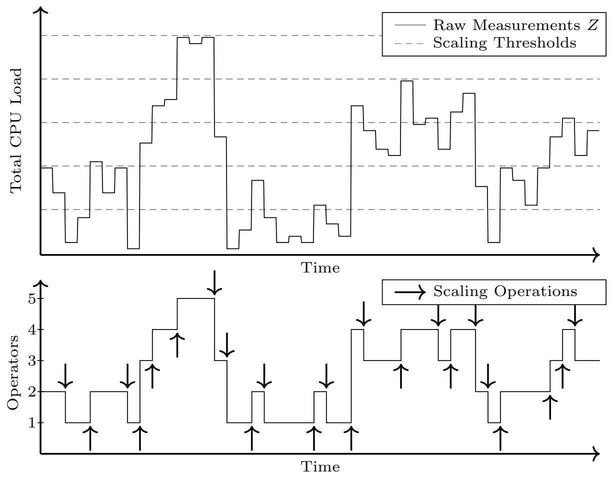

Figure 3: Scaling of operators according to thresholds of the actual load, resulting in 23 scaling operations.

past measurements. We therefore define $\lambda_Z(t)$ as the filtered value of Z at time t, given all other values $Z_i \in Z$ where $i \leq t$. For λ, various filters can be used. In our work, we use an EKF as a smoothing filter, which we will describe in Section 2.3. Other filters like Linear Smoothing (LS) [42], Total Variation Denoising (TVD) [41], or simpler versions of the EKF [3] are used for smoothing in literature, but in most cases not with regard to DSP (see Section 5).

We define the set of filtered measurements Z':

$$\forall Z_t \in Z : Z'_t = \lambda_Z(t) \tag{3}$$

$$Z' = \bigcup_{t \in T} Z'_t = \{Z'_{t_0}, Z'_{t_1}, \ldots, Z'_{t_n}\} \tag{4}$$

Figure 4 shows a possible resulting graph of the same data rate measurements as shown in Figure 3, using a filter, along with the resulting scaling behavior of the system. When compared to Figure 3, it becomes clear that the amount of scaling operations has decreased. Note that this approach does not guarantee that the system state is met with correct scaling at each point in time. There is also the possibility of under-provisioning or over-provisioning for short periods in time, depending on the used filter, as is the nature of online filtering. However, we show in Section 3 that the proposed EKF-based filter performs sufficiently well.

2.3 Extended Kalman Filter

In the following, we describe the EKF used in our approach, along with the concrete state transition model.

The EKF [27] is a nonlinear generalization of the Kalman Filter (KF) [29]. Kalman-type filters work by defining models for state transitions of the system, as well as models for the observations (measurements) of the system. While regular KFs use purely linear transition models, i.e., matrices and linear algebra, the EKF generalizes the approach for nonlinear models. Instead of matrices, the EKF uses functions as transition models, and requires both the transition and observation function to be differentiable point-wise.

Note that the EKF is not the only suitable model usable for forecasting multivariate processes. Especially in

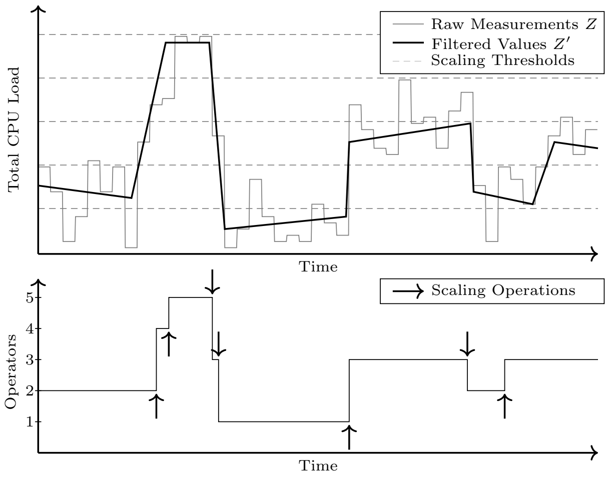

Figure 4: The same scenario, with additional filtering of data rate measurements. Instead of 23 scaling operations, the system only has to perform 7, while maintaining the same QoS level.

processes with a high potential for repeating patterns, autoregressive models such as ARMA or ARIMA are used [18, 44]. However, the unique advantage of EKF-based filters is that with some (even inaccurate) knowledge of the underlying system model, not only the measurements of system state are incorporated into the solution, but also the (very accurately known) system input. At the same time, the EKF maintains covariance matrices determining the current confidence into each data source (both system state and input).

The first steps for defining our EKF are as follows: The *system state* is denoted as x. This vector can include multiple state variables (metrics), such as CPU and memory utilization, network traffic, throughput or queue sizes. Since the state changes over time, we use x_t to indicate the state at time t. Furthermore, our system is controlled by an external input, which is the rate of data sent to the stream processing operator. We observe both the momentary data rate at time t, defined as D_t, as well as the change in data rate compared to the previous value, ΔD_t, with $\Delta D_t = D_t - D_{t-1}$. Together, we define the *input* to the stream processing operator at a given time t as $u_t = (D_t, \Delta D_t)$. Finally, since our operator is running on a real-world computer, and therefore is subject to certain fluctuations in performance, the state also encounters a particular *noise*. We denote this system noise as w, or, again, w_t for a given time t. Finally, we define our *state transition model*, which models the system state x_t, based on the previous system state x_{t-1}, the input u_{t-1}, and the system noise w_t:

$$x_t = f(x_{t-1}, u_{t-1}) + w_t \tag{5}$$

where $f(\cdot)$ represents the *state transition function*, which is based on the last state x_t and the system input u_t. The system noise w_t, according to the original definition of KFs [29], is assumed to be zero-mean Gaussian noise[2] with the covariance matrix Q:

$$w_t \sim \mathcal{N}(0, Q) \tag{6}$$

[2]We discuss this assumption of zero-mean Gaussian noise for our scenario in Section 3.3.

The state transition function $f(x, u)$ can be chosen independently of the remaining part of this state system. In our scenario, for simplicity, we use a linear state transition function, based on the current system state x, and the input $u = (D, \Delta D)$, defined as $f(x, u) = x + a \cdot D + b \cdot \Delta D$. However, due to the usage of EKF, a nonlinear state transition function could also be used if nonlinear dynamics are present and known. The vector parameters a and b define the sensitivity of the EKF to the input data rate, and must be defined according to the workload. Currently, these parameters are determined using ordinary least squares (OLS) linear regression. Future approaches may use more elaborate self-tuning techniques, e.g., Machine Learning (ML).

Next, we take into account the *measurement* of the system state. There are several mechanisms for measuring system load, e.g., stand-alone programs like `top` and `ps`, or APIs for direct measurements. The EKF definition includes a measurement function, which we denote as $h(\cdot)$. This function takes the system state x and transforms it to a measured value. This value is again subject to noise, this time, stemming from the measurement process itself. We call this the *measurement noise* and denote it as v_t. For physical sensors, this represents a measurement error or inaccuracy. In our scenario, this measurement error represents the inaccuracy of measuring CPU load. The resulting measurement is denoted as z_t, and defined as follows:

$$z_t = h(x_t) + v_t \tag{7}$$

where v_t, the measurement noise, is again assumed to be zero-mean Gaussian noise, and its covariance matrix is assumed to be R:

$$v_t \sim \mathcal{N}(0, R) \tag{8}$$

In contrast to physical sensors like temperature or light sensors, which often have nonlinear characteristics, or require additional unit conversion, we do not need transformations in the process of measurement. Therefore, we can simply define $h(x_t) = x_t$, and our measurement becomes:

$$z_t = x_t + v_t \tag{9}$$

Figure 5 gives an overview of the dynamics of the described state system. While the input to the system (u) is known but not controllable, the system's true state is hidden from the controller. This includes the noise influencing the system state itself (w), as well as the noise of the measurements (v). Only the result of the measurements (z) is visible to the controller. As described before, $u = (D, \Delta D)$, f is the state transition function, h is the identity function $h(x) = x$, $w \sim \mathcal{N}(0, Q)$, and $v \sim \mathcal{N}(0, R)$.

Note that the general definition of EKF allows both $f(\cdot)$ and Q, as well as $h(\cdot)$ and R to be dependent on the time t, i.e., the notations $f_t(\cdot)$, Q_t, $h_t(\cdot)$, and R_t are used, respectively. Since we use time-constant definitions for $f(\cdot)$, Q, $h(\cdot)$, and R in our approach, we drop the index t.

Therefore, our controller makes scaling decisions based on the rate of incoming data (u) and the (noisy) measurement of system state (z). As stated in Section 2.2, we do not directly use the measurement z, but instead, employ the estimation feature of the employed EKF, which yields an estimated version of the *next* system state, denoted as $\hat{x}$.

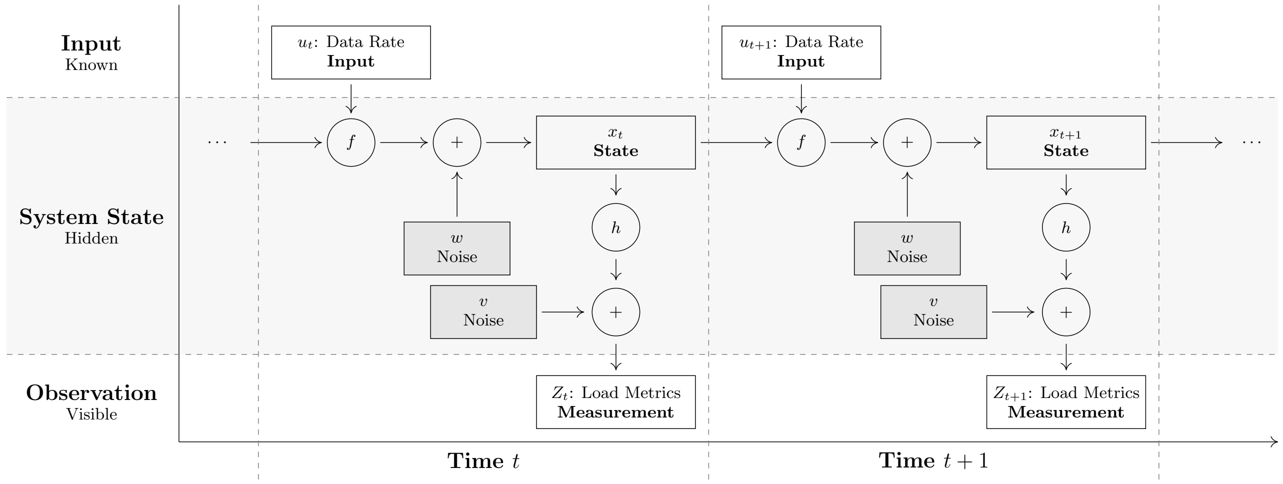

Figure 5: The state transition system we use as a base model, where the system input $u = (D, \Delta D)$ is the data rate and its change, f is the state transition function, x is the system state, z is the load measurement, h is the identity function $h(x) = x$, $w \sim \mathcal{N}(0, Q)$, and $v \sim \mathcal{N}(0, R)$.

The nature of EKF is that it performs a continuous loop of *predict-update* iterations. Given a current state, in the *predict* step, the EKF performs a prediction of the next system state. In addition, the EKF also provides the *prediction error.* Then, provided with a (noisy) measurement of the actual value, the EKF recalculates its prediction error, and provides a new prediction in the *update* step. As a result, the EKF is constantly correcting its prediction, provided with noisy measurements, while maintaining a balance between inaccuracy in measurements, as well as external disturbances. Furthermore, this process takes into account the *input* of the system, i.e., control variables which are manipulated externally. In our scenario, this is the amount of data rate and its change, i.e., $u = (D, \Delta D)$.

Predict Step: At any point in time t, from a given previous estimated system state $\hat{x}_{t-1}$ – either the initial state (see Section 2.4 for a description of bootstrapping in our approach) or the previously estimated state – and the previous system input u_{t-1}, the EKF derives both the estimated *a priori* next system state $\hat{x}^*_t$, the estimated next measurement $\hat{z}_t$, as well as the estimation error P_t:

$$\hat{x}^*_t = f(\hat{x}_{t-1}, u_{t-1}) \tag{10}$$
$$\hat{z}_t = h(\hat{x}^*_t) \tag{11}$$
$$P_t = F_{t-1}\, P_{t-1}\, F^T_{t-1} + Q \tag{12}$$

where F_{t-1} is the Jacobian matrix of f, i.e., the matrix of partial derivatives of the state transition function for the current state and input $f'(\hat{x}_{t-1}, u_{t-1})$, P_t is the prediction error (covariance matrix) at time t, and Q is the covariance of the system noise as defined above. P_t is computed by applying the Jacobian matrix of the state transition matrix F to the previous prediction error, then re-applying its transpose F^T, and finally adding the system noise covariance Q. This prediction error will later be used to calculate the *Kalman gain* G_t. Its definition is derived from the EKF proposal [27]. The initialization value for P_0 is discussed in Section 2.4.

Update Step: After a new measurement z_t is taken, the EKF updates its matrices and vectors to reflect the new data. First, the Kalman gain G_t is calculated, which is used to create the new *a posteriori* system state estimate $\hat{x}_t$. Note that the difference between the *a priori* estimate $\hat{x}^*_t$ and the *a posteriori* state estimate $\hat{x}_t$ is that the *a posteriori* state incorporates the new measurement (and therefore, new knowledge) into the value provided by the *a priori* state before the measurement.

The update step is calculated as follows:

$$G_t = P_t\, H^T (H\, P_t\, H^T + R)^{-1} \tag{13}$$
$$\hat{x}_t = \hat{x}^*_t + G_t(z_t - \hat{z}_t) \tag{14}$$

where H is the Jacobian matrix of h, i.e., the matrix of partial derivatives of the measurement function for the current measurement $h'(\hat{x}_t)$, and R is the covariance of the measurement noise as defined above. Again, this definition is derived from [27]. The Kalman gain G_t is used in the estimation of the next system state $\hat{x}_t$ and represents the (estimated) influence of the change in measurement on the actual system state.

Revisiting Section 2.2, we are now able to define the filtered version of z, i.e., $Z' = \lambda_Z(z_t)$, by using the cumulative output of the EKF estimations for each operator instance:

$$Z'_t = \lambda_Z(z_t) = \sum_{\mathrm{b} \in B} \hat{x}^b_t \tag{15}$$

where B, as defined at the beginning of Section 2, is the set of all operator instances for the operator type taken into account, b $\in B$ denotes the iteration over all operator instances, and $\hat{x}^b_t$ denotes the EKF estimation $\hat{x}$ at time t for the operator instance b.

2.4 Bootstrapping

First, the EKF must be initialized. Since especially at the beginning of the lifetime of an operator, a certain amount of time must be chosen in order for the operator to stabilize, we propose a simple bootstrapping process. In our work, we distinguish between a *cold start* and a *warm start.* If the operator has never been executed before, and therefore its behavior is unknown, a cold start is executed, and a default number of instances is initiated. In our current implementation, this default number is set to one, i.e., if the system has no knowledge about the operator, a single instance of it is spun up. In case the system has already used this operator before, and data about its behavior has been collected, we execute a warm start, and use the average number of instances of the operator used in the last run. This is done to use a value as close to the likely required scale as possible during the bootstrapping process.

After the initiation of the operator instances, we begin a two-step parameter bootstrapping. First, a *dead time* is implemented, during which no scaling decisions are taken, and only measurements of the input data rate (u) and the system state (z) are taken and collected. After the dead time, the EKF is initialized with the following parameters.

Measurement Noise Covariance Matrix R: Since we cannot distinguish between measurement noise and system noise just by measuring the system state (z), we propose calibration measurements using FakeLoad [43], which is a dedicated load generator. Given a relatively noiseless load generation, all measured variance represents measurement noise and constitutes our R.

Initial State Estimation $\hat{x}_0$: To initialize the state estimation, we use a simple weighted average of the measurement of system state (z) during the dead time. We weight the system state measurements by recentness, where each weight is indirectly proportional to the time passed since the measurement (i.e., its age):

$$\hat{x}_0 = \sum_{i=1}^{n} \frac{i}{\Delta_n} z_i \tag{16}$$

where n is the number of z measurements, z_i is the i^{th} measurement and Δ_n is the n^{th} triangular number[3].

Initial State Estimation Covariance Matrix P_0: This parameter determines the covariance of the prediction, i.e., gives a measurement of the confidence in the estimation of the initial system state $\hat{x}_0$. Since we derive $\hat{x}_0$ from a weighted mean of measurements of z during the dead time, we use the same technique to derive P_0:

$$P_0 = \sum_{i=1}^{n} \frac{i}{\Delta_n - 1} (z_i - \hat{x}_0)^2 \tag{17}$$

where $\Delta_n - 1$ represents *Bessel's correction* for an unbiased estimator of covariance [40].

System Noise Covariance Q: For the system noise covariance, we use the same value as for P_0, but reduced by the previously determined measurement noise:

$$Q = P_0 - R \tag{18}$$

where we assume that $P_0 > R$ always holds. The rationale behind this is that P_0, stemming from the observation during the dead time, should reflect both the system noise (Q) and the measurement noise (R). Note that while Q and R are constant in our approach, P_t is adapted by the EKF over time, so the relationship $Q = P_t - R$ only holds for $t = 0$. Afterwards, during the course of the operation of the EKF, as the EKF converges [27], P decreases over time.

After the dead time, we define an *ease-in time*, during which the EKF is executed, but its estimates are not yet used. Only after this second phase of the bootstrapping process, the EKF estimates are used for scaling decisions. The time durations used for both the dead time and the ease-in time are parameterizable. In our experiments, we find that 10 seconds are sufficient for both parameters.

2.5 Parameter and Complexity Analysis

In the following, we summarize the parameters required for our filtering approach, and discuss its computational complexity with respect to time and space.

We assume that the metrics selected for scaling (contained in the vector z) are decided in advance. For instance, in our evaluation in Section 3, we will use CPU and memory utilization metrics. Furthermore, we assume that the *input* to the system (u), is also defined in advance. In our evaluation, D and ΔD constitute this input. The measurements z are assumed to be performed with a certain accuracy, defined by a zero-mean Gaussian noise with covariance R. We show in Section 2.4 how to determine this parameter, and evaluate this in Section 3.3. Furthermore, the input transition function f constitutes a parameter of our approach. In our evaluation, we use the linear function $x + a \cdot D + b \cdot \Delta D$, where a and b are parameters determining the sensitivity of the EKF-based filter to the input data rate. Finally, the state x itself is assumed to be subject to zero-mean Gaussian system noise with covariance Q. Like R, Q constitutes a parameter, and in Section 2.4, we show how to determine its value. Finally, the duration of both the dead time and the ease-in time, also described in Section 2.4, constitute parameters relevant to the bootstrapping process.

We now analyze the computational complexity of our filtering approach. In the following, n denotes the number of elements of the system state vector x, and m denotes the number of elements in the measurement vector z. An EKF iteration (*predict-update*) is required every time new measurements are available. We use a measurement frequency of 2 Hz to remain well below the time required to spin up operator instances, and provide the EKF with sufficiently frequent data. The *predict* step, shown in (10)–(12), entails the application of f (an $m \times n$ operation), the estimation of z using h (an $n \times n$ operation), and the calculation of P_t (multiple $n \times n$ operations). The *update* step, shown in (13)–(14), consists of the calculation of the Kalman gain G_t (one $1 \times n$ and multiple $n \times n$ operations), and the estimation of the new system state $\hat{x}$, consisting of two $1 \times n$ and one

[3]Triangular numbers are defined as $\Delta_n = \sum_{x=1}^{n} x$.

$n \times n$ operation. In summary, the EKF computation time is in $\mathcal{O}(n^2 m)$.

With regard to space, EKF has the benefit of not keeping history, and therefore the EKF state size is constant over time. It consists of the two state estimation vectors $\hat{x}^*$ and $\hat{x}$ (cardinality n), the measurement estimation vector $\hat{z}$ (cardinality m), and the matrices P and G_t (cardinality $n \times n$). Overall, the space required for the EKF is in $\mathcal{O}(n^2 + m)$.

3. EVALUATION

In order to evaluate our solution, we perform a series of experiments. In the following, we describe the testbed, the experimental workload, and our evaluation method.

3.1 Experimental Testbed

As outlined in detail in Section 3.2, the experiments involve the parallel stream processing of large amounts of images using a private cloud platform. This platform is a KVM-based OpenStack instance running on eight nodes, each with four Intel Xeon E3-1230 v6 CPU cores. The total memory is 128 GB. The DSP operator instances consist of Java applications, created specifically for this experiment, and make use of ImageJ[4]. The system of operator instances constitutes our experimental DSP platform.

Operator instances are executed on VMs, where each VM is exclusively assigned one physical core. Depending on the scaling decision, we either spin up additional VMs with new operator instances, or spin down VMs. In this evaluation, we only regard one operator type, with potentially multiple operator instances. We use stateful operator instances, therefore, state must be managed during scaling (see Section 3.2). Including both VM boot time and state transfer, preliminary experiments have shown that an operator instance takes between 5 and 25 seconds to be ready. Once available, running operator instances are supplied with images on a round-robin basis from an input queue. While the input queue serves data to the operator instances in FIFO order, overall, FIFO order is not guaranteed, since instances might process data at different speeds. We perform no resynchronization after processing, as our workload processes images individually.

3.2 Workload

For our workload, we use images submitted to the DSP for processing, which is a well-known DSP use case, e.g., [47], as well as queries for certain pixel metrics gathered during the stream processing of the images.

We vary the amount of images submitted to a given operator according to patterns in three distinct workload scenarios. The images processed are taken from a real-world biomedical engineering use case [23], where images of biological cells, obtained from tissue samples and taken using fluorescence microscopy, are analyzed for certain properties. Each image has around 1.7 megapixels, is originally in TIFF format, and around 1.3 MB in size. Furthermore, each image is given Cartesian (X/Y) coordinates determining its position within the overall tissue sample. It is the task of the operators to apply a set of image filters (Gaussian blur, split into RGB channels, edge detection, and object count). In the real-world use case, these filters are used to count biological cells with a given fluorescent marker. Our workload is mainly computationally intensive and therefore CPU bound, state is stored in memory, and the only I/O requirement is the transmission of data itself.

With an average rate of 1 per 100 images, we also submit queries, where details about pixel intensities regarding a randomly chosen X/Y position in the last 1000 images are requested from the operator. The query consists of the X/Y position, and the return value is a vector containing the last 1000 pixel intensities (consisting of R, G, and B, each). This is done to calculate the distribution of pixel intensities for a specific pixel, allowing the scientists to assess the significance of this particular X/Y position for the result (i.e., to determine the significance of certain areas within the overall processed image). Therefore, the operator is required to maintain state containing this information, namely a sliding window of length 1000 (count-based, slide of 1).

We employ sharding for processing these queries, where each operator instance is assigned an equally large region within the X/Y plane of the overall cell tissue. Each instance must maintain any state required to respond to queries about its area of responsibility. This implies that during scaling operations, state must be transferred between operator instances, and queries for this particular area can only be processed after the state transfer is finished.

SLAs are employed for both the image processing requests, as well as the queries. The SLA for images is a maximum processing time of 5 seconds, while the SLA limitation for queries is one second. Any processing duration in excess of these deadlines poses an SLA violation.

The variation of workload amount is performed according to three different workload scenarios:

Pyramid: This scenario is generated synthetically using a step-wise increase of the amount of images submitted to the operator input queue, followed by a likewise reduction of this amount, and a repetition of this process for the entire duration of the simulation run. This generates a pyramid-like data rate pattern.

In this work, we use 0 and 60 as the minimum and maximum amount of images per second, respectively. The increase per step is 15 images per second, and each level is held for 130 seconds. These values are chosen to create an easily measurable pattern of usage, while using data rates and timings similar to the real data trace (*Lab*) described below.

Square: Another synthetic pattern is generated by alternating between a low amount of images, and a high amount of images per second used for input to the operator. In this work, we use 1 and 65 as the minimum and maximum amount of images, respectively, perform instant changes to the data rate, and hold both values, i.e., 1 and 65, for 370 seconds. This pattern allows us to analyze the impact of very sudden changes in workload on the filters.

Lab: This scenario stems from the used dataset and represents the amount of data processed in the lab. It is not synthesized, but represents real-world observation of the amount of recorded images. This real-life pattern contains both rapid and smooth changes and allows us to measure the behavior of our solution in a usage pattern close to a real-world scenario.

[4]`https://imagej.nih.gov/ij/`

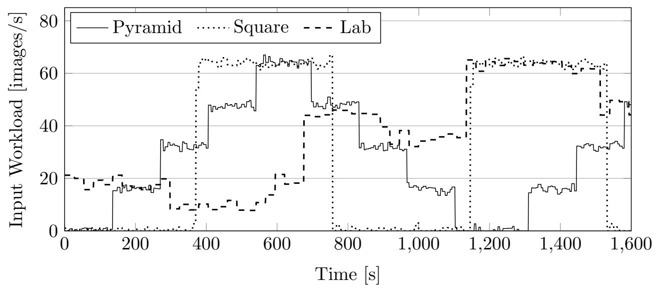

Figure 6: Excerpt of the workload scenarios used in our evaluation.

Figure 6 shows an excerpt of the input workload pattern for all three scenarios. We use the first two scenarios, i.e., the synthetic patterns *Pyramid* and *Square*, to analyze in detail the performance of our algorithm in extreme cases, such as the rapid increase or decrease of arriving data. We then additionally use the third scenario, i.e., the *Lab* scenario, to verify applicability in a real-world setup.

3.3 Evaluation Methodology

We perform the experiment in various configurations, and repeat each configuration 20 times (20 *runs*). Measured values are averaged and recorded together with their standard deviation σ. One run lasts 2700 seconds (45 minutes).

We use three filters in our experiments, consisting of the presented EKF-based approach (denoted as EKF) and two baselines for comparison (denoted as PURE and GW). The PURE filter is the identity function, i.e., no value filtering is used. The GW filter uses the Generalized Weierstrass (GW) transform. The GW transform adds the variance parameter t to the standard Weierstrass transform [48]. This parameter influences the *radius* (variation) of the smoothing effect provided by the Gaussian kernel. The kernel used in the convolution is:

$$\frac{1}{\sqrt{2\pi t}} e^{-\frac{x^2}{2t}} \tag{19}$$

where x represents the time ordinate and t denotes the variance. This notion is in line with existing literature [48].

For the implementation, we use a rectangle window function. Due to the low measurement frequency compared to the computational capacity (2 Hz), a sufficiently large window size is feasible and has no significant impact on the result, since the weights for very old measurements are marginally low. We use a window size of 1 minute.

The parameter t is dependent on the system used, and is best set to a value allowing for sufficient smoothing while minimizing the delay of edge detection. In the frequency domain of the Fourier analysis of our data, no signal distinguishable from noise is present above 0.3 Hz. We therefore set $t = 3^2 = 9$, i.e., the wavelength for 0.3 Hz in seconds, squared for variance. A discussion of Gaussian kernels and their usage within signal processing can be found in [33].

Since we perform the filtering live and cannot access future values for our calculation, only the left side of the symmetrical kernel is in effect, denoted by the range $x \leq 0$ and containing past measurements.

A major flaw of all algorithms using linear smoothing methods such as Gaussian-type transforms like the GW transform is the fact that due to the averaging performed in these algorithms, they also smooth out edges in the signal. In elastic stream processing, this means that changes in the data rate are not detected immediately, and thus scaling operations are delayed by design. Furthermore, such algorithms only provide decisions whether to scale up or down, while the EKF also indicates how many additional instances are required. This is due to the inclusion of extrinsic metrics by the EKF, such as the input data rate.

All filters used in the evaluation, i.e., PURE, GW, and EKF, are presented with measurements of the system state. In our scenario, we measure the CPU and memory utilization, however, the approach is generally applicable to any (numeric) metrics. Note that other metrics might require additional tuning of the transition function $f(\cdot)$ used in (5). The filtered version of the measurements, reflecting the approximated internal system state, is then used to find scaling decisions. We use threshold-based scaling to evaluate both filters, a technique commonly used in literature [6, 21, 37]. Scaling is performed based on the *average* value of all operator instances, taking into account every metric. Scaling up is performed if at least one metric is above the scale-up threshold. Scaling down is performed if all metrics are below the scale-down threshold. This is used to avoid bottlenecks stemming from single metrics.

We therefore define two thresholds, Θ^-, representing the scale-down threshold, and Θ^+, representing the scale-up threshold. There are various means of choosing thresholds, including automatic learning of parameters. Such learning is not within the scope of this paper, but existing approaches, e.g., [36], could be integrated into our work.

Instead, we perform a 20-fold sensitivity analysis to identify both thresholds. The sensitivity analysis is performed for all filters (PURE, EKF, GW) using the three evaluation scenarios (Pyramid, Square, Lab), thus covering the entire parameter domain of this evaluation. Only meaningful values for Θ^- and Θ^+, i.e., $\Theta^- < \Theta^+$ are used. For all threshold combinations, we measure the resulting amount of SLA violations. Table 2 shows the resulting averages of all filters and scenarios. We observe that $\Theta^+ = 80\%$ consistently provides optimal results. Furthermore, $\Theta^- = 50\%$ provides optimal results, however, with all three filters, $\Theta^- = 40\%$ is a close runner-up, and we therefore choose $\Theta^- = 45\%$ for our evaluation. Similar threshold values are used in existing literature [17, 30].

Table 2: Results of sensitivity analysis for Θ^- and Θ^+ ($n = 180$, all $\sigma < 1000$). Best results (least SLA violation) are underlined.

		Θ^+					
		50%	60%	70%	80%	90%	100%
Θ^-	40%	65.144	63.596	63.554	61.160	69.434	67.850
	50%		63.632	61.484	<u>60.056</u>	62.450	67.244
	60%			61.370	65.000	65.678	69.098
	70%				67.628	69.668	75.818
	80%					79.508	80.168
	90%						79.154

In addition, we provide experimental verification for the assumption proposed in Section 2.3, according to which the measurement noise of CPU load follows a zero-mean Gaussian distribution. By using FakeLoad [43] in preliminary experiments, we verify that the variance of the measurement is indeed distributed in a sufficiently uniform manner. We perform a 20-fold sensitivity analysis, and while the mea-

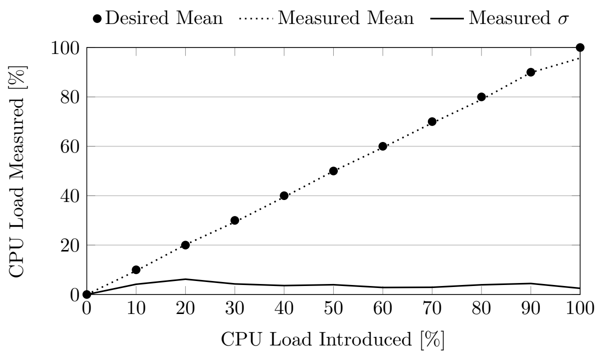

Figure 7: Sensitivity analysis of CPU load measurement noise to CPU load.

sured σ does vary, the skew we encounter is not significant to our evaluation. Figure 7 shows the mean data of these measurements. Measurements for memory utilization show almost no measurement noise.

4. EXPERIMENTS AND RESULTS

As described in Section 3, we evaluate our approach using a total of nine experiment configurations (three filters, three workload scenarios). Each configuration is executed 20 times, and each run lasts 45 minutes. In total, 135 hours of execution time are required for our scenarios.

During the experiments, we record the operators' CPU and memory utilization, the amount of running VMs, the image processing durations, and the SLA violations (see Section 3.2). For these metrics, we use the average of all 20 runs for each configuration in order to smooth out measurement noise introduced by the experimentation testbed.

We show an example of individual runs to demonstrate the overall functionality of our evaluation approach in Section 4.1. In Section 4.2, we present the aggregate results of our experiment runs. We discuss the results in Section 4.3.

4.1 Exemplary Runs

We first show results from excerpts of individual results, in order to demonstrate the functionality of our EKF approach and its effects on the scaling behavior of the system. Figure 8 shows an excerpt of three experiment runs. All three runs use the *Pyramid* workload scenario, and apply either the PURE, GW, or EKF filter. For all three runs, we show the resulting amount of running VMs on the left ordinate. Additionally, for reference, we show the input workload (in images per second) on the right ordinate.

We make the following observations during the analysis of this experiment excerpt:

- GW shows the anticipated delay in reaction (e.g., around $t = 200$, $t = 300$, $t = 410$), which is due to the fact that GW can only provide scale-up or scale-down instructions, but cannot dictate how many VMs are to be spun up or down. Since PURE reacts to changes directly, such delay is not observed for PURE.
- PURE shows excessive scaling operations, reflected in variation of VM count, which is due to the relatively high fluctuation of system state measurements. GW also suffers from such fluctuations, albeit to a lesser degree.

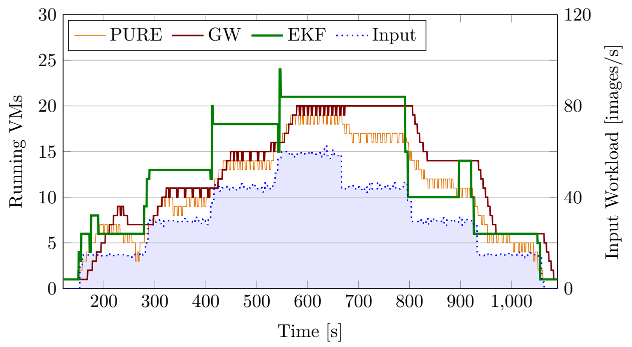

Figure 8: Three *Pyramid* experiment runs, using PURE, GW, and EKF. The resulting amount of running VMs is shown (left ordinate) together with the input workload (right ordinate).

- GW suffers from overshoot when transitioning from a no-load to a load condition (seen around $t = 220$). This is again due to the fact that GW only gives a scale-up response if Θ^+ is exceeded by any metric. As VMs are spun up, Θ^+ remains exceeded, and especially in the initial start of load ($t = 170$), this causes overshoot.
- EKF shows numerous spikes, where an excessive number of VMs is activated for a load increase. While these spikes are unfavorable, their amount is negligible compared to the frequent scaling of GW and PURE. Further fine-tuning of the state transition function can be used to further reduce these artifacts.
- The crucial advantage of EKF, its potential to not only provide scale-up and scale-down instructions, but also to dictate how many VMs are required to be spun up or down, can be observed. The EKF VM amount, once settled, remains mostly stable.
- EKF, similarly to GW, also exhibits a minor amount of fluctuation when transitioning from a no-load to a load condition. Several scale-up and also some scale-down operations take place around $t = 170$.
- However, EKF settles in a significantly more stable way, and seldom requires correction; an example for this can be seen around $t = 880$. While EKF also shows visible overshoot, this overshoot is minor, and is corrected very soon.
- It is visible that EKF often uses more VMs than GW and PURE, which is caused by its parameters a and b, i.e., the sensitivity to the input data rate. As we will later see in Section 4.2, while this effect causes more immediate cost due to an increase of VM time required, it leads to substantial reduction of SLA violations and total processing time.
- GW and EKF cause the system to react in an asymmetrical way when considering scale-up and scale-down operations. This is visible when observing the first reduction of workload around $t = 650$. In both cases, the system does not start scaling down until the next workload reduction. This behavior is due to the nature

of threshold-based scaling, where a given demand can have multiple different scales while keeping the system load within the thresholds.

- PURE does not exhibit this asymmetry, which we attribute to its frequent scaling operations. PURE is therefore more likely to traverse across its thresholds and does not "linger" in a scaled-up state.

In Figure 9, we study in detail the behavior of EKF during the run shown in Figure 8. Since during most of the experiments, the CPU load was the factor limiting scaling operations, we focus on this metric in this analysis. On the left ordinate, we show the measured CPU load, and the EKF-filtered measurement. Both of these numbers are represented averaged over all operator instances. In addition, like in Figure 8, for reference, we show the input workload on the right ordinate.

Here, we make the following observations:

- Generally, EKF provides the expected smoothing of the measured system state.

- While mostly ignoring noise, EKF reacts promptly to changes caused by actual workload increase. In some situations, EKF yields values larger than 1.0, which, in addition to the scale-up decision itself, provides an indication of how many more instances are required.

- We see that the correction around $t = 880$, mentioned in the previous findings, is due the load being close to Θ^+ after the workload decrease at $t = 800$, and finally reaching Θ^+ at $t = 880$, where the aforementioned correction takes place, scale-up is performed, and the average load drops again until $t = 920$.

Summarizing the findings from the exemplary runs, we confirm that the EKF-based filter is working as intended, and the results are qualitatively consistent with the expectations. While naturally the performance of GW could be further increased, especially the low-amplitude fluctuations could be further reduced by fine-tuning parameters, the overall aspects – that is, the delayed response, overshoot, and overall fluctuation of GW – remain.

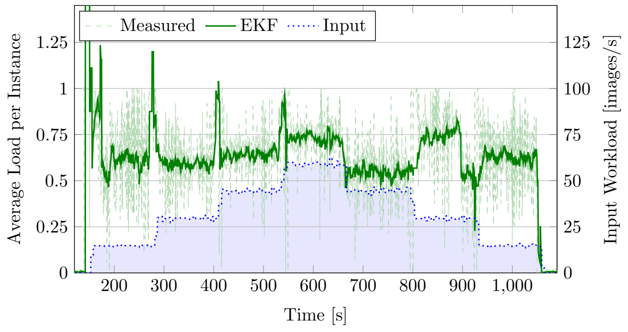

Figure 9: A detailed view of the measured and EKF-filtered CPU load in the EKF configuration run, together with the thresholds Θ^- and Θ^+ (all: left ordinate) and the input workload (right ordinate).

4.2 Aggregate Results

Table 3: Aggregate results for the *Pyramid* scenario. Lowest results are <u>underlined</u>.

Metric	PURE (σ)	GW (σ)	EKF (σ)
Total VM Time [1000 h]	<u>8.0</u> (0.2)	8.4 (0.2)	8.9 (0.3)
Scaling Events	420.6 (11.5)	304.6 (10.5)	<u>58.5</u> (8.5)
Image Processing Time [s]	2.7 (0.1)	2.1 (0.2)	<u>1.5</u> (0.7)
Query Processing Time [s]	0.6 (0.1)	0.5 (0.1)	<u>0.3</u> (0.2)
SLA Violations [1000]	53.8 (0.5)	51.7 (0.7)	<u>47.1</u> (1.3)

Table 4: Aggregate results for the *Square* scenario. Lowest results are <u>underlined</u>.

Metric	PURE (σ)	GW (σ)	EKF (σ)
Total VM Time [1000 h]	<u>8.6</u> (0.2)	9.0 (0.2)	9.2 (0.5)
Scaling Events	326.1 (3.3)	294.2 (2.4)	<u>28.5</u> (13.5)
Image Processing Time [s]	4.6 (0.1)	3.9 (0.2)	<u>1.8</u> (0.1)
Query Processing Time [s]	0.9 (0.1)	0.8 (0.1)	<u>0.3</u> (0.1)
SLA Violations [1000]	63.2 (0.4)	63.9 (0.2)	<u>47.6</u> (2.6)

Table 5: Aggregate results for the *Lab* scenario. Lowest results are <u>underlined</u>.

Metric	PURE (σ)	GW (σ)	EKF (σ)
Total VM Time [1000 h]	<u>9.9</u> (0.2)	10.6 (0.4)	12.1 (1.0)
Scaling Events	423.45 (12.6)	288.0 (18.2)	<u>54.6</u> (4.2)
Image Processing Time [s]	3.0 (0.1)	2.7 (0.1)	<u>1.6</u> (0.4)
Query Processing Time [s]	0.8 (0.1)	0.5 (0.1)	<u>0.3</u> (0.1)
SLA Violations [1000]	69.9 (0.9)	67.0 (0.9)	<u>63.8</u> (3.6)

In this section, we present the overall results of our experiments, averaged over 20 runs. We measure the total VM time consumed (measured per second, expressed in hours), the amount of scaling events, the average processing time for images and state queries, and the amount of SLA violations.

Tables 3, 4, and 5 show the results for *Pyramid*, *Square*, and *Lab*, respectively. Since we cannot assume equal variances, we use Welch's t-tests for determining statistical significance. We perform a test for each pair of values measured using PURE, GW, and EKF. We can reject H_0 (i.e., claim significance) for all value pairs except the total VM time using *Square* with GW and EKF (Table 4, first row, GW and EKF), where we cannot reject H_0. The p-value is 0.1092, i.e., rejecting H_0 would yield a 10.92% likelihood of a type I error. For all other tests, where we can reject H_0, the p-values are less than 0.004. Therefore, the following observations are based on statistically significant results.

The standard deviations of all measurements are relatively low (since all p-values are below 0.004), which indicates experimental consistency. While the order of standard deviations varies from scenario to scenario, due to the low deviations, this does not pose a risk to our evaluation.

Comparing the measured means, we observe that all scenarios show consistent results. EKF performs best for all metrics except the total VM time consumed. This means that EKF provides a reduction of scaling events, a decreased workload processing time, and less SLA violations, at the cost of VM time. We provide a break-even analysis for this trade-off in Section 4.3.

For all metrics, the order between PURE, GW, and EKF is consistent for all scenarios, i.e., GW values are always between PURE and EKF. The only exception is the number of SLA violations in the *Square* scenario.

When analyzing percental changes for a metric, we provide the change of EKF compared to GW, followed by the change of EKF compared to PURE in parentheses. For instance, a reduction of scaling events of 80.1% (86.1%) denotes that EKF provides a 80.1% reduction of events compared to GW, and a 86.1% reduction compared to PURE.

In all three scenarios, the total VM time is the highest for EKF, and the lowest for PURE. The effect is strongest in the *Lab* scenario, where the increase is 14.1% (22.2%), followed by *Pyramid*, where the increase is 6.0% (11.3%). The lowest increase, 2.2% (7.0%), is seen for *Square*. This indicates that abrupt, step-wise changes, followed by constant load, as present in *Pyramid* and even more extremely in *Square*, work in favor of EKF, while lower, constant changes in workload increase the additional VM time consumed by EKF. In total, EKF causes an increased VM time of 7.8% (13.9%).

The amount of scaling events is drastically lower for EKF in all scenarios. For *Pyramid*, EKF reduces scaling events by 80.1% (86.1%). For *Square*, the reduction is 90.8% (91.3%), and for *Lab* it is 81.0% (87.1%). Again, *Square* causes EKF to have the strongest effect, however, also the lowest change of 80.1% for *Pyramid* is drastic. In total, EKF causes a decrease of scaling events by 84.0% (87.9%).

The image and query processing times are also lowered by EKF, with a higher impact seen for image processing times. The following figures describe the processing times of all operations, i.e., image processing and state queries. For *Pyramid*, the decrease of processing time is 28.6% (44.5%). For *Square*, the decrease is 53.9% (60.9%). For *Lab*, it is 40.7% (46.7%). Therefore, in the case of processing times, the effect of rapid changes does not seem to have an impact comparable with the previously discussed metrics. In total, operation time is decreased by 43.7% (52.4%).

Finally, we inspect the impact of EKF on the amount of SLA violations. The reduction caused by EKF for *Pyramid* is 8.9% (12.5%), the reduction for *Square* is 25.5% (24.7%), and for *Lab*, it is 4.8% (8.7%). The total reduction of violations is 13.2% (15.2%). Here, again, the highest reduction is seen for *Square*, indicating an impact of rapid changes on the reduction of SLA violations by EKF.

4.3 Cost Analysis and Further Discussion

In the results presented above, we show that using EKF reduces three metrics (scaling events, processing time, and SLA violations), but increases the total VM time consumed. While the reductions provided are substantial, especially with regard to the processing times and the amount of scaling events, the increase of VM time implies a trade-off between increased VM time on the one hand, and a reduction of the remaining metrics on the other hand.

Cloud providers currently do not charge for spin-up and spin-down of machines. Therefore, the scaling events cannot be assumed to cause direct cost but nevertheless lead to higher processing times. In fact, increased processing time usually implies increased cost, either by reducing overall revenue, or by incurring SLA violation penalties.

We recall the total increase of 7.8% of VM time, and the reduction of SLA violations by 13.2%, compared to GW. We assume cost of c_v per VM hour, and cost of c_s per SLA violation, and formulate the following inequations:

$$1.078\, c_v < 1.132\, c_s \tag{20}$$

stating that the cost caused by additional VM time must be lower than the cost saved by reducing SLA violations.

From this, we can deduce:

$$c_v < 1.05\, c_s \tag{21}$$

denoting that the break-even point for EKF is when VM hours are less expensive than SLA violations, with an additional margin of 5%. At the time of writing, the Google Cloud Platform price for one core hour (Frankfurt) is \$0.0612. Assuming this price, EKF is profitable if the SLA violation penalty is more than \$0.059 per violating image or query.

Alternatively, we calculate the break-even point comparing VM hours to operation processing times. We recall the reduction of processing time by 43.7%, compared to GW. This results in an increase of $\frac{1}{1-0.437} = 1.776$, i.e., by 77.6% of processed operations. Again assuming cost of c_v per VM hour, and revenue of r_p per processed operation, we formulate the following inequation:

$$1.078\, c_v < 1.776\, r_p \tag{22}$$

$$c_v < 1.6475\, r_p \tag{23}$$

denoting the break-even point for EKF compared to GW. We see that EKF reduces cost if the cost for one VM hour is less than the revenue per operation by a factor of 1.6475. Assuming again \$0.0612 per VM hour, EKF is profitable if the revenue per processed element is higher than \$0.038.

Based on our experiments, we have shown realistic break-even points for EKF compared to GW, providing a cost-efficient operating range with regard to VM hour cost, SLA violation penalties, and operation processing revenue. The currently used workload can be extended by more complex queries and workload types, requiring additional metrics to be added to the system state described in Section 3.3.

EKF-based filtering of various data sources, such as disk utilization and service execution times [2], network traffic [26], or workload [32], has been demonstrated successfully, indicating applicability in the context of DSP. To add further metrics, any numerical data (as opposed to ordinal or nominal data) can be used, as long as it can be linearized point-wise [28]. Extending the state model includes the extension of the x_t vector, which in turn extends the remaining vectors ($\hat{x}_t$, $\hat{x}_t^*$, etc.). In addition, the state transition function f must be adapted according to the system behavior. If a query depends on data not stemming from within the system, but provided as input (i.e., additional data sources), the system input vector u_t must be expanded accordingly.

5. RELATED WORK

A number of approaches using machine learning to maintain elasticity have already been presented. For instance, Ortiz et al. [39] present PerfEnforce, using online learning to provide proactive scaling decisions for a cluster of VMs. Das et al. [8] discuss ElasTraS, allowing database systems within cloud platforms to perform multi-tenant scaling operations. An important consideration for elastic systems is the scaling overhead, which we aim to minimize. Its impact on cost has been studied by Corradi et al. [7] (in the context of cloud data centers) and by Mao et al. [35] (in the context of auto-scaling in cloud workflows). The common result of these two studies is that indeed, such overhead has significant impact and should be kept to a minimum. In other literature, the focus is put on overhead caused not by the scaling itself, but by the decision-making. Computational effort required to solve optimization problems can become quite high [10, 45], especially when using techniques such as Mixed-Integer Linear Programming (MILP) [34].

The aforementioned approaches do not directly address DSP, but rather elastic systems in general. However, DSP system exhibit very specific requirements, such as self-tuning, self-stabilization and self-healing [12]. Therefore, building on this research, but also considering the specific requirements of DSP systems, we focus on the state of the art of scaling in DSP systems. Mencagli et al. [37] use the Model-based Predictive Control (MPC) technique to create a trade-off between reconfiguration stability and amplitude. While the context (DSPs) is the same, and the aim (reduction of reconfiguration overhead) is similar to ours (reduction of the amount of scaling operations), the authors focus on the use of a distributed and cooperative approach, while we focus on the usage of multiple data sources and reduction of noise.

Floratou et al. [12] propose Dhalion, a self-healing and self-regulating extension for streaming systems, implemented on top of Twitter Heron. In this framework, metrics are used to detect symptoms of declining system health. Dhalion uses diagnosers to determine possible reasons (diagnoses) for such decline, and invokes resolvers to attempt to bring the system back to health. Self-monitoring is realized by checking whether actions taken have indeed resolved a symptom, and a learning mechanism blacklists unsuccessful resolutions. The EKF-based filtering presented in the paper at hand can be used in combination with Dhalion.

The usage of input data rate for scaling decisions has repeatedly been considered in literature [20, 31, 46], as was using threshold-based systems to deduce concrete scaling decisions [6, 21]. All of those approaches, however, suffer from the overhead problem described in Section 1. Some research has been conducted specifically to tackle this problem of overhead due to volatile input. A general recommendation is the usage of low-pass filters [7]. Another example of linear filters is found in the work by Gong et al. [14], where scaling decisions are based on a Fast Fourier Transform (FFT) and pattern recognition. In contrast to the work at hand, Gong et al. do not include extrinsic (environment) metrics, such as the input data rate. Instead, only hysteresis is used to perform smoothing of metrics. The filtering presented in our work can therefore be used as a stage prior to the scaling mechanism presented by Gong et al.

In our work, we use time series analysis, also used for scaling in other approaches. For instance, this is achieved by creating an auto-scaling algorithm using pattern matching [5], or using wavelet analysis [38]. This is complimented by either using ANN models for proactive and predictive analysis [24, 38], or reinforcement learning [11]. A combination of predictive and reactive approaches has been used in [13]. Predictive elements are used for long-term time scales, while reactive provisioning handles fine-grained, short-term peaks. However, none of these approaches use extrinsic metrics as a data source for reaching scaling decisions.

EKF-based processing is widely used for state estimation in other fields. In the context of autonomic computing, Barna et al. [2] estimate metrics such as service times, disk utilization, and CPU usage using EKF. Jain et al. [26] use EKF to process a noisy data stream of HTTP request numbers. In the context of enterprise computing systems, Kusic et al. [32] use EKF for estimating system load. Our approach builds on the technique of EKF-based time series processing, and shows its applicability in DSP. To the best of our knowledge, apart from our own preliminary work [3], no approaches use such methods in the context of DSP.

6. CONCLUSION AND FUTURE WORK

In this work, we propose EKF-based filtering for scaling in DSP systems, to reduce the amount of scaling operations and cost caused by SLA violations.

We present a model in which a time series of measurements of system state is filtered using EKF, and provide details to the application of EKF. We create a system capable of unifying intrinsic measurements (e.g., CPU and memory utilization) with extrinsic measurements (e.g., incoming data rate). The resulting filter quickly reacts to changes in the environment, while minimizing sensitivity to both process and measurement noise. We utilize these filtered values to reach scaling decisions.

We evaluate our work using a real-world dataset and workload to run experiments. While the VM time increases by up to 13.9%, we see a reduction of scaling events by up to 87.9%, a reduction of processing time by up to 52.4%, and a decrease in SLA violations by up to 15.2%.

In our evaluation, we consider CPU load and memory utilization as intrinsic metrics. While these popular metrics are useful scaling indicators, and prove effective in our evaluation, certain applications might, for instance, primarily entail high network utilization or disk usage. In such scenarios, solely considering CPU and memory utilization is not sufficient. Related work shows the applicability of EKF-based processing for other types of metrics in various fields [2, 26, 32] and therefore indicates applicability in the context of DSP. Naturally, further work is required to empirically demonstrate EKF-based filtering in DSP with additional metrics.

Another aspect worth investigating is topology-wide application. Currently, our approach considers one individual operator at a time. While it can be applied to all operators of a DSP topology, each operator performs scaling decisions on its own in a non-cooperative way. In future work, we will extend our approach to take into account system-wide behavior such as ripple effects and backpressure.

The image processing workload used for our evaluation represents a widely used scenario for stream processing [47]. While its wide usage makes it a good candidate for an initial evaluation, a wider evaluation would be required to determine specific challenges and possible approaches in other use cases. The main challenges of other scenarios are more complex queries and other workload characteristics, such as high I/O or network load. We discuss in Section 4.3 how our approach can be extended to such scenarios and queries.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive reviews of the paper. Furthermore, we would like to thank Pablo Hofbauer of the Institute of Molecular Biotechnology of the Austrian Academy of Sciences (IMBA) for kindly providing the dataset used in the evaluation, and for continuously offering insights into research techniques in biomedical engineering. This work is partially funded by COMET K1, FFG – Austrian Research Promotion Agency, within the Austrian Center for Digital Production.

8. REFERENCES

[1] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle. Elasticity in cloud computing: State of the art and research challenges. 11(2):430–447, 2018.

[2] C. Barna, M. Litoiu, and H. Ghanbari. Autonomic load-testing framework. In *ACM International Conference on Autonomic Computing*, pages 91–100. ACM, 2011.

[3] M. Borkowski, C. Hochreiner, and S. Schulte. Moderated resource elasticity for stream processing applications. In *Euro-Par 2017: Parallel Processing Workshops*, pages 5–16. Springer, 2017.

[4] R. Buyya, R. Ranjan, and R. Calheiros. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. *Algorithms and architectures for parallel processing*, pages 13–31, 2010.

[5] E. Caron, F. Desprez, and A. Muresan. Forecasting for grid and cloud computing on-demand resources based on pattern matching. In *International Conference on Cloud Computing Technologies and Science (CloudCom)*, pages 456–463. IEEE, 2010.

[6] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *International Conference on Management of Data (SIGMOD/PODS)*, pages 725–736. ACM, 2013.

[7] A. Corradi, M. Fanelli, and L. Foschini. Vm consolidation: A real case based on openstack cloud. *Future Generation Computer Systems*, 32:118–127, 2014.

[8] S. Das, D. Agrawal, and A. El Abbadi. ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems*, 38(1):5, 2013.

[9] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB*, 4(8):494–505, 2011.

[10] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *ACM SIGPLAN Notices*, volume 49, pages 127–144. ACM, 2014.

[11] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck. Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In *International Conference on Autonomic and Autonomous Systems (ICAS)*, pages 67–74, 2011.

[12] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron. *PVLDB*, 10(12):1825–1836, 2017.

[13] A. Gandhi, Y. Chen, D. Gmach, M. Arlitt, and M. Marwah. Hybrid resource provisioning for minimizing data center SLA violations and power consumption. *Sustainable Computing: Informatics and Systems*, 2(2):91–104, 2012.

[14] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *International Conference on Network and Service Management (CNSM)*, pages 9–16. IEEE, 2010.

[15] G. C. Goodwin, S. F. Graebe, and M. E. Salgado. *Control system design*. Prentice-Hall, 2001.

[16] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, 2012.

[17] R. Han, L. Guo, M. M. Ghanem, and Y. Guo. Lightweight resource scaling for cloud applications. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 644–651. IEEE/ACM, 2012.

[18] E. Hannan. *Multiple Time Series*. A Wiley publication in applied statistics. Wiley, 1970.

[19] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer. Online parameter optimization for elastic data stream processing. In *Symposium on Cloud Computing (SoCC)*, pages 276–287. ACM, 2015.

[20] C. Hochreiner, S. Schulte, S. Dustdar, and F. Lecue. Elastic stream processing for distributed environments. *Internet Computing*, 19(6):54–59, 2015.

[21] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar. Elastic stream processing for the internet of things. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 100–107. IEEE, 2016.

[22] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar. Cost-efficient enactment of stream processing topologies. *PeerJ Computer Science*, 3:e141, 2017.

[23] P. Hofbauer, J. P. Jung, T. J. McArdle, and B. M. Ogle. Simple monolayer differentiation of murine cardiomyocytes via nutrient deprivation-mediated activation of β-catenin. *Stem Cell Reviews and Reports*, 12(6):731–743, 2016.

[24] S. Islam, J. Keung, K. Lee, and A. Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1):155–162, 2012.

[25] S. Islam, K. Lee, A. Fekete, and A. Liu. How a consumer can measure elasticity for cloud platforms. In *International Conference on Performance Engineering (ICPE)*, pages 85–96. ACM/SPEC, 2012.

[26] A. Jain, E. Y. Chang, and Y.-F. Wang. Adaptive stream resource management using kalman filters. In *ACM International Conference on Management of Database (SIGMOD)*, pages 11–22. ACM, 2004.

[27] A. H. Jazwinski. *Stochastic processes and filtering theory*. Courier Corporation, 2007.

[28] S. J. Julier, J. K. Uhlmann, and H. F. Durrant-Whyte. A new approach for filtering nonlinear systems. In *American Control Conference (ACC)*, volume 3, pages 1628–1632 vol.3, June 1995.

[29] R. E. Kalman and R. S. Bucy. New results in linear filtering and prediction theory. *Journal of Basic Engineering*, 83(3):95–108, 1961.

[30] A. Kejariwal and J. Allspaw. *The Art of Capacity Planning: Scaling Web Resources in the Cloud*. O'Reilly Media, Inc., 2017.

[31] A. Khan, X. Yan, S. Tao, and N. Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *Network Operations and Management Symposium (NOMS)*, pages 1287–1294. IEEE, 2012.

[32] D. Kusic and N. Kandasamy. Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems. *Cluster Computing*, 10(4):395–408, Dec 2007.

[33] T. Lindeberg. Scale-space for discrete signals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(3):234–254, 1990.

[34] K. G. S. Madsen, Y. Zhou, and J. Cao. Integrative dynamic reconfiguration in a parallel stream processing engine. In *33rd International Conference on Data Engineering (ICDE)*, pages 227–230. IEEE, 2017.

[35] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12. IEEE, 2011.

[36] M. Maurer, I. Brandic, and R. Sakellariou. Self-adaptive and resource-efficient sla enactment for cloud computing infrastructures. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 368–375, 2012.

[37] G. Mencagli, M. Vanneschi, and E. Vespa. A cooperative predictive control approach to improve the reconfiguration stability of adaptive distributed parallel applications. *Transactions on Autonomous and Adaptive Systems*, 9(1):2, 2014.

[38] C. Napoli, G. Pappalardo, and E. Tramontana. A hybrid neuro–wavelet predictor for qos control and stability. In *Congress of the Italian Association for Artificial Intelligence*, pages 527–538. Springer, 2013.

[39] J. Ortiz, B. Lee, M. Balazinska, and J. L. Hellerstein. Perfenforce: a dynamic scaling engine for analytics with performance guarantees. *arXiv preprint arXiv:1605.09753*, 2016.

[40] W. J. Reichmann. *Use and abuse of statistics*. Penguin books, 1964.

[41] I. Selesnick. Total variation denoising (an mm algorithm). *NYU Polytechnic School of Engineering Lecture Notes*, 2012.

[42] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Symposium on Cloud Computing (SoCC)*, pages 5–18. ACM, 2011.

[43] M. Sigwart, C. Hochreiner, M. Borkowski, and S. Schulte. Fakeload: An open-source load generator. Technical Report TUV-1942-2018-01, Distributed Systems Group, Technische Universität Wien, 2018.

[44] M. Valipour, M. E. Banihabib, and S. M. R. Behbahani. Comparison of the arma, arima, and the autoregressive artificial neural network models in forecasting the monthly inflow of dez dam reservoir. *Journal of Hydrology*, 476:433–441, 2013.

[45] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini. Dejavu: accelerating resource allocation in virtualized environments. In *SIGARCH Computer Architecture News*, volume 40, pages 423–436. ACM, 2012.

[46] J. Xu, Z. Chen, J. Tang, and S. Su. T-storm: traffic-aware online scheduling in storm. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 535–544. IEEE, 2014.

[47] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[48] A. I. Zayed. *Handbook of function and generalized function transformations*. CRC press, 1996.

[49] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.