To be published in: Proceedings of the 21th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid 2021)

MEAD: Model-Based Vertical Auto-Scaling for Data Stream Processing

Gabriele Russo Russo*, Valeria Cardellini*, Giuliano Casale†, and Francesco Lo Presti*
*University of Rome Tor Vergata, Italy
Email: {russo.russo,cardellini}@ing.uniroma2.it, lopresti@info.uniroma2.it
†Imperial College London, UK
Email: g.casale@imperial.ac.uk

***Abstract*—The unpredictable variability of Data Stream Processing (DSP) application workloads calls for advanced mechanisms and policies for elastically scaling the processing capacity of DSP operators. Whilst many different approaches have been used to devise policies, most of the solutions have focused on data arrival rate and operator resource utilization as key metrics for auto-scaling. We here show that, under burstiness in the data flows, overly simple characterizations of the input stream can yet lead to very inaccurate performance estimations that affect such policies, resulting in sub-optimal resource allocation.**

We then present MEAD, a vertical auto-scaling solution that relies on online state-based representation of burstiness to drive resource allocation. We use in particular Markovian Arrival Processes (MAPs), which are composable with analytical queueing models, allowing us to efficiently predict performance at run-time under burstiness. We integrate MEAD in Apache Flink, and evaluate its benefits over simpler yet popular auto-scaling solutions, using both synthetic and real-world workloads. Differently from existing approaches, MEAD satisfies response time requirements under burstiness, while saving up to 50% CPU resources with respect to a static allocation.

***Index Terms*—data stream processing, auto-scaling, workload characterization, Markovian Arrival Processes**

I. Introduction

Nowadays, almost every aspect of our life is captured by sensors and one way or the other translated into continuous data flows, ranging from Internet-of-Things sensor measurements to high-resolution images recorded by surveillance cameras, and just recently, contact-tracing systems tied to our smartphones. The availability of such "big" data sets has fostered the development of efficient tools for data analytics. Among them, *Data Stream Processing* (DSP) systems have emerged as a *de facto* standard for low-latency processing of fast data streams. *Streams* are unbounded, ordered sequences of data units, usually indicated as *tuples*, emitted by one or more sources. DSP applications are directed acyclic graphs (DAGs), whose vertices are operators, and the edges represent streams flowing between them. *Operators* are processing elements that receive one or more streams as input, and output a new stream after applying data transformations (e.g., filtering). Executing operators in parallel across multiple CPU cores and possibly multiple distributed nodes, DSP systems can handle high-volume data streams in near real-time.

DSP application workloads are often highly variable, not allowing static resource allocation without the risk of resource under- or over-provisioning. For this reason, a lot of effort has been spent by researchers investigating solutions to *elastically* scale the computing capacity of DSP applications at run-time [1], by means of horizontal or vertical auto-scaling. *Horizontal* scaling entails adding or removing parallel instances of operators as needed, while *vertical* scaling instead adjusts resource allocation to operator instances (e.g., CPU, memory), without altering the parallelism. A major drawback of horizontal scaling is the significant overhead due to reconfiguration protocols, which are necessary to preserve integrity of data streams and internal state in the process. Vertical scaling instead is often implemented without pausing application execution, e.g., at hardware-level by scaling CPU core frequency, or at OS-level by setting the share of CPU time available to the operators. Therefore, whilst horizontal scaling is a fundamental mechanism to exploit infrastructure parallelism, vertical scaling is more suitable for seamless adaptation on short time-scales.

So far, operator auto-scaling policies have been devised leveraging, e.g., threshold-based heuristics [2], queuing theory [3], reinforcement learning [4]. However, most of the existing works rely on simple characterizations of the application workload (e.g., average input data rate). We show in this paper that overly simple models lead to very inaccurate performance predictions, hence poor scaling decisions, in presence of *burstiness*. As DSP applications are long-running and face varying conditions over time, including burstiness, accurate workload characterizations are necessary to derive good auto-scaling policies.

Few works have considered the effects of burstiness on DSP application performance so far (e.g., [5], [6]), but they either limit the analysis to application throughput or rely on aggregate burstiness indicators. The vertical auto-scaling solution we present, called MEAD (*Model-based vErtical Auto-scaling for DSP*), addresses the limitations of current methods by exploiting Markovian Arrival Processes (MAPs) for accurate online workload characterization. MAPs can model non-exponential inter-arrival times, and capture several statistical properties of the application input stream, including, e.g., correlation between consecutive tuples. MEAD leverages MAP-based queueing models to drive auto-scaling and satisfy performance requirements. In particular, since DSP applications are often latency-sensitive, we focus on maximum

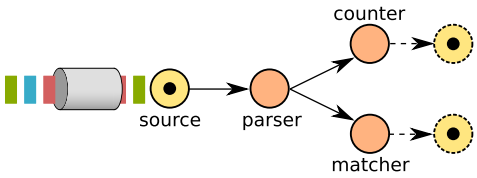


Fig. 1: Reference DSP application.

response time requirements, expressed in terms of mean value or percentiles. The choice of focusing on *vertical* scaling is motivated by various considerations. First, when horizontal scaling is used, input streams are partitioned among parallel instances, potentially mitigating the burstiness effects due to correlation between consecutive arrivals. Therefore, vertical scaling is the setting where accurate workload characterization is most needed. Furthermore, for DSP applications deployed in resource-constrained environments (e.g., Fog/Edge), acquiring additional CPU cores may be not trivial, whereas resource savings due to vertical scaling are attractive. As a final consideration, we remark that horizontal operator scaling has been investigated much more extensively than vertical scaling so far [1], despite its potential benefits in terms of reduced overhead, and we aim to contribute in filling this gap.

We show the need for accurate workload characterizations through a motivating example in Sec. II. Then, we present our contributions, which can be summarized as follows:

- We design a vertical auto-scaling framework for DSP, which - to the best of our knowledge - is the first solution where DSP application performance is evaluated through MAP-based models (Sec. III).
- We propose an online workload characterization solution that fits MAPs from monitoring data, which involves new challenges compared to traditional offline trace fitting (Sec. IV); the resulting MAP models are used to derive a model-based scaling policy (Sec. V).
- We integrate MEAD in Apache Flink (Sec. VI), and assess its benefits through experiments with synthetic and real world workloads (Sec. VII).

We review related work in Sec. VIII and conclude in Sec. IX.

II. Motivating Example

In this section, we show that operator auto-scaling strategies that rely on simple yet popular performance models (e.g., M/G/1 queues) can be far from optimal when dealing with bursty workloads. We consider a DSP application that analyzes logs, computing statistics and matching lines against patterns (e.g., to detect malicious requests). The application is composed of a data source, which reads data from a message queue, and three processing operators (see Fig. 1). The first operator (*parser*) parses the input lines, discarding any malformed tuple, and forwards a copy of each parsed entry to two stateful operators, named *counter* and *matcher*. The former counts the events logged by different system components; the latter tries to match log entries against a set of user-defined patterns, keeping a count of matched events.

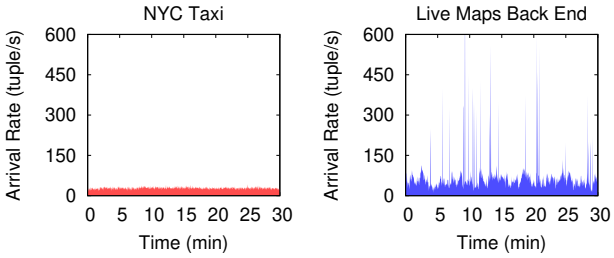


Fig. 2: Data arrival rate in the traces.

We run this application on top of *vanilla* Apache Flink, using two real traces for the workload. The first trace is extracted from a data set about taxi trips in New York City[1], which is representative of typical edge-generated events processed by DSP applications (see, e.g., [7]). The second trace, extracted from the *Microsoft Production Server - Live Maps Back End* traces[2], contains requests logs of a distributed storage system. We consider 30-minute segments of each trace. In order to have similar arrival rates for both the traces, we replay the NYC Taxi trace with a 10x speedup factor, without altering other statistical properties of the trace. Nonetheless, except for the similar average rate, the two traces are significantly different. As shown in Fig. 2, while the Taxi trace shows low variability in the arrival rate, the Live Maps trace is characterized by noticeable bursts.

We ask ourselves the fundamental question as of whether queueing models can accurately estimate operator performance under these workloads. As baseline approaches, we consider the M/G/1 queueing model and Kingman's approximation for GI/G/1 queues [8], which have been frequently used in the context of DSP systems (e.g., [3], [9]). As an alternative, we consider a queuing model where the arrival process is modeled as a Markovian Arrival Process [10]. To instantiate these models, we approximate the operator service process with phase type (PH), and specifically Erlang-2, distributions, and use linear regression to estimate the mean service time [11].

Hereafter, we focus on the performance of the *matcher* operator, as preliminary experiments showed that its resource demand is much higher than the other operators in the application. By varying the number of patterns the operator searches for, we evaluate the mean operator response time for different values of CPU utilization. As shown in Fig. 3, under the Taxi workload, the estimates provided by the different models are almost indistinguishable, matching well the measured operator response time. In the Live Maps scenario, we note instead that the M/PH/1 model, which assumes independent, exponentially distributed inter-arrival times (IATs), leads to very inaccurate performance predictions. Indeed, its mean response time estimates are almost two orders of magnitude lower than the measured one. When using Kingman's formula, the prediction error is only slightly reduced, and the results are still far from

[1]Data set provided by the New York City Taxi & Limousine Commission at https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[2]http://iotta.snia.org/traces/158

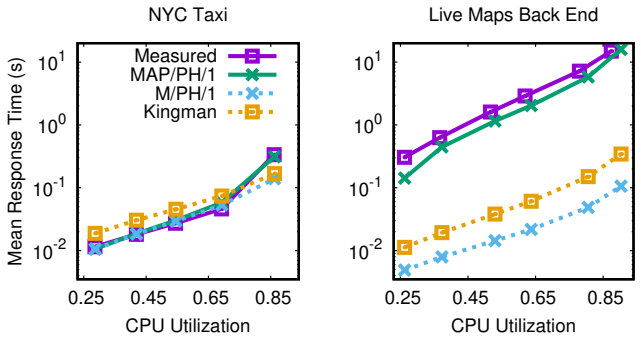


Fig. 3: Comparison of queueing models for operator response time evaluation for both the considered workloads.

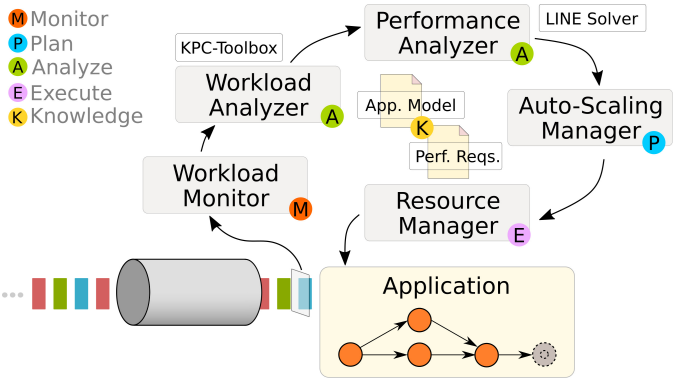


Fig. 4: Architecture of MEAD.

satisfactory. Conversely, the MAP/PH/1 model, which captures IATs correlation, is able to match the mean operator response time, both at low and high utilization values.

These results clearly suggest that (i) burstiness has an evident impact on application response time, which cannot be captured looking at the arrival rate only or the average utilization; (ii) simple yet popular models provide good accuracy under uncorrelated workloads, but in order to cater for bursts, more general models are necessary for auto-scaling control. As DSP systems likely face burstiness at run-time (see, e.g., [5], [12, Chapter 5]), these experiments motivate our work.

III. Overview of MEAD

In this section, we introduce MEAD, a model-based framework for operator vertical auto-scaling. In the scenario we consider, each operator during execution is associated with a *resource configuration* x, selected from a finite set $\mathcal{X} = \{1, 2, \ldots, N_{res}\}$, where 1 and N_{res} represent, respectively, the minimum- and maximum-capacity configurations. In particular, the resource we consider for vertical scaling in this work is CPU. Each configuration x has a cost $c(x)$, which may capture different aspects, e.g., energy consumption, or monetary cost due to resource acquisition in a pay-as-you-go scenario. At run-time, we aim to pick appropriate resource configurations for the operators, so as to minimize the overall resource cost while satisfying the application performance requirements.

MEAD associates DSP applications with new components to support run-time adaptation. In particular, MEAD is organized according to the well-known MAPE-K (*Monitor, Analyze, Plan and Execute with Knowledge*) pattern for self-adaptive systems, and comprises five main components: Workload Monitor, Workload Analyzer, Performance Analyzer, Auto-Scaling Manager, and Resource Manager (see Fig. 4). The *Workload Monitor* collects information about the application workload, monitoring the data stream that the application ingests from external sources. The *Workload Analyzer* uses this information to build a model of the current workload, which is used by the *Performance Analyzer* to evaluate application performance. Based on the resulting performance estimates, the *Auto-Scaling Manager* plans scaling actions for each operator, which are executed by the *Resource Manager*.

The analysis and planning components rely on a shared knowledge, which comprises a description of the application topology, including the estimated service demand of each operator, and a specification of the performance requirements (e.g., mean application response time). In this work, we have assumed that this information is collected offline and provided to the system when the application is deployed. We plan to extend the framework to automatically gather the required information online in the future. The functionality of each component and their interaction will be described in greater detail in the next sections.

IV. Workload Characterization using MAPs

In this section, we look at the Workload Analyzer component, which characterizes the application workload at run-time using MAPs. We briefly introduce MAPs, before illustrating our solution for online MAP fitting.

A. Overview of Markovian Arrival Processes

Consider a stream of tuples, with a timestamp T_k associated with the arrival time of the k-th tuple. In this paper, we characterize such a stream by modeling the tuple IATs time series $X_k = T_k - T_{k-1}$, $k = 1, 2, \ldots$ as a MAP [10]. MAPs can be regarded as generalizations of continuous-time Markov chains (CTMC) where transitions are either hidden or observable. *Hidden* transitions denote a change in the state of the process but do not result in arrivals, whereas *observable* transitions correspond to actual arrivals (and possibly to a state change). Formally, a MAP with N states, denoted as a MAP(N), is defined by a pair of $N \times N$ matrices, $\mathbf{D_0}$ and $\mathbf{D_1}$, which, respectively, specify the hidden and observable transitions, such that $\mathbf{Q} = \mathbf{D_0} + \mathbf{D_1}$ is the infinitesimal generator of the underlying CTMC.

A sequence of IATs X_k is generated from a MAP considering the time elapsed between successive activations of any two *observable* transitions. Because each sample X_k is generated according to the target state of the observable transition that generated X_{k-1}, statistical correlation may exist between consecutive MAP samples. This marks a fundamental difference with respect to, e.g., phase-type renewal processes, where arrivals are independent. Moreover, MAPs provide a key

advantage with respect to non-Markovian workload models (e.g., ARIMA) in that analytical tools are available for solving MAP/MAP/1 queuing models efficiently [13].

B. Fitting MAPs from data

Fitting a suitable MAP to the observed workload means determining a pair $(\mathbf{D_0}, \mathbf{D_1})$ such that model generated samples have the same statistical properties of the original data stream. Specifically, this entails matching the moments and the autocorrelation coefficients of the trace to those of a MAP(N), which are characterized by the following expressions:

$$E[X^n] = n!\pi_{\mathbf{e}}(-\mathbf{D_0})^{-n}\mathbf{e},\ n = 1, 2, \ldots \tag{1}$$

$$\rho_n = \frac{\pi_{\mathbf{e}}(-\mathbf{D_0})^{-1}\mathbf{P}^n(-\mathbf{D_0})^{-1}\mathbf{e} - E[X^2]}{E[X^2] - E[X]^2},\ n = 0, 1, \ldots \tag{2}$$

where $\mathbf{P} = (-\mathbf{D_0})^{-1}\mathbf{D_1}$ is the transition probability matrix of the embedded discrete-time Markov chain associated with $\mathbf{Q}$, $\pi_{\mathbf{e}}$ the equilibrium probabilities vector for $\mathbf{P}$, and $\mathbf{e}$ is the vector $(1, 1, \ldots, 1)$ of length N. Given the inherent complexity of the resulting problem, fitting algorithms focus on matching a limited number of moments and autocorrelation coefficients so as to approximate as accurately as possible the measured arrival process. To date, it is still an open problem how to select the best set of descriptors to be matched [14].

MEAD relies on the divide-and-conquer approach presented in [14], which, instead of directly fitting a large MAP, searches for J simpler MAP(2) models; leveraging the *Kronecker Product Composition* (KPC), the resulting J MAPs are composed into the desired MAP(2^J). KPC enjoys several nice properties that ease MAP fitting, most notably that: (i) moments and autocorrelation coefficients of a MAP obtained using KPC can be computed in closed form from the parameters of the constituent MAPs; (ii) the matrices $\mathbf{D_0}$ and $\mathbf{D_1}$ of a MAP(2) can be directly computed from their statistics.

MEAD leverages the KPC-based algorithm provided by KPC-TOOLBOX[3], which comprises three main steps: order selection, nonlinear fitting, and KPC composition. In the first step, KPC-TOOLBOX picks the number J of MAP(2) models to be fitted, hence the number of states of the final MAP, balancing model complexity and accuracy. Then, the J MAP(2) are determined through nonlinear least-squares fitting, so as to match the first three moments and a set of autocorrelation coefficients of the trace. In the last step, the final MAP is computed applying KPC.

C. Online MAP fitting with KPC-Toolbox

MAPs have been often applied so far for offline trace fitting. In MEAD, we need to tackle the harder task of fitting MAPs online from monitoring data. Our approach for online workload characterization relies on the Workload Monitor and the Workload Analyzer components. The former monitors the application input stream, recording the IATs of consecutive tuples and storing them in a monitoring buffer. Since DSP applications are typically long-running, to limit memory usage, the monitoring buffer has a fixed capacity of N_e events. When new event timestamps must be added to the monitoring buffer, the least recent ones are discarded, on a first-in first-out basis.

The Workload Analyzer is in charge of MAP fitting. When activated, either periodically or on request, this component first retrieves a snapshot of the monitored IATs from the Workload Monitor. Online MAP fitting involves a few challenges that are not encountered in traditional offline trace fitting. First of all, when searching for a workload model offline, the highest possible accuracy is usually desired. This is what KPC-TOOLBOX does by returning the MAP that minimizes a suitable loss function. Conversely, since we aim to use the fitted MAPs online to predict performance and drive auto-scaling, we would rather settle for a slightly less accurate trace matching, as long as we err on the safe side, avoiding resource under-provisioning and performance degradation.

To address this issue, we extend KPC-TOOLBOX so that, instead of returning a single MAP, it returns the best n MAPs identified during the least-squares fitting. The Workload Analyzer hence outputs multiple workload models, which can all be used for performance evaluation. Among the resulting estimates, adopting a conservative approach, we pick the worst-case prediction.

Another challenge arises from the fixed-size monitoring buffer that stores IATs. While in general a long trace is helpful to achieve higher fitting accuracy, online we also need to keep the trace short enough to quickly capture changes in the workload. To overcome this issue, the Workload Analyzer exploits multiple monitoring *windows*, i.e., traces of observed IATs. Given a snapshot of the monitoring buffer containing N_e observed arrivals, the *base fitting window* comprises the whole sequence of observed IATs. From the base window, multiple sub-windows are constructed by considering shorter observation intervals, e.g., picking the most recent $\frac{N_e}{2}$ events only, or the most recent $\frac{N_e}{4}$ events. By executing the fitting algorithm against each window, multiple workload models are obtained. By doing so, we enjoy the benefits of both small monitoring windows, which allow us to quickly detect recent workload changes, and larger windows, which provide more information for fitting. Combining the two mechanisms described above, at the end of the workload analysis phase multiple sets of MAPs, each associated with a specific fitting window, are made available to the Performance Analyzer component.

V. MODEL-BASED AUTO-SCALING

We now explain how MEAD exploits online workload characterization to drive operator auto-scaling.

A. Performance Analysis

The Performance Analyzer aims at identifying resource configurations that allow the application to satisfy its performance requirements. To this end, it leverages the workload models produced by the Workload Analayzer, along with a specification of the performance requirements and a model

[3] http://www.cs.wm.edu/MAPQN/kpctoolbox.html

of the application. The application model is a DAG $G_{dsp} = (V_{dsp}, E_{dsp})$, where V_{dsp} is a set of vertices (i.e., operators), and E_{dsp} is a set of edges (i.e., data streams flowing between operators). Each vertex is also associated with an estimate of the operator service demand.

When the Performance Analyzer is activated, G_{dsp} is used to define an open queueing network that models the DSP application. To this end, each operator in V_{dsp} is mapped to a single-server queue in the network, where the service process is specified according to (i) the estimated parameters provided in the shared knowledge, and (ii) the CPU configuration assigned to the operator. The edges in the application graph G_{dsp} are used to define the routing of jobs (i.e., tuples) between queues in the network. External arrivals to the system are assumed to be generated according to the MAP computed by the Workload Analyzer. The resulting queueing model is used to evaluate the application performance under different configurations. To do so, the Performance Analyzer exploits the *matrix-analytic method* (MAM) [13], an effective technique for the resolution of MAP-based queueing models.

As regards the performance requirements, we assume them to be expressed in terms of maximum response time, since DSP applications are often latency-sensitive. Specifically, for every path $\boldsymbol{\pi}$ in the application DAG, which usually corresponds to a specific query solved by the application, we define the response time as the time it takes for a tuple entering the system to be processed by all the operators along the path. Hence, given a path $\boldsymbol{\pi}$, a maximum value $R_{\boldsymbol{\pi}}^{max}$ can be specified for the response time along the path. MEAD accepts requirements specified either in terms of *mean* response time, or in terms of its *percentiles*.

B. Auto-Scaling Algorithm

The auto-scaling problem considered in MEAD consists in satisfying application response time requirements while minimizing the cost due to resource allocation. It is formulated as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{i \in V_{dsp}} c(x_i) \\ \text{subject to} \quad & R_{\boldsymbol{\pi}}(\boldsymbol{x}) \leq R_{\boldsymbol{\pi}}^{max}, \pi \in \boldsymbol{\Pi}_{dsp} \\ & x_i \in \mathcal{X}, \qquad i \in V_{dsp} \end{aligned}$$

where x_i is the resource configuration assigned to operator $i \in V_{dsp}$; $c(x)$ the resource cost in configuration x; and $R_{\boldsymbol{\pi}}(\boldsymbol{x})$ the response time along path $\boldsymbol{\pi}$ with the configuration vector $\boldsymbol{x}$; $\boldsymbol{\Pi}_{dsp}$ the set of the source-to-sink paths in G_{dsp}.

To solve the problem defined above, we use the heuristic algorithm shown in Fig. 5, which works as follows. At the beginning, each operator is assigned the minimum cost configuration (line 1). Then, the algorithm iteratively updates the resource allocation until the response time requirements are satisfied. Specifically, for each application path $\boldsymbol{\pi}$, we first evaluate the performance with the current configuration $\boldsymbol{x}$ (lines 2-3). Then, as long as the response time requirement is violated and there is at least one operator to which more resources can be allocated (line 4), we scale up the operator

Input: MAP M, queueing model Q, max. RT $\boldsymbol{R^{max}}$
Output: configuration $\boldsymbol{x}$, estimated RT $\boldsymbol{R}$

1: $x_i \leftarrow 1 \ \ \forall i \in V_{dsp}$
2: **for all** paths $\boldsymbol{\pi}$ in G_{dsp} **do**
3: $\quad R_{\boldsymbol{\pi}} \leftarrow$ solve(Q,$\boldsymbol{x}$,$\boldsymbol{\pi}$)
4: $\quad$ **while** $R_{\boldsymbol{\pi}} > R_{\boldsymbol{\pi}}^{max}$ and $\exists i \in \boldsymbol{\pi} : x_i < N_{res}$ **do**
5: $\quad\quad i \leftarrow$ findOperatorToScale($\boldsymbol{\pi}, \boldsymbol{x}, R_{\boldsymbol{\pi}}$)
6: $\quad\quad x_i \leftarrow x_i + 1$ ▷ scale-up i
7: $\quad\quad R'_{\boldsymbol{\pi}} \leftarrow$ solve(Q,$\boldsymbol{x}$,$\boldsymbol{\pi}$)
8: $\quad\quad$ **if** $R'_{\boldsymbol{\pi}} > R_{\boldsymbol{\pi}}^{max}$ and $R_{\boldsymbol{\pi}} - R'_{\boldsymbol{\pi}} < \delta R_{\boldsymbol{\pi}}^{max}$ **then**
9: $\quad\quad\quad x_i \leftarrow x_i - 1$ ▷ keep previous conf.
10: $\quad\quad\quad$ **break**
11: $\quad\quad$ **end if**
12: $\quad\quad R_{\boldsymbol{\pi}} \leftarrow R'_{\boldsymbol{\pi}}$ ▷ update estimated RT
13: $\quad$ **end while**
14: **end for**

Fig. 5: Auto-Scaling Algorithm

i with the highest response time (lines 5-6). After scaling, we solve the queueing model with the new resource configuration (line 7). If the obtained response time improvement is not enough to satisfy the requirement and smaller than a fraction δ of the target value, we revert the scaling action and skip to the next path to be optimized (lines 8-10). This avoids unnecessarily scaling up lightly loaded operators when the performance requirement cannot be satisfied.

The algorithm returns the chosen resource configuration $\boldsymbol{x}$ along with the predicted response time. Based on this information, the Auto-Scaling Manager plans the auto-scaling actions to be performed. It translates the identified configuration $\boldsymbol{x}$ to concrete CPU configurations to be set. In this planning phase, additional policies may be enforced. For example, the Auto-Scaling Manager can be configured to avoid scale-down operations that reduce the resource allocation by more than a single level, to achieve a more conservative behavior.

The actions planned by the Auto-Scaling Manager are eventually communicated to the Resource Manager, which executes them. The actual functionality of the Resource Manager depends on the specific vertical scaling mechanism in use. For example, the Resource Manager may re-configure the CPU frequency, or the amount of CPU shares assigned to a certain process, thread, or software container. We will describe our vertical scaling implementation in the next section.

VI. Integration in Apache Flink

We integrate MEAD in Flink [15], an open-source distributed streaming engine. Flink provides an expressive API to define DSP applications, along with higher-level libraries for common analytics use cases. For execution, Flink leverages a distributed architecture, which comprises two main components, namely JobManagers and TaskManagers. *JobManagers* coordinate the distributed execution and are responsible for state checkpointing and recovery in case of failure. *TaskManagers* execute the application *tasks* (i.e., the operators) and take care of data transfers between them.

To support vertical auto-scaling, we integrate MEAD components in Flink. The Workload Monitor is implemented as a new standalone component, which monitors the streams entering the data ingestion queue. In particular, the message queue we use is RabbitMQ[4], which is also used to let MEAD components communicate with each other. In particular, we rely on the *fanout* exchange of RabbitMQ, which broadcasts incoming messages to multiple associated queues, to let the incoming stream be consumed by the application independently from the Workload Monitor. New components are also introduced for the Workload Analyzer and the Performance Analyzer, which are implemented in MATLAB and leverage, respectively, KPC-TOOLBOX and the LINE solver[5].

Within the JobManager, we associate an instance of the Auto-Scaling Manager with each running application. Besides planning, the Auto-Scaling Manager is also responsible for the periodic activation of the analysis components, every T_A seconds.

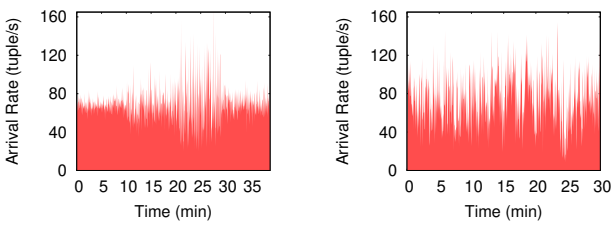
The Resource Manager is responsible for the execution of the scaling actions. For this purpose, we rely on the *cgroup*[6] subsystem of Linux. In particular, we use the `cpu` controller to dynamically adjust the share of CPU time assigned to each operator thread. When applications are deployed, a new cgroup is associated with each operator, receiving the default CPU share. At run-time, according to scaling decisions, cgroup parameters are updated as needed, with no overhead from the operator point of view.

VII. EVALUATION

For evaluation, we compare MEAD equipped with the MAP-based performance model to two alternative approaches, where (i) each operator is modeled as a M/G/1 queue, and (ii) Kingman's formula for GI/G/1 queues is used to estimate the operator response time. We also include in the comparison a popular threshold-based auto-scaling policy. According to this policy, when the monitored operator CPU utilization exceeds a scale-up threshold U_H, the CPU share assigned to the operator is increased by one level; analogously, when the utilization is lower than a scale-down threshold U_L, the assigned CPU share is decreased by one level.

Experimental setup. We use a cluster composed of three nodes: a server equipped with a 8-core Intel Xeon E5504 processor is used for hosting Flink; an identical server runs RabbitMQ and the data producer; a third machine equipped with a 4-core Intel 4710-HQ processor hosts the workload and performance analysis components.

Application and workloads. For the experiments, we consider the log analysis application described in Sec. II. As regards the workload, we consider a synthetic trace and 2 real traces. The synthetic trace (SYNT, for short) is generated from MAPs with different variability and allows us to experiment with abrupt workload changes (Fig. 6a). For the real traces, in addition to the already mentioned Live Maps Back End trace (LM), we consider the *BC-pAug89* (BC) trace from the Internet Archive[7], which is a network traffic trace, widely used in the context of workload characterization (Fig. 6b). As shown in Fig. 7, arrivals in the LM trace are the most correlated; BC trace also shows significant IATs autocorrelation, with SYNT being the least correlated one. We do not include the uncorrelated NYC Taxi trace from Sec. II in the evaluation, because it does not represent a challenge for auto-scaling (see Fig. 3), and thus the results would not contribute any novel analysis to the state-of-the-art, e.g., [4].



(a) SYNT Trace (b) BC Trace

Fig. 6: Trace arrival rate (LM trace shown in Fig. 2).

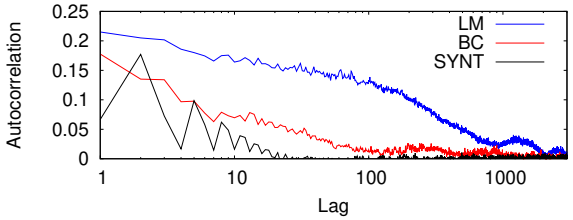


Fig. 7: IATs autocorrelation for the used traces.

Parameters. We consider two types of performance requirements. We first impose a constraint on the mean application response time. Specifically, having observed in preliminary experiments that the different characteristics of the traces lead to sensibly different response times, we pick a set of values for R^{max} for each trace as follows: 0.1, 0.2 and 0.3s for the SYNT trace; 0.3, 0.5 and 0.75s for the BC trace; 1.5, 3, and 5s for the LM trace. In the second evaluation scenario, we express the performance requirement in terms of 95^{th} response time percentile, and, specifically, we set R^{max} as follows: 0.5 and 0.75s for the SYNT trace; 0.5 and 1s for the BC trace; 7 and 10s for the LM trace. We set the same target response time for both the source-to-sink paths in the application graph.

For CPU scaling, we define five cgroup configurations $\mathcal{X} = \{1, \ldots, 5\}$ where the CPU share allocated to operators ranges between 40% for $x = 1$ and 100% (i.e., not limited) for $x = 5$. We let the resource configuration cost $c(x)$ simply correspond to the CPU share associated with the configuration x. When the application is started, every operator gets the maximum available CPU share, i.e., $\bar{x} = 5$.

The Workload Monitor uses a window with capacity $N_{E,1} = 50,000$ events, and for fitting we use an additional window with size $N_{E,2} = 25,000$. For comparison, we will

[4]https://www.rabbitmq.com/

[5]http://line-solver.sourceforge.net/

[6]https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[7]ita.ee.lbl.gov

also consider the case where the second window is not used. For each fitting window, the best 5 fitted MAPs are used for performance analysis. Workload and performance analysis are performed periodically, every $T_A = 60$ seconds. The choice of these parameters is done based on preliminary tests, where we verified that these values allow MEAD to satisfy given requirements with acceptable overhead (i.e., benefits of fitting more MAPs or with higher frequency are not evident).

For the threshold-based policy, we set $U_H = 0.7$ and $U_L = 0.5$. As this policy is purely reactive, it would be penalized by triggering scaling actions with the same frequency of MEAD. Therefore, we increase the auto-scaling frequency, letting $T_A = 10$ seconds in this case.

A. Mean response time requirement

We first consider the case where performance requirements are specified in terms of mean response time. Table I reports the mean measured application response time and allocated CPU shares for these experiments. Having observed that the *matcher* operator is much more loaded than the *counter*, hereafter we only report the response time value associated with the source-to-sink path containing the bottleneck operator. Nonetheless, we verified that the response time along the other path is always kept within the imposed bound.

We first look at the synthetic workload. Setting the maximum response time to 0.1s, we observe that both the M/G/1- and GI/G/1-based models lead to resource under-provisioning and do not satisfy the performance requirement. Looking at Fig. 8a, it is clear how the GI/G/1 model correctly predicts a low response time in the first part of the experiment and, consequently, avoids scale-up actions. However, as soon as the workload gets more bursty, estimates become inaccurate and response time exceeds the target value. Conversely, using the MAP-based model, MEAD is able to achieve a 0.056s mean response time, while allocating less than 55% of the CPU to the operators on average (Fig. 8d). A similar behavior is observed for larger values of R^{max}, where MEAD with the M/G/1 model never manages to satisfy the performance requirement. The GI/G/1 model leads to acceptable performance only with the loosest requirement, where the MAP-based policy still achieves a lower response time using slightly less resources.

The inaccuracy of auto-scaling policies based on M/G/1 and GI/G/1 models is even more evident when looking at the experiments with the BC workload, which shows more burstiness than the synthetic trace. In this scenario, those models lead to performance violation for all the considered values of R^{max} (see, e.g., Fig. 8b). The MAP-based model instead is able to meet the expected response time requirements, while saving up to 50% of the CPU capacity on average (see, e.g., Fig. 8e).

Looking at the experiments in which we use the LM trace, we can observe again the same behavior, with the gap between baseline models and MEAD getting even larger. The burstiness of this trace is not captured by the M/G/1 and GI/G/1 models, leading to performance violation for all the considered values of R^{max} (see, e.g., Fig. 8c). Conversely, the MAP-based policy satisfies the response time requirement while saving a significant amount of CPU capacity. The CPU resource allocation indeed ranges from 54% for $R^{max} = 5$s to 61% for $R^{max} = 1.5$s (Fig. 8f).

We also compare MEAD to a threshold-based auto-scaling policy (see results in Fig. 9). Reacting to increases in the CPU utilization, this policy is able to respond to bursts by scaling up the allocated CPU share. In particular, for the SYNT workload, the mean response time is equal to 0.122s, using on average 49% of the CPU capacity. For the BC trace, we observe a 0.310s response time, with 49% CPU allocation. For the LM trace, the response time is 2.4s, using 48% of the available CPU capacity. We can thus note that this widely adopted heuristic policy leads to better auto-scaling behavior than overly simple queueing models in face of burstiness. Nevertheless, compared to MEAD approach, this policy does not allow users to express a performance requirement, and instead relies on utilization-based thresholds. Therefore, in order to trade-off performance with resource usage we can only adjust the scaling thresholds following a trial-and-error approach. MEAD instead is able to adapt the auto-scaling policy based on the response time requirement.

B. Response Time Percentile Requirement

In this set of experiments, we consider performance requirements specified in terms of 95^{th} response time percentile. We only compare the auto-scaling policies relying on M/G/1 and MAP/G/1 queue models, because (i) Kingman's formula cannot be used to obtain information about the response time distribution of GI/G/1 queues, and (ii) the threshold-based policy is not aware of the response time requirement.

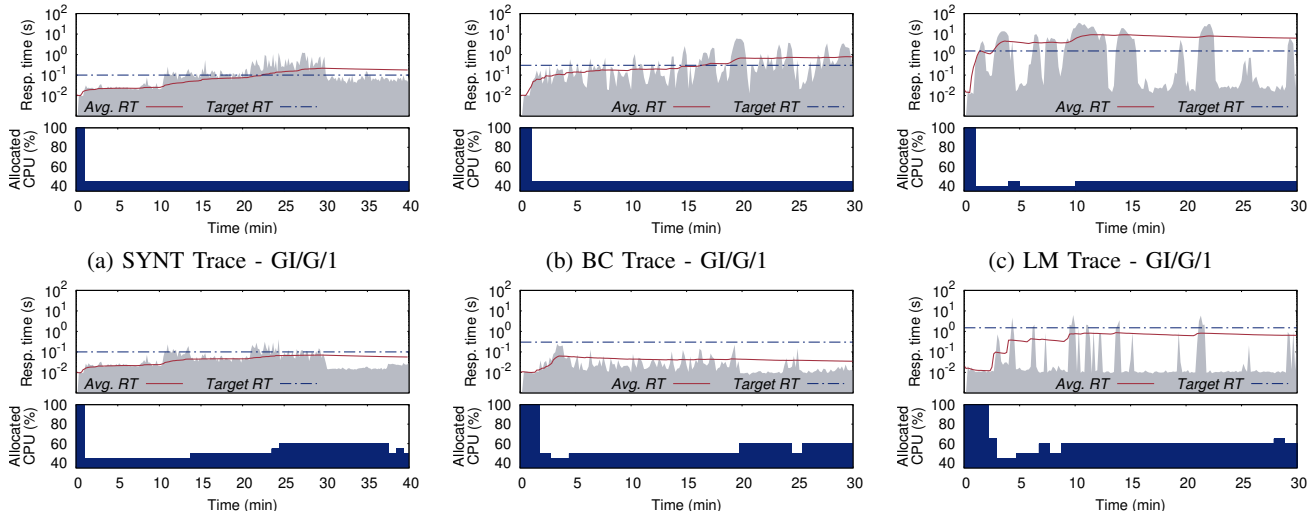
The results of these experiments are summarized in Tab. II. We can note that, for all the considered traces, only the MAP-based model guarantees satisfaction of the performance requirements. The inaccuracy of the M/G/1 model is particularly evident on the real traces. For instance, as shown in Fig. 10a, using the BC trace with $R^{max} = 0.5$s, MEAD with the MAP-based policy keeps the 95^{th} percentile below 0.3s, whereas with the M/G/1-based policy it is higher than 3.8s. Similarly, as shown in Fig. 10b, using the LM trace with $R^{max} = 7$s, the measured response time value is 4.8s for the MAP-based model and 30s with the M/G/1. These results confirm the significant inaccuracy of simpler yet popular workload models in presence of burstiness.

C. Impact of the Monitoring Window Size

To assess the benefits of fitting MAPs using multiple windows (see Sec. IV), we now consider the case where a single monitoring window is used. We consider settings where the size of the window N_E is set to 25,000, 50,000, and 100,000 events. Clearly, in the first part of the experiments, until the monitoring window gets filled up, there is no difference in changing the window size. When windows start being full, the smaller the window, the sooner the temporal memory of past events will decay.

TABLE I: Results with the mean response time requirement.

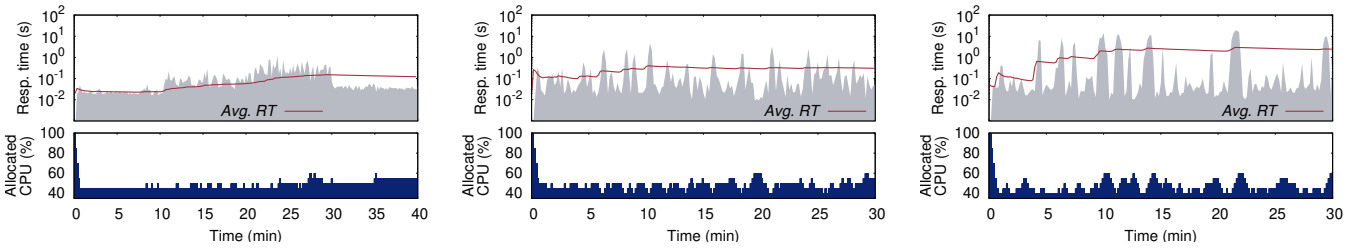
	SYNT Trace			*BC Trace*			*LM Trace*		
Model	$\mathbf{R^{max}}$	**RT** (s)	**CPU** (%)	$\mathbf{R^{max}}$	**RT** (s)	**CPU** (%)	$\mathbf{R^{max}}$	**RT** (s)	**CPU** (%)
M/G/1	0.1	0.254	46.4	0.3	0.906	46.9	1.5	7.499	45.4
GI/G/1	0.1	0.177	46.4	0.3	0.820	47.0	1.5	6.077	45.5
MAP/G/1	0.1	0.056	53.2	0.3	0.035	55.8	1.5	0.751	61.2
M/G/1	0.2	0.393	46.4	0.5	2.101	46.8	3	7.931	45.0
GI/G/1	0.2	0.208	46.4	0.5	1.194	46.9	3	6.344	45.4
MAP/G/1	0.2	0.127	50.2	0.5	0.158	53.1	3	1.580	56.9
M/G/1	0.3	0.609	46.5	0.75	2.800	46.9	5	9.382	45.4
GI/G/1	0.3	0.263	48.3	0.75	2.210	46.8	5	6.311	45.4
MAP/G/1	0.3	0.142	48.1	0.75	0.368	50.3	5	2.698	53.8



(a) SYNT Trace - GI/G/1 (b) BC Trace - GI/G/1 (c) LM Trace - GI/G/1

(d) SYNT Trace - MAP/G/1 (e) BC Trace - MAP/G/1 (f) LM Trace - MAP/G/1

Fig. 8: Response time and CPU allocation with the mean response time requirement. For the SYNT trace, $R^{max} = 0.1$s; for the BC trace, $R^{max} = 0.3$s; for the LM trace, $R^{max} = 1.5$s.



(a) SYNT Trace (b) BC Trace (c) LM Trace

Fig. 9: Response time and CPU allocation with the threshold-based policy.

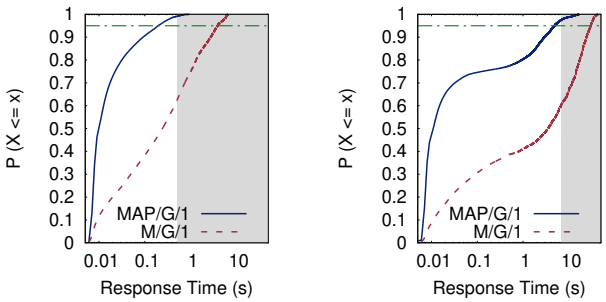
We report the results of these experiments in Tab. III. We can note that, using the synthetic trace, which has abrupt workload variations, setting $N_E = 100,000$ leads to performance violation, as changes are not quickly identified by the Workload Analyzer. A negative impact of the largest window is also experienced using the BC trace, with reduced error though. Using the LM trace, we instead notice a less evident impact of the window size, with the performance goal being met in all the considered cases. However, it is worth observing that for this trace larger windows lead to slightly better solutions, i.e., the response time requirement is satisfied using less CPU resources. These results show that identifying the best value for the window size is difficult, as this choice depends on the workload characteristics, which are not known in advance. MEAD overcomes this issue by considering multiple sub-windows within the overall monitored trace, and producing more than a single workload model accordingly.

D. Computational Cost

As regards MEAD overhead, we remark that the auto-scaling controller is executed asynchronously with respect to the data processing and possibly on a different machine, hence it does not interfere with operator execution. The

TABLE II: Results with the RT percentile requirement.

Trace	Model	$\mathbf{R^{max}}$	RT P_{95} (s)	CPU (%)
SYNT	M/G/1	0.5	1.478	46.6
SYNT	MAP/G/1	0.5	0.371	60.0
SYNT	M/G/1	0.75	1.184	46.6
SYNT	MAP/G/1	0.75	0.590	49.7
BC	M/G/1	0.5	3.871	47.1
BC	MAP/G/1	0.5	0.195	58.6
BC	M/G/1	1	3.711	47.1
BC	MAP/G/1	1	0.274	54.5
LM	M/G/1	7	30.057	45.5
LM	MAP/G/1	7	4.830	60.2
LM	M/G/1	10	29.642	45.7
LM	MAP/G/1	10	9.546	55.9



(a) BC Trace (b) LM Trace

Fig. 10: Response time CDF with different auto-scaling policies and workloads.

Workload Monitor, as explained, fetches input data in parallel with the application. We verified that the overhead incurred by RabbitMQ for handling the exchange and the additional consumer is limited: running the application with and without MEAD, we observed no change in the memory usage of RabbitMQ (about 200MB in both cases) and an increase in the CPU utilization that overall is still negligible (from 3% to 5%) compared to the resource savings enabled by MEAD.

We identified MAP fitting as the most computationally expensive task in MEAD, requiring on average 12s for the SYNT trace, 14s for the BC trace, and 20s for the LM trace.[8] As we used two fitting windows, MEAD spent between 25 and 40 seconds for workload analysis at each iteration. We considered this time acceptable in our scenario, where MEAD is activated every 60 seconds. Nevertheless, it is worth observing that, if necessary, by tuning algorithm parameters (e.g., parameters of the nonlinear solver used in KPC-TOOLBOX, or maximum number of MAP states) it is possible to trade-off accuracy with speed.

The resolution of the queueing model is much faster than fitting, especially when focusing on the mean response time, with each invocation taking about 200 ms (50 ms with the baseline M/G/1 model). When the response time percentiles must be computed, the resolution time increases to about 2.5 s (1.5 s with the M/G/1 model).

[8]The execution time differs under the various workloads, because KPC-TOOLBOX picks a different number of states depending on the trace characteristics.

TABLE III: Results varying the fitting window size N_e.

Trace	$\mathbf{N_e}$	$\mathbf{R^{max}}$	RT (s)	CPU (%)
SYNT	25,000	0.2	0.126	49.9
SYNT	50,000	0.2	0.141	48.7
SYNT	100,000	0.2	0.468	47.6
BC	25,000	0.5	0.175	53.0
BC	50,000	0.5	0.217	51.2
BC	100,000	0.5	0.516	49.8
LM	25,000	3	2.126	54.2
LM	50,000	3	2.415	52.3
LM	100,000	3	2.267	53.0

VIII. RELATED WORK

DSP operator auto-scaling has attracted significant interest within the research community, especially as regards horizontal scaling. As surveyed in [1], scaling policies have been devised using several techniques, including, e.g., threshold-based policies [2], [16], queuing theory [3], [17], game theory [18], control theory [19], reinforcement learning [4], [20]. Some effort has also been spent exploring mechanisms to reduce the often significant overhead of horizontal scaling (e.g., in [21]).

So far, vertical auto-scaling has been mainly adopted in the field of software containers, e.g., in ElasticDocker [22], which employs a threshold-based policy to scale CPU and memory for each container, and in Autopilot [23], which uses machine learning techniques to control both horizontal and vertical scaling for jobs in the Cloud. In the context of DSP, the work closest to ours is Q-Flink [24], where authors exploit cgroups to dynamically allocate CPU and memory to Flink operators so as to mitigate resource contention. They propose a policy based on *model-predictive control* (MPC), where GI/G/N queues, which do not capture burstiness, are used to estimate operator response time. De Matteis and Mencagli [19] also exploit MPC for elastic DSP on multi-core systems: they rely on horizontal scaling for performance, whilst vertical elasticity enables energy savings through CPU voltage and frequency scaling. They assume independent arrivals, relying on Kingman's formula for response time estimation.

ChronoStream [25], an elastic DSP system for the cloud, also leverages both horizontal and vertical auto-scaling, where vertical elasticity is achieved by scaling up the core usage of resource containers; no detail is provided about the scaling policy. In StreamMine3G [26], a distributed data streaming engine, vertical elasticity is achieved through a thread pool that can be expanded and shrunk; also in this case, the scaling policy is not detailed.

Besides the already mentioned ones, other works adopt queueing theory for DSP performance evaluation. Kingman's approximation is also used by Lohrmann et al. [3] to drive horizontal scaling, and by Truong et al. [9] for performance analysis of large-scale cloud DSP systems. Fu et al. [17] instead propose a horizontal elasticity solution that relies on extended Jackson networks, where independent arrivals are assumed as well. Tolosana-Calasanz et al. [27] study a VM auto-scaling solution for cloud DSP systems, which is based on feedback-control and relies on Little's Law for performance estimation.

Mencagli et al. [28] present a tool for static optimization of DSP applications that relies on queueing networks and consider the backpressure effect due to finite operator buffer capacity. However, their analysis is limited to application throughput, whereas we focus on response time optimization. Recently, Cooper et al. [29] proposed a queueing model of single streaming operators, where the cost of batched data transfers from an external queue is considered. However, differently from our work, they rely on the assumption of Poisson arrivals. We plan to extend MEAD in the future considering these backpressure and batching dynamics, as well as integrating more complex models for the analysis of stream forks and joins (e.g., [16], [30], [31]).

Differently from MEAD, none of the works referenced above considers workload burstiness, whose effects have been mostly neglected in DSP performance management. Exceptions to this trend are [5] and [6]. Drougas and Kalogeraki [5] cater for bursts impact on application throughput in their auto-scaling solution, but do not consider burstiness effects on response time. Mencagli et al. [6] instead consider the different problem of dynamic stream routing, taking into account burstiness in their control-theoretic solution. They use MAPs to generate traces for experiments, whereas burstiness is only measured by means of the index of dispersion of arrivals.

IX. Conclusion

We have presented MEAD, a framework for vertical auto-scaling of DSP operators, which exploits Markovian Arrival Processes to characterize workloads online and estimate performance. Having implemented MEAD in Flink, we evaluated our framework using both synthetic and real-world workloads. Our experiments show that MEAD is able to satisfy response time requirements under burstiness, whereas other models are highly inaccurate in this scenario.

Future research directions include the integration of horizontal auto-scaling as a complementary mechanism, which is necessary, on a larger time-scale, to scale operator processing capacity beyond the limits of single CPUs. We will also extend MEAD with models that specifically target stream fork-joins and the blocking effects due to finite operator buffers.

Acknowledgments

The work of G. Casale is partially supported by RADON, funded by the EC Horizon 2020 research and innovation program under grant agreement No. 825040.

References

[1] H. Röger and R. Mayer, "A comprehensive survey on parallelization and elasticity in stream processing," *Comput. Surveys*, vol. 52, no. 2, 2019.

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

[3] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *Proc. IEEE ICDCS '15*, 2015, pp. 399–410.

[4] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Gener. Comput. Syst.*, vol. 87, pp. 171–185, 2018.

[5] Y. Drougas and V. Kalogeraki, "Accommodating bursts in distributed stream processing systems," in *Proc. IEEE IPDPS '09*, 2009.

[6] G. Mencagli, M. Torquati, M. Danelutto, and T. De Matteis, "Parallel continuous preference queries over out-of-order and bursty data streams," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2608–2624, 2017.

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

[8] J. F. C. Kingman, "The single server queue in heavy traffic," *Math. Proc. Camb. Philos. Soc.*, vol. 57, no. 4, p. 902–904, 1961.

[9] T. M. Truong, A. Harwood, R. O. Sinnott, and S. Chen, "Performance analysis of large-scale distributed stream processing systems on the cloud," in *Proc. IEEE CLOUD '18*, 2018, pp. 754–761.

[10] M. F. Neuts, "A versatile Markovian point process," *J. Appl. Probab.*, vol. 16, no. 4, p. 764–779, 1979.

[11] S. Spinner, G. Casale, F. Brosig, and S. Kounev, "Evaluating approaches to resource demand estimation," *Perform. Eval.*, vol. 92, 2015.

[12] S. Chakravarthy and Q. Jiang, *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*, 1st ed. Springer, 2009.

[13] M. F. Neuts, "Matrix-analytic methods in queuing theory," *Eur. J. Oper. Res.*, vol. 15, no. 1, pp. 2–12, 1984.

[14] G. Casale, E. Z. Zhang, and E. Smirni, "KPC-Toolbox: Best recipes for automatic trace fitting using Markovian Arrival Processes," *Perform. Eval.*, vol. 67, no. 9, pp. 873–896, 2010.

[15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *Bull. IEEE Comp. Soc. Tech. Comm. Data Eng.*, vol. 36, no. 4, pp. 28–38, 2015.

[16] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez, "StreamCloud: An elastic and scalable data streaming system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, 2012.

[17] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang, "DRS: Auto-scaling for real-time stream analytics," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, p. 3338–3352, 2017.

[18] G. Mencagli, "A game-theoretic approach for elastic distributed data stream processing," *ACM Trans. Auton. Adapt. Syst.*, vol. 11, no. 2, 2016.

[19] T. De Matteis and G. Mencagli, "Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing," *SIGPLAN Not.*, vol. 51, no. 8, 2016.

[20] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *Proc. ACM DEBS '14*, 2014, p. 318–321.

[21] A. Shukla and Y. Simmhan, "Toward reliable and rapid elasticity for streaming dataflows on clouds," in *Proc. IEEE ICDCS '18*, 2018, pp. 1096–1106.

[22] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with ELASTICDOCKER," in *Proc. IEEE CLOUD '17*, 2017, pp. 472–479.

[23] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: Workload autoscaling at google," in *Proc. ACM EuroSys '20*, 2020, pp. 16:1–16:16.

[24] M. R. Hoseinyfarahabady, A. Jannesari, J. Taheri, W. Bao, A. Y. Zomaya, and Z. Tari, "Q-Flink: A QoS-aware controller for Apache Flink," in *Proc. IEEE/ACM CCGRID '20*, 2020, pp. 629–638.

[25] Y. Wu and K. L. Tan, "ChronoStream: Elastic stateful stream computation in the cloud," in *Proc. IEEE ICDE '15*, 2015, pp. 723–734.

[26] A. Martin, A. Brito, and C. Fetzer, "Scalable and elastic realtime click stream analysis using streammine3g," in *Proc. ACM DEBS '14*, 2014.

[27] R. Tolosana-Calasanz, J. Diaz-Montes, O. F. Rana, and M. Parashar, "Feedback-control queueing theory-based resource management for streaming applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 1061–1075, 2017.

[28] G. Mencagli, P. Dazzi, and N. Tonci, "SpinStreams: A static optimization tool for data stream processing applications," in *Proc. ACM/IFIP Middleware '18*, 2018, p. 66–79.

[29] T. Cooper, P. Ezhilchelvan, and I. Mitrani, "A queuing model of a stream-processing server," in *Proc. IEEE MASCOTS '19*, 2019.

[30] P. M. Fiorini and L. Lipsky, "Exact analysis of some split-merge queues," *SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 2, p. 51–53, 2015.

[31] Y. Zeng, J. Tan, and C. H. Xia, "Fork and join queueing networks with heavy tails: Scaling dimension and throughput limit," *SIGMETRICS Perform. Eval. Rev.*, vol. 46, no. 1, p. 122–124, 2018.