Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems

Federico Lombardi, Leonardo Aniello, Silvia Bonomi, and Leonardo Querzoni

Abstract—Distributed stream processing frameworks are designed to perform continuous computation on possibly unbounded data streams whose rates can change over time. Devising solutions to make such systems elastically scale is a fundamental goal to achieve desired performance and cut costs caused by resource over-provisioning. These systems can be scaled along two dimensions: the operator parallelism and the number of resources. In this paper, we show how these two dimensions, as two symbiotic entities, are independent but must mutually interact for the global benefit of the system. On the basis of this observation, we propose a fine-grained model for estimating the resource utilization of a stream processing application that enables the independent scaling of operators and resources. A simple, yet effective, combined management of the two dimensions allows us to propose ELYSIUM, a novel elastic scaling approach that provides efficient resource utilization. We implemented the proposed approach within Apache Storm and tested it by running two real-world applications with different input load curves. The outcomes backup our claims showing that the proposed symbiotic management outperforms elastic scaling strategies where operators and resources are jointly scaled.

Index Terms—Cloud, elasticity, elastic scaling, stream processing, storm

1 Introduction

Stream processing systems (SPSs) process unbounded streams of input tuples by evaluating them according to a given set of queries. Queries are usually modeled as graphs, where vertices represent processing elements called operators and edges correspond to streams of tuples moved between operators. This data processing model allows to break down complex computations into simpler units (the operators), independently parallelize them, and deploy the resulting system over any number of computing machines. Having the computation executed in parallel by several distinct operators on many machines is the core feature of *distributed stream processing systems*. Such flexibility allows to scale horizontally in such a way to provide the computational power required to sustain a given tuple *input load* with a reasonable processing latency. Thanks to these characteristics, SPSs today represent a fundamental building block for a large number of big data computing infrastructures [1].

A complex challenge SPSs need to cope with is input dynamism. Such systems, in fact, are designed to ingest data from heterogeneous and possibly intense sources like sensor networks, monitoring systems, social feeds, etc. that are often characterized by large fluctuations in the input data rates. Solutions based on over-provisioning are considered cost-ineffective in a world that moves toward on-demand resource provisioning built on top of IaaS platforms.

Recently, researchers introduced the idea of *elastic* SPSs that continuously adapt at runtime to changes in the input rates, to accommodate load fluctuations by provisioning more resources only when needed. The requirements for the controller of an elastic SPS have been informally defined in [2] as SASO properties [3]: *stability*,[1] *accuracy*,[2] *short settling time*[3] and *no overshoot*.[4] Several optimizations have been identified [4] and several approaches [5], [6] have been proposed to make SPSs elastically scale. These solutions scale the system by increasing operators' parallelism (*operator scaling* or *fission*) and accordingly provisioning new computing resources (*resource scaling*). However, by looking at how SPSs work under stressing workloads, it is apparent that operator and resource scaling address two distinct aspects of a same problem. In particular, operator scaling allows to subdivide the load of the specific computation implemented by an operator, thus enabling efficient resource usage through load balancing. On the other hand, correct provisioning through resource scaling is crucial to avoid excessive contention for the execution of the operators.

In this paper we claim that, although both aspects must be taken into account, they don't need to be always exercised jointly and that it is possible to build a more efficient elastic scaling solution for SPSs by accurately managing them. In particular, we advocate a "symbiotic" management of operator and resource scaling, where their independent and/or combined effects increase the global efficiency of the system. We introduce *Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems* (ELYSIUM), a new elastic scaling solution for distributed SPSs that scales operators and resources in a symbiotic fashion to let the system work

• *The authors are with the Research Center of Cyber Intelligence and Information Security, Department of Computer, Control, and Management Engineering Antonio Ruberti, Sapienza University of Rome, Roma 00185, Italy. E-mail: {lombardi, aniello, bonomi, querzoni}@dis.uniroma1.it.*

Manuscript received 28 Feb. 2017; revised 1 Aug. 2017; accepted 29 Sept. 2017. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Federico Lombardi.)
Recommended for acceptance by B. Urgaonkar.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2762683

1. *Stability*: the system configuration does not oscillate.
2. *Accuracy*: the system configuration maximizes the throughput.
3. *Short settling time*: the system quickly reaches a stable configuration.
4. *No overshoot*: the system does not use more resource than necessary.

always in a correctly provisioned configuration where the least amount of resources are wasted (4th SASO property). Scaling actions in ELYSIUM can be executed either in a reactive or proactive fashion. Indeed, ELYSIUM employs a prediction module to forecast variations in the input load and periodically checks if the current provisioning configuration needs to be scaled-in/out to accommodate for foreseeable load fluctuations. A tunable *assessment period* parameter allows ELYSIUM to avoid oscillations (1st SASO property); ELYSIUM first adapts the parallelism for each operator used by the application to avoid bottlenecks on operator instances. Then it checks if the current resource provisioning is the smallest that will let the system work without incurring any performance degradations. For this last check, ELYSIUM leverages a novel *resource estimator* to compute the expected resource consumption, given an input load and a configuration, so as to accurately and quickly adapt to the workload in a single reconfiguration (2nd and 3rd SASO properties). A monitoring system lets ELYSIUM collect at runtime fine-grained information on resource usage that is then used to decide how the system must be scaled. With this approach ELYSIUM can scale independently operators and resources as well as jointly scale them, whenever this is needed.

Summarizing, we provide the following contributions:

- we explain why operator and resource scaling impact on two distinct aspects of SPSs scalability, and propose how to symbiotically manage them to elastically provision the system in a more efficient way;
- we introduce ELYSIUM, a reactive/proactive elastic scaling solution for SPSs that consider operator and resource scaling as two distinct solutions that need to be combined only when necessary; ELYSIUM employs a fine-grained model of resource usage to estimate how the SPS will behave under a given load, which enables to properly choose how many instances (for each operator) and resources to set;
- we provide an in-depth evaluation of ELYSIUM's performance by testing a prototype on real stream processing applications under different workloads and comparing it with a standard elastic scaling solution employing only the joint scaling approach.

Paper Structure. Section 2 defines more formally the system model and the problem to tackle, so that in Section 3 we can present our approach. Section 4 presents the ELYSIUM implementation on Apache Storm, while the experimental evaluation is described in Section 5. Related works are discussed in Section 6 and, finally, Section 7 sums up the paper and points out future work.

2 System Model and Problem Statement

We model a computation in a SPS as a directed acyclic graph where vertices represent *operators* and edges represent *streams* of tuples between pairs of operators (see Fig. 1). We define such a graph as an *application*. Each operator carries out a piece of the overall computation on incoming tuples and emits downstream the results of its partial elaboration. In general, an operator has n_i input streams (0 for source operators) and n_o output streams (0 for sink operators). An application is also characterized by an input load that varies over time and represents the rate of tuples fed to the SPS for such application (*input rate*). Each input tuple generates multiple tuples that traverse several streams in the application graph. The processing of some of these tuples may possibly fail; in this case we say that the tuple is *failed*. Conversely, if all the tuples generated in the graph are correctly processed, then we say that the corresponding tuple is *acked*. The rate of tuples that are acked over time is referred to as *throughput*.

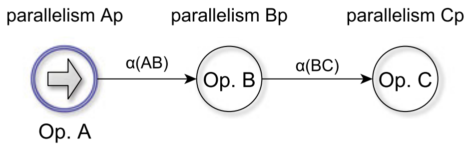


Fig. 1. SPS computation model.

For the sake of simplicity, and without loss of generality, we assume that a stream connecting operators A (upstream operator) and B (downstream operator) can be uniquely identified by the pair (A, B), which means that no two distinct streams can connect the same pair of operators. The *selectivity* for a stream (A, B) is defined as the ratio between the tuple rate of (A, B) and the sum of the tuple rates of all the input streams of A, i.e., the selectivity of (A, B) measures its tuple rate as a function of the total input rate of A [4]. In this paper we assume to work with SPS applications having constant average operators' selectivities at runtime, similarly to applications presented in [7], [8], [9].

To let the application scale at runtime each operator can be instantiated multiple times such that its instances will share the load provided by the input stream. However, the maximum number of instances for an operator op is upper-bounded by an application parameter max_parall_{op}. Each operator instance runs sequentially and uses a single CPU core at a time.[5] The number of available instances for a given operator is defined as its level of *parallelism*. As a consequence, a stream (A, B) can be constituted by several sub-streams, each connecting one of the instances of operator A to one of the instances of operator B. To simplify the discussion, we assume that the SPS is able to fairly distribute the load among the available instances of each operator. This is achieved by means of grouping functions that manage how tuples in a stream are mapped in its sub-streams [12], [13], [14].

When an application is run, the SPS uses a *scheduler* to assign the execution of each single operator instance to a *worker node* among the many available in a computing cluster. We assume that worker nodes in the cluster are homogeneous (same configuration for CPU cores, speed, memory, etc.) and can be activated on-demand as for an IaaS provider. At each point in time the application *configuration* is defined by the parallelism for each operator and the number of used worker nodes. We define three possible states for a worker node at runtime by comparing its CPU usage[6] to two thresholds (*cpu_min_thr* < *cpu_max_thr*):

- *cpu_low*: CPU usage < *cpu_min_thr*
- *cpu_avg*: *cpu_min_thr* ≤ CPU usage ≤ *cpu_max_thr*
- *cpu_stress*: CPU usage > *cpu_max_thr*

5. This operator execution model is common to several SPSs like Apache Storm [10] and Apache Flink [11].
6. We consider the cumulative CPU usage on all its cores, averaged over a sliding window to avoid oscillations.

Similarly, we define three possible states for an operator instance at runtime by comparing its CPU core usage to two thresholds (*core_min_thr* $<$ *core_max_thr*):

- *oper_low*: core usage $<$ *core_min_thr*
- *oper_avg*: *core_min_thr* $\leq$ core usage $\leq$ *core_max_thr*
- *oper_stress*: core usage $>$ *core_max_thr*

The configuration of a SPS is *correct* as long as no operator instance and no worker node is in the *stress* state. Note that we are here considering CPU-bound applications. A more complete model that considers memory and bandwidth consumption is subject of our future work. Since we assume a homogeneous cluster, we define *minimal configuration* as a correct configuration having the minimum number of worker nodes required to sustain a certain input load.

The SPS can be scaled by tweaking the operator parallelism or the number of available worker nodes. These two operations can be performed independently as they address two different issues: operator overloading and scarceness of computing resources, respectively. In some cases, increasing the level of parallelism for an operator, i.e., increasing the number of instances for that operator, may also impact resource provisioning demanding for further working nodes. Furthermore, we cannot exclude that in some (unfrequent) cases scaling up the parallelism of an operator may possibly induce a reduction of resource provisioning. The operations of increasing or decreasing an operator parallelism or the number of available worker nodes are named *scale out* and *scale in* respectively.

We consider that reconfigurations have a cost (*reconfiguration overhead*) due to (i) the elastic controller execution and (ii) a period of performance degradation whose amplitude and duration are mainly related to: (a) R_{state}, i.e., the operator state migration time; (b) $R_{restart}$, i.e., the time due to topologies restarting; (c) R_{queue}, i.e., the time to process tuples queued during $R_{state} + R_{time}$. These time periods strictly depend on the specific strategies employed by the SPS to handle application reconfigurations at runtime.

The problem we tackle in this paper is how to choose, at runtime, configurations for a SPS in such a way that all will be correct despite variations in the input load (i.e., number of tuples per second injected in the system). Ideally, these configurations should also be minimal but we cannot guarantee such a property.

3 ELYSIUM

3.1 Symbiotic Scaling Strategy

ELYSIUM is based on the following idea: stress at the operator instance level and stress at the worker node level are two different issues that can be addressed by separately scaling-in/out operators and worker nodes. In some cases, the two issues are interrelated in such a way that both operators and worker nodes will be scaled-in/out. Fig. 2 depicts the different scaling strategies used by ELYSIUM. Fig. 2a shows the operator scaling operation where for one or more operators the number of parallel instances is decreased or increased. This strategy can be adopted when an operator instance is in a *oper_stress* status, as this may indicate that a single instance is saturating a CPU core because it is overloaded by incoming tuples. By increasing the operator parallelism we increase the probability that its load will be shared among other instances, thus alleviating its stress state. Fig. 2b shows the dual *resource scaling* operation performed to scale-in/out resources by adding or removing worker nodes assignable by the SPS scheduler. This strategy can be adopted when one or more worker nodes have their CPU in a *cpu_stress* status, as this may indicate that the resources available to the SPS scheduler are insufficient to handle the global application input load. By increasing the amount of available resources we decrease stress on pre-existing worker nodes, thus allowing the SPS to ingest more data for the application. Finally, Fig. 2c shows a *joint scaling* operation where resources and operators are scaled-in/out together. This strategy can be adopted when the scale-out of one or more operators saturates available resources, thus requiring a resource scale-out operation. This is the strategy employed by most of the elastic scaling solutions for SPS present in the state of the art (see Section 6). The picture shows that we don't rule out the possibility of scaling-in resources after having scaled-out processes (and vice-versa). These counter-intuitive scenarios may arise in specific setting where, for example, after an operator scale-out decision, the SPS scheduler is a able to better distribute instances over the available worker nodes, thus reducing the global load on the cluster.

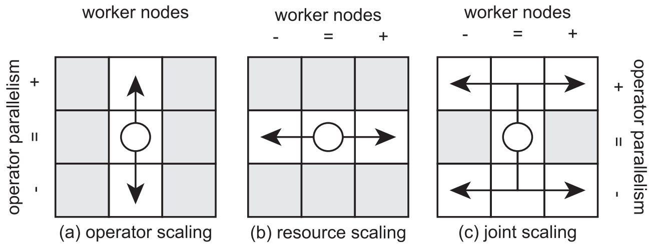


Fig. 2. Scaling options in a distributed stream processing system.

3.2 Architecture

ELYSIUM profiles the SPS and the applications running on top of it with the aim of producing accurate estimations about the resource consumption a specific configuration can cause given a certain input load. By leveraging such estimations, ELYSIUM periodically calculates a new configuration to be adopted by the SPS during the next *assessment period*. This calculation is performed striving to minimize the number of used worker nodes, while providing a configuration that will be correct with high probability for the whole duration of the next period. The assessment period can be tuned depending on the specific cluster characteristics, and accordingly to the desired tradeoff between (i) the need to reduce the amount of time the system will run in a non correct configuration, and (ii) the reconfiguration overhead caused by adopting each new configuration.

ELYSIUM can be used either in *reactive* or *proactive* mode. The difference lies in the input load used for the estimations: if the real current input load is used, then ELYSIUM scales *reactively*, otherwise, if input load is forecasted over a certain prediction horizon, then ELYSIUM scales *proactively*. In the former case the assessment is performed such that the new configuration is correct with respect to the recently observed input load. Conversely, in the latter case ELYSIUM uses the

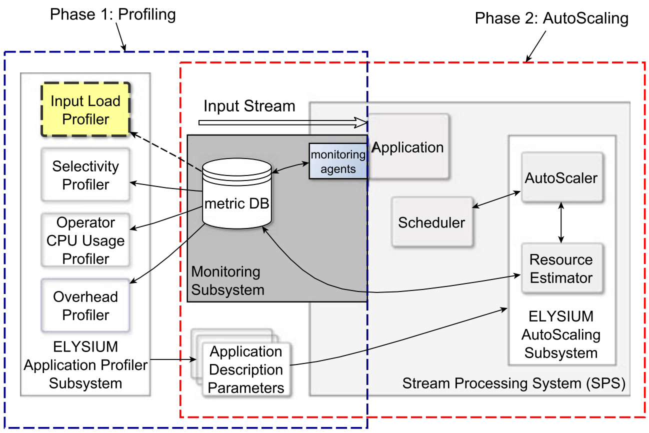


Fig. 3. ELYSIUM Architecture integrated in the SPS. The dotted blue line indicates modules involved in the first phase of application profiling, the red dotted one those involved in the second phase of autoscaling. The Input Load Profiler, represented with a yellow background, is used only when switching from reactive to proactive mode.

maximum predicted input load for the next assessment period as a metric to identify correct configurations.

While working in reactive mode, ELYSIUM profiles the applications to learn what would be the CPU usage for the worker nodes in a certain configuration when a given input load is fed to the SPS. This is accomplished by splitting ELYSIUM execution in two phases: a *profiling phase*, where it learns these information, and an *autoscaling phase*, where it makes periodical assessments leveraging learned application profile. While working in proactive mode, the profiling phase also includes an input load learning step used to enable load prediction.

ELYSIUM's architecture (Fig. 3) includes three subsystems: (i) a Monitoring subsystem which collects and provides the metrics required to carry out the two phases, (ii) an Application Profiler subsystem implementing the phase 1 and (iii) an AutoScaling subsystem for the phase 2.

Monitoring Subsystem. The Monitoring subsystem consists of a set of *monitoring agents* deployed over the worker nodes and a *metric DB* where metrics are stored. Each metric agent monitors the operator instances running on the same worker node where it is deployed, collects metrics and periodically stores average values computed over a sliding time window into the metric DB. Collected metrics are (i) inter-operator instance traffic, measured as the tuple rate for each pair of communicating operator instances, (ii) CPU usage of each operator instance and (iii) CPU usage of the whole worker node due the SPS. In proactive mode, also the input load is collected, and it is measured as the tuple rate in input to each application running in the SPS.

Application Profiler Subsystem. The Application Profiler subsystem is in charge of learning specific characteristics of a running application by analyzing the data stored in the metric DB after that application ran for a sufficiently long period of time (see Section 5). While working in reactive mode, it includes three distinct profilers, each aimed at learning a specific aspect of an application: (i) the *Selectivity Profiler* (SP) learns the selectivity of each operator (see Section 2), (ii) the *Operator CPU Usage Profiler* (OCUP) learns how the CPU usage of each operator instance varies as a function of its input rate and (iii) the *Overhead Profiler* (OP) learns how the CPU usage of a worker node varies depending on the sum of the CPU usages of its operator instances. The latter is required as, typically, SPSs impose some overhead over running applications to provide basic services like process management, message queue control threads, etc. Therefore, the worker node total CPU usage is the sum of the usage imposed by running operator instances and the overhead. While working in proactive mode, a further *Input Load Profiler* (ILP) is used, to learn input load patterns over time. The outputs from the profilers constitute the *application description parameters* (see Fig. 3) that will be used by the AutoScaling subsystem to estimate the state of worker nodes and operator instances (see Section 2). In the following, we detail the profilers and data they collect. For the sake of simplicity, the formalisms used to model managed data don't include applications' and operators' identifiers when they are obvious.

- *SP*—Extracts metrics related to inter-operator instance traffic from the metric DB to create a dataset with records in the form $\langle up\_op, dn\_op, tuple\_rate \rangle$, where $tuple\_rate$ is the average tuple rate of the stream from up_op upstream operator instance to dn_op downstream operator instance. The output of the SP is the selectivity for each stream, as defined in Section 2.
- *OCUP*—Retrieves data from the metric DB to create a dataset having records for each operator instance structured as $\langle tuple\_rate, cpu\_usage \rangle$, where $tuple\_rate$ is the average input rate of the operator instance, and cpu_usage is the CPU usage (in Hz[7]) that the worker node needs to run the operator instance. The output of the OCUP is a function for each operator that, given the input rate, returns the expected CPU usage that one of its instance entails.
- *OP*—Reads the metric DB to extract a dataset consisting of records in the form $\langle cpu\_usage\_ops, cpu\_usage\_sps \rangle$. Each record maps the sum of CPU usages of all operator instances running on a worker node (cpu_usage_ops) with the CPU usage of that worker node (cpu_usage_sps). Its output is a function that returns the expected CPU usage of a worker node due to the SPS overhead, given the sum of CPU usages due to the operator instances running on it.
- *ILP*—Profiles input load over time for each running application, on the base of data extracted from the metric DB. Such dataset includes records in the form $\langle ts, input\_load \rangle$, where $input\_load$ is the average input load observed during one minute[8] starting at timestamp ts. The output of the ILP is a function that returns the maximum input load expected during the next prediction horizon, given in input the day of the week, the hour, the minute, and the input loads seen in the last n_{ILP} minutes, where n_{ILP} is a configuration parameter whose value must be tuned empirically. This kind of input enables a combination of prediction approaches: one simply based on

7. Using Hz as a metric for CPU usage allows our system also to support heterogeneous nodes.
8. We considered a one minute granularity for collecting input load data as this value, from our experimental evidence, provided the best compromise for input load predictions [15].

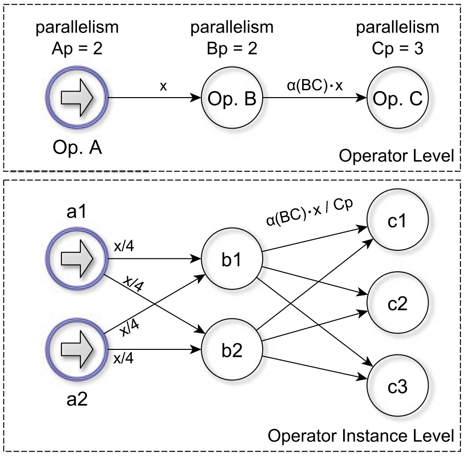


Fig. 4. Example of estimator's functioning on a 3 operator application. The estimator, through the method `getOperatorInputRate()`, starting from an input load x, traverses the application graph by using the selectivities provided by the SP to compute the input rate of each operator; in the figure above the input rate of the operator B is x, while the input rate of the operator C is $\alpha BC \cdot x$, where αBC is the selectivity of the stream BC. Through the method `getOperatorInstanceCpuUsage()` the estimator first obtains the input rate of each operator instance by dividing the input rate of each operator by its parallelism (figure below), then, by using the function provided by the OCUP, it infers the operator instance CPU usage. Finally, through the method `getCpuUsages()` the estimator infers the CPU usage of each worker node by taking from the SPS scheduler an allocation of operator instances on worker nodes. In proactive mode the input load x is predicted though the `predictInputLoad()` method.

current time (the day of the week, the hour, the minute) to profile periodic trends, and another based on time series (input loads seen in the last n minutes) to catch behaviors depending on patterns.

AutoScaling Subsystem. The AutoScaling subsystem starts to work once the profiling phase ends, so that it can leverage the application description parameters provided by the Application Profiler subsystem. It includes two components: (i) the *Estimator*, which uses fresh data from the metric DB to compute functions provided by the profilers so as to expose methods for obtaining estimations and predictions on specific applications, and (ii) the *AutoScaler*, which starts the assessments and leverages these Estimator's methods to decide the new configuration to use.

The Estimator exposes four methods:

- **`getOperatorInputRate()`**—Traverses the application graph and uses operator selectivities obtained by the SP to compute the expected operator input rates starting from the application input load.
- **`getOperatorInstanceCpuUsage()`**—Estimates CPU usage of operator instances by dividing the total expected input rate of an operator by its parallelism and then using this value to feed the profile function returned by OCUP.
- **`getCpuUsages()`**—Provides an estimation of the CPU usage of worker nodes given (i) the allocation of operator instances to worker nodes provided by the SPS scheduler and (ii) expected CPU usage for all operator instances. The estimation for a given node is obtained by summing the CPU usage of operator instances running on it, and then feeding this value to the profile function returned by OP.
- **`predictInputLoad()`**—This methods is used only in proactive mode and returns the *maximum* input load predicted for an application for the next prediction horizon. It is implemented by computing the function provided by the ILP on the inputs obtained from the metric DB.

Algorithm 1. AutoScaling Algorithm

1: **function** COMPUTECONFIG(Estimator E, Scheduler S, List ⟨ Application ⟩ $apps$, List ⟨ int ⟩ $input\_loads$)
2: **for all** application a_k in $apps$ **do**
3: **for all** operator o_i in a_k **do**
4: $ir_i \leftarrow E.getOperatorInputRate(input\_loads_k, o_i)$
5: $p_i \leftarrow 1$
6: **while** $E.getOperatorInstanceCpuUsage(o_i, \frac{ir_i}{p_i}) > core\_max\_thr \;\&\; p_i < max\_parall_{o_i}$ **do**
7: $p_i \leftarrow p_i + 1$
8: **while** $E.getOperatorInstanceCpuUsage(o_i, \frac{ir_i}{p_i}) < core\_min\_thr \;\&\; p_i > 1$ **do**
9: $p_i \leftarrow p_i - 1$
10: $worker\_nodes \leftarrow 1$
11: **while** $true$ **do**
12: $allocation \leftarrow S.allocate(apps, worker\_nodes)$
13: $cpu\_usages \leftarrow E.getCpuUsages(allocation, input\_loads)$
14: **if** $\forall x \in cpu\_usages : x \leq cpu\_max\_thr$ **then**
15: **return** $worker\_nodes, \{p_i\}$
16: $worker\_nodes \leftarrow worker\_nodes + 1$

Fig. 4 shows an example of how the Estimator works. The AutoScaler module works by invoking periodically its **`computeConfig()`** method (reported in Algorithm 1) accordingly to the configured assessment period. This operation allows to choose the configuration to apply in order to efficiently sustain an expected input load during the next assessment period. It takes as input (i) a reference to the Estimator component, (ii) a reference to the SPS scheduler used to compute allocations of operator instances to worker nodes, (iii) the list of applications currently running in the SPS, and (iv) the corresponding input loads. In reactive mode, these input loads are directly read from the metric DB, while in proactive mode they are predicted by the Estimator and obtained by calling the `predictInputLoad()` method for each running application. The computation of a new configuration is performed by two consecutive stages. First, the parallelism of each operator is adapted to avoid any CPU core overloading or under-utilization (*operator parallelism scaling*). Then, the minimum number of worker nodes is identified to run all the operator instances without saturating the CPU of any worker node (*resource scaling*). Each stage decides a scaling action along a different dimension, and the second one takes into account the possibly updated operators' parallelism decided in the first stage.

The first stage (lines 2-9 of Algorithm 1) analyzes for each running application a_k the status of their operator o_i. Estimator's methods `getOperatorInputRate()` and `getOperatorInstanceCpuUsage()` are invoked to evaluate the CPU load that each of the p_i instances (where p_i is initialized to 1) of o_i would produce on the CPU core where it is running, given the input load for a_k. Since it is assumed that the input rate of an operator gets equally split among its instances (see Section 2), the value of p_i can be adjusted

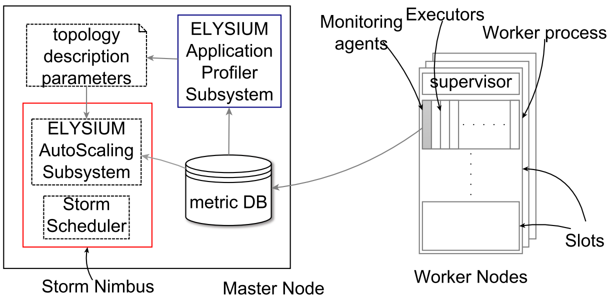


Fig. 5. ELYSIUM deployment in storm.

by increasing it in case of core overloading (estimated CPU core load greater than $core\_max\_thr$), or decreasing it in case of core under-utilization (estimated CPU core load lower than $core\_min\_thr$), until a steady point is reached, i.e., operators in state oper_avg.

In the second stage (lines 10-16 of Algorithm 1), multiple potential configurations, each differing for the number of used worker nodes are checked. The process starts by checking the configuration with the least number of workers nodes (i.e., 1 node) and proceeds by increasing the worker nodes one at a time until a configuration is found that has no bottleneck: this is the configuration that will be used in the next assessment period. For each configuration, the scheduler is requested to produce an allocation, which is a mapping of the operator instances of running applications to the worker nodes of the configuration to test. Such an allocation and the input load of each application are passed to the `getCpuUsages()` method exposed by the Estimator, and the list of CPU usages of the worker nodes in the configuration being checked is obtained. If any of such worker nodes is in stress state, then the current configuration does not contain enough available resources for the computation; one more worker node must be added and the new configuration needs to be checked again. Conversely, the number of worker nodes with the number of instances for each operator of the submitted applications is returned.

4 ELYSIUM Implementation in Storm

In this section we describe how we implemented each component of ELYSIUM and how we integrated it into Apache Storm [10], a widely adopted framework for distributed SPS. The way ELYSIUM is integrated with Storm is shown in Fig. 5 and described in following sections.

In Storm jargon applications, called *topologies*, are represented as acyclic graphs of operators, called *components*. Source components are called *spouts*, while all the others are named *bolts*. Spouts usually wrap external data sources and generate the input load for applications. At runtime, each component is executed by a configurable number of threads, called *executors*, which are the instances of the operators. Storm does not provide support for stateful operator migration at runtime. For this reason, in this implementation we consider $R_{state} = 0$.

A Storm cluster comprises a single master node (*Nimbus*) which coordinates all the other nodes each locally managed by a special process called *Supervisor*. Each Supervisor provides a fixed number of Java processes (*workers*) to run executors. A topology can be configured to run over a precise number of workers. The Nimbus is in charge of deciding the allocation of executors to available workers by running a *scheduling* algorithm. Application developers can use the embedded *even scheduler*, provided by Storm, or implement custom allocation strategies through a generic *scheduler interface*. As a rule of thumb, each topology should use a single worker per supervisor in order to avoid the overhead of inter-process communication. Indeed, the default scheduler strives to choose the workers for a topology in such a way. The Nimbus also provides a *rebalance* API to dynamically vary (i) the number of workers a topology can use to run its executors (*resource scaling*), and (ii) the number of executors for each component (*operator scaling*).

Monitoring Subsystem. The monitoring agents are threads that run inside the workers and monitor executor metrics by leveraging Storm's metrics framework. With reference to Section 3.2, monitored metrics are (i) the rate of tuples received by bolts (to monitor inter-operator traffic), (ii) the CPU usage of the executors, (iii) the CPU usage of the workers, and (iv) the rate of tuples emitted by spouts (to monitor the input load). Our prototype stores every 10 seconds into an Apache Derby DB [16] (the metric DB hosted on the Nimbus) average metric values computed over a sliding window of 1 minute.

Application Profiler Subsystem. Profilers are implemented as standalone Java applications. They access every 10 seconds the metric DB to extract the required data and build a dataset. Once the profiling phase ends, they produce the output functions and store them as Java objects serialized to file.

The SP provides the list of selectivities for each stream in the topology by averaging over time collected selectivities. This approach is motivated by the initial assumption on constant selectivities. Tests reported in Section 5 show that SP provides reliable predictions for selectivities also with real workloads that show little selectivity oscillations.

As output OCUP, OP and ILP produces *Artificial Neural Networks* (ANNs)[9] through Encog [17]. Specifically, the OCUP employs an ANN for each component of the topology, each one having a single input node for the input rate, and a single output node with the estimated CPU core usage. Similarly, the OP employs an input node for the sum of CPU usages due to executors and an output node with the estimated CPU usage of the worker node. The ILP employs a different ANN that takes as input the day of the week, the hour, the minute and the input loads seen in the last n_{ILP} minutes. More details about ANNs' setting and their training are discussed in Section 5.

AutoScaling Subsystem. The AutoScaling subsystem is implemented as a Java library to be imported by the Nimbus. It implements the scheduler interface, and the Nimbus is configured to be invoked periodically with period equals to the chosen assessment period. In this way, assessments are executed at the right frequency and have access to all the required information about allocations.

The Estimator is a Java object that accesses the metric DB and implements the methods introduced in Section 3.2. It

9. We consider that ANNs can be one of the best solutions for our requirements as they provide (i) a data-driven non-linear model and (ii) the ability to generalize and infer unseen parts of a population [15].

TABLE 1
Reward of Q-Learning for CPU max_threshold

CPU Max Threshold	Reward
0.60	0.53
0.65	0.57
0.70	0.62
0.75	0.66
0.80	**0.71**
0.85	0.65
0.90	0.43

loads the profiles produced by the Application Profiler subsystem by unserializing them.

The AutoScaler is the Java object that implements the scheduler interface and executes Algorithm 1. It wraps the default scheduler of Storm and uses it to simulate allocations when checking the effectiveness of configurations. In case the chosen configuration is different from the current one, it issues a rebalance operation through the Nimbus API to apply the new configuration, that is to assign (i) a different number of workers to a topology, and (ii) a different number of executors to each component.

5 Experimental Evaluation

5.1 Environment and Deployment

Testbed. The environment used to deploy and test ELYSIUM was composed by 4 blade servers *IBM HS22*, each equipped with 2 Quad-Core *Intel Xeon X5560* 2.28 GHz CPUs and 24 GB of RAM. We distributed the Storm framework on a cluster of 5 VMs, each equipped with 4 CPU cores and 4 GB of RAM. One was dedicated to hosting the Nimbus process and the *Apache Derby DB*, while the remaining 4 hosted the worker nodes. One further VM was used for the Data Driver process, in charge of generating the input load. This VM was equipped with 2 CPU cores and 4 GB of RAM. The Data Driver process generates tuples according to a given dataset, then sends them to a *HornetQ* [18] Java Messaging Service (JMS) queue. The spouts are connected to such JMS queue to get the tuples to inject into the topology.

Reference Applications and Dataset. To evaluate ELYSIUM we implemented two topologies that we refer to as T1 and T2 respectively: T1 performs a *Rolling-Top-K-Words* computation [19] and T2 implements *Sentiment Analysis* [20]. Each operator in the topologies has a parallelism in the range $[1; 4]$. We evaluated ELYSIUM by using both synthetic and real traces to generate the input load. As synthetic traces, we employed (i) a stair-shaped curve, (ii) a sine function, and (iii) a square wave. As real trace we used a subset of a 10 GB Twitter dataset containing 3 months of tweets captured during the European Parliament election round of 2014 from March to May in Italy. To make tests with the real trace practical, we applied to them a $60:1$ time-compression factor to allow the replay of the real trace with reasonable timing.

Evaluation Metrics. The effectiveness of ELYSIUM has been evaluated considering the following metrics:

- the *throughput degradation*, measured as the percentage difference over time between input load and throughput, where the throughput is rate of acked input tuples (see Section 2). The throughput degradation is computed as $\frac{|input\_load - throughput|}{input\_load}$. Note that throughput degradation becomes greater than 1 when there is a large number of input tuples buffered in the queue. In this case the throughput can become much larger than the input load, hence $|input\_load - throughput| > input\_load$;
- the percentage of *nodes saved* with respect to a statically over-provisioned configuration; let N be the number of assessments done during the evaluation, C the number of worker nodes defined in the over-provisioned configuration, c_i the configuration chosen by the ith assessment, this metric is computed as $1 - \frac{\sum_{i=1}^{N} c_i}{NC}$;
- the *latency*, i.e., the average tuple completion time.

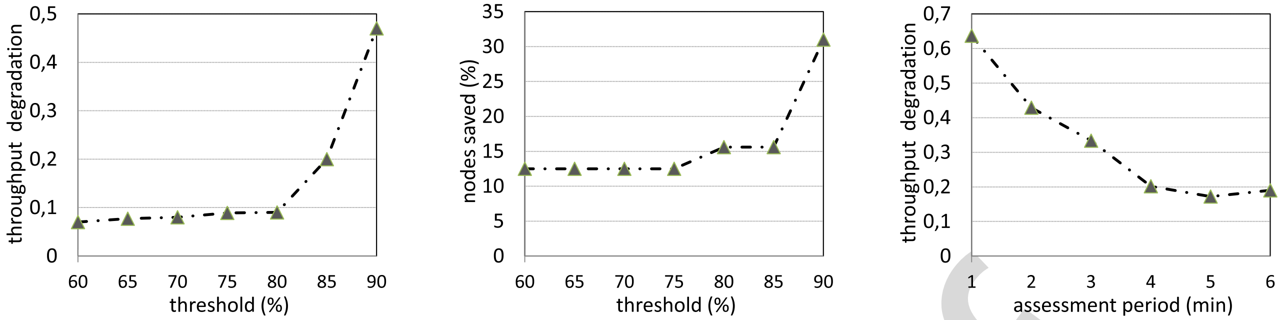
Whenever applicable these metrics have been computed over sliding time windows or as an overall value for the entire test.

Parameters Setup. All our tests were conducted using the prototype introduced in Section 4. To properly set the thresholds presented in Section 2, we adopted a methodology based on Reinforcement Learning. We used Q-Learning [21] during the profiling phase starting with no knowledge of the application behavior. To find the cpu_max_thr, the Reward Function $R(threshold)$ we propose aims at maximizing node usage, hence looks for the maximum CPU threshold that corresponds to the lowest throughput degradation; specifically $R(cpu\_max\_thr) = cpu\_max\_thr - throughput\_degradation$. The Q-Learning rewards are shown in Table 1 where it is possible to see that the max reward is given to a threshold of 0.8, i.e., 80 percent CPU usage. Fig. 6 shows how the throughput degradation and nodes saved metrics change in function of the max CPU threshold. Specifically, Fig. 6a backups the result that the 80 percent of CPU usage seems to be the best cpu_max_thr as larger values impose a larger throughput degradation. In a similar way we computed the other thresholds: their values are 0.25 for $core\_min\_thr$ and 0.65 for $core\_max\_thr$.

The ANNs have been tuned by following some empirical rules presented in [15]: the ILP ANN has 13 input nodes, 1 hidden layer with 24 neurons, 5 output nodes for *direct prediction* (i.e., a prediction for each future minute) and linear-tanH-tanH activation functions. Data are normalized with the min-max normalization $[0; 1]$ and the dataset was split 70 percent training and 30 percent test. For OCUP/OP ANN we set 1 hidden layer with 3 neurons, *tanH-tanH-tanH* activation functions. We trained the ANNs with the Resilient Backpropagation [22] and a 10-cross validation to avoid overfitting. The profiling phase duration is application dependent. Basically, the more data you collect, the more accurate the prediction will be. In our scenario we notice that injecting a variable workload for 30 minutes is enough to achieve a good prediction accuracy (see next section).

5.2 ELYSIUM Evaluation

Reconfiguration Overhead. The overhead introduced by ELYSIUM is negligible. The metrics monitoring CPU usage and



(a) Throughput Degradation (b) Nodes saved (c) Overhead due to the assessment period

Fig. 6. Throughput degradation (a,c) and nodes saved (b) due to parameters setup (threshold cpu_max_thr and assessment period).

traffic are extremely lightweight (they are collected every 10 seconds). The bandwidth consumed for metric collection is just few KB, depends on the number of operator instances, and it is independent from the input load. The real-time computation of the AutoScaler is lightweight and consumes an insignificant amount of CPU periodically. Furthermore, this computation is carried out on the machine hosting the Nimbus, so it doesn't compete for resources with running topologies. When the configuration has to be changed, the throughput of an application degrades. In our experiments, a reconfiguration is triggered by issuing a rebalance command to the Nimbus, which causes such degradation for two reasons mainly: first, topologies have to be restarted ($R_{restart}$), which takes 5 to 8 seconds in our testbed. During this period, the application cannot process tuples, so they are buffered before the spout component (into the JMS queue in our topologies). Second, once topologies become ready to work, the spouts start retrieving tuples from the input queues at the highest possible rate. This is likely to causes a non-negligible load peak with a consequent resource overloading, regardless of the actual input load curve. So, after the restart of the topology, a transient phase occurs where the cluster is likely to move in a stress state because applications need to drain accumulated input tuples (R_{queue}) to finally keep up with the real input load. The length of this transient phase depends on how many tuples are queued while the reconfiguration takes place. This is a common behavior for SPSs that, like Storm, do not allow dynamic reconfigurations of running applications at runtime.

To measure how the assessment period impacts the reconfiguration overhead, we deployed T1 over an over-provisioned configuration (no worker nodes nor operators in stress state) and injected 9 minutes of sinusoidal input load. In this setting, we computed the throughput degradation for different assessment periods. As expected, Fig. 6c clearly shows the throughput degradation gets larger as the assessment period is shortened.

TABLE 2
Selectivity of T1's Streams

Stream	Average	Std. Dev.
`WordGener - StopWordFilter`	17.86	0.54
`StopWordFilter - Counter`	0.68	0.02
`Counter - IntermRanker`	0.41	0.34
`IntermRanker - FinalRanker`	0.01	0.00

By comparing these results with the quasi-zero throughput degradation obtained without reconfigurations and in an over-provisioned setting (see Fig. 9), it can be noted that reconfiguration overhead is significant. Therefore, the assessment period has to be tuned accordingly to input load variability and throughput degradation tolerance. In our tests, we set the assessment period to 1 minute. Therefore, pessimistically assuming reconfigurations occur at each assessment, the baseline value of the throughput degradation for comparisons is 0.64 (with 2 minute assessment period, the throughput degradation would be 0.43).

Estimator Accuracy. The accuracy of the estimations provided by the Estimator depends in turn on the accuracy of the profiles learned by the SP, the OCUP, and the OP.

Table 2 shows average and standard deviation of the selectivities observed for the streams of T1, during a 30 minutes test with the stair-shaped curve as input load. Reported standard deviations are very small, which backups the implementation choice for the SP, described in Section 4, of modeling selectivities with constant values. The stream *Counter - IntermediateRanker* is the only one having a large standard deviation. This is due to the semantics of the *Counter* bolt; indeed, it sends tuples downstream to the *IntermediateRanker* bolt periodically, independently of its input rate. The impact on the estimation is negligible as at runtime the input rate of the bolts downstream the *Counter* bolt is very small and produces really small CPU usage. The accuracy of the OCUP is related to the estimations of CPU usage for an operator instance given its input rate. Average *mean percentage error* of estimations is under 3 percent. Fig. 7 reports the real CPU usage over time of the instances of two operators, aggregated by operator, and the corresponding estimations provided by the OCUP. In this test, a sinusoidal input load was injected in the topology for 25 minutes. As

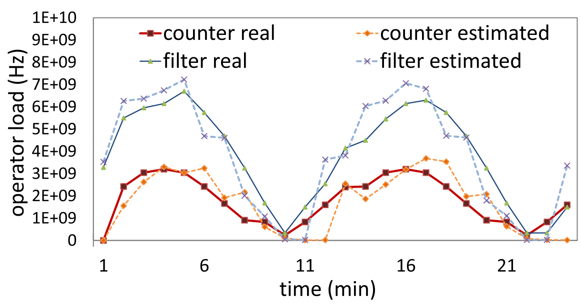


Fig. 7. Comparison between real and estimated total CPU usage (in Hz) for all instances of T1's Counter and StopWordFilter operators.

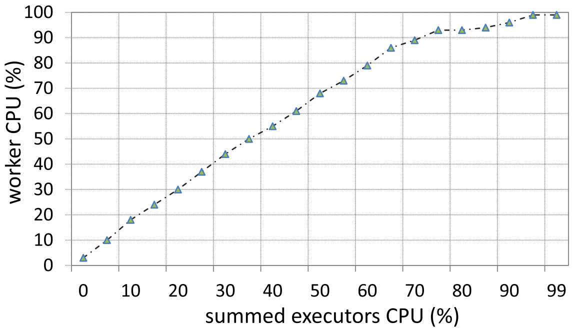


Fig. 8. Worker node CPU usage as a function of the sum of the CPU usage of all the executors running in such worker node.

the figure shows, the estimations faithfully predict the real CPU usage.

The OP estimates the CPU usage of a worker node as a function of the sum of the CPU usage of all the operator instances running in that node. In this way, it is possible to take into account the overhead caused by the SPS such as tuple dispatching and thread management. Fig. 8 depicts the profiling of such overhead in a worker node of our cluster. Such profiles provide all the information needed to infer the total CPU usage of a worker node.

Comparing Joint and Symbiotic Scaling. To define the policy enforced by the joint scaling approach, we took inspiration from [23]:[10] *operator scale-out entails adding a new resource, while operators scale-in and resource scale-in/out are independent*. This means that another worker node is added whenever any operator is scaled-out, while operator scale-in doesn't affect resource scaling. Furthermore, in case no operator is scaled, resources are scaled in or out on the base of current worker nodes' CPU usage. Since joint scaling is reactive, ELYSIUM was set in reactive mode as well and a same activation threshold for both approaches was considered, such to provide a fair comparison.

To highlight the advantage of scaling on a single dimension only, either operators or resources, we first show a case where scaling only operators, and not resources, can be enough to make the application sustain an input load peak. Fig. 10a shows a throughput comparison over T1 between a static configuration and an operator-only AutoScaler. The static configuration has 2 worker nodes and all operators with parallelism set to 1, so there are 6 executors over 8 cores (2 worker nodes with 4 cores) running operators, and 2 remaining cores used by other Storm processes.

The static configuration cannot sustain the input load change occurring at about second 180, and the throughput drops after a couple of minutes. The AutoScaler starts with the same configuration, then scales up when the peak occurs, as it detects an operator stress. It changes the parallelism of the stressed operator (*StopWordFilter* in this case) from 1 to 2 and the throughput, after some oscillations due to reconfiguration overhead, increases keeping up with the input rate. The inverse operation (operator scale-in) occurs at about second 600, where the two operator instances become under-utilized and the parallelism is set back to 1. The next experiment aims at underlining the limitation of

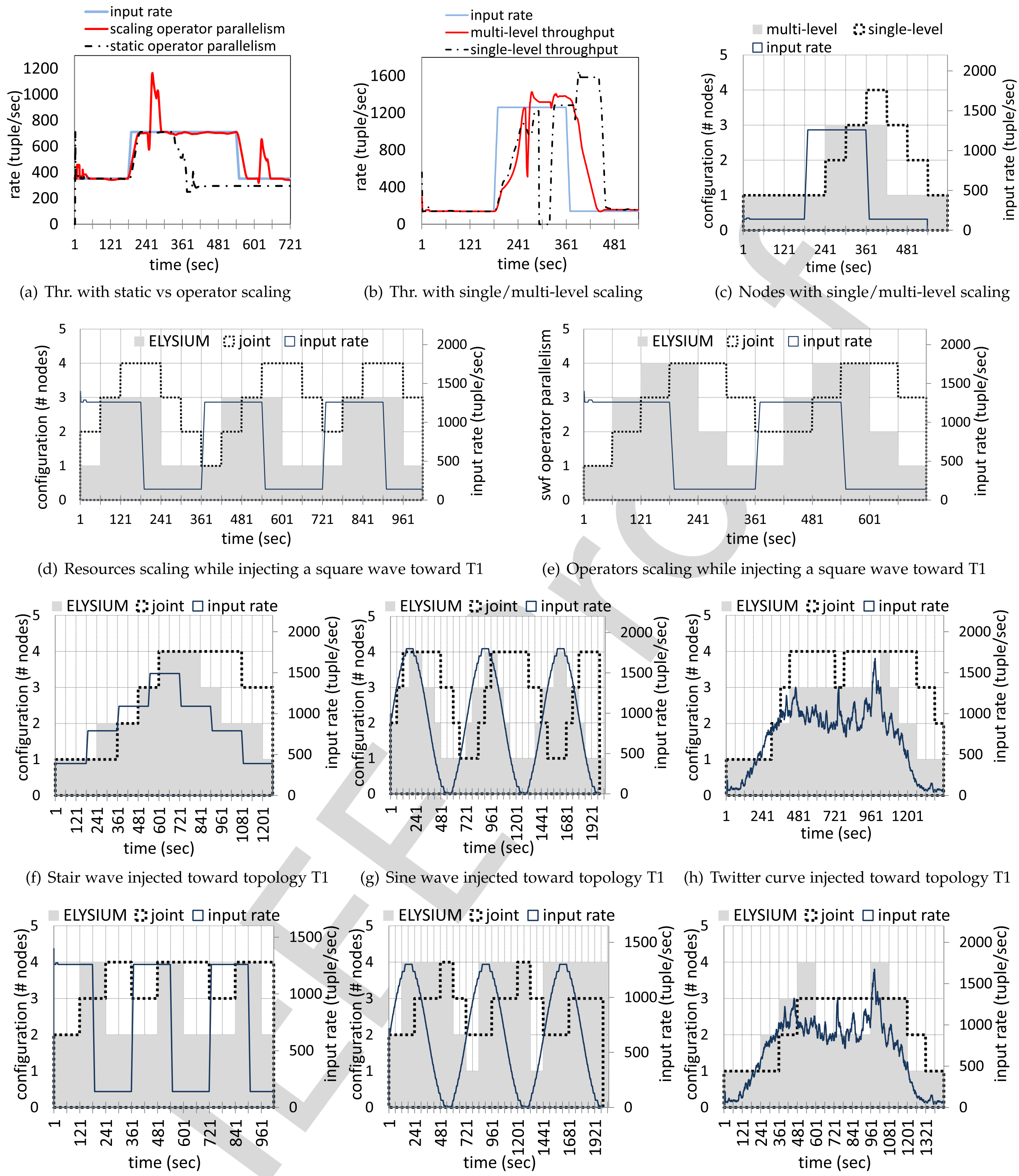
10. Note that here we aim at comparing symbiotic versus joint approaches and not the systems themselves as they are widely different.

Topology / Dataset	Figures	Scaling Type	Resources	Operator Parallelism	Throughput Degradation	Nodes Saved %
T1 stair dataset	--	--	4	4	0.05	0
	--	only operators	2	scalable	1.47	50
	--	only operators	4	scalable	0.98	0
	--	only resources	scalable	2	1.43	19
	--	only resources	scalable	4	0.97	33
	10(f) 13(a)	joint	scalable		0.97	32
		ELYSIUM (R)	scalable		0.81	43
		ELYSIUM (P)	scalable		0.81	43
T1 step	10(a-c)	joint	scalable		0.78	50
		ELYSIUM (R)	scalable		0.59	58
T1 square	10(d-e)	joint	scalable		1.49	25
	11(a)	ELYSIUM (R)	scalable		1.7	35
	13(b)	ELYSIUM (P)	scalable		1.2	35
T1 sine	10(g)	joint	scalable		1.25	25
	11(b)	ELYSIUM (R)	scalable		1.01	47
T1 twitter	10(h)	joint	scalable		1.25	24
	11(c)	ELYSIUM (R)	scalable		0.86	45
	13(c)	ELYSIUM (P)	scalable		0.9	45
T2 square	10(i)	joint	scalable		0.99	13
	11(d)	ELYSIUM (R)	scalable		1.03	33
T2 sine	10(j)	joint	scalable		0.63	30
	11(e)	ELYSIUM (R)	scalable		0.17	22
T2 twitter	10(k)	joint	scalable		0.49	48
	11(f)	ELYSIUM (R)	scalable		0.48	50
T1 step T2 stair	12(a-c)	ELYSIUM (R)	scalable		0.80	17
T1 sine T2 sine	12(d-f)	ELYSIUM (R)	scalable		0.46	21

Fig. 9. Evaluation summary. The second column indicates the reference to the figure in this paper; the third column refers to the scaling strategy, where ELYSIUM can be set either reactive (R) or proactive (P).

the joint scaling regarding its possibility to scale resources in/out of a single unit (*single-level*). On the contrary, the proposed scaling approach leverages the Estimator to choose the proper number of worker nodes to use (*multi-level*). For this test, we used a step-shaped input load over T1, as shown in Figs. 10b and 10c, where throughput and used worker nodes comparisons are shown, respectively. These figures clearly show that both the scaling strategies suffer the input load peak at the beginning. While the symbiotic scaling resumes sustaining the input load after 80 to 90 seconds from the peak, the joint scaling makes the application throughput break down for a few tens of seconds, then manages to keep up after about two and half minutes from the input load peak. When the input load decreases at minute 6, the symbiotic approach scales in the resources after one minute, and the throughput gradually decreases to match the input load. During this settlement period, the throughput is larger than the input load because of the reconfiguration overhead. The joint scaling performs worse as it requires more reconfigurations to reach the correct one, so it pays a much larger overhead, while ELYSIUM provisions the right amount of resources with a single reconfiguration. Indeed, joint scaling takes a few tens of seconds longer than ELYSIUM to generate a throughput equal to the input load. Besides providing smaller throughput degradation (0.59 against 0.78), the symbiotic approach allows to save resources as show in Fig. 10c (see also Fig. 9, where an overview of all the tests executed is reported). Throughput degradation of symbiotic scaling is slightly better than that obtained by reconfiguring every minute (see Fig. 6c).

To provide a better understanding on the way operators and resources are scaled symbiotically, we show the results



(a) Thr. with static vs operator scaling (b) Thr. with single/multi-level scaling (c) Nodes with single/multi-level scaling

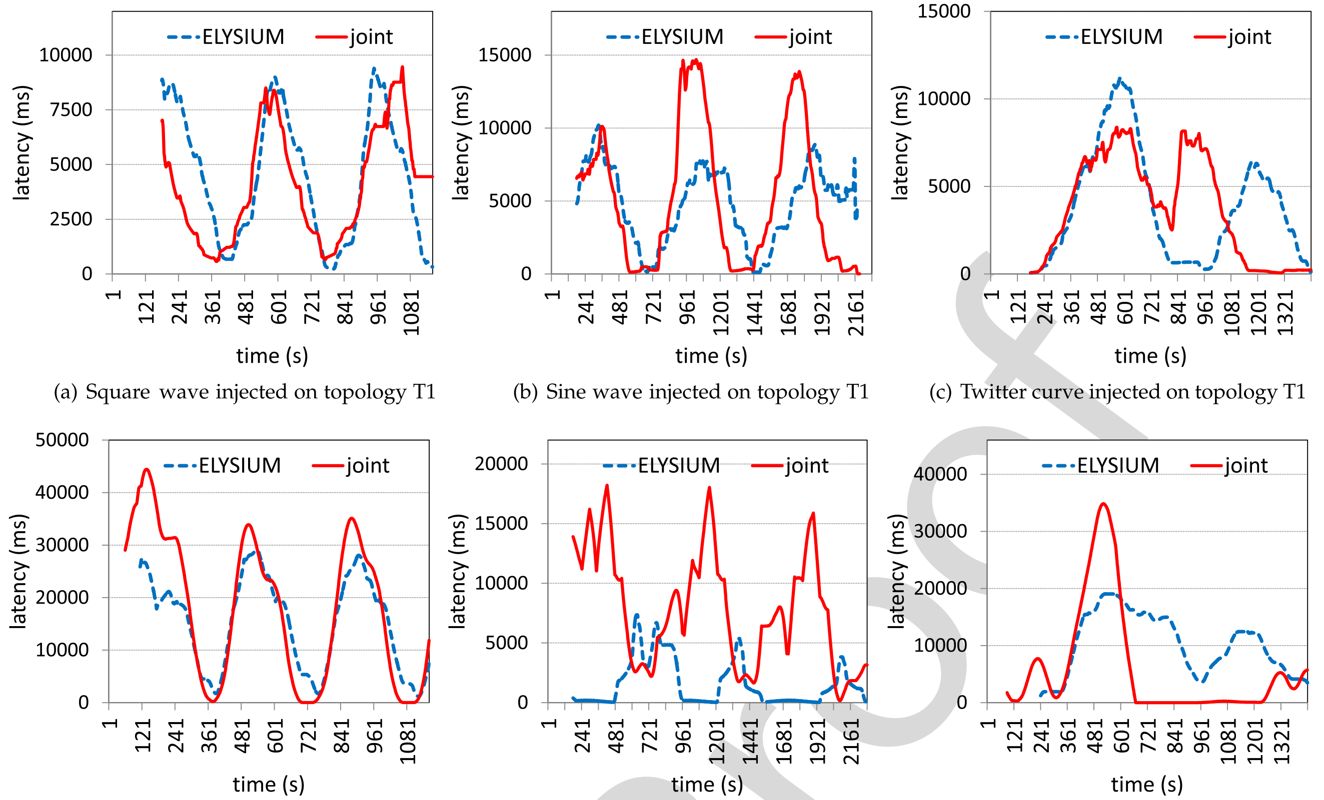
(d) Resources scaling while injecting a square wave toward T1 (e) Operators scaling while injecting a square wave toward T1

(f) Stair wave injected toward topology T1 (g) Sine wave injected toward topology T1 (h) Twitter curve injected toward topology T1

(i) Square wave injected toward topology T2 (j) Sine wave injected toward topology T2 (k) Twitter curve injected toward topology T2

Fig. 10. Comparison between joint scaling and symbiotic scaling (ELYSIUM) while injecting different traces toward T1 and T2.

of an experiment that used a square wave input load over T1. Figs. 10d and 10e present respectively how the number of worker nodes and the parallelism of T1's StopWordFilter (the most significant operator in T1) change over time, for joint and symbiotic scaling (i.e., ELYSIUM). With the symbiotic approach it is possible to adapt faster than with the joint one, for what regards both the resources and the operator parallelism. The throughput degradation is similar but larger with ELYSIUM (1.7 versus 1.49) as a lower number of nodes is used compared to the joint approach, which instead over-provisions the topology and does not experience overloading. Indeed, nodes saved are 25 percent for joint scaling and 35 percent for ELYSIUM. We experienced similar results with T2 (Fig. 10i): slightly larger throughput degradation (1.03 versus 0.99), but more nodes saved (33 percent versus 13 percent).



(a) Square wave injected on topology T1 (b) Sine wave injected on topology T1 (c) Twitter curve injected on topology T1

(d) Square wave injected on topology T2 (e) Sine wave injected on topology T2 (f) Twitter curve injected on topology T2

Fig. 11. Comparison on latency between ELYSIUM and joint scaling while injecting different input load curves toward the two topologies.

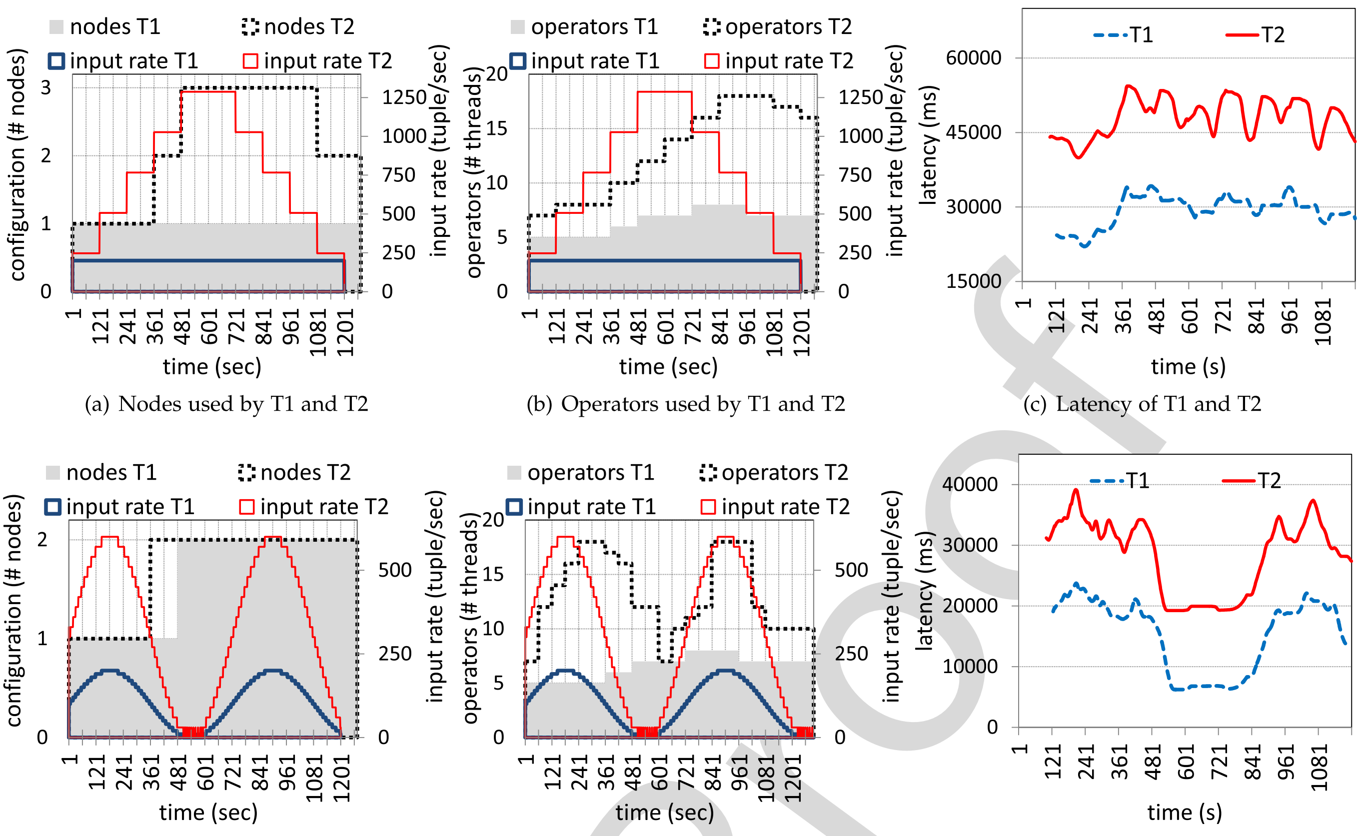
To complete the comparison between joint scaling and ELYSIUM, we show how they differ in used worker nodes over time for other distinct input load curves. Fig. 10f shows the comparison with a stair-shaped input load over T1. Globally the throughput degradation is smaller for ELYSIUM (0.81 versus 0.97 of the joint), while saving more resources (43 percent versus 32 percent with joint scaling).

Similar results are reported in Figs. 10g and 10j for a sinusoidal input load over T1 and T2 respectively. In both cases, ELYSIUM provides a lower throughput degradation ($1.01/0.17$ versus $1.25/0.63$), while they differently save nodes ($47/22$ percent versus $25/30$ percent). Finally, Figs. 10h and 10k show the results with the Twitter trace over T1 and T2. In both tests ELYSIUM has a lower throughput degradation ($0.86/0.48$ versus $1.25/0.49$) and more nodes saved ($45/52$ percent versus $24/48$ percent).

The performance of ELYSIUM compared to joint scaling in terms of latency are shown in Fig. 11. Specifically, it is possible to see that the trend of the latency of both approaches when injecting a square curve is similar for both T1 and T2 (Figs. 11a and 11d). For the twitter trace, instead, ELYSIUM and joint scaling, for both T1 and T2, differ in specific periods as they use a different policy to scale (Figs. 11c and 11f). The main differences are appreciable from tests using a sinusoidal wave as input load, as shown in Figs. 11b and 11e where ELYSIUM outperforms the joint approach, showing pretty smaller latency values.

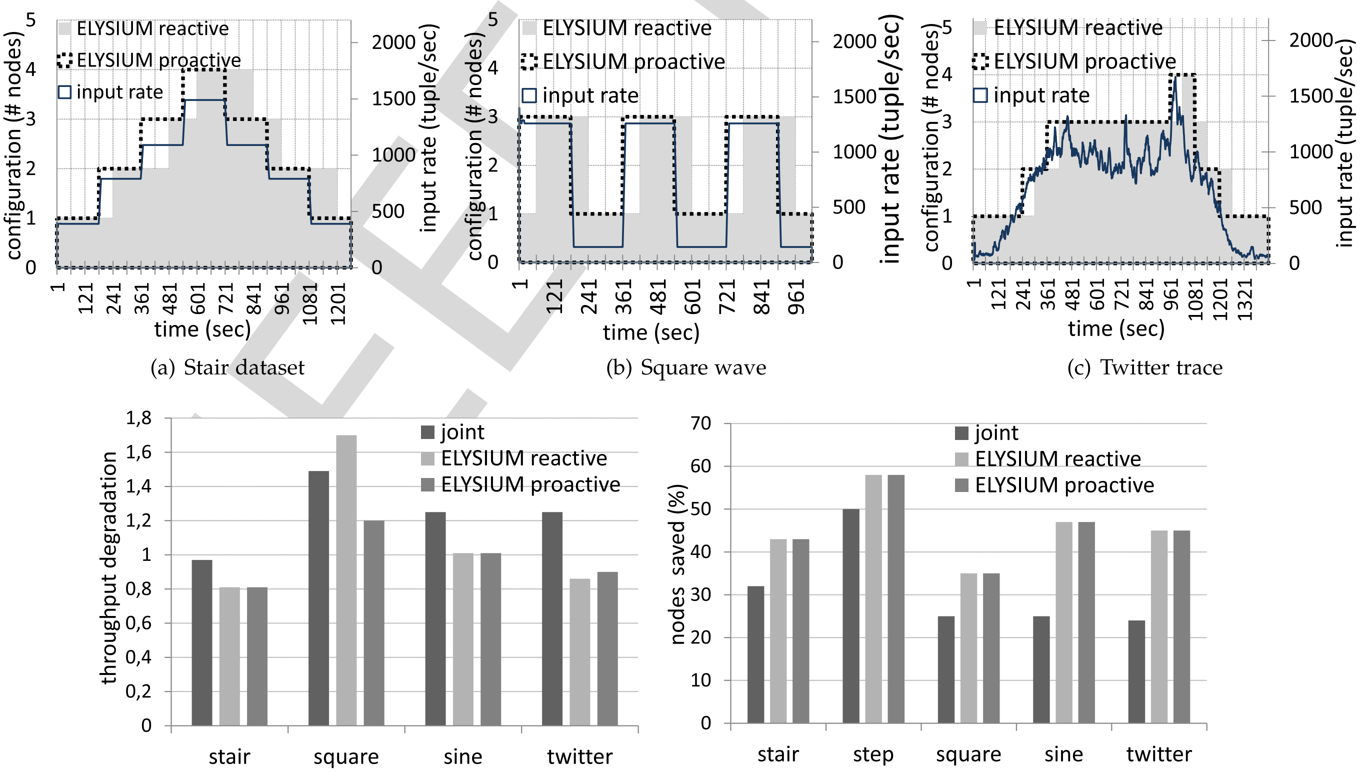
Managing Multiple Applications. To test the ability of ELYSIUM to scale in presence of multiple applications, we ran two tests with both T1 and T2 deployed in the same cluster. In the first test we injected a step input load of $200 req/s$ in T1 and a stair wave in T2. From Figs. 12a and 12b it is possible to see how the two topologies require different number of nodes as well as different number of operators. Specifically, the nodes of T2 change over time as it has to handle a larger workload, while T1 always uses a single node. Nevertheless, T1 frequently requires an increases of its operator parallelism, as the overhead due to reconfigurations leads to a larger usage of some operators. Conversely, in a second test we injected a sinusoidal input load in both T1 and T2 with different magnitudes. Figs. 12d and 12e show how T1 and T2 differently scale for nodes and operators. From Figs. 12c and 12f, it is possible to see how the latency of both T1 and T2 is quite stable and, obviously, T2 has in both cases larger values as it handles a larger workload.

Proactive Symbiotic Scaling. ELYSIUM can be used in either reactive or proactive mode. Proactive scaling can help reducing the delay between when the reconfiguration occurs and when its effects are actually needed. Here, we implemented a technique that over-provisions resources for the next temporal horizon. By setting for instance a horizon of h minutes, the proactive system computes a prediction of the input load for each minute from $t+1$ to $t+h$, and uses the highest forecasted input load to estimate required resources. Figs. 13a, 13b, and 13c show the comparison on used worker nodes between reactive and proactive ELYSIUM, while injecting three different input load curves toward T1. Note that the strategies used by the proactive and reactive systems are exactly the same, the unique difference lying in the reconfiguration point that for the proactive version results closer to the real demand point.



(a) Nodes used by T1 and T2
(b) Operators used by T1 and T2
(c) Latency of T1 and T2
(d) Nodes used by T1 and T2
(e) Operators used by T1 and T2 trace
(f) Latency of T1 and T2

Fig. 12. ELYSIUM handling two topologies with different workloads.



(a) Stair dataset
(b) Square wave
(c) Twitter trace
(d) Throughput degradation comparison
(e) Saved nodes comparison

Fig. 13. The figures above show the comparison on used worker nodes between reactive and proactive ELYSIUM according to specific input load curves toward T1. The figures below show the comparison on throughput degradation and nodes saved aggregates between joint and symbiotic reactive/proactive approaches, with different input load curves.

In terms of nodes saved, the differences are negligible (very few nodes), but available resources are used more efficiently. Figs. 13d and 13e shows the overall results of these comparisons in terms of throughput degradation and nodes saved. For the square wave input load (i.e., the most critical pattern for reactive ELYSIUM), the throughput degradation drops from 1.7 to 1.2 showing a notable improvement that clearly justifies the usage of a proactive approach.

Overall Result. The overall main results are: (i) ELYSIUM always outperforms the joint scaling approach in term of saved resources, (ii) ELYSIUM is anyway able to sustain the same workload, often with a lower throughput degradation and lower latencies due to its ability to scale more units per time and (iii) the proactive version of ELYSIUM can reduce the impact of the reconfiguration overhead and further improve performance of the symbiotic approach.

6 Related Work

Elastic scaling is a well known problem in the area of cloud-based platforms, where a lot of efforts have been devoted to the identification of efficient scalability policies [24] as well as metrics and benchmark methodologies [25]. How to scale SPSs has been extensively studied and analyzed from an application-level perspective by Hirzel et al. in [4]. Most of the works that tackle this problem at the application level [2], [26], [27] assume that a fixed amount of computing resources are available, and then strive to define the best allocation of operators to such resources. From this point of view, ELYSIUM works on a fully orthogonal direction, as we assume that a possibly infinite amount of resources is available (like in a cloud computing scenario), but aims at consuming the minimum amount needed to run the SPS with the goal of being cost efficient. Differently from previous solutions, ELYSIUM manages operator and resource scaling in a symbiotic fashion, deciding which one to apply or whether to apply both depending on the specific scenario.

A large fraction of solutions available in the literature scale one resource at a time. A notable exception is represented by [2], where the authors propose *rapid scaling* i.e., a solution that reduces the number of iterations needed to reach the target configuration. ELYSIUM further improves along this same direction by providing a solution that removes/provisions multiple resources in a single scale-in/out action on the basis of the resource usage estimated from either current or predicted input load, depending on whether the reactive or proactive mode is enabled.

Heinze et al. in [5] presented a solution to perform horizontal scaling according to the workload pattern evolution and by optimizing a cost function. Such prototype extends the FUGU stream processing system [6], where the authors compared threshold-based techniques with reinforcement learning techniques as defined in [24]. Furthermore, in [28] they proposed a latency aware solution. ELYSIUM, with respect to the previous solutions, is able to predict large load fluctuations and thus allows scale-in/out of multiple instances and resources at the same time.

While the majority of the works are reactive and based on thresholds, i.e., they act after an overload/underload detection, Ishii et al. in [29] proposed a proactive solution to move part of the computation to the cloud when the local cluster becomes unable to handle the predicted workload. The proactive model we propose is more fine-grained thanks to a resource estimator that allows to accurately compute the expected resource consumption given an input load and a configuration.

Recently, some efforts have also been spent to consider together problems related to elasticity and fault tolerance. In [23], the authors considered the problem of scaling stateful operators deployed over a large cloud infrastructure. In cloud environment, in fact, failures are common and managing replicated operators in presence of crash and recoveries introduces additional overheads with respect to those imposed by automatic scaling. The scaling strategy they propose, contrarily from us they (i) used a joint approach hence the detection of an overloaded operator leads to the allocation of new resources, and (ii) scale one unit per time.

7 Conclusion

In this paper we presented ELYSIUM, an elastic scaling framework for SPS. ELYSIUM first uses a Profiler to learn the behavior of a SPS application, then predictively scales the system symbiotically along two distinct dimensions: operator parallelism and resources. Through an experimental evaluation based on a real prototype integrated in Storm, we showed how ELYSIUM outperforms a joint scaling strategy, while always saving more resources.

As future directions, we aim to design a more complete model for resource estimation including memory and bandwidth so as to integrate shedding techniques to tackle bandwidth bottlenecks. Considering other optimization techniques proposed in [4], we also plan to integrate further solutions (e.g., smart operator placement to improve load balancing among resources [30]) and scaling according to predefined SLAs, such as maximum latency, as we similarly did in [31], in a more complete framework.

References

[1] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak, "Cloud-based data stream processing," in *Proc. ACM Int. Conf. Distrib. Event-Based Syst.*, 2014, pp. 238–245.

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

[3] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. Hoboken, NJ, USA: Wiley, 2004.

[4] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Comput. Survey*, vol. 46, 2014, Art. no. 46.

[5] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online parameter optimization for elastic data stream processing," in *Proc. ACM Symp. Cloud Comput.*, 2015, pp. 276–287.

[6] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *Proc. ACM Int. Conf. Distrib. Event-Based Syst.*, 2014, pp. 318–321.

[7] G. Jacques-Silva, Z. Kalbarczyk, B. Gedik, H. Andrade, K.-L. Wu, and R. K. Iyer, "Modeling stream processing applications for dependability evaluation," in *Proc. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2011, pp. 430–441.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas, "Operator scheduling in data stream systems," *Proc. VLDB Endowment*, vol. 13, no. 4, pp. 333–353, 2004.

[9] S. Schneider, M. Hirzel, and B. Gedik, "Tutorial: Stream processing optimizations," in *Proc. ACM Int. Conf. Distrib. Event-Based Syst.*, 2013, pp. 249–258.

[10] Apache, "Storm," (2011). [Online]. Available: http://storm.apache.org

[11] Apache, "Flink," (2015). [Online]. Available: https://flink.apache.org/

[12] N. Rivetti, E. Anceaume, Y. Busnel, L. Querzoni, and B. Sericola, "Online scheduling for shuffle grouping in distributed stream processing systems," in *Proc. ACM/IFIP/USENIX Middleware Conf.*, 2016, Art. no. 11.

[13] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola, "Efficient key grouping for near-optimal load balancing in stream processing systems," in *Proc. 9th ACM Int. Conf. Distrib. Event-Based Syst.*, Jun. 2015, pp. 80–91. [Online]. Available: http://www.dis.uniroma1.it/~midlab

[14] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini, "The power of both choices: Practical load balancing for distributed stream processing engines," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 137–148.

[15] G. Zhang, B. E. Patuwo, and M. Y. Hu, "Forecasting with artificial neural networks: The state of the art," *Int. J. Forecasting*, vol. 14, no. 1, pp. 35–62, 1998.

[16] Apache, "Derby DB," (2004). [Online]. Available: https://db.apache.org/derby/

[17] Heaton Research, "Encog machine learning framework," (2013). [Online]. Available: http://www.heatonresearch.com/encog/

[18] JBoss, "HornetQ," (2009). [Online]. Available: http://hornetq.jboss.org/

[19] N. Marz, "Twitter rolling top-K words storm topology," (2013). [Online]. Available: https://storm.apache.org/javadoc/apidocs/storm/starter/

[20] M. Illecker, "SentiStorm," (2015). [Online]. Available: https://github.com/millecker/senti-storm

[21] C. J. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, no. 3/4, pp. 279–292, 1992.

[22] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," in *Proc. IEEE Int. Conf. Neural Netw.*, 1993, pp. 586–591.

[23] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 725–736.

[24] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Comput.*, vol. 12, no. 4, pp. 559–592, 2014.

[25] N. R. Herbst, S. Kounev, and R. H. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *Proc. Int. Conf. Autonomic Comput.*, 2013, pp. 23–27.

[26] E. Zeitler and T. Risch, "Massive scale-out of expensive continuous queries," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 1181–1188, 2011.

[27] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "StreamCloud: An elastic and scalable data streaming system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2351–2365, Dec. 2012.

[28] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proc. ACM Int. Conf. Distrib. Event-Based Syst.*, 2014, pp. 13–22.

[29] A. Ishii and T. Suzumura, "Elastic stream computing with clouds," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2011, pp. 195–202.

[30] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proc. 7th ACM Int. Conf. Distrib. Event-Based Syst.*, 2013, pp. 207–218.

[31] L. Aniello, S. Bonomi, F. Lombardi, A. Zelli, and R. Baldoni, "An architecture for automatic scaling of replicated services," in *Proc. Int. Conf. Netw. Syst.*, 2014, pp. 122–137.

Federico Lombardi received the MSc in Engineering degree in computer science from Sapienza University of Rome with a thesis about an architecture for automatic scaling replicated services. He is working toward the PhD degree at Sapienza University of Rome. He is working in the Research Center of Cyber Intelligence and Information Security, Department of Computer, Control, and Management Engineering "Antonio Ruberti". His research mainly copes with scalability and elasticity problems of distributed and cloud services, distributed storage, and stream processing systems. He also works on security topics such as failure prediction, fault detection, data integrity, and blockchain.

Leonardo Aniello received the PhD degree in engineering in computer science from the University of Rome. He is a research fellow at the Research Center of Cyber Intelligence and Information Security, Department of Computer and System Sciences "Antonio Ruberti", "La Sapienza" University of Rome. His research studies include several topics in the fields of Big Data in large-scale environments, distributed storages and distributed computation techniques, with focus on the aspects of cyber security (malware analysis, intrusion prevention/detection, anonymization, privacy), integrity (blockchain-based storage), fault-tolerance, scalability and performance. He is author of more than 20 papers about these topics, published on international conferences, workshops, journals, and books.

Silvia Bonomi is working toward the PhD degree in computer science at Sapienza University of Rome. She is member of the Research Center of Cyber Intelligence and Information Security (CIS) in Sapienza. She is doing research on various computer science fields including Byzantine fault-tolerance, dynamic distributed systems, intrusion detection systems, and event-based systems. In these research fields, she published several papers in peer reviewed scientific forums. She has been involved in several National and EU-funded project where she addressed problems related to dependability and security of complex distributed systems like smart environment or critical infrastructures.

Leonardo Querzoni is an assistant professor with Sapienza University of Rome. His research interests range from distributed systems to computer security and focus, in particular, on topics that include distributed stream processing, dependability and security in distributed systems, large scale and dynamic distributed systems, and publish/subscribe middleware services. He regularly serves in the technical program committees of conferences in the field of dependability and event-based systems like DSN and ACM DEBS. He was general chair for the 2014 edition of the OPODIS conference and has been appointed as PhD Symposium co-chair for the 2017 edition of the ACM DEBS conference.