A Stepwise Auto-Profiling Method for Performance Optimization of Streaming Applications

XUNYUN LIU, University of Melbourne, Australia
AMIR VAHID DASTJERDI, University of Melbourne, Australia
RODRIGO N. CALHEIROS, Western Sydney University, Australia
CHENHAO QU, University of Melbourne, Australia
RAJKUMAR BUYYA, University of Melbourne, Australia

Data stream management systems (DSMSs) are scalable, highly available, and fault-tolerant systems that aggregate and analyze real-time data in motion. To continuously perform analytics on the fly within the stream, state-of-the-art DSMSs host streaming applications as a set of inter-connected operators, with each operator encapsulating the semantic of a specific operation. For parallel execution on a particular platform, these operators need to be appropriately replicated in multiple instances that split and process the workload simultaneously. Because the way operators are partitioned affects the resulting performance of streaming applications, it is essential for DSMSs to have a method to compare different operators and make holistic replication decisions to avoid performance bottlenecks and resource wastage. To this end, we propose a stepwise profiling approach to optimize application performance on a given execution platform. It automatically scales distributed computations over streams based on application features and processing power of provisioned resources, and builds the relationship between provisioned resources and application performance metrics to evaluate the efficiency of the resulting configuration. Experimental results confirm that the proposed approach successfully fulfils its goals with minimal profiling overhead.

CCS Concepts: •**Information systems** → **Data streams; Stream management;** Database performance evaluation; •**Social and professional topics** → Quality assurance; •**Software and its engineering** → *Software performance;*

Additional Key Words and Phrases: Stream Processing, Data Stream Management Systems, Performance Optimization, Resource Management

ACM Reference Format:
Xunyun Liu, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, Chenhao Qu, and Rajkumar Buyya, 2017. A Stepwise auto-profiling method for performance optimization of streaming applications. *ACM Trans. Autonom. Adapt. Syst.* 0, 0, Article 1 (January 2017), 33 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Stream processing—a paradigm that supports leveraging data in motion for analytics—is rapidly emerging due to continuous generation of data and the need for their timely processing. Usually, stream processing is realized by a data stream management system (DSMS), a platform that supports on-line analysis of rapidly changing data streams while hiding the underlying complexity of implementation from applica-

Authors' addresses: Cloud Computing and Distributed Systems (CLOUDS) Lab, School of Computing and Information Systems, The University of Melbourne, Australia; emails: {xunyunl@student., amir.vahid@, cqu@student., rbuyya@}unimelb.edu.au; Rodrigo N. Calheiros: Locked Bag 1797 Penrith NSW 2751 Australia; email: R.Calheiros@westernsydney.edu.au.

DOI: 0000001.0000001

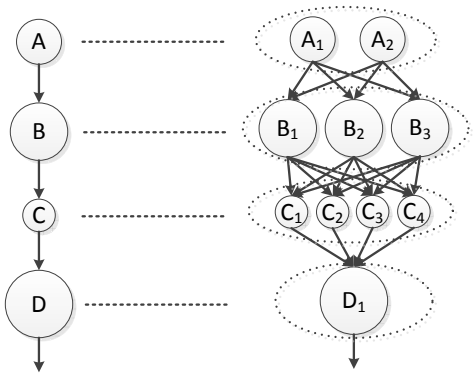


Fig. 1. The logical view of a streaming application on an operator-based DSMS.

tion developers. Currently, most state-of-the-art DSMSs such as Storm[1] and Samza[2] are data-driven and operator-based. In operator-based DSMSs, continuous operations on data are realized as logical operators standing on data streams, and the DSMS is responsible for the partition and distribution of resources among operators to achieve satisfactory performance [Aniello et al. 2013].

For a streaming application, resource partitioning largely depends on how operators are built and organised. To better explain this process, Figure 1 illustrates the logical view of a typical streaming application. The left part of Figure 1 shows that all the queries[3] have been translated into a pipeline of *operators* that perform transformations on the data. The relative size of operators represents the relative time complexity, with edges indicating data flows within the application. These operators and edges constitute the *application topology*, which can be modelled as a directed acyclic graph (DAG) that wires the operations together and denotes the sequence by which a single datum traverses the system.

When it comes to the implementation, the topology of a streaming application is further subdivided. To enable parallel processing, each operator may have several tasks scattering out over the platform. Each task is an operator instance that ingests a portion of operator input and executes the whole operator logic simultaneously. As the right side of Figure 1 shows, tasks of a downstream operator in the topology take the results of its precedents as input and continuously feed the successors with its output stream. Clearly, it is important for an operator to secure a sufficient number of parallel tasks, so that it could timely process its inbound load and avoid being the bottleneck that throttles the overall throughput of the system.

However, decision of the number of instances in each operator depends on the specific application deployment process, which involves provisioning resources from the underlying hardware infrastructure and determining how the logical representation is mapped to a physical point of view for real execution. The latter is known as a critical *transition* from logic notation to real implementation. Figure 2 shows an example of this transition process. Tasks are wrapped up by threads, which are usually considered as the minimum units of execution in terms of resource scheduling, then threads affiliated to several processes are deployed on the particular execution environment. It

[1]https://storm.apache.org/
[2]http://samza.apache.org
[3]By query, we mean formal statements of information needs that apply on continuous streams and that demand some computational capacity to be processed.

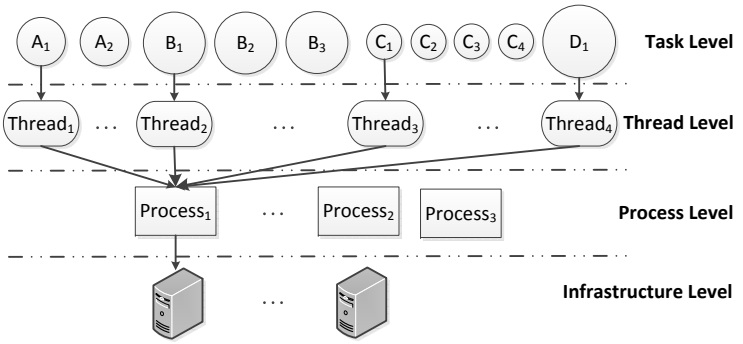


Fig. 2. The physical view of an example streaming application on an operator-based DSMS. After being wrapped up by threads and processes, the tasks of operators are finally deployed on several physical or virtual machines.

is non-trivial task to make optimal choices in such transition from logical to physical view because:

(1) Different operators can have diverse requirements on different types of resources (CPU, memory, network bandwidth, etc.).
(2) Changing the number of tasks for one operator may adversely affect the performance of other operators that are collocated in the same machine, causing unexpected bottleneck shift. Such kind of resource contention is hard to formally model.
(3) The transition is largely platform-dependent. Thus, without field testing, it is difficult to guarantee the effectiveness and efficiency of the transition decision.

Due to the difficulties stated above, the most common approach used to determine operator parallelism is to gradually measure the execution capacity of each operator and adjust the number of tasks according to the expertise of the developer. Obviously, this method involves a huge number of man-hours and may result in a suboptimal configuration. As existing research mainly focuses on the other side of the problem, which is scheduling threads on processes or arranging processes on machines [Cammert et al. 2007; Moakar et al. 2012; Aniello et al. 2013; Bellavista et al. 2014], the research question of automatically finding a proper and integral solution to this transition is largely overlooked.

Motivated by the goals of automation and enhanced developers' productivity, we design and implement a stepwise profiling framework that selectively evaluates several possible configurations, monitors feedbacks, and provides an entire solution to the transition. The objective of the proposed profiler is to determine the possible best performance[4] that the application can achieve in a particular execution environment. To the best of our knowledge, this is the first work using profiling to holistically probe the best configuration for an arbitrary streaming application, which is capable of striking a balance between the data source and data ingestion subsystems for it to achieve sustainable high performance. Specifically, our main **contributions** are threefold:

— Our profiling system automatically scales up the streaming application on a given platform. Such processing parallelization is achieved by profiling of both application features and processing power of provisioned resources. Therefore, developers are no longer required to provide parallel settings beforehand.

[4]Though the meaning of performance may vary under various definitions of QoS (Quality of Service), we refer to it as the ability to steadily handle an input stream of throughput T within an acceptable processing latency L. In this sense, higher T means better performance as long as the latency constraint is met.

— The profiling strategy is designed as a feed-back control loop that allows for self-adaptivity, scalability, and general applicability to a wide range of streaming applications, which is demonstrated in our experiments.
— Based on the result of profiling, the relationship between resource provisioning and performance metrics of application is built, enabling further evaluation of the efficiencies of candidate topologies that are implemented for the same streaming application.

2. MOTIVATION

The development cycle of a streaming application on an operator-based DSMS typically consists of two phases. The first phase consists in the logic development, where all the continuous queries or other data operations are implemented as logical operators working on data streams. The second phase consists in the application deployment, which mainly comprises a transition from logical to physical view. In this phase, the parallelism setting for each operator is determined and the decision on how tasks of operators are wrapped up and mapped to underlying resources is made, which are collectively referred to as a parallel *configuration*. Our primary motivation is to automate the transition and ensure that, in the resulting configuration, resources are properly partitioned among operators to enable better performance.

As mentioned above, optimization of the application deployment is a non-trivial process. Here are three fundamental prerequisites that a streaming application should meet before it comes into service.

Application scaling: Scaling up[5] is a critical process for a streaming application to use distributed resources. As scaling is both resource specific and topology dependent, there is no universal model able to provide a general solution. Therefore, the transition in the second phase has to be designed and performed by developers according to their own experiences, which causes additional development burden and may not yield efficient resource utilization. It becomes even more problematic when the underlying resource structure is configurable. State-of-the-art DSMSs are integrating elasticity into their implementation to enable resource consumptions customization with regard to fluctuating workloads. They support (1) dynamic resizing, e.g. DSMS can be scaled out by adding new machines, and (2) adjustable operator parallelization, which allows stateful and stateless operators to choose their number of tasks in order to suit different sizes of execution environment. However, applications running on an elastic DSMS do not have the ability to adapt their configuration to infrastructure changes, meaning that they are unable to automatically take advantage of newly added compute resources when the DSMS is scaled out, and may face severe resource contention due to excessive parallelization when the DSMS is scaled in. Our work fills in this gap by automating the scaling up process once the underlying system is updated, which complements efforts towards making DSMS scalable and elastic [Schneider et al. 2009; Schneider et al. 2012; Castro Fernandez et al. 2013; Gedik et al. 2014].

Besides, it is also desirable to quantitatively evaluate how efficient the transition is and automatically probe whether there is still room for improvements. However, due to the labor-intensive task of manual deployment, it is a common practice for developers to stop scaling up the application when a transition that meets the requirement of performance is found. Nevertheless, it may result in suboptimal resource utilization.

DAGs comparison: The topology of a streaming application is organised as a directed acyclic graph (DAG) of logical operators. However, the conversion of queries and operations on data streams into operators, which is performed in the first phase,

[5]Scaling up refers to further parallelizing the execution of logical operators to improve the resource utilization of a streaming application, whereas scaling out/in stands for adding or removing machines.

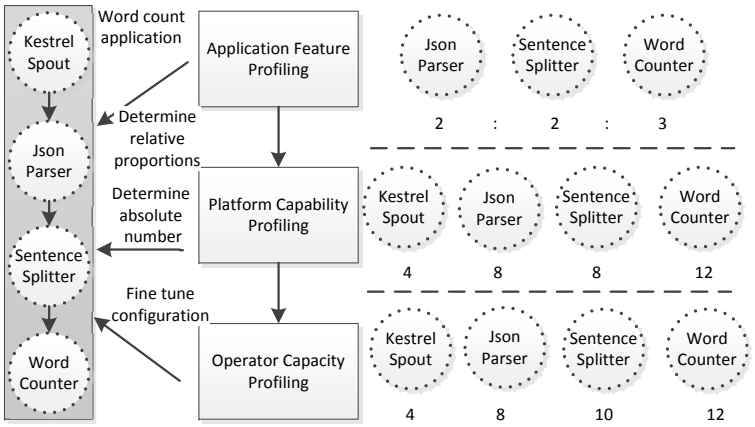


Fig. 3. Flowchart of stepwise profiling and a working demonstration on a word count application.

can be conducted in multiple ways, resulting in different topologies that are logically equivalent. It means that, although different types of DAGs are formed by operators, they take the same input stream and all produce correct answers. It is difficult but still necessary to determine which one is better with respect to their performances in a particular platform.

Resource requirement analysis: It is essential to know how many resources are needed to meet time constraints to handle the inbound stream. The answer depends on the volume of the input stream and the application resource needs per input data element. In the context of stream processing, the input stream may vary significantly in volume and speed and so does the amount of resource needed per element. Usually, application developers do not have control over the input data [Hummer et al. 2013], but tracking the latter could help them to guarantee real-time response with minimal resource consumption when the workload varies. Based on this, a rule-based auto-scaling approach could be proposed.

In this paper, we choose application profiling as an empirical and adaptive approach to fulfil the above targets. Compared to analytical models based on abstract modelling, profiling excels as it provides more reliable results via real experiments. Furthermore, by taking advantage of profiling, our method is generic enough to support different execution environments, including variations in characteristics of underlying resources, load balancing of DSMS, and the type of streaming application running on it. Besides, a recalibration mechanism has been introduced to ensure that the decision on parallel configuration is up-to-date. Therefore, possible changes to application and DSMS, as well as data-dependent variation affecting the execution time of data elements, will not compromise the accuracy of profiled knowledge.

3. STEPWISE PROFILING OVERVIEW

The profiling process works by selectively evaluating several possible configurations and finally choosing the one that shows the most promising performance potential, i.e. the one capable of processing more data streams per unit time while meeting the latency requirement.

Figure 3 describes the flowchart of our profiling approach and depicts how it applies to a word count application on Apache Storm. The topology of word count consists of four operators: the first operator, Kestrel Spout, pulls data from a queue server and generates a continuous stream of tweets as its output. The second operator, JSON

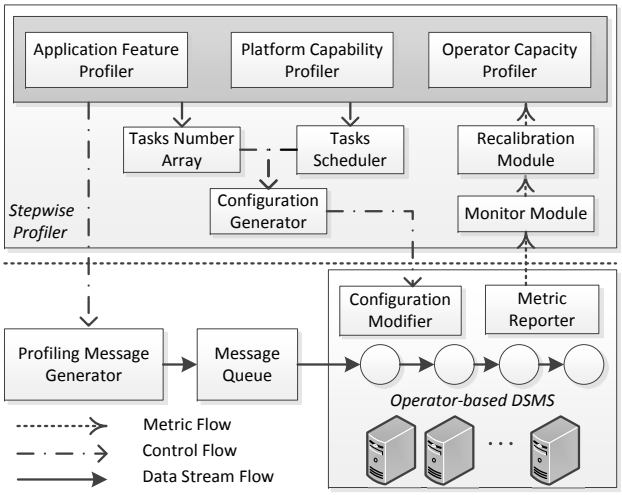


Fig. 4. The framework of stepwise profiling system. Components that constitute the stepwise profiler are presented in the top of the figure and the profiling environment is depicted in the bottom of the figure.

Parser, parses the stream and extracts the main message body. Next, the Sentence Splitter divides the main body of text into a collection of separate words, and finally the Word Counter is responsible for the final occurrence counting.

Regarding the profiling procedure, Application Feature Profiling (the first step) simulates the situation in which each task has adequate resources to conduct its data operation. It feeds the application with only a small size of input stream and aims to identify inherent application features that are not affected by the change of parallel configurations. On completion of this process, it determines the ratios of the numbers of tasks for the last three operators, which in this case is 2:2:3.

Platform Capability Profiling (the second step) stresses the platform with a high volume of input data to push it to its capability limit. At the end of this step, the actual number of tasks for each operator is determined.

Operator Capacity Profiling (the last step) makes necessary adjustment by monitoring the capacity of each operator. As our profiling model and measurement in the previous processes may have introduced some errors, this is the place where possible amendments are made.

The recalibration is essentially a repetition of the aforementioned profiling steps, triggered by performance degradation, detected via monitoring, when the resulted configuration is no longer suitable for the current system status.

4. STEPWISE PROFILING DESIGN

Figure 4 illustrates the architecture of our stepwise profiler (top half of the figure) and how it interacts with the operator-based DSMS in the profiling environment (bottom half of the figure).

The profiling environment consists of a profiling message generator, a message queue, and an operator-based DSMS. All the profiling input originates from the message generator, where real data collected from the production environment is sent to the message queue at a controllable speed. In the meantime, the message queue works as a data buffer to store possible backlogs when the DSMS cannot cope with the speed of data generation. The operator-based DSMS contains the primitive streaming application logic as well as supporting hardware resources.

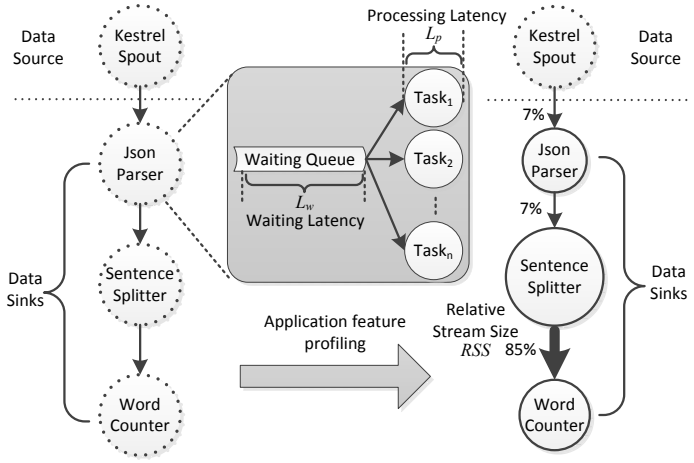


Fig. 5. An example of application feature profiling in operation.

Each single round of profiling is a feedback control process that corresponds to a MAPE (Monitoring, Analysis, Planning, Execution) loop, the approach of which is widely adopted in the field of autonomic computing to enable self-adaptivity [Kephart and Chess 2003].

The MAPE loop starts with the metric reporter running alongside the DSMS, which constantly collects current performance metrics from the evaluated streaming application. This information is then acquired by the monitor module and being organized as a set of window-based performance histories. The analysis phase is conducted by the three control units of our stepwise profiler as shown in the grey box of Figure 4, which are referred to as Application Feature Profiler, Platform Capability Profiler and Operator Capacity Profiler. These modules check the collected performance metrics according to their designated profiling strategies and make decisions on whether another round of profiling is needed. The MAPE loop proceeds to the planning phase if the stopping condition is not met. In this phase, the three control units make necessary amendment on operator parallelism and rely on the configuration generator to compose a viable deployment plan, which includes determining the speed of data generation for profiling, the number of tasks for each operator, and how these tasks are deployed on DSMS. In the last execution phase, the configuration modifier is responsible for applying changes and facilitating automation of application deployment.

The recalibration module also works in the analysis phase to check if the previously profiled configuration still suits the current system state. If not, it will plan for the next round profiling without using any prior knowledge.

4.1. Application Feature Profiling

As illustrated in Figure 5, the logical view of a streaming application is divided into two parts: *data source*, the operator that forms the initial stream by continuously pulling data from external sources, and *data sinks*, where inbound data is buffered into a waiting queue before being processed by one of the parallel tasks.

The application feature profiling aims to identify two invariant properties that an operator should maintain regardless of its parallel degree through the analysis of a data stream of a relative small size. The first property is the minimal processing latency $MinL_p$. As shown in Figure 5, the processing latency L_p is the time interval that a single datum would spend in a task for being processed, while $MinL_p$, illustrated

by the size of operator on the right side of Figure 5, denotes the minimum value that L_p can reach in this particular platform. The second property is the **R**elative **S**tream **S**ize (RSS), which indicates the relative data transmission intensity for the operator. The word "relative" means that the amount of data transmitted between operators is normalized with regard to the total sum to show the proportion among operators. As shown in the right of Figure 5, the width of the lines between the operators represents the size of data flow relative to other visible streams.

These two properties remain constant regardless of the parallel configuration changes for two reasons. Firstly, for a given operator, $MinL_p$ only depends on its processing logic and how long it takes for this logic to be executed in the platform. To measure $MinL_p$ it is important to assert that the profiling stream load is sufficiently small, so that each task of this operator obtains enough resources as it requires to process the workload. On the other hand, changes in the parallelism for an operator, such as adding new tasks for it, influence the waiting latency L_w rather than the processing latency L_p, because a single datum in the stream cannot be executed by multiple tasks concurrently. However, it is still possible that L_p increases due to improper configurations, e.g. congested tasks may cause severe resource contention that causes high processing latency.

Secondly, the relative size of the data stream is a reflection of the operator type, which also makes it parallel-configuration irrelevant. More specifically, a given operator could be categorized into one of three types based on the correlation between its input stream and output stream, as presented in Table I. S_i and S_o denote the relative size of input/output stream, respectively.

Table I. Categorization of operators based on its relative input/output stream size (selectivity).

Type of Correlation	Expression	Example Operators
Proportional	$S_o = C_{coef} * S_i$	Join, Function
Workload-Dependent	$S_o = f(workload) * S_i$	Split, Filter
Logic-Dependent	$S_o = g(logic)$	Top N, Aggregation

Proportional operators continuously work on one or more input streams and emit results based on each piece of input, which means that the size of the output stream is linear to the size of input stream, with C_{coef} as a constant that represents the linear coefficient. Examples for this category include stream joins and function operators. In the case of word count, the JSON Parser belongs to this category because its output size depends only on the particular input, and changes in the number of tasks do not affect the output size.

In the case of *workload-dependent operators*, the relative size of the output stream is not only decided by the size of input stream, but also it is influenced by the inherent property of the workload. For example, different tweets may have different number of countable words, making the size of output stream fluctuate even when the size of the input stream is stable. But in the case of Figure 5, it is observed that, on average, the size of output stream for Sentence Splitter becomes 12 times larger than the input streams, which means that the application profiling helps in identifying what the value of $f(workload)$ would be when subject to a production input. Obviously, this correlation is also not affected by changes in the number of tasks available for the Sentence Splitter.

There are also *logic-dependent* operators whose output streams solely depend on the processing logic. Some common examples include the Top N operator, which compares and emits the most popular items based on their occurrences, and stream aggregation

operator, which aggregates the input stream or regularly performs batch operations. The Word Counter operator in Figure 5 is used for aggregation and thus is an example of a logic-dependent operator.

Whenever an operator becomes a bottleneck, the DSMS has to throttle the upstream and downstream operators to maintain the system stability. This leads to the observation that the streaming application can be well-approximated with an intuitive queueing network of data flow, which runs on a computational system of unknown capability where contention affects all tasks in the same way. The latter assumption may not always hold true during the runtime, but it is reasonable for us to depict the relative parallelism requirement for each operator.

In light of this, more resources (in this context, more tasks) should be allocated to operators with relative larger input data streams and higher processing latency to prevent bottlenecks in the first place. After profiling the minimal processing latency and relative stream size for each operator, Algorithm 1 is proposed to provide an initial estimation on the number of tasks that an operator should incorporate considering its complexity and the amount of stream load it processes.

In summary, Algorithm 1 determines the parallelism ratio of each operator based on its calculated task load. The task load $TaskLoad_i$ of operator i is formulated as the product of its minimal processing latency $MinL_{p_i}$ multiplied by the sum of its input stream sizes $\sum_k RSS_{k,i}$ (index k iterates over all the input streams of operator i). After the algorithm finishes traversing all the operators, each element in the resulted array $\overrightarrow{R}$ is updated with a parallelism ratio relative to the weight of its task load (line 23). Note that in both line 16 and 23, the index s of $\max_{\forall s} TaskLoad_s$ iterates through all the operators in the topology.

However, there are two exceptions to this general rule. Some operators are logically *non-parallel*, which means that they can have only one task and thus are more likely to restrict the scalability of the whole system. Our algorithm takes these operators into consideration by fixing their parallelism degrees to 1 and quantitatively calculating the restriction they pose to the total size of stream flows ($TotalFlow$). $TotalFlow$ intuitively estimates the maximum sum of throughput generated by each operator. Based on this, Algorithm 1 decides the parallelism ratio or degree for other parallel operators to be able to handle the data stream of its share.

Secondly, some operators are *non-scalable* as they are dominated by logic-dependent operators, which means that there exists a logic-dependent operator in every path that connects this operator to the data source. Recalling that the size of output stream for a logic-operator is irrelevant to the input size, we can reasonably infer, in this case, that the size of input stream would be constant even if the system throughput increases. Therefore, instead of assigning parallelism ratios to them in $\overrightarrow{R}$, Algorithm 1 calculates the initial parallelism degrees for these operators in $\overrightarrow{P}$ based on $TotalFlow$ and task load, indicating that their parallelism degrees are not to be proportionally scaled in the next step.

The output $\overrightarrow{R}$ of Algorithm 1 only provides an array of decimals. However, the next step requires an array of integers, as this array represents the ratio of number of tasks. In the decimal to integer conversion, precision is not the primary concern since the results may be subject to measurement errors introduced by the profiling process. Therefore, the chosen value in the resulted integer array is reduced if possible, in such a way that it still roughly depicts the basic proportions. For this purpose, a parameter of unit task load called $slice$ is introduced to convert all the decimals in $\overrightarrow{R}$ into integers according to Equation 1.

ALGORITHM 1: Calculate the relative ratio or number of tasks for each operator.

Input: $MinL_{p_i}$: minimum processing latency of operator i
Input: $RSS_{i,j}$: relative stream size between consecutive operators i and j
Output: $\overrightarrow{R}$: parallelism ratio array of parallel operators, in which R_i corresponds to operator i
Output: $\overrightarrow{P}$: parallelism degree array of non-parallel and non-scalable operators, in which P_j corresponds to operator j

1 Initialize each element of $\overrightarrow{R}$ to 1;
2 $TotalFlow \leftarrow \infty$;
3 Identify all the operators that are dominated by logic-dependent operator, label them as Non-Scalable;
4 **foreach** *Operator* i **do**
5 **if** i *is Non-Parallel* **then**
6 $P_i \leftarrow 1$;
7 $TotalFlow \leftarrow \min(TotalFlow, \frac{1}{Min_{L_{p_i}} * \sum_k RSS_{k,i}})$;
8 **end**
9 **else**
  `/* Calculate TaskLoad for parallel operator` i `*/`
10 $TaskLoad_i \leftarrow MinL_{p_i} * \sum_k RSS_{k,i}$;
11 **end**
12 **end**
13 **foreach** *Parallel Operator* i **do**
14 **if** i *is Non-Scalable* **then**
15 **if** $TotalFlow = \infty$ **then**
16 $P_i \leftarrow \lceil \frac{TaskLoad_i}{\min_{\forall s} TaskLoad_s} \rceil$;
17 **end**
18 **else**
19 $P_i \leftarrow \lceil TaskLoad_i * TotalFlow \rceil$;
20 **end**
21 **end**
22 **else**
23 $R_i \leftarrow \frac{TaskLoad_i}{\max_{\forall s} TaskLoad_s}$;
24 **end**
25 **end**
26 **return** $\overrightarrow{R}$, $\overrightarrow{P}$;

$$R_i \leftarrow \lceil \frac{R_i}{slice} \rceil \quad slice \in (0, 1] \tag{1}$$

The value of $slice$ should be tailored to the specific streaming application. Our rule of thumb is to try small values (0.1, 0.2, etc.) and select the one that minimizes the profiling effort in the next step. Section 6.4 will shed more light on the parameter selection with real experiments.

It is also worth mentioning that in line 3 we omit the process of identifying dominance relationship for the sake of simplicity. Actually, there are some breadth-first searches starting from each logic-dependent operator to examine which operators are affected logic-dependent successors. In summary, the algorithm sequentially evaluates the operator located at the head of queue with regard to the status of its predecessors (each operator maintains a HashSet of all its status-undetermined predecessors for quick location and removal). If an operator has all its predecessors marked as ei-

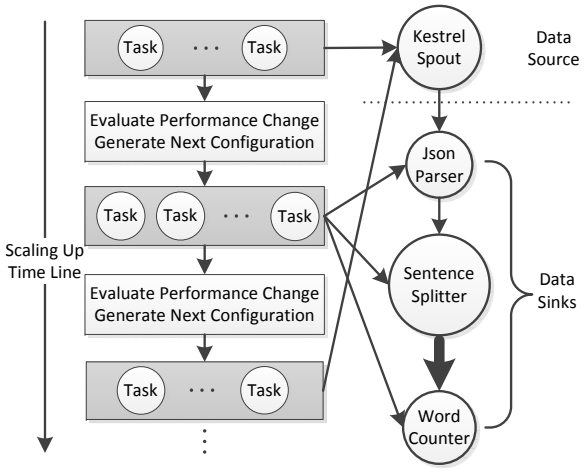


Fig. 6. An example of platform capability profiling in operation.

ther logic-dependent or already dominated, i.e. its HashSet of status-undetermined predecessors is emptied while this operator dequeues, it then should be identified as dominated and its successors are added to the tail of queue for further evaluation.

Algorithm 1 also has a computational complexity of $\mathcal{O}(n)$ with the worst case being $\mathcal{O}(n * (d^-_{avg} + 2))$, in which n is the number of operators and d^-_{avg} is the average vertex in-degree in the topology graph. The most time-consuming step lies in line 3 as each operator in the topology may be repeatedly visited, at most, its in-degree times to determine whether it has been dominated by logic-dependent operators or not. Besides line 3, the algorithm body traverses the topology graph only twice and all the required input can be collected with simply one round of profiling.

4.2. Platform Capability Profiling

Unlike the previous step which requires only a small data stream to probe application features, the platform capability profiling requires the message generator to produce a continuous data stream that is large enough to stress the streaming application. Given sufficient profiling data, the configuration of the application is changed through a trial-and-error process in order to determine the real capability of DSMS as well as its underlying infrastructure. The resulting configuration reveals a reasonable choice of resource partition in this platform where it is capable of handling a relatively large stream without violating the latency constraint[6].

As shown in Figure 6, each configuration trial is first evaluated in terms of system performance variation. Specifically, changes in throughput and latency are collected and reported to the monitor module, which can be used to identify if the new configuration improves the resource utilization. Configuration changes that have a negative impact on the system performance are discarded in this phase.

Based on the result of performance evaluation, the profiler applies changes to the configuration according to Algorithm 2 and generates a new one for the next round of profiling. The new configuration not only targets throughput improvement, but also aims at maintaining the balance between data source and data sinks. If it failed to

[6]Different applications may have different preferences with regard to their desirable performance. Though the final decision is left up to the application developer, as a default the profiler favours better throughput on the condition that the system still meets the pre-defined latency requirement.

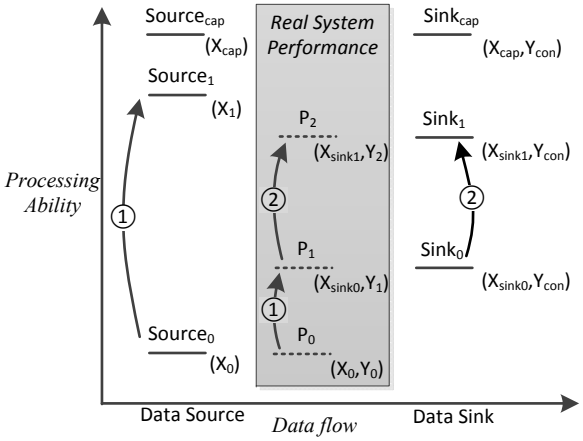


Fig. 7. Balancing data source and sinks in platform capability profiling. The vertical axis represents the processing ability ordering by throughput. The horizontal axis denotes the system components we concern about.

do so, an overly powerful data source may cause severe backlogs in data sinks and lead to a higher system latency, while an inefficient data source starves the following operators and encumbers the overall throughput. As a result, tasks are alternatively added to the data source or to data sinks to strengthen their processing abilities, and the search for the desired configuration leads the application to its performance limit where neither enhancing data source nor data sinks improves it.

Figure 7 illustrates the aforementioned scaling process with an example. At each edge of the figure, the solid short lines with labels indicate different configurations for data source and data sinks, while the dashed lines in the middle represent the overall system performance resulting from the configurations on each side. Different configurations lead to various processing potentials in terms of throughput and latency, which are shown in the right corner. Thus, short lines are all ranked by throughput with different heights in the figure, and the curves connecting them with numbers denote the potential throughput variances that resulted from different configuration changes by applying Algorithm 2. At first, the system is configured with $Source_0$ for data source and $Sink_0$ for data sinks, where $Source_0$ indicates that the data source is able to pull a data stream at a throughput of X_0, while $Sink_0$ with (X_{sink0}, Y_{con}) means that data sinks under this configuration are capable of dealing with a stream of size X_{sink0} within the user-specified latency constraint Y_{con}.

Given a particular platform, there should be a configuration that delivers the best performance in this profiling environment, indicating that the data source and data sinks have been properly coordinated. As denoted by the top two short lines in Figure 7, the data source $Source_{cap}$ and data sink $Sink_{cap}$ represent such ideal configuration that the profiler aims to achieve at the end of its operation. Initially, the data source $Source_0$ is less powerful than the data sink $Sink_0$. Thus, the system performance P_0 is confined at (X_0, Y_0), where X_0 is decided by $Source_0$ and $Y_0 < Y_{con}$ because the data sinks are underutilized. After detecting the latency margin, the profiler first scales up data source to $Source_1$, causing bottleneck shifts to data sinks $Sink_0$. This time the performance P_1 is limited at (X_{sink0}, Y_1) where $Y_1 > Y_{con}$ since that some backlogs have already been accumulated. Afterwards, the profiler enhances the ability of data sinks from $Sink_0$ to $Sink_1$, improving the performance from P_1 to $P_2 = (X_{sink1}, Y_2)$. However, as indicated by $Y_2 > Y_{con}$, the last modification is still inadequate and another sink scaling is needed in the next round.

ALGORITHM 2: Generation of a new configuration under the round-robin policy.

Input: T: Topology throughput
Input: α: Threshold for triggering reconfiguration
Input: T_b: Best throughput record
Input: $\overrightarrow{R}$: Ratio of parallelism of each operator

```
if Last change increased T then
    T_b ← T;
    if Latency constraint is not met then
        foreach operator do Add tasks according to R⃗ and the operator's position in the topology;
    end
    else Increase number of tasks of source by 1;
end
else if Last change did not significantly change T† then
    if Last change enhanced data sink then
        Increase number of tasks of source by 1;
    end
    else if Last change enhanced data source then
        foreach operator do Add tasks according to R⃗ and the operator's position in the topology;
    end
    else if Last change throttled the data source then
        if latency requirement has been met then
            Terminate the profiling;
        end
        else Increase throttle strength;
    end
end
else if Last change decreased T then
    if T < α * T_b then
        Return the system to the configuration where the best performance is observed;
        Throttle the data source;
    end
    else if Last change enhanced data sink then
        Increase number of tasks of source by 1;
    end
    else if Last change enhanced data source then
        foreach operator do Add tasks according to R⃗ and the operator's position in the topology;
    end
    else if Last change throttled the data source then
        Decrease throttle strength;
    end
end
```

†We adopt a Two Sample T-Test to determine whether a throughput change is significant or not, more details are given in Section 6.1 as a part of experiment setup.

Scaling up data source and data sinks works by adding new tasks to operators that need to be further parallelized, but it also raises a question of how to map the updated task graph to the underlying machines in order to achieve better resource utilization. This is also known as the task placement and scheduling problem. There are several policies available to decide the distribution of tasks across the platform, and certain applications may require a particular policy to suit a very specific need (e.g. assigning

a particular task to a particular machine due to licence restrictions). We therefore design the platform capability profiler to enable scheduling policies to be plugged in so that it can be used in conjunction with various scheduling heuristics with different optimization targets, such as minimizing inter-node communication [Aniello et al. 2013; Xu et al. 2014], reducing the average tuple processing time [Li et al. 2015], and being resource-aware to ensure the capability of each task to handle its task load [Peng et al. 2015]. Since the focus of this work does not lie in task placement and scheduling, we introduce our profiling approach in tandem with the widely adopted *round-robin policy*[7] and apply it in a platform with homogeneous computational resources for ease of presentation. The round-robin policy is particularly suitable for homogeneous platforms as tasks are evenly distributed among available machines to enable fault-tolerance and load-balancing.

Algorithm 2 shows the interplay between performance evaluation and configuration generation carried out by the profiler under the round-robin policy. Scaling data sources is a relatively lightweight operation: it only requires the number of tasks for the data source to be increased by 1, so that the application has one extra task pulling data from the message queue and thus increasing the input rate. However, decision about increasing the parallelism for a data sink operator depends on the type of operator and its position in the topology. For example, an operator should keep its number of tasks unchanged if it is a non-parallel operator, or if it is non-scalable dominated by logic-dependent operators as its input stream tends to be steady during the profiling process. As for other types of operators, $\overrightarrow{R}$ indicates the extent of enhancement for each operator.

Nevertheless, not every scaling effort, especially those applied for data sinks, can guarantee improvements. The reason why scaling data sinks is even more difficult than scaling data sources is that it has to exhaust current resources for additional computation and coordination. Therefore, to meet the latency constraint, our profiler performs a third operation on configuration, *source throttle*, which limits the size of input stream by controlling the amount of data that is allowed to sojourn in the system.

The complexity of computation required for configuration generation is constant. However, the profiling process that evaluates the effectiveness of a new configuration is relatively time-consuming since performance measurement must wait until the application is stabilized. To examine the number of profiling rounds required in the worst case, we regard Algorithm 2 as a search algorithm that explores a vectored value space, with each dimension confined by the actual parallelism degrees that can be seen in the ideal configuration. Given the fact that every three consecutive profiling efforts can increase the total number of used tasks at least by $\left\|\overrightarrow{R}\right\|_1$ through data sink enhancement (except for consecutive source throttles, which is rare), and that assigning excessive parallelism degree to an operator would harm the application performance, it is intuitive to deduce a conservative estimate that in the worst case there will be no more than $3 * \frac{nMax_p}{\left\|\overrightarrow{R}\right\|_1}$ rounds of profiling. In the expression, n is the number of operators in the topology, and Max_p represents the maximal parallelism degree among all the operators. However, Max_p is unknown before the actual profiling, but it can be approximated in practice by the number of threads able to run simultaneously in this particular platform (by multiplying the number of available cores by the number of thread(s) per core).

[7] http://grokbase.com/t/gg/storm-user/132fh5qyve/recommendations-for-setting-num-isolated-machines-num-workers-parallelism-hint

4.3. Operator Capacity Profiling

The previous step of profiling divided the streaming application in two parts (data sources and sinks), of which the parallel configurations of operators are collectively adjusted based on the overall performance of the system. Such coarse modifications may not be accurate enough to achieve the targeted configuration. Therefore, in the third step, profiling is carried out at operator level through the individual evaluation of performance of each operator. The goal of this step is to achieve finer granularity of performance tuning.

Operator capacity, which is formally defined in Equation 2, is used to quantitatively evaluate the degree of utilization of operators in data sinks. In the equation, $Operator\_latency$ is the average time that a single datum would spend in this operator over a specific time period. The length of such time period is called $Window\_size$ and the amount of data processed in this period is denoted by $Executed\_load$. Thus, capacity represents the percentage of the time in the observation time window that the operator spent executing inputs. The closer to 1 is this value, the more likely the operator is the bottleneck in our topology.

$$Capacity = \frac{Operator\_latency * Executed\_load}{Window\_size} \tag{2}$$

This step utilizes the same profiling environment used in the previous step. However, besides overall performance metrics such as throughput and latency, the profiler in this stage also collects the capacity information from each operator for fine-grained evaluation. The profiling strategy also resembles the previous one: the performance evaluation phase sheds light on the system status and the possible bottleneck, and the previous configuration change is revoked if it causes performance degradation. However, this process differs from the previous step in that it has only one operation, which is increasing the number of tasks by 1 for the operator that has the highest capacity and has not been enhanced nor revoked. If there is no performance improvement obtained from enhancing the operator with the highest capacity, the operator that has the second highest capacity is tested in the next round and so on.

There are two stopping conditions for the profiling. The first is when there are consecutive revocations observed indicating that recent scaling up efforts on candidate operators have failed. The second condition is when all the measured operator latencies approach the minimal processing latency $MinL_p$ by a factor k. We evaluate the effect of diverse values of k in the performance of the profiling later in Section 6.4.

4.4. Recalibration Mechanism

The application of the above three profiling steps yields a specific parallel configuration that builds a relation between provisioned resources and performance metrics. However, such relation is perceived to be volatile, since the performance under the same configuration may vary and the resulting configuration may need to be promptly modified due to the live changes that happen to the streaming application or platform. This section therefore discusses the recalibration mechanism, which repeats the profiling process when necessary to keep the configuration and operator profiling up-to-date with minimal adjustment cost.

In general, recalibration is triggered by any three types of changes: (i) resizing of DSMS, which leads to a new platform to be profiled after the infrastructure layer is dynamically scaled; (ii) re-deployment of the application, resulting from the alteration of application topology and the manipulation of some critical parameters that would greatly affect the application behaviour; and (iii) data-dependent variation, an uncon-

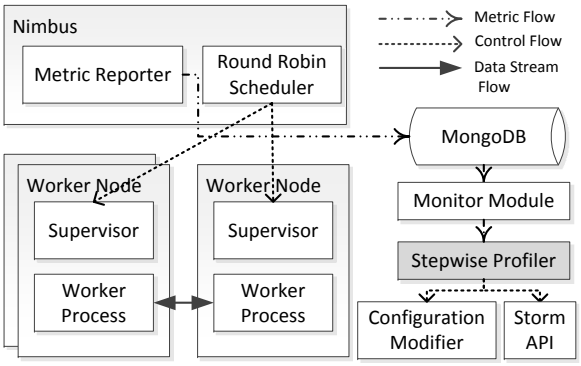


Fig. 8. The integration of the profiler prototype into Apache Storm.

trollable factor related to the characteristic of workload, causing performance to vary even if the configuration remains unchanged.

For the first two causes, the recalibration decision is straightforward. If the platform or application turn into a state that has not been previously profiled, all the profiling steps are repeated. However, the process is more challenging when it comes to dealing with data-dependent variation, as all the changes are independent of the platform and application. We can safely assume that all data elements within the same stream are of the same type, but the time and space complexity of execution may vary along with the changing element size or the density of information contained. The Sentence Splitter, in the word count topology, is a typical example to show the effect of data-dependent variation: its process latency and relative size of output stream depend on the average length of incoming tweets.

To deal with such variation, the recalibration mechanism requires a monitoring system to oversee the degradation of performance during runtime. It continuously monitors the length of the message queue, which indicates the capability of application to handle a certain level of throughput that previously demonstrated in the profiling phase, and the system latency, which examines if the user-specified latency constraint is still satisfied. In order to reduce the frequency of adjustment, we adopt a threshold-based method which postpones any recalibration action until the monitored values have exceeded the predefined threshold for a specific period of time.

5. SYSTEM IMPLEMENTATION

The architecture of the stepwise profiling system, as shown in Figure 4, consists of two main parts — the profiling environment and the stepwise profiler.

The setup of the profiling environment has been briefly introduced in Section 4. More specifically, the Profiling Message Generator[8] is a Java program that reads the workload file on demand in order to emit a particular size of profiling stream. The Message Queue connecting the streaming application to the Profiling Message Generator is built with Twitter Kestrel[9], a distributed queueing system that enables controllable message buffering. Developers could make use of the Thrift interface provided by Kestrel to retrieve the length of message queue and determine whether the streaming application has been overwhelmed by the profiling data.

As a specific DSMS was needed to enable the implementation and evaluation of the prototype, Apache Storm was chosen. This is because it is an open source software

[8]https://github.com/xunyunliu/MessageGenerator
[9]https://github.com/twitter-archive/kestrel

(and thus has all the source code available and detailed on-line documentation), and provides a built-in metric system and external configuration reader that facilitate the implementation of the stepwise profiler.

Figure 8 describes the integration of the profiler prototype into Apache Storm. The Stepwise Profiler module in the grey box are DSMS-independent, as it only interacts with other components of the architecture to make profiling decisions. Therefore, it is implemented as a stand-alone Java Program.

The other modules of the architecture interact directly with the DSMS to collect information or apply changes, thus the implementation of these modules are DSMS-dependent. The Metric Reporter component utilizes Storm's built-in metric system and the associated RESTful interface to collect performance information and publish results. Such metrics are then periodically sent to MongoDB[10] to facilitate tracking of performance changes. The Monitor Module[11], implemented as a Java program, inquires the MongoDB for the latest system status and reports it to the Stepwise Profiler for decision-making. In this process, some performance metrics, like complete latency (average time taken by a tuple and all its offspring to be completely processed by the topology), number of data emitted, and operator capacity can be directly used in the stepwise profiler. Some metrics, however, require certain post-processing in the Monitor Module. For example, there is no default definition for throughput among the built-in metrics. Thus, to avoid ambiguity, the Monitor Module calculates the overall throughput of a streaming application based on the observed number of acknowledgements or emitted data per unit of time, depending on whether the application adopts reliable message processing or not.

We also utilize some useful features of Storm in the process of generating and applying new configurations. Specifically, Storm not only supports reading parallelism setting of operators from an external configuration file, but also provides a command line tool (Storm API) to manage the topology with additional operational parameters. The stepwise profiler thus makes use of the Configuration Modifier component, which is implemented as a script file, to pack up all the profiling decisions in a deployment configuration file, and then invokes the command line tool to submit the application with the updated deployment scheme for the next round of profiling. The round-robin scheduler guarantees that tasks are evenly distributed across Worker Nodes and that load is equally distributed among machines.

Another aspect relating to implementation is the management of operator states during the scaling up process. We do not address dynamic stream rerouting and live state migration since the Configuration Modifier relies on the rebalance command to apply any deployment changes. This command, as a built-in Storm functionality, essentially pauses the application during the redeployment and then restarts it from scratch with the new configuration, following the so-called Pause and Resume protocol [Heinze et al. 2014a]. As our current prototype treats stateful operators the same way as stateless operators in terms of scaling, the management of operator states is not transparently handled by the profiling framework. Therefore, it is required that stateful operators preserve their states at the application level when the rebalance command is triggered, and these operators should also be initialized with the previous states when the application is restarted. However, there are some advanced mechanisms proposed in the literature that enable application-agnostic state management and interruption-free operator scaling, which is discussed in Section 7.

[10]https://www.mongodb.com/
[11]https://github.com/xunyunliu/MonitorModule

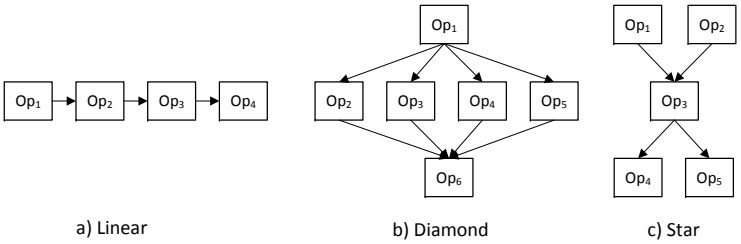


Fig. 9. Structure of the synthetic Micro-benchmark topologies.

6. PERFORMANCE EVALUATION

We have conducted three different sets of experiments to validate the effectiveness of our prototype.

(1) The first experiment presented in Section 6.2 evaluates whether the stepwise profiling effectively applies to a variety of streaming applications, and if it fulfils the other goals discussed in Section 2.
(2) The second one in Section 6.3 assesses the scalability of our prototype and showcases its runtime overhead under relative large test cases.
(3) The last experiment in Section 6.4 investigates the effect of different parameters on the profiler performance, based on which we suggest default preferences.

6.1. Experiment Setup

The experiment environment is set up on a private cloud running OpenStack. The environment consists of three IBM X3500 M4 machines, and each machine is equipped with 2 x Intel Xeon E5-2620 Processor (6 core@2.0GHz), 64 GB RAM and 2.1 TB HDD. The virtual cluster deployed on the physical environment is composed of a control machine, a ZooKeeper node and several processing nodes. The first two nodes are "m1.large" (4 VCPU and 8 GB RAM), while the rest of the processing nodes are "m1.medium" (2 VCPU and 4GB RAM per machine). The control machine host the Stepwise Profiler, Profiling Message Generator, and the Message Queue components of the architecture to avoid possible interference to the profiling result.

6.1.1. Test Applications. We adapt six streaming topologies as our evaluation applications[12]. These include three synthetic topologies (collectively referred to as Micro-benchmark) and three real-world streaming applications: Word Count (WC), Synthetic Word Count (SWC), and Twitter Sentiment Analysis (TSA). All applications are configured with acknowledgements enabled in order to track the complete latency, and they process the same type of workload to calculate comparable throughput. The profiling stream used for performance test is recursively generated from a single workload file, which contains 159,620 tweets in JSON format collected from 24/03/2014 to 14/04/2014. In addition, these applications are carefully tuned to avoid out-of-memory crash and other failures due to insufficient resource allocation, so that the only potential consequence of improper configuration is suboptimal performance, rather than abrupt termination of the application.

Micro-benchmark: the micro-benchmark topology is synthetically designed to evaluate how the stepwise profiler generalises to different topology structures. As shown in Figure 9, it covers three common structure patterns: *Linear*, *Diamond*, and *Star*,

[12]In the following section, we use application and topology interchangeably to refer to the streaming logic developed on Apache Storm.

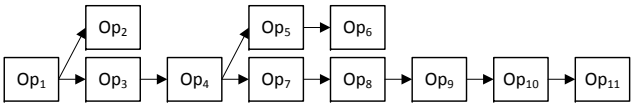


Fig. 10. Structure of the Twitter Sentiment Analysis (TSA) topology.

where an operator has (1) one-input-one-output, (2) multiple-outputs or multiple-inputs, and (3) multiple-inputs-multiple-outputs, respectively.

In addition, the *execute* method of each operator is implemented in three different patterns in order to reflect diverse time-space complexities. Some operators are (1) *CPU bound*, as they invoke a random number generation method Math.random() 10000 times for each tuple received. Some are (2) I/O *bound* with only a JSON parse operation applied on the incoming tuple, so that they spend more time on waiting for I/O operations rather than actually processing the current data. The rest of the operators are (3) *Sojourn time-bond*, which sleep for $5\ ms$ upon any tuple receipt. These operators are introduced to mimic the cases where an external service is requested to complete the tuple transaction. Consequently, they demand almost no CPU and memory usages on the execution platform, but still consume a substantial sojourn time for each incoming tuple to be processed.

All these operators have a function implemented to read the operator selectivity[13] from the external configuration file. Higher selectivity can be specified to produce saturated network usages, so that I/O bound operators could be overwhelmed by large internal streams.

Word Count and Synthetic Word Count: the Word Count topology is illustrated in Figure 3. The Synthetic Word Count topology adds a Waiting operator (a bolt[14] in Storm's terminology) between the Kestrel Spout and the JSON Parser, where each incoming tuple is kept for $1\ ms$ before being sent to the next operator. Therefore, WC and SWC are actually two different implementations for the same streaming application.

Twitter Sentiment Analysis: we adapted this topology from a mature open-source project hosted on Github[15] with the structure shown in Figure 10. It has 11 bolts constituting a tree-style topology that has 8 stages in depth. The processing logic of this application is straightforward: once a new tweet is pulled into the system through Kestrel Spout (Op_1), it is firstly stored by a file writer (Op_2) and examined by a language detector (Op_3) to identify which language it uses. If it is written in English, there is a sentiment analysis bolt (Op_4) that splits the sentence and calculates the sentimental score for the whole content using AFINN[16], which contains a list of words with their pre-computed sentiment valence from minus five (negative) to plus five (positive). There are also several bolts to count the average sentiment result (Op_5, Op_6) and to rank the most frequent hashtags occurring over a specific time window ($Op_7 \sim Op_{11}$).

6.1.2. Evaluation Methodology. We use throughput and complete latency to quantitatively evaluate the performance of streaming applications. Higher monitored throughput indicates higher performance potential, as long as the complete latency satisfies the desired target. In other words, if a streaming application has demonstrated throughput T in the profiling environment, we can confidently assume that it has ability to process any throughput $T^{'} < T$ without violating the latency constraint, unless

[13]The selectivity is defined as the ratio between the number of output tuples produced and the number of tuples consumed by this operator.

[14]Operators in Storm are called *spouts*—if they are data sources—or *bolts* otherwise.

[15]https://github.com/kantega/storm-twitter-workshop

[16]http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=6010

the profiling knowledge needs to be recalibrated. Therefore, to probe the maximum sustainable throughput, the profiling environment feeds the applications with large inputs, until the performance hits its highest stable point before recording it as the observed value.

The measurement of performance metrics first requires the test application to be deployed on the execution platform. Apart from complying with the generated configuration, we also set the number of workers to one per machine and the number of tasks to be the same as the number of executors, which conforms to the recommendation of the Storm community[17]. All the topologies run for 10 minutes to enable sufficient stabilization, and then performance data are collected every 30 seconds for 10 minutes, forming an array of 20 observations on throughput and latency. These settings were chosen because we observed that the fluctuation among the average results of repeat experiments did not exceed 3%, and the Lilliefors Test does not reject the null hypothesis that the observations on throughput are normally distributed (at the 5% significance level). However, other applications may require longer time to reach a stable state, or a larger monitoring interval to avoid drastic but periodic throughput variation.

As we have collected an array of throughput metrics in each profiling round, the significant change mentioned in Algorithm 2 can be determined by a Two Sample T-Test (at the 5% significance level) to determine if there is statistically significant difference between the performance of previous and new configuration.

For completeness, Table II summarizes the parameter settings used for setting up the stepwise profiler in our evaluation.

Table II. The parameter settings used by the stepwise profiler in evaluations.

Parameters	**Values**
Latency constraint (Y_{con})	500 ms
Task load unit ($slice$)	0.3
Stopping coefficient (k)	2
Threshold for triggering reconfiguration (α)	0.9

6.1.3. Comparable Methods. We compare the stepwise profiling prototype with two existing scaling up approaches: the threshold-based method and Stela [Xu et al. 2016].

The threshold-based method adjusts the parallelism hint of each operator based on its monitored capacity as formulated in Equation 2, in contrast to those in the literature that set up thresholds over the CPU utilization of worker nodes [Heinze et al. 2014b; Gulisano et al. 2012]. The scaling up threshold in our experiment is set to be 0.8 and we reduce the capacity of congested operators by gradually increasing their parallelism. In this sense, it may take several rounds to complete the scaling up process: the application is deployed with no parallelism configured[18] at the beginning. In the following rounds, the most overloaded operator will be provided with an extra task in an attempt to rectify the congestion and optimize performance.

Stela scales up the streaming applications with the same goal of optimizing post-scaling throughput. In contrast to the threshold-hold method that examines only the operator capacity for bottleneck detection, Stela prioritizes those congested yet influential operators in the scaling up process by calculating the ETP (Effective Throughput Percentage) metric [Xu et al. 2016]. Furthermore, it allows the parallelism degree of

[17] https://storm.apache.org/documentation/FAQ.html

[18] By default, Apache Storm initializes each operator in the topology with one task for execution.

multiple operators to be adjusted in a single monitoring round, thus greatly reducing the time span of scaling up process. However, Stela is initially designed for on-demand elasticity, hence some changes are required to make it comparable with our approach:

(1) The scaling out process is omitted as we intend to optimize the application performance on a pre-configured cluster. All infrastructural resources are made available to Stela from the beginning of the scaling up process.
(2) A single monitoring round of Stela corresponds to an on-demand scaling request in its original form, which may involve multiple scaling up iterations. During each iteration, Stela calculates the ETP for all operators and assigns a new task to the operator with the highest ETP. Before proceeding to the next iteration, the table of ETP is updated with projected values that estimate the consequence of scaling, such as the projected input rate and the processing rate of the targeted operator.
(3) Since the estimation of ETP is prone to error propagation, we limit the maximum number of scaling up attempts in a monitoring round to m, which is the number of worker nodes available at the infrastructure level. In this way, the efficacy of the scaling algorithm is assured as the table of ETP is revised with monitored data every m iterations; and the risk of over-scaling is controlled since each machine will be assigned with no more than one new task in a single monitoring round.

6.2. Applicability Evaluation

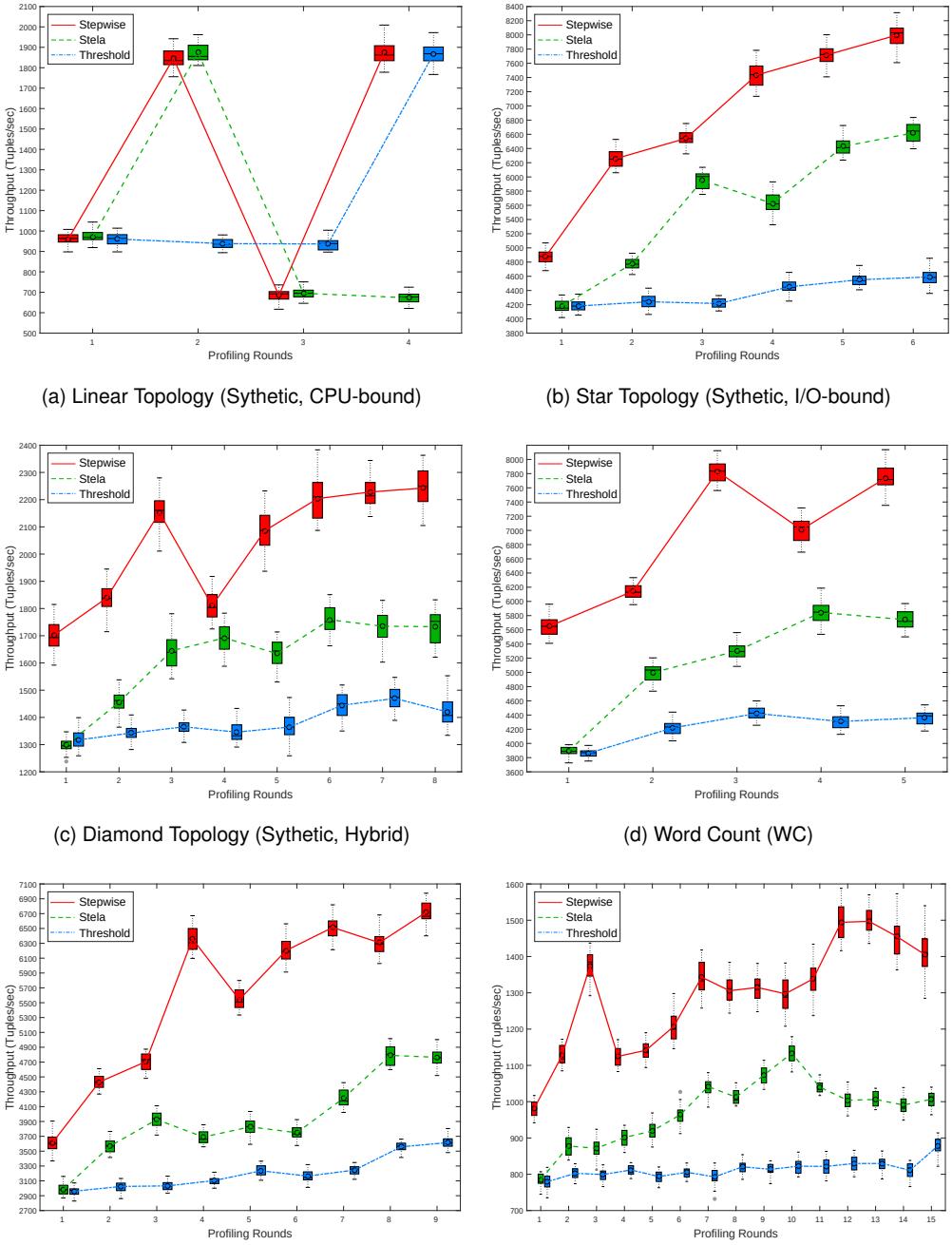
In the applicability experiment, all the topologies are executed in 6 worker nodes. We configured the micro-benchmark topologies with different resource complexities in order to examine how application diversity affects the performance optimization process. Specifically, the Linear topology incorporates only CPU-bound operators so that the whole application is bound by available CPU resources; while the Star topology consists of only I/O bound operators, causing its performance to be bound by communication capability[19]. The Diamond topology, on the other hand, is a hybrid streaming application that includes all sorts of operators (1 CPU bound, 1 I/O bound, and 2 Sojourn time-bound) in the intermediate tier, making its bottleneck more difficult to identify and resolve in the scaling up process.

The results in Figure 11 show that the stepwise profiler successfully scales up the targeted topologies. In particular, the Linear topology reaches its maximum throughput at 1876 with the parallelism set as $(1, 2, 2, 2)$[20], which is 95.7% higher than its initial throughput performance yielded by $(1, 1, 1, 1)$. It took 4 rounds for the scaling up process to converge: the stepwise profiler tried the configuration of $(1, 3, 3, 3)$ at round 3, but it then rejected such configuration change due to the observed performance degradation. Note that the operator capacity profiling is entirely omitted in this scaling up process, as the measured operator latencies have all fallen into the vicinity of the monitored $MinL_p$ by a factor of 2.

Being I/O intensive in nature, the Star topology requires much higher parallelism settings to enable satisfactory performance, which consequently leads to a longer scaling up process. In our evaluation, the scaling up process took 6 rounds to finish, with the parallelism finally set as $(3, 3, 48, 24, 24)$ delivering 64% higher throughput than the first round. Thank to the homogeneity of operator implementation, there is no need to fine tune the operator capacities as the stopping condition on latency has been met.

The Diamond topology, in contrast, spent 3 rounds in the third step to further scale up the I/O bound operator (Op_3). During the process of platform capability profiling, stepwise profiler successfully determines the right parallelism for CPU-bound and So-

[19]For I/O bound topologies (e.g. Star), we set Y_{con} to 100 ms to reflect stricter timeliness requirement.
[20]From left to right, each number corresponds to the number of tasks of each operator in the Linear Topology.



(a) Linear Topology (Sythetic, CPU-bound)

(b) Star Topology (Sythetic, I/O-bound)

(c) Diamond Topology (Sythetic, Hybrid)

(d) Word Count (WC)

(e) Sythetic Word Count (SWC)

(f) Twitter Sentiment Analysis (TSA)

Fig. 11. Scaling up testing applications on 6 processing nodes, the X axis represents a series of profiling rounds and the Y axis compares the throughput resulting from different configurations. In each profiling round, we use 3 boxplots that each contains 20 readings of throughput to denote the observation variances.

journ time-bound operators; however, it underestimates the number of tasks for Op_3 and causes it to be the throughput bottleneck. The reason of insufficient scaling is that Equation 1 made a conservative decimal conversion by using $slice$ of 0.3, which prevents Op_3 from scaling more than 4 times faster than the other operators. We will shed more light on the effect of parameter selection in Section 6.4.

In addition, by interpreting the scaling up process of real-world streaming applications, we conclude that our method is consistently better than the other two scaling approaches in the following three aspects. Firstly, stepwise profiling exploits the inherent feature of a streaming application and thus has a more reasonable starting point of profiling comparing to the other two baseline methods, which by contrast determine the initial configuration only based on the topology structure. Figure 11 illustrates that the application feature profiling for WC, SWC, and TSA improves the performance by 45%, 21.1% and 25% at the beginning, respectively.

Secondly, as platform capability profiling collectively adjusts the parallelism hints for a set of operators, it significantly enhances the performance gains obtained from the first few profiling rounds. On average, the relative performance improvement observed from the first four rounds in our method is 2.48 times as large as that of Stela, and 11.63 times compared to that of the threshold-based method. Besides, despite having the ability to tune multiple parallelism hints in a single round, Stela's estimation-based algorithm could lead to incorrect scaling decision, e.g. it added new tasks to logic-dependent operators and caused performance degradation at round 10 in Figure 11f. To make things worse, there is no reversal mechanism to rollback the wrong move.

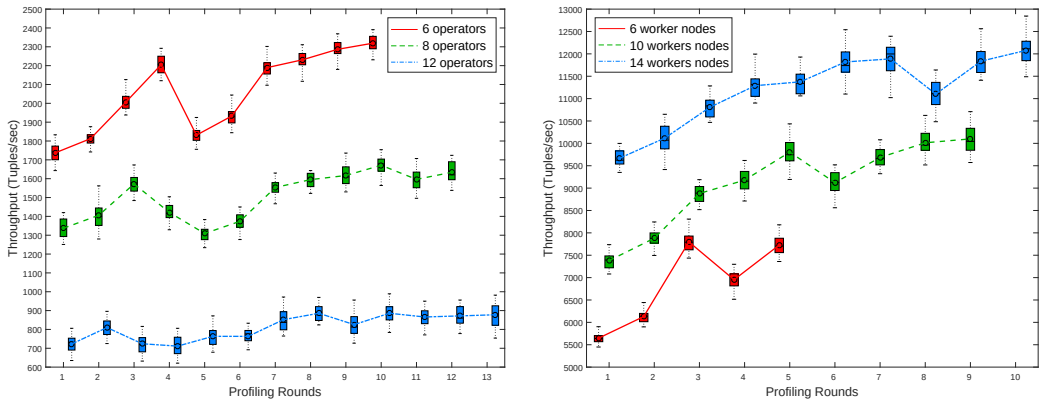
Finally, the stopping condition introduced in Section 4.3 greatly limited the number of profiling rounds. Specifically, stepwise profiling stops trying new configuration in TSA because there are successive revocations that show increasing parallelism hint no longer benefits the performance. In WC and SWC, the profiler execution terminates when the latencies for each bolt dropped into a range of (0, $2{*}MinL_p$], which indicates that the application has been sufficiently scaled up. In the end, our approach is 34.1%, 40.1%, 31.9% better than the best alternative in terms of the throughput resulted from the final configuration, respectively.

With the performance information profiled, the quality of different topology implementations in terms of their performance potentials can be easily observed. In this case, SWC is consistently worse than WC as the former implementation only reaches 86.8% throughput of the latter and it takes more effort (9 rounds vs 5 rounds) to probe a reasonable configuration.

6.3. Scalability Evaluation

We explore the scalability of our stepwise profiling prototype in two dimensions. The first dimension is topology complexity, which examines how the increasing number of operators in the topology affects the scaling up process. The other dimension is platform size, which checks if the prototype is able to deliver a reasonably higher post-scaling performance using more resources. Meanwhile, we also compare stepwise profiling with Stela in terms of the minimal resources needed to reach a specific performance target.

In the first experiment, we run the Linear topology with various types of operators on 6 worker nodes. The topology depth is further extended to 6, 8 and 12 in order to construct a more complex structure. Results in Figure 12a show that increasing the topology chain leads to a longer operator capacity profiling process, but the overall profiling effort does not scale linearly with the number of operators. This is because the monitored $MinL_p$ also increases along with the topology complexity and contributes to the timely termination of the profiling process. In fact, we observed that the stopping condition on latency is satisfied by most operators at the end of the platform capabil-



(a) Linear Topology (Sythetic, Hybrid) (b) Word Count (WC)

Fig. 12. Scalability evaluation of the stepwise profiler. The X axis represents a series of profiling rounds and the Y axis compares the throughput resulting from different configurations.

ity profiling, and only I/O bound operators demand further adjustment of capacity as their $MinL_p$ are relatively small and hard to approach. This observation enables the conclusion that the parameter selection process is application-dependent and a higher k should be set for I/O bound topologies.

Additionally, this experiment showcases that the increasing complexity of target application compromises the performance gain from profiling, with the monitored improvement being 33.5%, 24%, 20.5% in the three test cases, respectively. Therefore, a larger platform is required for complex streaming topologies to obtain satisfactory performance.

In the second experiment, we run the Word Count topology on 6, 10 and 14 worker nodes, respectively. Note that by using 14 worker nodes we can still guarantee that one virtual CPU corresponds to a physical core so as to avoid the interference of CPU overbooking. The results in Figure 12b demonstrate that the application features profiled in the first step, i.e. $MinL_p$ and RSS, are nicely maintained on larger platforms, therefore the process of platform capability profiling is accordingly extended to provide higher parallelism for different operators. However, the stepwise profiler is not able to achieve linear growth of performance using more resources. This is because other factors, such as task location and concurrency settings, also influence the throughput outcome, but they are not fine-tuned by the profiler due to the hardness of modelling.

Using WC as the test topology, we also applied Stela and stepwise profiling on an increasing number of nodes, from 2 to 14, to determine the performance limits of the topology given different resources. Both methods were executed with the same iterations to ensure fairness, and the results of scaling shed light on the minimal resource provision needed for the test application to reach a specific throughput target.

Figure 13 shows that the stepwise profiling is able to reduce resource usage by up to 57.1%. For example, stepwise profiling can achieve a target of 6000 tweets processed per second using only 4 nodes. On the other hand, Stela suggests 8 nodes to process such stream without breaking SLA, which results in a significant resource wastage.

Additionally, it took 200 minutes for the stepwise profiler to complete the scaling up process of WC on 16 machines (2 for control and coordination and 14 for execution). Given that a configuration trial in each round runs for 20 minutes, and that 10 config-

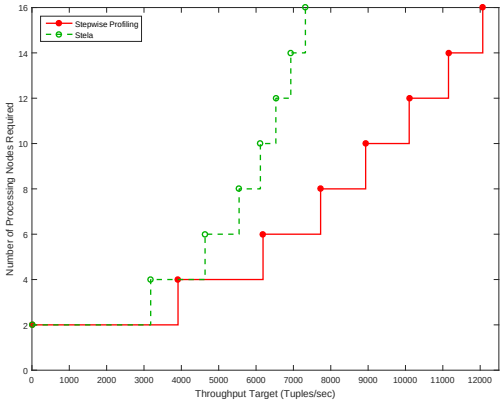


Fig. 13. Relationship between the throughput target and required resources to handle it without latency violation.

urations are evaluated, the overhead incurred by the profiling algorithm in the whole process is negligible.

6.4. System Parameters Evaluation

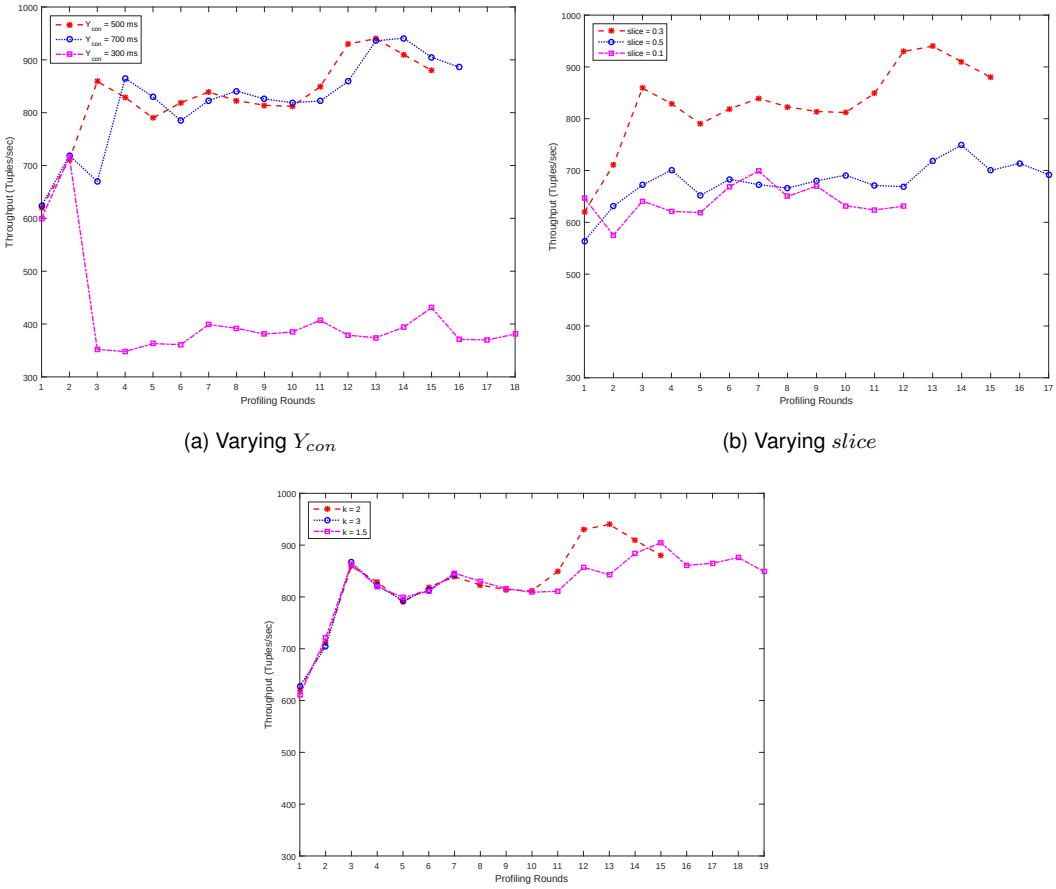
In this experiment, we evaluate the influences of three parameters in the performance of stepwise profiler. These include the user-specified latency constraint Y_{con}, the task load unit $slice$ and the stopping coefficient k. In particular, TSA is executed on 4 processing nodes. When a particular parameter is being examined, the others were set to their default values. Table III describes the evaluated and default values for each parameter.

Table III. Evaluated parameters and their values. Default values are showed in bold.

Parameters	**Values**
Latency constraint (Y_{con})	300 ms, **500 ms**, 700 ms
Task load unit ($slice$)	0.1, **0.3**, 0.5
Stopping coefficient (k)	1.5, **2**, 3

Results show that relaxing Y_{con} does not necessarily increase the throughput. In fact, it encourages the profiler to try further data source scaling operations in the second profiling step, checking if the bottleneck lies in insufficient data supply. As shown in Figure 14a, the third data point marked with a circle denotes the operation that scales up the data source, but it is revoked because the overall throughput is impaired by this change. On the other hand, the throughput would be significantly affected if Y_{con} were set to an overly small value. As indicated by the third data point marked with a square, our method has to throttle the data source at the end of the second profiling step to meet the latency requirement, and the following rounds in the third step do not compensate the performance degradation due to the strict latency constraint.

The behaviour of the platform capability profiling mainly depends on the value of $slice$. When $slice$ increases from 0.3 to 0.5, both the starting point and the performance gain from scaling are worse than the default case, reaching only 90.9% and 79.7% of the default case performance, respectively. This is because, in this case, $\overrightarrow{R}$ is no longer able to describe the proportion of different operators, which in turn causes heavy bottleneck in bolts in the whole topology. By contrast, a value of $slice$ that is too small exaggerates



(a) Varying Y_{con}

(b) Varying $slice$

(c) Varying k

Fig. 14. Influence of different parameters on the performance of stepwise profiling. For better readability, we only plot the average of throughput in each profiling round.

this proportion and makes each scaling attempt more extreme. Data points marked with squares in Figure 14b show that, even though the profiler managed to improve the performance of the starting point against the normal case, the following scaling trials in the second step all failed because of over-scaling — too many tasks were being added each time.

Variation of the parameter k mainly affects the number of rounds in the operator capacity profiling. Figure 14c shows that, when k changes from 2 to 3, the whole third step of profiling is omitted at round 7 because each operator satisfies the stopping condition, though only a suboptimal configuration is obtained in this case. On the contrary, when k is decreased to 1.5, more operators are involved in the third step, which causes a longer series of performance fluctuation. Note that more rounds of profiling in the third step do not guarantee a better throughput due to the nature of greedy heuristics.

7. RELATED WORK

In summary, our work introduces a controlled profiling environment allowing evaluation of different configurations, with the objective of finding and employing a tailored

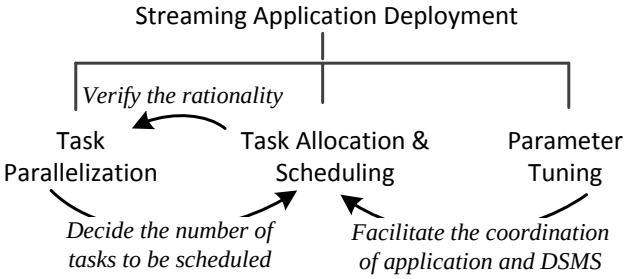


Fig. 15. Three processes of deploying a streaming application on the operator-based DSMS running in a cloud and cluster environment. Text in italic describes the interrelation between them.

deployment plan to capture relevant characteristics of both the application and target execution platform. Since our research goal is to achieve performance oriented deployment for applications on operator-based DSMS, and the adopted method falls into the broad scope of application profiling, this section reports relevant works in these two fields—performance oriented deployment and application profiling. There is also a line of work applicable to the previous generation of DSMS that focused on tuning performance without changing the semantic of a streaming application. Therefore, we succinctly review them and summarize how performance optimization is achieved on other types of DSMS.

7.1. Performance Oriented Deployment

As shown in Figure 15, there are three tightly coupled processes involved in streaming application deployment once the target cluster or cloud environment has been provisioned: (i) *task parallelization*, which involves decision of the parallelism degree for the logic DAG, such that each abstract operator is translated into a certain number of tasks to conduct real data operations; (ii) *task allocation & scheduling*, which involves allocation and scheduling of tasks among participating compute nodes; and (iii) *parameter tuning*, which concerns fine-grained adjustment of available parameters for better coordination of the application and the platform.

Only a handful of works investigated the task parallelization problem. Researchers in IBM [Schneider et al. 2009; Schneider et al. 2012; Gedik et al. 2014] tried to automate this process using a compiler and runtime system that is capable of identifying and levering potential data-parallel regions for applications on System S [Jain et al. 2006]. But instead of altering the parallelism to improve the application performance, their work mainly focused on addressing the safety challenge related to parallelization, which has already been handled by the implementation of state-of-the-art DSMSs. Fischer et al., who abstract the streaming application as a black box with an unknown performance function, proposed another similar work that regards the task parallelization as only a part of parameter tuning [Fischer et al. 2015]. Though the adopted Bayesian optimization method has demonstrated its effectiveness through extensive evaluation, it would lead to an inherent lengthy convergence process compared to our stepwise profiling approach in which operators are heuristically parallelized with insights obtained from the queuing model.

Elasticity in DSMSs has received increasing research attention as it enables cost-efficient handling of workload variations. Some works explored dynamically scale out/in streaming applications through the adjustment of parallelism settings as well as tuning relevant parameters. DRS is a resource scheduler that dynamically decides the parallelism hint for each operator based on queueing theory, with the goal of minimizing the total sojourn time of an average input [Fu et al. 2015]. However, it targets only

computation-intensive applications. Lohrmann et al. [Lohrmann et al. 2015] continuously rebalance the topology with new configurations according to a proposed latency model, and they double the parallelism of any operator found to be a bottleneck. Nevertheless, the proposed bottleneck resolving method is coarse-grained and may lead to resources wastage. Heinze et al. compared three different scaling techniques in terms of the quality of the produced scaling decisions, and the results demonstrated that reinforcement learning is more adaptive and robust than the threshold-based alternatives [Heinze et al. 2014b]. Hidalgo et al. combined the threshold-based method with the Markov chain model to dynamically change the operator parallelism, so that the short-term and mid-term workload variations can be handled with reactive and predictive approaches, respectively. [Hidalgo et al. 2017]. Besides, realizing elasticity for stateful operators requires non-trivial efforts to handle issues such as stream rerouting and state migration. While the adopted pause-and-resume strategy is commonly seen in the literature [Castro Fernandez et al. 2013; Cardellini et al. 2016; Madsen et al. 2016], there are also advanced protocols for operator movement and state management that allow for interruption-free elasticity [Wu and Tan 2015; Matteis and Mencagli 2017; Ravindra et al. 2017]. In future work, these techniques can be integrated in our prototype to improve its responsiveness against workload burst. As for parameter tuning, Das et al. [Das et al. 2014] proposed a control algorithm to automatically determine the most suitable batch for a given state, while online parameter optimization has been investigated by Heinze et al. to deal with the situation where the application needs to be dynamically scaled as a reaction to workload changes [Heinze et al. 2015]. Our target is different to all those above as we try to determine the configuration for any streaming application given the platform that maximizes the throughput under latency constraint.

In contrast, the task allocation and scheduling problem has received much more attention from the research community. Aniello et al. pioneered this area with two scheduling algorithms on Apache Storm: the off-line version makes all the scheduling decisions through a static analysis of the logic DAG, while the on-line version regularly collects runtime information to sort all the communicating pairs of tasks, with an attempt to sequentially co-locate them in the same node to reduce communication cost [Aniello et al. 2013]. Inspired by this idea, many works extended the on-line algorithm by adding some other aspects into consideration, such as scheduling overhead, resource awareness and energy efficiency. Chatzistergiou et al. proposed a linear time task allocation algorithm to adaptively reconfigure task locations in the presence of environment changes, resulting in a significant improvement from the existing quadratic time solutions [Chatzistergiou and Viglas 2014]. Fischer et.al. presented an application agnostic algorithm that supports scheduling of large-scale task graphs with regard to the communication pattern, the problem of minimizing inter-node messages is thus translated into a graph partitioning problem which can be solved by the use of METIS algorithm [Fischer and Bernstein 2015].

On the other hand, there are also some papers that explored the area of resource aware scheduling and put an emphasis on worker node consolidation. The algorithm used in T-Storm [Xu et al. 2014] tries to minimize both inter-node and inter-process traffic while avoiding overloading the dwindled worker nodes. Similarly, Peng et al. [Peng et al. 2015] considered the task scheduling as a variation of the Knapsack problem with several hard/soft resource constraints, so that it can be solved by the application of linear programming given that the user has provided the resource demand and availability information. Apart from the common target of reducing communication cost, Re-Stream, an energy-efficient resource scheduling mechanism by Sun et al., proposed the minimization of energy consumption as long as the response latency meets SLA requirements. This is achieved by an analytic model that depicts the rela-

tionship among energy consumption, response time, and the resource utilization [Sun et al. 2014; Sun et al. 2015]. Our work can be used along with these methods above as none of them address the issue of task parallelization.

Besides cluster and cloud environments that are the target of our approach, the problem of deploying streaming application on multi-core systems and distributed networks has also been discussed by several works. Hormati et al. [Hormati et al. 2009] proposed a framework that dynamically adapts applications to the changing characteristics of the multi-core resources in order to maximize the throughput, using a hybrid approach of static compilation and dynamic configurations adjustments. Similarly, Suleman et al. [Suleman et al. 2010] introduced a framework to tune the parallelism for each stage in a processing pipeline using a hill-climbing algorithm that can both save time and reduce the number of used cores. As for network deployment, Cardellini et al. [Cardellini et al. 2015] extended Apache Storm with a self-adaptive distributed scheduling mechanism, which allows execution of streaming applications on a geographically distributed environment with a certain level of QoS guarantee.

7.2. Application Profiling

Application profiling is a technique that actively extracts and evaluates the characteristics of applications, for example, the space or time complexity, to facilitate the use of computing resources. The profiled data sets can be either low-level usage traces of CPU, memory, and network bandwidth, or high level metrics that are part of application SLA, such as throughput, latency and fault-tolerance ability [Weingartner et al. 2015]. In order to make sure that the application profile would accurately reflect resource needs, the profiling process is normally conducted in a dedicated profiling environment following the MAPE-K autonomic loop (Monitor, Analyze, Plan, Execute - Knowledge) [Kephart and Chess 2003], which enables controllable organization of input data and eliminates variation factors that would affect the result collection and analysis.

Most of the state-of-the-art programming IDEs, such as Microsoft Visual Studio and Eclipse, provide tools to aid in determining bottlenecks in the code that affect the overall performance of a program. However, the research community has gone way beyond code-level performance profiling. Urgaonkar et al. [Urgaonkar et al. 2002] investigated the overbooking problem by the use of application profiling, which helps to deliver an accurate estimate of resource needs for application components co-located on shared hosts. Do et al. [Do et al. 2011] achieved better virtual machine placement with a performance prediction model derived from the application profile. To obtain higher profiling accuracy, they identify background load, which is the interference of other applications into consideration. Shen et al. [Shen et al. 2015] used profiling to automate the detection of performance bottleneck for web applications with a large set of input parameters. Similar to our work, the proposed profiling method is able to heuristically search the best configuration that maximizes the objective performance function. Qian et al. [Qian et al. 2011] developed a tool that profiles the cross-layer interaction within mobile applications, aiming to better reveal the performance and energy bottlenecks hidden in the inefficient resource usages. Still, our work is different to them in that we adopt the profiling method to guide the deployment process of streaming application, while the above-mentioned models mostly target batch-oriented (MapReduce) or interactive-oriented (web and mobile) applications and thus cannot be directly applied in streaming applications.

It is also worth mentioning that we have carefully designed the stepwise profiler to avoid DSMS lock-in. Besides Apache Storm, there are many operator-based DSMSs that support general purpose stream processing, including Microsoft TimeStream [Qian et al. 2013], Apache Samza, Apache Flink [Lohrmann et al. 2014],

and Twitter Heron [Kulkarni et al. 2015]. None of them has a built-in feature to automatically decide the parallel configuration for a particular application, and thus all of them can benefit from the proposed profiler.

7.3. Other Performance Optimization Techniques

It has been more than a decade since the first generation DSMSs, including Aurora [Abadi et al. 2003], Niagara [Chen et al. 2000] and Telegraph [Chandrasekaran et al. 2003], were introduced to facilitate the development and deployment of streaming applications. Along with the increasing adoption of DSMS, various optimization techniques have been developed to improve the performance of applications without changing their topology or semantics.

Operator placement optimization, for example, is a process of assigning operators to specific hosts and cores to reach a trade-off between communication cost and resource contention. Though it is still a kind of adjustment in application layout rather than spreading and scheduling tasks (as discussed in Section 7.1), operator placement in previous generation DSMS regards each operator as an indivisible entity that can only appear in one place at a time. In this context, Gordon et al. designed a software-programmable substrate capable of generating custom communication code to reduce message hops when placing operator on multi-core systems [Gordon et al. 2002]. Auerbach et al. proposed a placement mechanism to guarantee that operators compiled for an FPGA will always be placed on hosts with FPGAs [Auerbach et al. 2010]. In addition to resource matching, Wolf et al. [Wolf et al. 2008] considered other constraints during the placement process, such as licensing and security requirement.

Load balancing is another commonly used optimization technique to evenly distribute workload across available resources. This requires either a balanced operator placement plan or a runtime mechanism to dynamically assign stream tuples to operators. As examples of these two approaches, Xing et al. migrated conflicting operators that experience load spikes at the same time to separate locations to avoid resource contention and thus improving load balance [Xing et al. 2005], while Amini et al. [Amini et al. 2006] discussed the use of back-pressure in System S to compensate skews found in runtime.

However, these optimization techniques are no longer applicable to state-of-the-art streaming applications built on top of operator-based DSMS, as the implementation of DSMS has greatly evolved towards scalability and robustness, causing operator placement and load balancing to rely heavily on the parallelization and scheduling of tasks that constitute the operator.

8. CONCLUSIONS AND FUTURE WORK

We proposed a streaming application profiler that consists of three steps, namely (i) application feature profiling, which aims to identify the complexity and task load for each operator; (ii) platform capability profiling, which endeavours to scale up the application with the knowledge learned from the previous step; and (iii) operator capacity profiling, which makes necessary amendments on fine-grained level to further improve performance of the application. Our profiler can be used to scale up streaming application, build the relationship between the underlying resources and the performance metrics, and further evaluate the choice of resource provision. An evaluation of a profiler prototype applied to three real world applications showed that our approach is able to automatically improve the throughput up to 40.1% compared to Stela, a state-of-the-art alternative scaling approach.

As for future work, we plan to devise auto-scaling policies on top of the profiling results, which enables dynamic adjustment of provisioned resources according to real-time performance requirements of a variety of workloads. Since responsiveness is a

critical criterion for realizing runtime-adaptation, the envisioned auto-scaling policies should be built on top of interruption-free scaling mechanisms such as dynamic stream rerouting and live state migration. Deploying the streaming applications on the cloud and exploiting the use of VM with different configurations are also on our future plan. A distinct advantage of using cloud resources is that it supports resource customization. Therefore, there is a great potential in performance optimization to place different operators on tailored cloud instances that fit their special needs, such as hosting computation-intensive operators on fast CPU nodes and placing operators with intensive intercommunication in the same virtual node to minimize communication overhead. We would also like to leverage different cloud pricing models (On-Demand, Reserved, Spot) with the aim of minimizing the monetary cost of stream processing, making cloud a preferable platform for deployment of streaming applications.

REFERENCES

Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal* 12, 2 (Aug. 2003), 120–139.

Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. 2006. Adaptive Control of Extreme-scale Stream Processing Systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS '06)*. IEEE Computer Society, 71–77.

Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. 2013. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS '13)*. ACM, 207–218.

Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, 89–108.

Paolo Bellavista, Antonio Corradi, Andrea Reale, and Nicola Ticca. 2014. Priority-Based Resource Scheduling in Distributed Stream Processing Systems for Big Data Applications. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC '14)*. IEEE, 363–370.

Michael Cammert, Christoph Heinz, Jurgen Kramer, Bernhard Seeger, Sonny Vaupel, and Udo Wolske. 2007. Flexible Multi-Threaded Scheduling for Continuous Queries over Data Streams. In *Proceeding of the 23rd IEEE International Conference on Data Engineering Workshop (ICDE '07)*. IEEE, 624–633.

Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2015. Distributed QoS-aware Scheduling in Storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS '15)*. ACM, 344–347.

Valeria Cardellini, Matteo Nardelli, and Dario Luzi. 2016. Elastic Stateful Stream Processing in Storm. In *Proceedings of the 2016 International Conference on High Performance Computing Simulation (HPCS)*. IEEE, 583–590.

Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, 725–736.

Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. ACM, 668–668.

Andreas Chatzistergiou and Stratis D. Viglas. 2014. Fast Heuristics for Near-Optimal Task Allocation in Data Stream Processing over Clusters. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management (CIKM '14)*. ACM, 1579–1588.

Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. ACM, 379–390.

Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. 2014. Adaptive Stream Processing Using Dynamic Batch Sizing. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, 1–13.

Anh Vu Do, Junliang Chen, Chen Wang, Young Choon Lee, A.Y. Zomaya, and Bing Bing Zhou. 2011. Profiling Applications for Virtual Machine Placement in Clouds. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD '11)*. IEEE, 660–667.

Lorenz Fischer and Abraham Bernstein. 2015. Workload Scheduling in Distributed Stream Processors using Graph Partitioning. In *Proceedings of the 2015 IEEE International Conference on Big Data (BigData '15)*. IEEE Computer Society, 124–133.

Lorenz Fischer, Shen Gao, and Abraham Bernstein. 2015. Machines Tuning Machines: Configuring Distributed Stream Processors with Bayesian Optimization. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing (CLUSTER '15)*. IEEE, 22–31.

Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2015. DRS: Dynamic Resource Scheduling for Real-Time Analytics over Fast Streams. In *Proceedings of the IEEE 35th International Conference on Distributed Computing Systems (ICDCS '15)*. IEEE, 411 – 420.

Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2014. Elastic Scaling for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (June 2014), 1447–1463.

Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. 2002. A Stream Compiler for Communication-exposed Architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, 291–303.

Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. 2012. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Transactions on Parallel and Distributed Systems* 23, 12 (Dec. 2012), 2351–2365.

Thomas Heinze, Leonardo Aniello, Leonardo Querzoni, and Zbigniew Jerzak. 2014a. Cloud-based Data Stream Processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. ACM, 238–245.

Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. 2014b. Auto-scaling Techniques for Elastic Data Stream Processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. ACM, 318–321.

Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. 2015. Online Parameter Optimization for Elastic Data Stream Processing. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC '15)*. ACM, 276–287.

Nicolas Hidalgo, Daniel Wladdimiro, and Erika Rosas. 2017. Self-adaptive Processing Graph with Operator Fission for Elastic Stream Processing. *Journal of Systems and Software* 127 (2017), 205 – 216.

Amir H. Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. 2009. Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*. ACM, 214–223.

Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. 2013. Elastic Stream Processing in the Cloud. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 3, 5 (Sept. 2013), 333–345.

Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. 2006. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. ACM, 431–442.

Jeffrey O. Kephart and David M. Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (Jan. 2003), 41–50.

Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, 239 – 250.

Teng Li, Jian Tang, and Jielong Xu. 2015. A Predictive Scheduling Framework for Fast and Distributed Stream Data Processing. In *Proceedings of the 2015 IEEE International Conference on Big Data (BigData '15)*. IEEE Computer Society, 333–338.

Bjrn Lohrmann, Peter Janacik, and Odej Kao. 2015. Elastic Stream Processing with Latency Guarantees. In *Proceedings of the 2015 IEEE 35th International Conference on Distributed Computing Systems (ICDCS '15)*. IEEE, 399 – 410.

Bjrn Lohrmann, Daniel Warneke, and Odej Kao. 2014. Nephele Streaming: Stream Processing under QoS Constraints at Scale. *Cluster computing* 17, 1 (2014), 61–78.

Kasper Grud Skat Madsen, Yongluan Zhou, and Li Su. 2016. Enorm: Efficient Window-based Computation in Large-scale Distributed Stream Processing Systems. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS '16)*. ACM, 37–48.

Tiziano De Matteis and Gabriele Mencagli. 2017. Proactive Elasticity and Energy Awareness in Data Stream Processing. *Journal of Systems and Software* 127 (2017), 302 – 319.

Lory Al Moakar, Alexandros Labrinidis, and Panos K. Chrysanthis. 2012. Adaptive Class-Based Scheduling of Continuous Queries. In *Proceeding of the 28th IEEE International Conference on Data Engineering Workshop (ICDE '12)*. IEEE, 289–294.

Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. 2015. R-Storm: Resource-Aware Scheduling in Storm. In *Proceedings of the 16th Annual Middleware Conference (Middleware '15)*. ACM, 149–161.

Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. 2011. Profiling Resource Usage for Mobile Applications: A Cross-layer Approach. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys '11)*. ACM, 321–334.

Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the European Conference on Computer Systems (EuroSys '13)*. ACM, 1–14.

Sajith Ravindra, Miyuru Dayarathna, and Sanath Jayasena. 2017. Latency Aware Elastic Switching-based Stream Processing Over Compressed Data Streams. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, 91–102.

Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. 2009. Elastic Scaling of Data Parallel Operators in Stream Processing. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS '09)*. IEEE, 1 – 12.

Scott Schneider, Martin Hirzel, Bugra Gedik, and Kun-Lung Wu. 2012. Auto-parallelizing Stateful Distributed Streaming Applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, 53–64.

Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. 2015. Automating Performance Bottleneck Detection Using Search-based Application Profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA '15)*. ACM, 270–281.

Muhammad Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. 2010. Feedback-directed Pipeline Parallelism. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. ACM, 147–156.

Dawei Sun, Ge Fu, Xinran Liu, and Hong Zhang. 2014. Optimizing Data Stream Graph for Big Data Stream Computing in Cloud Datacenter Environments. *International Journal of Advancements in Computing Technology* 6, 5 (2014), 53–65.

Dawei Sun, Guangyan Zhang, Songlin Yang, Weimin Zheng, Samee U. Khan, and Keqin Li. 2015. Re-Stream: Real-time and Energy-efficient Resource Scheduling in Big Data Stream Computing Environments. *Information Sciences* 319 (Oct. 2015), 92–112.

Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. 2002. Resource Overbooking and Application Profiling in Shared Hosting Platforms. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 239–254.

Rafael Weingartner, Gabriel Beims Brascher, and Carlos Becker Westphall. 2015. Cloud Resource Management: a Survey on Forecasting and Profiling Models. *Journal of Network and Computer Applications* 47 (2015), 99 – 106.

Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. 2008. SODA: An Optimizing Scheduler for Large-scale Stream-based Distributed Computer Systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware '08)*. Springer-Verlag, 306–325.

Yingjun Wu and Kian-Lee Tan. 2015. ChronoStream: Elastic Stateful Stream Computation in the Cloud. In *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering*. IEEE, 723–734.

Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. 2005. Dynamic Load Distribution in the Borealis Stream Processor. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. IEEE Computer Society, 791–802.

Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. 2014. T-Storm: Traffic-Aware Online Scheduling in Storm. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems (ICDCS '14)*. IEEE Computer Society, 535–544.

Le Xu, Boyang Peng, and Indranil Gupta. 2016. Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand. In *Proceedings of the 2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 22–31.