SpinStreams: a Static Optimization Tool for Data Stream Processing Applications

Gabriele Mencagli
Department of Computer Science
University of Pisa, Italy
mencagli@di.unipi.it

Patrizio Dazzi
ISTI-CNR
Pisa, Italy
patrizio.dazzi@isti.cnr.it

Nicolò Tonci
Department of Computer Science
University of Pisa, Italy
tonci@di.unipi.it

ABSTRACT

The ubiquity of data streams in different fields of computing has led to the emergence of *Stream Processing Systems* (SPSs) used to program applications that extract insights from unbounded sequences of data items. Streaming applications demand various kinds of optimizations. Most of them are aimed at increasing throughput and reducing processing latency, and need cost models used to analyze the steady-state performance by capturing complex aspects like *backpressure* and *bottleneck* detection. In those systems, the tendency is to support dynamic optimizations of running applications which, although with a substantial run-time overhead, are unavoidable in case of unpredictable workloads. As an orthogonal direction, this paper proposes `SpinStreams`, a static optimization tool able to leverage cost models that programmers can use to detect and understand the inefficiencies of an initial application design. `SpinStreams` suggests optimizations for restructuring applications by generating code to be run on the SPS. We present the theory behind our optimizations, which cover more general classes of application structures than the ones studied in the literature so far. Then, we assess the accuracy of our models in `Akka`, an actor-based streaming framework providing a Java and Scala API.

CCS CONCEPTS

• **Information systems → Stream management**;

KEYWORDS

Data Stream Processing, Operator Fission, Operator Fusion, Backpressure, Akka

ACM Reference Format:

Gabriele Mencagli, Patrizio Dazzi, and Nicolò Tonci. 2018. SpinStreams: a Static Optimization Tool for Data Stream Processing Applications. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Data Stream Processing is gaining momentum and has received a great deal of attention owing to the diffusion of systems generating high-speed massive sequences of data items [3]. These systems are characterized by different organizations (ranging from centralized to highly distributed ones) and goals (e.g., real-time analytics that extract insights from raw data, software supporting decision-making processes, environmental monitoring, and many others).

Many different Stream Processing Systems (SPSs), such as Apache Storm [23] and Apache Flink [12], have been conceived, designed and developed across the years. They allow programmers to express applications as *topologies* (i.e. directed graphs) made of *operators* consisting in entities dealing with data processing and transformation [3], which are interconnected by unidirectional data streams. The design of a topology is performed by programmers according to their initial knowledge of the problem. Operators can be either developed on purpose or chosen among already existing ones, possibly adopted in other topologies. As a consequence of the heterogeneous origin of the different operators, the granularity characterizing the topology resulting from their composition may not be the optimal one, leading to the creation of bottlenecks. Moreover, such topology may be very tangled, composed of too many operators, resulting in a substantial overhead without actually improving performance.

To address these issues, some optimizations have been proposed so far. Two of the most promising solutions are: *operator fission* [37] (a.k.a. data parallelism) and *operator fusion* [19]. Fission has been largely and deeply studied. It consists in the replication of stateless operators or stateful ones having a partitionable state. *Operator fusion* is less common and consists in merging underloaded operators to save communication latency and reducing scheduling overhead. How to apply these optimizations is non-trivial. As an example, the choice of the degree of replication, in case of fission, is often in the hands of the programmer who hardly knows the optimal degree to adopt. A possible approach to address this problem is to employ adaptive mechanisms (such as *elasticity*) that dynamically change the degree of replication to efficiently manage variable workloads [17, 22, 35]. Despite their potential, adaptivity and elasticity mechanisms are usually intrusive and require sophisticated strategies to avoid downtimes of running operators [40].

The idea underpinning this paper follows a different approach. We aimed at providing a tool (`SpinStreams`) able to re-engineer streaming applications statically. To achieve this goal, `SpinStreams` applies *cost models* in the form of algorithms driven by profile-based measurements related to processing costs of operators and the probability distributions that model the frequency of data exchange between operators. Thus, by means of our tool, programmers are supported in the complex task of amending the initial topology of the application. `SpinStreams` shows the suggested optimizations along with the predicted outcome, computed by means of the cost models. The initial topology is provided as input using an

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

XML-based formalism and, after the optimization phase, the tool automatically generates the code for a target SPS.

Synthetically, SpinStreams provide the following features:

- for each application imported through the GUI, SpinStreams allows the user to perform a set of analyses: *i)* the evaluation of the steady-state performance in terms of throughput of the streaming application before running it; *ii)* a bottleneck elimination algorithm able to find the most appropriate degree of replication for each operator aimed at balancing the overall performances of the application; *iii)* evaluation of the opportunity to fuse operators: operators that can be fused without hampering the overall performance;
- when the user is satisfied with an optimized topology, the tool automatically generates the code for a target SPS. Although our ambition is to generalize the tool to support a large class of SPSs in the future, in this paper we focus on Akka [2], a widely utilized open source framework.

In presenting SpinStreams, this paper also provides the following research contributions:

- the definition of performance models and optimization algorithms driving the steady-state analysis of the throughput achieved by complex topologies in presence of *backpressure* [9, 10]: a flow control mechanism used in SPSs to stall an upward sending operator until a downward receiver is ready to process new items. Starting from this analysis, the tool provides algorithms to drive operator fission and fusion;
- an extended experimental phase in which SpinStreams has been tested under different conditions, using different kinds of (real-world) operators and constraints.

It is worth pointing out that SpinStreams is not intended to be a replacement of adaptivity supports, which are of paramount importance to deal with variable workloads. Instead, our claim is that SpinStreams can effectively support application designers to find the initial best configuration of their applications before starting the execution on a SPS. The paper is organized as follows. Section 2 introduces the motivating scenarios to clarify the extent and the goals of SpinStreams. Section 3 describes the cost modes and the optimization strategies leveraged by SpinStreams. Section 4 describes our proposed tool and its features. Section 5 reports the results of the evaluation in Akka. Section 6 reviews the relevant scientific literature comparing some key contributions against SpinStreams. Finally, Section 7 draws the conclusion of this paper and presents our ideas about future works.

2 MOTIVATING SCENARIOS

In this section, we describe the process followed by SpinStreams by showing two scenarios of restructuring and optimization strategies applied to streaming topologies.

Operator fission. Pipelining is the simplest form of parallelism. It consists of a chain (or pipeline) of operators. In a pipeline, every distinct operator processes, in parallel, a distinct item; when an operator completes a computation of an item, the result is passed ahead to the following operator. By construction, the throughput of a pipeline equals to the throughput of its slowest operator that represents the *bottleneck*. A technique to eliminate bottlenecks is to apply the so-called *pipelined fission* [16], i.e. to create as many replicas of the operator as needed to match the throughput of faster operators (possibly adopting proper approaches for item scheduling and collection, to preserve the sequential ordering). Figure 1 shows an example of pipelined fission applied to the second operator.

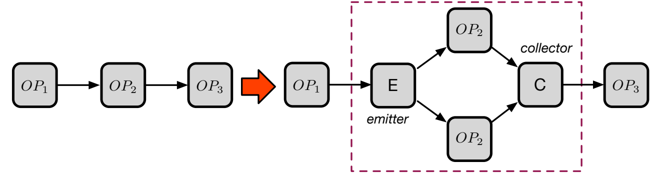


Figure 1: Pipelined fission of the second operator: original topology (left), re-designed topology (right). Depending on the SPS, scheduling and collecting entities can be visible in the runtime only.

The fission technique can be easily applied to any stateless operator by adopting a load-balanced distribution of items among the replicas (e.g., shuffle routing), whereas with stateful operators there are some specific caveats to take into account. Fission can be applied when the state of the original operator can be partitioned, with each partition accessed and modified only by a single replica. This case is known as *partitioned-stateful operators* [14]. Pipelined fission is often applied by programmers by hand, typically by specifying the replication degree through the API provided by the SPS (e.g., the `setParallism()` method in Apache Flink). To do that, *cost models*, in the form of analytical expressions related to specific performance metrics (such as processing times and scheduling costs), can be leveraged by programmers to study the overall expected performance by their applications [32], or other parameters like the overall memory usage [4] and power consumption [25].

Operator fusion. A streaming application could be characterized by a topology aimed at expressing as much parallelism as possible. In principle, this strategy maximizes the chances for its execution in parallel, however, sometimes it can lead to a misuse of operators. In fact, on the one hand, the operator processing logic can be very fine-grained, i.e. much faster than the frequency at which new items arrive for processing. On the other hand, an operator can spend a significant portion of time in trying to dispatch output items to downstream operators, which may be too slow and could not temporarily accept further items (their input buffers are full). This phenomenon is called *backpressure* [9, 10] and recursively propagates to upstream operators up to the sources. Figure 2 shows an example where the topology (left-hand side of the figure) consists of five operators.

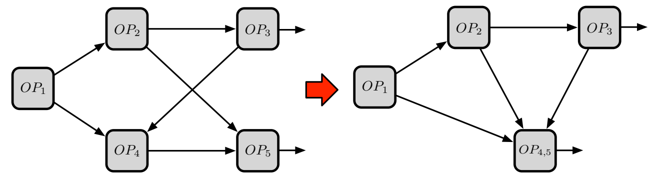


Figure 2: Fusion of $\mathcal{OP}_4$ and $\mathcal{OP}_5$: original topology (left), re-designed topology (right).

As an example, consider the following scenario. If the second operator ($\mathcal{OP}_2$) is a bottleneck, at steady state its input buffer will run out of space; when this happens $\mathcal{OP}_1$ will be delayed due to backpressure. As a consequence, the arrival rate to operator $\mathcal{OP}_4$ and thus to $\mathcal{OP}_5$ will be slower than expected making these operators underutilized (i.e. spending most of time waiting for new input items). If there is no parallelization scheme to apply in order to resolve the bottleneck ($\mathcal{OP}_2$), then an optimization is to fuse $\mathcal{OP}_4$ and $\mathcal{OP}_5$ together (right-hand side of the figure). The new operator, resulting from the fusion, receives items from $\mathcal{OP}_1$, $\mathcal{OP}_2$ or $\mathcal{OP}_3$, namely the operators in its input neighborhood. In case of items from $\mathcal{OP}_1$ or $\mathcal{OP}_3$, the business code associated with operators $\mathcal{OP}_4$ and $\mathcal{OP}_5$ are applied sequentially. In case of items from $\mathcal{OP}_2$, only the logic of $\mathcal{OP}_5$ (the original target) is applied, ensuring the semantic equivalence with respect to the initial topology.

As discussed in Section 6, operator fusion is less explored than fission. Most of the existing solutions apply fusion at a different semantics level than `SpinStreams` [18]. In fact, `SpinStreams` works at the programming model level: once operators are fused as a result of an optimization decided by the user, such operators are deployed and executed by the runtime system which is no longer aware that they were originally decomposed in some fine-grained operators. In COLA [24] instead, multiple operators can be mapped onto the same deployment unit to reduce communication costs, but they remain logically separated entities in the runtime system with a non-null, though small, intra-unit communication latency.

For the sake of completeness, it is worth noting that backpressure is not the only possible communication semantics in stream processing. A common alternative is to apply *load shedding* [33] to prevent the streaming buffers to indefinitely grow by discarding input items. However, data loss is not always acceptable in every context (e.g., for applications requiring the exactly-once semantics [1]). Therefore, backpressure is definitely the most diffused approach to deal with traffic spikes in SPSs.

3 PERFORMANCE MODELS

This section presents the cost models adopted by `SpinStreams` to analyze streaming topologies. Their ultimate goal is to study the performance in terms of *throughput* (number of input items ingested by the topology per time unit) achieved at steady state, i.e. when the system has worked for a sufficiently long period so that initial conditions do not longer affect the actual performance.

`SpinStreams` leverages specific algorithms that work on an abstract representation of streaming application topologies modeled as *queueing networks*. Each operator of the original topology is represented by a sequential entity (single replica) working as a queueing station, i.e. it receives input items by ingoing edges (modeling data streams) from antecedent operators, and performs a user-defined function on each received item by delivering zero, one or many output items to its outgoing edges. For the sake of presentation, *we start assuming that each operator produces one output item per input item consumed.* Then, we remove this constraint to model more general computations. We assume that backpressure is controlled by adapting the operators' throughput. In the Queueing Theory literature this approach is called *Blocking After Service* (BAS) [31]: when an output item attempts to enter into a full queue, that item is blocked until a free slot becomes available in the buffer. During this phase the sending operator is unable to process any other input item until the backpressure is deasserted.

3.1 Steady-State Analysis

Let $\mathcal{G} = (\langle \mathcal{V}, \mathcal{E} \rangle)$ be a graph representing the topology. Each vertex is an operator $\mathcal{OP}_i$ characterized by a *service rate* μ_i, i.e. the average number of input items that the operator can serve per time unit assuming that there are always items to process. This value can be obtained by measuring the average computation time per input item and the communication latency spent to send the result.

Goal. The goal of steady-state analysis is to design an algorithm that having $\mathcal{G}$ produces a new graph $\mathcal{G}'$ where each vertex $\mathcal{OP}_i \in \mathcal{V}$ is labeled with its steady-state *departure rate* δ_i. The departure rate is the average number of results delivered per time unit onto any output edge at steady state, but unlike the *service rate* it takes into accounts also bottlenecks slowing down the overall performance of the operator via backpressure.

Assumptions. As a matter of fact, this problem is complex to be solved in the general case as the steady-state behavior of the graph depends on many factors like the statistical distributions of the operators' service rates and of the external arrivals to the streaming application, the size of the buffers implementing the streams and the topological properties of the graph. As a consequence, `SpinStreams` makes the following assumptions:

- only *rooted acyclic graphs* can be analyzed, that is graphs with a single source (i.e. a vertex without input edges). Furthermore, we assume that every vertex is reachable from the source (the graph is a *flow graph*);
- when an operator has multiple input edges, items are received and processed in FIFO order[1]. In case of multiple output edges, each output item is delivered to one of the possible destinations according to a given probability. Such probability depends on the application semantics and it is assumed to be known and measurable;
- all the buffers of an operator have a fixed maximum capacity and are used according to the BAS semantics.

These assumptions allow a simple but precise performance modeling based on an intuitive *flow conservation* principle: at steady state the rate at which input items arrive at an operator is the same rate at which results leave the operator. This condition is always valid regardless of the statistical distributions of the service rates (e.g., Poisson, Normal or Deterministic). Furthermore, while the single source assumption can be circumvented by adding a fictitious source operator in the topology linked to the real sources, the acyclicity assumption is presently needed by our cost models. Of course, solutions to extend the generality of `SpinStreams` are our primary future research direction. The cost model is implemented as an algorithm shows in the next part.

Algorithm. The steady-state analysis requires an ordered visit such that when a particular vertex (operator) is visited, all the

[1] FIFO is the most widely used policy. Other semantics, like the one with priorities, are possible but not modeled by `SpinStreams` yet.

operators sending items to it have been explored and their steady-state departure rate computed. Such ordering is *topological* and is obtainable through a depth-first search of the graph.

Let $\mathcal{OP}_i$ be the i-th vertex in the input topological ordering and $\mathcal{OP}_1$ is the unique source operator. The vertices are visited from the source by following that ordering. When a generic vertex $\mathcal{OP}_i$ is visited, the algorithm computes its arrival rate λ_i as the sum of the departure rates of the vertices in its incoming neighborhood, each properly weighted by the probability of the corresponding edge. The *utilization factor* $\rho_i \in [0, 1]$ is computed as λ_i/μ_i and it is used to determine whether the vertex is a bottleneck or not:

- if $\rho_i \leq 1$ the operator is *not a bottleneck*. Its departure rate equals the arrival rate and we proceed with the next vertex;
- if $\rho_i > 1$ the operator is *a bottleneck*. At steady state, its input buffer grows reaching the maximum capacity and the upstream operators are delayed due to backpressure.

In queueing networks with blocking, $\rho_i > 1$ is a *transient condition* [31]. When $\rho_i > 1$, at steady state the departure rates of the other stations adapt to the rate of the bottleneck, leading its utilization factor to be not greater than one. This condition reflects in the following invariant:

Invariant 3.1. *When the algorithm visits the i-th vertex, all the operators $\mathcal{OP}_j$ such that $j < i$ have $\rho_j \leq 1$.*

The idea is to correct (lower) the departure rate from the unique source in order to take into account the backpressure generated by the bottleneck operator. To do that we use the following result:

Theorem 3.2. *Let $\mathcal{OP}_i$ be the bottleneck with $\rho_i > 1$ and δ_1 the departure rate from the source computed before visiting $\mathcal{OP}_i$. The departure rate δ_1' capturing the backpressure caused by $\mathcal{OP}_i$ is computed as $\delta_1' = \delta_1/\rho_i$.*

Proof. By invariant, all the vertices $\mathcal{OP}_j$ with $j < i$ have $\rho_j \leq 1$, thus $\delta_j = \lambda_j$. This allows us to write the arrival rate to the bottleneck operator as a function of the current departure rate from the source by taking into account all the possible paths from $\mathcal{OP}_1$ to $\mathcal{OP}_i$. Figure 3 shows an example where there exist two paths π_1 and π_2.

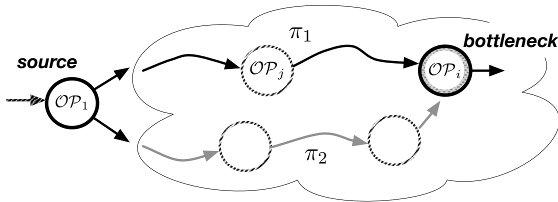


Figure 3: Correction of the source departure rate when a new bottleneck is discovered in the analysis.

We denote by $\mathcal{P}(i)$ the set of all the paths from the source to $\mathcal{OP}_i$, and a path π is a set of pairs (u, v) such that an edge from $\mathcal{OP}_u$ to $\mathcal{OP}_v$ exists in $\mathcal{E}$. The arrival rate to $\mathcal{OP}_i$ is:

$$\lambda_i = \delta_1 \cdot \sum_{\pi \in \mathcal{P}(i)} \left[\prod_{(u,v) \in \pi} p_{(u,v)} \right] \tag{1}$$

where $p_{(u,v)}$ is the probability of the edge. We lower the departure rate from the source to account for the backpressure caused by $\mathcal{OP}_i$. By flow conservation, the new departure rate δ_1' is such that the new arrival rate to the bottleneck equals its service rate:

$$\delta_1' \cdot \sum_{\pi \in \mathcal{P}(i)} \left[\prod_{(u,v) \in \pi} p_{(u,v)} \right] = \mu_i \tag{2}$$

The new departure rate can be written as $\delta_1' = \delta_1 \cdot \alpha$, that is the product of the previous departure rate with a *corrective factor* $0 < \alpha < 1$. We obtain:

$$\delta_1\, \alpha \cdot \sum_{\pi \in \mathcal{P}(i)} \left[\prod_{(u,v) \in \pi} p_{(u,v)} \right] = \mu_i$$

$$\delta_1 \cdot \sum_{\pi \in \mathcal{P}(i)} \left[\prod_{(u,v) \in \pi} p_{(u,v)} \right] = \frac{\mu_i}{\alpha}$$

$$\lambda_i = \frac{\mu_i}{\alpha} \tag{3}$$

Therefore, the corrective factor is the inverse of the utilization factor of the bottleneck operator, i.e. $\alpha = \dfrac{1}{\rho_i}$. □

With a single source, there is a unique way to tune the departure rate of such source by respecting the flow conservation property. Instead, in case of multiple-source graphs, there would be infinite ways to change the departure rates of the sources, and the final steady-state graph would depend on the statistical distributions of the service rates. This further explains the importance of the single-source constraint to ease the evaluation without knowing the statistical distributions modeling the behavior of the operators.

By applying Theorem 3.2 the invariance is maintained as explained below:

Proposition 3.3 (Invariant Maintenance). *By applying the result of Theorem 3.2 the Invariant 3.1 is preserved.*

Proof. The algorithm corrects the departure rate of the source as stated by Theorem 3.2 and restarts the graph traversal from the beginning. After the correction, when vertex $\mathcal{OP}_i$ is visited again, its utilization factor is $\rho_i = 1$. □

Algorithm 1 takes as input the graph $\mathcal{G}$ and a topological ordering, and returns the graph labeled with the steady-state utilization factors and departure rates. We denote by $IN(i)$ the set of the indices of vertices in $\mathcal{V}$ having an output edge directed to $\mathcal{OP}_i$.

Proposition 3.4 (Complexity). *The worst-case time complexity of Algorithm 1 is $O(|\mathcal{V}| \cdot |\mathcal{E}|)$.*

Proof. The algorithm runs several partial traversals of the graph. In the worst-case scenario, the algorithm finds a bottleneck every time a vertex is visited for the first time. Therefore, the number of partial traversals in the worst case is $|\mathcal{V}| = n$. At the very beginning, the algorithm visits two vertices, the source and the first bottleneck. In the second, according to Proposition 3.3, the algorithm visits three vertices, and so forth. The total number of times the operators are visited is:

$$N_{visited} = 2 + 3 + \ldots + 2n = \frac{n^2 + 3n - 2}{2} \sim O(n^2) \tag{4}$$

Algorithm 1: Steady-State Analysis

Input: a graph $\mathcal{G} = (\langle \mathcal{V}, \mathcal{E} \rangle)$ and a topological ordering $\{\mathcal{OP}_1, \ldots, \mathcal{OP}_{|\mathcal{V}|}\}$
Result: the graph $\mathcal{G}'$ with the final departure rate per operator

```
1:  δ_1 ← μ_1
2:  ρ_1 ← 1
3:  i ← 2
4:  while i ≤ |V| do
5:      λ_i ← Σ_{j∈IN(i)} (δ_j · p_(j,i))
6:      ρ_i ← λ_i/μ_i
7:      if ρ_i ≤ 1 then                      ▷ no bottleneck
8:          δ_i ← λ_i
9:          i ← i + 1
10:     else                                 ▷ bottleneck
11:         δ_1 ← δ_1/ρ_i
12:         ρ_1 ← δ_1/μ_1
13:         i ← 2
```

Consequently, each vertex is visited $O(n)$ times in the worst case and the algorithm scans the list of its input edges (line 5) to compute the arrival rate. Therefore, each edge (i, j) is traversed every time $\mathcal{OP}_j$ is visited and so the worst-case complexity is $O(|\mathcal{V}| \cdot |\mathcal{E}|)$. □

An interesting and intuitive result can be derived by analyzing the departure rates of the source and of the *sink* operators, i.e. the vertices without output edges:

Proposition 3.5. *In the output graph provided as result by Algorithm 1, the departure rate from the source operator equals the total departure rate from the sinks.*

Proof. From Invariant 3.1 follows that at the end of Algorithm 1, for each operator $\mathcal{OP}_i$, the utilization factor is less or equal than 1. Let $\mathcal{S} \subseteq \mathcal{V}$ be the set of the sinks as shown in Figure 4.

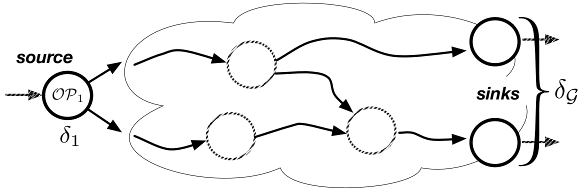


Figure 4: Steady-state behavior of the source and the sink operators.

The throughput of the whole topology is the sum of the departure rates from the sink operators, i.e. $\delta_{\mathcal{G}} = \sum_{\mathcal{OP}_i \in \mathcal{S}} \delta_i$. Since at the steady state each operator has utilization factor less or equal than 1, the departure rate from each sink can be determined as a function of the departure rate of the source:

$$\begin{aligned} \delta_{\mathcal{G}} &= \sum_{\mathcal{OP}_i \in \mathcal{S}} \left[\delta_1 \cdot \sum_{\pi \in \mathcal{P}(i)} \left(\prod_{(u,v) \in \pi} p_{(u,v)} \right) \right] \\ &= \delta_1 \cdot \sum_{\mathcal{OP}_i \in \mathcal{S}} \left[\sum_{\pi \in \mathcal{P}(i)} \left(\prod_{(u,v) \in \pi} p_{(u,v)} \right) \right] \end{aligned} \quad (5)$$

The second term of the product, that is the summation $\sum_{\mathcal{OP}_i \in \mathcal{S}}[\cdot]$, is the sum of the probabilities of all the paths from the source to the sinks, which according to our assumptions (every vertex is reachable from the source) is equal to 1. □

3.2 Bottleneck Elimination

Bottlenecks are eliminated via operator fission. We can determine the minimum degree of replication to unblock the bottleneck:

Definition 1 (Optimal Replication Degree). *Given an operator $\mathcal{OP}_i$ such that $\rho_i > 1$, the optimal number of replicas is computed as $n_i^{opt} = \lceil \rho_i \rceil$.*

The optimal degree of replication is computed under the assumption that the input flow of items can be evenly split among replicas; a condition easy to achieve with stateless operators. For partitioned-stateful operators [3], each replica is in charge of processing all the input items having the same value for a partitioning key attribute. Therefore, an even distribution can be achieved if the key domain is sufficiently large and the key frequency distribution not so skewed.

Goal. We want to design a systematic procedure to analyze a topology by: *i)* visiting the vertices and unblocking bottlenecks through operator fission; *ii)* when a bottleneck cannot be eliminated, the procedure evaluates the induced backpressure on the graph to adjust the replication degree of other vertices, accordingly.

Algorithm. Algorithm 2 proceeds by visiting the vertices in the input topological ordering. For each operator, the arrival rate and utilization factor is computed similarly as in Algorithm 1. If the operator is not a bottleneck (line 29), the algorithm passes to the next vertex. Otherwise, we properly react distinguishing among three different cases: stateless, partitioned-stateful or stateful operators. If the operator is stateless (line 8), the optimal replication degree is computed in line 9 unblocking the bottleneck and the algorithm moves to the next operator.

If the operator is partitioned-stateful (line 13), to each replica is assigned a subset of the partitioning keys. This logic is encapsulated in the call `KeyPartitioning()` that gets as input the set of keys $\mathcal{K}$ and their distribution frequencies $\{p_k\}_{k=1}^{|\mathcal{K}|}$. The call assigns keys to replicas such that the most loaded replica receives a fraction of the input items as close as possible to $1/n_i^{opt}$. To do that, some heuristics can be used to make the problem tractable like the ones in [14] (e.g., based on *consistent hashing* and its variants for addressing skewed distributions). As a result, the operator is parallelized with $\overline{n}_i \leq n_i^{opt}$ replicas where the most loaded one receives a fraction p^{max} of the incoming items (hopefully, $1/n_i^{opt}$). Once the parallelization is applied, it is possible that the operator is still a bottleneck (line 17). This may happen when the probability distribution is too skewed. For example, if the operator needs to be parallelized with $n_i^{opt} = 3$ replicas and 50% of the items have the same key, the bottleneck can be mitigated but not removed. In that case, the algorithm uses $\overline{n}_i = 2$ replicas and corrects the departure rate of the source according to Theorem 3.2 by restarting the analysis. Otherwise, when the bottleneck can be completely removed, the departure rate matches the arrival rate and the algorithm proceeds with the next vertex.

Finally, the last case is when the operator is stateful (line 24) and so fission cannot be used. We proceed as in Algorithm 1 by correcting the source's departure rate and restarting the visit from the beginning. In conclusion, the algorithm follows the same reasoning of the steady-state analysis and restarts the analysis from the source only when a bottleneck cannot be eliminated. Therefore, the worst-case time complexity is the same of Algorithm 1.

Algorithm 2: Bottleneck Elimination

```
Input: a graph G = (⟨V, E⟩) and a topological ordering {OP_1, ..., OP_|V|}
Result: the graph G' with the departure rate and the repl. degree per operator
 1: δ_1 ← μ_1
 2: ρ_1 ← n_1 ← 1
 3: i ← 2
 4: while i ≤ |V| do
 5:   λ_i ← Σ_{j∈IN(i)} (δ_j · p_(j,i))
 6:   ρ_i ← λ_i/μ_i
 7:   if ρ_i > 1 then                                         ▷ bottleneck
 8:     if OP_i is stateless then
 9:       n_i ← ⌈ρ_i⌉
10:       ρ_i ← λ_i / (μ_i·n_i)
11:       δ_i ← λ_i
12:       i ← i + 1
13:     if OP_i is partitioned-stateful then
14:       (n̄_i, p^max) ← KeyPartitioning(K, {p_k}_{k=1}^{|K|}, ρ_i)
15:       n_i ← n̄_i
16:       ρ_i ← (λ_i·p^max) / μ_i
17:       if ρ_i > 1 then                                     ▷ still bottleneck
18:         δ_1 ← δ_1/ρ_i
19:         ρ_1 ← δ_1/μ_1
20:         i ← 2
21:       else
22:         δ_i ← λ_i
23:         i ← i + 1
24:     if OP_i is stateful then
25:       n_i ← 1
26:       δ_1 ← δ_1/ρ_i
27:       ρ_1 ← δ_1/μ_1
28:       i ← 2
29:   else                                                    ▷ no bottleneck
30:     n_i ← 1
31:     δ_i ← λ_i
32:     i ← i + 1
```

Hold-off replication. Before the bottleneck elimination phase, the user can indicate a maximum boundary in terms of total amount of replicas to parallelize the topology. `SpinStreams` adopts a heuristic solution since the problem of mapping graphs onto graphs is generally NP-hard. Let $N = \sum_{i \in \mathcal{V}} n_i$ be the total number of replicas used in the optimized topology after the running of Algorithm 2, and let N_{max} be the maximum upper bound provided by the user. If $N_{max} < N$, `SpinStreams` computes a *reduction factor* $r > 0$ as $r = N_{max}/N$ and each replication degree is multiplied by r in order to obtain the degree to be used. Of course, this simple heuristics works well with large values of N and N_{max} and small rounding effects can lead to some anomalies, that can force adjustments of few units to the replication degrees.

3.3 Operator Fusion

As mentioned above, a too tangled and fine-grained structure of the streaming application may lead to inefficient exploitation of resources and cause notable scheduling overheads. When this happens, we try to merge different operators into a single functionally-equivalent operator.

Goal and assumptions. Given a topology $\mathcal{G} = (\langle \mathcal{V}, \mathcal{E} \rangle)$, a sub-graph of operators $\mathcal{G} = (\langle \mathcal{V}^{sub}, \mathcal{E}^{sub} \rangle)$ has $\mathcal{V}^{sub} \subseteq \mathcal{V}$ and the set of all the edges $\mathcal{E}^{sub} \subseteq \mathcal{E}$ connecting only those vertices in $\mathcal{V}^{sub}$. The objective is to replace the sub-graph with a single operator $\overline{\mathcal{OP}}$ semantically equivalent and evaluate the resulting performance. This problem is very complex to solve in the general case, thus `SpinStreams` makes some assumptions limiting the applicability of the fusion operation. However, such solutions are more general than most of the approaches presented so far in the literature [19]. The sub-graphs candidates for fusion must respect the following constraints:

- they must have a single *front-end vertex*: a unique operator in $\mathcal{V}^{sub}$ having at least one input edge originated from a vertex in $\mathcal{V} \setminus \mathcal{V}^{sub}$;
- the new topology obtained by replacing the sub-graph $\mathcal{G}^{sub}$ with $\overline{\mathcal{OP}}$ must still be acyclic.

Our goal is to evaluate the service rate of the new operator in order to run the steady-state analysis (cf. Algorithm 1) and check whether the *operator fusion* hampers performance.

Algorithm. The first step is to evaluate the service rate of the new operator $\overline{\mathcal{OP}}$. This is done by the function `fusionRate()` in Algorithm 3, which takes as inputs the sub-graph $\mathcal{G}^{sub}$ and its unique front-end vertex.

As hinted in Section 2, each input item arriving at the sub-graph will travel a path from the front-end vertex up to one of the ending vertices in the subgraph, which delivers the result out to the rest of the topology. The service rate of $\overline{\mathcal{OP}}$ can be computed as the weighted average of the aggregate service rates of all the paths where the weight is the path probability computed as the product of the probabilities of the traveled edges. The aggregate service rate of a path is defined as follows:

Definition 2 (Aggregate Service Rate). *Let $\pi = (\mathcal{OP}_i, \mathcal{OP}_{i+1}, \ldots, \mathcal{OP}_n)$ be a path starting from the front-end operator $\mathcal{OP}_i$ and reaching an ending vertex $\mathcal{OP}_n$ of the sub-graph. The aggregate service rate of π is $\mu_\pi = (\sum_{j=i}^{n} \mu_j^{-1})^{-1}$.*

This idea is applied in Algorithm 3, where we use $\mathcal{T}_i$ to denote the *service time* of $\mathcal{OP}_i$ defined as $\mathcal{T}_i = \mu_i^{-1}$ and $OUT(i)$ is the set of the indices of the vertices of $\mathcal{V}^{sub}$ having an input edge originated from $\mathcal{OP}_i$. In the worst-case the complexity is exponential in the number of vertices because we have to enumerate all the possible distinct paths. However, since most of the stream processing topologies have usually tens of operators [20], this does not represent a real practical concern.

Algorithm 3: `fusionRate()` function

```
Input: the sub-graph to be fused G^sub and one of its vertices OP_i
Result: the service time T_op of the operator OP replacing G^sub
 1: T_op ← 0
 2: for each j ∈ OUT(i) do
 3:   T_op ← T_op + p_(i,j) · fusionRate(G^sub, OP_j)
    return T_op
```

The algorithm is initially invoked with the sub-graph and its front-end vertex as input parameters, and it is used to estimate the service rate of $\overline{\mathcal{OP}}$. Then, the new topology is built where $\mathcal{G}^{sub}$ is replaced by $\overline{\mathcal{OP}}$. In this phase, it is possible that some edges directed to the same operator in the new topology had starting vertices that have been replaced by the operator $\overline{\mathcal{OP}}$. In that case, those edges are fused and their joint probability computed. Finally, the steady-state analysis (cf. Algorithm 1) is run to evaluate the performance,

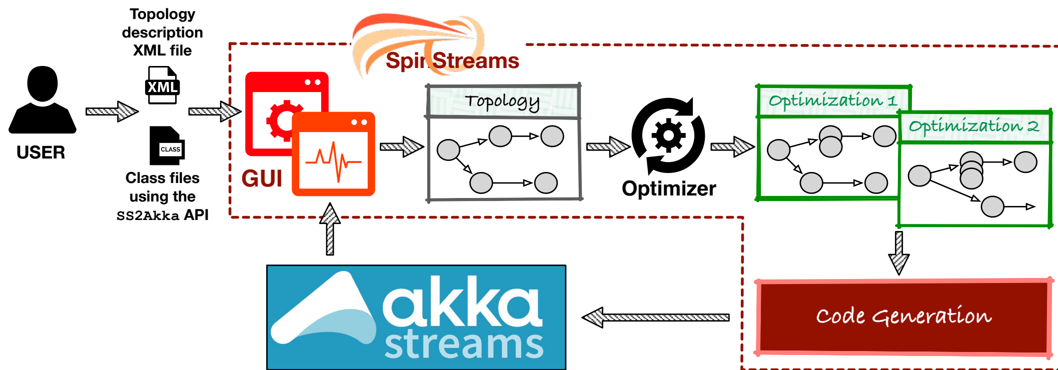


Figure 5: Conceptual workflow of `SpinStreams`: input topology (XML file and `.class` files based on the `SStoAkka` API), optimizations of the topology and generation of the code to be run by the `Akka` run-time system.

and the user is informed whether fusion impairs performance or not (i.e. if the new operator is a bottleneck).

Operator fusion leaves some aspects open. In particular, the code of the new operator should be automatically generated from the code of the original operators. This will be discussed in Section 4.

3.4 Extensions

So far, we assumed that operators produce one output item per input item consumed. Although this holds for a large set of operators like map and projection, it is not suitable to model operators like selection, joins and windowed operators where the relationship between items received and items consumed is more general.

In the stream processing literature, this relationship is controlled by two parameters of an operator, that is its *input* and *output selectivity* [3]. Input selectivity is the number of input items consumed before returning a new output item. This is the case of sliding-window computations [13] (e.g., aggregates), where the operator applies an internal processing function over the last $w > 0$ items received and repeats the processing every $s > 0$ new items. Instead, the output selectivity is a parameter stating the number of output items produced per input item. For example, in case of operators like flatmap, selection and joins either zero, one or more items are produced per input. These two cases can be covered in Algorithms 1, 2 and 3 as follows:

- ***input selectivity***: let $\mathcal{OP}_i$ be an operator with input selectivity $s > 0$, measured as the average number of input items to consume before producing a new output result. The departure rate from the operator can be computed as $\delta_i = \min\{\lambda_i, \mu_i\}/s$ while the utilization factor is still computed as $\rho_i = \lambda_i/\mu_i$;
- ***output selectivity***: let $\mathcal{OP}_i$ be an operator with output selectivity $s > 0$, that is the average number of output items produced per input item. The departure rate is $\delta_i = \min\{\lambda_i, \mu_i\} \cdot s$ with utilization factor sill equal to $\rho_i = \lambda_i/\mu_i$.

All the `SpinStreams` algorithms can be easily generalized to the case of operators with different selectivity parameters by computing the departure rate as discussed before.

4 SPINSTREAMS

In this section, we describe the general workflow followed by a programmer using `SpinStreams` and how our tool generates the code to be run on `Akka`[2].

4.1 Optimization Workflow

The entire workflow is summarized in Figure 5. The first step is to start the GUI by providing as input the application topology. We expect that the user knows some profiling measures, like the processing time spent on average by the operators to consume input items, the probabilities associated with the edges of the topology, and the operator selectivity parameters. This information can be obtained by executing the application *as is* for a reasonable amount of time and by instrumenting the code to collect profiling measures. To do that, some libraries exist to profile stream processing topologies in various languages, such as `Mammut` [36] for C++ programs and `DiSL` for Java [28, 34]. The main inputs to `SpinStreams` are:

- the structure of the topology and the profiling measurements expressed in an XML file. The syntax provides tags to specify the operators, with attributes for their name, the service rate (specifying the time unit), the pathname of the class file, the type (stateless, stateful, partitioned-stateful with the number of keys and the file with their probability distributions). Other tags specify the output edges and their probability, and the input/output selectivity;
- along with the XML file, the user provides, for each operator, a `.class` file obtained by compiling a source code written using a specific API. Such API is provided to allow the automatic code generation from the abstract representation used in `SpinStreams` to the code to be run on the target SPS. For `Akka` this API is called `SS2Akka`.

`SpinStreams` checks if the input topology satisfies the constraints (acyclicity and rooted graph) before creating a new *imported entry* that will contain all the versions prototyped for the topology. As a first step, the user can run the steady-state analysis (cf. Algorithm 1) which produces further annotations for the operators (their departure rates) and the predicted throughput of the

[2]The source code of the project will be made available at https://github.com/ParaGroup

application. After, the user can request SpinStreams to introduce some specific optimizations:

- identify and remove bottlenecks by means of Algorithm 2. The GUI is updated by opening a new tab with the result of the optimization–a new topology where operators are shown with the needed degree of replication;
- try a fusion optimization by selecting sub-regions of the graph. SpinStreams proposes a set of candidates after the steady-state analysis, ranked by their utilization factor in order to ease the process of selection of the sub-graph to be fused. Once chosen, the user starts the fusion optimization that produces a new topology whose performance at steady state is predicted and reported to the user. All the integrity checks are performed automatically before starting this process (i.e. single front-end operator and acyclicity).

After these steps, if the user is satisfied with one of the proposed optimizations, the code generation phase can start, the code for Akka is generated and eventually executed on the local machine. Real performance results are delivered to the user in a console opened by the SpinStreams GUI in order to provide an immediate feedback.

4.2 SS2Akka and Code Generation

SS2Akka is written in Java using generic programming. For each operator the user provides a class extending the Operator abstract class by overriding the operatorFunction() method. The method takes as input a data item of type Item (a template parameter) and returns an object of type WrappedItem. Other overloaded definitions take as input a *collection* of Item objects and return a *collection* of WrappedItem instances to model operators with different selectivity. The methods are non-static, so they can use an internal state (based on the operator type passed to the constructor).

The API decouples the abstract topology from the real implementation. Akka [2] is a framework for concurrent programs in Java and Scala. It is based on the actor-based model, where the computation is a graph of actors whose execution is triggered by the arrival of new messages in their mailboxes (finite buffers). Actors exchange messages asynchronously and are executed by a pool of threads in the run-time system, which guarantees that different invocations of an actor (for different messages) can never be executed concurrently. In SpinStreams, to further raise the abstraction level, actors in Akka are treated as *executors* of our operators, as shown in Figure 6. In the following of this section are listed the different cases taken into account for the code generation.

Generation of standard operators. In the standard case (i.e. operators with a single replica or stateful ones), an operator is executed by a dedicated actor in Akka. For each input item received in its mailbox, the actor calls the operatorFunction() method of the *Operator* instance to which it was assigned during the construction. The result data type WrappedItem is a template class encapsulating an object of type Item (the real massage) and the unique identifier of the destination operator. After the processing, the actor is responsible for forwarding the result message to the proper mailbox of the actor executing the destination operator.

Generation of parallel operators. In case of operators with replication degree greater than one, an actor is created for each

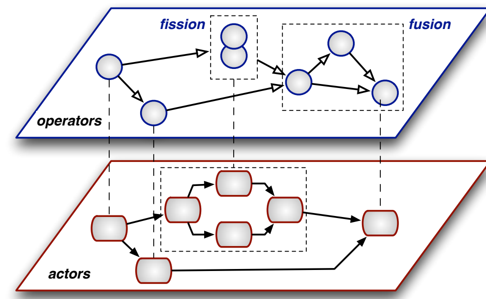


Figure 6: Abstraction layer on top of the Akka programming model. Actors are executors of abstract operators of the topology.

replica working as described at the previous point. Furthermore, two additional actors are created to do the scheduling of input items (*emitter*) to the replicas and to collect their results (*collector*). For stateless operators, items are distributed in a circular manner. In case of partitioned-stateful operators, the emitter actor does the distribution using the hash function provided to the constructor of the corresponding Operator instance. Such actors are in general fast as they execute single point-to-point communications. However, in principle with very fast input rate, they could become bottlenecks. When this happens, Algorithm 2 can be extended to check this condition in order to prevent new replicas to be uselessly added.

Generation with operator fusion. The decoupling between operators and actors allows the code generation of fused operators, which incorporate original fine-grained operators of the initial topology. This operator is represented in SpinStreams as a *meta-operator*, which includes the references to the original operators of the fused sub-graph. The meta-operator is executed by a *single* actor whose pseudo-code is shown in Algorithm 4. Since in the sub-graph there must be a unique front-end operator, each input message in the actor's mailbox is processed by executing the operatorFunction() method of that operator. If the result is headed to another operator in the same sub-graph, its operatorFunction() method is called by the actor and the reasoning process is re-iterated. Once the result is headed outside the sub-graph, the actor sends the data item to the mailbox of the corresponding actor. A similar behavior, although more complex to be shortly described, is likewise performed by operators with input and output selectivity greater than one.

Algorithm 4: Akka actor executing a meta-operator

1: **for each** *item* in the actor's mailbox **do**
2: $< dest, msg > \leftarrow$ frontEndOp.operatorFunction(*item*)
3: **while** *dest* is a vertex of the sub-graph $\mathcal{G}^{sub}$ **do**
4: $< dest, msg > \leftarrow$ *dest*.operatorFunction(*msg*)
5: **enqueue** *msg* in *dest*'s mailbox

We recall that the sub-graph is acyclic by construction, so the algorithm always terminates. Furthermore, in SpinStreams is not possible to apply fission to meta-operators, since the user is interested in merging under-utilized operators and this is effective if this does not introduce a new bottleneck.

It is worth noting that SpinStreams generates code using the akka-actor library, which represents the common layer shared by several other libraries provided by the Akka toolkit for specific purposes. One of these is akka-streams, a library aimed at reducing the programming effort in writing streaming applications. Like our SS2Akka, the library decouples processing stages and actors in the runtime. Interestingly, in akka-streams all the processing stages are by default executed by a single actor, unless this behavior is not explicitly changed by adding proper *asynchronous boundaries* to separate groups of stages to execute concurrently. In SpinStreams we did not make use of akka-streams since the execution semantics of stage fusion is different than the one expected by our steady-state algorithms. In fact, in akka-streams the processing stages assigned to the same actor are executed and process input items in any order that preserves the data dependencies. In contrast, in our semantics, for each input item popped from the actor's mailbox, the execution in Algorithm 4 is functionally equivalent to the sequential composition of the processing functions called along the path followed by the item in the sub-graph, a semantics that allows the service rate of the meta-operator to be easily predicted by static cost models.

5 EVALUATION

In this final part of the paper, we will evaluate SpinStreams by using a set of streaming topologies made of real-world operators with user-defined functions. The goal of this analysis is to assess the accuracy and the viability of the cost models and the optimizations presented in Section 3, which are the core part of the methodology proposed by SpinStreams.

5.1 Experimental Setup

To comprehensively evaluate SpinStreams and its cost models, we need a large set of applications having heterogeneous features in terms of topologies (connections among operators), and bottlenecks with various choices of where they are located in the graph (closer of farther from the source). To this end, we developed 20 different real-world operators. Some of them are stateless operators like filters and maps, which apply transformations on a tuple-by-tuple basis (a *tuple* is an item representing as record of attributes). We also implemented stateful operators based on count-based windows for aggregation tasks (i.e. weighted moving average, sum, max, min and quantiles), spatial queries (i.e. skyline and top-k [38]) and join operators performing band-join predicates on count-based windows. We shuffled and randomly connected subsets of those operators turning out into 50 acyclic topologies which represent our testbed. It should be noted that, although operators are connected randomly, their workload and computational features are fully realistic and based on existing operators.

Algorithm 5 presents the procedure used to randomly generate a topology and that will be used to build our testbed. It takes as inputs the amount of vertices V and the expected number of edges E to be generated. The algorithm may generate a slightly greater number of edges than E, as it will described shortly. The value V is chosen randomly in the interval $[2, 20]$. E is set equal to $(V-1)\cdot\beta$, where β is the connecting factor randomly generated in the range $[1, 1.2]$ according to the discrete uniform distribution in order to obtain quite sparse graphs that result in topologies of loosely coupled operators. Indeed, to the best of our knowledge, this is the most common type of topologies for streaming applications.

The algorithm starts generating the V vertices that are numbered with a progressive identifier starting from zero. This ordering will be a topological one of the final graph. The first vertex (index 0) is the source. In the first phase (line 5), $V-1$ random edges are generated by respecting the topological ordering, i.e. a generated edge (i, j) will have $i < j$. Then, the algorithm generates the remaining number of edges (line 8) up to E connecting random pairs of vertices respecting the topological ordering in order to not introduce any cycle. In the pseudo-code the call randInt(a,b) returns a random integer uniformly distributed in the interval $[a..b]$.

Following the aforementioned algorithm, it could happen that a certain set of vertices S, not intended to be the source, will end up without any input edge. When this happens, additional edges (line 13) are added to the graph, connecting the source with the vertices belonging to S. This can increase the number of edges that may be slightly greater than E.

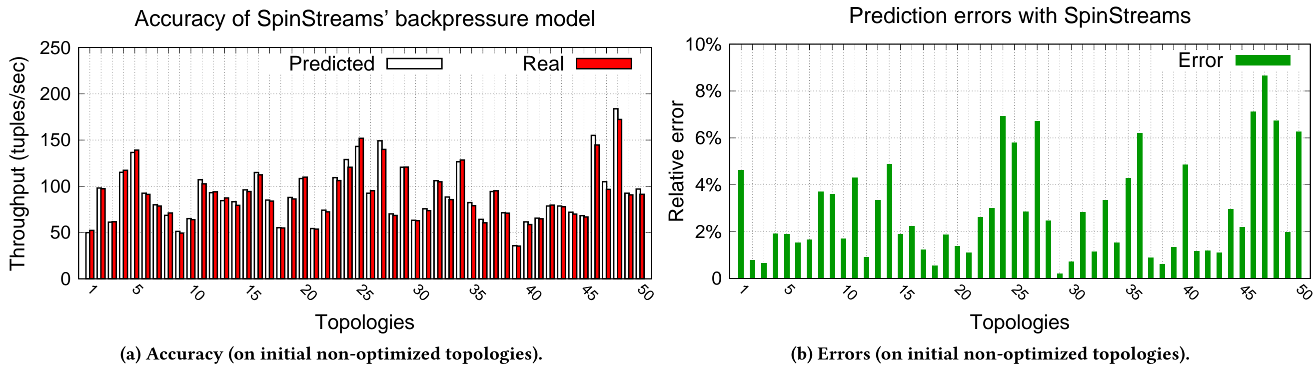
In the last step, vertices are assigned to real-world operators (line 16). Except for the source operator that is in charge of generating the input stream, operators are assigned randomly but taking into account possible constraints limiting the compliance with certain assignments, e.g. joins can be assigned only to vertices with more than one input edge. Once the topology is built, a discrete probability distribution is assigned to each vertex having multiple output edges (not shown in the pseudocode for brevity). We generate those probabilities using a power-law model (*ZipF* distribution) with a scaling exponent $\alpha > 1$ generated randomly in order to obtain distributions with different skewness.

Target machine and software used. The experiments have been executed on a machine equipped with two Intel Xeon E5-2695 @2.40GHz CPUs, running CentOS Linux release 7.2.1511. We use Akka version 2.5.12, configured to implement the communication semantics modeled by SpinStreams. Each actor uses the

Algorithm 5: Generation of a random topology

Input: the number of vertices V and of expected edges E to be generated
Result: a random topology $\mathcal{G}$ respecting the SpinStreams's constraints

```
1:  if E > V(V-1)/2 then
2:      raise_an_error ("too many edges")
3:  if E < (V-1) then
4:      raise_an_error ("too few edges")
                                          ▷ create edges
5:  for i ← 0 to V-2 do
6:      v ← randInt(i+1, V-1)
7:      Edges.addEdge(i, v)
8:  while Edges.size() < E do
9:      u ← randInt(0, V-1)
10:     v ← randInt(0, V-1)
11:     if u < v and (u, v) ∉ Edges then
12:         Edges.addEdge(u, v)
                                          ▷ obtain a single source
13: for i ← 1 to V-1 do
14:     if (*, i) ∉ Edges then
15:         Edges.addEdge(0, i)
                                          ▷ assign operators to vertices
16: Vertex[0] ← createSource(...)
17: for i ← 1 to V-1 do
18:     Vertex[i] ← assignRandomOperator(...)
    return G = (Vertex, Edges)
```



(a) Accuracy (on initial non-optimized topologies).

(b) Errors (on initial non-optimized topologies).

Figure 7: Accuracy of the SpinStreams model to estimate the backpressure in streaming topologies. Predicted throughput *vs.* real one measured on Akka (a) and relative prediction error (b).

BoundedMailbox which, besides having a fixed capacity, blocks the sending actor if the destination mailbox is currently full. The length of this blocking phase can be controlled by setting a timeout after that the item is discarded by the sending operator. In order to avoid dropping of items, we set the timeout to be significantly higher than the maximum operators' service time (we use five seconds in the experiments of this section). Furthermore, in order to have stable results, the machine is exclusively used by our experiments and each actor is associated with a dedicated thread (the Akka runtime is configured to fully use all the 24 hyperthreaded cores of the machine). We repeat 10 times each experiment. In all cases, we measure a very small standard deviation, thus we will avoid showing error bars in the plots. Before running SpinStreams, we performed an initial profiling of all the operators. For sliding-window operators we consider three different window lengths and slide parameters (chosen randomly during the topology construction): 1000, 5000 and 10000 tuples sliding every 1, 10 or 50 new items received. The average service time per input tuple is in the fastest case of some hundreds of microseconds while in the worst case it is up to few hundreds of milliseconds. The source's service rate has been set to a properly high value such that the source is never a bottleneck, and so some operators act as bottleneck and the effect of backpressure can be appreciated in every topology.

5.2 Accuracy of the Backpressure Model

The first question that we pose is: ***does* SpinStreams *really provide an accurate estimation of the backpressure effect?*** To give an answer to this question, we evaluate SpinStreams in our testbed of 50 topologies. Our goal is to compare the *effective topology's throughput* estimated by SpinStreams with the one really measured by running the application in Akka. Throughput is defined as the average number of input tuples that the topology is able to ingest per second in the long running. It is measured as the departure rate of the source at steady state. The results of this set of tests are reported in Figure 7a. As can be easily realized, the SpinStreams model (cf. Algorithm 1) is able to precisely estimate the long-running performance of the topologies as the predicted throughput is very close to the measured one for all the topologies in the testbed. Namely, backpressure was properly detected and its effects precisely predicted. Figure 7b shows the relative error which is, on average, less than 3%. To perform a more in-depth analysis, we conducted a very fine-grained experiment in order to measure, for each operator, the relative error between the predicted departure rate and the real one. The main aim was to assess the ability to predict the backpressure for all the 678 operators composing the 50 random topologies. The results are reported in Figure 8 showing a prediction error that is 6.14% on average (standard deviation of 5%).

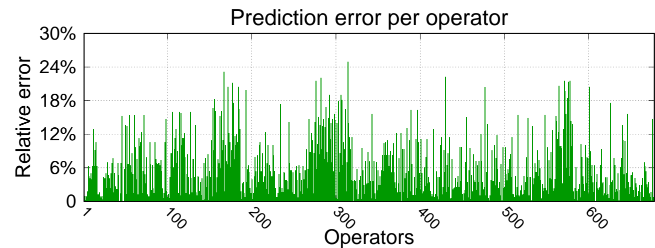
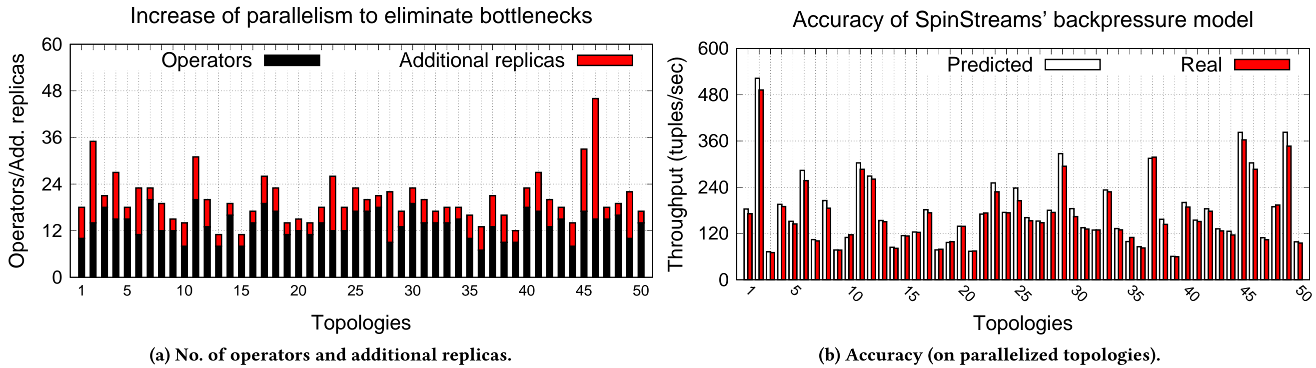


Figure 8: Relative error between the predicted departure rate and the measured one per operator in the tested topologies.

There are a few cases in which the error is higher than 20% (24.9%). A more in-depth analysis showed that this happens with operators that are not in their steady-state yet. This is a side-effect of the process used for generating the probabilities associated with the edges. In fact, some paths in the topology can be characterized by very low probabilities. As a consequence, this may result in a long time requested for reaching the steady-state, leading to a substantial error between the measured departure rate and the predicted one for operators along those paths. However, as a consequence of the very low probability, the contribution of such edges to the aggregate throughput of the whole topology is minimal and thus the prediction error for those operators has a limited effect.

5.3 Removing Bottlenecks

A second question that we pose is: ***does the bottleneck elimination phase of* SpinStreams *really result in an effective approach for optimizing throughput?*** To give an answer to this



(a) No. of operators and additional replicas.

(b) Accuracy (on parallelized topologies).

Figure 9: Analysis of the topologies after the bottleneck elimination phase of `SpinStreams`. No. of operators and total no. of additional replicas used (a), and accuracy of the backpressure model evaluated on the parallelized topologies (b).

question, we run the parallelization phase of `SpinStreams`. In the process dealing with the generation of topologies (cf. Algorithm 5), the `createRandomOperator()` function assigns operators to vertices. In case of partitioned-stateful operators, we also generate a random set of key groups with a distribution frequency generated by a random ZipF law. Furthermore, in order to guarantee that bottlenecks exist and the current throughput attained by the topologies is sub-optimal, the speed of the source is set to be 33% higher than the service rate of the faster operator in the topology.

The results of the parallelization phase are reported in Figure 9a, where we show for each topology in the testbed the initial number of operators and the total number of additional replicas used in each topology (an operator with $n > 0$ replicas has $n - 1$ additional replicas). Figure 9b shows the accuracy of the backpressure model for the optimized topologies. As it can be observed, the relative error is similar to the one characterizing the previous evaluation focusing on non-optimized topologies (about $3 - 3.5\%$ on average). Furthermore, we conducted an in-depth analysis on the parallelized version of each topology in the testbed:

- for 43 out of 50 topologies the ideal throughput has been reached after the parallelization step. This means that all the bottlenecks have been removed and the application throughput is equal to the generation rate of the source (we point out that in each topology the source has a different generation rate);
- for 7 out of 50 topologies the ideal throughput cannot be reached as some bottlenecks still remain. These bottlenecks are operators that have been marked with the *stateful* flag during their generation, i.e. they cannot be replicated (this is to mimic cases where operators cannot be parallelized). In that case, the aggregate throughput of the whole application is the one imposed by the backpressure generated by the bottleneck, which has been accurately predicted by `SpinStreams`.

In all cases, partitioned-stateful operators have been successfully parallelized when they were bottlenecks of the topologies, and emitter/collector actors, whenever they have been introduced, are not bottlenecks (their service time lasts few microseconds at most).

Sometimes, it can be useful to limit the effects of the bottleneck elimination phase since the optimal replication degree of certain operators might be very high. In this way the application designers can control the total amount of operators and replicas taking into account the amount of resources they plan to have available for processing. To this end, `SpinStreams` gives the possibility to add a maximum bound to the total amount of replicas (cf. Section 3.2). As matter of fact, using bounds that are lower than the optimal replication degree, the expectation is to obtain a proportional de-scalability of the performance with respect to the one could be achieved by the optimal topologies. To confirm this expectation we conducted specific tests; Figure 10 shows the results of an experiment using three random topologies. We compared the throughput of the initial topologies against the one after the bottleneck elimination phase, where we applied three different bounds: 30, 35 and 40 total replicas. Furthermore, we show the throughput achieved by parallelizing without any bound.

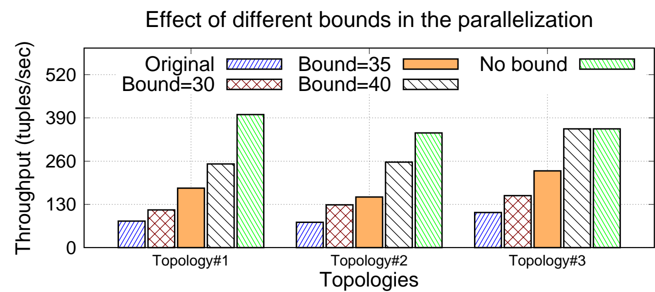


Figure 10: Application of different maximum bounds in the parallelization phase performed by `SpinStreams`.

As we can see, a proportional de-scalability is quite clear in the results. Interestingly, in the third topology the performance achieved with the highest bound is equal to the one without bounds. This happens because the topology needs fewer than 40 replicas to sustain the ideal generation rate of the source without bottlenecks.

5.4 Application of Operator Fusion

The last question that we pose is: ***does* `SpinStreams` *accurately estimate the outcome obtained by fusing a complex sub-graph in the topology?***. We first notice that fusion is not yet an automatized process in SpinStreams as the users are manually involved in selecting the candidate sub-graph used to apply fusion. Without pretending to be exhaustive, and in order to exemplify the application of this optimization, we present an example of a topology with six operators interconnected as in Figure 11(left).

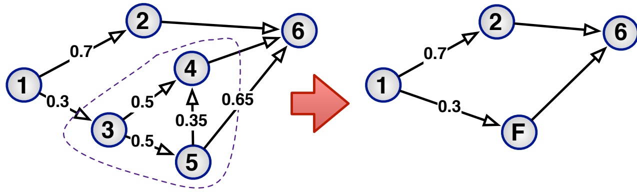


Figure 11: Example of operator fusion in SpinStreams. Original topology (left) and topology after the fusion of operators 3, 4 and 5 (right).

Original topology	Metric	1	2	3	4	5	6
	μ^{-1} (ms)	1.00	1.20	0.70	2.00	1.50	0.20
	δ^{-1} (ms)	1.00	1.42	3.33	4.93	6.67	1.00
	ρ	1.00	0.83	0.21	0.40	0.23	0.20
Throughput (tuples/sec)		**1,000** (predicted)			**961** (measured)		
Topology after fusion	Metric	1	2		F		6
	μ^{-1} (ms)	1.00	1.20		2.80		0.20
	δ^{-1} (ms)	1.00	1.42		3.33		1.00
	ρ	1.00	0.83		0.84		0.20
Throughput (tuples/sec)		**1,000** (predicted)			**970** (measured)		

Table 1: The proposed fusion of the sub-graph is feasible and does not impair performance.

Original topology	Metric	1	2	3	4	5	6
	μ^{-1} (ms)	1.00	1.20	1.50	2.70	2.20	0.20
	δ^{-1} (ms)	1.00	1.42	3.33	4.93	6.67	1.00
	ρ	1.00	0.83	0.45	0.55	0.33	0.20
Throughput (tuples/sec)		**1,000** (predicted)			**961** (measured)		
Topology after fusion	Metric	1	2		F		6
	μ^{-1} (ms)	1.00	1.20		4.42		0.20
	δ^{-1} (ms)	1.33	1.90		4.42		1.33
	ρ	0.75	0.63		1.00		0.15
Throughput (tuples/sec)		**760** (predicted)			**753** (measured)		

Table 2: The proposed fusion of the sub-graph introduces a new bottleneck in the topology.

Table 1 reports the predicted performance obtained by running SpinStreams (cf. Algorithm 1). Operators 3, 4 and 5 have a quite low utilization factor (i.e. they are underutilized). As a consequence, the user proposes to fuse such operators by generating the topology in Figure 11(right). SpinStreams predicts the service time $\mathcal{T}$ of the new operator (F) (cf. Algorithm 3) which is of 2.80 milliseconds on average. The analysis predicts that the fusion does not introduce any bottleneck as also confirmed by running the original and the modified topologies in Akka to collect the real measurements. Interestingly, a slight increase of throughput is measured by running the optimized topology in this particular case.

We run another experiment where we changed the functions used by some operators. The new service times are in Table 2 and the three operators 3, 4 and 5 are slightly slower now, with a predicted service time for the new operator F of about 4.42 milliseconds. Again, the user asks the outcome of the fusion but this time SpinStreams generates an alert stating that such fusion would impair the performance (a bandwidth degradation of 20% predicted before running the modified topology), as it has been also confirmed by the real execution.

6 RELATED WORK

A catalog of stream processing optimizations has been proposed [19] to provide a comprehensive survey of the existing techniques developed over the years. *Operator placement* allows for trading off communication cost against resource utilization by mapping operators onto computing resources. This can be done either statically [5] or dynamically [27], with the latter involving complex state migration activities to enforce replacement decisions while the application is running. SpinStreams does not deal with placement decisions, which are responsibility of the SPS once the optimized topology has been built (e.g., out from our tool).

Operator fission is by far the most studied optimization. A large volume of papers presented state migration techniques to reduce the overhead for changing the replication degree of partitioned-stateful operators. Some papers [35] studied this problem for single operators only. Other works [17, 39] applied fission to pipelines of stateless or partitioned-stateful operators that can be replicated as a whole. A recent paper proposes an elastic support on top of Heron [11]. The replication degree is adjusted by moving Heron instances in different containers to deploy on available resources. When the application is a complex topology, there is a *stability* problem in the changes performed on distinct operators due to the interplay among their steady-state performance [29, 30]. Another work [16] defines a cost model for backpressure used for dynamic adaptation. However, it assumes topologies with specific structures (e.g., linear sequences of operators).

As stated in this paper, *operator fusion* is less explored than fission. Some papers have applied fusion at compile-time. StreamIt [18] applies fusion to coarsen the granularity of the topology by adapting it to the number of available cores. Frameworks like Aurora [6, 7] fuse operators as long as the fused sub-graph performs less work than the maximum capacity provided by the machine executing it. COLA [24] is a compiler for System S [15] that fuses operators into run-time software units called Processing Elements, which

correspond to dedicated processes to run on the Operating System. Communications between operators mapped onto the same PE are replaced by function calls to avoid inter-process communications. Their algorithm follows a top-down strategy starting from a unique PE containing all the operators that is recursively split until a right granularity is reached. It is quite different from SpinStreams. COLA is a complier that finds the best application of fusion through heuristics, while SpinStreams is a tool supporting the users in a constructive way step-by-step. From a modeling viewpoint, SpinStreams overcomes some actual limitations of COLA, such as giving a precise modeling of the aggregate service rate of the fused operator that is not explicitly modeled in COLA. It is important to observe that such aforementioned approaches to fusion merge a set of operators into a single thread in the run-time system, i.e. they are tailored with a specific SPS. In contrast, the approach to fusion proposed in SpinStreams works at a higher abstraction level, as logical operators are fused into logical entities of the target SPS (e.g., actors in Akka), which are in turn executed by a pool of processes or threads at the system level. This decoupling allows for a general support for fusion, which can be generalized to any SPSs instead of being bounded to a specific runtime. This is one of the ambitions of our future work in generalizing SpinStreams to a variety of SPSs in addition to Akka.

Finally, there are relevant papers applying fusion dynamically. Flextream [21] applies fusion by stopping the application execution, recompiling the code with the new topology, and restoring the processing. Although interesting this may cause a long downtime that may not be always tolerated. Other approaches [26, 39] apply fusion while the application is running without pausing it. However, they actually limit fusion to operators along the same pipeline with constraints on their selectivity. SpinStreams can fuse much more general sub-graphs although it is a static tool.

7 CONCLUSIONS AND FUTURE WORK

This paper presented SpinStreams, a tool supporting the programmer during the design phase of data stream processing applications. SpinStreams accurately models the backpressure effect in complex topologies by leveraging appropriate cost models and optimization algorithms. Topologies are provided to SpinStreams as XML files along with files with actual implementations used to generate the final code for a target SPS. In the current release SpinStreams generates code for Akka, and extensive experiments have been conducted to show the accuracy of our approach in a set of random topologies of operators performing real-world computations.

Our work can be extended in several directions. First, we plan to conduct experiments in distributed environments (e.g., using the Remoting library of Akka) and to extend the models in order to cover cyclic topologies and multiple sources. Second, we would like to automatize the operator fusion process by making SpinStreams able to automatically choose the best sub-graph suitable for fusion without manual intervention by the user. Finally, we would like to extend the code generation part for other agent-based frameworks like CAF [8, 41] and SPSs like Apache Storm [23] and Flink [12], and we aim at studying the joint combination of static and dynamic optimizations (e.g., elasticity) in the future, to assess the full potential of the most popular stream processing optimizations.

REFERENCES

[1] Lorenzo Affetti, Riccardo Tommasini, Alessandro Margara, Gianpaolo Cugola, and Emanuele Della Valle. 2017. Defining the execution semantics of stream processing engines. *Journal of Big Data* 4, 1 (26 Apr 2017), 12. https://doi.org/10.1186/s40537-017-0072-9

[2] Jamie Allen. 2013. *Effective Akka.* O'Reilly Media, Inc.

[3] Henrique Andrade, Buğra Gedik, and Deepak Turaga. 2014. *Fundamentals of Stream Processing.* Cambridge University Press. Cambridge Books.

[4] C. Bertolli, G. Mencagli, and M. Vanneschi. 2010. Analyzing Memory Requirements for Pervasive Grid Applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing.* 297–301. https://doi.org/10.1109/PDP.2010.71

[5] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2016. Optimal Operator Placement for Distributed Stream Processing Applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS '16).* ACM, New York, NY, USA, 69–80. https://doi.org/10.1145/2933267.2933312

[6] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. 2003. Operator Scheduling in a Data Stream Manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29 (VLDB '03).* VLDB Endowment, 838–849. http://dl.acm.org/citation.cfm?id=1315451.1315523

[7] Uğur Çetintemel, Daniel Abadi, Yanif Ahmad, Hari Balakrishnan, Magdalena Balazinska, Mitch Cherniack, Jeong-Hyon Hwang, Samuel Madden, Anurag Maskey, Alexander Rasin, Esther Ryvkina, Mike Stonebraker, Nesime Tatbul, Ying Xing, and Stan Zdonik. 2016. *The Aurora and Borealis Stream Processing Engines.* Springer Berlin Heidelberg, Berlin, Heidelberg, 337–359. https://doi.org/10.1007/978-3-540-28608-0_17

[8] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. 2014. CAF - the C++ Actor Framework for Scalable and Resource-Efficient Applications. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control (AGERE! '14).* ACM, New York, NY, USA, 15–28. https://doi.org/10.1145/2687357.2687363

[9] X. Chen, Y. Vigfusson, D. M. Blough, F. Zheng, K. L. Wu, and L. Hu. 2017. GOVERNOR: Smoother Stream Processing Through Smarter Backpressure. In *2017 IEEE International Conference on Autonomic Computing (ICAC).* 145–154. https://doi.org/10.1109/ICAC.2017.31

[10] A. Destounis, G. S. Paschos, and I. Koutsopoulos. 2016. Streaming big data meets backpressure in distributed network computation. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications.* 1–9. https://doi.org/10.1109/INFOCOM.2016.7524388

[11] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-regulating Stream Processing in Heron. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1825–1836. https://doi.org/10.14778/3137765.3137786

[12] Ellen Friedman and Kostas Tzoumas. 2016. *Introduction to Apache Flink: Stream Processing for Real Time and Beyond* (1st ed.). O'Reilly Media, Inc.

[13] Buğra Gedik. 2014. Generic Windowing Support for Extensible Stream Processing Systems. *Softw. Pract. Exper.* 44, 9 (Sept. 2014), 1105–1128. https://doi.org/10.1002/spe.2194

[14] Buğra Gedik. 2014. Partitioning Functions for Stateful Data Parallelism in Stream Processing. *The VLDB Journal* 23, 4 (Aug. 2014), 517–539. https://doi.org/10.1007/s00778-013-0335-9

[15] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. 2008. SPADE: The System s Declarative Stream Processing Engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08).* ACM, New York, NY, USA, 1123–1134. https://doi.org/10.1145/1376616.1376729

[16] B. Gedik, H.G. Ozsema, and O. Ozturk. 2016. Pipelined fission for stream programs with dynamic selectivity and partitioned state. *J. Parallel and Distrib. Comput.* 96 (2016), 106 – 120. https://doi.org/10.1016/j.jpdc.2016.05.003

[17] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2014. Elastic Scaling for Data Stream Processing. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (June 2014), 1447–1463. https://doi.org/10.1109/TPDS.2013.295

[18] 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. *SIGARCH Comput. Archit. News* 30, 5 (Oct. 2002), 291–303. https://doi.org/10.1145/635506.605428

[19] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46, 4, Article 46 (March 2014), 34 pages. https://doi.org/10.1145/2528412

[20] Christoph Hochreiner, Michael Vagler, Stefan Schulte, and Schahram Dustdar. 2017. Cost-efficient enactment of stream processing topologies. *PeerJ Computer Science* 3 (Dec. 2017), e141. https://doi.org/10.7717/peerj-cs.141

[21] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. 2009. Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous

Architectures. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 214–223. https://doi.org/10.1109/PACT.2009.39

[22] Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. 2013. Elastic Stream Processing in the Cloud. *Wiley Int. Rev. Data Min. and Knowl. Disc.* 3, 5 (Sept. 2013), 333–345. https://doi.org/10.1002/widm.1100

[23] Ankit Jain. 2017. *Mastering Apache Storm: Real-time Big Data Streaming Using Kafka, Hbase and Redis.* Packt Publishing.

[24] Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel Wolf, Kun-Lung Wu, Henrique Andrade, and Buğra Gedik. 2009. COLA: Optimizing Stream Processing Applications via Graph Partitioning. In *Middleware 2009*, Jean M. Bacon and Brian F. Cooper (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 308–327.

[25] Yunbo Li, Anne-Cécile Orgerie, Ivan Rodero, Betsegaw Lemma Amersho, Manish Parashar, and Jean-Marc Menaud. 2018. End-to-end energy models for Edge Cloud-based IoT platforms: Application to data stream analysis in IoT. *Future Generation Computer Systems* 87 (2018), 667 – 678. https://doi.org/10.1016/j.future.2017.12.048

[26] Björn Lohrmann, Daniel Warneke, and Odej Kao. 2014. Nephele Streaming: Stream Processing Under QoS Constraints at Scale. *Cluster Computing* 17, 1 (March 2014), 61–78. https://doi.org/10.1007/s10586-013-0281-8

[27] K. G. S. Madsen, Y. Zhou, and J. Cao. 2017. Integrative Dynamic Reconfiguration in a Parallel Stream Processing Engine. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 227–230. https://doi.org/10.1109/ICDE.2017.81

[28] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A Domain-specific Language for Bytecode Instrumentation. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development (AOSD '12)*. ACM, New York, NY, USA, 239–250. https://doi.org/10.1145/2162049.2162077

[29] G. Mencagli and M. Vanneschi. 2011. QoS-control of Structured Parallel Computations: A Predictive Control Approach. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. 296–303. https://doi.org/10.1109/CloudCom.2011.47

[30] G. Mencagli, M. Vanneschi, and E. Vespa. 2013. Control-theoretic adaptation strategies for autonomic reconfigurable parallel applications on cloud environments. In *2013 International Conference on High Performance Computing Simulation (HPCS)*. 11–18. https://doi.org/10.1109/HPCSim.2013.6641387

[31] Harry G. Perros. 1994. *Queueing Networks with Blocking.* Oxford University Press, Inc., New York, NY, USA.

[32] Constantin Pohl, Philipp Goetze, and Kai-Uwe Sattler. 2017. A Cost Model for Data Stream Processing on Modern Hardware. In *ADMS Workshop*. http://www.adms-conf.org/2017/camera-ready/adms2017_final.pdf

[33] Nicoló Rivetti, Yann Busnel, and Leonardo Querzoni. 2016. Load-aware Shedding in Stream Processing Systems. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS '16)*. ACM, New York, NY, USA, 61–68. https://doi.org/10.1145/2933267.2933311

[34] Andrea Rosà, Lydia Y. Chen, and Walter Binder. 2016. Profiling Actor Utilization and Communication in Akka. In *Proceedings of the 15th International Workshop on Erlang (Erlang 2016)*. ACM, New York, NY, USA, 24–32. https://doi.org/10.1145/2975969.2975972

[35] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K. Wu. 2009. Elastic scaling of data parallel operators in stream processing. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–12. https://doi.org/10.1109/IPDPS.2009.5161036

[36] Daniele De Sensi, Massimo Torquati, and Marco Danelutto. 2017. Mammut: High-level management of system knobs and sensors. *SoftwareX* 6 (2017), 150 – 154. https://doi.org/10.1016/j.softx.2017.06.005

[37] B. Shinde and S. T. Singh. 2016. Data parallelism for distributed streaming applications. In *2016 International Conference on Computing Communication Control and automation (ICCUBEA)*. 1–4. https://doi.org/10.1109/ICCUBEA.2016.7859983

[38] Julien Subercaze, Christophe Gravier, Syed Gillani, Abderrahmen Kammoun, and Frédérique Laforest. 2017. Upsortable: Programming Top-k Queries over Data Streams. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1873–1876. https://doi.org/10.14778/3137765.3137797

[39] Y. Tang and B. Gedik. 2013. Autopipelining for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems* 24, 12 (Dec 2013), 2344–2354. https://doi.org/10.1109/TPDS.2012.333

[40] Quoc-Cuong To, Juan Soto, and Volker Markl. 2018. A survey of state management in big data processing systems. *The VLDB Journal* (02 Aug 2018). https://doi.org/10.1007/s00778-018-0514-9

[41] M. Torquati, T. Menga, T. De Matteis, D. De Sensi, and G. Mencagli. 2018. Reducing Message Latency and CPU Utilization in the CAF Actor Framework. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 145–153. https://doi.org/10.1109/PDP2018.2018.00028