WindFlow: High-Speed Continuous Stream Processing With Parallel Building Blocks

Gabriele Mencagli, Massimo Torquati, Andrea Cardaci, Alessandra Fais, Luca Rinaldi, and Marco Danelutto

Abstract—Nowadays, we are witnessing the diffusion of Stream Processing Systems (SPSs) able to analyze data streams in near real-time. Traditional SPSs like STORM and FLINK target distributed clusters and adopt the continuous streaming model, where inputs are processed as soon as they are available while outputs are continuously emitted. Recently, there has been a great focus on SPSs for scale-out machines. Some of them (e.g., BRISKSTREAM) still use the continuous model to achieve low latency. Others optimize throughout with batching approaches that are, however, often inadequate to minimize latency for live-streaming applications. Our contribution is to show a novel software engineering approach to design the runtime system of SPSs targeting multicores, with the aim of providing a uniform solution able to optimize throughput and latency. The approach has a formal nature based on the assembly of components called building blocks, whose composition allows optimizations to be easily expressed in a compositional manner. We use this methodology to build a new SPS called WindFlow. Our evaluation showcases the benefits of WindFlow: it provides lower latency than SPSs for continuous streaming, and can be configured to optimize throughput, to perform similarly and even better than batch-based scale-up SPSs.

Index Terms—Data stream processing, multcore programming, parallel computing

1 INTRODUCTION

With the increasing availability of large data volumes in the form of streams, there has been an eager demand for Stream Processing Systems (SPSs) able to process streams in a continuous fashion on commodity hardware by providing timely responses to the end users.

Streaming applications are modeled as Directed Acyclic Graphs (DAGs) of operators exchanging data items called tuples. Operators process tuples and produce outputs based on their business logic code. Traditionally, streaming applications have been developed through dialects of SQL (e.g., CQL [1]), where the semantics of relational algebra has been adapted to the streaming domain with proper constructs to deal with unbounded streams rather than finite tables. However, recent SPSs have extended the scope of their supported applications to go beyond the domain of relational algebra. This is done by dealing with both structured and unstructured data, and supporting the execution of complex computational tasks, even adding the possibility to leverage external tools for specific domains (e.g., TensorFlow and PyTorch for Machine Learning [2]).

In terms of their runtime system, popular SPSs like STORM [3] and FLINK [4] are based on the Java Virtual Machine (JVM), and are designed to scale out on several interconnected machines. This has an impact on a set of aspects (data de-/serialization, inter-process communication and resource scheduling) that are inefficient when the processing is done on a single scale-up machine [5], [6], [7].

The efficient exploitation of multicores requires to rethink the runtime system (or simply runtime) design of SPSs. Systems adopting the continuous streaming model allow the processing of inputs as soon as they are available, and operators are executed by independent threads. This approach is adopted by popular SPSs like STORM and FLINK, and by some research prototypes for multicores like BRISKSTREAM [8], with the ultimate goal of minimizing latency. An alternative trend for designing new SPSs for scale-up servers is to change the runtime system design to enhance the throughput by fitting at best the memory bandwidth of the machine. Some new prototypes implement this approach by adopting the discretized streaming model (inspired by the morsel-driven parallelism in [9]), where inputs are buffered in batches that are then scheduled to a pool of available threads. Threads execute entire pipelines of operators as a tight loop on the batch elements, until a pipeline breaker (e.g., a keyby distribution) is reached. Recent research works (e.g., STREAMBOX [10] and others [11]) successfully adopt this approach, and are able to implement applications composed of relational algebra operators exchanging structured records of data. However, most of their optimizations (e.g., automatic code generation) are hard to be extended to more general application domains. Furthermore, the buffering required to build properly sized batches (often in the order of thousands of inputs to amortize the scheduling cost) prevents achieving small end-to-end latencies. For this reason, and thanks to their peak throughput on multicores, such prototypes are mostly used to process offline...
streaming tasks with in-memory data rather than real live-streaming applications.

From this past experience, two main trends for designing SPS runtimes on multicore can be identified: one more latency oriented, and the other more throughput oriented. In this regard, the primary contribution of this work is to present a novel software engineering strategy proposing a unified approach that increases the abstraction level of the implementation of SPS runtimes. To the best of our knowledge, this has not been studied before from the software engineering perspective. Our approach inherits several features of traditional runtimes for continuous streaming, like having independent threads per operator connected by queues for data exchange. In addition, it identifies a minimal set of customizable building blocks that can be composed according to a formal semantics. Building blocks are recurrent data-flow compositions of interconnected activities, and they can be used as an abstraction layer representing an algebra of programs. Some features of our building blocks are:

- we provide formal transition rules and a grammar of building blocks. This formal approach defines a powerful abstraction layer able to describe many streaming DAGs suitable for modeling applications in a wide set of domains needing streaming support;
- our novel abstraction layer allows system designers to reason about optimizations in an easier way, by developing new strategies in terms of how building blocks are composed;
- their compositions model operator chaining and the removal of centralization points. Furthermore, the input size and processing granularity can be tuned to optimize latency or increase throughput;
- their producer-consumer semantics simplifies the explicit memory management in languages like C++, and helps to make it transparent to the user;
- they help in mapping the operators onto the cores of the machine in an effective manner through pinning strategies leveraging the structured nature of the runtime;
- they can be implemented in different ways (e.g., with thread-based, actor-based or task-based parallelism).

We provide an implementation of this methodology within the WindFlow library targeting multicore. WindFlow is written in modern C++17, and provides an efficient implementation of building block compositions. In particular, our contributions with the library are:

- our building blocks are implemented using thread-based parallelism and lock-free queues with different concurrency control mechanisms, to efficiently deal with backpressure and exploitation of hardware multi-threading;
- the experimental validation is based on seven real-world streaming applications. The evaluation shows that WindFlow is able to model common DAGs, and it is faster than FLINK and STORM and the recent JVM-based prototype BRISKSTREAM [8];
- WindFlow has also been compared with the C++-based solution STREAMBox. This allows us to show that high throughput can be achieved owing to a proper combination of building blocks, configured to exchange and process sufficiently sized inputs, rather than changing the runtime to schedule large batches dynamically;
- the source code of the library and all applications implemented with all frameworks have been released open-source for reproducibility.1

The methodology developed in this paper extends our prior work [12] in several directions: i) we provide a formal description and semantics of our building blocks to implement the whole runtime system of a SPS expressing general streaming DAGs, while our previous publication studied the implementation of specific parallel operators (sliding-window operators) without providing any formal semantics; ii) this new study analyzes the performance impact of several implementation techniques in terms of pinning strategies, concurrency control mechanisms, and small batching not studied before; iii) the experimental evaluation in this paper is done by comparing against research-based SPSs (BRISKSTREAM and STREAMBox), and not only against traditional SPSs (STORM and FLINK).

The outline of the paper is as follows. Section 2 presents an overview of the library with its C++17 API. Section 3 introduces our building blocks and their grammar. Section 4 describes the formal design of the WindFlow library. Section 5 presents the experimental evaluation, with the considered applications and the results in terms of throughput and latency. Section 6 provides a discussion of related works and Section 7 draws the conclusions of this paper.

## 2 WindFlow Overview

WindFlow is a header-only library, designed by using generic programming leveraging the recent features of the C++17 standard for template argument deduction by the compiler (Class Template Argument Deduction–CTAD [13]). In this part, we will give an overview of the library with the set of available operators and the API to create applications.

### 2.1 Operators

WindFlow provides a set of basic and windowed operators that can be interconnected in data-flow graphs. Operators can be internally replicated to increase their throughput, with internal replicas working on a subset of the inputs received by the operator. Table 1 reports the operators offered by WindFlow. The column distribution shows how inputs are delivered to the operator replicas: forward means that every input can be assigned to any replica, key by sends all the inputs having the same key attribute (e.g., a specific field of the tuples) to the same replica, and complex distributions deliver the same input to one or more replicas according to some predefined policies. For the Source the distribution is undefined since its replicas never receive inputs.

The Source is in charge of generating a stream of tuples all having the same type. The Sink is responsible for absorbing the inputs without generating any output. The Filter drops all the tuples not respecting a user-defined predicate. The Map produces one output per input while the FlatMap

---

1. WindFlow is available at this link https://github.com/ParaGroup/WindFlow
produces zero, one or more outputs per input (inputs and outputs may have different data types). The Accumulator maintains a state for each value of the key attribute. For each input, a user-defined processing function works on that input and on the corresponding state of its key, and the new value of the state is produced in output by the operator.

Some applications require to periodically repeat user-defined computations over finite portions of the stream having often the form of moving windows [14], [15], which can be of different types (e.g., tumbling, sliding and hanning). WindFlow provides specific operators to express window-based computations and to execute them in parallel when windows activate very frequently.

Differently from relational algebra SPSs (e.g., historical ones like in [16], and more recent ones like [10], [17]), WindFlow operators work on inputs conveying structured or unstructured data, and the transformation logic can be fully defined by the user. Furthermore, stateful computations (with keyby partitioning) can be configured in customized operators, by keeping an internal state implemented by a user-defined data structure in each replica.

### 2.2 API

We designed the API of WindFlow to be similar to the ones of traditional SPSs, like Storm and Flink. Hence, our goal is to target general-purpose stream processing, with support for generic operators as well as their stateful definition (with user-defined states provided in input to the operator business logic code). Therefore, our API is targeting end users with data analytics problems. For specific application domains like streaming queries, expressible with relational algebra, we can design in the future a SQL-based domain specific language on top of the WindFlow’s API, like in other tools. However, this is not the scope of this paper. In the following, we will focus on how operators can be created and how applications are composed in WindFlow.

#### 2.2.1 Creating Operators

WindFlow provides a compositional fluent interface based on builder classes. Fig. 1 shows how to create a Map using its builder class Map_Builder. By leveraging the CTAD feature of C++17, the template arguments for instantiating the Map class (the data types input_t and output_t in the figure) are automatically deduced by the signature of the function provided to the builder constructor.

```cpp
class Map_Function {
public:
  output_t operator[](const input_t &t) {
    // code here
  }
};
Map_Function myMap;
Map map = Map_Builder(map).withName("myMap") .withParallelism(5).build();
```

Fig. 1. Example of instantiation of the Map operator with the C++17 fluent interface.

The library does not rely on class inheritance to define the logic of the operators (to avoid the little overhead of virtual function calls). WindFlow accepts several signatures for each operator, where the logic can be provided either as a plain function, as an anonymous lambda, or through functor objects like in Fig. 1. This latter option allows logics that are not purely functional to be used by the operator: the state can be implemented as internal variables of the functor object, and the runtime system guarantees that each replica uses a distinct copy of the object.

The customization is done with the method chaining technique, where configuration options are set with specific methods. Finally, the build() method creates the instance of the properly configured operator. In the figure, the Map is created with five replicas and with a name (the string “myMap”) used for logging purposes. Other options are available depending on the operator type.

#### 2.2.2 Creating Applications

Applications are developed using the MultiPipe and the PipeGraph constructs. In its basic definition, a MultiPipe is a set of parallel pipelines of operator replicas, where a replica of an operator in a pipeline communicates either with one or with all the replicas of the next operator. When a replica receives inputs only from one replica of the previous operator (in the same pipeline), we call this communication pattern direct connection. A shuffle connection is when a replica receives inputs from all the replicas of the previous operator (e.g., because tuples are distributed on a keyby basis).

A MultiPipe can be created by adding a Source to the PipeGraph, which is the environment used to create and run the application. Fig. 2 shows an example of instantiation, where the created MultiPipe is fed by a Source (previously created, and not shown in the code snippet for brevity). Then, two previously created operators are added to the MultiPipe. Finally, a Sink is added to the MultiPipe. The application is run by invoking the run() method on the PipeGraph. It terminates when all the Source replicas terminate, and all the generated values have been fully processed.

To model more general DAGs, merge and split operations can be applied to MultiPipes. Fig. 3 shows a MultiPipe obtained by merging two MultiPipes fed by different Sources, which is then split into two MultiPipes having

| Operator   | Acronym | Distribution |
|------------|---------|--------------|
| Source     | SRC     |              |
| Sink       | SNK     | Forward/KeyBy|
| Filter     | F       | Forward/KeyBy|
| Map        | M       | Forward/KeyBy|
| FlatMap    | FM      | Forward/KeyBy|
| Accumulator| A       | KeyBy        |
| Keyed Windows | EW   | KeyBy        |
| Parallel Windows | PW | Complex     |
| Paned Windows | PW | Complex     |
| Map-Reduce Windows | MRW | Complex    |

### Table 1

Standard Operators Available in WindFlow
different Sinks. All the operators in the figure have two replicas for simplicity. The API provides the `merge()` and the `split()` methods, the latter invoked on a `MultiPipe` by providing a `splitting function` stating how the outputs are assigned to the destination `MultiPipes` (called `splitting branches`). The function returns a vector of identifiers, one for each destination, in order to model unicast, multicast or broadcast distributions. In the last two cases, a copy of the input tuple is provided to each destination. Each `MultiPipe` in a branch is obtained with the `select()` method invoked on the parent `MultiPipe`, and can be filled with new operators.

Merged `MultiPipes` must produce outputs of the same type, and the first operator added in each splitting branch must receive inputs with the same type of the outputs produced by its parent. These checks are done in the `run()` method by using the RTTI (RunTime Type Identification) feature of C++. In the next section, we present the formal approach that we used to design the runtime system.

### Building Blocks

We adopt a formal software engineering approach based on the concept of elementary concurrent components called Building Blocks (BBs) [18], [19]. We point out that the existence of BBs, as well as their composition, is part of the runtime system level, and thus invisible to the end users of our library developing streaming applications. More specifically, we will use the Sequential Building Blocks (SBBs) and the Parallel Building Blocks (PBBs) described in Table 2. Each SBB processes the inputs in a First-Come First-Served manner and produces outputs. PBBs are structured graphs of interconnected SBBs and PBBs.

SBBs are wrappers executing sequential code and combiners. `Seq(f)` wraps a function `f` into a SBB applying `f` on each input sequentially. Each application of `f` may produce zero, one or multiple outputs. Combiners are defined based on other wrappers or even on other combiners. The processing semantics of `Comb(Seq(f_1), Seq(f_2))` is the one of the sequential composition of functions: each output produced by `f_1` becomes the input of `f_2`, and its outputs become the outputs of the whole combiner block. The application of the two functions is performed serially. This idea has been extended in the `fan-out` combiner `Comb(Seq(f), {Seq(g_i)}_{i=1}^n)`, executing `f` on each input, and then for each output produced by `f` the function `g_i` of one of the SBBs in the set `{Seq(g_i)}_{i=1}^n` is applied (randomly chosen or selected based on the properties of the output delivered by `f`). The outputs from `g_i` finally become the outputs of the combiner block. SBBs can be nested as in the production `Σ` of the grammar in Table 2. Fig. 4 shows the graphical notation of BBs.

PBBs allow SBBs to be interconnected in regular structures. An ordered chain of SBBs is allowed with the `Pipe` block, where the outputs of a SBB become inputs to the next one. Differently from the `Comb`, SBBs in the `Pipe` run in parallel on subsequent elements (stream). PBBs `Δ_1, ..., Δ_n` can be grouped into a container `Δ_1, ..., Δ_n`. We use the succinct notation `Δ^n{Δ_1, ..., Δ_n}`. When all the PBBs are identical, we use `Δ^n` (without the index `i` on `Δ`). Furthermore, if the container contains one PBB only, we use `Δ` directly in place of `Δ^n{Δ_1, ..., Δ_n}`. Finally, the `all-to-all` block (A2A) allows PBBs to be interconnected in any regular structure.

### Table 2

| Name             | Syntax        | Description                                                                 |
|------------------|---------------|-----------------------------------------------------------------------------|
| **Sequential Building Blocks (SBBs)** |               |                                                                             |
| `Seq` wrapper     | `Seq(f)`      | Wraps the sequential function `f` into a data-flow node (an execution entity) executing `f` sequentially on each input |
| Combiner          | `Comb(Σ_1, Σ_2)` | Merges two SBBs `Σ_1` and `Σ_2` into a SBB executing the semantics of the sequential composition of the processing logic computed by `Σ_1` and `Σ_2` |
| Fan-out Combiner  | `Comb(Σ', {Σ_i}^n_{i=1})` | Merges `Σ'` and a list of `n > 1` SBBs `Σ_1, ..., Σ_n` into a single SBB. For each input to the combiner, the logic of `Σ'` is first executed, and then for each output of `Σ'` the logic of one of `Σ_1, ..., Σ_n` (e.g., selected based on the properties of the output delivered by the logic of `Σ'`) is applied to produce the outputs of the combiner |

| **Parallel Building Blocks (PBBs)** |               |                                                                             |
| Pipe              | `Pipe(Δ_1, ..., Δ_n)` | Connects `n` SBBs in a pipeline, where `Δ_i` receives from `Δ_{i-1}` and sends to `Δ_{i+1}`. The unary case `Pipe(Δ)` is admissible, and it has the same semantics of `Δ`. |
| Parallel Container| `{Δ_i}^n_{i=1}` | A set of `Δ_1, ..., Δ_n` PBBs. The blocks are independent (not interconnected together), so they work in parallel on different inputs and produce outputs |
| All-to-All        | `A2A({Δ_i}^n_{i=1}, {Δ_i}^n_{i=1})` | All the PBBs in the left side `{Δ_i}^n_{i=1}` send outputs that can be delivered to any PBB in the right side `{Δ_i}^n_{i=1}`. The connection topology is fully connected (`n × n` block-to-block connections) |

The grammar shows which composition of BBs is admissible in the description language.
interconnected in a shuffle communication pattern, where the rightmost SBBs within each PBB in the left container communicate with all the leftmost SBBs within the PBBs in the right container. When a SBB has multiple outgoing connections, each output is delivered to one of the output channels chosen randomly, or on the basis of some key attribute field, or using other complex policies. The production $\Delta$ in Table 2 shows the possible nesting of PBBs. The Pipe is used to interconnect SBBs, while the A2A interconnects two containers of PBBs allowing the nesting of A2A with Pipe blocks and with other A2A blocks.

### 3.1 Utility Functions on PBBs
We introduce some utility functions performing transformations of PBBs that we use in our design in the next section:

- **combine-with-last:** $\triangleright : \Delta \times \Sigma \rightarrow \Delta$ combines a copy of the SBB $\Sigma$ with all the rightmost SBBs in $\Delta$;
- **pipe-with-last:** $\triangleright : \Delta \times \Sigma \rightarrow \Delta$ appends a copy of the SBB $\Sigma$ to all the rightmost pipes in $\Delta$ (which are extended with this additional block at the end).

Fig. 5 shows the semantics of these functions formally described through some transition rules (having the usual form of a set of premises and a consequence). One rule per function ($\triangleright_{pipe}$ and $\triangleright_{pipe}$) is applied when $\Delta$ is a Pipe, which represents the base case. Other two are applied when $\Delta$ is an A2A ($\triangleright_{a2a}$ and $\triangleright_{a2a}$) or a parallel container ($\triangleright_{pc}$ and $\triangleright_{pc}$), where the transformations are applied recursively.

### 4 WindFlow Design
The WindFlow library has been designed leveraging the previous set of BBs used with the formal composition rules presented in this section. We introduce the concept of Matryoshka (referred to as $\mathcal{M}$), a compound BB obtained as composition of our BBs. We show what Matryoshka models, and how it has been used to build the MultiPipe structure, whose API has already been sketched in Section 2.2.

#### 4.1 Matryoshka
The Matryoshka is the basic element of the MultiPipe. It models a set of pipelines that may have shuffle connections from one pipeline to another one. Examples of this structure are the ones within a rounded box in Fig. 3. For example, direct connections exist between the two replicas of SRC1 and OP1, while shuffle connections interconnect the replicas of OP1 and OP2 in the first box of the figure.

We define these structures with the following grammar, where $\Phi$ is a Pipe with any arbitrary length $m > 0$

$$
\Phi ::= \text{Pipe}(\Sigma_1, \ldots, \Sigma_m) \quad \mathcal{M} ::= [\Phi]^n \mid \text{A2A}([\Phi]^n, \mathcal{M}).
$$

A “Matryoshka doll”, in the Russian tradition, is a set of decreasing size dolls placed one inside another. The innermost one is called seed. Following this analogy, our seed is a parallel container of $n > 0$ identical pipelines, while the recursive case is instances of the A2A block where a new inner Matryoshka is nested in the right-hand side (fixed to have one block only). We define three utility functions. The first OutCard : $\mathcal{M} \rightarrow \mathbb{N}$ returns, given an input Matryoshka, the number of Pipes in its seed. The second $\bowtie : \mathcal{M} \times [\Phi]^n \rightarrow \mathcal{M}$ puts a parallel container of Pipes $[\Phi]^n$ (with any $n$) as the new seed of $\mathcal{M}$. LastOP : $\mathcal{M} \rightarrow T_{op}$ returns the type of the last operator added to the input Matryoshka. $T_{op}$ will be defined shortly. The transition rules of the utility functions are shown in Fig. 6.

We introduce the three main operations that can be used to create and to modify Matryoshkas:

- **creation of a Matryoshka starting from an operator,**
  - $\text{create} : \text{OP} \rightarrow \mathcal{M}$ where $\text{OP}$ is the set of operators;
- **adding of an operator into a Matryoshka,** $\text{add} : \mathcal{OP} \times \mathcal{M} \rightarrow \mathcal{M}$. Depending on the number of replicas of the operator, and on its distribution logic (e.g., forward, keyby or complex), the replicas are appended as SBBs to the Pipes in the innermost Matryoshka (direct connections) or, alternatively, by adding a new Matryoshka as the new seed (shuffle connections);
- **chaining of an operator into a Matryoshka,** $\text{chain} : \mathcal{OP} \times \mathcal{M} \rightarrow \mathcal{M}$. If possible, this transformation combines the SBB implementing the replica of the operator to each rightmost SBB present in the Pipes of the seed.

Operators are denoted by a tuple $< f_d, f_{op}, n > \in \mathcal{OP}$, where $f_d$ is the distribution function returning for each input $t$ one or more pairs $< t, dst >$, where $dst$ is the index of the replica in charge of processing $t$ ($n > 0$ is the number of replicas). In case of multiple output pairs, the same input is delivered to more than one replica. The operators $\mathcal{M}$, $\text{FM}$,
F, and SNK use the forward distribution unless they are created with the keyby modifier. The function $\text{TypeId}(f_d)$ returns the type of the distribution $f_d$. The element $f_{op}$ is the processing logic of the operator returning zero, one or multiple outputs per input. We also introduce $\text{TypeId}(f_{op})$ to return the type of the operator, which is in the set $\text{Top} = \{\text{SRC}; \text{SNK}; \text{M}; \text{FM}; \text{F}; \text{A}; \text{KW}; \text{PW}; \text{PAW}; \text{MRW}\}$.

Fig. 7 shows the rules of the create, add and chain operations. The semantics describes well-formed Matryoshkas where after a Sink no other operator can be added or chained (the Matryoshka is closed). In the basic case, Matryoshkas are created starting from a Source operator. However, we have relaxed this constraint because Matryoshkas can be created starting from any operator to model the split and merge of MultiPipes (see the next section).

The rule “create” generates a new Matryoshka (a seed) starting from an operator. Each replica is a Seq within a Pipe running the operator logic $f_{op}$. The rule “add-direct” is applied when the new operator has a forward distribution and the number of its replicas is equal to the number of Pipes in the seed of the input Matryoshka. In this case, we append a new Seq running $f_{op}$ to each Pipe of the seed. The rule “add-shuffle” is executed if the distribution type is not forward, or when the number of replicas is not equal to the number of Pipes in the seed of the input Matryoshka. The transformation inserts a new seed composed of the replicas of the operator logic implemented by a set of Seqs each within a Pipe. Before doing this, the input Matryoshka is modified by combining with its rightmost SBBs a Seq running the distribution logic of the new operator ($\text{M} \triangleright \text{Seq}(f_{op})$). In this way, outputs are correctly routed to the replicas of the new operator through the new shuffle connections. Finally, the last two rules model the chain, which has effect if the distribution type is forward and the number of replicas coincide. Rule “chain,” combines a Seq running $f_{op}$ to the rightmost SBB within each Pipe in the current seed, such that it is executed serially after the logic of the previous operator (through a function call). Rule “chain,” models the case where the chain is not admitted, and the add is executed otherwise.

4.2 MultiPipe

The MultiPipe allows Matryoshkas to be interconnected in acyclic graphs called S/M graphs (Split-and-Merge). In Fig. 3, we used the merge operation to allow the replicas of OP4 to receive inputs from the replicas of OP2 and OP3, while the split operation is applied to distribute the outputs from OP4 to OP5 and SNK2. The library is able to model all the graphs generated by the following grammar:

$$
\Gamma ::= \mathcal{M} \mid \text{A2A}(\Gamma), \mathcal{M}^+ \mid \text{A2A}(\Gamma^+, \Gamma^+, \Gamma^+), \mathcal{M}^+$$

(2)
A MultiPipe $\Gamma$ is either: i) a Matryoshka (terminal symbol), or ii) an A2A having $n > 0$ MultiPipes in the left-hand side and one $\Gamma$ in the right-hand side, or iii) an A2A having one $\Gamma^+$ in the left-hand side and $n > 0$ MultiPipes in the right-hand side. $\Gamma$ and $\Gamma^+$ are two MultiPipe structures that are either terminal or an A2A with one terminal symbol in the left-hand side ($\Gamma$) or with one terminal symbol in the right-hand side ($\Gamma^+$). Fig. 8 shows one of the derivation trees of the graph in Fig. 3. The grammar does not generate graphs with $n \times m$ fully-connected Matryoshkas (with both $n, m > 1$). Extending the type of graphs supported can be done in the future.

### 4.2.1 Split of MultiPipes

The split operation has effect on one of the rightmost Matryoshkas present in the WINDFLOW application, i.e., one that has not already been split or merged before and that is not closed (without a final Sink). We introduce the notion of splitting descriptor $\zeta \in Z$ as a triple $\zeta = <M, \{op\}_{i=1}^{n}, f_s>$ with the following fields:

- $M$ is one of the rightmost Matryoshkas, the one where we want to apply the split;
- $\{op\}_{i=1}^{n}$ is the set of $n > 1$ operators used to create the Matryoshkas that will receive the output values produced by $M$. The destinations of the split are called branches;
- $f_s$ is the splitting function (Section 2.2.2) returning for each input $t$ a pair $<t, i>$ where $i = 0, 1, \ldots, n - 1$ is the index of the branch where $t$ is delivered. In case of multicast/broadcast distributions, $f_s$ returns more pairs $<t, i>$ with different $i$ values for the same $t$.

The splitting is defined as $\text{split} : Z \times \Gamma \rightarrow \Gamma$, and it is applied to the topmost MultiPipe describing the whole application. The semantics is stated by the rules in Fig. 9. The first two rules call recursively the split operation in the right-hand side. The third and fourth rule are applied to terminal symbols (Matryoshkas). The rule “split-noeffect” is applied when the visit reaches a terminal symbol that is not the Matryoshka where the split must be applied, or, if it is the right one, it has already been terminated by a Sink. In both cases, the split operation does not have effect. The last rule split$_3$ applies the split. The result is a new A2A where the Matryoshka $M$ is the only block in the left-hand side, while the right-hand side is composed by $n > 1$ new Matryoshkas, each created starting from the corresponding operator in the splitting descriptor. The outputs delivered by the replicas of the last operator in $M$ (op in Fig. 9) must first be properly routed to one of the branches based on the splitting function, and then to one of the replicas within the destination operator based on its distribution function $f_d$. This is done by combining with the rightmost SBBs in $M$ a fan-out combiner $\Sigma'$ defined starting from $f_s$ and $\{f_{d_i}\}_{i=1}^{n}$.

### 4.2.2 Merge of MultiPipes

The merge operation unifies the output streams from independent MultiPipes, or from different rightmost Matryoshkas of the same MultiPipe, into a unique flow of values to be processed by next operators. We identify two different cases:

- **merge − ind**: $\Gamma \times OP \rightarrow \Gamma$ models the merge of $n > 1$ independent MultiPipes into a new MultiPipe. Independent means that the MultiPipes are not part of the same outermost MultiPipe (i.e., they are all at the topmost level). This operation takes as input the MultiPipes $\Gamma_1, \ldots, \Gamma_n$ and the operator that will be fed by the union of their coming streams;
- **merge**: $\Sigma \times \Gamma \rightarrow \Gamma$. We introduce the notion of merge descriptor $\xi \in \Sigma$ defined as a pair $\xi = <\{M_i\}_{i=1}^{n}, op >$. Given a MultiPipe $\Gamma$, we want to merge the subset $\{M_i\}_{i=1}^{n}$ of its $n > 1$ rightmost Matryoshkas. The unified stream will feed the new operator $op$.

The semantics is based on the rules in Fig. 10. They use the utility function $\text{Rmost} : \Gamma \rightarrow \mathcal{P}(M)$ to get the set of the rightmost Matryoshkas within a MultiPipe. The rule

2. Although the grammar (2) allows a split with one branch only, this case is avoided in the API because it does not have a practical merit.

3. The case $n = 1$ is allowed by the grammar (2) although it is not useful. The API avoids $n$ to be equal to 1.
“merge-ind” merges independent MultiPipes. The idea is depicted in the first example (left) of the figure. The result of the merge is an A2A having \( G_1 \) and \( G_2 \) in the left-hand side and the new Matryoshka in the right-hand side with the new operator receiving the unified stream. To do the distribution, a Seq running the distribution logic of the operator is combined with all the rightmost SBBs within \( G_1 \) and \( G_2 \). To be correct, independent MultiPipes can be merged if none of their rightmost Matryoshkas are closed (terminated by a Sink). We check this with the utility function noSink: \( \Gamma \rightarrow \{\text{True, False}\} \). Its formal semantics is straightforward, and we omit it for brevity. If noSink returns False in at least one of the MultiPipes to be merged, merge - ind is undefined (this is avoided by the API).

The other rules apply within a MultiPipe to merge some of its rightmost Matryoshkas \( \{ M_i \}_{i=1}^n \). The rule “merge-noeffect” does not perform any change, because we recursively reached a sub-structure not involved in the transformation. The rule “merge1” applies recursively the merge on the right-hand side, while rule “merge2” applies recursively the merge on one branch \( G_j \) in the right-hand side, because \( \{ M_i \}_{i=1}^n \) are all in \( G_j \). Rule “merge1” applies the merge when \( \{ M_i \}_{i=1}^n \) are all the rightmost Matryoshkas of \( n > 1 \) MultiPipes that are sibling branches of the same split. In the second example in Fig. 10, \( M_{2,3,4} \) are all the rightmost Matryoshkas of the second and the third branch of the same split. The merge operation creates a new MultiPipe \( \Gamma \) having \( M_2 \) and \( M_2 \) in the left-hand side and the new Matryoshka (with the operator receiving the merged stream) in the right-hand side. Then, \( \Gamma_1 \) is restructured to have \( M_1 \) and \( \Gamma' \) in its right-hand side. All the rightmost SBBs within \( M_2 \) and \( M_2 \) are combined with a Seq running the distribution logic of the new operator. The last example is similar, where we want to merge the rightmost Matryoshkas of all the sibling branches of the same split. Rules “merge” does not modify the MultiPipe, because the requested merge operation is not supported.

4.3 PipeGraph

The PipeGraph is the environment where the user can add Source operators, get access to the MultiPipes to be filled with new operators, and apply merge/split operations. The PipeGraph maintains the list of MultiPipes and their relationships. When the user wants to merge/split MultiPipes, the PipeGraph functionalities check if the transformation can be applied according to the semantics rules. The run() method builds the structure in terms of BBS, instantiates the corresponding threads (see the next part), and starts the application until the sources terminate generating the streams and all the tuples are processed.

4.4 Implementation

In this section, we present the implementation details of the BBS. WINDFLOW leverages the BBS implementation available in the FastFlow library (ver. 3.0) [18]. Since the BBS can be useful to develop other streaming libraries, they have been implemented in a separate software layer that extends, and specializes for the streaming domain, the one provided by FastFlow. In the ensuing description, we refer to a node as a SBB (a sequential wrapper or a combiner) not used within another combiner block. A node is a parallel execution entity receiving inputs from its incoming streams and delivering outputs to its out-coming streams.

4. FastFlow is an open-source parallel programming library available at https://github.com/fastflow
Concurrency Model. BBs are a layer decoupling the runtime design from its implementation. Nodes can be implemented in different ways, e.g., by dedicated threads, or through logical executors like Actors [20] or Tasks [21] that are scheduled on a pool of threads. Our current implementation is based on thread-based parallelism – each node is executed by a dedicated thread – which is also the model adopted by Storm, Flink and by some research prototypes [8]. This avoids the scheduling overhead of logical executors and allows the use in the operator functions of blocking calls with external devices/systems (e.g., key-value storage and logging systems) that may be inefficient with logical executors. However, in this model the use of more threads than the available cores leads to over-subscription and to time-sliced execution which might generate reduced performance. Chaining (available in WindFlow and in Flink) is a practical approach to mitigate this problem by fusing replicas of different operators into the same thread.

Data Forwarding. Streams are implemented by Single-Producer Single-Consumer (SPSC) lock-free queues, where the positions are memory pointers and atomic instructions protect the accesses to the queue without locks [22]. This lock-free design has been proved to be very efficient [23]. While the default queue used in WindFlow has a bounded capacity, the runtime can be configured to use an unbounded lock-free version [24] leveraging the bounded SPSC queue as memory buffers. The overhead of dynamic allocation of memory buffers is mitigated by keeping them in a fixed size cache. In terms of concurrency control, the queue supports blocking and non-blocking policies [25]. In the blocking policy, when the queue is empty or full (in case of fixed size capacity), the thread can be put to sleep on a condition variable, while for the non-blocking policy, it performs a busy-waiting loop with a small back-off between retries to improve responsiveness. The blocking variant is similar to the behavior of existing SPSs, while the lock-free non-blocking approach allows achieving highest performance as long as the number of threads does not exceed the number of physical cores. This will be shown in Section 5.

Thread Pinning and Mapping. In the WindFlow runtime, threads are pinned onto specific cores of the machine. This is in contrast with the design of traditional SPSs, where threads are scheduled by the OS. However, finding the best way to map threads onto cores is a complex problem, and strongly dependent on the application and the platform. Complex heuristics, based on profiling of the operator functions, have been developed in the past [8] and are orthogonal to our work since they can be adopted in any framework. Instead, we use an approach that is application and platform agnostic (and so not necessarily giving the best mapping). The idea is to rely on the fact that operators cooperate according to the producer-consumer paradigm, and that it is generally more efficient to execute communicating replicas on sibling cores sharing some levels of cache to reduce the data forwarding latency and the overhead of data accesses. Our approach leverages the structure of the applications in terms of nested BBs. Cores are assigned to nodes through a sort of Depth First Search visit of the BBs derivation tree. Specifically, in the Pipe, cores are assigned to the internal blocks in a leftmost manner, while in the A2A the cores are assigned visiting the blocks from the left to the right in an interleaved manner, since blocks within the same parallel container do not communicate with each other. In Section 5, we will show the effectiveness of this heuristic strategy.

5 Evaluation

We present a performance comparison between WindFlow and traditional research SPSs. We compare with Storm (version 2.1.0) and Flink (version 1.9.0), two traditional SPSs based on the continuous streaming model (we disabled their fault-tolerance support since we execute on a single machine). We extend the analysis with a comparison with the scale-up prototype BriskStream [8] (still based on the JVM). Although the use of a system-level programming language like C++ brings a certain performance improvement per se, this comparison is useful to assess the potential of WindFlow with respect to existing (and in some cases) widely utilized tools. However, to make the performance evaluation stronger, we also compare with StreamBox [10], a C++-based SPS that uses a different streaming model (morsel-driven parallelism) [9].

We use Java 11.0.5 and gcc 9.0.1 (-O3 optimization flag turned on). The machine is a two CPUs AMD EPYC 7551 with 128GB of RAM. Each CPU has 32 cores (64 hardware threads) with groups of four cores sharing an L3 cache of 8MB. Each core has a clock rate of 2.4 GHz and an L2 of 512KB. Except for one specific experiment, we keep the logical thread contexts disabled to have stabler results. For all Java tests, we configured the maximum heap space to 32 GB in order to avoid memory shortage errors in reading the input datasets. No significant performance difference has been measured with larger heap sizes.

5.1 Applications

We analyze seven applications5 used in the literature [2] and whose DAGs are in Fig. 11. FraudDetection (FD) applies a Markov model [26] to calculate the probability of a credit card transaction being a fraud. SpikeDetection (SD) finds out the spikes in a stream of sensor readings using a moving-average operator and a filter. TrafficMonitoring (TM) processes a stream of events emitted from taxis in the city of Beijing. An operator leverages a geospatial library for detecting the road traveled by the vehicle given its GPS coordinates in the input event. The next operator computes the updated per-road average speed given the vehicle's speed and road ID received in each input event. WordCount (WC) counts the number of instances of each word present in a text file. An operator splits the sentences into words, another operator counts the word instances. The Yahoo! Benchmark (YB) emulates an advertisement application. The goal is to compute 10-seconds windowed counts of advertisement campaigns that have the same type. Linear-Road (LR) emulates a tolling system for the vehicle expressways. The system uses a variable tolling technique [27] accounting for traffic congestion and accident proximity to calculate toll charges. Finally, VoipStream (VS) has been used in the evaluation of BlockMon [28]. It detects

5. The source code is publicly available in GitHub: https://github.com/ParaGroup/StreamBenchmarks.
telemarketing users by analyzing call detail records using Bloom filters.

The applications have been originally developed in Java. For porting them into WINDFLOW, we translated the source code in C++17. We maintained the code identical by replacing the use of Java containers and hash tables with the equivalent ones available in the C++ Standard Template Library. Only for the TM application, we needed to translate the calls to an external library (GeoTools) not available in C++ into the equivalent calls available in GDAL, a C++ library for geo-spatial data. For VS, the original source code for STORM utilizes different stream identifiers to enable different distribution strategies for tuples transmitted onto the same physical connection. To represent this in WINDFLOW, we modified the graph in order to be semantically equivalent but representable in the library. The same version (shown in the figure) has been implemented in FLINK and STORM to have a fair comparison. Fig. 11 also shows the tuple distribution policies between operators (BD, FW and KB stand for broadcast, forward and keyby, respectively).

We point out that the selected applications belong to different domains. While some of them are representable with relational algebra operators (e.g., SD and LR), others involve the processing of unstructured data and user-defined functions. For example, VS instantiates several Bloom filters, FD instantiates a Markov model through a custom user-defined class, WC processes unstructured data (i.e., texts) while TM implements a stateful filter using as predicate the outcome of the evaluation of a geospatial library. Therefore, most of the applications are not expressible with relational algebra languages for streaming, like CQL and its dialects. Furthermore, in terms of code productivity, the implementation of the seven applications in STORM/FLINK/WINDFLOW consists of approximately the same amount of lines of code.

5.2 Building Blocks Utilization

Table 3 reports the number of BBs used in the seven applications. They are automatically created and composed following the formal transition rules described in Section 4. The application programmer is only involved in the definition of the operator business logic code, the used data structures, and in interconnecting operators through add/chains and split/merge operations on MultiPipes. The chosen applications allow us to test all the BBs, composing them in complex patterns. This is especially true in the case of the last two applications (LR and VS). Since they are both based on a complex DAG, they require a rich composition of BBs. Although there is not a unique way to compose our BBs to represent a given DAG, the table reports the number of used blocks based on the transition rules and the definitions given before, which represent the design choice adopted by the WINDFLOW runtime.

5.3 Throughput Analysis

We first evaluate the applications configured with one replica (i.e., thread) per operator. The sources generate tuples at maximum speed for 300 seconds and each run is repeated 50 times. The best configuration for FLINK is to use one task-manager process and to run its streaming environment within the same JVM of the calling program. In this way, all the threads share the same memory space by avoiding inter-process communications [29]. For BRISKSTREAM, the optimization proposed in [8] finds the best mapping of operators onto the cores to reduce remote memory accesses. Although bringing some performance improvements, we did not use this feature for two reasons: i) we are interested in the performance of the raw runtime system, while this optimization would be useful in all the SPSs, not only in BRISKSTREAM; ii) this optimization is hard-coded for the experiments and the machines used in [8], since it needs profiling of the operators in a representative workload before the real run (which is not always realistic) and
requires manual profiling of the memory latencies. For this reason, we evaluated BriskStream in its standard execution mode.

Fig. 12a reports the throughput and Fig. 12b the speedup of WindFlow against the other SPSs. On average, WindFlow is 11.5, 9.2 and 9.8 times faster than Storm, Flink and BriskStream. The largest speedup is with TM, where the improvement also depends on the differences between the two libraries used (GeoTools versus GDAL). In general, Flink is better than Storm, while BriskStream outperforms Storm in all the applications. We found that one of the main reasons for the superior throughput advocated by BriskStream in [8] is the use of jumbo tuples, while Flink and Storm process tuples one-at-a-time to minimize latency. To have a fair comparison, and to assess the performance of BriskStream in processing the streams in a continuous fashion, the results in the figure are collected using jumbo tuples of one item only. The use of larger jumbo tuples will be described later in this section.

**Use of Parallel Operators.** Finding the optimal replication plan (amount of replicas for each operator) is a complex task. We use an approximate method where we measure the processing time per tuple spent in each operator and we compute its selectivity (number of outputs produced per input). Based on that, we compute the weighted average of the operator processing time on each tuple. To have a fair comparison, and to assess the performance of BriskStream in processing the streams in a continuous fashion, the results in the figure are collected using jumbo tuples of one item only. The use of larger jumbo tuples will be described later in this section.

Use of Parallel Operators. Finding the optimal replication plan (amount of replicas for each operator) is a complex task. We use an approximate method where we measure the processing time per tuple spent in each operator and we compute its selectivity (number of outputs produced per input). Based on that, we compute the weighted average of the operator processing time on each tuple. To have a fair comparison, and to assess the performance of BriskStream in processing the streams in a continuous fashion, the results in the figure are collected using jumbo tuples of one item only. The use of larger jumbo tuples will be described later in this section.

**Operator Chaining.** In Flink and WindFlow the operators with the same number of replicas and connected with a forward distribution can be chained. Chained operators are executed by one thread by replacing data forwarding through queues with function calls. While this transformation reduces pipeline parallelism, it allows a potential better utilization of the CPU cores provided that the performance of the chained operators scales with more replicas. Fig. 11 shows the groups of chainable operators with a red dashed

![Fig. 12. Comparison between WindFlow against Storm, Flink, and BriskStream. Throughput and speedup with sequential and parallel configurations of the operators (one tuple-at-a-time processing).](image1)

![Fig. 13. Impact of chaining.](image2)

| TABLE 4 Use of Parallel Operators (Scalability) |
|---|---|---|---|---|---|---|---|
| | FD | SD | TM | WC | YB | LR | VS |
| Storm | 17.7 | 2.75 | 26.4 | 5.20 | 11.1 | 1.27 | 1.06 |
| Flink | 24.8 | 2.38 | 21.6 | 6.48 | 8.46 | 1.23 | 1.39 |
| BriskStream | 17.0 | 6.14 | 16.3 | 8.57 | 12.0 | 1.30 | 1.20 |
| WindFlow | 27.1 | 2.70 | 38.2 | 3.25 | 6.48 | 1.76 | 1.46 |

Fig. 13. Impact of chaining.
box. We run each application with various replication degrees for each chained box and we report in Fig. 13 the improvement/deterioration obtained in the best case.

In FD the effect is marginal since the additional cores saved by chaining the sink with the predictor do not help to increase throughput (FD has a scalability of about 27 even if the architecture has 64 cores). In VS and LR chaining is detrimental since the chainable operators have a negligible impact on the overall application. In VS most of the operators communicate with a keyby distribution and cannot be chained, while in LR throughput is limited by the broadcast distribution after the dispatcher. In the other applications, chaining allows increasing throughput significantly, since the operators are very fine grained and handle high input rates, and chaining reduces communication overheads.

**Impact of Small Batching.** To increase throughput, BriskStream can be configured to batch inputs into jumbo tuples. The reduction of the processor stalls achieved with batching [8] is obtained by computing the whole jumbo tuple within the same user function in the operators, which implies that the original source code needs to be modified to enable this optimization. We study the impact of this small batching in WindFlow by implementing in WC the same batching scheme applied with jumbo tuples of \( b > 0 \) items in BriskStream (where \( b = 1 \) means that each jumbo tuple contains one input only). We select WC because the splitter operator has a high selectivity (it produces 11.3 words per sentence on average) and it is the application in our suite that most benefits from the use of jumbo tuples. The results are shown in Fig. 14(left) with one replica per operator.

Both the SPSs benefit from processing batches within a single operator function to reduce processor stalls as advocated in [5]. Our library is 3.3 times faster than BriskStream in processing items one-at-a-time. It maintains about the same advantage by using jumbo tuples. Batches of \( b = 1 \div 8 \) are typical values adopted in prior works. To highlight WindFlow efficiency, we observe that BriskStream with \( b = 8 \) obtains similar performance to WindFlow with \( b = 1 \). Although this is not a general result, WindFlow is a fast tuple-at-a-time SPS, whose throughput can be further enhanced with small batching.

**Use of Concurrency Control Mechanisms.** BBs are provided with two concurrency control mechanisms (blocking versus non-blocking) to handle synchronizations on the lock-free queues implementing the channels (see Section 4.4). Non-blocking synchronizations (used in the previous experiments) enhance responsiveness, by aggressively checking if push/pop operators succeed. This busy-waiting loop might impair performance when more threads run on different contexts of the same physical core. The busy-waiting thread contributes to fill the core pipeline with “useless” instructions that may interfere with the useful work of the other thread running on the same core. We present a study of this effect in Fig. 14(right). We selected TM because it is the most coarse-grained application in the suite. In the previous analysis, we looked for the best parallel configuration by having at most one thread per core. For TM, the highest throughput is achieved with 50 replicas of the Map-Matcher operator while the remaining 14 cores are used by the other three operators. The figure shows that in this configuration, the non-blocking policy obtains a small improvement (3 percent). We repeated the experiments by using all the logical cores of the machine (i.e., 128). In this configuration, some heavily utilized threads of the Map-Matcher operator are placed on the same core of some replicas of the other operators, which are less utilized and, in the non-blocking configuration, spend a significant fraction of time in the busy-waiting loop of their input queues. In this case, the use of the blocking mechanism increases the throughput of 12 percent, since it delays the execution of idle threads until they can do useful work. The library is configured to use the non-blocking policy if the number of threads does not exceed the available physical cores, otherwise it switches to the blocking mode.

**Effect of Thread Mapping and Pinning.** As stated in Section 4.4, WindFlow comes with a thread mapping and pinning strategy driven by the BBs structure. This mapping is based only on the topological connections between threads and does not leverage statistics of previous runs of the application as in more sophisticated strategies [8]. However, our policy does not need profiling of the application before its actual run. Fig. 15 reports the normalized throughput of the applications executed in WindFlow without any mapping policy (threads are scheduled by the OS) compared with the previous results with mapping/pinning enabled. The results show that for the first three applications, the throughput loss is small (3.5 percent) while it becomes significant in the other applications (43 percent). In general, with more complex DAGs involving broadcast distributions (like in LR and VS), our mapping strategy plays an important role, where communicating threads are placed in sibling cores potentially sharing the L3 cache. We compared the L3 cache misses using the AMDuProf [30] profiling tool officially provided by AMD. We selected LR for this analysis. With threads scheduled by the OS, the application issued 9.12 misses/k events while only 1.89 misses/k events are issued with mapping/pinning enabled. Interestingly, even...
without this optimization WindFlow is still faster than the other considered SPSs in all the seven applications.

**Impact of Different Composition Rules.** The transition rules in Section 4 avoid centralization points in the distribution of outputs to the next operator, and they try to reduce thread oversubscription by combining on the same threads the tuple distribution tasks and the operator’s functional logic. Experimentally, we have found that this strategy achieves better performance results. However, other transition rules could be used. We consider two alternative rules: i) every time a shuffle connection is needed (e.g., for keyby or broadcast tuple distributions), the runtime could use dedicated threads to perform the distribution, by removing the cost of this activity from the threads executing the replicas of the previous operator; ii) in the same case, the runtime could use one centralized thread performing the distribution of inputs to the replicas of the next operator. Both cases require to modify the transition rule “add-shuffle” in Fig. 7. Fig. 17 shows the effect of the composition rules (the WindFlow ones and the two discussed alternatives) on the structure of the SD application (we denote the distribution task with $D$).

We report in Fig. 18 the relative throughput obtained by these two alternative composition rules compared to WindFlow’s rules. The results previously obtained in Fig. 12c serve as a baseline. The use of dedicated threads for the distribution logic can be conceptually useful in specific cases where the distribution is a bottleneck and the previous operator cannot be sufficiently replicated. However, in practice, it proved to be useless and even detrimental for performance, because the number of threads increases significantly, and oversubscription leads to time sliced execution which induces overheads. The use of centralized distribution threads alleviates this problem. However, it reduces performance for fine-grained applications because it causes a throttling in the streaming flow, and throughput is limited by the output rate of the centralized distributors. Interestingly, the only application not suffering from this choice is TM, because the MapMatcher operator is a major bottleneck and the distribution tasks are not critical. We point out that one of the benefits of our BBs-based approach is the flexibility to easily implement new rules in the runtime system, and eventually to make the end user of the library able to select one specific composition approach by providing high-level hooks in the API for choosing the desired behavior.

### 5.4 Latency Evaluation

We present an analysis of the end-to-end latency, measured as the elapsed time from when the input has been generated/received by the source (represented by the tuple’s timestamp field) to its eventual arrival at the sink. The idea is to generate inputs from the sources alongside with a timestamp representing the time at which they have been generated. Operators along the path from a source to a sink, copy the timestamp of the input item into the timestamp of all outputs produced by the same input. Finally, we collect statistics shown in Fig. 16. Boxplots report the 5th, 25th, 50th, 75th and 95th percentile of the latency. We chose five out of seven applications for this analysis. We excluded TM, because it is based on a different external library, and YB, which is based on 10-seconds windowed counts that make the latency values similar among the systems.

The latency is affected by the specific features of the SPSs, like the size of the message queues. To have a fair comparison, the applications have been run with a controlled input rate of $10K$ tuples/second, and we fixed the size of the message queues to $32K$ entries. The results show that WindFlow provides small and stable latency values. The BriskStream latency (in its best case with jumbo tuples equal to 1) is very similar to Storm (except in FD and VS). Among the traditional SPSs, Flink provides the lowest latency. On average,
the mean latency obtained by WindFlow is 12.7, 8.49 and 9.28 times smaller than the one of Storm, Flink and BriskStream. Furthermore, we point out that WindFlow is able to achieve a significantly lower tail latency, which is of great importance for latency-sensitive streaming domains like trading and real-time intrusion detection.

**Latency Breakdown.** In this final part, we would like to break down the latency to identify the impact of the user code within each operator, which has been translated from Java to C++, and the time spent in the runtime system code (e.g., during the distribution and collection phases). For this reason, we repeated the latency measurement on LR and VS. To nullify the enqueuing time before operators, results have been collected using a low input rate of 20 inputs/second. Fig. 19 shows the latency breakdown on WindFlow and Flink, the two SPSs providing the smallest latency values on these two applications.

The system time dominates latency. The user time is reduced by 5.2 and 1.3 times in LR and VS, while the system time is reduced by 5.5 times for both LR and VS. Therefore, besides a faster operator code thanks to the translation from Java to C++, WindFlow has a faster runtime system enabling efficient continuous stream processing.

### 5.5 Comparison With C++ SPSs

In this final part, we compare with the C++-based SPS StreamBox [10]. StreamBox uses the morsel-driven parallelism model [9], where inputs are buffered in batches of records (called bundles), and dynamically scheduled for processing on a pool of threads. In StreamBox, the scheduling activity of bundles is performed in a centralized fashion, by leveraging lock-based primitives.

For the comparison we use the WC application, which was developed by the authors of StreamBox. We modified the WindFlow version in order to split strings using the same functions used by StreamBox (based on Libcpp instead of Libcpp). This improves the throughput with respect to the previous experiments with WC. The results are shown in Fig. 20 with two different bundle sizes of $b = 10$ (small) and $b = 100$ (the default one). For WindFlow, the value of $b$ corresponds to the size of jumbo tuples. In both systems, the two concepts have the same meaning: operators exchange messages containing several inputs (e.g., sentences between source and splitter, words between splitter and counter). We plot the throughput (in MB/s) achieved with different number of cores on our machine.

WindFlow provides better throughput and scales well up to 16 and 32 cores, with small and larger batches respectively. StreamBox, due to its centralized scheduling, scales ideally up to few cores. Its maximum scalability with ($b = 100$) is of 14.35 with 32 cores, while in the same scenario the scalability of WindFlow is of 27.8. StreamBox exhibits latency times of at least one order of magnitude greater than WindFlow ones (results are omitted for brevity).

### 6 Related Works

The work in [5] analyzes two main inefficiencies of traditional SPSs when executed on a single multi-core architecture. First, the large instruction footprint between consecutive invocations of the operator business logic code. Second, the cost of remote memory accesses in NUMA machines. To mitigate them, BriskStream [8] proposes the use of jumbo tuples and a profile-based mapping strategy of operators onto physical cores. WindFlow shows superior performance thanks to both the lock-free design of our BBs and the use of combiner blocks to reduce thread oversubscription. Furthermore, the mapping strategy of BriskStream is generally hard to be configured by the end users developing applications. WindFlow, instead, has an implicit mapping strategy driven by the BBs composition, which is transparent to programmers using WindFlow and does not require offline profiling of the operators.

Another family of SPSs for scale-up systems adopt the morsel-driven model [9]. StreamBox [10] makes use of lock-based primitives to protect scheduling phases of batches onto threads, and indeed exhibited limited scalability compared with WindFlow. A recent work still adopting morsel-driven parallelism has been described in [11]. It advocates a code generation approach to improve performance by fusing in a single tight loop several operators in pipeline. However, although useful for specific and important cases, such generative approach has some limitations in practice. First of all, the fact that it can only be applied when applications are expressed in terms of a declarative approach like SQL, where operators are the ones of relational algebra. This allows a high-level description of the query, that can be compiled by generating efficient runtime code. However, applications with generic DAGs and generic stateful operators with a user-defined state definition cannot be expressed. Indeed, the only stateful operator supported is aggregation (performed using sliding windows and standard aggregation functions).

An interesting research direction is to design new SPSs exploiting GPUs and FPGAs. For GPUs, the two main contributions are Saber [17] and FinaStream [31]. They adopt the
same model of StreamBox. Both works support relational algebra queries and stateful operators, which are mostly window-based operators performing aggregation. When compared with traditional SPSs like Storm and Flink, they exhibit at least one order of magnitude higher throughput. However, GPU processing for streaming applications is still limited to the ones in the domain of relational algebra queries, while no support for general-purpose stateful stream processing (actually supported by both Storm and Flink) has been provided nor discussed. When comparing systems, both performance and expressive power should be considered. In this sense, the general-purpose model of WindFlow can be a valuable candidate to support general streaming on GPUs in the future. The same model can also be exploited to support FPGAs. Indeed, one prior work [32] in this regard still supports the sole compilation of relational algebra queries on FPGAs.

Some recent C++ systems for big data computations are PiCo [33] and Thrill [34]. They target batch processing and distributed architectures. An interesting SPS for distributed architectures is StreamMine3G [35]. This system has some commonalities with WindFlow (e.g., a C++ interface and the possibility to customize operators with user-defined code), but they target different scenarios. While WindFlow is targeting a single multicore, StreamMine3G focuses on orthogonal properties like elasticity and fault-tolerance. The two approaches are thus complementary. The lightweight fault-tolerance approach developed in StreamMine3G can be adapted to our Building Block design in the future, to target distributed domains.

7 Conclusion

WindFlow is a C++ library for data stream processing on multicores. The design of its runtime system has been done using a formal approach based on BBs, whose combinations are based on a formal semantics. These rules model important features of streaming applications, like shuffle communications and operator chaining. The experimental evaluation has been done against traditional and research SPSs. More specifically, WindFlow is in the worst/average/best case 2.23/11.5/48 times faster than Storm, 1.15/9.2/47 times faster than Flink, and 1.28/9.8/50 times faster than BriskStream. Furthermore, it exhibited twice the scalability of the C++ micro-batching SPS StreamBox on a selected benchmark application.

In the future, we would like to investigate the full potential offered by our novel BBs abstraction layer. One critical issue in most of the SPSs is the right configuration of streaming applications in terms of both chaining and parallelism per operator, which requires a lot of effort by the application programmer. Moreover, our BBs could be used to design Machine Learning predictive approaches built on the structured domain of their possible composition and nesting. This could help in the development of improved auto-tuning approaches.

Acknowledgments

This work was supported in part by the European H2020 Project TEACHING under Grant 871385.

References

[1] A. Arasu, S. Babu, and J. Widom, “The CQL continuous query language: Semantic foundations and query execution,” VLDB J., vol. 15, no. 2, Jun. 2006, Art. no. 121142.
[2] M. V. Bordin, D. Griebler, G. Mencagli, C. F. R. Geyer, and L. G. Fernandez, “DisFlow: A suite of benchmark applications for distributed data stream processing systems,” IEEE Access, vol. 8, pp. 222900–222917, Dec. 2020.
[3] “Apache storm,” Accessed: Feb. 26, 2020, [Online]. Available: http://storm.apache.org/
[4] “Apache flink,” Accessed: Feb. 26, 2020, [Online]. Available: https://link.apache.org/
[5] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinz, “Revisiting the design of data stream processing systems on multi-core processors,” in Proc. 2017 IEEE 33rd Int. Conf. Data Eng., Apr. 2017, pp. 659–670.
[6] Z. Li, H. Shen, and L. Ward, “Accelerating big data analytics using scale-up/out heterogeneous clusters,” in Proc. 2019 28th Int. Conf. Comput. Commun. Netw., 2019, pp. 1–9.
[7] A. Addisie and V. Bertacco, “Collaborative accelerators for in-memory MapReduce on scale-up machines,” in Proc. 24th Asia South Pacific Des. Automat. Conf., New York, NY, USA, 2019, p. 747–753.
[8] Ş. Zhang, J. He, A. C. Zhou, and B. He, “BriskStream: Scaling data stream processing on shared-memory multicore architectures,” in Proc. Int. Conf. Manage. Data, New York, NY, USA, 2019, pp. 705–722.
[9] V. Leis, P. Boncz, A. Kemper, and T. Neumann, “Morse-driven parallelism: A NUMA-aware query evaluation framework for the many-core era,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, New York, NY, USA, 2014, p. 743–754.
[10] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, “Streambox: Modern stream processing on a multicore machine,” in Proc. 21st Conf. Useixin Annu. Tech. Conf., Santa Clara, CA, USA, 2017, p. 617–629.
[11] G. Theodorakis, A. Koliouis, P. R. Pietzuch, and H. Pirk, “LightSaber: Efficient window aggregation on multi-core processors,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, Portland, OR, USA, 2020, pp. 2505–2521.
[12] G. Mencagli, M. Torquati, D. Griebler, M. Danelutto, and L. G. Fernandez, “Raising the parallel abstraction level for streaming analytics — c++,” IFAC Proc. Volumes, vol. 52, no. 1, pp. 131944–131961, Sep. 2019.
[13] “Programming languages — c++,” Draft International Standard N4660, ISO/IEC, Geneva, Switzerland, Mar. 2017. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4660.pdf [https://web.archive.org/web/20170325052926/ http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4660.pdf]
[14] H. C. M. Andrade, B. Gedik, and D. S. Turaga, Fundamentals of Stream Processing: Application Design, Systems, and Analytics, 1st ed. New York, NY, USA: Cambridge Univ. Press, 2014.
[15] B. Gedik, “Generic windowing support for extensible stream processing systems,” Softw. Pract. Exper., vol. 44, no. 9, pp. 1105–1128, Sep. 2014.
[16] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, “NiagaraCQ: A scalable continuous query system for internet databases,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, New York, NY, USA, 2000, p. 379–390.
[17] A. Koliouis, M. Weidlich, R. C. Fernandez, A. L. Wolf, and P. Pietzuch, “Saber: Window-based hybrid stream processing for heterogeneous architectures,” in Proc. Int. Conf. Manage. Data, New York, NY, USA, 2016, pp. 555–569.
[18] M. Torquati, “Harnessing parallelism in multi/many-cores with streams and parallel patterns,” Ph.D. dissertation, Univ. Pisa, Italy, 2019.
[19] M. Aldinucci, S. Cappa, M. Danelutto, P. Kilpatrick, and M. Torquati, “Design patterns percolating to parallel programming framework implementation,” Int. J. Parallel Prog., vol. 42, no. 6, pp. 1012–1031, Dec. 2014.
[20] D. Charousset, R. Hiesgen, and T. C. Schmidt, “Revisiting actor programming in C++,” Comput. Langs., Syst. Structures, vol. 45, pp. 105–131, Apr. 2016.
[21] M. Voss, R. Asenjo, and J. Reinders, C++ Parallel Programming with Threading Building Blocks, 1st ed. New York, NY, USA: Apress, 2019.
[22] J. Giacomoni, T. Moseley, and M. Vachharajani, “Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue,” in Proc. 13th ACM SIGPLAN Symp. Pract. Prac. Parallel Prog., New York, NY, USA, 2008, pp. 43–52.

[23] S. Schneider and K.-L. Wu, “Low-synchronization, mostly lock-free, elastic scheduling for streaming runtimes,” in Proc. 38th ACM SIGPLAN Conf. Program. Lang. Des. Implementation, New York, NY, USA, 2017, p. 648–661.

[24] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, “An efficient unbounded lock-free queue for multico-re systems,” in Proc. Euro-Par Parallel Process., 2012, pp. 662–673.

[25] M. Torquati, D. De Sensi, G. Mencagli, M. Aldinucci, and M. Danelutto, “Power-aware pipelining with automatic concurrency control,” Concurrency and Comput. Pract. Experience, vol. 31, no. 5, p. e4652, 2019.

[26] D. Iyer, A. Mohanpurkar, S. Janardhan, D. Rathod, and A. Sar-deshmukh, “Credit card fraud detection using hidden Markov model,” in Proc. World Congr. Inf. Commun. Technol., Dec. 2011, pp. 1062–1066.

[27] A. Arasu et al., “Linear road: A stream data management benchmark,” in Proc. 13th Int. Conf. Very Large Data Bases, 2004, p. 480–491.

[28] F. Huici, A. di Pietro, B. Trammell, J. M. G. Hidalgo, D. Martinez Ruiz, and N. dHeureuse, “Blockmon: A high-performance composable network traffic measurement system,” SIGCOMM Comput. Commun. Rev., vol. 42, no. 4, p. 79–80, Aug. 2012.

[29] S. et al., “TurboStream: Towards low-latency data stream processing,” in Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst., Jul. 2018, pp. 983–993.

[30] “AMD uProf,” Accessed: Feb. 28, 2020. [Online]. Available: https://developer.amd.com/amd-upprof/

[31] F. Zhang, L. Yang, S. Zhang, B. He, W. Lu, and X. Du, “Finestream: Fine-grained window-based stream processing on CPU-GPU integrated architectures,” in Proc. USENIX Annu. Tech. Conf., Jul. 2020, pp. 633–647.

[32] R. Mueller, J. Teubner, and G. Alonso, “Streams on wires: A query compiler for PICAs,” in Proc. VLDB Endow., Aug. 2009, pp. 229–240.

[33] C. Misale, M. Drocco, G. Tremblay, A. R. Martinelli, and M. Aldinucci, “PiCo: High-performance data analytics pipelines in modern C++,” Future Gener. Comput. Syst., vol. 87, pp. 392–403, 2018.

[34] T. Bingmann et al., “Thrill: High-performance algorithmic distributed batch data processing with C++,” in Proc. IEEE Int. Conf. Big Data, Dec. 2016, pp. 172–183.

[35] A. Martin, A. Brito, and C. Fetzer, “StreamMine3G: Elastic and fault tolerant large scale stream processing,” in Encyclopedia of Big Data Technologies. Cham, Switzerland: Springer, 2018, pp. 1–10.

Gabriele Mencagli is currently an assistant professor with the Computer Science Department, University of Pisa, Italy. He has coauthored about 60 peer-reviewed papers that appeared in international conferences, workshops, journals, and one book. His research interests are in the area of parallel and distributed systems, and data stream processing. He is a member of the Editorial Board of Future Generation Computer Systems and Cluster Computing.

Massimo Torquati is currently an assistant professor of computer science with the University of Pisa, Italy. He has authored or coauthored more than 100 peer-reviewed papers in conference proceedings and journals, mostly in the field of parallel and distributed programming. He has been involved in several Italian, EU, and industry-supported research projects. He is the maintainer and main developer of the FastFlow parallel programming library.

Andrea Cardaci is currently working toward the master’s degree in computer science and networking with the University of Pisa, Italy. He has experience in programming for mobile devices. His research interests include high performance computing and computer security.

Alessandra Fais received the bachelor’s and master’s degrees both from the the Department of Computer Science, University of Pisa. She is currently working toward the PhD degree with the Department of Information Engineering, University of Pisa. Her main research interests include data stream processing applications in the networking domain, high performance network processing, data plane acceleration, SmartNICs, and software defined networks.

Luca Rinaldi is currently working toward the PhD degree with the Department of Computer Science, University of Pisa, Italy. He has coauthored more than ten papers related to his research interests. His research interests include parallel programming, actor-based programming, and high-level languages for parallel computing.

Marco Danelutto is currently a professor with the Department of Computer Science, University of Pisa, Italy. He was responsible for the University of Pisa research unit in different EU funded projects (CoreGRID, GRIDcomp, ParaPhrase, REP-ARA, and RePhrase). He has authored more than 150 papers in refereed international journals and conferences. His main research interests include parallel programming models, particularly in the area of parallel design patterns and algorithmic skeletons.