Stream Processing with Dependency-Guided Synchronization

Konstantinos Kallas*
kallas@seas.upenn.edu
University of Pennsylvania
Philadelphia, PA, USA

Filip Niksic*†
fniksic@gmail.com
University of Pennsylvania
Philadelphia, PA, USA

Caleb Stanford*
castan@cis.upenn.edu
University of Pennsylvania
Philadelphia, PA, USA

Rajeev Alur
alur@cis.upenn.edu
University of Pennsylvania
Philadelphia, PA, USA

Abstract

Real-time data processing applications with low latency requirements have led to the increasing popularity of stream processing systems. While such systems offer convenient APIs that can be used to achieve data parallelism automatically, they offer limited support for computations that require synchronization between parallel nodes. In this paper, we propose *dependency-guided synchronization (DGS)*, an alternative programming model for stateful streaming computations with complex synchronization requirements. In the proposed model, the input is viewed as partially ordered, and the program consists of a set of parallelization constructs which are applied to decompose the partial order and process events independently. Our programming model maps to an execution model called *synchronization plans* which supports synchronization between parallel nodes. Our evaluation shows that APIs offered by two widely used systems—Flink and Timely Dataflow—cannot suitably expose parallelism in some representative applications. In contrast, DGS enables implementations with scalable performance, the resulting synchronization plans offer throughput improvements when implemented manually in existing systems, and the programming overhead is small compared to writing sequential code.

CCS Concepts: • **Software and its engineering** → **Parallel programming languages**; *Domain specific languages*; • **Information systems** → *Stream management*.

*Equal contribution.
†now at Google.

Keywords: data parallelism, sharding, synchronization, distributed stream processing

ACM Reference Format:
Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. 2022. Stream Processing with Dependency-Guided Synchronization. In *27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22), April 2–6, 2022, Seoul, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3503221.3508413

1 Introduction

A wide range of applications in domains such as healthcare, transportation, and smart homes are increasingly relying on real-time data analytics with low latency and high throughput requirements. This has motivated a significant amount of research on *stream processing*, spanning different layers of abstraction in the software stack. At the lowest level, ***stream processing systems*** (e.g. Flink [16], Samza [51], Storm [6], Spark Streaming [63], Trill [17], Heron [37], Beam [7]) handle scheduling, optimizations, and operational concerns. At the intermediate level, ***stream processing APIs and programming models*** (e.g. MapReduce online [19], SPADE [26], SP Calculus [59], Timely Dataflow [47, 50], StreamIt [60], Flink's DataStream API [5]), usually based on a form of dataflow [27, 41], abstract the computation in a way that hides implementation details, while exposing parallelization information to the underlying system. At the top level, ***high-level query languages*** (e.g. Streaming SQL [11, 32], SamzaSQL [53], Structured Streaming [9], StreamQRE [45], CQL [8], AFAs [18]) provide convenient abstractions that are built on top of the streaming APIs. Of these layers, streaming APIs play a central role in the successful scaling of applications since their expressiveness restricts the available parallelism. In this paper we focus on rethinking the dataflow model at this intermediate layer to enable parallel implementations for a broader range of programs.

The success of stream processing APIs based on the dataflow model can be attributed to their ability to simplify the task of parallel programming. To accomplish this, most APIs expose a simple but effective model of data-parallelism called *sharding*: nodes in the dataflow graph are replicated into many parallel instances, each of which will process a different partition of the input events. However, while sharding is intuitive for programmers, it also implicitly limits the scope of parallel patterns that can be expressed. Specifically, it prevents arbitrary *synchronization across parallel instances* since it disallows communication between them. This is limiting in modern applications such as video processing [31] and distributed machine learning [52], since they require both synchronization between nodes and high throughput and could therefore benefit from parallelization. Further evidence that sharding is limiting in practice can be found in a collection of feature requests in state-of-the-art stream processing systems [23, 34, 54], asking either for state management that goes beyond replication or for some form of communication between shards. To address these needs, system developers have introduced extensions to the dataflow model to enable specific use cases such as *message broadcasting* and *iterative dataflows*. However, existing solutions do not generalize, as we demonstrate experimentally in Section 4.2. For the remainder of applications, users are left with two unsatisfying solutions: either ignore parallelization potential, implementing their application with limited parallelism; or circumvent the stream processing APIs using low-level external mechanisms to achieve synchronization between parallel instances.

For example, consider a fraud detection application where the input is a distributed set of streams of bank transaction events. Suppose we want to build an unsupervised online machine learning model over these events which classifies events as fraudulent based on a combination of *local* (stream-specific) and *global* (across-streams) statistical summaries. The problem with the traditional approach is that when classifying a new event, we need access to both the local and the global summaries; but this cannot be achieved using sharding since by default shards do not have access to a global summary. One extension to the dataflow model, implemented in some systems [5, 47] is the *broadcast* pattern, which allows the operator computing the global summary to broadcast to all other nodes. However, broadcasting is restricted since it does not allow bidirectional communication; the global summary needs to be both broadcast to all shards, but also updated by all shards. Cyclic dataflows are another partial solution, but do not always solve the problem, as we show in Section 4.2. In practice, applications like this one with complex synchronization requirements opt to manually implement the required synchronization using external mechanisms (e.g. querying a separate a key-value store with strong consistency guarantees). This is error prone and, more importantly, violates the requirements of many streaming APIs that operators need to be effect-free so that the underlying system can provide exactly-once execution guarantees in the presence of faults.

To address the need to combine parallelism with synchronization, we make two contributions. First, we propose *synchronization plans*, a tree-based execution model which is a restricted form of communicating sequential processes [30]. Synchronization plans are hierarchical structures that represent concurrent computation in which parallel nodes are not completely independent, but communicate with their ancestors on special synchronizing events. While this solves the problem of being able to express synchronizing parallel computations, we still need a streaming API which exposes such parallelism implicitly rather than explicitly. For this purpose, we propose *dependency-guided synchronization* (DGS), a parallel programming model which can be mapped automatically to synchronization plans.

A DGS program consists of three components. First, the user provides a *sequential implementation* of the computation; this serves to define the semantics of what they want to compute assuming the input is processed as a sequence of events. Second, the user indicates which input events can be processed in parallel and which require synchronization by providing a *dependence relation* on input events. This relation induces a partial order on the input stream. For example, if events can be processed completely in parallel without any synchronization, then all input events can be specified to be independent. Third, the user provides a mechanism for parallelizing state when the input stream contains independent events: *parallelization primitives* called *fork* and *join*. This model is inspired by classical parallel programming, but has streaming-specific semantics which describes how a partially ordered input stream is decomposed for parallel processing.

Given a DGS program, the main technical challenge is to generate a synchronization plan, which corresponds to a concrete implementation, that is both *correct* and *efficient*. More precisely, the challenge lies in ensuring that a derived implementation correctly enforces the specified input dependence relation. To achieve correctness, we formalize: (i) a set of conditions that ensure that a program is consistent, and (ii) a notion of P-*valid* synchronization plans, i.e., plans that are well-typed with respect to a given program P. To achieve efficiency, we design the framework so that correctness is independent of *which* synchronization plan is chosen—as long as it is P-valid. The idea of this separation is to enable future work on optimized query execution, in which an optimizing component searches for an efficient synchronization plan maximizing a desired cost metric without jeopardizing correctness. We tie everything together by proving that the end-to-end system is correct, that is, any concrete implementation that corresponds to an P-valid plan is equivalent to a program P that satisfies the consistency conditions.

In order to evaluate DGS, we perform a set of experiments to investigate the data parallelism limitations of Flink [16]—a representative high-performance stream processing system—and Timely Dataflow [50]—a representative system with iterative computation. We show that these limits can be partly overcome by manually implementing synchronization. However, this comes at a cost: the code has to be specialized to the number of parallel nodes and similar implementation details, forcing the user to sacrifice the desirable benefit of *platform independence*. We then develop Flumina, an end-to-end prototype that implements DGS in Erlang [10], and show that it can automatically produce scalable implementations (through generating synchronization plans from the program) independent of parallelism. In the extended version of the paper [36], we also evaluate programmability via two real-world case studies. In particular, we demonstrate that the effort required—as measured by lines of code—to achieve parallelism is minimal compared to the sequential implementation.

In summary, we make the following contributions:

- *DGS:* a novel programming model for parallel streaming computations that require synchronization, which allows viewing input streams as partially ordered sets of events. (Section 2)
- *Synchronization plans:* a tree-based execution model for parallel streaming computations that require synchronization, a framework for generating a synchronization plan given a DGS program, a prototype implementation, and an end-to-end proof of correctness. (Section 3)
- An evaluation that demonstrates: (i) the throughput limits of automatically scaling computations on examples which require synchronization in Flink and Timely; (ii) the throughput and scalability benefits achieved by synchronization plans over such automatically scaling computations; and (iii) the programmability benefits of DGS for synchronization-centered applications (Section 4).

Flumina, our implementation of DGS, is open-source and available at github.com/angelhof/flumina.

2 Dependency-Guided Synchronization

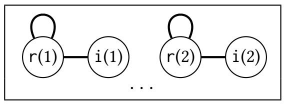
A DGS program consists of three components: a *sequential implementation*, a *dependence relation* on input events to enforce synchronization, and *fork* and *join* parallelization primitives. In Section 2.3 we define program *consistency*, which are requirements on the *fork* and *join* functions to ensure that any parallel implementation generated from the program is equivalent to the sequential one.

2.1 DGS programs

For a simple but illustrative example, suppose that we want to implement a stream processing application that simulates a map from keys to counters, in which there are two types of input events: *increment* events, denoted i(k), and *read-reset* events, denoted r(k), where each event has an associated key k. On each increment event, the counter associated with that key should be increased by one, and on each read-reset event, the current value of the counter should be produced as output, and then the counter should be reset to zero.

```
// Dependence Relation
depends: (Event, Event) -> Bool
depends(r(k1), r(k2)) = k1 == k2
depends(r(k1), i(k2)) = k1 == k2
depends(i(k1), r(k2)) = k1 == k2
depends(i(k1), i(k2)) = false
```

```
// Types
Key = Integer
Event = i(Key) | r(Key)
State = Map(Key, Integer)
Pred = Event -> Bool
```



```
// Sequential Code
init: () -> State
init() =
    return emptyMap()
update: (State, Event)
      -> State
update(s, (i(k), ())) =
    s[k] = s[k] + 1;
    return s
update(s, (r(k), ())) =
    output s[k];
    s[k] = 0;
    return s
```

```
// Fork and Join
fork: (State, Pred, Pred)
      -> (State, State)
fork(s, pred1, pred2) =
    // two forked states
    s1 = init(); s2 = init()
    for k in keys(s):
        if pred1(r(k)):
            s1[k] = s[k]
        else:
            // pred2(r(k)) OR
            //   r(k) in neither
            s2[k] = s[k]
    return (s1, s2)
join: (State, State) -> State
join(s1, s2) =
    for k in keys(s2):
        s1[k] = s1[k] + s2[k]
    return s1
```

Figure 1. DGS program implementing a map from keys to counters. The `depends` relation is visualized as a graph with two keys shown; edges indicate synchronization, while non-edges indicate opportunities for parallelism.

Sequential implementation. In our programming model, the user first provides a sequential implementation of the desired computation. A pseudocode version of the sequential implementation for the example above is shown in Figure 1 (left); Erlang syntax has been edited for readability, and we use `s[k]` as shorthand for the value associated with the key k in the map *or* the default value 0 if it is not present. It consists of (i) the state type `State`, i.e. the map from keys to counters, (ii) the initial value of the state `init`, i.e. an empty map with no keys, and (iii) a function `update`, which contains the logic for processing input events. Conceptually, the sequential implementation describes how to process the data assuming it was all combined into a single sequential stream (e.g., sorted by system timestamp). For example, if the input stream consists of the events i(1), i(2), r(1), i(2), r(1), then the output would be 1 followed by 0, produced by the two r(1) (read-reset) events.

Dependence relation. To parallelize a sequential computation, the user needs to provide a dependence relation which encodes which events are independent, and thus can be processed in parallel, and which events are dependent, and therefore require synchronization. The dependence relation abstractly captures all the dependency patterns that appear in an application, inducing a partial order on input events. In this example, there are two forms of independence we want to expose. To begin with, *parallelization by key* is possible: the counter map could be partitioned so that events corresponding to different sets of keys are processed independently. Moreover, each event is processed atomically in our model, and therefore *parallelizing increments* on the counter of the same key is also possible. In particular, different sets of increments for the same key can be processed independently; we only need to aggregate the independent counts when a read-reset operation arrives. On the other hand, read-reset events are synchronizing for a particular key; their output is affected by the processing of increments as well as other read-reset events of that key.

We capture this combination of parallelization and synchronization requirements by defining the dependence relation `depends` in Figure 1 (also visualized as a graph) (see Section 2.2 for a formal definition). In the program, the set of events may be *symbolic* (infinite): here `Event` is parameterized by an integer `Key`. To allow for this, the dependence relation is formally a *predicate* on pairs of events, and is given programmatically as a function from pairs of `Event` to `Bool`. For example, `depends(r(k1), r(k2))` (one of four cases) is given symbolically as equality comparison of keys, `k1 == k2`. The dependence relation should also be *symmetric*, i.e. `e1` is in `depends(e2)` iff `e2` is in `depends(e1)`; the intuition is that `e1` can be processed in parallel with `e2` iff `e2` can be processed in parallel with `e1`.

Parallelization primitives: *fork* and *join*. While the dependence relation indicates the possibility of parallelization, it does not provide a mechanism for parallelizing state. The parallelization is specified using a pair of functions to `fork` one state into two, and to `join` two states into one. The fork function additionally takes as input two predicates of events, such that the two predicates are *independent* (but not necessarily disjoint): every event satisfying `pred1` is independent of every event satisfying `pred2`. The contract is that after the state is forked into two independent states, each state will then *only be updated using events satisfying the given predicate.* A fork-join pair for our example is shown in Figure 1. The `join` function simply adds up the counts for each key to form the combined state. The `fork` function has to decide, for each key, which forked state to partition the count to. Since read-reset operations `r(k)` are synchronizing, i.e., depend on all events of the same key, and require knowing the total count, it partitions by checking which of the two forked states is responsible for processing read-reset operations, if any.

The programming model *exposes* parallelism, but the implementation (Section 3) determines when to call forks and joins. To do this, the implementation instantiates a synchronization plan: a tree structure where each node is a stateful worker with a predicate indicating the set of events that it is responsible for. Nodes that do not have an ancestor-descendant relationship process independent but not necessarily disjoint sets of events. When a node with children needs to process an event, it first uses `join` to merge the states of its children, and then it `fork`s back its children states using the combined predicates of its descendants, `pred1` for the left subtree, and `pred2` for the right subtree. The implementation can therefore instantiate synchronization plans with different shapes and predicates to enable different kinds of parallelism. For example, to indicate *parallelization by key*, the left child with `pred1` might contain all events of key 1 and the right child with `pred2` might contain all events of key 2. On the other hand, to indicate *parallelization on increments*, `pred1` and `pred2` might both contain `i(3)`, and in this case neither would contain `r(3)` (to satisfy the independence requirement). The latter example also emphasizes that `pred1` and `pred2` need not be disjoint, nor need they collectively cover all events. For the events not covered, in this case `r(3)`, a join would need to be called before an `r(3)` event can be processed. Parallelization can also be done repeatedly; the fork function can be called again on a forked state to fork it into two sub-states, and each time the predicates `pred1` and `pred2` will be even further restricted.

2.2 Formal definition

A DGS program can be more general than we have discussed so far, because we allow for multiple *state types*, instead of just one. The initial state must be of a certain type, but forks and joins can convert from one state type to another: for example, forking a pair into its two components. Additionally, each state type can come with a *predicate* which restricts the allowed events processed by a state of that type. The complete programming model is summarized in the following definition.

Definition 2.1 (DGS program). Let `Pred(T)` be a given type of *predicates* on a type `T`, where predicates can be evaluated as functions `T -> Bool`. A program consists of the following components:

1. A type of input events `Event`.
2. The dependence relation `depends: Pred(Event, Event)`, which is symmetric: `depends(e1, e2)` iff `depends(e2, e1)`.
3. A type for output events `Out`.
4. Finitely many state types `State_0`, `State_1`, etc.
5. For each state type `State_i`, a predicate which specifies which input values this type of state can process, denoted `pred_i: Pred(Event)`. We require `pred_0` = `true`.

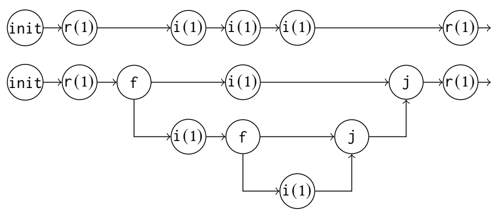


Figure 2. Example of a sequential (top) and parallel (bottom) execution of the program in Figure 1 on the input stream r(1), i(1), i(1), i(1), r(1) (f and j denote forks and joins).

6. A sequential implementation, consisting of a single initial state `init: State_0` and for each state type `State_i`, a function `update_i: (State_i, Event) -> State_i`. The update also produces zero or more outputs, given by a function `out_i: (State_i, Event) -> List(Out)`.
7. A set of parallelization primitives, where each is either a *fork* or a *join*. A fork has type

```
(State_i, Pred(Event), Pred(Event)) -> (State_j, State_k),
```

and a join has type `(State_j, State_k) -> State_i`, for some i, j, and k.

Semantics. The semantics of a program can be visualized using wire diagrams, as in Figure 2. Computation proceeds from left to right. Each wire is associated with (i) a state (of type `State_i` for some i) and (ii) a predicate (of type `Pred(Event)`) which restricts the input events that this wire can process. Input events are processed as updates to the state, which means they take one input wire and produce one output wire, while forks take one input wire and produce two, and joins take two input wires and produce one. Notice that the same updates are present in both sequential and parallel executions. It is guaranteed in the parallel execution that *fork and join come in pairs*, like matched parentheses. Each predicate that is given as input to the `fork` function indicates the set of input events that can be processed along one of the outgoing wires. Additionally, we require that updates on parallel wires must be on independent events. In the example, the wire is forked into two parts and then forked again, and all three resulting wires process i(1) events. Note that r(1) events cannot be processed at that time because they are dependent on i(1) events. More specifically, we require that the predicate at each wire of type `State_i` implies `pred_i`, and that after each `fork` call, the predicates at each resulting wire denote independent sets of events. This semantics is formalized in the following definition.

Definition 2.2 (DGS Semantics)**.** A *wire* is a triple written using the notation ⟨`State_i`, pred, s⟩, where `State_i` is a state type, `s: State_i`, and `pred: Pred(Event)` is a predicate such that `pred` implies `pred_i`. We give the semantics of a program through an inductively defined relation, which we denote ⟨State, pred, s⟩ $\xrightarrow[\texttt{v}]{\texttt{u}}$ ⟨State, pred, s'⟩, where ⟨State, pred, s⟩ and ⟨State, pred, s'⟩ are the starting and ending wires (with the same state type and predicate), `u: List(Event)` is an input stream, and `v: List(Out)` is an output stream. Let `l1 + l2` be list concatenation, and define `inter(l, l1, l2)` if `l` is some interleaving of `l1` and `l2`. For `e1, e2: Event`, let `indep(e1, e2)` denote that `e1` and `e2` are not dependent, i.e. `not(depends(e1,e2))`. There are two base cases and two inductive cases. (1) For any `State`, `pred`, `s`, ⟨State, pred, s⟩ $\xrightarrow[\texttt{[]}]{\texttt{[]}}$ ⟨State, pred, s⟩. (2) For any `State`, `pred`, `s`, and any `e: Event`, if `e` satisfies `pred` then ⟨State, pred, s⟩ $\xrightarrow[\texttt{out(s, e)}]{\texttt{[e]}}$ ⟨State, pred, update(s, e)⟩. (3) For any `State, pred, s, s', s'', u, v, u',` and `v'`, if ⟨State, pred, s⟩ $\xrightarrow[\texttt{v}]{\texttt{u}}$ ⟨State, pred, s'⟩ and ⟨State, pred, s'⟩ $\xrightarrow[\texttt{v'}]{\texttt{u'}}$ ⟨State, pred, s''⟩, then ⟨State, pred, s⟩ $\xrightarrow[\texttt{v + v'}]{\texttt{u + u'}}$ ⟨State, pred, s''⟩. (4) Lastly, for any instances of `State`, `State1`, `State2`, `pred`, `pred1`, `pred2`, `s`, `s1'`, `s2'`, `u`, `u1`, `u2`, `v`, `v1`, `v2`, `fork`, and `join`, suppose that (the conjunction) `pred1(e1)` and `pred2(e2)` implies `indep(e1, e2)`, `pred1` implies `pred`, and `pred2` implies `pred`. Let `fork(s, pred1, pred2) = (s1, s2)` and `join(s1', s2') = s'`. If we have `inter(u, u1, u2)`, `inter(v, v1, v2)`, ⟨State1, pred1, s1⟩ $\xrightarrow[\texttt{v1}]{\texttt{u1}}$ ⟨State1, pred1, s1'⟩, and ⟨State2, pred2, s2⟩ $\xrightarrow[\texttt{v2}]{\texttt{u2}}$ ⟨State2, pred2, s2'⟩, then ⟨State, pred, s⟩ $\xrightarrow[\texttt{v}]{\texttt{u}}$ ⟨State, pred, s'⟩.

Finally, the *semantics* $[\![P]\!]$ of the program P is the set of pairs (u, v) of an input stream u and an output stream v such that ⟨State_0, true, init⟩ $\xrightarrow[\texttt{v}]{\texttt{u}}$ ⟨State_0, true, s'⟩ for some `s'`.

Representing predicates. In the running example, a predicate on a type `T` was represented as a function `T -> Bool`, but note that the programming model above allows other representation of predicates, for example using logical formulas. The tradeoff here is that a more general representation allows more dependence relations to be expressible, but also complicates the implementation of an appropriate `fork` function as it must accept as input more general input predicates. In our implementation (see Section 3), we assume that an event consists of a pair of a `Tag` (relevant for parallelization) and a `Payload` (used only for processing), where predicates are given as sets of tags (or pairs of tags, for `depends`). This allows simpler logic in the fork function whose input predicates are then `Tag -> Bool` and don't depend on the irrelevant payload. In our example, i(k) or r(k) would be tags (not payload) as they are relevant for parallelization.

2.3 Consistency conditions

Any parallel execution is guaranteed to preserve the sequential semantics, i.e. processing all input events *in order* using the `update` function, as long as the following *consistency conditions* are satisfied. The sufficiency of these conditions is shown in Theorem 2.4, which states that *consistency implies* *determinism up to output reordering*. This is a key step in the end-to-end proof of correctness in Section 3.5. Consistency can be thought of as analogous to the commutativity and associativity requirements for a MapReduce program to have deterministic output [22]: just as with MapReduce programs, the implementation does *not* assume the conditions are satisfied, but if not the semantics will be dependent on how the computation is parallelized.

Definition 2.3 (Consistency)**.** A program is *consistent* if the following equations always hold:

```
join(update(s1,e),s2) = update(join(s1,s2),e)        (C1)
join(fork(s,pred1,pred2)) = s                        (C2)
update(update(s,e1),e2)) = update(update(s,e2),e1))  (C3)
```

subject to the following additional qualifications. First, equation (C1) is over all joins `join: (State_j, State_k) -> State_i`, events `e: Event` such that `pred_i(e)` and `pred_j(e)`, and states `s1: State_j`, `s2: State_k`, where `update` denotes the update function on the appropriate type. Additionally the corresponding output on both sides must be the same: `out(s1, e) = out(join(s1, s2))`. Equation (C2) is over all fork functions `fork: (State_i,Pred(Event),Pred(Event)) -> (State_j, State_k)`, all joins `join: (State_j, State_k) -> State_i`, states `s: State_i`, and predicates `pred1` and `pred2`. Equation (C3) is over all state types `State_i`, states `s: State_i`, and pairs of *independent* events `indep(e1, e2)` such that `pred_i(e1)` and `pred_i(e2)`. As with (C1), we also require that the outputs on both sides agree:

```
out(s, e1) + out(update(s, e1), e2)
  = out(update(s, e2), e1) + out(s, e2).
```

Let us illustrate the consistency conditions for our running example (Figure 1). If `e` is an increment event, then condition (C1) captures the fact that counting can be done in parallel: it reduces to `(s1[k] + s2[k]) + 1 = (s1[k] + 1) + s2[k]`. Condition (C2) captures the fact that we preserve total count across states when forking: it reduces to `s[k] + 0 = s[k]`. Condition (C3) would not be valid for *general* events `e1`, `e2`, because a read-reset event does not commute with an increment of the same key (`s[k] + 1` $\neq$ `s[k]`), hence the restriction that `indep(e1, e2)`. Finally, one might think that a variant of (C1) should hold for `fork` in addition to `join`, but this turns out not to be the case: for example, starting from `s[k] = 100`, an increment followed by a fork might yield the pair of counts $(101, 0)$, while a fork followed by an increment might yield $(100, 1)$. It turns out however that commutativity only with joins, and not with forks, is enough to imply Theorem 2.4.

Theorem 2.4. *If P is consistent, then P is deterministic up to output reordering. That is, for all* $(\mathtt{u}, \mathtt{v}) \in [[P]]$*, the multiset of events in stream* `v` *is equal to the multiset of events in* `spec(u)` *where* `spec` *is the semantics of the sequential implementation.*

Proof. We show by induction on the semantics in Definition 2.2 that every wire diagram is equivalent (up to output reordering) to the sequential sequence of updates. The sequential inductive step (3) is direct by associativity of function composition on the left and right sequence of updates (no commutativity of updates is required). For the parallel inductive step (4), we replace the two parallel wires with sequential wires, then apply (C1) repeatedly on the last output to move it outside of the parallel wires, then finally apply (C2) to reduce the now trivial parallel wires to a single wire. ◻

3 Synchronization Plans

In this section we describe *synchronization plans*, which represent streaming program implementations, and our framework for generating them from the given DGS program in Section 2. Generation of an implementation can be conceptually split in two parts, the first ensuring correctness and the second affecting performance. First a program P induces a set of P-*valid*, i.e. correct with respect to it, synchronization plans. Choosing one of those plans is then an independent optimization problem that does not affect correctness and can be delegated to a separate optimization component (Section 3.3). Finally, the workers in synchronization plans need to process some incoming events in order while some can be processed out of order (depending on the dependence relation). We propose a selective reordering technique (Section 3.4) that can be used in tandem with heartbeats to address this ordering issue. We tie everything together by providing an end-to-end proof that the implementation is correct with respect to a consistent program P (and importantly, independent of the synchronization plan chosen as long as it is P-valid) in Section 3.5. Before describing the separate framework components, we first articulate the necessary underlying assumptions about input streams in Section 3.1.

3.1 Preliminaries

In our model the input is partitioned in some number of input streams that could be distributed, i.e. produced at different locations. We assume that the implementation has access to *some* ordering relation O on pairs of input events (also denoted $<_O$), and the order of events is increasing along each input stream. This is necessary for cases where the *user-written* program requires that events arriving in different streams are dependent, since it allows the implementation to have progress and process these dependent events in order. Concretely, in our setting O is implemented using *event timestamps*. Note that these timestamps do not need to correspond to real time, if this is not required by the application. In cases where real-time timestamps are required, this can be achieved with well-synchronized clocks, as has been done in other systems, e.g. Google Spanner [21].

Each event in each input stream is given by a quadruple $\langle tg, id, ts, v \rangle$, where tg is a *tag* used for parallelization, id is

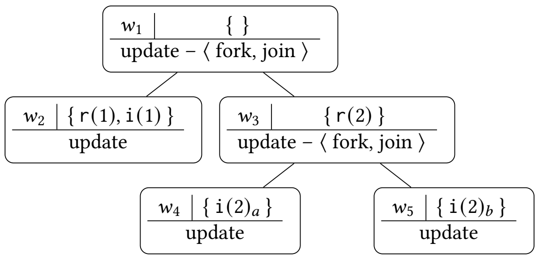


Figure 3. Example synchronization plan derived from the program in Figure 1 for two keys $k = 2$ and five input streams r(1), i(1), r(2), $\mathtt{i(2)}_a$, $\mathtt{i(2)}_b$. Implementation tags $\mathtt{i(2)}_a$, $\mathtt{i(2)}_b$ both correspond to i(2) events but are separate because they arrive in different input streams.

a unique identifier of the input stream, *ts* is a *timestamp*, and v is a *payload*. Of these, only the tag and payload are visible to the programming model in Section 2, and only the tag is used in predicates and in the dependence relation. Our implementation currently requires that the number of possible tags *tg* is finite (e.g. up to some maximum key) as well as the number of identifiers *id*.

For the rest of this section, we write events as $\langle \sigma, ts, v \rangle$ where the pair $\sigma = \langle tg, id \rangle$ is called the *implementation tag*. This is a useful distinction because at the implementation level, these are the two components that are used for parallelization. The relation `depends: (Tag, Tag) -> Bool` in the program straightforwardly lifts to predicates over tags and to implementation tags.

3.2 Synchronization plans

Synchronization plans are binary tree structures that encode (i) parallelism: each node of the tree represents a sequential thread of computation that processes input events; and (ii) synchronization: parents have to synchronize with their children to process an event. Synchronization plans are inspired by prior work on concurrency models including fork-join concurrency [25, 40] and CSP [30]. An example synchronization plan for the program in Figure 1 is shown in Figure 3. Each node has an id w_i, contains the set of implementation tags that it is responsible for, a state type (which is omitted here since there is only one state type `State`), and a triple of update, fork, join functions. Note that a node is responsible to process events from its set of implementation tags, but can potentially handle all the implementation tags of its children. The leaves of the tree can process events independently without blocking, while parent nodes can only process an input event if their children states are joined. Nodes without a ancestor-descendant relationship do not directly communicate, but instead learn about each other when their common ancestor joins and forks back the state.

Definition 3.1 (Synchronization Plans)**.** Given a program P, a synchronization plan is a pair $(\overline{w}, \mathrm{par})$, which includes a set of workers $\overline{w} = \{w_1, \ldots, w_N\}$, together with a parent relation $\mathrm{par} \subseteq \overline{w} \times \overline{w}$, the transitive closure of which is an ancestor relation denoted as $\mathrm{anc} \subseteq \overline{w} \times \overline{w}$. Workers have three components: (i) a state type w.state which references one of the state types of P, (ii) a set of implementation tags w.itags that the worker is responsible for, and (iii) an update w.update and possibly a fork-join pair w.fork and w.join if it has children.

We now define what it means for a synchronization plan to be P-*valid*. Intuitively, an P-valid plan is well-typed with respect to program P, and the workers that do not have an ancestor-descendant relationship should handle independent and disjoint implementation tags. P-validity is checked syntactically by our framework and is a necessary requirement to ensure that the generated implementation is correct (see Section 3.5).

Definition 3.2 (P-valid)**.** Formally, a P-valid plan has to satisfy the following syntactic properties: (V1) The state `State_i` $= w$.state of each worker w should be consistent with its update-fork-join triple and its implementation tags. The update must be defined on the node state type, i.e., w.update : `(State_i, Event) -> State_i`, `State_i` should be able to handle the tags corresponding to w.itags, and the fork-join pair should be defined for the state types of the node and its children. (V2) Each pair of nodes that do not have an ancestor-descendant relation, should handle pairwise independent and disjoint implementation tag sets, i.e., $\forall w, w' \notin \mathrm{anc}(w, w'), w.\mathrm{itags} \cap w'.\mathrm{itags} = \emptyset \wedge \mathrm{indep}(w.\mathrm{itags}, w'.\mathrm{itags})$.

As an example, the synchronization plan shown in Figure 3 satisfies both properties; there is only one state type that handles all tags and implementation tag sets are disjoint for ancestor-descendants. The second property (V2) represents the main idea behind our execution model; independent events can be processed by different workers without communication. Intuitively, in the example in Figure 3, by assigning the responsibility for handling tag r(2) to node w_3, its children can independently process tags $\mathtt{i(2)}_a$, $\mathtt{i(2)}_b$ that are dependent on r(2).

3.3 Optimization problem

As described in the previous section, a set of P-valid synchronization plans can be derived from a DGS program P. This decouples the optimization problem of finding a well-performing implementation, allowing it to be addressed by an independent optimizer, which takes as input a description of the available computer nodes and the input streams. This design means that different optimizers could be implemented for different performance metrics (e.g. throughput, latency, network load, energy consumption). The design space for optimizers is vast and thoroughly exploring it is outside of the scope of this work. For evaluation purposes, we have implemented a few simple optimizers, one of which tries to minimize communication between workers by placing them close to their inputs. Intuitively, it searches for divisions of the event tags into two sets such that those sets are "maximally" independent, using those sets of tags for the initial fork, and then recursing on each independent subset. Its design is described in more detail in the extended version of the paper [36].

3.4 Implementation

Each node of the synchronization plan can be separated into two components: an event-processing component responsible for executing `update`, `fork`, and `join` calls; and a mailbox component responsible for enforcing ordering requirements and synchronization.

Event processing. The worker processes execute the functions (`update`, `fork`, and `join`) associated with the tree node. Whenever a worker is handed a message by its mailbox, it first checks if it has any active children, and if so, it sends them a join request and waits until it receives their responses. After receiving these responses, it executes the join function to combine their states, executes the update function on the received event, and then executes the fork function on the new state, and sends the resulting states to its children. In contrast, a leaf worker just executes the update function on the received event.

Event reordering. The mailbox of each worker ensures that it processes incoming dependent events in the correct order by implementing the following selective reordering procedure. Each mailbox contains an event buffer and a timer for each implementation tag. The buffer holds the events of a specific tag in increasing order of timestamps and the timer indicates the latest timestamp that has been encountered for each tag. When a mailbox receives an event $\langle \sigma, ts, v \rangle$ (or a join request), it follows the procedure described below. It first inserts it in the corresponding buffer and updates the timer for σ to the new timestamp ts. It then initiates a cascading process of releasing events with tags σ' that depend on σ. During that process all dependent tags σ' are added to a dependent tag workset, and the buffer of each tag in the workset is checked for potential events to release. An event $e = \langle \sigma, ts, v \rangle$ can be released to the worker process if two conditions hold. The timers of its dependent tags are higher than the timestamp ts of the event (which means that the mailbox has already seen all dependent events up to ts, making it safe to release e), and the earliest event in each buffer that σ depends on should have a timestamp $ts' > ts$ (so that events are processed in order). Whenever an event with tag σ is released, all its dependent tags are added to the workset and this process recurses until the tag workset is empty.

Heartbeats. As discussed in Section 3.1, a dependence between two implementation tags σ_1 and σ_2 requires the implementation to process any event $\langle \sigma_1, t_i, v_i \rangle$ after processing all events $\langle \sigma_2, t_j, v_j \rangle$ with $t_j \leq t_i$. However, with the current assumptions on the input streams, a mailbox has to wait until it receives the earliest event $\langle \sigma_2, t_j, v_j \rangle$ with $t_j > t_i$, which could arbitrarily delay event processing. We address this issue by periodically generating *heartbeat* events at each producer, which are system events that represent the absence of events on a stream. Heartbeats are interleaved together with standard events of input streams. When a heartbeat event $\langle \sigma, t \rangle$ first enters the system, it is broadcast to all the worker processes that are descendants of the worker that is responsible for tag σ. Each mailbox that receives the heartbeat updates its timers and clears its buffers as if it has received an event of $\langle \sigma, t, v \rangle$ without adding the heartbeat to the buffer to be released to the worker process. Similar mechanisms are often used in other stream processing systems under various names, e.g. heartbeats [33], punctuation [61], watermarks [16], or pulses [55].

In our experience, heartbeat rates are successful in improving the latency of the system unless they are configured to be very large or very low values. For a wide range of heartbeat values(~10-1000 per synchronization event), the performance of the system is stable and exhibits minor variance (see more details in the extended version [36]).

3.5 Proof of correctness

We show that *any* implementation produced by the end-to-end framework is correct according to the semantics of the programming model (Theorem 3.5). First, Definition 3.3 formalizes the assumptions about the input streams outlined in Section 3.1, and Definition 3.4 defines what it means for an implementation to be correct with respect to a sequential specification. Our definition is inspired by the classical definitions of distributed correctness based on observational trace semantics (e.g., [44]). However, we focus on how to interpret the independent input streams as a sequential input, in order to model possibly synchronizing and order-dependent stream processing computations. The proof of Theorem 3.5 can be found in the extended version of the paper [36].

Definition 3.3. A *valid input instance* consists of k input streams (finite sequences) `u_1`, `u_2`, ..., `u_k` of type `List(Event | Heartbeat)`, and an order relation O on input events and heartbeats, with the following properties. (1) *Monotonicity:* for all i, `u_i` is in strictly increasing order according to $<_O$. (2) *Progress:* for all i, for each input event (non-heartbeat) `x` in `u_i`, for every other stream j there exists an event or heartbeat `y` in `u_j` such that `x` $<_O$ `y`.

Given a DGS program P, the *sequential specification* is a function `spec: List(Event) -> List(Out)`, derived from the sequential implementation by applying only `update` and no `fork` and `join` calls. The output specified by `spec` is produced incrementally (or *monotonically*): if `u` is a prefix of `u'`, then `spec(u)` is a subset of `spec(u')`. Define the *sort* function $\texttt{sort}_O$: `List(List(Event | Heartbeat)) -> List(Event)` which takes k sorted input event streams and sorts them into one sequential stream, according to the total order relation O, and drops heartbeat events.

Definition 3.4. A distributed implementation is *correct* with respect to a given sequential specification `spec: List(Event) -> List(Out)`, if for every valid input instance O, `u_1`, ..., `u_k`, the set of outputs produced by the implementation is equal to $\texttt{set(spec(sort}_O\texttt{(u\_1, ..., u\_k)))}$.

Theorem 3.5 (implementation correctness)**.** *Any implementation produced by our framework is correct according to Definition 3.4.*

4 Experimental Evaluation

In this section we conduct a set of experiments to investigate tradeoffs between data parallelism and *platform independence* in stream processing APIs. That is, we want to distinguish between parallelism that is achieved automatically and parallelism that is achieved manually at the cost of portability when the details of the underlying platform change. To frame this discussion, we identify a set of platform independence principles (PIP) with which to evaluate this tradeoff:

PIP1: ***parallelism independence.*** Is the program developed without regard to the number of parallel instances, or does the program use the number of parallel instances in a nontrivial way?

PIP2: ***partition independence.*** Is the program developed without regard to the correspondence between input data and parallel instances, or does it require knowledge of how input streams are partitioned to be correct?

PIP3: ***API compliance.*** Does the program violate any assumptions made by the stream processing API?

Having identified these principles, the following questions guide our evaluation:

Q1 For computations requiring synchronization, what are the throughput limits of automatic parallelism exposed by existing stream processing APIs?

Q2 Can *manual* parallel implementations, i.e., implementations that may sacrifice **(PIP1–3)** above, that emulate synchronization plans achieve *absolute* throughput improvements in existing stream processing systems?

Q3 What is the throughput scalability of the synchronization plans that are generated automatically by our framework?

Q4 In summary, for each method of achieving data parallelism with synchronization, what platform independence tradeoffs are made?

In order to study these questions, we design three applications with synchronization requirements in Section 4.1. Our investigation compares three systems at different points in the implementation space with varying APIs and performance characteristics. First, Apache Flink [5, 16] represents a well-engineered mainstream streaming system with an expressive API. Second, the Rust implementation of Timely Dataflow [47, 50] (Timely) represents a system with support for iterative computation. Third, Flumina is a prototype implementation of our end-to-end framework that supports the communication patterns observed in arbitrary synchronization plans, some of which are not supported by the execution models of Flink and Timely.

Experimental setup. We conduct all the experiments in this section in a distributed execution environment consisting of AWS EC2 instances. To account for the fact that AWS instances can introduce variability in results, we chose m6g medium (1 core @2.5GHz, 4 GB) instances, which do not use burst performance like free-tier instances. We use instances in the same region (us-east-2) and we increase the number of instances for the scalability experiments. Communication between nodes is managed by each respective system (the system runtime for Flink and Timely, and Erlang for Flumina).

We configure Flink to be in true streaming mode by disabling batching (setting buffer-timeout to 0), checkpointing, and dynamic adaptation. For Timely, it is inherent to the computational model that events are batched by logical timestamp, and the system is not designed for event-by-event streaming, so our data generators follow this paradigm. This results in significantly higher throughputs for Timely, but note that these throughputs are *not* comparable with Flink and Flumina due to the batching differences. Because the purpose of our evaluation is not to compare absolute performance differences due to batching *across systems*, we focus on relative speedups *on the same system* and how they relate to platform independence **(PIP1–3)**.

4.1 Applications requiring synchronization

We consider three applications that require different forms of synchronization. All three of the applications do not perform CPU-heavy computation for each event so as to expose communication and underlying system costs in the measurements. The conclusions that we draw remain valid, since a computation-heavy application that would exhibit similar synchronization requirements would scale even better with the addition of more processing nodes. The input for all three applications is synthetically generated.

Event-based windowing. An *event-based window* is a window whose start and end is defined by the occurrence of certain events. This results in a simple synchronization pattern where parallel nodes must synchronize at the end of each window. For this application, we generate an input consisting of several streams of integer *values* and a single (separate) stream of *barriers*. The task is to produce an aggregate of the values between every two consecutive barriers, where *between* is defined based on event timestamps. We take the aggregation to be the sum of the values. The computation is parallelizable if there are sufficiently more value events than barrier events. In the input to our experiments, there are 10K events in each value stream between two barriers.

Page-view join. The second application is an example of a streaming join. The input contains 2 types of events: *page-view* events that represent visits of users to websites, and *update-page-info* events that update the metadata of a particular website and also output the old metadata when processed by the program. All of these events contain a unique identifier identifying the website and the goal is to join page-view events with the latest metadata of the visited page to augment them for a later analysis. An additional assumption is that the input is not uniformly distributed among websites, but a small number of them receive most of the page-views. To simulate this behavior in the inputs used in our experiments, all views are distributed between two pages.

Fraud detection. Finally, the third application is a version of the ML fraud detection application mentioned in the introduction, where the synchronization requirements are the same but the computation is simplified. The input contains *transaction* events and *rule* events both of which are integer values. On receiving a rule, the program outputs an aggregate of the transactions since the last rule and a transaction is considered fraudulent if it is equivalent modulo 1000 to the sum of the previous aggregate (simulating model retraining) and the last rule event. As in event-based windowing, we generate 10K transaction events between every two rule events.

4.2 Implementations in Flink and Timely

In this section we investigate how the Flink and Timely APIs can produce scalable parallel implementations for the aforementioned applications. We iterated on different implementations resulting in a best-effort attempt to achieve good scalability. These implementations are summarized below, and the source code is presented in the extended version of the paper [36]. For each of the implementations, we then ran an experiment where we increased the number of distributed nodes and measured the maximum throughput of the resulting implementation (by increasing the input rate until throughput stabilizes or the system crashes). The results are shown in Figure 4.

Event-based windowing. Flink's API supports a `broadcast` construct that can send barrier events to all parallel instances of the implementation, therefore being able to scale with an increasing parallel input load. Note that Flink does not guarantee that the broadcast messages are synchronized with the other stream events, and therefore the user-written code

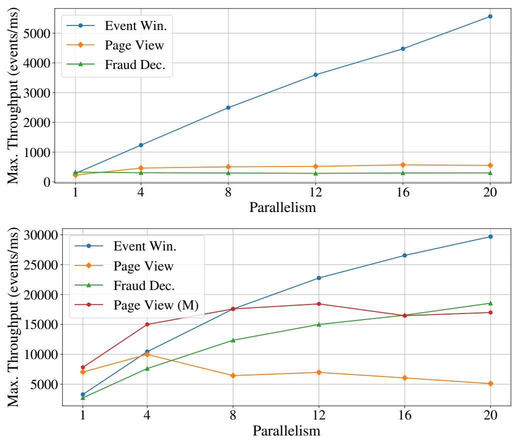


Figure 4. Flink (top) and Timely (bottom) maximum throughput increase with increasing parallel nodes for the three applications that require synchronization. For the page-view example in Timely, two implementations are shown: Page View uses automatic parallelism while Page View (M) is the manual parallel implementation in Figure 5.

```
updates.broadcast().filter(move |x| {
  x.name == page_partition_fun(NUM_PAGES, worker_index)
});
```

Figure 5. Snippet from the Timely manual (M) implementation of the page-view join example, satisfying **PIP1** and **PIP3** but not **PIP2**.

has to ensure that it processes them in order. By transforming these barriers to Flink watermarks that indicate window boundaries, we can then aggregate values of each window in parallel, and finally merge all windows to get the global aggregate. Similarly, Timely includes a `broadcast` operator on streams, which sends all barrier events to all parallel instances; then the `reclock` operator is used to match values with corresponding barriers and aggregate them. Both the Flink and Timely implementations scale because the values are much more frequent than barriers, i.e., a barrier every 10K events, and are processed in a distributed manner.

Page-view join. The input of this application allows for data parallelism across keys, in this case websites, but also for the same key since some keys receive most of the events. First, for both Flink and Timely, we implemented this application using a standard keyed join, ensuring that the resulting implementation will be parallel with respect to keys. As shown by the scalability evaluation, this does not scale to beyond 4 nodes in the case that there are a small number of keys receiving most or all of the events.

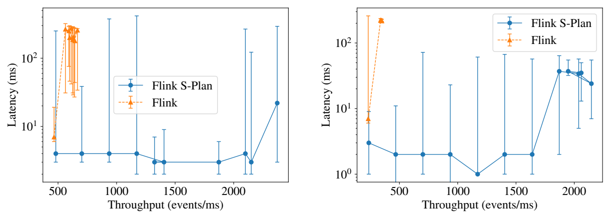
We wanted to investigate whether it was possible to go beyond the automatic implementations and scale throughput for events *of the same key*. To study this, we provide a "manual" (M) implementation in Timely (Figure 5 and Figure 4, bottom). Here, we broadcast *update-page-info* events, then filter to only those relevant to each node, i.e. corresponding to *page-views* that it is responsible for processing. A similar implementation would be possible in Flink. Unfortunately, our implementation sacrifices **PIP2**, i.e., the assignment of events to parallel instances becomes part of the application logic—there are explicit references to the physical partitioning of input streams (`page_partition_fun`) and the the worker that processes each stream (`worker_index`). Additionally, the implementation broadcasts *all* update events to *all* sharded nodes (not just the ones that are responsible for them), introducing a linear synchronization overhead with the increase of the number of nodes. An alternative choice would have been to not only broadcast events, but also keep state for *every* page at every sharded replica: this would satisfy **PIP2** because nodes no longer need to be aware of which events they process, but it does not avoid the broadcasting issue and thus we would expect performance overheads. Overall, we observe inability to automatically scale this application without sacrificing platform independence in both Flink and Timely.

Fraud detection. The standard dataflow streaming API cannot support cross-instance synchronization, and therefore we can only develop a sequential implementation of this application using Flink's API. Timely offers a more expressive API with *iterative* computation, and this allows for an automatically scaling implementation: after aggregating local state to globally updated the learning model, we have a cyclic loop which then sends the state back to all the nodes to process further events. The results show that this implementation scales almost as well as the event-based window. This effectively demonstrates the value of iterative computation in machine learning and complex stateful workflows.

Take-away (Q1): The streaming APIs of Flink and Timely cannot automatically produce implementations that scale throughput for all applications that have synchronization requirements without sacrificing platform independence.

4.3 Manual synchronization

To address Q2, we next investigate whether synchronization, implemented manually and possibly sacrificing **PIP1–3**, can offer concrete throughput speedups. We focus this implementation in Flink, and consider the two applications that Flink cannot produce parallel implementations for, namely *page-view join* and *fraud detection*. We write a DGS program for these applications and we use our generation framework to produce a synchronization plan for a specific parallelism level (12 nodes). We then manually implement the communication pattern for these synchronization plans in Flink,



(a) Page-view join. **(b)** Fraud detection.

Figure 6. Throughput (x-axis) and 10th, 50th, 90th percentile latencies on the y-axis for increasing input rates (from left to right) and 12 parallel nodes. Flink (orange) is the parallel implementation produced automatically, and Flink S-Plan (blue) is the synchronization plan implementation.

```
public Integer joinChild(
    final int subtaskId,
    final Integer state
) {
    final int parentId = subtaskId / pageViewParallelism;
    final int childId = subtaskId % pageViewParallelism;
    joinSemaphores.get(parentId).get(childId).release();
    forkSemaphores.get(parentId).get(childId)
        .acquireUninterruptibly();
    return zipCode.get(parentId);
}
```

Figure 7. Snippet from the implementation of the manual synchronization `join` in Flink. This implementation does not satisfy **PIP1–3**.

and we measure their throughput and latency compared to the parallel implementations that the systems produced in Section 4.2. The results for both applications are shown in Figure 6. Flink does not achieve adequate parallelism and therefore cannot handle the increasing input rate (low throughput and high latency).

Page-view join. The synchronization plan that we implement for this application is a forest containing a tree for each key (website) and each of these trees has leaves that process page-view events. Each time an update event needs to be processed for a specific key, the responsible tree is joined, processes the event, and then forks the new state back.

Fraud detection. The synchronization plan that we implement for this application is a tree that processes rule events at its root and transactions at all of its leaves. The tree is joined in order to process rules and is then forked back to keep processing transactions.

Implementation in Flink. In order to implement the synchronization plans in Flink we need to introduce communication across parallel instances. We accomplish this by using a centralized service that can be accessed by the instances using Java RMI. Synchronization between a parent and its

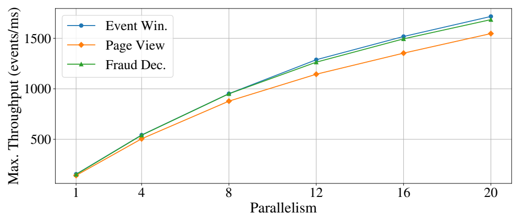


Figure 8. Flumina (DGS) maximum throughput increase with increasing parallel nodes for the three applications.

children happens using two sets of semaphores, J and F. A child releases its J semaphore and acquires its F semaphore when it is ready to join, and a parent acquires its children's J semaphores, performs the event processing, and then releases their F semaphores (Figure 7). This implementation of manual synchronization sacrifices all three platform independence principles **PIP1–3**. For **PIP1** and **PIP2**, it refers explicitly to the number of parallel instances and the partitioning (`pageViewParallelism`, `subtaskId`, etc.). For **PIP3**, it is not API-compliant because it uses an external service (semaphores) to implement synchronization, whereas Flink's documentation requires that operators lack side effects. This requirement is imposed because, among other considerations, the use of semaphores might cause the program to fail in cases where work is interrupted and/or repeated after a node failure. The full code for the Flink implementation can be found in the extended version of the paper [36].

Take-away (Q2): Synchronization plans achieve higher throughputs (4-8x for 12 parallel nodes) over the automatic parallel implementations produced by Flink's API.

4.4 Implementation in Flumina

To answer Q3, we implement Flumina, a prototype of our end-to-end framework that can automatically achieve parallelism given a DGS program. Flumina receives a DGS program written in Erlang, uses the generation framework that was described in Section 3 to generate a correct and efficient synchronization plan, and then implements the plan according to the description in Section 3.4. We implemented all three applications (source code in extended version [36]) in Flumina, and measured maximum throughput increase with the addition of parallel nodes (Figure 8).

Event-based windowing and fraud detection. The DGS program for event-based windowing contains: (i) a sequential update function that adds incoming value events to the state, and outputs the value of the state when processing a barrier event, (ii) a dependence relation that indicates that all events depend on barrier events, and (iii) a `fork` that splits the current state in half, together with a `join` that adds two states to aggregate them. The DGS program for fraud detection is the same with the addition that the `fork` also duplicates the sum of the previous transaction and last rule modulo 1000.

Page-view join. In addition to the sequential update function, the program indicates that events referring to different keys are independent, and that page-view events of the same key are also independent. The `fork` and `join` are very similar to the ones in Figure 1 and just separate the state with respect to the keys.

Take-away (Q3): Across all three examples, Flumina produces parallel implementations that scale throughput without sacrificing platform independence.

4.5 Summary: development tradeoffs

Finally, regarding Q4, Table 1 shows all the tradeoffs that need to be made for each of the programs in this section together with the throughput increase for 12 nodes. Note that throughput scaling comparison is only relevant for the same system (each of which is denoted with a different color) and not across systems due to differing sequential baselines. Of all the implementations in Section 4.2, the Timely manual page-view example sacrifices **PIP2**. The manually synchronizing Flink implementations in Section 4.3 show good throughput scaling at the cost of **PIP1–3**.

Take-away (Q4): Of the three APIs studied, only DGS can achieve scalable implementations across all examples without sacrificing either parallelism independence, partition independence, or API compliance.

5 Related Work

Dataflow stream processing systems. Applications over streaming data can be implemented using high-performance, fault tolerant stream processing systems, such as Flink [16], Trill [17], IBM Streams [29], Spark Streaming [63], Storm [6], Samza [51], Heron [37], and MillWheel [3]. The need for synchronization in these systems has resulted in a number of extensions to their APIs, but they fall short of a general solution. Naiad [50] proposes *timely dataflow* in order to support iterative computation, which enables some synchronization but falls short of automatically scaling without high-level design sacrifices, as shown in our evaluation. S-Store and TSpoon [1, 48] extend stream processing systems with online transaction processing (OLTP), which includes some forms of synchronization, e.g. locking-based concurrency control. Concurrent with our work, Nova [64] also identifies the need for synchronization in stream processing systems, and proposes to address it through a shared state abstraction.

Actor-based databases. As data processing applications are becoming more complex, evolving from data analytics to general event-driven applications, some stream processing

	Event window			Page-view join					Fraud detection			
Development tradeoff	F	TD	DGS	F	FM	TD	TDM	DGS	F	FM	TD	DGS
(PIP1) Parallelism independence	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓
(PIP2) Partition independence	✓	✓	✓	✓	✗	✓	✗	✓	✓	✗	✓	✓
(PIP3) API compliance	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓
Scaling	10x	8x	8x	2x	9x	1x	2x	8x	1x	9x	6x	8x

Table 1. Development tradeoffs for each program, together with throughput scaling for 12 nodes, in Flink (F), Flink with manual synchronization (FM), Timely (TD), Timely with manual partitioning (TDM), and our system (DGS).

and database systems are moving from dataflow programming to more general actor models [12, 13, 15, 56, 62]. For example, Flink has recently released Stateful Functions, an actor-based programming model running on top of Flink [2, 24]. Actor models can encode arbitrary synchronization patterns, but the patterns still need to be implemented manually as message-passing protocols. DGS and synchronization plans can be built on top of the actor abstraction, and in fact our own implementation relies on actors as provided by Erlang [10].

Programming with synchronization. In the broader context of distributed and parallel programming, synchronization is a significant source of overhead for developers, and a good deal of existing work can be viewed as addressing this problem. Our model draws inspiration from fork-join based concurrent programming [25, 40], bringing some of the expressiveness in those models to the streaming setting, where parallelism is much less flexible but essential for performance, but also extending them, since in our setting the system (and not the user) decides when to fork and join by choosing a synchronization plan. A particularly relevant example is Concurrent Revisions [14], which is a programming model that guarantees determinism in the presence of concurrent updates by allowing programmers to specify isolation types that are processed in parallel and then merged at join points. The difference of our work is that it targets a more restricted domain providing automation, not requiring programmers to manually specify the execution synchronization points. Another related domain is monotonic lattice-based programming models, including Conflict-Free Replicated Data Types [57], Bloom^L [20], and LVars [38, 39], which are designed for coordination-free distributed programming. These models guarantee strong eventual consistency, i.e., eventually all replicas will have the same state, but, in contrast to our model, CRDTs and Bloom^L do not allow synchronization between different workers. LVars, which focuses on determinism for concurrent updates on shared variables, extends lattice-based models with a freeze operation that enforces a synchronization point, inducing partial order executions that are similar to the ones in our model. Some similarities with our work can be found in the domain of consistency for replicated data stores. Some examples include RedBlue consistency [43], MixT [49], Quelea [58], CISE [28], Carol [42], all of which support a mix of consistency guarantees on different operations, inducing a partial order of data store operations.

Correctness in stream processing. Finally, in prior work, researchers have pointed out that parallelization in stream processing systems is not semantics-preserving, and have proposed methods to restrict parallelization so that it preserves the semantics [46, 55]. In particular, dependence relations have been previously used in this context as to specify and ensure correct parallelism [35, 46] and as a type system for synchronization [4]. However, these works do not propose a general programming model or generation of a parallel implementation.

6 Future Work

One problem that is not yet adequately explored in our framework is optimization: given a DGS program, how to select a valid synchronization plan which is most efficient according to a desired cost metric. Traditional optimization algorithms for stream processing systems cannot be directly applied to the complex tree structure of synchronization plans. There are also possibilities for dynamic optimization, in which the synchronization plan is modified online in response to profiling data from the system. Besides optimization, the implementation of synchronization plans needs to address a plethora of systems related issues, such as (i) the efficient management and communication of forked state in a distributed environment, (ii) execution guarantees in the presence of faults, and (iii) supporting performance optimizations such as batching and backpressure.

Acknowledgments

We want to thank Anton Xue, Badrish Chandramouli, Nikos Vasilakis, and many others, including attendees at the Heidelberg Laureate Forum 2019 and PRECISE Industry Day 2019, for their comments on earlier versions of this work over the years. We also want to thank the reviewers of this and all past versions of this paper for their valuable feedback leading to substantial improvements. This research was supported in part by NSF award 1763514.

References

[1] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. 2020. TSpoon: Transactions on a stream processor. *J. Parallel and Distrib. Comput.* 140 (2020), 65–79.

[2] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. 2019. Stateful Functions as a Service in Action. *Proc. VLDB Endow.* 12, 12 (2019), 1890–1893. https://doi.org/10.14778/3352063.3352092

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[4] Rajeev Alur, Phillip Hilliard, Zachary G Ives, Konstantinos Kallas, Konstantinos Mamouras, Filip Niksic, Caleb Stanford, Val Tannen, and Anton Xue. 2021. Synchronization Schemas. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems.* 1–18.

[5] Apache. 2019. Apache Flink. https://flink.apache.org/. [Online; accessed March 31, 2019].

[6] Apache. 2019. Apache Storm. http://storm.apache.org/. [Online; accessed March 31, 2019].

[7] Apache. 2021. Apache Beam. https://beam.apache.org/. [Online; accessed April 16, 2021].

[8] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[9] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data.* 601–613.

[10] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. 1993. Concurrent Programming in Erlang.

[11] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. 2019. One SQL to Rule Them All-an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. In *Proceedings of the 2019 International Conference on Management of Data.* 1757–1772.

[12] Philip A. Bernstein, Mohammad Dashti, Tim Kiefer, and David Maier. 2017. Indexing in an Actor-Oriented Database. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p29-bernstein-cidr17.pdf

[13] Philip A Bernstein, Todd Porter, Rahul Potharaju, Alejandro Z Tomsic, Shivaram Venkataraman, and Wentao Wu. 2019. Serverless Event-Stream Processing over Virtual Actors.. In *CIDR.*

[14] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. 2010. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications.* 691–707.

[15] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond Analytics: The Evolution of Stream Processing Systems. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2651–2658. https://doi.org/10.1145/3318464.3383131

[16] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[17] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.

[18] Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2010. High-performance dynamic pattern matching over disordered streams. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 220–231.

[19] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce online.. In *Nsdi*, Vol. 10. 20.

[20] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and Lattices for Distributed Programming. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12).* Association for Computing Machinery, New York, NY, USA, Article 1, 14 pages. https://doi.org/10.1145/2391229.2391230

[21] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[22] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[23] Flink 2020. FLIP-8: Rescalable Non-Partitioned State - Apache Flink - Apache Software Foundation. https://cwiki.apache.org/confluence/display/FLINK/FLIP-8%3A+Rescalable+Non-Partitioned+State. https://cwiki.apache.org/confluence/display/FLINK/FLIP-8%3A+Rescalable+Non-Partitioned+State

[24] Flink 2020. Stateful Functions 2.0 - An Event-driven Database on Apache Flink. https://flink.apache.org/news/2020/04/07/release-statefun-2.0.0.html.

[25] Matteo Frigo, Charles E Leiserson, and Keith H Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation.* 212–223.

[26] 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* (Vancouver, Canada) *(SIGMOD '08).* ACM, New York, NY, USA, 1123–1134. https://doi.org/10.1145/1376616.1376729

[27] Kahn Gilles. 1974. The semantics of a simple language for parallel programming. *Information processing* 74 (1974), 471–475.

[28] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16).* Association for Computing Machinery, New York, NY, USA, 371–384. https://doi.org/10.1145/2837614.2837625

[29] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K. L. Wu. 2013. IBM Streams Processing Language: Analyzing Big Data in motion. *IBM Journal of Research and Development* 57, 3/4 (2013), 7:1–7:11. https://doi.org/10.1147/JRD.2013.2243535

[30] Charles Antony Richard Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (1978), 666–677.

[31] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. 2018. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC).* IEEE, 115–131.

[32] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. 2008. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1379–1390.

[33] Theodore Johnson, Shanmugavelayutham Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. 2005. A heartbeat mechanism and its application in gigascope. In *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 1079–1088.

[34] Kafka 2020. KTable state stores and improved semantics - Apache Kafka - Apache Software Foundation. https://cwiki.apache.org/confluence/display/KAFKA/KIP-114%3A+KTable+state+stores+and+improved+semantics.

[35] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. 2020. DiffStream: differential output testing for stream processing programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29.

[36] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. 2021. Stream Processing with Dependency-Guided Synchronization (Extended Version). arXiv:2104.04512 [cs.PL]

[37] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. ACM, New York, NY, USA, 239–250. https://doi.org/10.1145/2723372.2742788

[38] Lindsey Kuper and Ryan R. Newton. 2013. LVars: Lattice-Based Data Structures for Deterministic Parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing* (Boston, Massachusetts, USA) *(FHPC '13)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/2502323.2502326

[39] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. 2014. Freeze after Writing: Quasi-Deterministic Parallel Programming with LVars. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 257–270. https://doi.org/10.1145/2535838.2535842

[40] Doug Lea. 2000. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*. 36–43.

[41] Edward A Lee and David G Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[42] Nicholas V Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Černỳ. 2019. Sequential programming for replicated data stores. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–28.

[43] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 265–278.

[44] Nancy A Lynch. 1996. *Distributed algorithms*. Elsevier.

[45] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G Ives, and Sanjeev Khanna. 2017. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 693–708.

[46] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G Ives, and Val Tannen. 2019. Data-trace types for distributed stream processing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 670–685.

[47] Frank McSherry. 2020. Timely Dataflow (Rust). https://github.com/TimelyDataflow/timely-dataflow/. [Online; accessed September 30, 2020].

[48] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, et al. 2015. S-Store: streaming meets transaction processing. *Proceedings of the VLDB Endowment* 8, 13 (2015), 2134–2145.

[49] M. Milano and Andrew C Myers. 2018. MixT: A language for mixing consistency in geodistributed transactions. *ACM SIGPLAN Notices* 53, 4 (2018), 226–241.

[50] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[51] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (Aug. 2017), 1634–1645. https://doi.org/10.14778/3137765.3137770

[52] Matthew Eric Otey, Amol Ghoting, and Srinivasan Parthasarathy. 2006. Fast distributed outlier detection in mixed-attribute data sets. *Data mining and knowledge discovery* 12, 2-3 (2006), 203–228.

[53] Milinda Pathirage, Julian Hyde, Yi Pan, and Beth Plale. 2016. SamzaSQL: Scalable fast data management with streaming SQL. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1627–1636.

[54] Samza 2020. Side Inputs for Local Stores - Apache Samza - Apache Software Foundation. https://cwiki.apache.org/confluence/display/SAMZA/SEP-27%3A+Side+Inputs+for+Local+Stores.

[55] S. Schneider, M. Hirzel, B. Gedik, and K. Wu. 2015. Safe Data Parallelism for General Streaming. *IEEE Trans. Comput.* 64, 2 (Feb 2015), 504–517. https://doi.org/10.1109/TC.2013.221

[56] Vivek Shah and Marcos Antonio Vaz Salles. 2018. Reactors: A Case for Predictable, Virtualized Actor Database Systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 259–274. https://doi.org/10.1145/3183713.3183752

[57] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

[58] Krishnamoorthy C Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative programming over eventually consistent data stores. *ACM SIGPLAN Notices* 50, 6 (2015), 413–424.

[59] Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. 2010. A universal calculus for stream processing languages. In *European Symposium on Programming*. Springer, 507–528.

[60] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*. Springer, 179–196.

[61] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003), 555–568.

[62] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. 2021. Move Fast and Meet Deadlines: Fine-grained Real-time Stream Processing with Cameo. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 389–405.

[63] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

[64] Yunjian Zhao, Zhi Liu, Yidi Wu, Guanxian Jiang, James Cheng, Kunlong Liu, and Xiao Yan. 2021. Timestamped State Sharing for Stream Analytics. *IEEE Transactions on Parallel and Distributed Systems* 32, 11 (2021), 2691–2704.