Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems

Luo Mai[1], Kai Zeng[2], Rahul Potharaju[2], Le Xu[3], Steve Suh[2], Shivaram Venkataraman[2], Paolo Costa[1,2], Terry Kim[2], Saravanan Muthukrishnan[2], Vamsi Kuppa[2], Sudheer Dhulipalla[2], Sriram Rao[2]

[1]Imperial College London, [2]Microsoft, [3]UIUC

[1]luo.mai11@imperial.ac.uk, [2]{kaizeng, rapoth, stsuh, shivaram.venkataraman, pcosta, terryk, sarmut, vamsik, sudheerd, sriramra}@microsoft.com, [3]lexu1@illinois.edu

ABSTRACT

Stream-processing workloads and modern shared cluster environments exhibit high variability and unpredictability. Combined with the large parameter space and the diverse set of user SLOs, this makes modern streaming systems very challenging to statically configure and tune. To address these issues, in this paper we investigate a novel control-plane design, Chi, which supports continuous monitoring and feedback, and enables dynamic re-configuration. Chi leverages the key insight of embedding control-plane messages in the data-plane channels to achieve a low-latency and flexible control plane for stream-processing systems.

Chi introduces a new reactive programming model and design mechanisms to asynchronously execute control policies, thus avoiding global synchronization. We show how this allows us to easily implement a wide spectrum of control policies targeting different use cases observed in production. Large-scale experiments using production workloads from a popular cloud provider demonstrate the flexibility and efficiency of our approach.

PVLDB Reference Format:
Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, Sriram Rao. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. *PVLDB*, 11 (10): xxxx-yyyy, 2018.
DOI: https://doi.org/10.14778/3231751.3231765

1. INTRODUCTION

Large-scale Internet-service providers such as Amazon, Facebook, Google, and Microsoft generate tens of millions of data events per second [5]. To handle such high throughput, they have traditionally resorted to offline batch systems [22, 13, 4]. More recently, however, there has been an increasing trend towards using streaming systems [3, 10, 28, 32] to ensure timely processing and avoid the delays typically incurred by offline batch systems.

Proceedings of the VLDB Endowment, Vol. 11, No. 10

DOI: https://doi.org/10.14778/3231751.3231765

Fully achieving the benefits promised by these online systems, however, is particularly challenging. First, streaming workloads exhibit high *temporal* and *spatial* variability, up to an order of magnitude compared to the average load [28, 32]. Second, large shared clusters exhibit high hardware heterogeneity and unpredictable concurrent usage. Third, modern streaming systems expose a large parameter space, which makes tuning hard, even for the most experienced engineers. Further, different users and jobs have a diverse set of Service Level Objectives (SLOs), leading to divergent configuration settings. Addressing these issues requires introducing continuous monitoring and feedback as well as dynamic re-configuration into all aspects of streaming systems, ranging from query planning and resource allocation/scheduling to parameter tuning. We call the system layer in charge of such control mechanisms the *control plane*, to distinguish it from the *data plane* (the layer in charge of data processing).

Through our interaction with product teams and cloud operators, we identified the following requirements that a control plane should satisfy. First, it should be possible to define new custom control operations, tailored to different scenarios [25]. Second, this should be achieved with only minimal effort by the developers and through a simple and intuitive API to minimize the chance of bugs, which are particularly challenging to detect in a distributed environment. Finally, the control overhead should be kept to a minimum, even in the presence of high event throughput and large computation graphs. This is particularly critical in today's landscape with different cloud providers competing hard to offer the best SLOs to their customers. Ideally the control plane should match the data-plane SLO (usually in the order of seconds or less).

Unfortunately, to the best of our knowledge, none of the existing streaming systems can satisfy all these requirements. Heron [28] and Flink [10] have a monolithic control plane that supports only a limited set of predefined control policies (e.g., dynamic scaling and back pressure), and lacks a clean API, which makes it hard for users to define custom policies. Spark Streaming [42], adopts a Bulk-Synchronous Parallel (BSP) model [39] in which a set of events is buffered and processed as a batch. While this allows the system to modify a dataflow between batches, it has limited flexibility due to the hard batch boundaries and incurs high overhead due to the synchronization and scheduling operations required.

To overcome the shortcomings of today's systems and meet the aforementioned requirements, we explore a novel control-plane design for stream processing systems. Inspired by the idea of punctuations [38] being used for data operators, we propose embedding

control-plane messages into the data stream. By leveraging the existing data pipeline, control messages can be streamed at low latency and in a scalable fashion, without requiring any ad-hoc mechanism (§7.2). This seemingly simple approach, however, requires support from the underlying streaming infrastructure, to execute distributed control operations with consistency requirements while minimizing synchronization overhead and enabling an extensible control programming model.

In this paper, we describe Chi, a control plane built on this idea of embedding control messages in the dataflow to execute low-latency control operations. We introduce a reactive programming model for handling control messages that allows users to encode a number of complex control policies. This hides the complex distributed nature of the underlying system and provides developers an intuitive model to understand the boundaries of data events to which the control events are applied. We then design a mechanism to execute these control policies in an asynchronous manner at each operator, avoiding synchronization and high runtime overhead. We show that by using our programming model and execution mechanism we can efficiently implement a number of control policies ranging from basic functionalities such as checkpointing and replay, to advanced ones with strong global consistency requirements, such as continuous monitoring, plan re-optimization, and parameter tuning (§5). Finally, we also discuss how our control plane infrastructure can be easily parallelized by having a separate control loop per operation, making the control plane scalable.

To validate our claims and evaluate the runtime performance of our new control-plane design, we implement Chi on top of *Flare*, one of the internal stream processing systems used in our clusters. Flare is built on top of Orleans [7], a highly efficient distributed actor framework, and uses Trill [16] as the underlying stream processing engine. While we choose to showcase our approach on top of Flare for ease of implementation and deployment on our internal clusters, our design is not tied to a specific platform, and with some additional engineering effort, it can be applied to existing systems, including Heron and Flink. Our experimental evaluation, based on production workloads and industry-standard benchmarks, shows that Chi is able to perform reconfiguration for a large stateful dataflow in 5.8 seconds on a 32-server cluster. Real-world use cases further show the effectiveness of Chi in helping developers automatically tune critical system parameters and reduce latency by 61%, reducing workload skew during runtime.

In summary, this paper makes the following contributions:

- An extensive study of today's stream-processing workloads through the logs collected from more than 200,000 production servers.
- A scalable and efficient control-plane that allows a streaming system to efficiently adapt to changes in workloads or environment.
- A flexible control-plane API that enables developers to easily implement custom control operations.
- Evaluation of our prototype implementation on a 32-server Azure VM cluster using production workloads from a cloud service provider and across a large set of control operations, including dynamic scaling, failure recovery, auto-tuning, and data skew management.

2. MOTIVATION

We performed an extensive measurement study by analysing the logs generated by more than 200,000 servers of a data-analytics cluster of a popular cloud provider. These clusters generate a significant amount of log data (10s PB/day), and queries on this data are executed by developers for debugging, monitoring, etc. Given the data size, we require a large cluster with adequate network and compute resources to process data in real time. Our setup is consistent with a recent analysis of Google production logs [20].

We begin by summarizing the main results of our analysis and discuss the opportunities that arise.

Workload Unpredictability: The load on servers ingesting log events is highly variable. Fig. 1(a) shows the heat-map of the normalized number of tuples produced per minute across a random subset of streams in our cluster. This shows two important phenomena. First, as evidenced by the different color patterns across horizontal lines, each stream exhibits a unique workload characteristic (*spatial* variability). Second, while some streams are dark (high volume) for most time, several streams exhibit burstiness (*temporal* variability), as shown by color spots the same horizontal line.

There are three main reasons for such variability in the log events:

- *Heterogenous Workloads*: The clusters generating these logs handle a variety of workloads ranging from typical big data jobs (e.g., filter, transform, join) to iterative machine learning workloads on behalf of hundreds of teams.
- *Failures & Debugging*: Failures are common at large scale. These failures (e.g., networking issues, power issues etc.) generate a large amount of error logs that lead to traffic bursts. In addition, when ingestion data is not sufficient, developers temporarily activate more verbose logs to perform in-depth debugging, which results in a higher volume stream.
- *Diverse Stream Semantics*: The logs we ingest have diverse semantics (e.g., info, verbose, debug, error, etc.) with various components in the service emitting with different characteristics. A service at the lowest level (e.g., storage layer) naturally produces the most logs since most requests involve store I/O transactions.

To quantify the degree of variability, in Fig. 1(b) we show a box-and-whiskers plot with the Y-axis representing the delta in terms of event count per minute on a subset of incoming data streams. The height of the box shows the difference in counts between the 25^{th} and 75^{th} percentiles. Beside the significant difference in the behavior of each stream, it is worth noting the high range of change observed *within* the same stream, denoted by the large number of outliers (blue dots in the figure). For example, for certain event streams, the event count can increase by up to tens of millions in just one minute, indicating that a timely response is critical for sustaining load spikes.

Data Diversity Another key element in determining the optimal query plan and resource allocation is the data distribution. To analyze its dynamics, we focus on the *key selectivity*, defined as the number of tuples that fall into a particular bucket. To analyze the dynamics of this parameter, in Fig. 1(c) we plot the selectivity of various grouping keys across time while in 1(d) we plot the selectivity over time across multiple grouping keys. The wide skew in the selectivity observed in both plots indicates that one-shot query planning (e.g., traditional cardinality-based optimizer) either globally or on a per-key basis will likely be sub-optimal over time.

Multi-tenant control policies: In our production environment, the same streaming system is used by many teams. Thus there is a big diversity of SLOs across queries from different teams, or even across different queries from the same team, leading to the need for multiple control policies to be active at the same time. For instance, we have many customers who are interested in consuming *verbose*, *info* and *error* logs. While verbose and error logs are used for debugging, info logs are often used to compute important metrics (e.g., billing metrics). One popular request made by our developers has been to provide stronger delivery semantics (e.g., exactly-once) for info logs and weaker delivery semantics (e.g., best-effort, at-least-once) for verbose/error logs. This highlights the importance of supporting multiple control policies either on a per-stream level or per-tenant level.

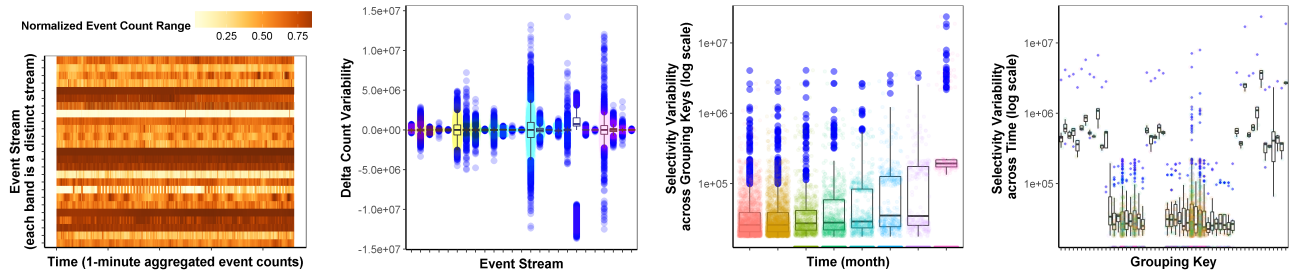


Figure 1: (a) Heatmap of a subset of data streams in our ingestion workload. Each horizontal band signifies a unique data stream being ingested, (b) Box plot representing the variability of delta count (order of millions) per minute (Y-axis) for a subset of incoming data streams (X-axis). (c),(d) Box plot depicting the selectivity properties of grouping keys used in a streaming join query from production traces. (c) shows selectivity of various grouping keys across time while (d) shows selectivity variability across the grouping key space.

Multi-tenancy also introduces new opportunities in system management and optimization. In our production traces we observe queries that share a significant overlap in terms of data sources and operators (e.g., users parsing logs in an identical way). Thus control policies could look at sharing computation across users or materializing intermediate data for later reuse. Further, with insight into the data streams and workloads, control policies could be used to automatically and transparently choose new data layouts (e.g., with partitioning) and storage engines (e.g., in-memory, database) that can improve execution efficiency. While all these optimizations are well understood in isolation [34, 35, 41, 33], applying them in an integrated manner to optimize the streaming workloads requires flexibility and scalability in the control plane.

Takeaway: Our trace analysis uncovered several interesting characteristics of our workloads – high volume (10s of PB/day), low latency requirement (SLO of seconds), widely diverse (100s of data streams) and dynamic (due to the nature of services producing these logs). Based on this, we can derive the following list of requirements for the control plane:

1. **Efficient and extensible feedback-loop controls**: Because of the diversity of our workloads, it is important to allow users or applications to make flexible late-binding decisions on their data processing logic for optimizing performance. From an extensibility standpoint, it should be possible to integrate with dedicated components (e.g., policy controllers such as Dhalion [25]).
2. **Easy control interface**: Since we intend the control plane to be used by application developers, having an easier programming interface is critical to its adoption.
3. **Minimal impact on the data plane**: The control plane should have limited or no impact on the latency and throughput of the data plane and it should be able to seamlessly cope with high control frequency and large dataflow graphs.

3. BACKGROUND

Chi is primarily designed for streaming systems that are based on a streaming dataflow computation model. In this section we provide the reader with the necessary background on the streaming dataflow computation model.

Many existing streaming systems, such as Naiad [31], StreamScope [29] and Apache Flink [10] adopt the streaming dataflow computation model. In this model, a computation job is represented as a directed acyclic graph (DAG) of stateful operators, where each operator sends and receives logically timestamped events along directed edges. Each operator maintains mutable local state. Upon receiving events, an operator updates its local state, generates new events, and sends them to downstream operators. Operators without incoming channels are source operators; those without outgoing channels are sink operators.

For instance, consider an example where the user is interested in tokenizing sentences into words, and count the windowed accumulated count (e.g., per hour) of each word across all sentences in a data parallel way. This query can be expressed in a LINQ-style language as below:

EXAMPLE 1 (WORD COUNT EXAMPLE).

```
stream.SelectMany(line => Tokenize(line))
      .GroupByKey(word => word)
      .TumblingWindow(OneHour).Count()
```

An instance of a dataflow graph for Example 1 is shown in Stage I of Fig 2. Note that operators $\{R_1, R_2\}$ maintain the accumulated counts for the words as their mutable local states. These states cover a disjoint set of keys, and jointly represent the entire key subspace, e.g., R_1 maintains the accumulated count for all words with the starting letter in the range ['a'-'l'], while R_2 maintains those for the range ['m'-'z'].

Dataflow Computation Model: Formally, a dataflow computation is represented as a DAG $G(V, E)$, where V represents the set of operators, and E, the set of edges connecting the operators, $u \rightarrow v$ represents a directed edge from operator u to v. We use $\{\cdot \rightarrow v\}$ to denote the input edges to v, and $\{v \rightarrow \cdot\}$ the output edges from v. An operator $v \in V$ is described by a triple (s_v, f_v, p_v), where s_v is the state of v; f_v defines a function that captures the computation run on v, i.e., $f : s_v, m_{e_i \in \{\cdot \rightarrow v\}} \longrightarrow s'_v, \{m'_{e_o \in \{v \rightarrow \cdot\}}\}$, meaning a function takes a single input message m, from an input edge e_i, and based on the current state s_v, it modifies the state of v to a new state s'_v, and generates one or more messages on a set of output edges $\{m'_{e_o \in \{v \rightarrow \cdot\}}\}$. Operators without input edges are called sources, and operators without output edges are called sinks. For generality, we represent the properties associated with v that are not part of state as p_v, e.g., p_v can define the maximum memory used by v. An edge e does not have any state but can hold properties p_e, e.g., the token size of windows before triggering back-pressure.

4. DESIGN

The intuition behind embedding the control plane into the data plane is that this enables re-using the existing, efficient data-plane infrastructure and offers developers a familiar API to write control operations, i.e., the same used to handle data events. Further, having control messages directly trailing data events provides a natural way to create custom boundaries between sequences of events. This makes it easy to implement asynchronous control operations

because control messages can be used to capture causal dependencies without requiring expensive global synchronization operations.

In this section, we show how we incorporate these principles into our control-plane design and provide several examples to describe how we can easily build different control operations on top. We conclude the section by describing some of the more advanced features of our design and discussing how Chi can be adapted to support BSP-style computations.

4.1 Overview

Chi relies on the following three functionalities. First, channels between operators support *exactly-once* and *FIFO* delivery of messages. Back-pressure is used to stop the message propagation when the buffer of the downstream operator fills up. Second, operators process messages one at a time and in the order that they have been received. Finally, the underlying engine provides basic operator lifecycle management capabilities. Specifically it allows us to start, stop and kill an operator. These functionalities are already supported by Flare, our internal streaming system, but they can also be found in other existing systems [10, 28, 40, 42].

Our system design uses *dataflow controllers* that are responsible for monitoring dataflow behavior and external environmental changes, and triggering control operations whenever needed. Users can define control operations, and submit them to the controllers.

A control operation is carried out through a control loop that is comprised of a dataflow controller and the dataflow topology itself. In general, a control loop consists of three phases: (Phase-I) The controller makes a control decision and instantiates a control operation with a unique identifier. A control operation is defined by implementing a reactive API (Section §4.2.2), and has control configurations for each operator (e.g., the new topology to scale out the dataflow stream). The control operation is serialized into a *control message* (Section §6). (Phase-II). The control message is broadcasted by the controller to all source operators of the dataflow. The control messages then propagate through the dataflow, interleaved with normal data messages. During the propagation, upon receiving a control message, each operator triggers the corresponding control actions—which can optionally attach additional data (e.g., the repartitioned state) to the control message—and broadcast the control message to all downstream operators. See Section §4.2.2 and Section §6 for more implementation details. (Phase-III) In the end, the sink operators propagate the control messages back to the controller, and the controller carries out post-processing.

We use Fig. 2 to illustrate this process. We consider a case where the controller wants to increase the throughput by modifying the topology and adding a new reducer.

(I) At the beginning, there are two map operators $\{M_1, M_2\}$ and two reduce operators $\{R_1, R_2\}$ that compute word counting for ingested sentences. These operators are stateful. For example, $\{R_1, R_2\}$ hold the accumulated count for all words, where R_1 maintains the counts for all words starting with ['a'-'l'], and R_2 maintains those for ['m'-'z']. The controller C is responsible for monitoring the memory usage of all operators and reconfiguring parallelism if needed. To simplify we omit the control messages that collect memory usage and focus on the parallelism reconfiguration process in the following discussions.

(II) Once the controller C detects that the aggregated memory usage of the reducers goes beyond a threshold, it makes a reconfiguration decision to start a new reducer R_3 to increase memory provisioning. In the new topology, the states of $\{R_1, R_2, R_3\}$ should be repartitioned so that R_1 holds the word counts for the range ['a'-'h'], R_2 for ['i'-'p'], and R_3 for ['q'-'z']. This reconfiguration needs to maintain the consistency of the states. It starts by broadcasting a control message with the new topology configuration to all the source nodes ($\rightarrow \{M_1, M_2\}$).

(III) When the source (mapper M_1 or M_2) receives this control message, the mapper immediately blocks the input channel while processing the message, updates its routing table with the new topology and broadcasts the message downstream ($\rightarrow \{R_1, R_2, R_3\}$).

(IV) When the reducer R_1 (or R_2) receives the control message, it blocks the input channel on which the message has been received. When the control messages from all input channels have been received, it updates its routing table and checkpoints its state. Next, it splits the state into two parts: the accumulated word counts for the range ['a'-'h'] and the range ['i'-'l'] (or for the ranges ['m'-'p'] and ['q'-'z']) and attaches the state that needs to be handled by R_3, i.e., the word counts for ['i'-'l'] (or for ['m'-'p']) to the control message and broadcasts along all output channels ($\rightarrow \{R_3, C\}$).

(V) When R_3 receives a control message, it blocks that input channel. If the control message originates from R_1 (or R_2), it records the state from the control message. When it receives control messages from all input channels, it proceeds to merge all the states received, generate the new state of the accumulated word counts for the range ['i'-'p'], and install a new function (from the control message) using the new state. Finally, it broadcasts on the output channel ($\rightarrow C$).

(VI) When C receives control messages from all the expected sink nodes $\{R_1, R_2, R_3\}$, the scale-out operation is completed. The controller then keeps monitoring the memory usage of these operators in the new topology and can decide to scale out/in if needed. This forms a feedback-loop control.

4.2 Control Mechanism

Next, we describe the core mechanisms underpinning Chi. We start by formally defining the dataflow computation model and explain how graph transformations occur. Then, we discuss the controller and operator APIs and provide an example control operation implementation. We provide a proof of correctness in §4.2.3.

4.2.1 Graph Transitions through Meta Topology

Formally, a user control operation C can be modeled as a transformation that converts a dataflow execution graph $G(V,E)$ to a new graph $G^*(V^*,E^*)$. For an operator v, such a transformation can change one or more entries in the triple (S,f,P). For an edge e, such a transformation can optionally change p_e. In particular, since the operator state S can capture state accumulated over a long time period, special care is needed to capture the transformation of states during reconfiguration (i.e., $G \rightarrow G^*$). That is, for $v^* \in V^*$, S_{v^*} is defined by a transformation function T on one or more nodes $\{v\} \subseteq V$, i.e., $T(\{S_v\}) = S_{v^*}$. In cases without ambiguity, we relax the notation and use $T^{-1}(v^*)$ to represent the set $\{v\} \subseteq V$ whose states S_{v^*} depends on.

Most existing systems (e.g., [10, 40]) adopt a *freeze-the-world* approach to perform the transformation i.e., stop G by appropriate checkpointing mechanisms, start G^*, migrate the old checkpointed state on G to G^* and resume the dataflow. However, this would likely trigger back-pressure, causing increased latency and loss of throughput, and in turn limits the frequency of execution and expressivity of dataflow reconfigurations. Therefore, in the design of Chi we opted for an asynchronous alternative: instead of affecting the transformation directly (i.e., $G \rightarrow G^*$), we introduce an interme-

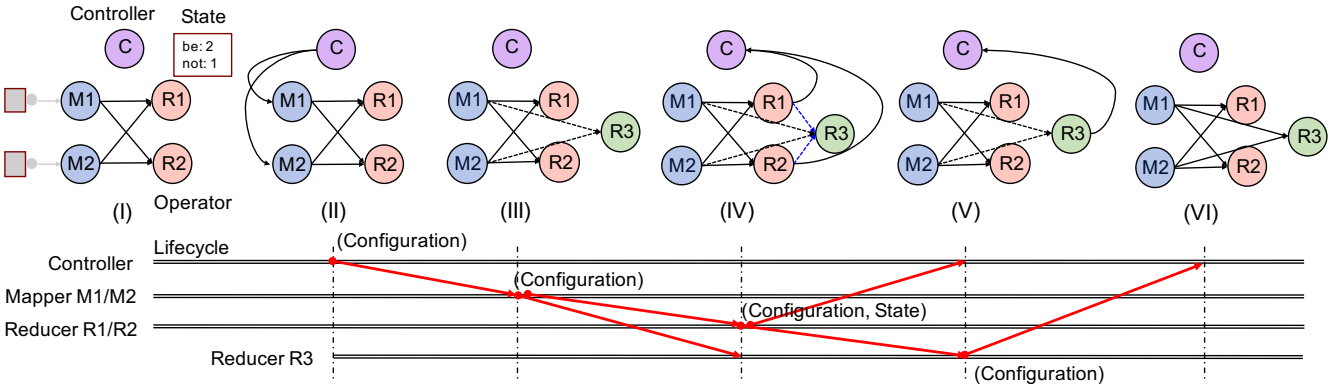


Figure 2: Scaling-out control in action where the user is interested in changing the number of reducers in Example 1.

diate *meta topology* G', which the control operation can temporarily utilize in order to complete the transformation asynchronously. That is, during propagation of the the control messages, each operator broadcasts messages to its downstream according to G'. The operators in $G' - G^*$, after processing the control messages, will shut down; while the operators in $G' - G$ will only start processing data messages after finishing the processing of control messages. When the control message propagation finishes, the resulting topology will be equivalent to G^*.

We derive the meta-topology G' for a control operation C as follows. In the most general case, we set $G' = G \cup G^* \cup E_{V,V^*}$, where $E_{V,V^*} = \{(v, v^*) | \forall v^* \in V^*, \forall v \in T^{-1}(v^*)\}$. In other words, the propagation graph for C consists of all the operators and channels from both the old and new execution graph, and channels that capture the dependency relationship between states of the old and new operators. While this approach can lead to doubling the size of the dataflow execution graph during reconfiguration, in practice, we can significantly reduce the propagation topology through appropriate pruning. For instance:

- **State invariance.** If a control operation does not change a node v's state S_v, we can collapse the corresponding new node $v^* \in G^*$ with $v \in G$, and merge the input and output channels adjacent to v. For example, in Fig 3(a), M_1^* (M_2^*) can be merged with M_1 (M_2) respectively.
- **Acyclic invariance.** Aggressively merge the old and new topology as long as we can guarantee the graph acyclicity. For instance, in Fig 3(a), we can further collapse R_1^* (R_2^*) with R_1 (R_2) without breaking the acyclicity. This is guaranteed by (i) the functional query interface which ensures initial dataflow topology is acyclic as well as (ii) the pruning algorithm which ensures that no cycles is introduced during optimizing a meta topology. For example, for scale-out/in reconfiguration, the pruning algorithm uses consistent hashing as the state allocation scheme to avoid introducing cycles when re-partitioning states.

By applying the above pruning rules repeatedly in Fig 3(a), we obtain the graph shown in Stage (IV) in Fig 2.

4.2.2 Control API

We next describe features of our control API that enable developers to implement complex control operations. Chi's API allows expressing different behavior across the following dimensions: (1) *spatial*, e.g., behavior of $\{M_1, M_2\}$ being different than $\{R_1, R_2, R_3\}$, and (2) *temporal*, e.g., behavior of R_3 when receiving the first control message vs. the last in Fig 2.

To enable such a flexibility, we abstract control operations and provide the following capabilities to users:

- **Configuration injection**: We allow the same control operation to carry different configurations for different operators. The configurations instruct operators to take different control actions. Configurations are injected into a control message (see Fig. 6 for implementation). The runtime transparently instantiates the control operation appropriately with the correct configuration at each operator. In the scaling-out example shown in Fig. 2, the injected configurations need to instruct (1) mappers to reconnect output channels, (2) the reducers $R1$ and $R2$ to migrate states, and (3) the new reducer $R3$ to accept migrated states. Shown in Algorithm 1(L1-11), $R1$ is injected with *SplitState* (L6) and *LoadFunc* (L9) instructions, and $R3$ with *MergeState* (L8) and *LoadFunc* instructions.
- **Reactive execution**: Chi exposes a reactive (event-driven) programming interface that users can leverage to define control operations. A control operation comprises two sets of event handlers: those executed at the controller {*OnInitAtController*, *OnBeginAtController*, *OnNextAtController*, *OnCompleteAtController*, *OnDisposeAtController*}, and those executed at the operators { *OnBeginAtOperator*, *OnNextAtOperator*, *OnCompleteAtOperator*, *OnDisposeAtOperator*}. These event handlers offer users great flexibilities to collect metrics or modify configurations when the controller and operators receive the first, next and last control messages. The *OnInitAtController* is called when initializing the control operation and allows users to inject configurations into the control message. The *OnDisposeAtController* and *OnDisposeAtOperator* are called when the control operations are disposed. They are usually used for releasing resources. The runtime handles the correct invocation of these handlers and state transition as shown in Fig. 3(b), thus supporting expressing complex control logic in a safe manner.

Blocking behavior: In the example in Fig. 2, operators always block the input channel upon receiving a control message from it. We find that this is a fairly common pattern in many control scenarios e.g., checkpointing and scale-in/out operations. To simplify implementation of complex control, we provide a layer of abstraction that allows the users to implement their control operations in both blocking and non-blocking ways. We do this by classifying control messages into two categories: (1) *blocking*: where the operator blocks the corresponding channel on which the control message is received and subsequently unblocks it only when all the control actions are finished on that operator, and (2) *non-blocking*: where the operator does not block the input channel and continues to receive other data/control messages on that channel. We believe such abstraction is useful for users to express more advanced control operations. For instance, blocking control messages are usually useful for control that affects states, while non-blocking control messages are useful for the other cases, e.g., monitoring.

Example: We demonstrate usage of the control API using the example shown in Fig. 2 where we want to scale-out from G into

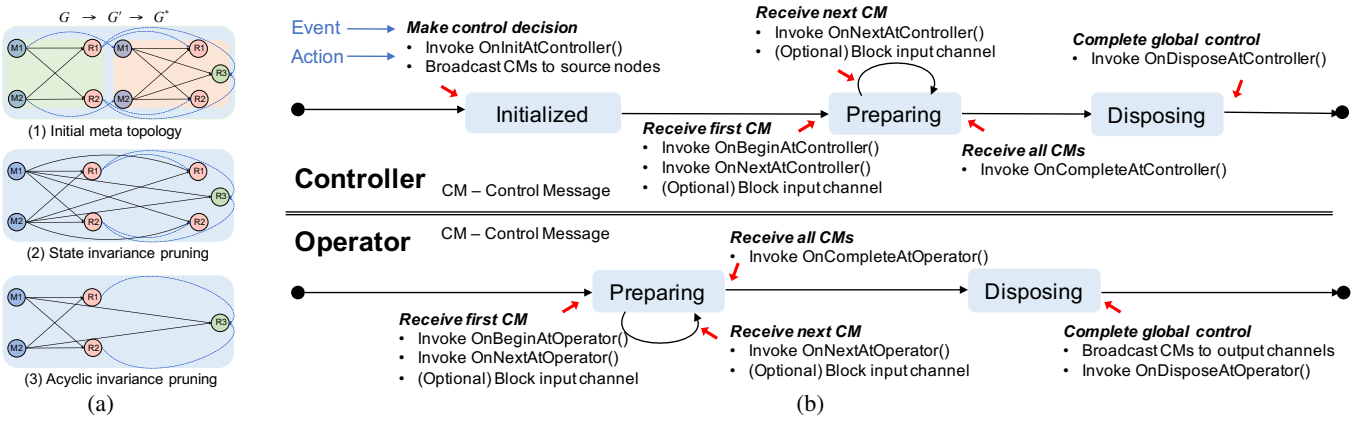


Figure 3: (a) Pruning the meta topology using state invariance and acyclic invariance. The final stage here is equivalent to Stage (IV) in Fig 2. (b) State machine transitions for controller and operator

G^* through G' (shown in Fig. 3(b)). Algorithm 1 shows a pseudo implementation of the dataflow reconfiguration control operation. Once the reconfiguration decision is made, the developer creates a *blocking* control message. In *OnInitAtController*, a blocking control message is created and injected with configurations for triggering control actions at different operators. Specifically, as explained in §4.2.1, an edge $(v, v^*) \in E_{V,V^*}$ describes a dependency between the states of v^* and v, operator v needs to be configured to split its states and ship the corresponding part of the state to v^*, while operator v^* needs to be configured to load and merge the received states (L5-8). For example, as in Fig. 2, R_1 needs to split the state and ship the accumulated counts for the range ['i'-'l'] to R_3. Once the topology or state is changed, the operators need to reset associated computation functions (L9). Such a control message is broadcast to source operators. It first initializes *session variables* that hold migrated states and control actions (L15-16) in the *OnBeginAtOperator* function. The migrated state is gradually accumulated until all parts of the state are received by *OnNextAtOperator* (L18-19). Once receiving all messages, the operator (shown in the *OnCompleteAtOperator* function) performs control actions including move away states that do not hold any more according to the new state key range (L23-25), merges states given by others (L26-27), and resets the function (L28-29). Once the controller receives the control messages from sink operators, the control operation is marked completed in *OnCompleteAtController* (L12).

4.2.3 Correctness Properties

Chi provides correctness properties that can help users prove the correctness of their control operations.

THEOREM 1. *Consider a control operation that changes a graph from G to G^* using a control message and a state transformation function T. The control operation has the following properties:*

1. *The control operation will terminate in finite time.*
2. *If a pair of operators v, v' satisfies (a) $v \to v'$ is an edge in G or G^*, or (b) $v \in T^{-1}(S_{v'})$, then v will always invoke OnCompleteAtOperator before v'.*

Furthermore, we introduce *safe blocking control operations*—a special type of blocking control operations whose control actions at each operator only read/write the corresponding operator state in *OnCompleteAtOperator*. Safe blocking control operations have stronger properties—the semantics of safe blocking control operations is equivalent to the freeze-the-world approach—which can facilitate users to understand the semantics and correctness of their customized control operations. For detail explanation and proof of Theorem 1 and the properties of safe blocking control operations, please refer to Appendix B.

4.3 Advanced Functionalities

We discuss key advanced functionalities of Chi necessary to cope with production workloads.

Multiple Controllers Thus far our description assumes a single dataflow controller that enforces a control operation. Our design is able to naturally scale out the controller by allowing multiple concurrent controllers for a single dataflow. For instance, users can have one controller per desired functionality, e.g., one controller takes care of monitoring the health of the dataflow execution, another one periodically checkpoints operator states, while yet another takes charge of reconfiguring dataflow execution graphs in case of workload spikes. Multiple controllers can function at the same time as long as the serializability of control messages for different control operations are guaranteed. In order to perform cross-dataflow control operations, e.g., coordinating resource allocation across multiple dataflows, we can introduce a global controller that can interact with each dataflow's controller.

Broadcast/aggregation trees. In practice, a dataflow graph usually has a large number of source operators (and sometimes, sink operators). In such a topology, the controller can quickly become a bottleneck due to the large fan-in/out. To mitigate this, we leverage a simple technique such as inserting a spanning broadcast (aggregation) tree before (after) the source (sink) operators.

Dealing with congestion/deadlock. When congestion arises, e.g., due to network or CPU bottlenecks, our back-pressure mechanism is triggered and all messages, including control messages, are delayed. This could be particularly critical if these messages are part of a control operation to alleviate congestion. One option might be to have two separate queues and give control messages higher priority, so that in case of congestion they are delivered first. This, however, would break the ordering of control and data messages, thus making it hard to maintain consistency. Therefore, we wait for finishing processing the message. This is similar to the approach taken by other systems such as Flink [10] and Spark Streaming [42].

Fault tolerance One of the main benefits of integrating control and data plane is that failures in the control plane are handled in the same way as failures in the data plane. More specifically, if control messages are lost, the underlying data channel is responsible to retransmit them until they are acknowledged by the other end. In case of network partition, the controller will eventually time out and will restart the control operation.

Algorithm 1 Dataflow Reconfiguration Operation

Assumption: Each operator has a context (*ctx*). In the controller, control operation can access G' and G^* as well as G' and E_{V,V^*} (See §4.2.1). Users can create session variables (names start with the $ mark) that live through function scopes. A graph vertex has properties including stage (*stg*), state key-range, and function (*func*). A control message has properties including source (src), destination (dest), configuration dictionary (*confs*) and control payload. A configuration is assigned with a list of instructions.

```
1: function OnInitAtController
2:     msg := new BlockingControlMessage()
3:     for v in G' do
4:         instructions = []
5:         if v in Src(E_{V,V*}) then
6:             instructions.Add(new SplitState(v.keyrange))
7:         if v in Dest(E_{V,V*}) then
8:             instructions.Add(new MergeState(v.keyrange))
9:         instructions.Add(new LoadFunc(v.func))
10:        msg.confs[v] := instructions
11:    return msg
12: function OnCompleteAtController
13:    ctx.CompleteOperation(this)
14: function OnBeginAtOperator(msg)
15:    $inState := new Dict<Key, State>()
16:    $instructions := msg.confs[msg.dest]
17: function OnNextAtOperator(msg)
18:    for key, state in ParseState(msg.payload) do
19:        $inState[key] := state
20: function OnCompleteAtOperator
21:    outState := new Dict<Key, State>()
22:    for i in $instructions do
23:        if i is SplitState then
24:            for key, state in ctx.state.Split(i.keyrange) do
25:                outState[key] := state
26:        if i is MergeState then
27:            ctx.state.Merge($inState, i.keyrange)
28:        if i is LoadFunc then
29:            ctx.func.Load(i.func)
30:    return outState
```

Chi allows developers to implement various policies to handle operator failures. For ease of adoption, we implement a checkpoint-replay policy on top of Chi for handling operator failures by default. This policy will first rollback the data flow stream to the last checkpoint and then it will re- insert the lost control messages. Failures in the controllers are handled through checkpointing its own state into a durable store and restoring the state upon launch of a new instance of the failed controller (typically handled by a watchdog).

4.4 Compare Chi with Existing Models

We next compare Chi with existing control models for streaming systems (Table 1 summaries the comparison). There is a variety of control models being developed, tailored for different computation paradigms, i.e., BSP-based micro-batching versus record-at-a-time. We compare the different models according to the consistency, ease-of-use, overheads and scalability.

Consistency. Many useful control operations, e.g., scaling-out, state repartitioning and checkpointing, do demand consistency. The consistency can be easily realized in BSP systems through the uses of the synchronous barriers between parallel computation (named Synchronous Global Control). This kind of consistency can be achieved by Chi too using blocking control messages, acting as a barrier asynchronously moving inside a dataflow. If required, Chi can replicate the BSP model. Specifically, the controller can act as such a barrier node as shown in Fig. 4(a). When a stage starts, it generates (1) a blocking control message (denoted by *S*) that installs tasks at operators, followed by (2) a data message (denoted by *D*) that describes input data, and (3) a blocking control message (demoted by *E*) that marks the completion of a stage. When receiving all completion messages, the controller starts the next stage by repeating the same sequence of messages.

Consistency can also be realized in record-at-a-time systems by freezing the entire dataflow during reconfiguration. This, however, requires the system to halt (like BSP), and thus motivates systems like SEEP [11] that reconfigure workers asynchronously (named Asynchronous Local Control); but sacrificing the barrier semantics. The asynchronous local control can be implemented using Chi's non-blocking control messages as well. The absence of barriers makes it hard to implement many useful control operations. Consider a dataflow that applies filters to a stream that has x and y fields (shown in Fig. 4(b)), where the filter parameters are stored in two data stores due to privacy regulation. Hence, the stream must be replicated to be filtered in parallel. The filter results are joined in the end. In a system that provides only asynchronous local controls, it is unable to support control requests that request simple concurrent reconfigurations, such as changing filter selectivity from $x < A$ and $y < B$ to $x < C$ and $y < D$. This is because each replicated tuple must be processed with all new configurations at once, rather than a mixture of them. To ensure consistency, as shown by the Algorithm 3 in SEEP [11], the dataflow has to block ingestion, recover operator checkpoints, apply new configurations, replay the output of the replicate operator since its last checkpoint, and unblock ingestion.

Ease-of-Use. In addition to consistency guarantees, Chi provides a flexible control plane interface with comprehensive automation support. Users declare control logics in a reactive manner, and rely on a runtime to automatically manages the states of control operations, handles failures, and performs operations asynchronously. In contrast, existing control models lack programmability and runtime support. For example, Flink implements a distributed check-

	SGC Models	ALC Models	Chi
Consistency	Barrier	None	Barrier/None
Semantic	Simple	Hard	Simple
Latency	High	Low	Low
Overhead	High	Implementation-dependent	Low
Scalability	Implementation-dependent	Implementation-dependent	High

Table 1: Comparing Chi with the Synchronous Global Control (SGC) and Asynchronous Local Control (ALC) models.

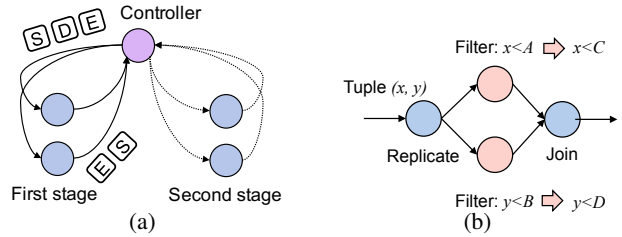


Figure 4: (a) Executing a two-stage dataflow using the BSP model implemented by Chi. (b) A dataflow that needs to modify filters at once.

point algorithm which is unable to support general control operations that, for example, require changing state. More recently, Dhalion studied the high-level representation of a control policy, with a particular focus on identifying symptoms and therapies of detected anomalies. It relies on the underlying system, i.e., Heron, to provide actual reconfiguration capability. Hence, Dhalion does not has a control plane runtime as Chi which can be applied onto general record-at-a-time streaming systems.

Overhead. When reconfiguring a BSP system, the entire dataflow is halted. This is particularly detrimental for online streaming systems and affects both latency and throughput. Furthermore, the BSP barriers severely constrain the frequency and timing of control operations. To amortize the scheduling and communication cost, a BSP barrier interval is often set to be no smaller than seconds [40].

As mentioned above, reconfiguring a record-at-a-time system requires freeze-the-world as in Flink (Section §7), or re-computation through data replays as in SEEP. On the one hand, freeze-the-world incurs a similar overhead than the synchronous global barrier. On the other hand, replaying data is expensive in production, despite reconfigured operators being often already short of resources. Our traces show that buffering all intermediate outputs in preparation for replays require a significant amount of memory, several orders of magnitude larger than the one consumed by the operator computation state. Replaying such a large buffer state not only consumes significant bandwidth, but also blocks the operator for a long time.

In Chi, changes are applied on the asynchronous global barriers. There is no need for freeze-the-world or data replay. Blocking behaviors are local to each operator. There is no global blocking that freezes the entire dataflow, thus reducing data plane overheads.

Scalability. Existing streaming systems usually has a separate control plane. This duplicates resources for the control plane to deal with typical distributed system issues such as fault tolerance and scalability. Chi, however, embeds the control plane into the data plane. As the data plane is optimized for handle high volume of data, Chi benefits from the same optimization, e.g., zero-copy data movement and broadcast/aggregation trees for large fan-in/out in dataflows.

Furthermore, existing systems mostly adopt centralized controllers. Reconfiguration requires a large number of control messages, thus incurring single-point of failure and performance bottleneck. On the contrary, Chi can naturally partitions workloads on a controller. Dataflows are managed by parallel controllers. A dataflow controller further splits to operate control operations in parallel. All these controllers run on multiple servers and keep state on a distributed store, implying a high-scale architecture.

5. APPLICATION EXAMPLES

To demonstrate the flexibility of our approach, we illustrate three control applications that we implemented using Chi.

Continuous Monitoring. Due to unpredictability of our workloads, unlike traditional batch processing systems where jobs can be tuned offline to achieve optimal performance, streaming pipelines have to be continuously monitored and optimized on-demand in order to achieve high performance and robustness. We show an example of using Chi to continually collect measurement data of all operators, a foundational block for detecting and acting upon interesting characteristics of the system such as overload/underload, skew/-drift in data distribution, intermittent environment-related bottlenecks and mitigating stragglers. The monitoring control operation implementation is shown in Appendix A.

Notice that the controller no longer needs to ping each individual operator separately to collect statistics. Instead, the metrics are collected and aggregated along in a scalable tree fashion. §7 shows the

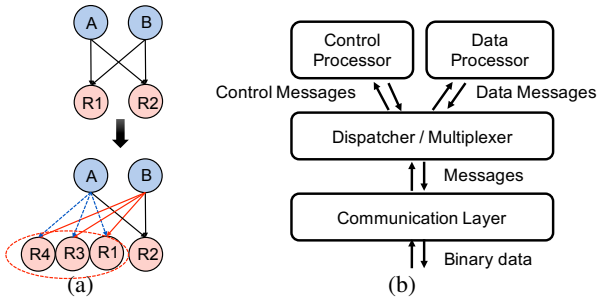


Figure 5: (a) Handling Skew (b) A Chi-enabled Flare operator architecture.

evaluation of using the monitoring control for collecting per-join-key cardinality that helps in identifying skewness in the join space (which can then be used for straggler-mitigation).

Dataflow Reconfiguration. Dataflow reconfiguration is important for several interesting use-cases such as adding/removing operator from a given query, increasing degree of parallelism of an operator and exploiting computational reuse.

Besides scale-in/out, one can also carry out more complex reconfigurations including changing the query plan. For instance, Fig. 5(a) demonstrates how we can change the query plan to alleviate stragglers when we find skew in a streaming join query. Assume originally streams A and B are joined using a shuffle join [27], where mappers read data from A and B respectively, and partition and route the data to the corresponding reducer based on the join key; reducers on receiving the data, join the tuples with the same key together. Due to skewed key space, reducer R_1 receives much more data than R_2. At this point, we can change the query plan by adding reducers $\{R_3, R_4\}$ to share the load for R_1. The idea is to let A (assume A has a significantly higher workload than B) partition R_1's load into $\{R_1, R_3, R_4\}$; B broadcasts R_1's load to $\{R_1, R_3, R_4\}$. This reconfiguration requires R_1 to replicate its internally maintained hash table of data from B to R_3 and R_4, while partitioning and redistributing the hash table of data from A to R_3 and R_4.

Auto Parameter Tuning. Big data systems have many parameters that are very hard to tune even for very experienced engineers. Chi can be used for automatic parameter tuning by leveraging both monitoring and dataflow reconfiguration in a tight control loop to simulate A/B testing of multiple instances of a single query. For instance, many existing streaming systems use micro-batching to tradeoff between latency and throughput. Whereas a large batch size provides good throughput, it does so at an increased latency. Tuning the right batch size is a tricky problem. One solution through Chi would be to continuously monitor latency of the data plane and adjust the batch size when we see considerable fluctuations in the observed latency until we obtain the maximum throughput with the desired latency.

6. IMPLEMENTATION & DISCUSSION

Distributed runtime. To showcase the performance and flexibility of Chi, we implemented it on top of Flare, a streaming system used internally by our team. Flare is built on top of Orleans [7]—a virtual actor framework—as a runtime and Trill [16] as the operator-level stream processing engine. By leveraging Orleans, Flare achieves decentralized scalability, specifically: (1) nodes can join/leave the cluster without notifying master nodes, and (2) the lifecycle of an actor is automatically managed by the platform, which transcends the lifetime of in-memory instantiation and particular servers.

An operator in Chi has a stacked architecture embedded into the Flare operator which is a single-threaded environment, as shown

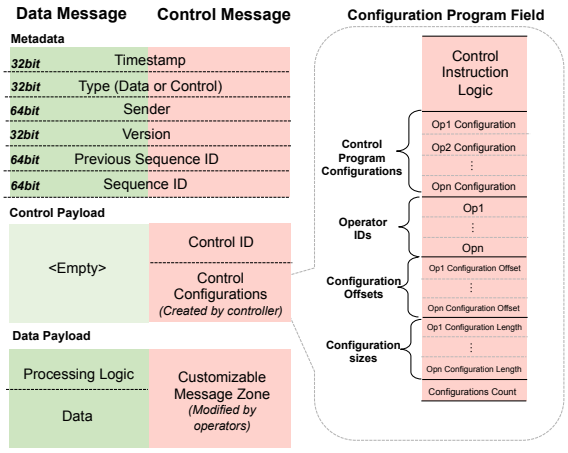


Figure 6: Control message structure in Chi

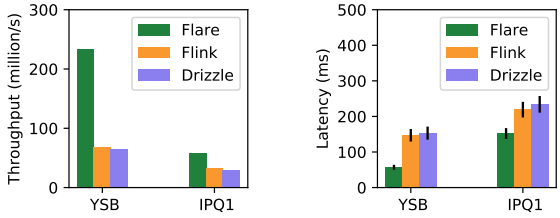
in Fig. 5(b). (i) The *communication layer* provides FIFO exactly-once data communication channels with back-pressure that mimics the TCP. (ii) The *message dispatcher/multiplexer* invokes the corresponding processing module based on the types of messages, and multiplexes their outputs down to the communication layer. (iii) The *data processor* applies a Trill pipeline onto data messages. (iv) The *control processor* invokes a series of local control actions based on received control messages. It loads the corresponding control configuration, manages state machines, and invokes user-defined control actions accordingly.

Flare further provides the following functionalities to simplify control operations: (i) a user-friendly state management functionality which models operator states as a key-value map and can automatically split and merge states using a user-defined partition scheme on the key, (ii) a query compiler (extensible with custom optimization rules) similar to Spark's catalyst [4] that converts LINQ queries into a distributed dataflow, and (iii) file server that allows runtime extensibility, where users can upload new control operation dependencies and have them loaded at runtime.

Custom serialization. Control messages may carry/propagate large payloads including configurations and states/metrics/etc. from each operator. Since serialization and deserialization at each operator may introduce unnecessary overhead, we implemented a zero-copy operation that allows us to extract the necessary pieces (e.g., configuration, payload of interest) from the byte buffers without deserializing the entire message.

Fig 6 shows the structure of a control message. Each message includes three basic components: 1. Metadata field that is used to ensure FIFO exactly-once delivery; 2. Configuration payload field to store configurations for different operators; and 3. Data payload for an operator to insert any control-specific data e.g., re-partitioned state while scaling-out. Control messages are generated either by a controller (before control instructions are applied to any operators) or by an operator (after a control operation has triggered but before control message has been propagated to succeeding operators).

Portability. While we implemented our approach on top of Flare for ease of implementation/deployment in our internal clusters, our design is not tied to a specific platform. Chi can be applied to other systems as long as the systems provides FIFO at-least-once or exactly-once delivery semantics, and in-order message processing. These modest requirements are offered by most contemporary streaming systems such as Flink [10] and SEEP [11]. A typical porting plan includes (1) porting the communication layer if the underlying system does not provide FIFO exactly-once message delivery, (2) porting the message dispatcher/multiplexer and the control processor, and (3) reusing the existing data processor.



(a) Throughput (b) Latency

Figure 7: Flare's throughput and latency against Flink and Drizzle for the YSB and IPQ1 workload

7. EVALUATION

In this section, we evaluate the performance of Chi using a number of micro-benchmarks and two real-world benchmarks. The first real-world benchmark focuses on dynamically scaling in/out resources while the second assesses Chi's ability to handle control and data failures. These results demonstrate that Chi incurs negligible overhead, even under high data/control load and large dataflow graphs, and it is able to quickly react to changes in the workload or failures. To show the flexibility of our approach, we also report on the experiments with two more advanced case studies, i.e., handling a skewed key distribution and auto-tuning for meeting SLOs.

7.1 Experimental Setup

Our experimental cluster comprises 32 DS12v2 instances in Azure. Each virtual machine has 4 vCPUs, 28 GB RAM and a 10 Gbps network connection. We consider one public workload, the Yahoo! Streaming Benchmark (*YSB*) [43], and one private workload based on production traces, *IPQ1*, which consists of multi-stage queries with complex window aggregations and streaming joins. YSB is rather light-weight in terms of data handling (bytes/event) while IPQ1 is computationally heavier (KB/event). As explained in §6, we implemented Chi on top of Flare, a streaming engine built on .NET CLR and is used internally by our team. When required, we compare against Drizzle [40], a fork of Apache Spark v2.0.0 (a BSP-style engine) with an optimized scheduler for streaming scenarios, and Apache Flink v1.3.2 [10] (a continuous dataflow engine). For all our experiments, we warm up the JVM and CLR before taking measurements to discount bootstrapping effects.

Flare performance To help understand whether the underlying engine that we used for Chi is competitive with existing systems, we compare the base performance of Flare against Flink and Drizzle. In Fig. 7, we show the results in terms of throughput and latency for the three systems when using the YSB and IPQ1. For the throughput experiments (Fig. 7(a)), we set the latency SLO to 350 ms and maximize the throughput while for the latency experiment we fix the ingestion rate at 20 million tuples/s and minimize the latency. Following a common practice [40], we define latency as the time it takes all events in the window to be processed after the window has ended. For instance, if a window ends at time a and the last event from the window is processed at time b, the processing latency for this window is calculated as $b - a$. The results obtained for Flink and Drizzle are consistent with what previously reported in the literature [40, 26] and confirm that Flare's performance is comparable with these systems.

7.2 Micro-benchmarks

In this sub section, we study the interplay between the control and data planes. Specifically, we are interested in the operational

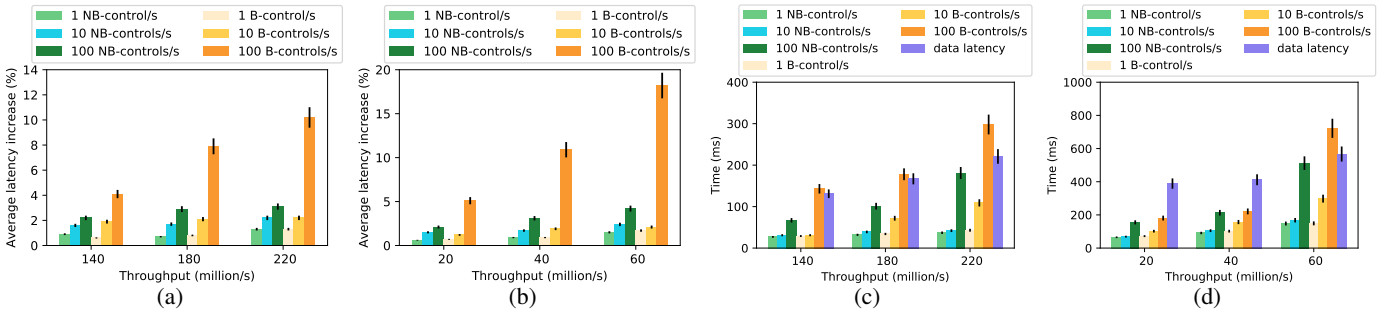


Figure 8: (a)(b) Control-plane overhead under different control load for YSB and IPQ1, (c)(d) completion time for control messages under different control load for YSB and IPQ1.

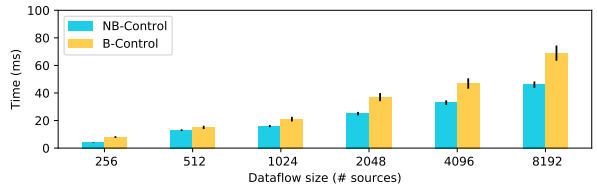


Figure 9: Control messages completion time under different dataflow sizes.

overhead and scalability aspects of Chi. To this end, we vary the data-plane load under three different CPU regimes (resp. 50%, 75%, and 90%) and set the ingestion rates for the two workloads YSB and IPQ1 accordingly. For the computation-heavy IPQ1 this results in 20, 40, and 60 million events/s (corresponding to 56%, 73% and 94% average CPU in our 32-server cluster) while for the communication-heavy YSB, this leads to 140, 180, and 220 million events/s (resp. 51%, 74%, and 89% average CPU). For the control load, we consider two instances of the control message, using blocking (B-Control) and non-blocking (NB-Control) messages. To accurately isolate the impact of Chi and avoid biasing the result with custom control logic (e.g., CPU-intensive user code), we use only NoOp control messages in our experiments.

Does control plane affect the data plane? We now assess the overhead introduced by Chi. In Fig. 8(a) and 8(b), we show the relative increase in latency for the two workloads when varying the control-plane load from one control message/s (representative of dataflow management tasks, e.g., checkpointing, and reconfiguration ones, e.g., scale in/out) to 100 control messages/s (representative of monitoring tasks). This range covers all control scenarios that we have observed in production and we believe that the same should hold for the vast majority of use cases.

These results show that both non-blocking and blocking control have low overhead. This is because neither of these controls requires global synchronization — control events are propagated and processed asynchronously just like data events. For example, at the highest load of the computation-intensive IPQ1 workload (60 million events/s), Chi incurs lower than 20% latency penalty to the data plane, even for high control load (100 messages/s). This is important because the average CPU utilization was already at 94% (the latency could be further reduced by scaling out). As expected, blocking control messages are more expensive than non-blocking ones. This is due to the fact that blocking control messages require local synchronization which block the input channel temporarily.

Does a busy data plane limit the control plane? Next, we study the impact of the data-plane load on the completion time of control messages. The concern is that by merging control and data events, high data-plane load might negatively affect the performance of the control messages. To verify this, we repeat the experiment and we measure the completion time of the blocking and non-blocking messages (see Fig. 8(c) and 8(d)). As a reference point, we also show the latency of the data plane ("data latency" bar in the graph). The message latency is defined as the difference between the timestamps of the message M entering the system and the timestamp of when the last message triggered by M leaves the system. In addition to queuing delays, the completion time of a data (resp. control) message also depends on the complexity of the actual data processing (resp. control logics). Hence, a control message can complete faster than data messages if there is no backlog and messages are processed immediately upon receiving.

In most cases control messages complete relatively quickly in comparison to the data message delivery. Also, when the control load is low (one or ten control messages/s), the completion time is relatively unaffected by the increasing data-plane load. We observe, however, that at the highest load (resp. 220 million events/s for YSB and 60 million events/s for IPQ1), the completion time for a control message is similar or higher than the data latency. This is particularly evident for the blocking messages. The reason is that the latter introduces a local synchronization at each operator and, at high load, there is a higher variability in latencies along different paths in the dataflow (e.g., due to work imbalance). As already discussed in §4.2.2, however, we observe that blocking and non-blocking control messages are semantically equivalent, although they differ in both their implementation and execution model. Thus, to reduce completion time, developers can always convert the blocking control messages to a non-blocking version.

Is the control plane scalable? As discussed in §6, Chi can scale to large dataflows with a large number of sources (sinks) by using a broadcast (aggregation) tree to exchange control events between the controller and the operators. To evaluate its effect, in Fig. 9 we show the completion time of a control message as we increase the number of sources. The results show that the completion time increases logarithmically and remains well below 100 ms even for very large dataflow graphs (8,192 sources).

7.3 Adaptivity and Fault-Tolerance

The previous sections have shown that Chi incurs low overhead and completion time, and can scale to large data-flow graphs. Hereafter, we study how Chi leverages these properties to improve the adaptivity to workload changes (dynamic elasticity) and to failures (fault recovery) of the data plane using the YSB workload.

Dynamic Elasticity. We set the ingestion rate of our 32-server cluster to $30M$ tuples/sec. At time $t = 40$ s, we double the ingestion rate to emulate a workload spike. At the same time, we start the scale-out process on all three streaming engines, using their respective strategies. For instance, in Apache Flink, when the scale-out process begins, we immediately invoke a Savepoint [24] operation.

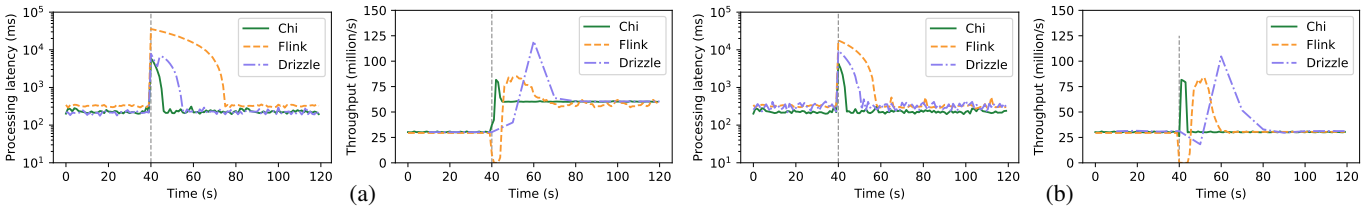


Figure 10: Latency and throughput changes under (a) dynamic scaling and (b) failure recovery

It externally stores a self-contained checkpoint that is useful for stop-and-resume of Flink programs, and restarts the dataflow with a larger topology. We plot the results in Fig. 10(a):

CHI. At the scale-out time ($t = 40s$), there is no noticeable drop in throughput while latency temporarily spikes to 5.8s. However, the system quickly recovers within 6s to a stable state.

APACHE FLINK. At $t = 40s$ throughput drops to zero (due to freeze-the-world initiated by Savepoints) for five seconds. Processing latency, instead, spikes up to 35.6 s and it takes 31 s after restart to return to a steady state.

DRIZZLE. At $t = 40s$, there is no visible drop in throughput because, since Drizzle uses a BSP model, it was not feasible to measure throughput continuously. Similar to Flink, the processing latency spikes to 6.1 s and it took an additional 10 s before stabilizing. Notably, Drizzle starts reacting to the workload spike after 5 s. This is because the workload spike happened in the middle of a scheduling group [40], where the system cannot adjust the dataflow topology — the larger the scheduling group size, the slower it reacts to workload changes.

Fault Tolerance: Next, we examine the default failure recovery operation (Section §4.3) implemented on top of Chi. As in the previous experiments, we set the ingestion rate to $30M$ tuples/sec. We also enable checkpointing for all three streaming systems every 10 s. At time $t = 40s$, 5 s after the last checkpoint, we kill a virtual machine to simulate a failure in the dataflow and we start the recovery process on all three streaming engines (see Fig 10(b)):

CHI At failure time ($t = 40s$), there is no noticeable drop in throughput while latency temporarily spikes to 4.3 s. However, the system quickly recovers to a stable state within 5 s.

APACHE FLINK At $t = 40s$, throughput drops to zero for five seconds (system downtime), as Flink's recovery mechanism redeploys the entire distributed dataflow with the last completed checkpoint [23]. The processing latency spikes up to 17.7 s. It then takes 13 s after restart to return to a steady state.

DRIZZLE At $t = 40s$, we observe no drop in throughput due to the reason described in the scale-out experiment. The processing latency spikes to 9.1 s and it takes 11 s to restore the stable state. There is a throughput drop (during recovery) around $t = 50s$ due to Spark's recovery mechanism as Spark needs to re-run the lineage of the failed batch.

7.4 Chi in Action

We conclude our evaluation by showing the performance of Chi using two real-world complex control operations, one focusing on parameter auto-tuning to meet SLOs and the other addressing workload skew.

Auto-Tuning for SLOs: Streaming systems include a number of parameters that are used to tune the system behavior. This provides great flexibility to the user but at the same time greatly complicate her task due to the large size of the parameter space. As a representative example, we focus our attention on the batch size. Batching is used by streaming system to amortize the per-tuple processing cost. The batch size has a direct impact on system performance. Identifying a priori the optimal value is often a non-trivial task as we show in Fig. 11(a) in which we plot the relationship between latency and batch size for the IPQ1 workload.

To show how Chi can help to correctly tune the batch size, we implement a control operation consisting of a *monitoring* task, which collects latency information, coupled with a *reconfiguration* task, which updates the batch size to meet the desired trade-off between latency. To illustrate how this works, we show an example run in11(b) in which we set up the control operation to optimize the batch size given an ingestion rate of 60 million events/s and an upper bound on latency of 500 ms. Every 30 s the controller collect latency samples, updates the moving average of the processing latency, and, if needed, executes an optimization step.

Initially (Phase-I) the controller opts for a conservative batch size of 40K events while measuring the latency and throughput. It quickly realizes that this batch size is not sufficient for meeting the ingestion rate — it overwhelms the system causing frequent backpressure, which is clear from the throughput fluctuations in the figure. Then, starting at the 30-second mark (Phase-II), the controller doubles the batch size to 80K which leads to a more stable throughput reducing the processing latency to $\approx 500ms$. At the 60-second mark (Phase-III), the controller attempts a second optimization step by doubling the batch size to 160K but soon detects that by doing so it will not be able to meet the latency SLO (500 ms). So finally it reverts the batch size back to 80K (Phase-IV).

Detecting and Adapting to Workload Skew: As shown by the analysis of our production workloads in Fig. 1(c) and Fig. 1(d), join keys exhibit high temporal and spatial skewness in data distribution. This variance can cause imbalance, lead to stragglers, and ultimately violate SLOs. To show the impact of this, we use another production workload, *IPQ2*, which exhibits a high degree of skewness. This workload is a streaming-join query that does a self-join on an extremely large and high throughput live user-activity log. The presence of a self-join magnifies key skewness.

As in the previous example, we implemented a control operation consisting of a monitoring task and a reconfiguration one, which behaves as described in Fig. 5(a). Fig. 11(d) and Fig. 11(e) show the distribution of the key groups before and after the reconfiguration respectively. Before the reconfiguration, a few tasks handle most of the key space (as indicated by the peaks in the distribution) while after the reconfiguration, the key space is evenly distributed across tasks. This has a direct beneficial consequence on latency and throughput: after the reconfiguration the throughput increases by 26% while latency drops by 61% (Fig. 11(c)).

8. RELATED WORK

Stream processing has been well-studied both in single-node [16, 1, 31] and distributed settings [42, 14, 37, 10, 40, 32, 29, 3]. At the high level, we can divide the approaches underlying these systems into three categories: continuous operator model [31, 16, 29, 10, 17, 37, 28] and BSP model [42, 14, 3]. While there has

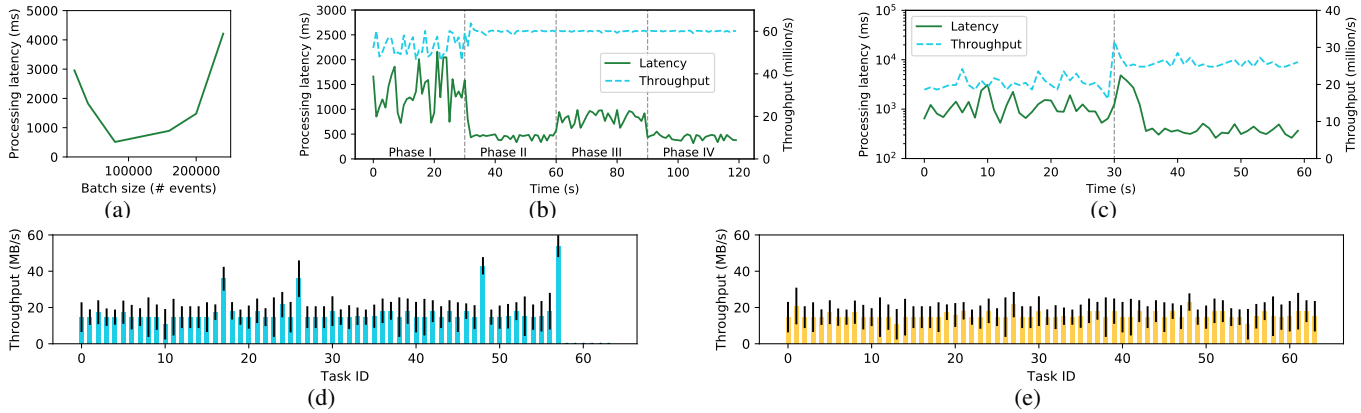


Figure 11: (a) The relationship between latency and batch size, (b) impact of batch size on latency and optimization phases, (c) temporal processing latency/throughput of IPQ2 before and after workload skew adaption, and (d)(e) work distribution across tasks of the streaming-join operator in IPQ2 before and after workload skew adaptation

been significant effort in improving the data plane in both these approaches [16, 6, 5, 11], there has been relatively little work towards improving the control plane [9, 11, 10]. The control plane is still required to deal with common issues faced in distributed system implementation such as fault tolerance, scalability, and adaptivity. In contrast, in Chi, by leveraging the data plane to deliver control messages, control operations can take advantage of the same optimizations introduced for the data plane.

Existing work mostly focused on specific aspects of the control plane in the continuous-operator-model-based streaming systems, e.g., the asynchronous checkpointing algorithms [9]. Apache Flink [10] uses a freeze-the-world approach to change the parallelism of stages through *savepoint* and dataflow restart. Apache Storm [37] provides very limited dataflow re-balance capability — since the system does not provide native stateful operator support, users have to manage and migrate the operator states manually. This makes it hard for users to correctly re-balance a stateful application.

There has been limited amount of work looking at programming the control plane. SEEP [11] proposed an operator API that integrates the implementations of dynamic scaling and failure recovery. The SEEP API, which focuses on managing parallelism, is a subset of what Chi provides in terms of functionality. Chi allows more flexible control operations, e.g., addressing data skews and continuous monitoring. In addition, Chi supports general control operations thanks to a wide spectrum of consistency models. SEEP, instead, provides only a limited set of operations as its control signals are not synchronized. Further, Chi simplifies control plane programming through automating complex tasks. These tasks have to be manually implemented and managed in SEEP. Recently, Dhalion [25] proposed a control policy abstraction that describes the symptoms and solutions of detected anomalies. As discussed in Section §4.4, Dhalion's policies can be implemented on top of Chi.

Punctuations have been widely adopted by streaming systems. They were originally used [38] for partitioning continuous streams in order to cap the states of unbound query operators, such as group-by and join. However, punctuations lack a synchronization mechanism that is critical to support any advanced control operations that require global consistency guarantees. Aurora/Medusa [19] and the successor Borealis [8] are seminal efforts that explore query adaption in distributed stream processing. Borealis uses control lines to modify stream queries but their synchronization needs to be carefully handled by developers, which is a non-trivial task in a distributed setting. Also, control lines are limited to managing query parameters. Esmaili et al. [36] used punctuations to modify continuous queries too. However, punctuations are not synchronized, and thus it can support only control operations applied onto a single input and output stream. Recent streaming systems, such as Flink [10], MillWheel [2] and GigaScope [21], mainly adopt punctuations for limited maintenance tasks such as checkpoint barriers, flushing early results and checking heartbeats. Their punctuation algorithms cannot support general control operations as Chi.

It is natural for BSP streaming systems [39] to reconfigure a dataflow at synchronization barriers. However, the notion of a barrier can be detrimental for streaming workloads due to its negative impact on latency and throughput. Therefore, several techniques have been proposed to mitigate the overhead introduced by synchronization in BSP [42, 40]. For instance, Drizzle [40] studied the scheduling aspect towards a fast adaptable BSP-based streaming system, how to use group-scheduling and pre-scheduling to reduce latency but still provide reasonable adaptivity for sudden environmental changes. Since Chi's primary focus is in embedding control into the data plane, our work is complementary to these efforts.

Offline data systems also have strong demands for reconfiguration during runtime. Chandramouli et al. [15] studied the optimal execution plans for checkpointing and recovering a continuous database query. Recently, S-Store [12] added the streaming semantics on top of a transactional database. Controls can be applied between transactions; however, the system suffers from similar synchronization overhead as in BSP systems.

To support emerging reinforcement learning algorithms, Project Ray [30] developed a task scheduling framework for large dynamic dataflow. In Chi, the task scheduling framework is provided by Orleans, and Chi is responsible for providing the API for customizing control operations and distributed execution mechanisms. A possible future direction is to port Chi onto Ray. This would enhance Ray with the continuous monitoring and reconfiguration capabilities for its artificial intelligence and streaming jobs.

9. CONCLUSION

Chi takes a principled approach to control in data streaming. Rather than using separate control plane channels, Chi propagates control messages along with data message on the data plane. This enables supporting important streaming requirements such as zero system downtime and frequent reconfigurations without compromising on ease of use. Chi's implementation and verification on production workloads not only provide the insights that validate the design choices, but also offer valuable engineering experiences that are key to the success of such a cloud-scale control plane.

10. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *PVLDB*, 12(2):120–139, 2003.

[2] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *PVLDB*, 6(11):1033–1044, 2013.

[3] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8:1792–1803, 2015.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *SIGMOD*, pages 1383–1394. ACM, 2015.

[5] P. Bailis, E. Gan, K. Rong, and S. Suri. MacroBase, A Fast Data Analysis Engine. In *SIGMOD*. ACM, 2017.

[6] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders. Thrill: High-performance algorithmic distributed batch data processing with c++. In *IEEE International Conference on Big Data*, pages 172–183. IEEE, 2016.

[7] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 16. ACM, 2011.

[8] F. J. Cangialosi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[9] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.

[10] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[11] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, pages 725–736. ACM, 2013.

[12] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul, et al. S-store: a streaming newsql system for big velocity applications. *PVLDB*, 7(13):1633–1636, 2014.

[13] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[14] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. *ACM Sigplan Notices*, 45(6):363–375, 2010.

[15] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang. Query suspend and resume. In *SIGMOD*, pages 557–568, New York, NY, USA, 2007. ACM.

[16] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 8(4):401–412, 2014.

[17] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD*, pages 668–668. ACM, 2003.

[18] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[19] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[20] Z. Chothia, J. Liagouris, D. Dimitrova, and T. Roscoe. Online reconstruction of structural information from datacenter logs. In *EuroSys*, pages 344–358. ACM, 2017.

[21] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD*, pages 647–651. ACM, 2003.

[22] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[23] A. Flink. Recovery. *https://goo.gl/AJHiFu*, 2017.

[24] A. Flink. Savepoints. *https://goo.gl/dT4zY2*, 2017.

[25] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron. *PVLDB*, August 2017.

[26] J. Grier. Extending the yahoo! streaming benchmark. *URL http://data-artisans.com/extending-the-yahoo-streaming-benchmark*, 2016.

[27] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core cpus. *PVLDB*, 2(2):1378–1389, 2009.

[28] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *SIGMOD*, 2015.

[29] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. Streamscope: Continuous reliable distributed processing of big data streams. In *NSDI*, pages 439–453, 2016.

[30] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging ai applications. *arXiv preprint arXiv:1712.05889*, 2017.

[31] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[32] NetFlix. Stream-processing with mantis. `https://medium.com/netflix-techblog/stream-processing-with-mantis-78af913f51a6`, 2016.

[33] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: sharing across multiple queries in

mapreduce. *PVLDB*, 3(1-2):494–505, 2010.

[34] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260. ACM, 2000.

[35] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[36] K. Sheykh Esmaili, T. Sanamrad, P. M. Fischer, and N. Tatbul. Changing flights in mid-air: a model for safely modifying continuous queries. In *SIGMOD*, pages 613–624. ACM, 2011.

[37] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. pages 147–156. ACM, 2014.

[38] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng.*, 15(3):555–568, 2003.

[39] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[40] S. Venkataraman, A. Panda, K. Ousterhout, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. *Spark Summit*, 2016.

[41] G. Wang and C.-Y. Chan. Multi-query optimization in mapreduce framework. *PVLDB*, 7(3):145–156, 2013.

[42] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[43] Yahoo! Streaming Benchmarks. `https://github.com/yahoo/streaming-benchmarks`.

APPENDIX

A. CONTINUOUS MONITORING

In Algorithm 2, the monitor operation creates a non-blocking control message that is injected with a CPU metric collection collection action (L1-5). When this message pass through an operator, it first create a summary dictionary that contains metrics per operator (L15-17). It is then augmented it with upstream measurement data (L18-21), and finally with local metrics (L23-25). The local summary is then returned to the runtime and packed into the control payload, and keep propagating until reaching the controller (L26). The controller repeats the similar steps to combine measurement data from all sink operators (L6-11) and finally print the global summary at the screen (L13). In fact, there are many more metrics that can be collected, for example, per-path latency or per-operator workload.

Algorithm 2 Continuous Monitoring Operation

```
 1: function OnInitAtController
 2:     msg := new NonBlockingControlMessage()
 3:     for v in G' do
 4:         msg.confs[v] := new CollectMetric('cpu')
 5:     return msg
 6: function OnBeginAtController(msg)
 7:     $summary := new Dict<Key, Metrics>()
 8: function OnNextAtController(msg)
 9:     for key, metrics in ParseMetrics(msg.payload) do
10:         if key not in $summary.keys then
11:             $summary[key] := metrics
12: function OnCompleteAtController
13:     print $summary
14:     ctx.CompleteOperation(this)
15: function OnBeginAtOperator(msg)
16:     $summary := new Dict<Key, Metrics>()
17:     $action := msg.confs[msg.dest].Single()
18: function OnNextAtOperator(msg)
19:     for key, metrics in ParseMetrics(msg.payload) do
20:         if key not in $summary.keys then
21:             $summary[key] := metrics
22: function OnCompleteAtOperator
23:     if $action.metrics not in ctx.metrics then
24:         ctx.StartMeasure(action.metrics)
25:     $summary[ctx.key] := ctx.CollectMetrics(action.metrics)
26:     return $summary
```

B. CORRECTNESS PROPERTIES

In this section, we provide the proof for Theorem 1 as well as a detailed explanation and proof for safe blocking control operations.

Proof Theorem 1. 1^o Proof of Property 1. Assume G is the topology before the control operation starts, and G^* is the topology after a control operation finishes. Both G and G^* are directed acyclic graphs. Furthermore, given that $E(V,V^*)$ always directed from V to V^*, $G \cup G^* \cup E(V,V^*)$ is a directed acyclic graph.

Termination is guaranteed by the FIFO exactly-once delivery of channels and the in-order generation of control messages. As channels are reliable, messages will eventually be received as long as the operators are alive. Furthermore, an operator in $G \cup G^* \cup E(V,V^*)$ is always reachable from some source operator, or it is a source operator itself.

2^o Proof of Property 2. If (a) $v \to v'$ is an edge in G or G^*, or (b) $v \in T^{-1}(S_{v'})$, then there is an edge $v \to v'$ in the meta topology. v' will invoke *OnCompleteAtOperator* only after it receives control messages from all input channels, including $v \to v'$. And v will only broadcast control messages along $v \to v'$ after the invocation of *OnCompleteAtOperator*. Hence, v will always invoke *OnCompleteAtOperator* before v'. □

In the following, we prove the correctness of safe blocking control operations. We use a similar model as [18] to describe the distributed system that executes a streaming dataflow: The execution process of a dataflow system is a sequence of events. An event $\mathfrak{e}$ is a five-element tuple $\langle v, s_v, s'_v, m_{e_i}, \{m'_{e_o}\}\rangle$, describing an atomic action that an operator v receives a message m_{e_i} from an input channel e_i, updates its internal state from s_v to s'_v, and sends zero or more messages $\{m'_{e_o}\}$ along some output channels $\{e_o\}$ respectively. Note that the sequence of messages received along a channel is an initial sequence of the sequence of messages sent along the channel. Therefore, we do not explicitly model the channel state, as the state of a channel is always the sequence of messages sent along the channel excluding the messages received along the channel. A global state of the dataflow, denoted by $\mathbb{S}$, is the set of all operator and channel states. The occurrence of an event can change the global state. An event $\mathfrak{e}$ can occur in a global state $\mathbb{S}$ if and only if (1) the state of v in $\mathbb{S}$ is s in $\mathfrak{e}$, and (2) the head of the state of e_i in $\mathbb{S}$ is m_{e_i}.

Let $seq = (\mathfrak{e}, 0 \le i < n)$ be a sequence of events, and the global state before e_i be $\mathbb{S}$. We say *seq* is a *computation* of a system if and only if event $\mathfrak{e}_i$ can occur in $\mathbb{S}$.

Next we show that for safe blocking control operations, we can have stronger properties that can facilitate users to understand the

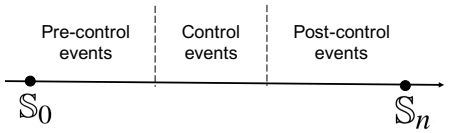


Figure 12: The relation between events and global states.

semantics and correctness of customized control operations. Consider a control operation in a dataflow system that starts at state S_0, takes a computation sequence $seq = (\mathfrak{e}_i, 0 \leq i < n)$ and ends at state $\mathbb{S}_n$. To facilitate our discussion, we group $\{\mathfrak{e}_i\}$ in *seq* into three types: (1) *control events*—the events where an operator when the operator receives control messages, takes control actions and optionally broadcasts control messages, (2) *pre-control events*—the events that occur at an operator before the operator receives any control message along the same channel, and (3) *post-control events*—the events that occur at an operator after the operator receives a control message along the same channel.

Specifically, we want to prove a safe blocking control operation can be permuted to an equivalent computation sequence where all the pre-control events happen before all the control events, which in turn happen before all the post-control events, as shown in Fig. 12. Intuitively, the control events can be treated as if they happened all at the global synchronization barrier as in a freeze-the-world control approach. Formally,

THEOREM 2. *Consider a safe blocking control operation of Chi in a dataflow system that starts at state $\mathbb{S}_0$, takes a computation sequence $seq = (\mathfrak{e}_i, 0 \leq i < n)$ and ends at state $\mathbb{S}_n$. There exists a computation seq' where*

1. *all pre-control events precede all control events, and*
2. *all control events precede all post-control events, and*
3. *all pre-control events precede all post-control events*

PROOF. Assume that there is a control event $\mathfrak{e}_{j-1}$ before a pre-control event $\mathfrak{e}_j$ in *seq*. We shall show that the sequence obtained by interchanging $\mathfrak{e}_{j-1}$ and $\mathfrak{e}_j$ must also be a computation. Let $\mathbb{S}_i$ be the global state immediately before $\mathfrak{e}_i, 0 \leq i < n$ in *seq*.

Case 1^o where $\mathfrak{e}_j$ and $\mathfrak{e}_{j-1}$ occur on different operators. Let v be the operator at which $\mathfrak{e}_{j-1}$ occurs, and let v' be the operator at which $\mathfrak{e}_j$ occurs. First, there cannot be a message sent at $\mathfrak{e}_{j-1}$ which is received at $\mathfrak{e}_j$ because $\mathfrak{e}_{j-1}$ sends out control messages, but $\mathfrak{e}_j$ receives a data message. Second, $\mathfrak{e}_{j-1}$ can occur in global state $\mathbb{S}_{j-1}$. This is because (a) the state of v' is not altered by the occurrence of $\mathfrak{e}_{j-1}$ because $\mathfrak{e}_{j-1}$ is in a different operator; (b) if $\mathfrak{e}_j$ is an event in which v' receives a message m along a channel e, then m must have been the message at the head of e before $\mathfrak{e}_{j-1}$ since $\mathfrak{e}_{j-1}$ and $\mathfrak{e}_j$ receives messages from different channels. Third, $\mathfrak{e}_{j-1}$ can occur after $\mathfrak{e}_j$ since the state of v is not altered by the occurrence of e_j. Because of the above 3 arguments, the sequence obtained by interchanging the $\mathfrak{e}_j$ and $\mathfrak{e}_{j-1}$ is a computation.

Case 2^o where $\mathfrak{e}_j$ and $\mathfrak{e}_{j-1}$ occur on the same operator. First, $\mathfrak{e}_{j-1}$ can occur in global state $\mathbb{S}_{j-1}$. This is because (a) $\mathfrak{e}_{j-1}$ does not alter the operator due to the safe blocking control operation constraint; (b) $\mathfrak{e}_{j-1}$ and $\mathfrak{e}_j$ receives messages from different channels. Second, $\mathfrak{e}_{j-1}$ can occur after $\mathfrak{e}_j$ since $\mathfrak{e}_{j-1}$ does not read the operator state due to the safe blocking control operation constraint. Because of the above 2 arguments, the sequence obtained by interchanging $\mathfrak{e}_j$ and $\mathfrak{e}_{j-1}$ is a computation.

In conclusion, the sequence obtained by interchanging $\mathfrak{e}_{j-1}$ and $\mathfrak{e}_j$ must also be a computation.

Similarly, one can show that if there is a post-control event $\mathfrak{e}_{j-1}$ before a pre-control/control event $\mathfrak{e}_j$, the sequence obtained by interchanging $\mathfrak{e}_{j-1}$ and $\mathfrak{e}_j$ must also be a computation. The only difference is that $\mathfrak{e}_j$ and $\mathfrak{e}_{j-1}$ cannot occur on the same operator due to the fact that an operator can start processing post-control messages only after it has received control messages from all input channels. By repeatedly applying the above interchanges, one can easily see that we can generate a computation seq' that satisfies the 3 requirements as listed in Theorem 2. □

In the end, we show how Theorem 1 and Theorem 2 can help Chi users prove the correctness and semantics of their control operations. Take Algorithm 1 as an example. This algorithm is a safe blocking control operation. According to Theorem 1, Algorithm 1 will always terminate in finite time. According to Theorem 2, Algorithm 1 is equivalent to the freeze-the-world control approach: (1) suspends the dataflow, (2) finishes processing all the messages on the fly, (3) starts the control operation and waits for its completion, and (4) resumes the dataflow. It is easy to see the correctness of the freeze-the-world approach, and thus the correctness of Algorithm 1 follows.