Trisk: Task-Centric Data Stream Reconfiguration

Yancan Mao
National University of Singapore
maoyancan@u.nus.edu

Yuan Huang
National University of Singapore
dcsyhg@nus.edu.sg

Runxin Tian
National University of Singapore
tianrunxin@u.nus.edu

Xin Wang
National University of Singapore
dcswan@nus.edu.sg

Richard T. B. Ma
National University of Singapore
tbma@comp.nus.edu.sg

ABSTRACT

Due to the long-run and unpredictable nature of stream processing, any statically configuredexecution of stream jobs fails to process data in a timely and efficient manner. To achieve performance requirements, stream jobs need to be reconfigured dynamically. In this paper, we present Trisk, a control plane that support versatile *reconfigurations* while keeping high efficiency with easy-to-use programming APIs. Trisk enables versatile reconfigurations with usability based on a task-centric abstraction, and encapsulates primitive operations such that reconfigurations can be described by compositing the primitive operations on the abstraction. Trisk adopts a partial pause-and-resume design for efficiency, through which synchronization mechanisms in the native stream systems can further be leveraged. We implement Trisk on Apache Flink and demonstrate its usage and performance under realistic application scenarios. We show that Trisk executes reconfigurations with shorter completion time and comparable latency compared to a state-of-the-art fluid mechanism for state management.

ACM Reference Format:
Yancan Mao, Yuan Huang, Runxin Tian, Xin Wang, and Richard T. B. Ma. 2021. Trisk: Task-Centric Data Stream Reconfiguration. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3472883.3487010

1 INTRODUCTION

With the development of Internet-scale services, data is generated in high volume, velocity and variety. Applications with time constraints are increasingly implemented in the form of stream processing, where the arrived data is processed immediately with low latency and high throughput. Today, many distributed stream systems, e.g., Samza [32], Flink [9], Heron [24], Storm [41] and Spark Streaming [44], have been developed to parallelize, deploy and manage stream jobs for users.

As data stream is by nature fluctuating with dynamic rates and distribution over time, to satisfy low latency requirements, stream jobs must process data timely [6, 8]. This requires stream systems to be able to *reconfigure* part of the dataflow computation dynamically during execution without affecting the correctness of processing logic. We define such actions as *reconfigurations* on stream jobs. In practice, reconfigurations are often applied by a *control policy* to achieve certain performance goals. Based on prior literature [10], we summarize that a good stream system should enable reconfigurations with three desirable properties: versatility, efficiency, and usability.

Versatility. A stream system should support a wide variety of reconfigurations, such that various control polices that require different types of reconfigurations can be implemented. Common reconfigurations mainly include operations along three dimensions, i.e., *resources*, *workloads*, and *execution logic*. The *resources* and *workloads* often need to be re-assigned to handle data skewness and changes of input rates, while the *execution logic* needs to be updated to fix bugs and handle emerging events [7, 9].

Efficiency. Reconfigurations should be executed and completed in short time, having minimum impact on the original stream job execution. Stream jobs are physically executed by a set of parallel tasks, to guarantee the correctness of job execution during reconfigurations, synchronization is required among those parallel tasks, which blocks the system temporarily. Thus, it is important to execute reconfigurations efficiently to minimize the unavoidable unavailablity time during reconfigurations.

Usability. A stream system should also provide intuitive and easy-to-use APIs for users to implement their control policies, ideally without assuming that users understand the details of reconfiguration execution.

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11.
https://doi.org/10.1145/3472883.3487010

Although existing works provide some of the desirable properties, they are unable to achieve all. Due to the use of a kill-and-restart method to execute reconfigurations, Flink [9], Samza [32], and Heron [24] enable reconfigurations at a high cost of efficiency. Research prototypes such as Megaphone [19] and Rhino [29] proposed efficient state management primitives with high usability, but lack of the support for other types of reconfigurations such as *change of logic* to update execution logic. Chi [28] used a control message based programming model to support various control logic, but was not mainly designed for reconfigurations. As specific system-level operations need to be specified to implement a reconfiguration, Chi was targeted for advanced users that manage system internals.

In this paper, we present Trisk[1], a control plane solution that supports reconfigurations of stream jobs with all three properties. The core of Trisk is a task-centric abstraction that describes the *execution plan* of the target stream job. The *execution plan* of a stream job maintains the configurations of its physical tasks and is used to deploy the job on a cluster. Since any reconfiguration boils down to change the existing execution plan to a new one, it can be formally described by the operations applied on the current execution plan. To provide usability, we classify the operations into three types of primitive operations, so that various reconfigurations can be implemented by applying a combination of primitive operations on the Trisk abstraction. To execute reconfiguration efficiently with low system overhead, we adopt a partial pause-and-resume mechanism by leveraging synchronization mechanisms in the native stream, where only part of the stream job will be paused and updated. We implement Trisk on top of Apache Flink by leveraging the checkpoint mechanism to achieve synchronization, and show that Trisk achieves sub-second completion time to execute reconfigurations. In summary, we make the following contributions:

- We propose a control plane solution, Trisk, that maintains a task-centric abstraction with three-dimensional primitive operations to implement versatile reconfigurations with high usability.
- We design and implement a prepare-sync-resume pipeline to execute reconfigurations by leveraging synchronization mechanisms in the native stream.
- We integrate Trisk with Flink and leverage the checkpoint mechanism in Flink to execute reconfigurations.
- We evaluate Trisk via comprehensive experiments using both real-world applications and synthetic micro-benchmark. We also compare Trisk with native Flink on the performance of supporting control policies and executing reconfigurations.

[1]The source codes are available at: https://github.com/sane-lab/Trisk

2 BACKGROUND AND MOTIVATION

In this section, we first introduce the terminologies used in this paper, and then motivate the necessity of supporting reconfigurations with the three proposed properties.

A distributed stream job runs as a physical deployment of an *execution plan* which instantiates *operators* to physical parallel *tasks*. An *execution plan* describes the configurations of a stream job, and can be represented as a directed graph, where vertices in the graph represent *tasks* instantiated from *operators*, and edges represent the data flow between *tasks*. Specifically, *operators* maintain the user-defined *execution logic* to process the input data, and *tasks* that are instantiated from the same operator share the same *execution logic*. The input data of an operator forms the *workloads* to be processed by tasks in parallel. The *workloads* of an operator are commonly grouped by keys and partitioned across *tasks*. Each *task* is allocated with certain *resources* such as CPU cores and memory on a node in cluster for physical execution.

To achieve performance requirements, users often apply control policy on stream jobs. A control policy involves two steps. First, it monitors the stream job and decides whether or not to update the current execution plan based on the symptoms detected, e.g., backpressure in the pipeline. Second, the control policy needs to identify the performance bottleneck in stream jobs and invoke reconfigurations to optimize it accordingly. Different control policies make decisions based on different kinds of metrics [14, 15, 22] in both system level, e.g. CPU utilization, and application level, e.g. observed arrival rate and backpressure. In this work, we focus on the execution of reconfigurations given the decisions of control policies, while metrics retrieval mechanisms are regarded as a part of the control logic.

Reconfigurations need to dynamically change the physical execution plan of a stream job, which boils down to reconfigure its resources, workloads, and execution logic. Such a variety of reconfigurations are required by control policies to achieve different performance goals. For example, to achieve a SLO/SLA objective for general stream jobs, prior works such as Henge [23], Dhalion [14], DS2 [22], and DRS [15] introduce control policies based on *scaling* to reallocate resources for stream jobs. To achieve balanced load and better resource utilization, prior works such as DKG [40] propose control policies to detect data skewness and apply *load balancing* to manage the workloads of stream jobs. Furthermore, for machine learning based stream jobs such as online anomaly detection [18], because new scenarios and input data are emerging over time, the model with current parameters may fail to process them accurately and effectively. To solve this problem, the model needs to be updated appropriately, where *change of logic* can be applied to achieve dynamic model tuning.

Table 1: Overview of existing work enables reconfigurations in stream systems.

	Methodology	Versatility	Usability	Efficiency
Flink [9]	Dataflow model + Redeploy	Medium	High	Low
Heron [24]	Dataflow model + Redeploy	Medium	High	Low
Seep [12]	State management primitives + Partial redeploy	Low	High	Medium
Rhino [29]	State management primitives + Partial update	Low	High	High
Megaphone [19]	State management primitives + Non-stop partial update	Medium	High	High
Chi [28]	Message-based programming model + Partial update	High	Medium	High
Trisk	Three-dimensional task-centric abstraction + Partial update	High	High	High

Although reconfiguration is best supported with three properties [10]: *versatility*, *efficiency*, and *usability*, existing systems and research fall short in achieving all of them. We summarized existing works that support reconfigurations for stream jobs in Table 1, and classify them into three types of implementations.

Built-in reconfiguration leverages the original dataflow model and programming interfaces provided by stream systems to enable reconfigurations. For example, Flink [9] and Heron [24] redeploy the stream job with updated context for all tasks, i.e. restarting the job with modified source code and configuration files. Although reconfigurations can be easily invoked through the original programming interfaces provided by the stream systems, they are executed in low efficiency and incur high system overhead and performance degradation due to the nature of kill-and-restart.

Reconfiguration for state management has been designed in prior works such as SEEP [12], Rhino [29] and Megaphone [19]. These works proposed state management primitives that provide interfaces to manage the state of stream jobs efficiently. Stateful stream jobs maintain state to process each of the assigned keys, which is regarded as a workload-related configuration in our context. With the provided interfaces, reconfigurations that cover workloads redistribution for stateful jobs can be implemented with high usability and efficiency. However, such primitives are limited to state management and do not support other types of reconfigurations such as placement and change of logic.

Reconfiguration via a control plane encapsulates mechanisms for applying various control logics on stream jobs, which supports a variety of reconfigurations. Chi [28] proposes a programming model based on control message injection, through which new reconfigurations can be implemented by applying fine-grained instructions on each task and embed them into control messages. Tasks are updated asynchronously upon receiving the instructions in the control messages. However, since the task update logic is defined by users, they need to be familiar with the execution details of the stream system and implement instructions accordingly, which requires non-trivial engineering efforts.

Targeted for achieving all three desirable properties for stream reconfigurations, Trisk is designed as a control plane solution applicable to general stream systems, and encapsulates mechanisms for general control policies. To achieve versatility, Trisk uses a task-centric abstraction, which describes the configurations of each task in three dimensions i.e., resources, workloads and execution logic. The Trisk abstraction is designed around tasks, as the states of tasks describe configurations at the minimum granularity, i.e., reconfigurations can be achieved by updating a subset of tasks. For example, load balancing redistributes workloads among tasks, scaling cancels or deploys tasks, placement redeploys tasks on other nodes, and change of logic updates the execution logic of tasks. Based on the abstraction, Trisk implements three primitive operations (Section 3.1) on updating tasks along the three dimensions and encapsulates them as a set of APIs. For usability, Trisk provides common reconfigurations (Section 3.3) for users to implement control policies easily; while any general reconfigurations can be implemented by compositing primitive operations (Section 4.2). Trisk uses a prepare-sync-update execution pipeline to execute reconfigurations efficiently (Section 3.2), under which tasks are partially paused and updated asynchronously. This enables Trisk to leverage the synchronization mechanisms in native stream systems with low system overhead.

3 DESIGN

We focus on the problem of reconfiguring stream jobs on-the-fly, and our goal is to design a control plane that enables versatile reconfigurations while maintaining usability and efficiency. In this section, we first introduce the design of the Trisk abstraction, and then describe the mechanisms that enable asynchronous execution of the reconfiguration. Last, we present the reconfiguration APIs and show how users can implement control policies by using them.

3.1 The Trisk Abstraction

The Trisk abstraction maintains an abstract execution plan that is independent of stream systems for extensibility. This

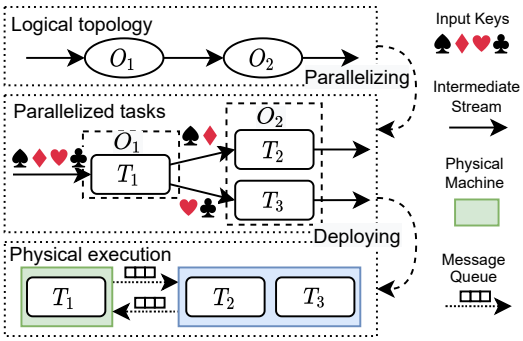


Figure 1: Deployment steps of jobs in stream systems

is achieved by specifying the execution plan in terms of the configurations with respect to individual tasks, which can be classified along three dimensions: execution logic, workloads, and resources. In other words, any reconfiguration consists of mainly three types of operations. The intuition behind the three-dimensional Trisk abstraction is derived from the three-step deployment of stream jobs shown in Figure 1. This stream job has two operators (O_1, O_2), which are instantiated as three tasks (T_1, T_2, T_3) physically deployed across two machines. The keyspace of the data stream contains four unique keys and is partitioned into two substreams.

In the first step, a stream job is defined by its logical topology described as a DAG, where vertices represent operators and edges represent the intermediate data streams. At this stage, the execution logic is configured and associated with each operator, implying that all instances of parallel tasks of the operator will use the same execution logic to process the assigned input streams so as to generate outputs.

In the second step, the stream job specifies the number of parallel tasks to be instantiated for each operator. Input data is often defined over a key space, and each task will be assigned with a partition of an non-overlapping subset of keys for independent data processing. The configuration is maintained by both upstream and downstream tasks. In particular, the upstream tasks maintain the routing information, which maps their processing results to downstream tasks. Each downstream task keeps a subset of input keys representing the subset of substreams to be processed and the corresponding states to be managed. At this stage, the configuration of workloads needs to be specified for the individual tasks.

In the final step, the stream job deploys tasks on physical machines. In particular, each task is assigned to a resource slot configured with resources that determine its performance. For example, computational resource such as CPU cores affects the processing rate and memory resource is used to store on-going processing states and affects the speed of I/O operations. Furthermore, data streams between an upstream and a downstream tasks go through networks if both tasks are deployed in different physical machines.

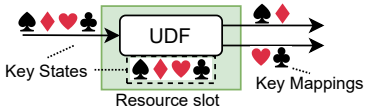


Figure 2: Configurations of tasks in Trisk abstraction.

Figure 2 illustrates the four configurations associated with each task specified in the Trisk abstraction, i.e., *Key State*, *User-Defined Function* (UDF), *Key Mapping*, and *Resource Slot*.

- Along the execution logic dimension, *User-Defined Function* (UDF) defines the processing logic on each input tuple that it received. After processing, the results from the UDF form the output streams. For stateful tasks, UDF has access to its processing state, which is generated according to the processing history of arrived data, and will update the state after new tuples being processed.
- Along the workloads dimension, the distribution of workloads among the tasks of an operator is described by the *Key State* distributed across the tasks and the *Key Mapping* in the upstream tasks. *Key State* represents the assigned subset of input keys to be processed and the associated processing state to be maintained. *Key Mapping* defines how a task maps the keys of output results to downstream tasks. The *Key Mapping* in the upstream tasks also represents the global *Key State* assignment of downstream tasks, i.e., the combination of *Key State* of all downstream tasks.
- Along the resources dimension, *Resource Slot* denotes the amount of resources allocated to a task, e.g., CPU cores and memory obtained from the resource management system; it also describes the location of task to be deployed, which is important for communication efficiency and avoiding resource contention.

Our task-centric abstraction is general for providing the versatility of reconfigurations, because any reconfiguration boils down to updating the three types of task configurations, originally executed by the initial deployment of stream jobs. Besides the chosen configurations, the Trisk abstraction can be easily extended, since all configurations are generated during the initial deployment. For example, the batch size in mini-batch processing can be classified as a type of execution logic configuration to define how input tuples are batched. Based on the dimensions of execution logic, workloads and resources, Trisk implements common reconfigurations of change of logic, load balancing and placement, respectively. Furthermore, by using operations along the dimensions of workloads and resources, Trisk also implements scaling.

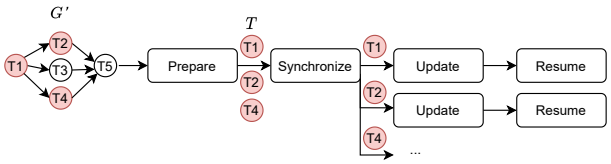


Figure 3: An example of reconfiguration execution steps with 3 affected tasks to be updated: T1, T2, T4.

3.2 Reconfiguration Execution

To realize a reconfiguration, a new execution plan is generated based on which a sequence of operations will be applied on tasks. There are two main requirements for this reconfiguration execution. On the one hand, the execution is required to be completed in short time with low system overhead, so as to avoid affecting the original stream processing. On the other hand, the execution also needs to keep the consistency and correctness of the stream processing. Because reconfiguration execution is task-centric and each of the parallel task instances can be modified independently, it can result in inconsistent processing if tasks are not synchronized before being updated. The goal of task synchronization is to make all parallel tasks to be paused at the same logical time to avoid data loss or data duplication during reconfiguration execution.

With the consideration of efficiency and system overhead, Trisk adopts a partial pause-and-resume scheme and is able to leverage the mechanisms in native stream systems for tasks synchronization. As the native stream systems often have their own synchronization mechanisms to guarantee the consistency and fault tolerance of data processing, we leverage such mechanisms to pause all affected tasks during reconfigurations. In this way, Trisk has minimum performance impact on the native systems, while keeping the consistency and fault tolerance properties provided by the stream system. After the affected tasks are paused during the synchronization, they can be updated and resumed asynchronously. For task reconfiguration, instead of redeploying the task by killing the current instance and creating a new one, Trisk updates the corresponding components in the task instance, reducing the overhead of modifying the configurations.

Trisk embeds a coordinator (Section 4.1) in the native stream system to coordinate the execution of reconfigurations. Upon receiving a remote call with an updated Trisk abstraction, the coordinator instructs the stream system to execute reconfigurations in the following three steps illustrated in Figure 3. 1) Prepare. The coordinator finds the affected tasks T by comparing the current execution plan G with the new generated execution plan G', and prepares the system-specific configurations based on the task configurations in the Trisk abstraction. 2) Synchronize. The coordinator starts to synchronize the affected tasks T, which are to be paused and send back acknowledgements to the coordinator. The pause operation does not include any unaffected tasks, for the downstream tasks that consume output from affected tasks, they may wait for new data when the intermediate buffer becomes empty. 3) Update. Once all the affected tasks are paused, i.e., the coordinator receives all acknowledgements from them, each affected task T_i updates with its new configurations independently and can be resumed immediately once the update is completed.

Trisk is more efficient than the approaches taken by existing stream systems, which try to kill and restart the entire stream job. Although recent research prototypes [19, 28, 29] also adopted partial pause-and-resume approaches to execute reconfigurations, their designs are bound to specific synchronization mechanisms, and therefore, are difficult to be integrated with existing stream systems. In particular, an additional synchronization mechanism introduces system overhead and compatibility issues with existing mechanisms, e.g., for achieving fault tolerance. Comparatively, Trisk has a more modularized design that encapsulates the steps of prepare, synchronization and update separately. This design enables easy integration with stream systems by leveraging their native synchronization mechanism. For example, for integrating with Flink, we leverage its asynchronous checkpoint mechanism [9] to synchronize tasks, which is achieved by applying aligned barrier-passing on the entire pipeline. This design is also compatible with common Zookeeper-based coordination [32] used by stream systems such as Samza.

While Trisk maintains fault tolerance of stream systems with low system overhead, it can be slowed down due to the excessive delay to execute synchronization in native stream systems. For example, checkpoints in Flink may take longer to complete when the pipeline is backlogged, which increases the completion time of reconfigurations in Trisk (Section 5.3). We notice that the Flink community has recently proposed the unaligned-checkpoint [13] to reduce barrier-passing latency, which potentially improves the performance of Trisk on Flink, and we leave this as our future work.

3.3 Reconfiguration API

We next describe our reconfiguration APIs that enable users to enforce control policies for their stream jobs. To define a control policy, users implement a controller, a Trisk runtime component, that wraps control policies. When a new stream job starts, the Trisk runtime will be launched alongside and execute the control policy embedded in the user-defined controller automatically.

Controller Instantiation. Trisk supports two methods for controller submission. First, users can implement the

controller and configure it in Trisk directly, and the controller will be instantiated and started by Trisk runtime once the stream job is running. Second, Trisk also exposes restful APIs for submitting the source codes of controllers, and the Trisk runtime will dynamically compile and instantiate controller instances accordingly. Through our restful APIs, Trisk accepts new control policies in the runtime, through which users can update their control policies at any time after the stream job is deployed and running.

Supported Reconfigurations. There are four high-level reconfigurations APIs, i.e., `loadBalancing(opId,dist)`, `changeOfLogic(opId,func)`, `placement(opId,deploy)` and `scaling(opId,dist,deploy)`, through which users can implement control policies using these supported reconfigurations without worrying about the system-level implementation details. The first three APIs are implemented by three primitive low-level operations along the three-dimensional configuration space to enable load balancing, change of logic and placement, respectively. As a commonly used reconfiguration in controllers [12, 14, 15, 22], scaling is implemented via compositing the primitive operations. Besides the supported reconfigurations, users can also implement customized reconfigurations (Section 4.2) in the controller by leveraging the lower-level primitive operations on Trisk abstraction, similar to how the scaling API is implemented in Trisk.

```
1  // 1. Extend a Controller from AbstractController
2  class LoadBalancer extends AbstractController {
3    // 2. Override to define a new control policy
4    protected void defineControlAction() {
5      // The user-defined load balancer monitors
       data skewness among tasks every second.
6      // It proposes a new workloads distribution if
        load imbalanced.
7      while(true) {
8        Map<taskID, List<Key>>
9          workloads = detectSymptoms();
10       if (workloads != null) {
11         // 3. Invoke reconfiguration API
12         loadBalancing(operatorId, workloads);
13       }
14       sleep(1000);
15     }
16   }
17 }
```

Listing 1: Example of load balancing controller.

An Example. We illustrate an implementation of a controller whose control policy detects load imbalance among tasks every second and balances the workload when the load is skewed. It involves three steps shown in Listing 1. First, users need to extend a new controller from `AbstractController`, which is the class exposed by Trisk to provides APIs of supported and customized reconfigurations. Second, users can override their own control policy in `defineControlAction()` method. In particular, control policies need to specify when and what reconfigurations to be executed. In this example, the load balancer can detect symptoms based on the statistics of input keys distribution among tasks, and decide to apply load balancing if the key distribution among tasks are skewed. Third, users invoke the provided reconfiguration APIs in the `AbstractController` accordingly. When using these APIs, users need to specify the new configurations to be updated as input parameters.

4 IMPLEMENTATION

The implementation of Trisk consists of around 10,000 lines of code in Java. In this section, we describe the system architecture and illustrate how it supports the design of Trisk. We will also discuss the detailed decisions we made on Trisk's implementation.

4.1 Implementation Architecture

The Trisk architecture consists of two parts: the Trisk runtime and the corresponding instrumentation in a stream system. Trisk leverages the Netty framework to remote communicate with the stream system. The Trisk runtime runs as a standalone process and is the entrypoint that provides the Trisk abstraction and reconfiguration APIs for users. The instrumentation in stream system performs efficient execution of reconfigurations requested from the Trisk runtime. Figure 4 shows the architecture of Trisk with four main entities.

- **Controller** is defined by users to specify their control policies and invoke the supported reconfiguration APIs or implement customized reconfigurations with APIs of primitive operations.
- **StreamManager** runs as a backend runtime, which maintains a web service to receive new controller submission and maintains the Trisk abstraction. It also instructs stream systems to execute the reconfigurations in prepare-sync-update manner.
- **JobReconfigCoordinator** is maintained in the job master of a stream job. It maps the configurations from Trisk abstraction to the configurations in stream systems, and can leverage the synchronization mechanisms in native stream systems for reconfigurations.
- **TaskConfigManager** is maintained by each task, which manages the configurations of the task. It receives remote instructions from ReconfigCoordinator and update task configurations accordingly.

A reconfiguration is executed in the following three steps. ① Controller gets the Trisk abstraction from StreamManager, and invoke primitive APIs to update the abstraction. ② Once StreamManager receives the updated

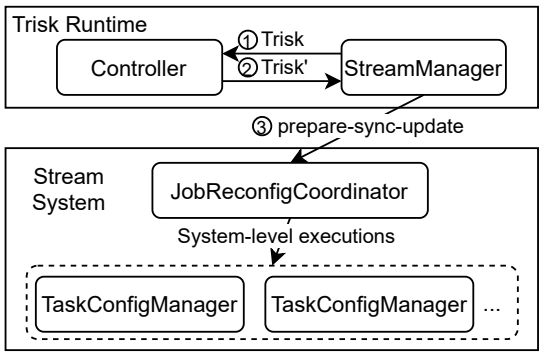


Figure 4: Trisk system architecture

Trisk abstraction, it parses the recorded , primitive operations applied inside and constructs instructions to be sent. ③ StreamManager instructs ReconfigCoordinator to execute reconfigurations through the prepare-sync-update pipeline.

4.2 Reconfiguration Implementation

In Trisk, reconfigurations are described in terms of the Trisk abstraction and executed via encapsulated primitive operations. In particular, the prepare-sync-update steps of reconfiguration instructions are constructed based on the difference between an update Trisk abstraction and the existing one. Consequently, users can also construct new reconfigurations by modifying the Trisk abstraction using the provided primitive operations. Such a design of encapsulation hides the details of the reconfiguration execution from users. As any reconfiguration may involve operations along the three dimensions, Trisk exposes three primitive APIs, i.e., `assignLogic(opId,func)`, `assignResource(opId,dist)` and `assignWorkload(opId,deploy)`. To be user-friendly, each API takes an operator ID and the updated configurations of tasks under the operator as input parameters, through which Trisk identifies the affected tasks by checking whether the configurations of tasks under the operator are changed. To prepare new configurations, users can query the current configurations in the Trisk abstraction and modify accordingly. For the specific configurations in each API, `dist` refers to the workloads distribution of the tasks in the associated operator, which is a map of tasks to key states; `func` refers to a new UDF, where the input and output keys have to remain the same with the original one; `deploy` refers to the deployment of tasks in the associated operator, which is represented as a map of tasks to the newly allocated resource slot. When applying a primitive operation on the Trisk abstraction, a new Trisk abstraction is to be generated, and the primitive operation will be eventually executed during the update phase of reconfiguration execution.

The `assignLogic(opId,func)` API accepts a user-defined function (UDF) object representing execution logic that will replace the original UDF object in the task. The function object is provided by the user-defined controller at runtime, and Trisk provides three methods for a controller to create a new function object: 1) Instantiate the function object from the same function class of the stream job with different initialization parameters. 2) Instantiate the object from another existing function class. 3) Instantiate from a new function class that does not exist previously.

Upon receiving model codes, `Controller` needs to instantiate function object accordingly; however, the typical instantiation method `new()` cannot be used, since the application context is not known to `Controller` when it tries to compile the newly received source codes. Fortunately, Java has a *reflection* mechanism that allows an executing program to examine or "introspect" upon itself, and manipulate internal properties of the program. In Trisk, `Controller` uses Java reflection to obtain the properties of classes as they are dynamically loaded, so as to achieve the aforementioned three methods. To create an object from a new class, the `Controller` also needs to compile submitted source codes and load it into the JVM. In this way, we generalize the `assignLogic(opId,func)` operation to only concern the function object while the object itself could be created in many ways decided by controllers.

Besides the four supported reconfigurations, Trisk also enable users to implement customized reconfigurations. This can be done by chaining the primitive operations with desirable parameters in a specific order. We demonstrate how Trisk implements the supported scaling reconfiguration in such a way in the following Listing 2.

```
class Controller extends AbstractController {
  private void scaling(operatorID operatorId,
    Map<taskID, Node> resources,
    Map<taskID, List<Key>> workloads) {
      // get a copy of the Trisk abstraction
      Trisk trisk = getTrisk();
      // update Trisk abstraction
      trisk
        .assignResource(operatorId, resources)
        .assignWorkload(operatorId, workloads);
      // execute the new abstraction
      execute(trisk);
    }
}
```

Listing 2: Code sketch of the *scaling* reconfiguration.

4.3 Integration with Apache Flink

To integrate with Flink, we map the Trisk abstraction to the configurations in Flink's `JobGraph` and `ExecutionGraph`. The `JobGraph` maintains each operator configuration in a `JobVertex`, which maintains common configurations in all tasks under the operator such as the UDF. The

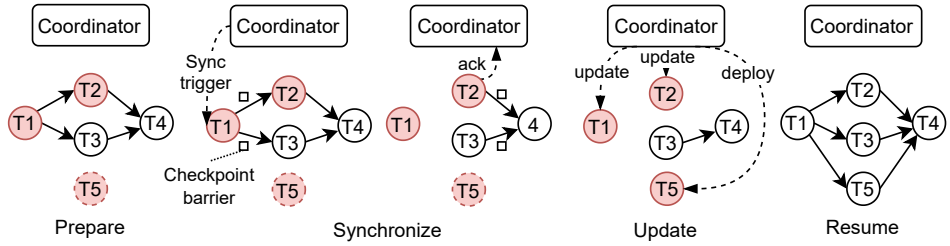


Figure 5: An illustration of scaling out and migrates workloads from the task 2 to the new task 5.

`ExecutionGraph` maintains the configurations for each task in a `ExecutionVertex`, where task-specific configurations are maintained, e.g., allocated resources and workloads.

We utilize the asynchronous checkpoint mechanism [8] to achieve synchronization, where tasks can be paused on receiving all *stream barriers* from upstream tasks and start to update independently. For the update in each individual task, we leverage the native methods in Flink to instantiate components for the corresponding configuration, and update the components by following the same way to create new versions of components and substitute the old components.

We use an example of scaling out to show the procedure of an end-to-end reconfiguration in Trisk on Flink, illustrated in Figure 5. The coordinator, i.e., `JobConfigCoordinator`, is responsible for synchronizing the entire pipeline and instructing the affected tasks to update their configurations accordingly. In this scale-out example, a new execution plan has been proposed, where a new task T_5 is to be created, the workloads are to be redistributed between the existing task T_2 and the new task T_5, and the upstream task T_1 needs to update the key mapping accordingly. The scaling is executed in three phases. In the prepare phase, the system-specific task configurations are to be generated. In the synchronize phase, T_1 in the upstream and T_2 in the middle stage are to be paused, which is executed in two steps. Step 1, the coordinator triggers a synchronization by starting a checkpoint procedure, which injects barriers, which are to pass through the entire pipeline. Step 2, once an affected task receives all the barriers from its upstream tasks, it will 1) broadcast the barrier, 2) pause the current execution, and 3) acknowledge to the coordinator. In the update phase, when the coordinator receives all acks from affected tasks, it will inform the affected tasks to update with the given new configurations asynchronously. The reconfiguration completes when all tasks get updated and resumed.

5 EVALUATION

We have implemented Trisk and integrated it with Apache Flink. To evaluate the performance of Trisk, we have conducted experiments to answer the following questions.

Q1. How can Trisk controllers leverage reconfigurations to help optimize stream processing in real-world? Does it achieve three aforementioned properties on supporting reconfigurations?

Q2. What are the impacts of workload characteristics on the reconfiguration execution in Trisk?

Q3. How about the efficiency of Trisk reconfiguration compared to existing work?

We present our experimental results as follows. In Section 5.2, we design three Trisk controllers on two real-world stream jobs to handle different scenarios, and compare the performance of a Trisk controller with that implemented in Flink to answer **Q1**. In Section 5.3, we evaluate the impact of the varying characteristics of workload using a benchmark application to answer **Q2**. In Section 5.4, we compare the performance of Trisk reconfigurations with that of vanilla Flink and Megaphone [19] to answer **Q3**.

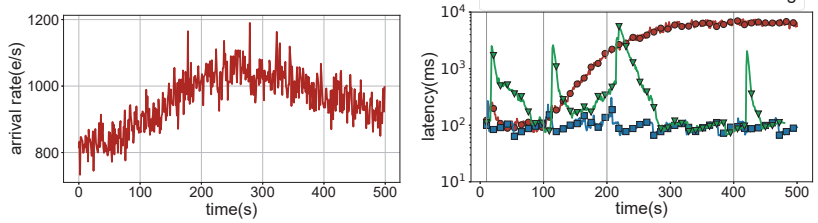
5.1 Experimental Setup

Experiment Environment. We run experiments on a cluster of 4 machines. Each has an Intel Xeon Silver 4216 @2.10GHz processor with 16 cores and 64GB of RAM, running Ubuntu 18.04.4 LTS. All machines are in the same rack, connected by a high-speed switch. The bandwidth of each machine is 1 Gbps. We run Trisk on Flink-1.10.0 configured with 4 TaskManagers, and each with 8 slots for all experiments. To stress test the performance under reconfigurations, we disable the flow control mechanism in Flink to exclude the impact of backpressure.

Workload. We use three applications to evaluate Trisk.

1. *Stock trading transaction processing*. We use a real-world dataset extracted from a major stock exchange in Asia. The dataset contains 500 seconds of quotes created by sellers and buyers. We implemented a stock exchange application that matches the quotes of each stock from both sides and outputs the transaction results continuously. It also accumulates a large amount of pending quotes as its state.

2. *Fraud detection application*. We use a credit card transactions generator mentioned in a simulated credit card transaction dataset published in Kaggle [21]. We use this generator to generate a dataset contains 2,188,073



(a) Arrival curve for stock exchange (b) Latency comparison of Trisk and Flink.

Figure 6: Dynamic scaling and load balancing on stock trading.

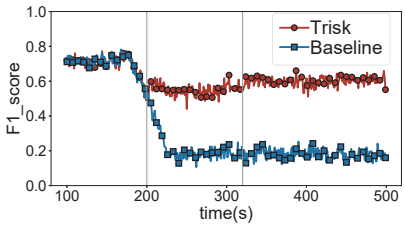


Figure 7: Dynamic model updates for fraud detection.

transactions in 10 minutes which keeps a fraud rate of 7% over the entire data. To simulate emerging data with time, the generation parameters of the data changes with time. We implemented a fraud detection application by adopting a rule-based decision tree model.

3. *Word count.* We use a representative stateful application, i.e., word count, as a micro-benchmark to evaluate the behavior of Trisk under different workload characteristics in detail. The generator of the words is configurable that we can set different sizes of words and arrival rates to control the state size and resource utilization.

Performance Metrics. We evaluate Trisk's performance with respect to controller and reconfiguration execution.

1. *Controller.* We evaluate the performance of an end-to-end controller implementation by using different metrics. Since our controllers are designed for different user requirements, we mainly measures the metrics related to the specified requirements. For the latency-aware controller whose goal is to minimize the processing latency by scaling and load balancing, we mainly measure the latency of the stream job over time to evaluate whether the latency is minimized and stable. For the service-quality-aware controllers that aims to do change of logic to process input data more appropriately, we measure the application-level metrics. In our experiment, fraud detection is implemented by the decision tree model, where we mainly use the F1 score [17] to evaluate the accuracy of the model. For placement controller, which is designed to re-assign resources to reduce resource contention, we measure the CPU utilization of each machine and the application level latency to evaluate the effectiveness of placement.

2. *Reconfiguration execution.* We evaluate the performance of reconfiguration execution by the completion time and the end-to-end processing latency during the execution. The completion time is defined as the time passed from making the decision to all affected tasks updated and resumed successfully. The end-to-end latency during reconfiguration shows the impact of reconfigurations on the original stream processing.

5.2 Trisk in Action

In this subsection, we implement three controllers for stock trading and fraud detection applications to show the effectiveness of Trisk, i.e., versatile and efficient reconfigurations with usability. Each controller applies different types of reconfigurations for different optimization purposes. We also implemented a controller on the native Flink for efficiency comparison. We enabled native reconfigurations in Flink by using the savepoint mechanism to do a global snapshot, and re-submit a stream job with new configurations.

Through experiments we demonstrate that: 1) Trisk supports versatile reconfigurations that can be applied to optimize stream processing in various real-world scenarios, 2) Trisk performs reconfigurations better than native Flink in terms of supporting controllers with higher efficiency and achieving user requirements such as low latency stream processing, and 3) Trisk controller can be written in around a hundred lines of code, making it easy to use.

Latency-aware controller. We design a latency-aware controller by using load balancing and scaling to adjust the workloads and resources allocated to tasks, to achieve low-latency stream processing. We set the initial configuration of the stock exchange application as follows: 1) Parallelism of transaction processing operator is 10, the parallelism of source and sink operator has been set to 1. 2) The key distribution follows the default key mapping strategy in Flink, which assigns a consecutive equisized set of keygroups to each task. The processing rate of each task is upper-bound by 100 transactions per second. The arrival curve of the dataset is shown in figure 6a, the data stream has shown a fluctuating arrival pattern, the arrival rate is increasing from 0s to 300s and decreasing from 300s to 500s. The keys in the stock stream shows a long tail pattern, and the key distribution is highly skewed, which has a skewness of -5.23 calculated from Pandas skew method [34].

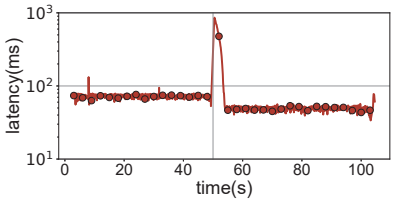
To handle the characteristics of stock transcation streams, we implement a latency-aware controller according to the arrival curve. The controller is written in 78 lines of code,

in which 30 lines of code are to construct parameters for reconfigurations and the rest constructs the core of control logic. The controller makes the following decisions in response to the arrival of workload: 1) load balancing to re-assign workload over all tasks at beginning to handle the data skewness, 2) scaling out by one task at time 100s and 200s to adapt to the increase of arrival rate, 3) scaling in by one task at time 400s to save computation resources.

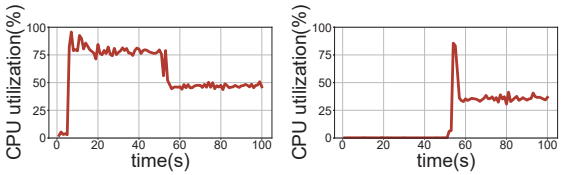
The comparisons in terms of processing latency under Trisk and native Flink are shown in Figure 6b. We use an initialized static configuration as our baseline for the stock trading application. The results show that although the static configuration works well at the beginning, it cannot handle the fluctuating arrival data, resulting in increasing latency two orders of magnitude higher. Comparatively, the controller implemented using Flink's native reconfiguration is able to adapt to the changes in workload; however, it induces high latency spikes, one to two order of magnitude higher, during the execution of reconfiguration, which makes the stream job failed to process the arrival data with low latency. In contrast, with the partial pause-and-resume mechanism, the controller on Trisk that made the same decisions shows negligible latency increment during reconfigurations and can process the input data in low latency during the stream processing.

Service-quality-aware controller. We design a service-quality-aware controller for fraud detection to optimize the detection precision. The machine learning-based fraud detection consists of two parts - training and serving. The training part is written by using the Python `scikit-learn` [36] library and the serving part is written in a Flink stream job. The controller gets the latest model parameters via Restful APIs and update them via the change of logic reconfiguration on the corresponding tasks in the Flink job. We set the initial configurations as follows. 1) We use the latest transaction data to train the model via scikit-learn and update them to the Flink job at 100 seconds. 2) We set the parallelism of the stream job to be 16. The controller updates the model parameters when the F1 score of predicting fraudulent transactions is lower than 0.6. The controller is written in 110 lines of code, in which 15 lines of code are to construct parameters for reconfigurations and the rest are for control logic.

Figure 7 plots the F1 scores for fraud detection, comparing the prediction accuracy with and without the dynamic changes of execution logic. When the emerging fraud data arrives, the F1 score of fraud detection in the static configuration has decreased to 0.2 in around 30 seconds. To handle this problem and increase the performance, Trisk controller updates the decision tree parameters that has been trained with the latest arrival data at time 200s and 320s when the F1 score was decreasing drastically. Compared



(a) Latency behavior of fraud detection.



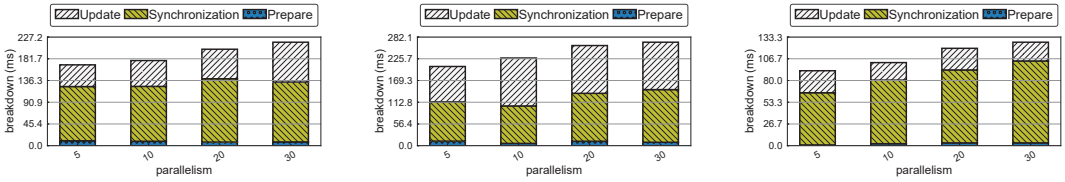
(b) CPU utilization on node 1.(c) CPU utilization on node 2.

Figure 8: Placement in fraud detection to reduce resource contention and processing latency.

to the baseline, the change of logic controller optimized the performance of fraud detection. Although the F1 score can be higher with more parameters tuning effort when updating the model, the current change of F1 score shows the effectiveness of change of logic.

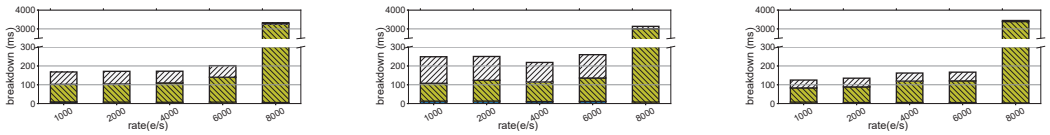
Placement Controller. We built a controller that applies placement to reassign resources. This placement controller is written in 102 lines of code, which mainly contains a resource slot re-allocation control logic. The default scheduling strategy in Flink tries to allocate resources locally to fully utilize the resources in a machine. Because the CPU resource is not isolated, such scheduling strategy causes CPU resource contention for computation intensive jobs. To mitigate this problem, we apply a placement to equally allocate tasks to machines at time 50s, such that tasks can have enough resources to support their services.

Figure 8 shows the application latency and the CPU utilization of both machines under the placement reconfiguration, where we illustrate the effectiveness of placement by comparing the latency before and after placement. The CPU utilization of the machines are measured over time by using the perf tool in Linux to understand more detailed behaviors. We have made two major observations based on the experimental results. First, the latency figure shows that placement helps the application achieve nearly 50% lower latency compared to the original setting. For CPU utilization, machine 1 and machine 2 have balanced CPU utilization after migrating half of tasks. Second, the placement incurs a latency spike at around 600 ms during the reconfiguration, which drops back in a couple of seconds. This is because placement needs to reassign resources and is executed by stopping tasks and restarting them with the newly allocated ones.



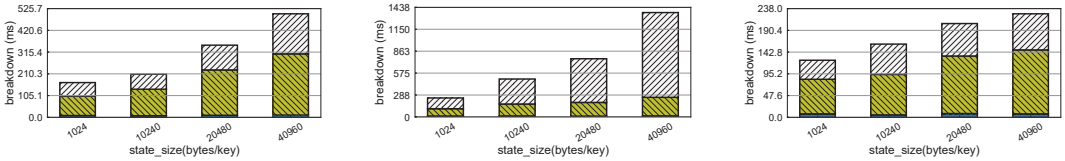
(a) load balancing (b) scaling (c) change of logic

Figure 9: Impact of parallelism.



(a) load balancing (b) scaling (c) change of logic

Figure 10: Impact of arrival rate.



(a) load balancing (b) scaling (c) change of logic

Figure 11: Impact of state size.

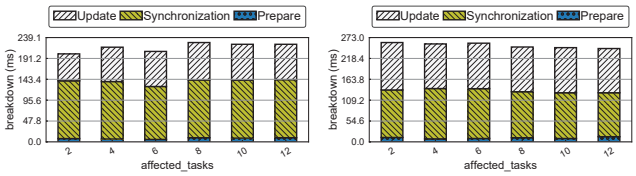
5.3 Micro Benchmarks

In this subsection, we study the details of reconfiguration execution in Trisk on Flink. Specifically, we are interested in understanding the behavior of reconfigurations under different workload characteristics via a micro-benchmark. To show detailed behaviors, we breakdown the execution of reconfiguration according to the prepare-sync-update pipeline, and plot the completion time of each phase. We consider the possible parameters that affect the performance of reconfiguration including the degree of parallelism, data arrival rate, state size, and the number of affected tasks during the reconfiguration. We evaluate the performance under three reconfigurations, i.e., load balancing, scaling, and change of logic, that execute different types of primitive operations in the update phase, respectively.

We set the default configuration of the micro-benchmark as follows. 1) The default degree of parallelism of the counter operator is set to 20, and that of the source operator is set to 5. 2) The default arrival rate is 6,000 tuples per second which is 75% of the maximum processing capacity. 3) We set the number of keys to 1,000, and the default state size per key is 1024 bytes. 4) We trigger a reconfiguration after 30 seconds, when all the keys will be appeared in each task and the state size is more stable. 5) All reconfigurations are applied on the counter operator, which is implemented in a stateful `RichFlatMapFunction` in Flink. For load balancing, we choose two tasks to randomly shuffle their workload. For scaling, we scale out the counter operator by one task, and shuffle the workload randomly among all tasks. For change of logic, we add a simple log-printing logic to the original execution logic, and keep the state unchanged.

Overall, we observe that the reconfigurations in Trisk can be completed in milliseconds level with good scalablity. According to the breakdown results, we have three major findings. The prepare phase spent negligible time to complete. The synchronization time mainly depends on the original synchronization mechanism and state management logic in stream systems. The main performance bottleneck of the update phase is highly related to the status of the original stream processing in terms of current state size.

Parallelism. In this experiment, we choose the parallelism of the counter tasks from 5 to 30, and the results are as shown in Figure 9. In general, Trisk keeps good scalability where the completion time are in hundreds of milliseconds level. The reconfiguration completion time slightly increases with the degree of parallelism. We have made three observations from the parallelism experiments as follows. First, experiments in load balancing and scaling show a monotonically increased completion time, in which the update time is increased. This is mainly resulting from



(a) load balancing **(b) scaling**

Figure 12: Impact of affected tasks.

the cross nodes state migration. The parallelism setting of 5 and 10 are smaller than the update time in 20 and 30, because state update in small parallelism is executed locally. Second, scaling requires longer update time compared to load balancing, because an additional time-consuming step of resource allocation is needed during the update. Third, change of logic shows a consistent update time under different degrees of parallelism, because there is no state migration overhead during the function update. The synchronization time increases steadily with parallelism, because in change of logic, all tasks under the operator are required to be updated and the number of affected tasks equals the degree of parallelism.

Arrival rate. In this experiment, we vary the arrival rate of each counter task from 1,000 to 8,000, which covers different utilization of each task, and show measurement results in Figure 10. From the figure, we have made two observations. First, although the completion time increases under a higher arrival rate, the increment is relatively small. When the arrival rate exceeds the maximum processing capacity of the tasks, the synchronization time has increased drastically, because the task is backlogged and cannot process input data in time, and therefore, stream barriers need to be queued until the data before it has been processed, which takes a long time. Such high synchronization overhead can be reduced by using the unaligned checkpoints [13], where the barriers are preemptive to be processed in advance. Second, the update times of the three reconfigurations keep consistent under different arrival rates, which indicates that the performance of the update mechanism does not get affected by the arrival rate.

State size. In this experiment, we vary the state size of each key from 1024 bytes to 40,960 bytes and show the results in Figure 11. Two observations are made from the figure. First, the larger state size results in higher completion time for all reconfigurations. In particular, the synchronization time has increased with larger state size caused by the checkpoint mechanism in Flink, where each task needs to return its local state snapshot to the job manager once it receives all the barriers from its upstream. The overhead can be optimized with incremental checkpoint [39], where the snapshot state size can be reduced by recording the differences between each checkpoint. Second, scaling spends more time on updating with a larger sized state, because scaling affects all tasks in the counter operator, and the size of state to be migrated is larger than that in load balancing. Third, the update time in change of logic increases with state size. This is because for stateful tasks in Flink, the execution logic and computation involve the state to satisfy stateful data processing, which is part of the instance variables; and therefore, reloading a new function object requires to reconstruct the state variables accordingly.

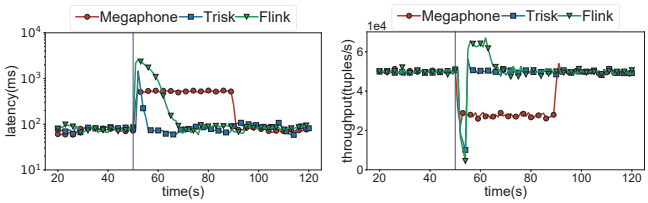
Affected tasks. In this experiment, we evaluate the impact of affected tasks and vary the number of affected tasks for scaling and load balancing. The affected tasks for scaling refers to number of new tasks to be scaled out, and the affected tasks for load balancing is the number of tasks whose states need to be updated. As shown in Figure 11, in general, the number of affected tasks in scaling and load balancing does not affect the completion time of reconfiguration much, which is consistent with our findings in the parallelism experiment, because the update on each task is highly asynchronous. Furthermore, another reason is that the state size is relatively small, which incurs negligible overhead on synchronization.

5.4 Performance Comparison

We compare the reconfiguration performance of Trisk with that of two existing frameworks: native Flink and Megaphone. For stream jobs, the processing latency is usually the key performance criteria; however, for consistency and correctness, systems unavoidably bring some latency overhead into the running jobs while applying reconfigurations. Therefore, in this experiment, we use the latency overhead as the performance criteria to evaluate the reconfiguration frameworks. We analyze the latency overhead in two ways: 1) how much is the overhead, i.e., increased latency during the reconfiguration; and 2) how long does the overhead exist, i.e., the reconfiguration completion time. We use the word-count workload and set the state size of each key to be 40,960 bytes. The controller triggers a global load balancing at 50 seconds, which shuffles the workload among all tasks of the counter operator, i.e., all tasks under the counter operator are the affected tasks.

For comparison, we implemented Megaphone on top of Flink following prior work [19]. In particular, we leverage a Apache Kafka message queue to enable output timestamp probe for upstream tasks in Megaphone.

Figure 13 shows latency and throughput of the three systems for comparisons, from which we have made two major observations. First, the partial pause-and-resume mechanism in Trisk achieves lower completion time than the native reconfiguration in Flink and Megaphone's fluid



(a) Latency (b) Throughput

Figure 13: Load balancing under Trisk, Megaphone and Flink.

migration mechanism. In particular, Trisk takes 1,462 ms to complete the load balancing, while Flink needs 2,177 ms with higher peak latency incurred during the reconfiguration. In contrast, Megaphone needs much longer time which is 39,258 ms to complete the load balancing, but because of the nature of fluid migration, the latency during migration is within 500 ms. Second, the throughput figure shows that the throughput of all systems have dropped during the reconfiguration. The throughput in Trisk is 10,075 tuples per second, which is higher than that in Flink but is still lower than the input rate. This main reason is because all tasks under the counter operator are affected tasks, and will be paused until the `JobConfigCoordinator` receives all acknowledgements from the affected tasks. Nevertheless, Trisk has less backlogged data than Flink because of shorter completion time. The throughput of Megaphone reaches on average 30,000 tuples per second during load balancing, which is higher than both Trisk and Flink, and there are less backlogged data after the completion of the load balancing. However, this comes at the cost of a much longer completion time, an order of magnitude higher than that of Trisk and Flink. In summary, Trisk achieves competitive performance compared to state-of-the-art solutions with respect to completion time, latency and throughput.

6 RELATED WORK

In this section, we describe the related works not cover in previous sections and discusses how they are related to Trisk.

Stream systems. A lot of stream systems have emerged in both academia and industry [1–5, 16, 20, 25, 30, 31, 38, 43, 46]. Existing systems can be divided into pure stream systems [11, 24, 32, 41, 41] that process data once it arrives and mini-batch systems [20, 31, 44] that adopt the *bulk synchronous parallel* (BSP) model [42] by operating on micro batches. Trisk is a control plane designed for pure stream systems, where tasks in a stream pipeline are deployed as long-run instances. Trisk is able to execute reconfigurations by updating the computation of those long-run tasks.

Controllers for stream processing. Controllers for stream systems maintain control policies that decide when and how to dynamically reconfigure the systems to optimize performance. Prior works have proposed various controllers with different performance objectives [12, 14, 15, 22, 23, 26]. DS2 [22] and Dhalion [14] propose controllers that make scaling decisions to maximize throughput. In particular, DS2 retrieves the arrival rate and the useful time of each task periodically to detect the bottleneck of a stream job; while Dhalion monitors the backlog of a stream job and makes decisions according to the backpressure. DRS [15] and Nephele [26] introduce controllers that focus on latency guarantees and make decisions using queuing models, which require latency and service time from stream jobs. Henge [23] achieves SLO/SLAs while maximizing the overall system utilization by introducing an automata-based cluster resource management. Particularly, it performs cluster-wide reconfiguration to increase and/or decrease resources for stream jobs. Trisk provides versatile and efficient reconfigurations, such that controllers can be easily implemented by migrating control policies on Trisk and execute reconfigurations by using built-in Trisk APIs based on control decisions.

Controllers in other areas. Beyond stream processing, controllers have been developed in other areas with different optimization purposes and different reconfiguration techniques [27, 33, 35, 37, 45]. Similar to the controllers in stream systems, these works introduced control policies based on the specific requirements of jobs, and optimize the processing of the target jobs by applying the system-specific reconfigurations. In cloud computing area, Wiera [33] is designed to optimize data placement with users-specified requirements in geo-distributed datacenters to handle the changes of network, workload and storage access pattern. KungFu [27] is designed to enable adaptive training for distributed machine learning, which allows users to implement policies to tune the parameters during training. There are also policies designed for database management systems [35, 37, 45] to optimize query processing. Peloton [35] introduces a policy to tune the execution of operations with various database-specific techniques such as indexing and data partitioning.

7 CONCLUSION

In this paper, we design and implement Trisk, a control plane solution that enables versatile, efficient and user-friendly reconfigurations for stream systems. Trisk maintains a task-centric abstraction and describes various reconfigurations with encapsulated APIs. Trisk executes reconfigurations via a prepare-sync-update pipeline, a type of partial pause-and-resume mechanism, by leveraging the synchronization mechanism in the native stream systems with low system overhead. The experiments confirm that Trisk supports versatile reconfigurations with sub-second completion time.

8 ACKNOWLEDGEMENT

We would like to thank Indranil Gupta, the anonymous reviewers and our shepherd Abhishek Chandra from the ACM SoCC program committee for the insightful comments and suggestions. This research was supported in part by the Ministry of Education of Singapore Academic Research Fund (AcRF) R-252-000-A67-114 and R-252-000-B18-114.

REFERENCES

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The design of the Borealis stream processing engine.. In *CIDR*. 277–289.

[2] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *The VLDB Journal* 12, 2 (2003), 120–139.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at Internet scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[5] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal* 23, 6 (2014), 939–964.

[6] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 601–613. https://doi.org/10.1145/3183713.3190664

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

[8] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1718–1729.

[9] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

[10] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond Analytics: The Evolution of Stream Processing Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2651–2658. https://doi.org/10.1145/3318464.3383131

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[12] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. ACM, 725–736.

[13] Apache Flink. 2020. *Flink Unaligned Checkpoints*. https://flink.apache.org/2020/10/15/from-aligned-to-unaligned-checkpoints-part-1.html.

[14] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1825–1836.

[15] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2015. DRS: dynamic resource scheduling for real-time analytics over fast streams. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE, 411–420.

[16] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 213–231.

[17] Cyril Goutte and Eric Gaussier. 2005. A probabilistic interpretation of precision, recall and F-score, with implication for evaluation. In *European conference on information retrieval*. Springer, 345–359.

[18] Manish Gupta, Jing Gao, Charu C Aggarwal, and Jiawei Han. 2013. Outlier detection for temporal data: A survey. *IEEE Transactions on Knowledge and data Engineering* 26, 9 (2013), 2250–2267.

[19] Moritz Hoffmann, Andrea Lattuada, and Frank McSherry. 2019. Megaphone: latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1002–1015.

[20] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*. ACM, 59–72.

[21] Kaggle. 2020. *Credit Card Transactions Fraud Detection*. https://www.kaggle.com/kartik2112/fraud-detection.

[22] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 783–798.

[23] Faria Kalim, Le Xu, Sharanya Bathey, Richa Meherwal, and Indranil Gupta. 2018. Henge: Intent-driven multi-tenant stream processing. In *Proceedings of the ACM Symposium on Cloud Computing*. 249–262.

[24] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 239–250.

[25] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. Streamscope: continuous reliable distributed processing of big data streams. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 439–453.

[26] B. Lohrmann, P. Janacik, and O. Kao. 2015. Elastic Stream Processing with Latency Guarantees. In *IEEE 35th International Conference on Distributed Computing Systems*. 399–410. https://doi.org/10.1109/ICDCS.2015.48

[27] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. 2020. Kungfu: Making training in distributed machine learning adaptive. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 937–954.

[28] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, et al. 2018. Chi: a scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1303–1316.

[29] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM.

[30] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 439–455.

[31] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*. 113–126.

[32] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. 2017. Samza: stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1634–1645.

[33] Kwangsung Oh, Nan Qin, Abhishek Chandra, and Jon Weissman. 2019. Wiera: Policy-driven multi-tiered geo-distributed cloud storage system. *IEEE Transactions on Parallel and Distributed Systems* 31, 2 (2019), 294–305.

[34] The pandas development team. 2020. *pandas-dev/pandas: Pandas*. https://doi.org/10.5281/zenodo.3509134

[35] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*, Vol. 4. 1.

[36] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[37] Thao N Pham, Panos K Chrysanthis, and Alexandros Labrinidis. 2016. Avoiding class warfare: managing continuous queries with differentiated classes of service. *The VLDB Journal* 25, 2 (2016), 197–221.

[38] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 1–14.

[39] Stefan Ricther and Chris Ward. 2018. *Managing Large State in Apache Flink: An Intro to Incremental Checkpointing*. https://flink.apache.org/features/2018/01/30/incremental-checkpointing.html.

[40] Nicoló Rivetti, Leonardo Querzoni, Emmanuelle Anceaume, Yann Busnel, and Bruno Sericola. 2015. Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. ACM, 80–91.

[41] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 147–156.

[42] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[43] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

[44] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM symposium on operating systems principles*. 423–438.

[45] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. 2013. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. *Proceedings of the VLDB Endowment* 6, 12 (2013), 1374–1377.

[46] Yunhao Zhang, Rong Chen, and Haibo Chen. 2017. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*. 614–630.