2025 IEEE 45th International Conference on Distributed Computing Systems (ICDCS) | 979-8-3315-1723-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICDCS63083.2025.00101

Spacker: Unified State Migration for Distributed Streaming

1st Yancan Mao
School of Computing
National University of Singapore
maoyancan@u.nus.edu

2nd Shuhao Zhang
School of Computer Science and Technology
Huazhong University of Science and Technology
shuhao_zhang@hust.edu.cn

3rd Richard T. B. Ma
School of Computing
National University of Singapore
tbma@comp.nus.edu.sg

***Abstract*—State migration is a crucial aspect of managing stateful stream processing applications, enabling load balancing, fault tolerance, and dynamic scaling. Existing state migration solutions make performance trade-offs between *completion time, latency spike, and system overhead*; however, they lack the flexibility to adjust these trade-offs across different application scenarios. In this paper, we propose *Spacker*, a unified framework that enables configurable state migration for flexible performance trade-offs. *Spacker* decomposes state migration into fine-grained key-level operations and introduces an abstraction of *planning strategy*, featuring three tuning knobs, that allow for flexible planning of operations. To further improve the efficiency, we design a *non-disruptive migration protocol* that minimizes the blocking of data processing during state migration. We have integrated *Spacker* with Apache Flink and implemented an adaptive planning strategy as an example that realizes the abstraction. Our results show that *Spacker*, with the planning strategy, can make adaptive planning decisions, based on analyzing the decision trade-offs under varying workload characteristics. It can reduce latency spikes while maintaining appropriate completion time and system overhead compared to statically configured migration solutions.**

***Index Terms*—Distributed System, Stream Processing, Cloud Computing, Resource Management**

I. Introduction

Streaming applications are long-running and process continuous data streams in real-time, e.g., monitoring social media feeds [1] or analyzing live sensor data from IoT devices [2]. They are often designed to run across multiple task instances in order to provide scalability, allowing them to handle large volumes of data. Because data streams are inherently dynamic with fluctuating rates and distributions over time, to achieve data processing with low latency and high throughput, Stream Processing Engines (SPEs) [3]–[7] need to reconfigure part of dataflow computation of jobs, e.g., re-scaling and load balancing [8], at runtime. For stateful streaming jobs, reconfiguration requires state migration, a crucial process that needs to ensure consistent redistribution of the data stream and its associated state across tasks.

The problem: Lack of flexibility. Existing state migration solutions are designed with specific performance goals, such as a shorter completion time or a lower latency spike. However, different stream processing scenarios may require state migration to achieve individual performance goals, while current solutions have fixed state migration strategies and lack the flexibility to achieve varying goals. In fact, none of the solutions can achieve all the goals, instead, they make trade-offs on the various performance factors. To illustrate this, we categorize existing solutions [9]–[15] into three main types, each with its own performance goals and associated trade-offs: 1) *All-at-once State Migration* [12]–[15] focused on short completion time, but results in high latency spikes. 2) *Progressive State Migration* [9], [10] aims to smooth latency spikes, but takes longer to complete. 3) *Proactive State Replication* [11] reduces completion time and latency spikes but at the cost of higher system overhead (e.g., additional CPU and I/O cost) during normal stream processing. This limited flexibility highlights the need for a more configurable state migration approach that better accommodates different stream processing scenarios.

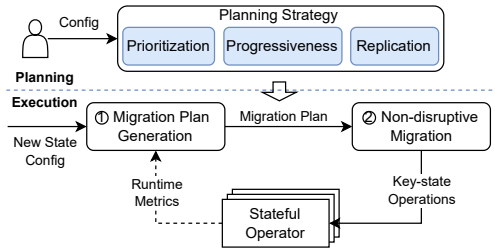


Fig. 1: An overview of the planning and execution workflow of state migration in *Spacker*.

Our solution: We introduce *Spacker*[1], a unified framework that enables configurable state migration in SPEs. As illustrated in Figure 1, *Spacker* separates a state migration solution into two components: *Planning*, a logical strategy that defines the behaviors of state migration, and *Execution*, a physical process that executes the strategy. Through this, *Spacker* allows users to fine-tune state migration strategies for their managed streaming jobs at a high level, while providing implementations that can leverage the specifics of the SPE. First, to enable flexible planning, *Spacker* builds an abstraction of *planning strategy* upon the key-level granularity and focuses on the *key-state operations* that migrate the state of the associated keys. In particular, the configurable *planning strategy* features three tuning knobs, i.e., *Prioriti-*

[1] *Spacker* Source Code: https://github.com/sane-lab/Spacker

2575-8411/25/$31.00 ©2025 IEEE
DOI 10.1109/ICDCS63083.2025.00101

zation, *Progressiveness*, and *Replication*, to plan for key-state operations. Given a configured *planning strategy*, *Spacker* will generate a *migration plan* that specifies the execution of key-state operations for each invocation of state migration during runtime. Second, to provide efficient execution, *Spacker* uses a *non-disruptive migration protocol* that encodes key-level state operations into non-blocking control messages, minimizing the impact on the processing of non-migrating keys during state migration. Our main contributions are summarized as follows:

- In Section II, we review existing state migration solutions and analyze the causes of their performance trade-offs.
- In Section III, we present Spacker, a unified framework that decouples state migration into planning and execution. It introduces a configurable planning abstraction to support flexible strategy definition and employs a non-disruptive protocol to minimize blocking during execution.
- In Section IV, we integrate Spacker into Apache Flink with a scalable runtime architecture. We also implement a heuristic strategy that adaptively plans key-state operations based on workload characteristics.
- In Section V, we evaluate Spacker under diverse streaming scenarios. Our results show that Spacker enables adaptive planning with low overhead and provides insights into how different planning strategies affect system performance.

II. Background

A. Distributed Stream Processing

The computation logic of a streaming job is described by a *logical topology* represented as a directed acyclic graph (DAG), where vertices are *operators* and edges depict the flow of intermediate data between *operators*. To exploit data parallelism, data streams are often defined on and partitioned by a key space [3], [15], and each *operator* is instantiated with multiple *tasks* to process the different stream partitions in parallel during physical execution. The assignment of *keys* among the parallel *tasks* is referred to as *state configuration*. Tasks from stateful operators need to maintain the latest *state* for their assigned keys, while their upstream needs to route data according to the *state configuration*. Stateful stream processing needs to guarantee consistency for exactly-once semantics. We denote a record with a time t be e_t and the latest state after processing e_t as s_t, consistency in this context means a) at any time, records of each key is only routed to one downstream task once, and b) downstream tasks always have the latest state s_{t-1} when processing a new record e_t.

State migration is the process of applying a new *state configuration*, requiring the redistribution of keys and their associated state among parallel tasks. As data streams from upstream to downstream tasks, during migration, upstream tasks need to update routing, and downstream tasks need to redistribute state. These updates correspond to the change of routing in the upstream operator and the migration of the state for keys in the downstream operator. For each key, these updates are achieved by its key-state operation, and different key-state operations can be executed independently. Executing these updates asynchronously might cause inconsistency. For example, if only routing is updated by upstream tasks after producing a record e_t while state redistribution is not completed when downstream receives e_t, downstream tasks may miss new data for keeping the state up-to-date. Thus, tasks need to be coordinated to update routing and redistribute state while preserving consistency conditions.

B. Related Work and Motivation

Due to the increasing demand for real-time data processing, many SPEs [3], [5]–[7], [16]–[22] have been proposed to enable stream processing on the cloud and multicore. These systems often require dynamic reconfiguration, such as re-scaling or load balancing, to maintain performance under fluctuating workloads. A key enabler of such reconfiguration is state migration, which ensures consistent redistribution of key state across parallel tasks.

1) Dynamic Reconfiguration: Dynamic reconfiguration has been widely studied as a mechanism for improving the elasticity and performance of SPEs. Prior works in this area focus on two main components: controllers and control planes. a) *Controllers* [23]–[27] maintain control policies for making decisions for when and how to dynamically reconfigure the systems. These policies often aim to maximize throughput [23], [24], [26] and minimize latency [25], [27]. b) *Control planes* [13], [14], [24], [28]–[30] provide programmable APIs for implementing various control policies. They offer diverse and efficient reconfiguration mechanisms, such as dynamic scaling and load balancing, and can be utilized by user-defined control policies. While these works focus on reconfiguration policies and mechanisms, Spacker complements them by enabling efficient and customizable state migration to improve reconfiguration execution.

2) State Migration Techniques: Existing solutions are designed with specific goals and differ in strategies of migrating key state, and they typically make trade-offs between latency spikes, completion time, and system overhead. We categorize prior techniques into three main paradigms: a) *All-at-once State Migration* [12]–[14] batches all key-state operations and execute them all-at-once. It pauses the processing of migrating keys until all state is migrated successfully. *All-at-once State Migration* takes a shorter time to complete and does not introduce system overhead during normal stream processing; however, it incurs higher latency spikes due to the simultaneous pausing of migrating keys. b) *Progressive State Migration* [9], [10] groups key-state operations into multiple chunks and migrates them progressively. It migrates fewer keys per chunk and smooths latency spikes compared to *All-at-once State Migration*, but incurs a longer state migration time. Meces [10] further prioritizes the state to migrate according to the arrival order of keys to optimize latency spike and completion time. c) *Proactive State Replication* [11] proactively replicates state to remote nodes during normal stream processing. This paradigm executes all key-state operations at once by reloading the replicated state from the destination node instead of transferring state from source to destination

tasks. *Proactive State Replication* reduces overall state migration time and experiences lower latency spikes; however, at the cost of extra state replication overhead during normal stream processing.

While these techniques are effective in specific scenarios, they adopt fixed strategies that lack adaptability across diverse workloads. No single approach can simultaneously optimize for all performance metrics. However, different workloads may demand different trade-offs between latency, migration time, and system overhead. For instance, latency-sensitive workloads such as stock exchange applications may prioritize minimizing latency spikes, whereas IO-intensive workloads like streaming ETL pipelines may require faster migration with lower overhead for resource efficiency. As a result, existing solutions lack the configurability and flexibility needed to adapt migration behavior across diverse runtime conditions. Spacker fills this gap by providing a unified and configurable migration framework under a unified framework. It exposes a configurable design space that enables users to flexibly navigate trade-offs based on workload demands. By decoupling planning from execution and introducing three tuning knobs, Spacker enables efficient, fine-grained, and adaptable state migration for diverse streaming scenarios.

III. *Spacker* DESIGN

In this section, we first discuss *Spacker*'s decoupled architecture to enable flexible state migration and analyze the strategy space of planning. Then, we introduce an abstraction of planning strategy that allows configure strategies through three tuning knobs. Next, we present the migration execution pipeline that executes key-state operations efficiently.

A. Design Overview

As depicted in Figure 1, a unique design of *Spacker* is the decoupling of execution from planning. The planning (Section III-B) describes the logical behaviors of state migration, while the execution (Section III-C) physically executes state migration following the planning strategy. *Spacker* decomposes state migration into key-state operations, allowing users to configure a planning strategy for these operations. *Spacker* further provides the runtime that can make planning decisions and execute these operations in a fine-grained. As such, the problem of enabling state migration with flexible performance trade-offs boils down to an abstraction of planning strategy with a rich strategy space that generalizes existing state migration solutions.

Strategy Space of Planning. To understand the strategy space, we break down the general process of state migration from the perspective of task instances based on fine-grained key-state operations as follows. Upon an invocation of state migration specified by a new state configuration, a set of keys is reassigned from their original tasks, i.e., the sources, to new tasks, i.e., the destinations. When a source migrates keys out, the all-at-once [12]–[14] and progressive [9], [10] paradigms suggest that there is a whole spectrum of planning decisions that determine **when** each key will be migrated. As a logical decision, it determines the sequential order of migration among the keys. However, during the physical execution, migrating keys one by one may take longer to complete, as the frequent invocation of state serialization/deserialization during migration introduces additional I/O overhead. As a result, the source also decides **how** to group multiple keys [13], [14] for simultaneous migration to achieve shorter completion time and lower overhead. When a destination migrates keys in, although the state is typically transferred directly from the source, which maintains and updates them, the proactive paradigm [11] suggests that state replication could potentially affect the destination task's decision on **where** to retrieve state.

B. State Migration Planning

According to the aforementioned general process of state migration, we identify three tuning knobs to form the strategy space of planning: *Prioritization*, *Progressiveness*, and *Replication*. As our objective is to provide configurable state migration that can provide flexible trade-offs between performance goals, we further discuss how the three tuning knobs can be used to trade-off between latency spike, completion time, and system overhead.

1) Abstraction of Planning Strategy: Based on the strategy space, *Spacker* builds the abstraction of planning strategy with three tuning knob "Functions", i.e., *Prioritization* function, *Progressiveness* function, and *Replication* function. Any planning strategy can be specified by realizing the abstraction. The corresponding programming APIs can be illustrated in Algorithm 1. Users can implement a new planning strategy that overrides the original functions. Then, *Spacker* can load the new planning strategy to enable desirable migration behaviors.

Prioritization defines the sequencing strategy that decides how early a key will be migrated. Users may define *prioritization functions* that take migrating keys K as input and prioritize them into a sequence $Ordered\_Seq$. For example, users can define a hotkey-first *prioritization function* by prioritizing any popular key k_i over other keys according to the key-level workload distribution.

Progressiveness defines the grouping strategy that groups ordered keys into non-overlapping chunks for sequential migration. The progressiveness is determined by the number of chunks, i.e., the more (or fewer) chunks to migrate means more (or less) progressive. Users may define *progressiveness functions* that take the $Ordered\_Seq$ as input and buffer them into ordered chunks $Chunks$. Instead of simply tuning per chunk size, *Spacker* allows users to customize the number of chunks and the size of each chunk. This exposes a larger space for more complicated scenarios. For example, when state access distribution is highly skewed, grouping hotkeys together can recover load imbalance faster.

Replication defines a replication configuration that determines runtime state distribution and further affects where the state can be retrieved during state migration. Users may define *replication functions* to specify the configuration $Config.k_i$ for each key, which consists of the number of replicas, i.e., $Num\_Replica$, and its distribution policy, i.e.,

Algorithm 1: Programming APIs Overview.

```
Function Prioritization_Function(K):
    foreach key k_i ∈ K do
        Ordered_Seq[j] ← Prioritize k_i;
Function Progressiveness_Function(Ordered_Seq):
    Initialize n Chunks with specified sizes;
    foreach key k_i ∈ Ordered_Seq do
        Chunk ← Buffer k_i;
        if Chunk is full then
            Chunk = Chunks.next;
Function Replication_Function(K_space):
    foreach key k_i ∈ K_space do
        Config.k_i ← Set_Num_Replica;
        Config.k_i ← Set_Replicate_Policy;
```

Replication Config.

Key	Num	Policy
k1	0	-
k2	2	Uniform
k3	0	-
k4	0	-
k5	1	Uniform
k6	0	-

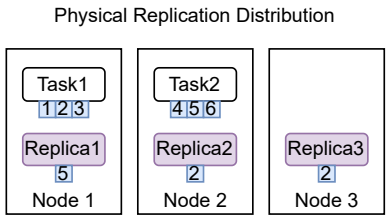


Fig. 2: An example of the replication configuration and the physical replication distribution at runtime.

Replicate_Policy. For example, users may choose different strategies, such as consistent hashing used in prior work [11] to decide where to replicate state. We show an example of replication configuration and its physical replication distribution in Figure 2. In this scenario, the state of keys $k2$ and $k5$ is replicated across a three-node cluster, each node hosting a replica. The state replication follows the given replication configuration, which assigns two replicas for $k2$ and one replica for $k5$, using a uniform distribution policy.

Although *Prioritization* and *Progressiveness* will be applied upon each state migration request, *Replication* will proactively replicate the state to the different nodes according to the configuration from the beginning of stream processing, through which destination tasks can retrieve the state from any node that contains the latest state for the associated keys during state migration.

2) Performance impact of tuning knobs: We illustrate the planning strategy's qualitative impacts on the performance metrics in Figure 3, which will be quantitatively verified in a later section. First, as illustrated in Figure 3a, latency spikes can be mitigated at the cost of an increasing completion time by tuning to be more progressive. Because key-state operations need to temporarily block the processing of the associated keys to maintain consistency, less progressive with fewer chunks but larger chunk sizes will block the processing of more keys and cause more accumulated backlog, which will result in higher latency spikes. In contrast, more progressive with more chunks but smaller chunk sizes can reduce such latency spikes, but takes longer to complete the entire migration process

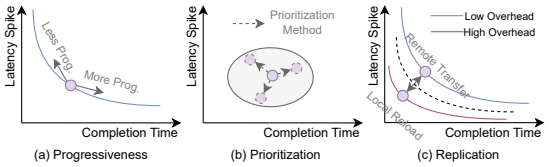


Fig. 3: Impact overview of three tuning knobs.

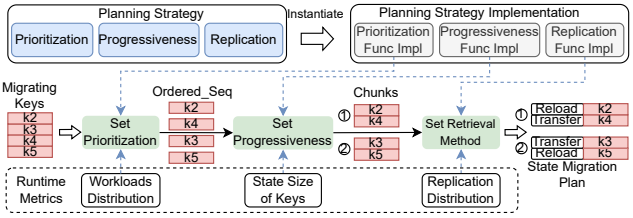


Fig. 4: The workflow of migration plan generation.

for all chunks, since the more frequent invocation of data serialization/deserialization will cause higher I/O overhead. Second, as illustrated in Figure 3b, different prioritization methods can also affect the performance of state migration. Choosing an appropriate prioritization method that aligns with the specific stream processing scenario can reduce latency spikes and completion time. For example, by migrating the hotkeys early, the resource contention in an overloaded task can be mitigated sooner; by migrating the keys that arrive earlier [10], the backlog accumulated during the period of blocking will be reduced. Third, as illustrated in Figure 3c, tuning the replication configuration of keys may trade off the completion time of migration with the system overhead incurred during normal stream processing. Since replication provides a potential alternative to load the state from a closer replica and avoid the I/O contention of source tasks, it reduces network latency for transferring state, resulting in lower migration time and latency spikes in general. However, to keep replicas fresh, all replicas need to be synchronized periodically, which incurs system overhead.

C. State Migration Execution

Given a configured planning strategy, the state migration execution is designed with two modules: migration plan generation and non-disruptive migration execution, to realize flexible and efficient migration execution.

1) Migration Plan Generation: Upon an invocation of state migration, the set of keys that need to be migrated can be identified by comparing the new state configuration with the current one. The execution of the corresponding key-state operations will follow an instance of planning decision, which we call a *migration plan*, generated through a pipelined procedure. It is through this procedure that we apply the specified planning strategy for the streaming job at runtime, because a planning strategy only defines a policy that might require the runtime status/metrics of the job and cannot finalize the decision beforehand. We present the execution workflow of this procedure in Figure 4. In particular, a migration plan

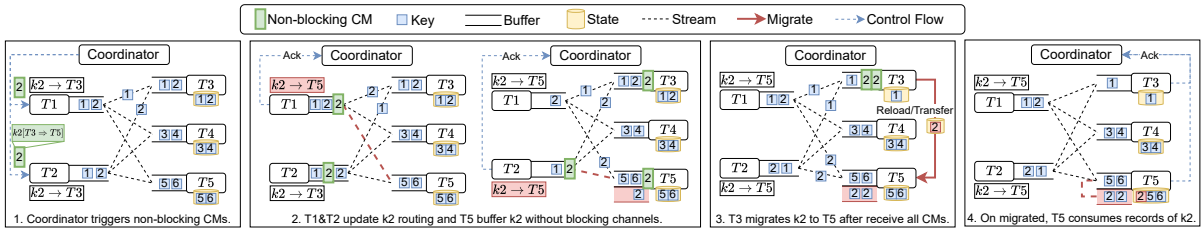


Fig. 5: The workflow of the non-disruptive migration protocol.

is generated by following three sequential steps that make decisions with the implemented planning strategy and the associated runtime metrics as inputs.

1) *Set Prioritization*: This step generates an ordered sequence of migrating keys by applying the user-defined *Prioritization* function and the corresponding runtime metrics. For example, using a hotkey-first *Prioritization* function, the procedure prioritizes migrating keys ($k2 \sim k5$) by checking the current workload distribution and generates an ordered sequence ($k2, k4, k3, k5$).

2) *Set Progressiveness*: This step buffers the ordered sequence into ordered chunks based on a user-defined *Progressiveness* function and runtime metrics. For example, given a *Progressiveness* function that attempts to buffer the ordered sequence into two chunks while equalizing the state migration time for each chunk, the procedure may generate chunks ($[k2, k4], [k3, k5]$) based on the state size of keys. Note that *Set Progressiveness* does not change the order of migrating keys; it is primarily configurable for how to buffer keys sequentially into chunks.

3) *Set Retrieval Method*: This final step suggests the method for destination tasks to retrieve state based on the current distribution of state replicas. Specifically, the destination task can retrieve state from either the local replica or from the source task. If the destination node contains the replica for the state, the procedure sets a mark "Reload" for efficient local retrieval; otherwise, destination tasks need to retrieve the state from a remote node. Since the source task always has the latest state, the procedure sets a mark "Transfer" to transfer the state from the source task. In this example, the procedure sets a reload locally method for $k2, k5$ and a transfer remotely method for $k3, k4$ according to the current replication configuration and distribution. Note that this suggestion will be finalized during the actual state redistribution based on the latest state conditions.

2) Non-disruptive Migration: To achieve consistency and efficiency, we propose a non-disruptive migration protocol that is designed based on the fast and non-blocking control message (CM) to achieve key-level state migration with the minimum blocking impact for non-migrating keys.

The Protocol. The key techniques of our non-disruptive migration protocol are in two folds: 1) *Fast CM traveling*. Since state migration primarily affects the intermediate stream between upstream and downstream operators, instead of broadcasting a CM across the entire pipeline for slow application-level synchronization, our protocol leverages a fast CM that only travels between upstream and downstream tasks. 2) *Non-blocking synchronization*. Existing solutions block the corresponding channel for a task when the CM is received, only unblocking it when all CMs are received by the task, causing unnecessary blocking of non-migrating keys. To avoid this blocking impact, our protocol applies CMs in a non-blocking manner. It encodes migrating keys' information into CMs, allowing tasks to identify and buffer migrating keys in the data channel while continuing data processing of non-migrating keys upon receiving a CM. 3) *Key-level state redistribution*. Parallel tasks that contain migrating keys are to redistribute state from the source to the destination after synchronization. Since the state access of keys is independent, the state redistribution process of migrating keys can proceed without causing delays for non-migrating keys. To achieve this, *Spacker* exposes key-level state management primitives from existing SPEs for state redistribution, such as key-level state snapshots, remote transfers, and state table entry updates.

State Migration Process. We demonstrate an execution workflow of non-disruptive migration protocol over a streaming pipeline in Figure 5. This example aims to migrate the state of $k2$ from source task $T3$ to destination task $T5$. It consists of four main steps: control message passing, routing update, state redistribution, and buffered records consumption. *Spacker* leverages a centralized coordinator (discussed in Section IV-A) with a global view of the state distribution to coordinate the execution of the state migration.

1) *Control Message Passing*: When state migration triggers, the coordinator passes a non-blocking control message to upstream tasks $T1, T2$. The control message is encoded with key-state operations for migrating keys. In this example, the key-state operation migrates $k2$ from $T3$ to $T5$.

2) *Routing Update*: Once upstream tasks $T1$ and $T2$ receive CMs from the coordinator, they update their routing for migrating keys accordingly. Subsequently, they inject CMs into the data channels of downstream tasks $T3$ (i.e., source task) and $T5$ (i.e., destination task). When upstream tasks change their routing while state redistribution for destination tasks is not yet completed, destination tasks buffer the associated records for migrating keys until completion. In this example, when $T5$ receives the first CM from $T1$, it gets notified and starts to buffer the input records for $k2$. The buffer cannot be consumed until $T5$ retrieves the latest state of $k2$ from $T3$ in the next step. Note that $T3$ can continue consuming records

with the non-migrating key $k1$ from upstream tasks because of the non-blocking CM.

3) *State Redistribution*: $T3$ continues processing records of $k2$ until it processes the last CM from its data channel. Subsequently, $T3$ updates the state of $k2$ to the latest and starts to redistribute it to $T5$. It builds a temporary network connection for $T5$ to retrieve the key-level state. At this point, $T3$ checks whether the suggested retrieval method in key-state operation is "Reload", i.e., the destination node contains the replica for $k2$. If so, it further checks whether the replica has the latest state. After the replication status check, $T3$ notifies $T5$ through the connection to retrieve the state by either transferring from the remote or reloading from local.

4) *Buffered Record Consumption*: Once the destination task $T5$ retrieves the latest state $k2$ to its local state table, it consumes the buffered records for fast recovery. Consistent with the checkpoint mechanism, each task sends an acknowledgment to the coordinator after completing its local state operations. The coordinator marks the migration of the current chunk as completed when it receives acknowledgments from all affected tasks, i.e., upstream tasks and downstream source/destination tasks.

The protocol satisfies consistency conditions as defined in Section II-A. Consider a migration at time t (i.e., injecting a non-blocking CM at time t) that migrates a key k with state s_t from a source to a destination task. The protocol guarantees that: 1) *Routing update*: Upstream tasks that produce records for k change their routing after receiving the CM. Thus, any new records $e_{t'}$ with time $t' > t$, will be routed to the destination task. The source task will not receive any records of $e_{t'}$ later. 2) *State redistribution*: After processing the last CM from its data channel, the source task receives all records of k before time t and can update k's state to the latest state s_t. Thus, it can migrate the latest state s_t to the destination task. After processing the first CM from its data channel, the destination task starts to buffer the received records for k. Since all new records $e_{t'}$ of k with time $t' > t$ arrive after the first CM, the destination task can buffer all these records and consume them sequentially after the state is migrated.

Compared to existing key-level protocols [9], [10], our proposed protocol is at a "keygroup-level", which emphasizes a general paradigm compatible with various SPEs. Streaming applications often have a large key space in data streams. To manage this effectively, SPEs typically divide the key space into smaller, fixed-size keygroups [4]. Existing works carefully applied system architecture-level modifications to support key-level data management during state migration. Although state migration on the applications' key level is the finest-grained and minimally impacts non-migrating keys, it leaves compatibility and management issues for existing SPEs. For example, Meces [10] has to remap the applications' key space to the system's keygroups space for each state migration. Its fetch-on-demand approach may break the tuple processing order during alignment. Thus, we operate on the system's key-level, or "keygroup-level". This approach allows our protocol to integrate with SPEs using their native state management

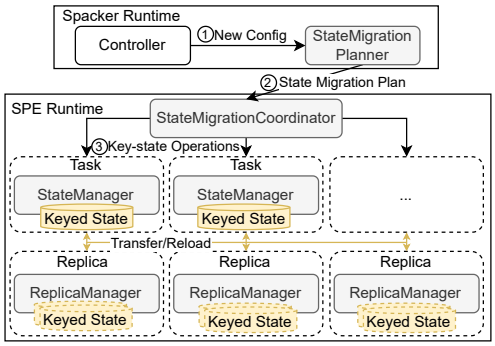


Fig. 6: *Spacker* Architecture Overview.

primitives and natively support our abstraction of planning strategy. It also avoids breaking the tuple processing order by leveraging keygroup-level primitives.

IV. Implementation

In this section, we first introduce the system architecture of *Spacker* and its integration with Apache Flink. Then, we discuss an adaptive planning strategy implementation.

A. System Architecture

Figure 6 depicts Spacker's architecture, which comprises two main components: the Spacker runtime and the SPE runtime. The Spacker runtime generates migration plans based on a given planning strategy and is designed to be compatible with existing SPEs such as Apache Flink. Within the SPE runtime, we extend several pluggable modules, including the StateMigrationCoordinator, LocalStateManagers, and ReplicaManagers, to execute key-state operations according to the migration plan.

The execution of state migration involves three main steps: ❶ In the *Spacker runtime*, a *Customized Controller* such as DS2 [23] may issue reconfigurations and propose new state configurations. ❷ The *StateMigrationPlanner* generates a migration plan based on a user-defined planning strategy implementation (Section III) and sends the migration plan to SPE runtime for execution. ❸ The *StateMigrationCoordinator* in *SPE runtime* receives the migration plan and executes key-state operations through non-disruptive migration protocol. The *StateManager* in each running task manages the local keyed state table. Once it receives key-state operations, the state retrieval method can be determined according to the replicated keyed state managed by *ReplicaManagers*.

While Spacker employs a coordinator to manage control flow during migration, the system remains highly scalable. The coordinator only issues migration instructions, while actual state transfers occur directly between StateManagers of parallel tasks over the underlying high-throughput I/O channel. This separation of control and data paths enables efficient, parallelized migration without central bottlenecks.

B. Integration with Flink

In the following, we provide an example of integrating *Spacker* with Apache Flink to demonstrate its compatibility

with existing state management frameworks. We focus on the implementation details and functionalities that enable seamless integration, efficient state migration, and fault tolerance.

Integration Components. A StateMigration Coordinator is deployed for each streaming job as a long-running thread in Flink's JobManager, ensuring seamless integration with Flink's existing infrastructure. This coordinator implements the non-disruptive migration protocol by encoding key-level information into Flink's checkpoint barriers to ensure consistency during state migration. A StateManager is encapsulated for each parallel task to manage the local state table, leveraging Flink's state management primitives for fine-grained operations. Additionally, inspired by prior work [31], we deploy a ReplicaManager as a standby task on Flink's TaskManager to maintain replicated states for running tasks. This design allows ReplicaManagers to be seamlessly managed by Flink, leveraging its built-in fault tolerance mechanisms, while also enabling efficient interactions with other components through the network stacks in Flink.

Fault Tolerance. *Spacker* works in conjunction with existing state management frameworks, relying on their built-in fault tolerance mechanisms for recovery. For instance, Flink periodically saves snapshots to a global store, enabling streaming jobs to recover from the latest checkpoint upon failure. *Spacker* extends this fault tolerance, and any failure during state migration can be recovered from the last checkpoint. Replicated states in standby tasks are cleared and reconstructed from the next checkpoint.

C. Planning Strategy Instantiation

We implement a default planning strategy that users can take as a reference when implementing their strategies. Specifically, the default planning strategy exposes multiple options in each function. It effectively optimizes the latency spikes with appropriate completion time and system overhead.

Workload Characteristics. The default planning strategy considers 5 inputs: 1) *Input Rate*: data arriving per second. 2) *Skewness*: key distribution in the input stream. 3) *Key State Size*: average size of each key's state. 4) *Number of Keys to Migrate*: total keys involved in migration. 5) *Ratio of State Access*: proportion of keys accessed per second.

Default Planning Strategy. The implemented planning strategy exposes multiple options within three tuning knobs. Note that quantitative descriptions of high (or large) and low (or small) are relative, with quantitative values depending on actual hardware and workloads.

1) *Prioritization Function Implementation* determines the prioritization approaches. It recommends the *Hotkey-first* approach when both the input rate and skewness are high, as it reduces processing delays by prioritizing hotkeys associated with heavy workloads. In cases where skewness is low, the function suggests the *Random* approach to maintain balanced migration across tasks with minimal prioritization overhead.

2) *Progressiveness Function Implementation* decides the granularity of migration by adjusting the batch size. For large key state sizes and a small number of migrating keys, the *More Progressive* option minimizes serialization overhead and limits latency spikes. When both state size and number of keys to migrate are large, the *Less Progressive* approach balances progressiveness and completion time. In scenarios with small state sizes, the *All-at-once* approach completes the migration quickly with minimal system impact.

3) *Replication Function implementation*: manages how the state is replicated during normal operation and migration. We configure two types of state replication schemes by considering the ratio of keys to replicate, i.e., *No Replication* and *High Ratio of Keys to Replicate (50%)*. The function recommends the *High Ratio of Keys to Replicate* when key state sizes are large and the ratio of state access is low, as this reduces state transfer overhead during migration. For workloads with high access frequencies, the *No Replication* strategy avoids excessive replication overhead, ensuring efficient state migration.

TABLE I: Summary of Experimental Workloads and Settings

Workload	Description	Default Settings
Stock Exchange	Stock exchange application based on the dataset from [32], with two input streams (quote/trade). Configured with auto-scaling and periodic state reshuffling.	16 parallel tasks; millions of keys; 2^{15} keygroups; state size $\sim$5GB; state access ratio $\sim$2.7%; Zipf skew $\sim$0.15.
Micro Benchmark	Synthetic key-counting workload with tunable migration behavior. Supports on-demand state migration.	8 parallel tasks; 2^{15} keys; 2^{10} keygroups; 1MB state per key; state access ratio 2% (uniform).

V. Experiment

In this section, we have conducted detailed experiments to answer the following questions:

- How can *Spacker* help conduct state migration for varying stream processing scenarios?
- What are the impacts of workload characteristics on the decision-making of planning strategy?
- How does detailed migration execution behavior change over varying workloads and strategies?

A. Experiment Setup

Hardware and Software Configurations. All experiments are conducted on a cluster of four nodes, each with an Intel Xeon Silver 4216 @2.1 GHz CPU with 16 Cores and 64GB of RAM, Ubuntu 18.04.4 LTS. All machines are in the same rack, connected by a switch with 1 Gbps bandwidth. We implement *Spacker* on Flink-1.8 configured with 8 TaskManagers, each with 10 slots. We set the default state backend to the in-memory state backend for in-memory state replication. We implement all the aforementioned techniques on Apache Flink. Although Flink-1.8 is an old version, its core components, e.g., state management and data processing modules, remain largely unchanged in later versions. Thus, *Spacker* can be upgraded to newer versions with modest engineering effort.

Workload Settings. We adopt two workloads, i.e., *Stock Exchange Analysis* and *Micro Benchmark*, to cover both real-world and synthetic streaming scenarios. The former captures practical characteristics such as evolving workload behavior and periodic migration re-planning, while the latter enables controlled evaluation with tunable parameters. Together, they form a representative and complementary testbed. Detailed configurations are shown in Table I. We compare *Spacker*

TABLE II: Auto-scaling in Stock Exchange Analysis.

Strategy	Completion (ms)	Spike (ms)	Checkpoint (ms)
Spacker	**8563**	**229***	**137**
All-at-once	4489	1739	70
Progressive	21782	221	66
Replication	2412	797	220

against three representative migration solutions: *All-at-once State Migration*, *Progressive State Migration*, and *Proactive State Replication*, based on the source code and original papers of Trisk [13], Megaphone [9], and Rhino [11], respectively. During experiments, we directly measure latency spikes and completion time from runtime metrics. For system overhead, since proactive replication is executed asynchronously, it does not block stream processing but increases checkpoint duration due to backend state replication. Therefore, we use the checkpoint time to indicate the overhead incurred by state replication settings. The higher checkpoint time means higher system overhead.

B. Performance Evaluation

In this section, we conduct a series of experiments to demonstrate the advantages of *Spacker* compared to existing state migration solutions under varying scenarios.

1) Spacker for Stock Exchange: In this experiment, we demonstrate that *Spacker* can tailor-made state migration planning for real-world workloads to better help applications achieve their performance goals. We consider the auto-scaling scenario for real-time stock exchange analysis. The initial arrival rate of the quote and the traded streams is ∼40k, and it drops to ∼5k after several seconds. Thus, auto-scaling scenarios aim to find the optimal parallelism for the stock exchange analysis after the workload change. Auto-scaler [23] usually takes several continuous scaling actions to converge. In this experiment, the application is configured with 16 parallel tasks, and it takes 3 continuous scaling actions ($16 \rightarrow 4 \rightarrow 8 \rightarrow 6$) to converge to the optimal parallelism of 6. Because the stock exchange is latency-sensitive, scaling actions have to be applied with low latency spikes and completion time for fast convergence to rate changes.

Table II compares convergence time, latency spikes, and checkpoint time under different state migration solutions. In this workload, the state size to migrate is large, the number of keys to migrate is large, the state access ratio is low, and the input rate is low. Hence, *Spacker* chooses the following planning decisions: random prioritization, less progressive, and a high ratio of keys to replicate (i.e., replication 50% of keys). The results show that *Spacker* strives to achieve similar latency spikes with Progressive State Migration, which is up to 5x lower than All-at-once and Proactive State Migration, while taking a significantly shorter time (i.e., ∼2.6x) to converge to the optimal parallelism than Progressive State Migration. This indicates that *Spacker* can smoothly help change the state configuration of the stock application without affecting the upper-layer application's performance. Meanwhile, since *Spacker* does not replicate all keys, the *Spacker*'s replication time is also 40% lower, i.e., causing less system overhead,

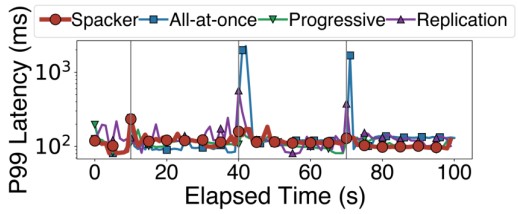
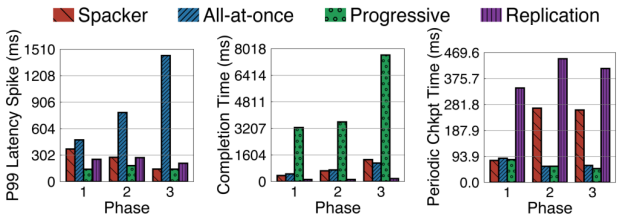


Fig. 7: Latency curve overview of dynamic workload.



(a) Latency Spikes. (b) Completion Time. (c) Checkpoint Time

Fig. 8: Detailed evaluation on dynamic workload.

compared to Proactive State Migration. This confirms the flexibility of *Spacker* to adapt state migration planning for real-world applications.

2) Spacker for Dynamic Workloads: In this experiment, we demonstrate that *Spacker* can make better performance trade-offs when applying state migration under dynamically changing workloads. We use Micro Benchmark as the base application and divide the workloads into three phases. Figure 7 compares the end-to-end latency over time, while Figure 8 compares the latency spikes, completion time, and the average checkpoint time against three existing state migration strategies. We mark the state migration triggered in each phase of dynamic workloads using grey vertical lines. Overall, the results show that *Spacker* enables flexible state migration with three-dimensional performance trade-offs. It can adaptively select the right planning decisions that achieve competitive latency spike compared to existing migration strategies while reducing up to ∼9x lower completion time than Progressive State Migration, and up to ∼3x lower system overhead than Proactive State migration.

In the first phase, the state migration is triggered to migrate 25% of the entire key space, and thus, the number of keys to migrate is low, and the state size to migrate is small. The default planning strategy in *Spacker* opts to migrate keys all at once. Therefore, *Spacker* achieves similar performance to All-at-once State Migration, completing state migration in 360ms while introducing 370ms latency spikes. This decision is appropriate since the completion time is 9x lower, while the latency spike is within the millisecond level. Additionally, it does not introduce extra system overhead, resulting in a ∼3x lower checkpoint time during normal stream processing compared to Proactive State Migration.

In the second phase, the state migration shuffles the entire key space, so that the number of keys to migrate is larger, leading to a larger state size. The default planning strategy chooses a less progressive (i.e., set chunk size to 16) and

configures a high ratio of keys to replicate (i.e., replicating 50% of keys). *Spacker* achieves a latency spike of 270ms, which is the lowest compared to the other three solutions. *Spacker* also has a 5x lower completion time compared to Progressive State Migration, and 70% lower replication overhead compared to Proactive State Migration.

In the third phase, the Zipf skew ratio of the input stream is changed to 1, meaning the workload distribution among keys is highly skewed, and state migration shuffles the entire key space to rebalance the workload among parallel tasks. The planning strategy in *Spacker* switches to the hotkey-first prioritization. *Spacker* achieves a latency spike of 140ms, which is the lowest compared to other state migration mechanisms, while the completion time is 5x lower than Progressive State Migration and the replication overhead is 60% lower than Proactive State Migration.

C. Impact of Planning Decisions

In this section, we evaluate the impact of varying planning decisions under different workload characteristics using Micro Benchmark due to its flexibility.

1) Impact of Progressiveness: We first study the impact of progressiveness on state migration. Figure 9 shows the completion time breakdown and the latency spikes for different chunk sizes on a state migration. As expected, smaller chunk sizes result in longer completion times (up to $\sim$3x) and lower latency spikes (up to $\sim$10x). By further understanding the time breakdown, we observe that smaller chunk sizes lead to both longer redistribution and synchronization time. Consistent with our analysis in Section III-A, the redistribution time increases because of the de-/serialization overhead for migrating each chunk of the state. Although our migration protocol is key-level and non-disruptive, the synchronization time can still increase because barrier processing includes performing local state snapshots for migrating keys, which increases when keys to be transferred increases.

We further study the decisions of different chunk sizes affected by *key state size* and *number of keys* to migrate as discussed in Section IV-C. Figure 13 shows the impact of selecting different chunk sizes under different key state sizes. The results show that Non-Progressive (i.e., *Batch-all*) works better with similar latency spikes and shorter completion time than Progressive State Migration when the key state size is smaller. When the key state size is larger, Less Progressive (i.e., *Chunk-16*) works better. More Progressive (i.e., *Chunk-1*) consistently achieves the lowest latency spikes but longer completion time in all cases. This is mainly because when the number of keys to migrate is large, *Chunk-1* requires more time for synchronization. To verify this, Figure 14 shows the impact of selecting different progressiveness under different numbers of keys. The results indicate that when the number of keys to migrate is small, *Chunk-1* can still achieve the lowest latency spikes while the completion time can be competitive compared to other choices.

2) Impact of State Replication: We then study the impact of state replication on state migration. Figure 10 shows the completion time breakdown and the latency spikes for different ratios of keys to replicate on a state migration. As expected, the more state of keys have been replicated proactively, the shorter completion time (up to $\sim$3x) and lower latency spikes (up to $\sim$5x) have been introduced. By further understanding the completion time breakdown, we observe that the state redistribution time is shorter when the ratio of keys to replicate is higher. This is consistent with our discussion in Section III-C1, where the replicated state can be retrieved from the local replica without remote state transfer. It is also worth noting that the synchronization time in Repl-100% is much lower compared to others. This is because the local state to be snapshotted for direct transfer reduces significantly as the state has been replicated to replicas in advance. The small portion of synchronization time and low latency spikes further confirm the efficiency of the non-disruptive migration protocol.

We further study the decisions of different ratios of keys to replicate affected by *ratio of state access* and *key state size*. Figure 15 shows the impact of selecting different ratios of keys to replicate under varying ratios of state access. The results indicate that the higher ratios of keys to replicate, e.g., *Repl-100%*, perform better with both lower latency spikes and shorter completion time when the ratio of state access is low. When the ratio of state access goes higher, the higher ratios of keys to replicate may introduce higher latency spikes and longer completion times. The different ratios of state replication eventually reach a similar performance when the ratio of state access is 100%. The results are as expected and consistent with prior work [10] because the state to be transferred directly can be larger since the size of the changed state is larger under a higher ratio of state access. Figure 16 shows the impact of selecting different ratios of keys to replicate under varying key state sizes. The results indicate that when the key state size is larger, higher ratios of keys to replicate gain more performance. This is mainly because the larger state size needs more time to transfer during state migration. Applying state replication in larger state sizes can reduce time spent on state redistribution and result in lower latency spikes and shorter completion times.

3) Impact of Prioritization: We finally study the impact of prioritization methods on state migration. In this experiment, we set the default Zipf skew ratio of the workload to 0.5 to evaluate the effectiveness of different prioritization methods. Figure 11 shows the completion time breakdown and the latency spikes for hotkey-first and random prioritization methods on a state migration. The results conclude that when the data is skewed while the input rate is high, the hotkey-first prioritization method introduces lower latency spikes (up to $\sim$50%) and shorter completion time (up to $\sim$25%) compared to the random one. By further understanding the completion time breakdown, we observe that the main performance difference comes from the synchronization time, where hotkey-first spends a shorter synchronization time. This is because control messages are queuing in data channels. Tasks that are consuming hotkeys can accumulate large amounts of records in their data channels, and thus, the queuing time for passing

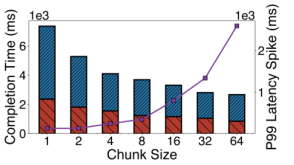


Fig. 9: Varying chunk sizes.

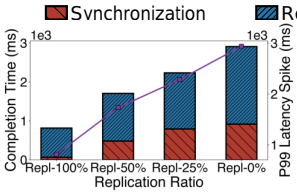


Fig. 10: Varying ratios of keys to replicate.

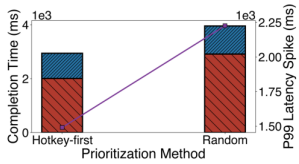


Fig. 11: Varying prioritization methods.

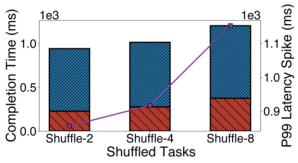
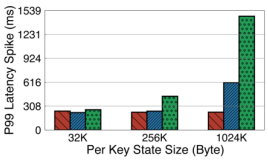
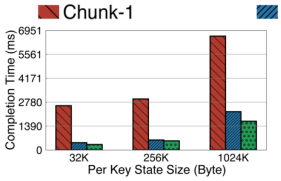


Fig. 12: Varying number of shuffled tasks.

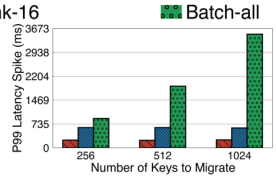


(a) Latency Spikes.

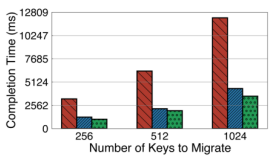


(b) Completion time.

Fig. 13: Progressiveness under varying key state size.

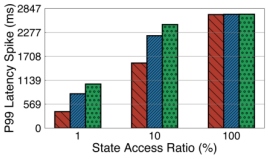


(a) Latency Spikes.

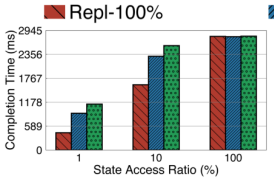


(b) Completion time.

Fig. 14: Progressiveness under varying keys to migrate.

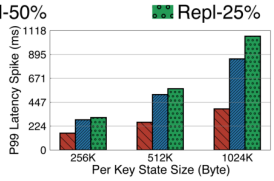


(a) Latency Spikes.

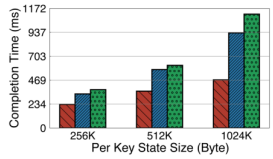


(b) Completion time.

Fig. 15: Replication under varying state access ratios.

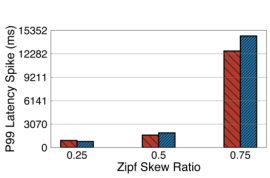


(a) Latency Spikes.

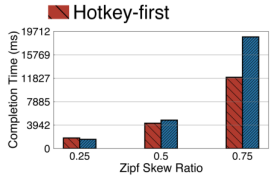


(b) Completion time.

Fig. 16: Replication under varying key state sizes.

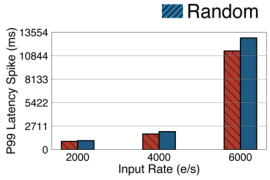


(a) Latency Spikes.

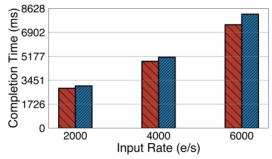


(b) Completion time.

Fig. 17: Prioritization methods under varying skewness.



(a) Latency Spikes.



(b) Completion time.

Fig. 18: Prioritization methods under varying input rates.

control messages is longer than other tasks. Migrating hotkey-first can reduce backlogs in data channels, and thus, it can achieve faster synchronization to reduce migration time and latency for state migration.

We further study the decisions of different prioritization methods affected by *skewness* and *input rate*. Figure 17 shows the impact of selecting different prioritization methods under different input data skewness. The results indicate that *hotkey-first* works better with both lower latency spikes and shorter completion time compared to others when the skewness is high, while the performance of the two prioritization methods becomes similar when skewness gets lower. This confirms our discussion in Section IV-C. Figure 18 further shows the impact of selecting different prioritization methods under different input rates. The results have shown that the hotkey-first performs better when the input rate is higher. This is because the higher input rate means the skewed parallel tasks are more likely to be overloaded, and migrating hotkeys first can reduce the workload assigned to those parallel tasks faster.

4) Impact of Number of Shuffled Tasks: We also measured the performance impact of varying numbers of shuffled tasks for *Spacker* with Flink. In our experiment setting, "n" shuffled tasks form "n" pairs of tasks to migrate state. Each pair of tasks will shuffle the same amount of state. By default, all state migration strategies are configured to *All-at-once State Migration*. Figure 12 shows varying numbers of shuffled tasks ranging from 2 to 8. Ideally, state migration of each pair of tasks can be run in parallel, and different numbers of shuffled tasks should have a similar performance. However, since the local state table update in Apache Flink needs to be executed synchronously, which is primarily due to the need to update shared metadata among assigned keys, such as "KeyGroupRange". To this end, the larger number of shuffled tasks during state migration incurs higher completion time and latency spikes. Nevertheless, we can still observe that the completion time and latency spikes grow non-linearly ($\sim$ 1.2x by comparing Shuffle-2 and Shuffle-8), confirming that state migration for different sources to destinations can run in parallel during synchronization and state transfer.

VI. Conclusion

In this paper, we propose *Spacker* to achieve configurable and efficient state migration for streaming jobs. *Spacker* de-

composes state migration into key-state operations. Subsequently, it builds an abstraction of planning strategy upon the key-level granularity, featuring three tuning knobs, that allows flexible planning of key-state operations. It executes key-state operations efficiently with a non-disruptive migration protocol. The experiment results confirm that *Spacker* supports configurable and efficient state migration for flexible performance trade-offs.

REFERENCES

[1] A. Anagnostopoulos, L. Becchetti, C. Castillo, A. Gionis, and S. Leonardi, "Online team formation in social networks," in *Proceedings of the 21st international conference on World Wide Web*, 2012, pp. 839–848.

[2] T. Leppänen, J. Riekki, M. Liu, E. Harjula, and T. Ojala, "Mobile agents-based smart objects for the internet of things," *Internet of Things based on Smart Objects: Technology, Middleware and Applications*, pp. 29–48, 2014.

[3] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at linkedin," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[4] 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*, vol. 36, no. 4, 2015.

[5] 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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 239–250.

[6] 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," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 147–156.

[7] 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*, 2013, pp. 423–438.

[8] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 46:1–46:34, 2014-03.

[9] M. Hoffmann, A. Lattuada, F. McSherry, V. Kalavri, J. Liagouris, and T. Roscoe, "Megaphone: Latency-conscious state migration for distributed streaming dataflows," *Proceedings of the VLDB Endowment*, vol. 12, no. 9, pp. 1002–1015, 2019.

[10] R. Gu, H. Yin, W. Zhong, C. Yuan, and Y. Huang, "Meces: Latency-efficient rescaling via prioritized state migration for stateful distributed stream processing systems," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 539–556.

[11] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl, "Rhino: Efficient management of very large distributed state for stream processing engines," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2471–2486.

[12] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "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*, 2013, pp. 725–736.

[13] Y. Mao, Y. Huang, R. Tian, X. Wang, and R. T. Ma, "Trisk: Task-centric data stream reconfiguration," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 214–228.

[14] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa *et al.*, "Chi: A scalable and programmable control plane for distributed stream processing systems," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1303–1316, 2018.

[15] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink: Consistent stateful distributed stream processing," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1718–1729, Aug. 2017-08.

[16] 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*, vol. 36, no. 4, 2015.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS operating systems review*. ACM, 2007, pp. 59–72.

[18] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "CIEL: a universal execution engine for distributed data-flow computing," in *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, 2011, pp. 113–126.

[19] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, "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*, vol. 8, no. 12, pp. 1792–1803, 2015.

[20] Y. Mao, J. Zhao, S. Zhang, H. Liu, and V. Markl, "Morphstream: Adaptive scheduling for scalable transactional stream processing on multicores," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–26, 2023.

[21] S. Zhang, Y. Wu, F. Zhang, and B. He, "Towards concurrent stateful stream processing on multicore processors," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1537–1548.

[22] Y. Mao, R. Yin, L. Lei, P. Ye, S. Zou, S. Tang, Y. Guo, Y. Yuan, X. Yu, B. Wan *et al.*, "Bytemq: A cloud-native streaming data layer in bytedance," in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, 2024, pp. 774–791.

[23] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "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 18)*, 2018, pp. 783–798.

[24] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: self-regulating stream processing in heron," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1825–1836, 2017.

[25] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang, "DRS: dynamic resource scheduling for real-time analytics over fast streams," in *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE, 2015, pp. 411–420.

[26] F. Kalim, L. Xu, S. Bathey, R. Meherwal, and I. Gupta, "Henge: Intent-driven multi-tenant stream processing," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 249–262.

[27] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *IEEE 35th International Conference on Distributed Computing Systems*, June 2015, pp. 399–410.

[28] Z. Wang and S. Ni, "Fries: fast and consistent runtime reconfiguration in dataflow systems with transactional guarantees," *Proceedings of the VLDB Endowment*, 2023.

[29] Y. Mao, Z. Chen, Y. Zhang, M. Wang, Y. Fang, G. Zhang, R. Shi, and R. T. Ma, "Streamops: Cloud-native runtime management for streaming services in bytedance," *Proceedings of the VLDB Endowment*, vol. 16, no. 12, pp. 3501–3514, 2023.

[30] Y. Mei, L. Cheng, V. Talwar, M. Y. Levin, G. Jacques-Silva, N. Simha, A. Banerjee, B. Smith, T. Williamson, S. Yilmaz *et al.*, "Turbine: Facebook's service management platform for stream processing," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1591–1602.

[31] P. F. Silvestre, M. Fragkoulis, D. Spinellis, and A. Katsifodimos, "Clonos: Consistent causal recovery for highly-available streaming dataflows," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1637–1650.

[32] S. Zhang, Y. Mao, J. He, P. M. Grulich, S. Zeuch, B. He, R. T. Ma, and V. Markl, "Parallelizing intra-window join on multicores: An experimental study," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2089–2101.