Fast Parallel Recovery for Transactional Stream Processing on Multicores

Jianjun Zhao†, Haikun Liu†*, Shuhao Zhang‡, Zhuohui Duan†, Xiaofei Liao†, Hai Jin†, Yu Zhang†
†National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
‡School of Computer Science and Engineering, Nanyang Technological University, Singapore
{curry_zhao, hkliu, zhduan, xfliao, hjin, zhyu}@hust.edu.cn, shuhao.zhang@ntu.edu.sg

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00122

***Abstract*—*Transactional stream processing engines* (TSPEs) have gained increasing attention due to their capability of processing real-time stream applications with transactional semantics. However, TSPEs remain susceptible to system failures and power outages. Existing TSPEs mainly focus on performance improvement, but still face a significant challenge to guarantee fault tolerance while offering high-performance services. We revisit commonly-used fault tolerance approaches in stream processing and database systems, and find that these approaches do not work well on TSPEs due to complex data dependencies. In this paper, we propose a novel TSPE called MorphStreamR to achieve fast failure recovery while guaranteeing low performance overhead at runtime. The key idea of MorphStreamR is to record intermediate results of resolved dependencies at runtime, and thus eliminate data dependencies to improve task parallelism during failure recovery. MorphStreamR further mitigates the runtime overhead by selectively tracking data dependencies and incorporating workload-aware log commitment. Experimental results show that MorphStreamR can significantly reduce the recovery time by up to 3.1× while experiencing much less performance slowdown at runtime, compared with other applicable fault tolerance approaches.**

***Index Terms*—Stream processing, Transaction, Parallel recovery, Multicore**

I. Introduction

Stream processing has been a significant research domain [1]–[4] for decades, underpinning a variety of applications such as fraud detection, dynamic car pricing, online bidding, stock trading, and real-time harvesting analysis. The evolution of stream applications [4]–[9] has increasingly necessitated supports for shared mutable states, entailing concurrent state accesses across different operators. This necessity has not been supported correctly [10] and efficiently [11]–[13] by today's *stream processing engines* (SPEs) like Storm [14], Flink [15], and Spark-Streaming [16].

Recently, *transactional stream processing* (TSP) [9], [10], [12], [13], [17] have been proposed to incorporate transactional semantics into stream processing. In the context of TSP, a set of state accesses involved in processing a single input event are modeled as a *state transaction*. As illustrated in Figure 1, a typical TSP application–Streaming Ledger [6], [9], [12] processes streaming requests of depositing/transferring money among users. Each request triggers a state transaction involving concurrent accesses to two (account, asset) tables. Thus, it is essential to correctly schedule multiple concurrent state transactions to guarantee the *order* of streaming events and ACID properties (formally discussed in Section II).

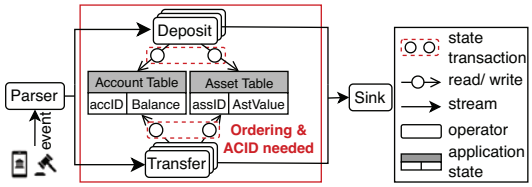


Fig. 1: Streaming Ledger: a typical application of TSP

Despite notable progress in the performance improvement of TSPEs [12], [13], there remains a challenge to guarantee fault tolerance [9] upon system failures and power outages. In some mission-critical cases such as financial services [6], [18] or healthcare [7], [19], state inconsistency or service disruption are catastrophic errors. They may put service providers at risk of financial losses, and even plunge them into potential legal disputes. Thus, it is crucial for TSPEs to support fast failure recovery without compromising the application performance.

Generally, there are two approaches to failure recovery for TSPEs: global checkpointing (CKPT) [20], [21] and logging [22]–[24]. With global checkpointing, the TSPE takes periodical application checkpoints (only including input events and application states), and reprocesses input events from the latest checkpoint in case of failures. With logging such as *write-ahead logging* (WAL), the TSPE persists log records (i.e., command logs) before state transactions are committed, and redoes command logs to recover the lost state. There have been a number of studies on parallel recovery for the logging mechanisms, such as DistDGCC (DL) [23] and Taurus (LV) [24]. These proposals track and record data dependencies at runtime, and enable parallel recovery for transactions that do not have data dependencies.

To explore whether the above fault tolerance mechanisms are effective and efficient for TSPEs, we apply them to a state-of-the-art TSPE—MorphStream [12], and measure its runtime performance and recovery time upon a system failure. Figure 2 shows the experimental results of a typical TSP application—Streaming Ledger [6]. The recovery time represents the duration in which an application recovers from

*Haikun Liu is the corresponding author.

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00122

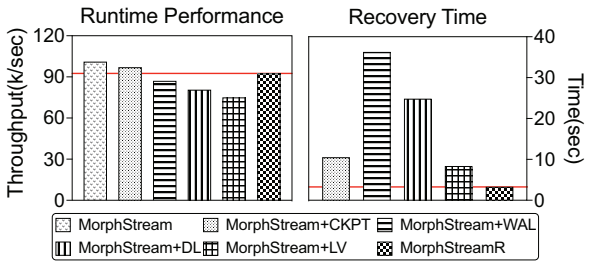


Fig. 2: Comparisons of applicable fault tolerance approaches

the latest checkpoint to the failure point. We can find that none of the applicable solutions can support fast failure recovery while guaranteeing high performance at runtime.

Since CKPT needs to reprocess all inputs after the latest checkpoint, it usually results in a long recovery time (10 seconds) on average. WAL significantly increases the recovery time to 37 seconds because it simply redoes command logs sequentially. Both DL and LV track/record data dependencies among transactions (i.e., dependency graph and vectors of logical sequence numbers, respectively), resulting in notable performance degradation at runtime. During recovery, both DL and LV redo transactions according to recorded data dependencies, and thus only transactions without data dependencies can be recovered in parallel. However, analyzing dependencies from log records, such as dependency graph reconstruction, is even more costly than reprocessing input events by MorphStream. Additionally, the task parallelism during recovery is still constrained to the inherent data dependencies among transactions, especially for workloads with high contention of state accesses. Thus, DL and LV cannot reduce the recovery time effectively, and cause even more overhead than CKPT.

In this paper, we present MorphStreamR, a high-performance TSPE that enables fast parallel recovery while guaranteeing low performance overhead at runtime. MorphStreamR is designed by fully considering the *transactional* and *streaming* features of TSP applications. First, the transactional feature usually incurs intertwined data dependencies among state transactions and thus limits the recovery parallelism. Unlike DL [23] and LV [24] that record **inter-transaction dependencies** through special data structures, the key idea of MorphStreamR is to record **intermediate results of resolved dependencies** at runtime, and directly use these results to **eliminate dependencies** during recovery. In this way, MorphStreamR can significantly improve the task parallelism during recovery, without suffering from the costly dependency resolution. This strategy offers several benefits: (i) If a state transaction is aborted due to violating the consistency property, MorphStreamR can directly abort it by checking the resolved dependencies during recovery, avoiding unnecessary computations. (ii) MorphStreamR can resolve dependencies among multiple threads to avoid lock contention during failure recovery, and thus state access operations can be linearized in independent chains and executed in parallel. (iii) MorphStreamR enables optimized task scheduling based on accurate cost estimation for each state access operation and thus improves data locality and load balancing during parallel recovery.

Second, due to the streaming feature of TSP applications, it is costly to track and persist all intermediate results of resolved dependencies at runtime. To reduce this overhead, we propose two logging mechanisms. First, we propose selective logging to diminish the logging overhead while still guaranteeing recovery efficiency. MorphStreamR only tracks dependencies across partition boundaries because the communication cost among threads is dominant. Second, we advocate workload-aware log commitment epochs to make a trade-off between the logging overhead and the recovery performance. For instance, for workloads with less state access contention, MorphStreamR adopts a longer log commitment epoch to exploit the inherent high parallelism for recovery.

To evaluate MorphStreamR, we implement several typical fault tolerance mechanisms commonly-used in contemporary stream processing and database systems, providing a broad base for comparative analysis across diverse workload types. Experimental results demonstrate that MorphStreamR outperforms all other schemes in the recovery phase for all applications while experiencing less runtime overhead. Moreover, when we increase the number of CPU cores, MorphStreamR still maintains high performance for different workloads, demonstrating significant scalability and adaptability during recovery. In summary, our contributions are as follows:

- We propose a fast parallel recovery mechanism that records intermediate results of resolved dependencies at runtime, and thus can eliminate data dependencies to improve the task parallelism during failure recovery.
- We further propose selective logging incorporated with workload-aware log commitment to mitigate the runtime overhead of logging.
- We implement MorphStreamR and evaluate it with several typical TSP applications. Experimental results show MorphStreamR can significantly reduce the recovery time by up to 3.1×, and incur much less performance slowdown at runtime, compared with other applicable fault tolerance approaches [10], [23], [24].

II. Background

In this section, we present the characteristics, programming model, and failure model of transactional stream processing.

A. Transactional Stream Processing

Transactional stream processing engines (TSPEs) process continuous streams of data while providing transactional guarantees [5]. In contrast to conventional SPEs such as Storm [14], Flink [15], and Spark-Streaming [16], TSPEs maintain shared mutable states [10]–[13], [17], which can be referenced and updated by multiple threads spawned from the same stream application. The concurrent accesses (i.e., read and write) to the shared mutable states must satisfy predefined constraints to ensure transactional semantics. In the following,

we illustrate several definitions [5], [7], [10]–[13], [17] of TSP using Streaming Ledger.

***Definition** 1 (state access operation):* A *state access operation* is a read or write operation on shared mutable states, denoted as $O_i = R_t(k)$ or $W_t(k, v)$, where t represents the timestamp when a state access operation is triggered by an input event, k denotes the state of a key to read or write, and v is the value to write. Note that v may be the value of a list of states, such as $v = f(k_1, k_2, ..., k_n)$, where f denotes a user-defined function.

***Definition** 2 (state transaction):* The set of state accesses involved in processing a single input event is defined as a state transaction, represented by txn_t=<$O_1, ..., O_n$>. Operations of the same transaction have the same timestamp.

As shown in Figure 3, the arrival of transfer event e_2 generates one state transaction txn_2=<O_2, O_3>, which contains two state access operations: O_2 and O_3. The former subtracts the transferred balance from the source account, and the latter adds the same value to the target account.

TSPEs need to ensure both ACID properties [5], [7] and stream ordering [12], [13] when scheduling state transactions. Specifically, with a set of state transactions represented as T=< $txn_{t1}, ..., txn_{tn}$ >, the schedule is correct if it is *conflict-equivalent* to $(txn_{t1} \prec ... \prec txn_{tn})$, where $\prec$ represents that the left operand precedes the right one. A key objective of scaling transactional stream processing is to maximize system concurrency while maintaining a correct schedule. This poses a substantial challenge due to the inter- and intra-dependencies among state transactions. There are mainly three types of data dependencies, as shown in Figure 3:

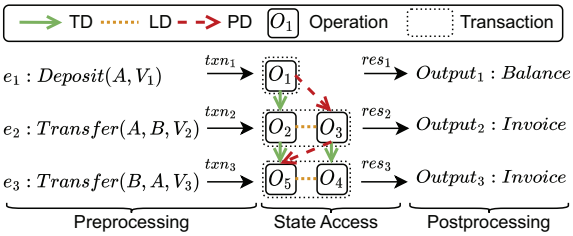
Temporal Dependency (TD): O_i temporally depends on O_j if they belong to different state transactions but access the same state, and O_i has a larger timestamp than O_j. For example, O_2 temporally depends on O_1 as they access the same record with different timestamps.

Logical Dependency (LD): O_i and O_j exhibit logical dependency if they belong to the same state transaction. For example, in txn_2 and txn_3, the transfer must change both accounts at a time. Hence, O_3 and O_5 logically depend on O_2 and O_4, and vice versa.

Parametric Dependency (PD): $O_i = W(k_i, v)$, with $v = f(k_1, k_2, ..., k_m)$, parametrically depends on $O_j = W(k_j, v')$ if $k_j \neq k_i$, $k_j \in k_1, k_2, ..., k_m$, and O_i has a larger timestamp. For example, keys handled in O_1 and O_3 are A and B, respectively. Whether O_3 can be performed depends on a user-defined function, such as guaranteeing that the transferring amount (V) is less than the balance of the source account (A). Therefore, O_3 parametrically depends on O_1 because O_3 relies on the state of A, as indicated by $f_3(B, A, V_2)$.

B. Programming Model

Many variants of TSPEs [5], [7], [10]–[13], [17] have been proposed in the last decade. The common programming model can be abstracted in a *three-step procedure*, as shown in Figure 3. These three steps are recursively conducted for every batch of input events. (i) *preprocessing*, where



TS	State Access Operation	Function (f)
1	$O_1 = W_1(A, f_1(V_1))$	$f_1 : R_1(A) + V_1$
2	$O_2 = W_2(A, f_2(A, V_2))$	$f_2 : R_2(A) - V_2$ if $R_2(A) > V_2$
2	$O_3 = W_2(B, f_3(B, A, V_2))$	$f_3 : R_2(B) + V_2$ if $R_2(A) > V_2$
3	$O_4 = W_3(B, f_4(B, V_3))$	$f_4 : R_3(B) - V_3$ if $R_3(B) > V_3$
3	$O_5 = W_3(A, f_5(A, B, V_3))$	$f_5 : R_3(A) + V_3$ if $R_3(B) > V_3$

Fig. 3: An illustration of TSP using Streaming Ledger

input events are transformed into state transactions, including deterministic read/write sets; (ii) *state access*, where all state access operations are performed, such as updating the balance in the target account; (iii) *postprocessing*, where the input events are further processed based on the results of state access operations, such as generating an invoice after the balance has been transferred. Subsequently, the corresponding output is passed to downstream operators.

C. Failure Model

In this paper, we focus on shared-memory multicore architectures, specifically addressing failures that cause a single-node stoppage [20], [25]. In this setup, the node is connected to external sources/sinks through a reliable network and has access to storage that survives failures. Upon failures, operators need to recover from the failure point.

In the context of fault tolerance in TSP, two crucial requirements must be met upon failures: (i) *Delivery guarantees*: this means that an incoming event will apply its effects to the computation state of the system exactly once and is reflected in the final output. (ii) *Correctness guarantees*: this means that state transactions triggered by events must be correctly scheduled. As shown in Figure 3, event e_1 should be processed exactly once, including the execution of txn_1 and the delivery of corresponding $Output_1$. Moreover, the scheduling of state transactions T=< $txn_1, txn_{t2}, txn_{t3}$ > should adhere to the correctness criteria as mentioned in Section II-A.

III. Revisit Fault Tolerance for TSPEs

This section revisits single-node fault tolerance mechanisms and explores their limitations when they are applied to TSPE.

A. Global Checkpointing

Global checkpointing [20], [21], [26] persists operator states periodically and performs a global rollback in case of a failure. Consistent checkpoint coordination is achieved by checkpoint barriers, which periodically traverse the operators from sources to sinks. Upon receiving barriers from all upstream operators, each operator stores its state in durable storage and propagates

the barrier to downstream operators. Besides operator states, global checkpointing mechanisms also persist all events at the input of the workflow. Upon failure, all operators restore their states from the last checkpoint and reprocess lost input events.

Limitations. Global checkpointing is a simple-yet-effective approach to guarantee fault tolerance for TSPEs, but its recovery time is constrained by the extensive computation of reprocessing lost input events. Although frequent checkpoints can mitigate the overhead of reprocessing input events, this approach compromises the runtime performance due to increased I/O overhead.

B. Logging

Logs [22]–[24], [27], [28] are widely-used by in-memory *database management systems* (DBMSs) to guarantee the durability of transactions. DBMSs persist log records before transactions are committed and replay them to recover the lost state upon a system crash. A naive logging mechanism usually results in extremely long recovery time due to sequential log replay. Thus, dependency tracking algorithms such as DistDGCC [23] and Taurus [24] are proposed to improve the recovery parallelism.

However, DistDGCC and Taurus can not adapt to TSPEs directly because they cannot guarantee exactly-once delivery during recovery due to the streaming processing feature. Specifically, upstream operators only redo committed transactions to restore the application state, without regenerating any output during recovery. As a result, downstream operators may fail to recover the following transaction correctly due to missing input from upstream operators. To adapt DistDGCC and Taurus to TSP systems, we group all state transactions triggered by a single input event across the streaming topology and commit them together. Since most TSP workloads [10], [12], [13] have only a few operators in the streaming topology, the group commit incurs trivial overhead. However, DistDGCC and Taurus are still inefficient during runtime and recovery due to their inherent limitations as follows.

High Performance Overhead at Runtime. DistDGCC tracks dependencies among updated data, including incoming and outgoing edges. However, since the size of log records increases linearly with the number of dependencies, it results in more computation and storage overhead for complex dependencies in TSP. Taurus presents a lightweight parallel logging approach that encodes inter-transaction dependencies into a vector of *logical sequence numbers* (LSNs), but incurs significant computation overhead at runtime.

Inefficient of Failure Recovery. During recovery, DistDGCC reconstructs the dependency graph from log records, while Taurus checks and updates the global recovery LSN vector to guarantee a partial order among transactions. Therefore, multiple transactions without data dependencies can be recovered in parallel. Nevertheless, these approaches are not cost-effective for high-performance TSPEs [12], [13] because the cost of rebuilding dependencies or frequently checking the LSN vector is even higher than reprocessing input events. Moreover, the task parallelism during recovery is still constrained to inherent data dependencies among transactions.

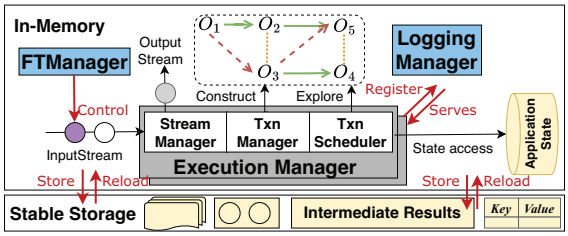


Fig. 4: System architecture of MorphStreamR

IV. MorphStreamR Overview

In this paper, we design a high-performance TSPE called MorphStreamR. The goal of MorphStreamR is to achieve fast failure recovery while guaranteeing low performance overhead at runtime. We propose two novel designs to achieve these goals: 1) **a fast parallel recovery mechanism** by exploiting a set of dependency resolutions, and 2) **selective logging and workload-aware log commitment** to mitigate the runtime overhead of logging. We implement MorphStreamR based on a state-of-the-art TSPE, MorphStream [12]. Figure 4 shows an overview of MorphStreamR, which consists of a set of Execution Managers, a Logging Manager, and a Fault-tolerance Manager.

Execution Manager (**EM**). EM is the core component of TSPE, consisting of three components: StreamManager, TxnManager, and TxnScheduler. MorphStreamR adopts a dual-phase approach [12], [13], including both stream and transaction processing. In the stream processing phase, StreamManager conducts preprocessing and postprocessing for each batch of input events. State transactions are issued during preprocessing but are not immediately processed. The TxnManager identifies dependencies among these state transactions and constructs a *task precedence graph* (TPG). In this graph, vertices correspond to state access operations, and edges represent fine-grained dependencies between these operations. The stream processing phase periodically transforms to the transaction processing phase, controlled by punctuation markers [29]. During the transaction processing phase, TxnScheduler explores the TPG to perform concurrent operations. Finally, input events are further processed based on the results of state access operations to generate outputs for downstream operators.

Logging Manager (**LM**). LM is responsible for recording intermediate results of resolved dependencies in a log file implemented as a hash table. LM carefully determines what and when to log based on the characteristics of the workload to reduce runtime overhead (Section VI). During recovery, LM provides dependency inspection services for EMs to achieve various dependency-aware recovery optimizations, such as aborting pushdown, operations restructuring, and optimized task assignment (Section V).

Fault-tolerance Manager (**FM**). FM orchestrates global checkpointing, logging, and recovery processes within the

system. It employs markers [29], a strategy used in previous stream processing systems, to synchronize EMs and LM. FM introduces these markers with adaptable intervals to facilitate the coordination. Upon receiving a marker, all EMs perform operations within the same epoch, including creating global checkpoints and committing logs. FM also ensures the persistence of input events to prevent data loss upon failures. In this way, the system can recover from the failure point and guarantee data consistency.

V. Efficient Recovery from Failures

In this section, we first discuss dependency inspection. Next, we detail dependency-aware recovery optimizations. Finally, we describe our recovery protocol.

A. Dependency Inspection

MorphStreamR records intermediate results of resolved dependencies among state transactions to achieve parallel recovery. We argue that the limited parallelism of state transactions during recovery mainly stems from *logical dependencies* (LDs) and *parametric dependencies* (PDs). Since state access operations with *temporal dependencies* (TDs) target the same state, threads can process operations following the TDs in parallel, with each thread targeting a different state. However, LDs and PDs represent dependencies between different states, requiring additional communications and synchronizations among threads to resolve these dependencies. More importantly, LDs and PDs can be eliminated using deterministic results. Based on these observations, MorphStreamR tracks LDs and PDs and records the following intermediate results during runtime to enable fast parallel recovery and ensure data consistency.

1) Results of logical dependencies: LDs implies that aborting one state access operation often leads to aborting all state access operations in the same state transaction. Therefore, we record the unique identifier of the aborted state transaction. This enables MorphStreamR to efficiently handle transaction aborts during recovery and reduce the overhead due to retrying operations and ensuring ACID properties.

2) Results of parametric dependencies: PDs implies that the update to a value depends on the execution of another operation. To guarantee deterministic computations and data consistency during recovery, MorphStreamR has to obtain the same value as before the failure. Therefore, we record the intermediate results of parametric dependencies at runtime.

The data structures of the intermediate results are illustrated in Figure 5. As explained in Section IV, input events are divided into epochs using punctuation markers. Consequently, the `AbortView` and `ParametricView` are organized into segments corresponding to these punctuation markers (`Epoch_ID`). The transaction ids (`Txn_ID`) of aborted transactions are directly stored in the `AbortView`. Each `ParametricView` contains the necessary information to resolve dependencies and can be referenced using the (`From_key, To_key`) pair. For example, assume there are two operations $O_1 = Write(A)$ and $O_2 = Write(B, v)$,

AbortView	
Epoch_ID	Txn_ID

ParametricView				
Epoch_ID	Txn_ID	From_key	To_key	Result

Fig. 5: Data structures of intermediate results

where $v = f(A)$. Thus, O_2 has a PD on O_1. Once the dependency is resolved, we record the intermediate result of $f(A)$, which can be referenced using the (A, B) pair.

B. Dependency-aware Recovery Optimization

By utilizing dependency inspection, MorphStreamR enables several optimizations to improve the parallelism of state transactions during recovery as follows.

1) Abort Pushdown: Abort pushdown is a recovery optimization technique for aborting transactions early if these transactions would abort eventually during their normal executions. MorphStreamR directly aborts input events based on the intermediate results of resolved LDs (`AbortViews`). The abort pushdown mechanism offers several advantages: (i) *Simplified error handling:* As mentioned in Section II-B, the processing of each event involves three phases: preprocessing, state access, and postprocessing. The transaction aborting is usually handled during the state access phase. However, if we discard input events that eventually lead to transactions aborting before the preprocessing stage, MorphStreamR can avoid the subsequent unnecessary computations in the pipeline. (ii) *Improved parallelism during recovery:* Abort pushdown eliminates the need to identify and verify LDs among state access operations, and thus further improves parallelism during recovery and reduces synchronization operations between threads. (iii) *Reduced overhead:* By identifying and discarding input events that could lead to transaction aborts in advance, MorphStreamR can efficiently eliminate the computation overhead of transaction rollbacks and redo operations.

2) Operation Restructuring: It is essential to guarantee the correct order of state transaction processing during the recovery of TSPEs. However, unresolved dependencies can impede state access operations and limit their parallelism. To address this issue, MorphStreamR records intermediate results of resolved dependencies at runtime and directly retrieves the result to resolve dependencies, without communication among threads. As a result, state access operations can be rearranged into separate chains that can be executed by threads without lock contention.

Figure 6 illustrates the operation restructuring during the recovery process. When transactions arrive, MorphStreamR converts them into atomic state access operations and identifies dependencies. Since MorphStreamR has already filtered the events that would lead to transaction aborts through abort pushdown, it can eliminate LDs ($O_2 \rightarrow O_3$ and $O_4 \rightarrow O_5$) for these transactions. When PDs ($O_1 \rightarrow O_3$ and $O_3 \rightarrow O_5$) need to be resolved, operations can directly retrieve the intermediate results from `ParametricView`. Then, these operations are partitioned based on the target state keys and inserted into a sorted list known as chains. Operations in each chain

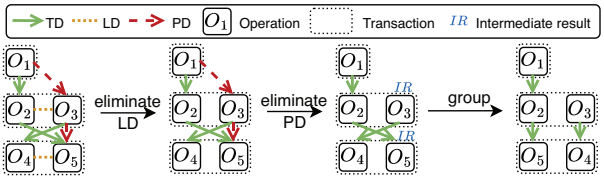


Fig. 6: Restructuring state access operations

are arranged according to their timestamps to facilitate the identification of TDs among operations. Finally, operations are restructured into two independent sets (O_1, O_2, O_5) and (O_3, O_4). In this way, the two chains can be executed in parallel.

3) Optimize Task Assignment: MorphStreamR efficiently assigns tasks among worker threads via two steps: grouping state access operations and task assignment.

Grouping state access operations: Using abort pushdown and operation restructuring mechanisms, there are only TDs among state access operations. Thus, we group state access operations within a single chain into a single task to improve data locality. This approach ensures that state access operations for a given state are executed by one thread, thereby reducing cache misses and improving performance.

Task assignment: MorphStreamR can effectively eliminate potential transaction aborts or arbitrary synchronization among threads by using abort pushdown and operation restructuring techniques. As a result, the task execution time is mainly determined by the number of state access operations performed by each thread. In our implementation, we utilize a greedy algorithm to achieve a fair task assignment among threads. The algorithm starts by sorting tasks according to their weights which are equivalent to the number of state access operations. Subsequently, it iterates over the sorted tasks in a decreasing order and greedily assigns each task to worker threads with the minimum workload in that iteration. This process continues iteratively till all tasks are assigned.

C. Recovery Protocol

MorphStreamR exploits the above dependency inspection and dependency-aware recovery optimization techniques to improve the recovery parallelism. Figure 7 shows the recovery process of MorphStreamR. We highlight the steps of our recovery protocol in red.

Restore from Checkpoints: During recovery, the Fault-tolerance Manager loads the metadata of global checkpoints from stable storage ❶. Subsequently, MorphStreamR restores all executors which then restore application states from the latest checkpoint ❷.

Construct Intermediate Results: To support dependency-aware recovery optimization, the *Logging Manager* (LM) constructs the intermediate results from the log records before replaying lost input events ❸. It delivers two key services by using these historical records: it enables *Execution Managers* (EMs) to verify whether a state transaction would abort eventually, allowing for handling potential aborts in advance. Moreover, it allows EMs to access the intermediate results of resolved dependencies to eliminate PDs.

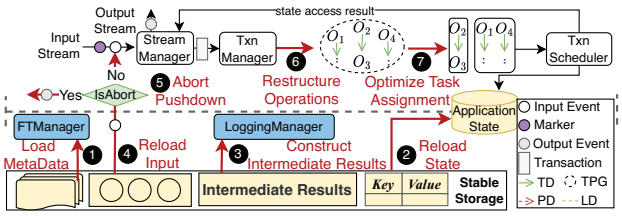


Fig. 7: The workflow of MorphStreamR during recovery

Replay Input Events: When MorphStreamR replays input events, the dependency-aware recovery optimizations discussed in Section V-B are applied. MorphStreamR reloads the input events from a specified offset according to the latest checkpoint ❹. Then, EM handles all transactions that would abort in the future by checking the resolved LDs ❺. During the preprocessing phase, EM does not track LDs for the remaining transactions and accesses the intermediate results to eliminate PDs. Therefore, operations are restructured into independent sets ❻. Subsequently, MorphStreamR assigns tasks to EMs based on optimized task assignment ❼, and EMs perform all state accesses in parallel during the state access phase.

VI. Efficient Runtime

To reduce the performance overhead at runtime, MorphStreamR exploits selective logging to determine which dependencies should be recorded, and leverages workload-aware log commitment to determine how logs are committed.

A. Selective Logging

MorphStreamR selectively tracks data dependencies across partition boundaries. Since these dependencies lead to a lot of communication overhead among threads, MorphStreamR can reduce the size of logs while maintaining recovery efficiency by eliminating these dependencies. To achieve selective logging, MorphStreamR partitions state access operations into different groups at runtime and handle dependencies within partitions during recovery as follows.

1) Graph-based Partitioning: To determine which data dependencies should be tracked, MorphStreamR has to partition state accesses into different groups. We should make a careful trade-off between the logging cost and the recovery time by considering two objectives: (i) ensuring that each worker thread receives a similar amount of workload, and (ii) reducing the number of dependencies that need to be recorded. To achieve these goals, MorphStreamR takes into account data locality by treating each chain of state access operations (according to TDs) as a vertex, where the weight of the vertex corresponds to the number of operations. Similarly, the weight of an edge represents the number of LDs and PDs between two chains. This graph partitioning is a well-known problem [30], and we utilize a greedy algorithm proposed by Yao et al. [31] to partition the graph. This algorithm ensures load balancing across partitions and reduces the number of dependencies among partitions.

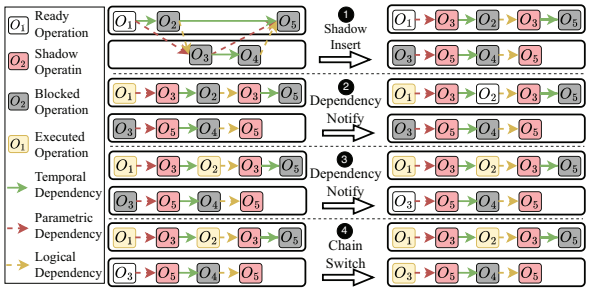


Fig. 8: Shadow-based exploration

Then, MorphStreamR only tracks and maintains dependencies among partitions at runtime.

2) Shadow-based Exploration: Since the selective logging approach only records inter-partition dependencies, we still have to resolve intra-partition dependencies during recovery. Thus, MorphStreamR introduces shadow operations as placeholders for operations with unresolved dependencies within a partition. Each shadow operation represents a dependency for a specific operation and is inserted into the chain which its dependent operations belong to. We set the condition-variable-check [30] as the first state access operation on which other operations in the same state transaction are logically dependent.

During operation restructuring (Section V-B2), PDs and LDs among partitions are speculatively resolved using intermediate results. For an operation with dependencies that lack intermediate results, MorphStreamR inserts shadow operations into the chains of its dependent operations. Operations in these chains are sorted based on timestamps to accurately resolve TDs. Shadow operations are inserted into the place where the operations they depend on. Note that, shadow operations do not introduce new dependencies as they merely serve as placeholders for unresolved dependencies. During recovery, when MorphStreamR encounters a shadow operation, it reduces the count of dependencies associated with that operation by one, indicating that one of the dependencies of this operation has been resolved. If an operation still has unresolved dependencies in another operation chain, MorphStreamR turns to process the corresponding chain until all operations have been executed.

For instance, there are five operations involved in two operation chains in Figure 8. For PD ($O_1 \rightarrow O_3$) and LD ($O_2 \rightarrow O_3$), MorphStreamR inserts two shadow operations for O_3 after O_1 and O_2, respectively. Likewise, MorphStreamR inserts two shadow operations for O_5 after O_3 and O_4, respectively ❶. After constructing the operation chains, MorphStreamR initiates execution by selecting the operation chain starting with O_1. As O_1 is executed, MorphStreamR encounters the shadow operation of O_3 and reduces the count of dependencies for O_3 ❷. After O_2 has been executed, all dependencies of O_3 are resolved. Thus, O_3 is marked as a ready operation ❸. Next, MorphStreamR identifies that O_5 depends on O_3 and O_4, and these dependencies remain unresolved. Accordingly, MorphStreamR processes

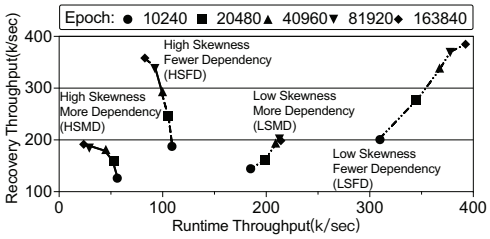


Fig. 9: Runtime vs. recovery throughput under different log commitment epochs

the corresponding operation chain (O_3, O_4) to resolve the remaining dependencies ❹.

B. Workload-Aware Log Commitment

Since different workload characteristics have an impact on the TSP performance during runtime and recovery, we propose workload-aware log commitment to make a trade-off between the logging overhead and the recovery efficiency.

1) Exploiting Workload Characteristics: To accurately profile workload characteristics, two critical factors are taken into account: (i) the skewness of state accesses. We model it using a Zipfian [12], [13] distribution, i.e., certain states may be accessed more frequently than others. (ii) the number of data dependencies. The ratio of different transaction types usually affects the number of data dependencies. For example, a larger proportion of state transactions involving multiple partitions may result in more PDs. To characterize these factors for dynamic workloads, some techniques such as time series analysis [32] and machine learning based prediction [33] are required for online profiling. Since these techniques are not the primary focus of this paper, we intend to study this problem in the future.

2) Adaptive Log Commitment: We further propose an adaptive log commitment mechanism to make a trade-off between the logging overhead and the recovery performance. Figure 9 shows the TSP performance during runtime and recovery under various log commitment epochs, in which a number of events are successfully handled. These curves represent distinct levels of state access contention for the same workload. The marker position represents the performance corresponding to different log commitment epochs. Based on experimental results and theoretical analysis, we can make a tradeoff between the runtime overhead and the recovery efficiency as follows.

For workloads with *low data contention* (LSFD), a larger epoch is beneficial for the performance of both runtime and recovery. This is because state accesses are uniformly distributed. Thus, larger epochs allow to batch more operations for commitment, resulting in improved performance. However, for workloads with *low skewness and more dependencies* (LSMD), large epochs do not necessarily improve the recovery performance. This is because larger epochs result in more overhead in indexing intermediate results for dependency inspection, which offsets the benefit of group commit.

For workloads with *high data contention* (HSMD and HSFD), MorphStreamR shows inverse performance trends

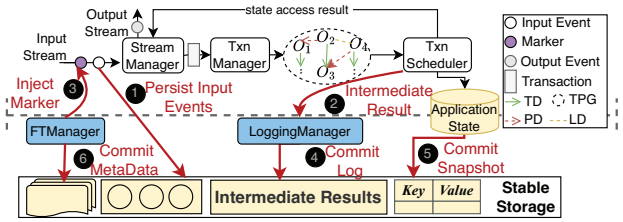


Fig. 10: The workflow of MorphStreamR during runtime

during runtime and recovery when the commitment epoch increases. MorphStreamR shows worse runtime performance with larger epochs because skewed state accesses often result in load imbalance. With smaller epochs, operations have a potential to be distributed more evenly, and thus the system performance is improved. However, large epochs are still effective for improving the recovery efficiency because MorphStreamR has more opportunities to explore task parallelism through dependency-aware optimizations.

C. Runtime Operations

Figure 10 illustrates the workflow (highlighted in red) of MorphStreamR during runtime. It includes the following steps:

Persist Input Events: In this step, a spout is responsible for persisting input events in batches ❶. These events are stored in a reliable storage system before further processing. This enables correct recovery of TSPEs from the failure point, avoiding data loss and ensuring data consistency.

Record Intermediate Results: State transactions are issued during the stream processing phase but are not immediately processed. The *Execution Manager* (EM) first identifies dependencies among these state transactions and further processes them during the transaction processing phase. The *Logging Manager* (LM) is responsible for recording the intermediate results of resolved dependencies. When a dependency is resolved, the corresponding EM delivers the intermediate result to the LM ❷.

Orchestrate FM, EM, and LM: The *Fault-tolerance Manager* (FM) injects markers ❸ in reconfigurable intervals to orchestrate FM, EM, and LM. These markers include three types: (i) transaction marker, which controls the transition between stream processing and transaction processing; (ii) commit marker, which notifies the LM to persist the intermediate results; and (iii) snapshot marker, which commands the database to take a snapshot of its current state. By default, the transaction marker and commit marker are aligned. To adjust the log commitment epoch based on data contention levels, MorphStreamR generates commit markers with different frequencies, as discussed in Section VI-B.

Checkpoint/Log Commit and Garbage Collection: To optimize the I/O bandwidth of SSDs, EMs utilize non-blocking APIs with asynchronous notifications to persist snapshots and intermediate results. This allows EMs to continue the post-processing since no state access operations are involved during this phase. The intermediate results can be immediately committed ❹ after persistence as they are only used as a reference during recovery. However, the snapshot must wait until all subsequent processing is completed before being committed ❺. Once all outputs are generated, the EM notifies the FM to commit the snapshots. FM waits for notifications from all EMs before committing the snapshot ❻. The recorded input events and intermediate results can be deleted upon the completion of the current checkpoint.

VII. Discussions

In this paper, we focus on fault tolerance for TSPEs in shared-memory multicore architectures. However, adaptation of TSPEs and MorphStreamR to distributed environments needs careful considerations for distributed execution models, failure-recovery models, and logging overhead.

There are several challenges to adapt TSPEs to distributed environments. 1) In multicore architectures, MorphStreamR relies on dependency graph based scheduling mechanisms to achieve high-performance state transactions. However, in distributed environments, the dependency construction and resolution of distributed state transactions are much more costly due to high overhead of inter-node communication. 2) Since applications' states are shared by multiple TSPEs in distributed environments, caching becomes more essential due to the high latency of remote state accesses. However, it is challenging to implement a high-performance software-managed DRAM cache because guaranteeing cache coherence is extremely expensive in distributed environments. One potential approach to these challenges is to leverage high-throughput and low-latency RDMA networks for fast access to distributed shared application states. Moreover, programmable RDMA switches [34] can be exploited for in-network locking operations, and thus can significantly reduce the cache coherence overhead and mitigate network congestion.

There are also several challenges to achieve fast failure recovery for TSPEs in distributed environments. As application states are stored in the local memory of each node and shared across distributed nodes, it is challenging to guarantee data consistency among failed nodes and surviving nodes. 1) When considering distributed shared states, global checkpointing [35] becomes more complex compared with traditional distributed SPEs because application states distributed on different nodes may be updated asynchronously by distributed state transactions. It is challenging to achieve high performance at runtime while guaranteeing transaction-consistent global checkpoints [36]. 2) When a failed node is restoring, the shared mutable state may have been updated by other surviving nodes, resulting in nondeterministic outputs. It is challenging to achieve high availability while guaranteeing strong data consistency. One potential approach to these challenges is to leverage lineage-based [37] or causal-based [38] logging mechanisms. We plan to study fast fault-tolerance mechanisms for distributed TSPEs in the future.

In distributed environments, the logging overhead at runtime can be magnified significantly. To address this issue, we plan to explore the following mechanisms: 1) removing the logging operation from the critical path of task execution, like Lineage

Stash [37]; 2) leveraging emerging computational storage devices [39] for log compression/decompression/truncation, and garbage collection; 3) exploring RDMA [40] networks and persistent memory [41], [42] to accelerate log persisting.

VIII. Evaluation

In this section, we evaluate MorphStreamR by comparing it with several fault tolerance protocols. The source code and experiments in this paper are available at Github https://github.com/CGCL-codes/MorphStreamR [43].

A. Methodology

Hardware Platform. We run all experiments on a 2-socket Intel Xeon Gold 5220 server with 512GB of memory and a local 480GB Intel Optane SSD (write bandwidth: 2GB/s, IOPS:146k). Each socket contains thirty-six 2.20GHz cores and 24.75 MB of LLC cache. By default, all experiments are conducted using cores on a single socket.

Comparisons. The state-of-the-art MorphStream [12] does not support fault tolerance. To make a comprehensive and fair comparison with MorphStreamR (MSR), we implement several fault-tolerance mechanisms and apply them to MorphStream [12].

Native (NAT). We use the native MorphStream [12] as a baseline, serving as a performance upper bound during runtime.

Global Checkpointing (CKPT). We implement the commonly used global checkpointing mechanism and apply it to MorphStream.

Write-Ahead Logging (WAL). We apply command logging to MorphStream because it can lower the pressure on I/O [22].

LSN Vector (LV). Like Taurus [24], we encode inter-transaction dependencies into a vector of logic sequence numbers to preserve partial orders between dependent transactions during recovery.

Dependency Logging (DL). We also implement the logging scheme of DistDGCC [23], which records fine-grained dependency graphs [23]. Each log record contains dependency information such as incoming and outgoing edges. During recovery, the dependency graph is rebuilt first, and then the log is replayed.

Benchmarks. We use the following representative benchmarks [12], [13] for a comprehensive evaluation.

Streaming Ledger (SL) represents a real-world stream application suggested by a recent commercial TSPE [6]. It transfers money and assets between accounts, including *transfer* state transactions that transfer balances between user accounts and assets tables, and *deposit* state transactions that top up user accounts or assets.

Grep and Sum (GS) represents a workload scenario where an application modifies shared mutable states [12], [13]. Each *Sum* state transaction reads a list of states and writes the summation results back to the first one.

Toll Processing (TP) simulates traffic on roads divided into segments, and calculates tolls based on congestion [18]. Two mutable tables record the average traffic spend of a road segment and the counts of unique vehicles. Each vehicle report triggers one *toll processing* state transaction that modifies related records (road speed and the counts of unique vehicles) and calculates the toll.

Workload Characteristics. These typical workloads cover a wide range of TSP features. Specifically, SL has a relatively high number of dependencies, where state access operations may be parametrically dependent on each other. GS shows a more skewed workload compared with others. TP represents a workload where transaction aborting is common. By choosing workloads with different characteristics, we can comprehensively evaluate the effectiveness of our techniques, which are designed to handle various types of workloads.

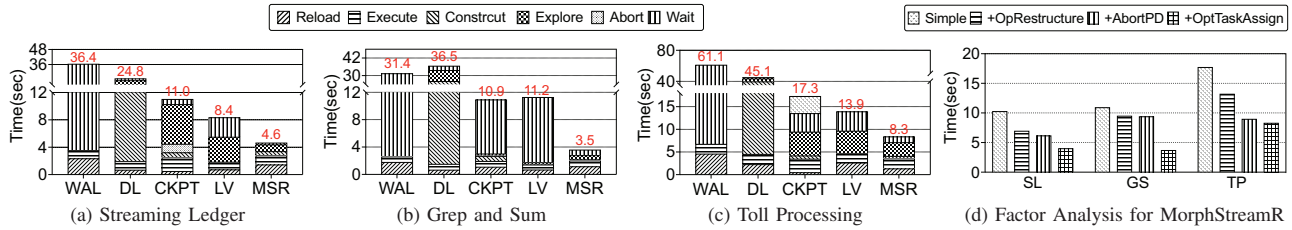
B. Recovery Performance

In this section, we evaluate the recovery time of different fault-tolerance mechanisms.

Recovery Time. As shown in Figure 11, MorphStreamR demonstrates a significant reduction of recovery time compared with other schemes for all applications. It reduces the recovery time by 3.1 times for GS, 1.7 times for TP, and 1.8 times for SL, compared with sub-optimal approaches. In the following, we provide the recovery time and factor analysis to give a deeper understanding of the benefits of various dependency-aware recovery optimizations.

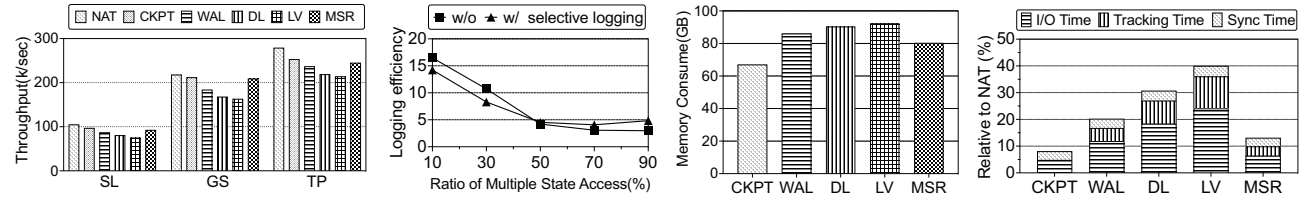
Breakdown Analysis. We show the breakdown of the recovery time for the following operations. 1) *Reload Time* refers to the time spent on reloading states, input events, and log records. 2) *Execute Time* refers to the time spent on performing state access operations and user-defined computations. 3) *Construct Time* refers to the time spent on identifying dependencies and constructing the auxiliary data structures. 4) *Abort Time* refers to the time spent on handling state transactions aborts. 5) *Explore Time* refers to the time spent on exploring available operations to process. 6) *Wait Time* refers to the time spent on synchronization, including potential waiting time due to load imbalance.

Figure 11 shows the execution time of different operations during recovery for different applications. We have four major observations. First, WAL exhibits the longest *wait time* among various recovery techniques due to sequentially redoing command logs. Surprisingly, WAL also spends the longest time on *reloading*. Our further experiments reveal the reason is that all threads need to ensure a global order of committed logs, and thus a lot of time is spent on sorting. Second, LV and DL pose other problems while improving the task parallelism during recovery. DL spends substantial time on *constructing* dependency graphs because it needs to identify data dependencies among a large number of log records. Also, LV exhibits higher *explore time* for SL due to a large number of data dependencies. This is because it heavily relies on the inherent parallelism of workloads. In contrast, MorphStreamR can further improve the task parallelism during recovery by using the intermediate results of resolved dependencies, leading to minimal *explore time* in all workloads. Third, MorphStreamR benefits from



(a) Streaming Ledger (b) Grep and Sum (c) Toll Processing (d) Factor Analysis for MorphStreamR

Fig. 11: Subfigure (a), (b), (c) show the execution time of different operations during recovery for different approaches using three TSP applications. Subfigure (d) presents the recovery time when different optimizations of MorphStreamR are incrementally added from left to right.



(a) Runtime Throughput (b) Impact of Selective Logging (c) Maximum Memory Footprint (d) Runtime Overhead

Fig. 12: Subfigure (a) shows the runtime throughput of different systems, and subfigure (b) shows the effectiveness of selective logging using SL. Subfigure (c) and (d) show the overhead of MorphStreamR using SL.

abort pushdown, which eliminates the time used to handle transaction *aborts* during recovery. Fourth, MorphStreamR can achieve optimal task assignment and scheduling because logical dependencies and parametric dependencies are eliminated. This is particularly useful for load-imbalanced applications such as GS because MorphStreamR substantially eliminates the straggler problem.

Factor Analysis. In this study, we conduct factor analysis to identify the key operations that affect the recovery efficiency. `Simple` denotes running MorphStreamR without recovery optimizations. `+OpRestructure` represents restructuring state access operations that use the intermediate results to resolve dependencies (Section V-B2). `+AbortPD` further reduces the recovery time of handling transaction aborts (Section V-B1). `+OptTaskAssign` further adopts our optimized task assignment (Section V-B3). We note that these optimizations are incrementally added. Figure 11d shows the impact of each optimization on the recovery performance for different workloads. For workloads (SL) with a large amount of dependencies, operation restructuring (`+OpRestructure`) yields the largest performance improvement by reducing the number of dependencies and improving task parallelism. In contrast, for skewed workloads (GS), the optimized task assignment (`+OptTaskAssign`) offers a significant performance improvement by distributing tasks evenly. At last, for workloads with a high number of transaction aborts (TP), the abort pushdown (`+AbortPD`) mechanism delivers a significant performance improvement by early aborting transactions.

C. Runtime Performance

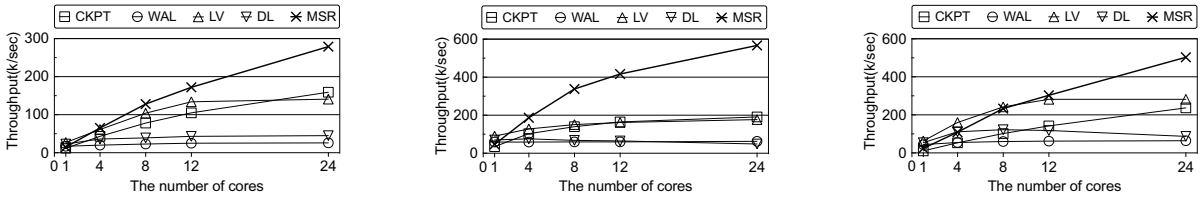
In this section, we compare the runtime performance of various fault-tolerance schemes and validate the effectiveness of selective logging.

Runtime Performance. Figure 12a shows that CKPT leads to the least performance overhead at runtime as it does not record any log. However, it results in more time spent in failure recovery, as shown in Figure 11. MorphStreamR incurs acceptable runtime overhead, about 12.1% and 5.4% performance degradation compared with NAT and CKPT, respectively. Compared with other log-based approaches (WAL, DL, LV), MorphStreamR can improve the runtime throughput by up to 30% because the logging overhead is significantly reduced.

Effectiveness of Selective Logging. To assess the efficiency of selective logging, we introduce a new metric–logging efficiency, which is calculated as the recovery performance improvement divided by the runtime performance degradation. A higher value implies higher recovery performance with less impact on the runtime performance. Figure 12b shows the results of our evaluation, comparing MorphStreamR with and without selective logging for workloads with varying numbers of dependencies. When the proportion of multi-partition transactions is relatively low (e.g., 10% to 50%), the logging efficiency without selective logging is higher because the overhead of logging (e.g., tracing and serializing log records) is not significant in scenarios with fewer dependencies, making the algorithmic overhead of selective logging more prominent. However, when the proportion of multi-partition transactions increases, the number of dependencies in the workload also increases. In such cases, it is beneficial for selectively logging critical dependencies using an algorithmic approach because it reduces the size of log records.

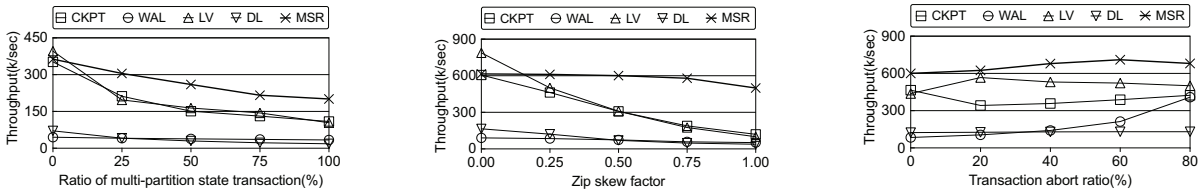
D. Overhead Analysis

In this section, we evaluate the overhead introduced by MorphStreamR in terms of memory consumption and additional execution time.



(a) Streaming Ledger (b) Grep and Sum (c) Toll Processing

Fig. 13: Scalability comparison among different approaches



(a) Impact of Multi-partition Transactions (b) Impact of State Access Skewness (c) Impact of Aborting Transactions

Fig. 14: Workload sensitivity study

Memory Footprint. In this study, we use SL to evaluate the system's memory footprint. Figure 12c shows the maximum memory consumption of SL with different fault-tolerance mechanisms at runtime. MorphStreamR, LV, and DL increase about 20%, 38%, and 35% more memory footprints than CKPT. MorphStreamR shows lower storage overhead than other mechanisms since our logging mechanisms can reduce the size of logs. Moreover, since the memory resource occupied by logs is reclaimed periodically by the JVM garbage collector [12], the maximum memory footprint of MorphStreamR is acceptable.

Runtime Overhead. Figure 12d shows the runtime overhead relative to the native execution (NAT). It includes the following factors: 1) *I/O Time* refers to the time spent on serializing and persisting important data, such as log records, and application state. 2) *Tracking Time* refers to the time spent on tracking dependencies and constructing log records required for recovery. 3) *Sync Time* refers to the time spent on synchronization for a global consistent snapshot and log commitment. We have two major observations. First, selective logging can significantly reduce the I/O and tracking time due to the reduced number of intermediate results that need to be recorded. In contrast, LV incurs the most I/O and tracking overhead as it has to track all dependencies and generate LSN vectors. Second, I/O overhead is still the major performance bottleneck for all mechanisms. Advanced data compression techniques [20], [44], [45] can be exploited to reduce the cost and memory consumption of I/O operations.

E. Scalability Study

In this section, we evaluate the scalability of MorphStreamR as the number of cores increases. Figure 13 shows the recovery performance (input events recovered per second) of different applications under various approaches. First, MorphStreamR scales effectively across all applications with an increasing number of cores. This is because, MorphStreamR can restructure the operations into independent chains by dependency inspection, which can be evaluated in parallel without lock contention. Second, CKPT demonstrates good scalability in low-contented workloads, such as little parametric dependencies (TP) or uniformly distributed state access (SL), benefiting from adaptive scheduling strategies [12]. However, it is bounded by synchronization overhead in high-contented workloads (GS), which hinders recovery performance as the number of cores increases. Third, when the number of cores is low, WAL, DL, and LV perform slightly better than MorphStreamR, especially for TP. When only one core is available, both approaches execute state transactions sequentially. However, MorphStreamR involves constant overhead due to dependency-aware recovery optimization. Fourth, LV does not exhibit good scalability across all workloads. As the number of cores increases to a certain degree, it is constrained by the inherent parallelism of the workload such as imbalance state access (GS) or complex dependencies among state transactions (SL).

F. Workload Sensitivity Study

In this section, we evaluate the impact of workload characteristics using Grep&Sum due to its flexibility.

Impact of Multi-Partition State Transaction. To isolate the impact of multi-partition transactions, we set the state access skew factor to 0 and exclude aborting transactions. The results are shown in Figure 14a with two key observations. First, as the ratio of multi-partition transactions increases, MorphStreamR outperforms other approaches because dependency inspection mitigates the impact of increased exploration overhead caused by parametric dependencies among partitions. Second, as the ratio of multi-partition transactions further increases, the performance of MorphStreamR degrades. This is because the construction of intermediate results becomes more time-consuming as more dependencies need to be tracked. The additional

overhead in constructing intermediate results offsets the benefits of MorphStreamR's dependency inspection, resulting in decreased performance.

Impact of State Access Skewness. We conduct a write-only workload in this study, where the ratio of multi-partition state transactions is 0 and no transaction aborts is encountered. Figure 14b shows the impact of varying state access skewness. First, LV performs best when state accesses are uniformly distributed due to its low overhead in preserving recover order using LSN vectors. In contrast, MorphStreamR and CKPT require auxiliary data structures for scheduling, resulting in higher overhead. Second, MorphStreamR is tolerant to state access skewness, while LV and CKPT perform worse as skewness increases due to load imbalance. This is because MorphStreamR can optimize task assignment, resulting in improved overall performance. Third, varying the state access skewness has a minor effect on system performance under DL and WAL. This is because their performance are dominated by other overheads. As shown in Figure 11, DL spends more time constructing dependency graphs, while WAL has to redo command logs in sequential order.

Impact of Aborting Transactions. Figure 14c shows the throughput during recovery as the percentage of events triggering abort transactions changes from 0% to 80%. First, CKPT experiences an initial performance drop when the percentage of aborting transactions increases (e.g., 20%), due to the overhead of redoing transactions. However, as the percentage further increases, the overall throughput of CKPT gradually improves as the number of aborted operations reduces the redo time. Second, MorphStreamR has better performance than other approaches by discarding input events that would lead to aborted transactions early. However, this performance improvement is not always guaranteed (e.g., 80%) as it involves overhead of reloading and checking the intermediate results. Third, the performance of WAL improves with an increasing percentage of aborting transactions since it only needs to redo the committed log records.

IX. Related Work

TSP has been increasingly studied in recent years [5], [7], [10]–[13], [17]. Previous TSPEs mainly focus on performance optimization of stream transactions, and very few efforts have been made on the fault-tolerance. For instance, S-Store [10] adopts the checkpoint and command-log implemented in a DBMS–H-Store [46]. TSpoon [11] incorporates write-ahead logging into Flink's global checkpointing. Some TSPEs such as T-Stream [13] and MorphStream [12] even do not adopt any fault-tolerance mechanism. To the best of our knowledge, MorphStreamR is the first study on fault-tolerance of TSPEs.

Despite little work related to fault tolerance for TSPEs, there have been many fault tolerance mechanisms for SPEs. Some SPEs such as Storm [14] and Heron [47] provide best-effort fault tolerance, with at-most-once or at-least-once processing semantics for each event. Other SPEs like AF-Stream [48], [49] provide approximate fault-tolerance for specific scenarios such as data synopsis and online machine learning. However, both best-effort and approximate fault tolerance mechanisms can not guarantee ACID properties of state transactions because they may lose states or input data upon system failures. A few SPEs such as Flink [15], [21] and Spark Streaming [50] support exactly-once semantics. When their fault tolerance mechanisms are applied to TSPE, they often result in a long recovery time due to complex data dependencies among state transactions. Thus, these approaches are impractical when directly applying them to TSPEs. In contrast, MorphStreamR also guarantees exactly-once semantics and further offers several dependency resolution techniques to achieve fast parallel recovery.

Recently, a number of dependency tracking algorithms [23], [24] have been proposed to improve the recovery parallelism in DBMSs. Taurus [24] uses a vector of logical sequence numbers to encode inter-transaction dependencies and maintains their partial orders during recovery. DistDGCC [23] speeds up system recovery by logging fine-grained dependency graphs. However, since they only record dependencies among transactions, the application performance during recovery is limited by the inherent parallelism of workloads, especially for workloads with high state access contention. In contrast, MorphStreamR records the intermediate results of resolved dependencies to eliminate data dependencies, and thus further exploits task parallelism during recovery.

X. Conclusion

TSPEs are vulnerable to system crashes and power outages, especially for long-term and compute-intensive data streams. In this paper, we propose MorphStreamR to achieve fast failure recovery while guaranteeing low performance overhead at runtime. By recording intermediate results of resolved dependencies at runtime, MorphStreamR introduces dependency-aware execution optimizations for recovery, including abort pushdown, operations restructuring, and optimized task assignment. Moreover, MorphStreamR exploits selective logging and workload-aware log commitment to reduce the runtime overhead. Experimental results demonstrate that MorphStreamR can significantly improve the recovery performance at scale for various workloads, compared with state-of-the-art fault-tolerant mechanisms.

Acknowledgment

This work is supported by National Key Research and Development Program of China under grant No.2022YFB4500303, and National Natural Science Foundation of China under grants No.62072198, 62332011, 62302178, and Natural Science Foundation of Hubei Province under grant No.2021CFA037, and Huawei Grant (No.YBN2021035018A7). Shuhao Zhang's work is supported by a MoE AcRF Tier 2 grant (MOE-T2EP20122-0010), and a startup grant of NTU (023452-00001). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] A. Jayarajan, W. Zhao, Y. Sun, and G. Pekhimenko, "TiLT: A Time-Centric Approach for Stream Query Optimization and Parallelization," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023, pp. 818–832.

[2] S. Zhang, H. T. Vo, D. Dahlmeier, and B. He, "Multi-Query Optimization for Complex Event Processing in SAP ESP," in *Proceedings of the 2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, 2017, pp. 1213–1224.

[3] R. C. 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 International Conference on Management of data (SIGMOD)*, 2013, pp. 725–736.

[4] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos, "Beyond Analytics: The Evolution of Stream Processing Systems," in *Proceedings of the 2020 ACM International Conference on Management of Data (SIGMOD)*, 2020, pp. 2651–2658.

[5] I. Botan, P. M. Fischer, D. Kossmann, and N. Tatbul, "Transactional Stream Processing," in *Proceedings of the 15th International Conference on Extending Database Technology (EDBT)*, 2012, pp. 204–215.

[6] Serializable ACID Transactions on Streaming Data, 2018, https://www.ververica.com/blog/serializable-acid-transactions-on-streaming-data.

[7] D. Wang, E. A. Rundensteiner, H. Wang, and R. T. Ellison III, "Active Complex Event Processing: Applications in Real-Time Health Care," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, pp. 1545–1548, 2010.

[8] O. C. Sahin, P. Karagoz, and N. Tatbul, "Streaming Event Detection in Microblogs: Balancing Accuracy and Performance," in *Proceedings of the International Conference on Web Engineering (ICWE)*, 2019, pp. 123–138.

[9] S. Zhang, J. Soto, and V. Markl, "A Survey on Transactional Stream Processing," *The VLDB Journal (VLDBJ)*, vol. 33, pp. 451–479, 2024.

[10] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang, "S-Store: Streaming Meets Transaction Processing," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, pp. 2134–2145, 2015.

[11] L. Affetti, A. Margara, and G. Cugola, "TSpoon: Transactions on a Stream Processor," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 140, pp. 65–79, 2020.

[12] Y. Mao, J. Zhao, S. Zhang, H. Liu, and V. Markl, "MorphStream: Adaptive Scheduling for Scalable Transactional Stream Processing on Multicores," in *Proceedings of the 2023 ACM International Conference on Management of Data (SIGMOD)*, 2023, pp. 1–26.

[13] S. Zhang, Y. Wu, F. Zhang, and B. He, "Towards Concurrent Stateful Stream Processing on Multicore Processors," in *Proceedings of 2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1537–1548.

[14] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@ Twitter," in *Proceedings of the 2014 ACM International Conference on Management of Data (SIGMOD)*, 2014, pp. 147–156.

[15] P. C. A. Katsifodimos, S. E. V. Markl, and S. H. K. Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering (Bull. IEEE Comput. Soc. Tech. Commmon. Data Eng.)*, vol. 36, pp. 28–38, 2015.

[16] 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 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 423–438.

[17] L. Affetti, A. Margara, and G. Cugola, "FlowDB: Integrating Stream Processing and Consistent State Management," in *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems (DEBS)*, 2017, pp. 134–145.

[18] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear Road: A Stream Data Management Benchmark," in *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, 2004, pp. 480–491.

[19] Monitor and Prevent Healthcare-Associated Infections, 2023, https://www.cdc.gov/infectioncontrol/iicb/hai.html.

[20] G. Theodorakis, F. Kounelis, P. Pietzuch, and H. Pirk, "Scabbard: Single-Node Fault-Tolerant Stream Processing," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 15, pp. 361–374, 2021.

[21] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 10, pp. 1718–1729, 2017.

[22] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker, "Rethinking Main Memory OLTP Recovery," in *Proceedings of the 2014 IEEE 30th International Conference on Data Engineering (ICDE)*, 2014, pp. 604–615.

[23] C. Yao, M. Zhang, Q. Lin, B. C. Ooi, and J. Xu, "Scaling Distributed Transaction Processing and Recovery based on Dependency Logging," *The VLDB Journal (VLDBJ)*, vol. 27, pp. 347–368, 2018.

[24] Y. Xia, X. Yu, A. Pavlo, and S. Devadas, "Taurus: Lightweight Parallel Logging for In-Memory Database Management Systems," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 14, pp. 189–201, 2020.

[25] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys (CSUR)*, vol. 34, pp. 375–408, 2002.

[26] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A Timely Dataflow System," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 439–455.

[27] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks using Write-Ahead Logging," *ACM Transactions on Database Systems (TODS)*, vol. 17, pp. 94–162, 1992.

[28] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy Transactions in Multicore In-Memory Databases," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 18–32.

[29] P. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting Punctuation Semantics in Continuous Data Streams," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 15, pp. 555–568, 2003.

[30] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal (BSTJ)*, vol. 49, pp. 291–307, 1970.

[31] C. Yao, D. Agrawal, G. Chen, Q. Lin, B. C. Ooi, W.-F. Wong, and M. Zhang, "Exploiting Single-Threaded Model in Multi-Core In-Memory Systems," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 28, pp. 2635–2650, 2016.

[32] K. Choi, J. Yi, C. Park, and S. Yoon, "Deep Learning for Anomaly Detection in Time-Series Data: Review, Analysis, and Guidelines," *IEEE Access*, vol. 9, pp. 120 043–120 065, 2021.

[33] A. Akbar, A. Khan, F. Carrez, and K. Moessner, "Predictive Analytics for Complex IoT Data Streams," *IEEE Internet of Things Journal (IoT-J)*, vol. 4, pp. 1571–1582, 2017.

[34] Q. Wang, Y. Lu, E. Xu, J. Li, Y. Chen, and J. Shu, "Concordia: Distributed Shared Memory with In-Network Cache Coherence," in *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, 2021, pp. 277–292.

[35] D. Manivannan, R. H. B. Netzer, and M. Singhal, "Finding Consistent Global Checkpoints in a Distributed Computation," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 8, pp. 623–627, 1997.

[36] M. Haubenschild, C. Sauer, T. Neumann, and V. Leis, "Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines," in *Proceedings of the 2020 ACM International Conference on Management of Data (SIGMOD)*, 2020, pp. 877–892.

[37] S. Wang, J. Liagouris, R. Nishihara, P. Moritz, U. Misra, A. Tumanov, and I. Stoica, "Lineage Stash: Fault Tolerance Off the Critical Path," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019, pp. 338–352.

[38] P. F. Silvestre, M. Fragkoulis, D. Spinellis, and A. Katsifodimos, "Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows," in *Proceedings of the 2021 ACM International Conference on Management of Data (SIGMOD)*, 2021, pp. 1637–1650.

[39] A. Barbalace and J. Do, "Computational Storage: Where Are We Today?" in *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, 2021.

[40] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl, "Rethinking Stateful Stream Processing with RDMA," in *Proceedings of the 2022 ACM*

International Conference on Management of Data (SIGMOD), 2022, pp. 1078–1092.

[41] C. Ye, Y. Xu, X. Shen, Y. Sha, X. Liao, H. Jin, and Y. Solihin, "Reconciling Selective Logging and Hardware Persistent Memory Transaction," in *Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 664–676.

[42] Z. Duan, H. Lu, H. Liu, X. Liao, H. Jin, Y. Zhang, and S. Wu, "Hardware-supported remote persistence for distributed persistent memory," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021, pp. 1–14.

[43] MorphStreamR, 2024, https://github.com/CGCL-codes/MorphStreamR.

[44] P. Damme, A. Ungethüm, J. Hildebrandt, D. Habich, and W. Lehner, "From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms," *ACM Transactions on Database Systems (TODS)*, vol. 44, pp. 1–46, 2019.

[45] X. Zeng and S. Zhang, "Parallelizing Stream Compression for IoT Applications on Asymmetric Multicores," in *Proceedings of the 2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 950–964.

[46] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-Store: A High-performance, Distributed Main Memory Transaction Processing System," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 1, pp. 1496–1499, 2008.

[47] 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 International Conference on Management of Data (SIGMOD)*, 2015, pp. 239–250.

[48] Q. Huang and P. P. Lee, "Toward High-performance Distributed Stream Processing via Approximate Fault Tolerance," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 10, pp. 73–84, 2016.

[49] Z. Cheng, Q. Huang, and P. P. Lee, "On the Performance and Convergence of Distributed Stream Processing via Approximate Fault Tolerance," *The VLDB Journal (VLDBJ)*, vol. 28, pp. 821–846, 2019.

[50] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark," in *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, 2018, pp. 601–613.