Scaling Stream Processing with Transactional State Management on Multicores

Shuhao Zhang[1]*, Yingjun Wu*[2]*, Feng Zhang*[3]*, Bingsheng He*[1]
[1]*National University of Singapore,* [2]*IBM Almaden Research Center,* [3]*Renmin University of China*

Abstract

Transactional state management relieves users from managing state consistency during stream processing by themselves. This paper introduces `TStream`, a highly scalable data stream processing system (DSPS) with built-in transactional state management. `TStream` is specifically designed for modern shared-memory multicore architectures. `TStream`'s key contribution is a novel asynchronous state transaction processing paradigm. By detaching and postponing state accesses from the stream application computation logic, `TStream` minimizes unnecessary stalls caused by state management in stream processing. The postponed state accesses naturally form a batch, and we further propose an operation-chain based execution model that aggressively extracts parallelism opportunities within each batch of state access operations guaranteeing consistency without locks. To confirm the effectiveness of our proposal, we compared `TStream` against four alternative designs on a 40-core machine. Our extensive experiment study show that `TStream` yields much higher throughput and scalability with limited latency penalty when processing different types of workloads.

1 Introduction

Data stream processing systems (DSPSs) are gaining their popularities in powering modern IoT (Internet-of-Things) and data streaming applications [15, 42, 21, 1, 3, 28]. Thanks to the popularity of modern commodity machines with massively parallel processors, researchers and practitioners nowadays can perform ultra-fast stream processing on a single multicore machine [46, 27, 32, 47, 44]. Unfortunately, existing DSPSs did not pay much attention to *transactional state management*, which aims to provide built-in application state storage and retrieval capability with consistency guarantee.

DSPS with transactional state management relieves users from managing state consistency by themselves, and has recently received attention from both academia [31, 7] and industry community [4, 5]. However, scaling stream processing while providing transactional state management is challenging. On the one hand, to achieve both low latency and high throughput, DSPSs are designed to process multiple input events at the same time in order to aggressively exploit parallelism [46, 47, 32, 27]. On the other hand, processing different events concurrently may lead to conflict accesses (reads and writes) to the same application state, hence leading to higher chances of violating transactional state consistency [16]. To make things worse, more than simply guaranteeing the ACID properties preserved in the relational database systems, DSPSs further need to enforce the state access order according to the input event timestamp order. The intuition behind is that event happens earlier should make effect to the world earlier than events happen later.

Due to the complexity, today's DSPSs often force user to select from either strict state consistency guarantee with limited concurrency or high concurrency with limited (or no) state consistency guarantee (e.g., Storm [3], Flink [1]). Recent studies on transactional state management in DSPSs [31, 16, 14, 38, 7] typically choose the prior one. Based on a synchronized execution model, these works perform state maintenance within the stream application computation logic. There are two major issues with this design. Firstly, executing the state access operations strictly following the dependencies encoded in the computation logic can cause extra long execution stalls in stream processing as operators must wait for state access to finish, consequently degrading the overall performance. Secondly, enforce ordering of each state access with synchronization primitives (e.g., monotonically increasing counter and locks) severely limits system concurrency in processing state accesses preventing prior systems from scaling well.

Witnessing limitations of prior works, we propose `TStream`, a new DSPS that can support highly scalable stream processing with transactional state consistency guarantee on shared memory multicore architectures. `TStream` achieves high performance via two key designs:

(1) To resolve the first limitation, `TStream` performs its internal state management asynchronously by detaching

Table 1: Summary of Terminologies

Term	Definition
Event	Input stream event with a monotonically increasing timestamp
Operator	Basic unit of a stream application, continuously process event streams; can be carried by multiple execution instance (i.e., threads)
Shared states	Mutable states that are concurrently accessible by multiple execution instance of an operator
State transaction	A set of state access operations triggered by a single input event
ACID+O	A consistency property satisfying ACID properties and event timestamp ordering
Watermark	A special event guaranteeing no future event has smaller timestamp than it

state access operations from the application computation logic. Through postponing state accesses, `TStream` minimizes unnecessary stalls caused by state management in stream processing.

(2) `TStream` adopts *watermark* as periodic signal to trigger the actual transaction process. Postponed state accesses between two subsequent watermarks hence naturally form a batch, and the batch length can be then tuned to tradeoff latency and throughput. To take advantage of modern multicore architectures, `TStream` adopts an *operation-chain* based execution model that aggressively extracts parallelism opportunities within each batch of state transactions. We further apply evaluation pushdown and NUMA-aware placement to improve its processing efficiency.

To confirm `TStream`'s effectiveness, we compared it against four alternative schemes on a high-performance 40-core machine. Our extensive experimental study shows that `TStream` achieves more than two times higher throughput on average over existing solutions with similar or even smaller processing latency.

We organize the paper as follows: Section 2 introduces the computational model of DSPS with transactional state management. Section 3 reviews limitations of existing solutions. Section 4 discusses design of `TStream`. We report extensive experiment results in Section 6. Section 7 reviews related works and Section 8 concludes this paper.

2 Stream model and background

In this section, we discuss the stream processing model and terminology through a running example. We summarize terminologies in Table 1.

We describe the execution model of stream processing with a general definition [32]. We briefly present here for completeness. Stream processing continuously processes on one or more streams of *events* E. Each event $e_{ts} \in E$ has a timestamp ts that indicates its temporal sequence. For simplicity, we assume events arrived at the system has a monotonically increasing timestamp. A streaming application contains a sequence of *operators* $O = O_1, O_2, ..., O_n$. Each operator continuously processes events and optionally emits events. To sustain high input stream ingress rate, each operator can be further carried in multiple *execution instances* (e.g., Java threads), which handle multiple input events concurrently.

To support complex real-time analytics, operator may need to maintain large *shared mutable states* [38, 14, 31] (e.g., user account) sharing among all of its execution instances. As a result, any uncoordinated accesses to the same state (e.g., update and read the same account at the same time) can cause state inconsistencies. This problem is exacerbated if more complex state storage and retrieval queries such as scan and range lookup is required, which is overlooked in most existing DSPSs, such as Storm [3], Flink [1]. The usage of shared mutable states in stream processing are prevalent and has been studied in several prior works [38, 14, 31, 7].

Definition 1 *We define the set of state access operations triggered by a single input event e_{ts} in an operator as one **state transaction** txn_{ts}. The timestamp ts of a state transaction is assigned to be the same of its triggering event.*

As multiple state transactions may be generated concurrently, state transaction execution must also satisfy `ACID` properties, similar to conventional database transaction.

- *Atomicity:* state access operations of one state transaction shall be processed all at once or none.
- *Consistency:* the state changes as a result of evaluating state transaction shall not violate any state constraint. For example, user account can not become negative.
- *Isolation:* multiple conflicting state transactions shall run independently without interfering each other.
- *Durability:* the changes made to the state shall be made persistent.[1]

A *concurrency control* (CC) protocol can be used to serialize state transactions to ensure `ACID` properties for state transactions. However, DSPSs additionally need to ensure that the resulting serial order follows the input event timestamp ordering [38, 14, 31, 7].

Definition 2 *We define a consistency property satisfying both `ACID` properties and event timestamp ordering constraint as `ACID+O` properties. Formally, a DSPS under `ACID+O` needs to ensure the state transaction schedule must be conflict equivalent to $txn_1 \prec ... \prec txn_n$.*

[1] In this work, we assume shared states can be kept in memory and we defer the durability in future work.

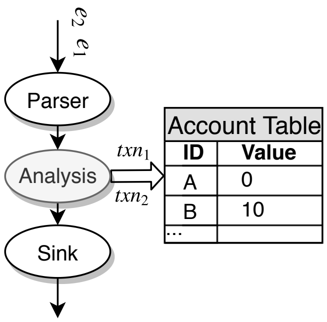


Figure 1: Running example application.

step	txn_1	txn_2
1	lockX(A)	
2	write(A)	
3		lockS(B)
4		read(B)
5		unlock(B)
6	lockX(B)	
7	write(B)	
8	unlock(B)	
9	unlock(A)	

Figure 2: Example trace under 2PL.

We use a simplified application as depicted in Figure 1 as a running example. The original application processes events describing wiring money and assets between different user accounts. Detailed descriptions can be found in [4] and are omit here for brevity. The running example contains the three operators: 1) `Parser` parses received input events and pass the processed events to `Analysis`. 2) `Analysis` performs some analytics on the received events, and optionally need to read or update to some user accounts, which has to be shared among all instances of `Analysis`. The analysis results are further passed to `Sink`. 3) `Sink` reports the processing results to end users.

Processing each input event serially in their timestamp order eliminates the challenge of preserving state consistency (e.g., SEI [38]). For example, one can restrict the `Analysis` operator to have only one execution instance (e.g., single thread). However, it severely restricts system concurrency. To achieve both low latency and high throughput, DSPSs need to exploit parallelism aggressively, and operators are typically carried by multiple instances. However, there is a *potential conflict* between those concurrently running instances as they may access the same user account at the same time. As a result, higher concurrency also intensifies the state access collision.

Although this is a pretty common problem in traditional database system, we are targeting at a streaming context, where state accesses are triggered by input streaming events rather than user. Naively locking the state or relying on a third-party database to manage states not only degrades the performance but also leads to consistency violation [16]. For example, a read operation triggered by an event shall never see an updated state that is modified by a write operation triggered by *future* events. The intuition behind is that event happens earlier should make effect to the world earlier than events happen later. Unfortunately, such state access ordering guarantee is not well-supported in conventional database systems.

We use an example execution trace to illustrate the failure of simply using a conventional database to manage shared states. Let us assume that two events e_1 and e_2 (e_1 has a smaller timestamp) are concurrently processed by two instances of `Analysis` independently. Assume the processing of e_1 triggers an update of A and B, i.e., $txn_1 : W(A), W(B)$, while e_2 triggers a read of B, i.e., $txn_2 : R(B)$.

Conventional CC protocols serialize transactions in an order that is conflict-equivalent to *any* certain serial schedule. Taking 2-phase locking (2PL) [10] as an example, a possible schedule is shown in Figure 2. The resulting serial order is $txn_2 \prec txn_1$, which leads to a wrong reading of B in processing e_2. In contrast, system under `ACID+O` properties ensures the processing of e_2 successfully obtain the correct updated value of B due to e_1.

3 Existing Solutions and Limitations

Scaling stream processing with transactional state management is challenging and has motivated a number of related systems [38, 7, 14, 4, 31]. The fundamental challenge is that the state access operations triggered by one event is unaware of the access pattern of operations concurrently triggered by other events.

Prior solutions [7, 31, 14, 38] can confront severe scaling bottlenecks, primarily due to the requirement of tracking order of each state transaction. Figure 3 compares how different algorithms process three state transactions ($txn_3 : R(C)$) of our running example shown previously.

Lock Ahead 2PL (LAL). An earlier work from Wang et al. [38] described a strict two-phase locking (S2PL) based algorithm that allows multiple state transactions run concurrently while maintaining state consistency. Extended from the original S2PL [10], it relies on a *lock-ahead* mechanism to make sure that locks are granted in the correct order (see the `Lock(order)` in Figure 3(a)) in order to preserve the aforementioned order-preserving property. As the lock-ahead operations from different transactions have to be performed serially, LAL severely restrict system concurrency.

Low Water Mark (LWM). Wang et al. [38] propose an improved version of LAL by adding multi-versioning, called *low water mark (LWM)*. LWM leverages a status variable (i.e., `lwm`) of each state to guard the processing sequence: write must be performed monotonically (by sequentially increasing `lwm` during transaction commits), and will not be blocked by read locks; a read may not be blocked by write locks as long as it is able to read the correct version of shared states. That is, the read operation has a timestamp larger than `lwm` of the corresponding state so it can read an *readily* version of state. As shown in Figure 3(b), `LockS(B)` will not be blocked by txn_1's lock on `B`, which is its major improvement compared to LAL. However, `Read(B)` has to wait for `lwm` to be updated by txn_1 during its commit. Furthermore, LWM still requires the

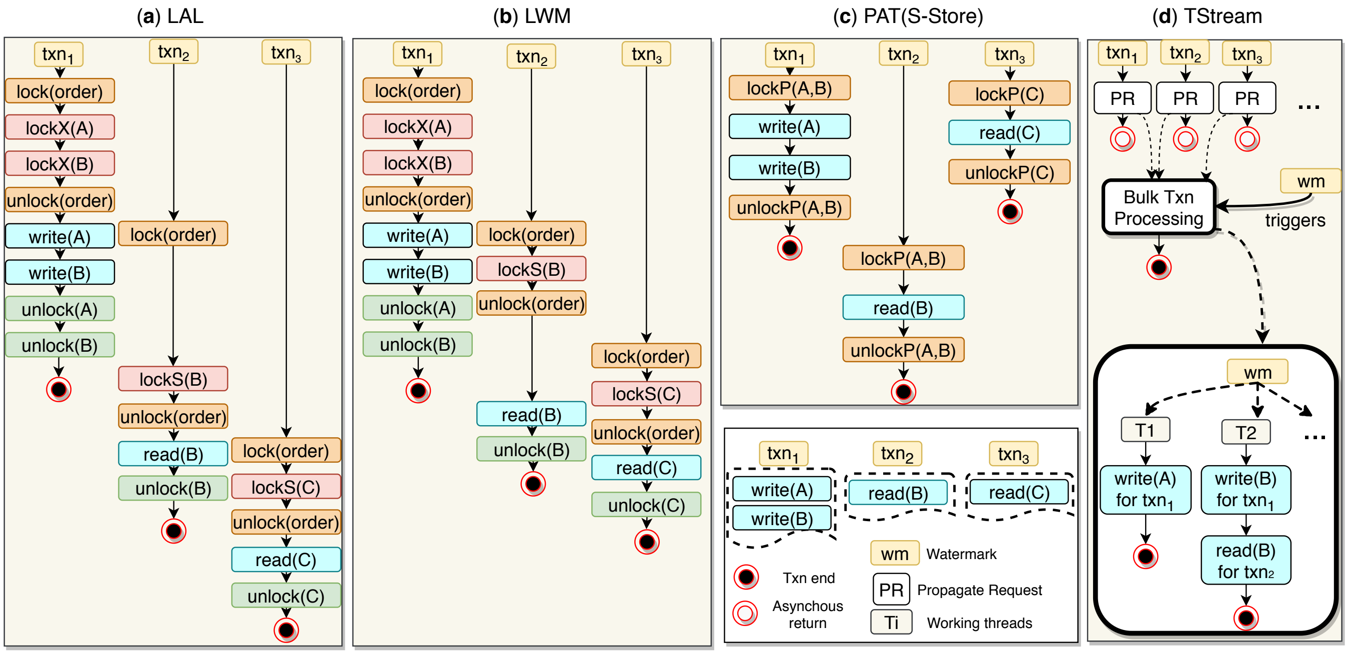


Figure 3: Examples of how different algorithms process three state transactions.

`Lock(order)` mechanism to ensure locks are inserted at a correct order following event sequence.

Partition-based approach (PAT). S-Store [31] fuses OLTP and streaming into one engine, which is built based on H-Store – a shared-nothing deterministic database system [26]. Similar to how H-Store splits database, S-Store splits the shared states into multiple disjoint *partitions*, and hence only needs to guard accessing order in each partition. Unfortunately, the original implementation of S-Store uses a single-thread [31] for all shared state accesses, which severely limits system concurrency. We hence implement a mechanism called *PAT* to represent the similar idea of S-Store without such restriction. To guarantee the state consistency, PAT requires partition-level locks (*LockP*) to synchronize the accesses to each partition, which acts like an array of $lock(order)$ of multiple partitions.

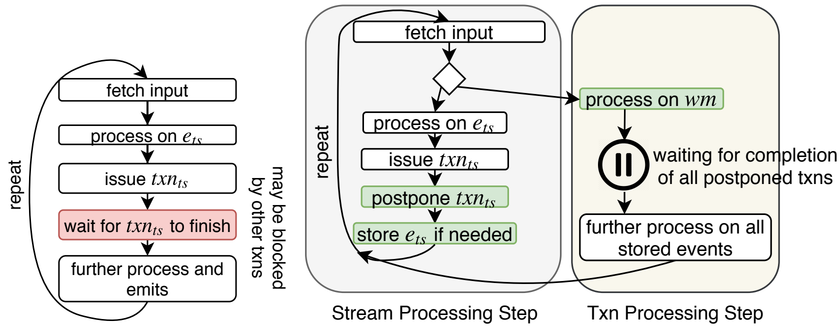
Let us assume state A and B are grouped into the same partition and C are split into different partitions. As shown in Figure 3(c), the execution of txn_3 (only touches state C) will never compete with the other two transactions. However, as txn_1 and txn_2 target at the same partition, their execution must be performed serially. In other words, to achieve good performance, PAT requires the system to perfectly partition shard states beforehand and each individual transaction only need to touch on single partition with minimum conflicts among each other. However, multi-partition transactions can significantly degrade the overall performance – a common problem for all partition-based approaches [33]. For example, consider txn_3 also needs to access B, or C is split in the same partition of B. In either case, all three transactions will be executed serially under PAT.

There are a few more related studies [14, 7], which are omitted here either due to they follow similar approaches [14] described before or lack implementation details [7]. In summary, prior solutions [31, 16, 14, 38] commonly adopt a synchronized execution model (shown in Figure 4a), which severely restrict system concurrency and scalability. *Firstly*, executing the state access operations strictly following the dependencies encoded in the streaming computation logic can cause extra long execution stalls in stream processing, consequently degrading the overall system throughput. *Secondly*, each event is synchronized with locks to guard correct locking sequence, which can cause high performance overhead under highly contended workloads. Although, PAT (i.e., S-Store) can potentially reduce such overhead by careful partition shared states beforehand, it quickly falls back to LAL with more multi-partition transactions. Therefore, we need a new solution for scaling the transactional data management in DSPSs on modern multicore architectures.

4 System Design Overview

Witnessing the scalability limitation in existing solutions, we propose `TStream` with two key novel designs.

Asynchronous state management. Synchronized state accesses are not only *harmful* leaving many parallelism opportunities unexplored but also often *unnecessary*. To allow streaming operators to continue process more input events without waiting for its currently issued state



(a) synchronized execution (b) asynchronized execution (TStream)

Figure 4: Workflow overview of an operator under different execution models.

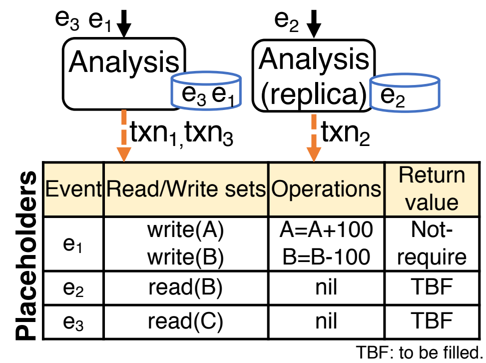


Figure 5: Example updates of placeholders.

transaction to finish, TStream adopts an asynchronize execution model that detaches state transaction processing from application computation logic. Through postponing state accesses by recording necessary information (e.g., read/write sets) into a conceptual data structure – placeholders, TStream minimizes unnecessary stalls caused by state management in stream processing.

TStream adopts *watermark* as periodic signal to trigger the actual transaction process. A watermark guarantees that all later input events must have larger timestamp than it. Postponed state transactions between two subsequent watermarks hence naturally form a batch, and the batch length can be then tuned to tradeoff system latency and throughput.

Operation-Chains parallel execution. To take advantage of modern multicore architectures, TStream adopts an *operation-chain* based execution model, that greedily extract parallelism opportunists among each batch transactions. As shown in Figure 3(d), multiple operation chains are built and can be (ideally) concurrently processed with no locks while still guaranteeing the aforementioned consistency properties. We further apply dynamic work-scheduling and NUMA-aware placement techniques to improve its processing efficiency.

4.1 Asynchronous State Management

One key design principle of TStream is to minimize unnecessary computation stalls caused by state transactions in stream processing. To achieve that, TStream detaches state transaction processing from application computation logic. Specifically, the workflow of an operator in TStream is separated into two steps as depicted in Figure 4b. (1) During stream processing, operator in TStream will not wait for its currently issued state transaction to be executed. Instead, it only *postpones* the transaction, and *stores* triggering event if needed. (2) When it receives a *watermark* (wm), it enters *transaction processing step*. Specifically, it pauses further stream processing and waits for postponed transactions to be evaluated. Once all transactions are finished execution, it will be notified to process on stored events, and continue further processing.

Stream processing step. In order to postpone state transaction, necessary input information including read/write sets of each state transaction are recorded in a conceptual data structure called *placeholders*. Figure 5 shows an example process of postponing two state transactions. After txn_1 is recorded in placeholders, Analysis can continue work on other input events (e.g., e_3). This eliminates expensive synchronization on each individual transaction.

The placeholder is only conceptually centralized and is implemented as an auxiliary data structure of each event hence no centralized contention. Events may need to be stored temporarily as they may require further process deepening on evaluation results of corresponding state transaction. For example, e_2 has to be marked as *in-complete* since it requires the value of state B for further computation. Reference (i.e., pointer) to the event's placeholder will be embedded with transaction to be evaluated later. During the processing of txn_2, e_2's placeholder will be updated with the value of state B through the reference. When operator is resumed after all transactions are processed, e_2's placeholder will then contain the desired value to support further process.

Transaction processing step. Instead of eagerly evaluate each received state transaction, TStream postpones and accumulates them into a batch. Subsequently, TStream adopts *watermark* as periodic synchronization signal to trigger the process of a batched transactions.

Figure 6 shows an example workflow of receiving watermarks. We assume a watermark with timestamp of 5 is arrived, and it triggers Analysis operator to start transaction processing. Note that, the same watermark must be broadcast to all instances of Analysis before transaction processing can start. This is implemented by a *Countdown Latch*[2] in TStream. During transaction processing, all instances are paused and no further input events (e.g., e_6, e_7) are allowed to enter the system. When all postponed

[2] https://www.baeldung.com/java-countdown-latch

transactions are evaluated, operator will resume the process of future input events, and the current watermark is forward to all instances of next operator.

Following previous work, we rely on stream sources and operators (e.g., Spout) to create watermarks based on their knowledge of the stream data [32]. The interval size of two subsequent watermark plays an important role in tuning system throughput and processing latency. Having a large interval, the system will wait for longer to start transaction processing (i.e., synchronization overhead in `TStream`), which potentially worse processing latency. Conversely, having a small interval size, the system throughput may drops with insufficient parallelism (too few transactions in the batch), which may end up with more input events queued up, and eventually leads to high processing latency. We evaluate the effect of watermark interval in our experiments.

4.2 Operation-Chains Parallel Execution

Enabling more concurrency has become particularly important with the proliferation of multicore and large-scale systems. As a watermark guarantees that no subsequent input events has a timestamp smaller than any events a prior of it, `TStream` has the complete knowledge of all state transactions to be handled *without worrying future* transactions. In other words, `TStream` only needs to concentrate on improving processing throughput of every batch of state transactions arrived between two consecutive watermarks.

Observing that the order-preserving property essentially determines the execution sequences of conflicting operations of each state. For example, a write operation must precede a read operation of the same state if the write operation is triggered by an earlier stream event. We propose an *operation-chain* based execution model to process every batch of state transactions. Specifically, all state transactions are decomposed into atomic state access operations. Each operation, either read/write targets at one state. Operations targeting the same state form an `operation chain` in the timestamp order, which naturally preserves `ACID+O` properties. These operation chains can be then (ideally) executed concurrently and independently, and hence allows `TStream` to achieve a maximum concurrency of the number of operation chains constructed for every batch of state transactions.

Construct operation chains. Figure 7 shows an example of decomposing three transactions. Two operation chains are formed as a result. First, $txn_{1,2,3}$ are decomposed into atomic operations, where each operation is annotated with timestamp of its original transaction, targeting state, access type, and optional parameters (e.g., value to update state). Then, those decomposed operations are hash partitioned based on their targeting state's primary key. Finally, operations of the same partition group form an operation chain that is sorted by timestamp. Each operation chain is implemented by a concurrent ordered data structured (e.g., concurrent skip-list), allowing concurrent insertion while still guaranteeing that items (i.e., operations) are ordered by the triggering event timestamp.

Evaluate operation chains. Constructed operation chains can be then (ideally) processed independently by multiple threads concurrently. Recall that all instances of the operator entering transaction processing step are paused (i.e., waiting to be notified), hardware resources (e.g., CPU cores) can be utilized to process transactions. In `TStream`, we use a dedicated thread pool with predefined number of threads to evaluate operation chains.

A simple strategy is to statically assign an equal number of operation chains (as tasks) to every working thread. However, such static approach may not always achieve good load balancing. Instead, `TStream` adopts dynamic work-scheduling [12] to achieve better load balancing. Specifically, `TStream` maintains a pool of operation chains, where multiple threads (from the thread pool) can fetch one operation chain (as a task) from the pool to work with. Once a thread finishes its task (e.g., a short operation chain with few operations), it fetches another until there is no task left in the pool. Furthermore, once an operation chain has only read-only operations unprocessed, it can be cooperatively processed by multiple threads.

Handling cross-states dependency. The aforementioned evaluation approach unfortunately does not work correctly when operations on one state may depend on another state. Assuming txn_7 is to update C with the value of B plus one (i.e., txn_7:C=B+1). Denote the corresponding decomposed operation as O_7. To support such cross-chain dependency is tricky as we need to make sure all (at least those a-prior of O_7) updates on B are applied before execute O_7. Otherwise, O_7 may read a wrong version of B violating `ACID+O` properties.

`TStream` handles such issue from two aspects. An example of handling txn_7 is illustrated in Figure 8.

(1) Encoding. `TStream` encodes dependency information and update operations on the depended state inside each operation. For example, O_7's dependency on B is encoded as its parameters. It also needs to record all update operations of depended state with a smaller timestamp. This can be done by traversing operation chain of depended state. In this example, O_6 is recorded and O_9 is not.

Such encoding step has to be performed only when watermark is received by all instances and before actual transaction execution starts. This ensures the complete knowledge of all operations to handle.

(2) Querying. During transaction execution, one thread will eventually encounter operation with dependency information encoded. It then enquires the correct value of depended state. Figure 8 shows that O_7 fails to obtain correct version of state B as O_6 has not been executed. The thread can switch out and evaluate other operation chains. After

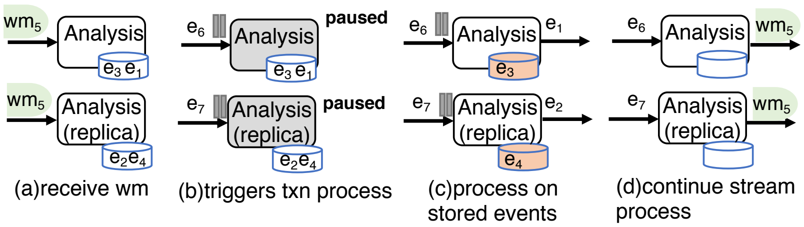


Figure 6: Example workflow of receiving watermarks.

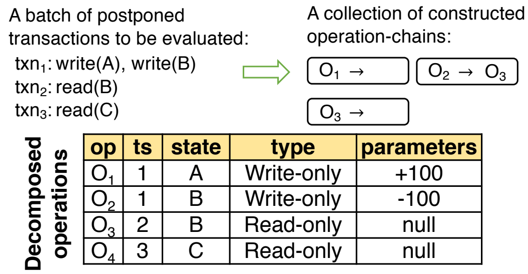


Figure 7: Example operation chains construction.

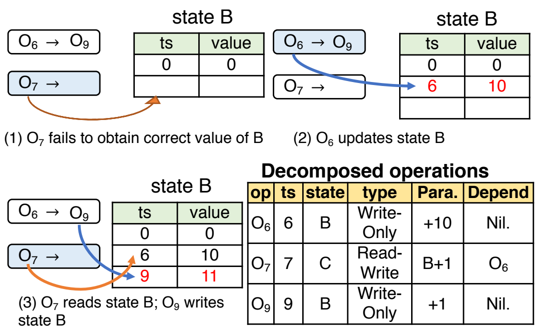


Figure 8: Handling cross-state operation.

O_6 is executed and the value of B is updated, O_7 can then continue its process. Note that, TStream keeps multiple versions of each state updated at different timestamp to allow write without waiting for read. Therefore, the process of O_7 will not block the process of O_9.

4.3 Further Optimizations

Evaluation pushdown. Consider a write operation, which requires to *double* a state. Naive implementation would require the system return the state value to stream operator, who will multiply the value by two, and write the value back to state. To reduce such round-trip communication, TStream encodes the multiplication with operation itself, similar to how TStream handles cross-states dependency. During the processing of the write operation, the multiplication and write will be executed consecutively after read. Conversely, if an operator requires the state value for further computation (e.g., in subsequent stream computation), then TStream has to write back the state value to placeholders during transaction processing.

NUMA-awareness. Following previous work [34], we configure the data placement of the shared state in an island-aware manner for NUMA-awareness. Specifically, we first distribute the shared states evenly to each NUMA node, so that the workload assigned to each node is (ideally) balanced. Then, the working threads for transaction processing is divided into several groups, where each group is responsible for workloads of part of the CPU sockets (i.e., island). We evaluate the effect of different grouping and select the best one for all algorithms in our experiments.

5 Implementation details

TStream is built on BriskStream [47] – a multicore optimized stream processing system with extensions to support transactional state management. Specifically, we extend BriskStream with state transaction APIs and transaction processing engine, which is developed based on Cavalia [41] database. For comparison, we implement multiple related algorithms including LAL, LWM and PAT (S-Store) into TStream. This allows us to abstract away the implementation details of each approach and concentrate on the algorithm itself.

No Order Preserving CC (NOCC). At its minimum, the system needs to ensure that accesses to shared states are processed with ACID guarantees. We hence additionally implement *NOCC* (w/o order-preserving consistency) as a baseline based on the two-phase-locking with no-wait deadlock prevention strategy [18]. Other conventional protocols (such as MVCC [40]) are applicable but similarly *do not* guarantee order-preserving consistency as we have discussed before.

5.1 Discussion

Limitations. The main disadvantage of TStream is that earlier issued transactions can not be immediately processed until subsequent watermark arrived to trigger the processing. This potentially increases latency. However, such lazy evaluation approach brings many opportunities in extracting higher concurrency while preserving event order. As a result, the significant improved performance minimizes the latency penalty. Another limitation is that, the write-set of a state transaction must be deducible before the transaction begins. Fortunately, all the tested workloads (including both of our two use cases), that we are going to discuss shortly later, provides read/write sets before execution and TStream's limitation does not apply here. Specifically, stream operator already knows which state(s) to access according to the input event it processes. Note that, use case 1 is a real-world application suggested by a commercial transactional stream processing system, namely Ledger [4]. We admit that this is not a must, and we plan to investigate the situation when read/write sets are not available. A potential solution is to

Table 2: Comparisons in related systems

	ACID	ACID +O	Large State	Multicore/ Manycore	Stream Processing
Storm[3]/Flink[1]	×	×	×	×	√
BriskStream[46]/ StreamBox[32]	×	×	×	√	√
Cavalia[42]	√	×	√	√	×
SDG[20]	×	×	√	√	√
LWM[38]/PAT[31]	√	√	√	limited	√
TStream	√	√	√	√	√

rely on pre-analysis [35], and we defer it as future work to explore.

System Comparison. We list the comparison to some related systems[3] in Table 2. DSPSs like Storm [3] and Flink [1] provide simple API to express streaming application but scale poorly on modern multicore processors [46]. StreamBox [32], BriskStream [47], and Saber [27] are able to achieve very high throughput and low latency of stream processing, but they are lack of a built-in support of transactional state management. State-of-the-art OLTP database such as Cavalia [41] does not provide order-preserving consistency nor stream processing capability. SDG [20] manages large mutable state distributedly but does not provide transactional guarantees nor order-preserving consistency. The closest works to `TStream` are DSPSs supporting ACID+O properties [38, 31], which are however limited at their scalability on modern multicores. In the next section, we compare `TStream` with those prior solutions guaranteeing ACID+O properties, and thus omit the comparison with others (e.g., Storm/Flink and SDG).

6 Evaluation

In this section, we evaluate the effectiveness of `TStream`. We first study (Section 6.3) the case when shared states cannot be partitioned beforehand, or has to be stored in a single partition. Then, we study the effect of states partitioning in Section 6.4, where we assume the shared states are pre-partitioned into disjoint subsets. We study two use case workloads in Section 6.5. Finally, we perform sensitivity study to understand the design trade-off in `TStream` with different workloads in Section 6.6.

6.1 Experimental Setup

We conduct the experiment on a 4-socket Intel Xeon E7-4820 server with 128 GB DRAM. Each socket contains ten 1.9GHz cores and 25MB of L3 cache. The operating system is Linux 4.4.0-62-generic. The number of cores devoted to the system and the size of watermark interval are system parameters, which can be varied by the system administrator. We vary both parameters in our experiments. In our evaluation, there is a one-to-one binding between threads and cores. We reserve two cores in the system: one for running data stream producer (i.e., Spout), and the other for output stream receiver (i.e., Sink). Therefore, we devote 2 to 38 cores for running other operators of the application to evaluate the system scalability.

Following the previous work [43], we also report how much time each transaction spends in different components of the system. Specifically, we divide the time spend during a transaction execution into the following components. *Useful* is the time that the transaction is really operating on records. *Abort* is the time spent due to transaction abort (e.g., failure for acquiring write locks). *Lock* stands for the total amount of time that a transaction spends due to lock acquisition. *Sync* is the time that a transaction spends due to synchronization to enforce order-preserving property. A transaction may need to synchronize for either a global synchronization primitives (e.g., *lockOrder* in LAL and LWM), partition locks in PAT or watermarks in `TStream`. *Others* stands for all other overheads including indexing, timestamp allocation, etc.

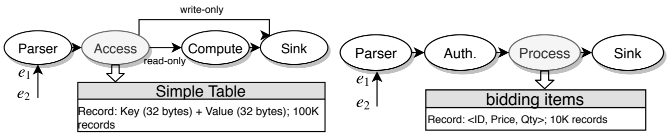
6.2 Benchmark Workloads

We use both microbenchmark and two use case workloads for performance evaluation.

MicroBenchmark. Figure 9(a) shows the topology of our microbenchmark containing four operators. `Parser` continuously feeds synthetic input events to `Access` operator. `Access` issues an state access transaction (read-only or write-only) for each input event. Information including both access type (read/write), targeting states (length is set to ten) and optional parameters (the value to write) are extracted from each input event. As `Access` may be carried in multiple instances, consistency needs to be preserved. If an event triggers a read-only transaction, `Access` emits the returned state values as one event to `Compute`; otherwise, it simply forwards its input event (which triggers write-only transaction) to `Sink`. `Compute` performs a summation calculation of the returned state values from `Access`, and emits the results as one event to `Sink`. We use `Sink` to record the output stream from `Compute` and `Access` to monitor system performance. Despite its simplicity, the microbenchmark shall be applicable to cover a wide range of different workloads by varying different workload parameters such as the skew of keys (*theta*), read-write ratio and state partition. We vary all those factors in our experiments.

Use case 1: Online Book Shopping. We have described use case 1 in Section 2, and we highlight here its configurations used in our experiments. Detailed descriptions can be found in [4]. We set a balanced ratio of transfer and deposit requests (i.e., 50% for each) in `Parser`. Transaction length is four for transfer request (i.e., each

[3]Ledger [4] recently introduces transactional state management to Flink, but is unfortunately close-sourced.



(a) Microbenchmark (b) Online Bidding Systems

Figure 9: Application topology of two workloads.

request touches four records) and is two for deposit request. We vary the skew of keys (theta) from 0.6 to 0.8. The table contains 100k unique records. The ratio of multi-partition transaction is 0.5 under PAT.

Use case 2: Online Bidding System. The second use case represents a simplified bidding system [36], where users bid for items without brokers. The application contains four operators as depicted in Figure 9(c), and we only consider item table for simplification. `Parser` continuously emits events describing requests from either buyers or sellers. `Auth.` authenticates the requests such as validating the request's ip address and dispatches valid requests for further process. We use a dummy function to simulate such authentication process, and the selectivity is one in this work. There are three types of requests handling in `Process` operator. (1) *bid request* for one item from buyers, if the bid hits the ask price for an item, which has sufficient quantity available, the request is processed (i.e., reduce the quantity of the item), otherwise rejected; It is noteworthy that, as multiple users may compete for the same item, the request *made earlier* will be *processed earlier*, which is naturally guaranteed by the `ACID+O` properties. (2) *alter request* modify prices of a list of items. (3) *top request* tops up quantity of a list of items. `Process` has a selectivity of one in this work, that is, it generates one event for each input event describing the request processing results. We use `Sink` to record the output stream from `Process` to monitor system performance.

We use a single table with 10k records to represent the bidding items. The ratio of bid, alter, and top requests is 6:1:1. The ratio of multi-partition transaction is set to be 0.75 under PAT, and theta is set to be 0.6. Such configuration represents a rather competitive seller's market scenario and transactions conflict severely.

6.3 Single Partition Analysis

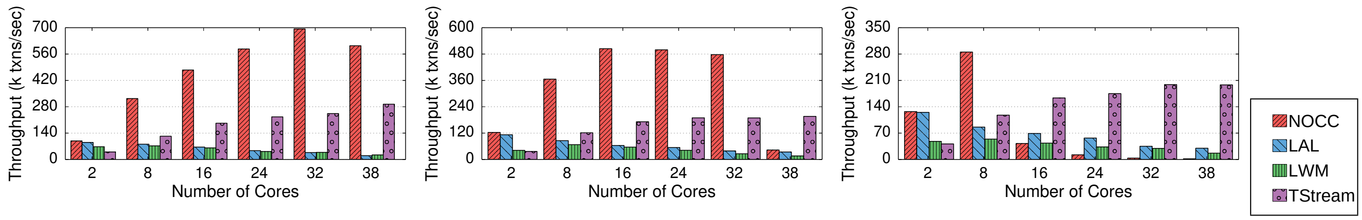
We first use microbenchmark to evaluate different algorithms described in Section 3. We use a single table consisting of 100,000 records to represent a non-partitionable shared application states. Each record has a size of 64 bytes, and the watermark interval is set to 100 ms in `TStream`. Records are stored in a single partition and all worker threads can access any states. Hence, the partition-based scheme (PAT) is excluded in this section.

Read-Only Workload. In the first scalability analysis, we configure input events to trigger only read requests. We set the key skew factor to be 0, and hence the state is accessed with uniform frequency. The read-only workload helps to stress the handover between stream computation and state management of `TStream` as much as possible. Specifically, all input events have to be marked as *in-complete* and the pending transactions need to be processed (all at once) once the watermark is arrived. `Compute` can only start its computation after `Access` successfully obtains state value.

Figure 10(a) shows the results of our experiment, and we have three main observations. First, the throughput of the most relaxed scheme *without order-preserving guarantee* – NOCC, increases almost linearly with the number of cores. This is expected as there is no lock contention in this workload. Second, lock-ahead based algorithms (LAL and LWM) perform well when the number of cores is small, but they stop scaling when more than 8 cores are used. The time breakdown in Figure 11(a) indicates that *sync time* dominates the runtime of both LAL and LWM with a large core count. This is primarily caused by their lock-ahead mechanism. Third, `TStream` performs much better than LAL and LWM while guaranteeing the same consistency. However, Figure 11(a) indicates that `TStream` still spends a large portion of time in synchronization, mainly due to the waiting for watermarks. Furthermore, operation chain construction process also contributes to the high others overhead in `TStream`. There is still large room for further improvement.

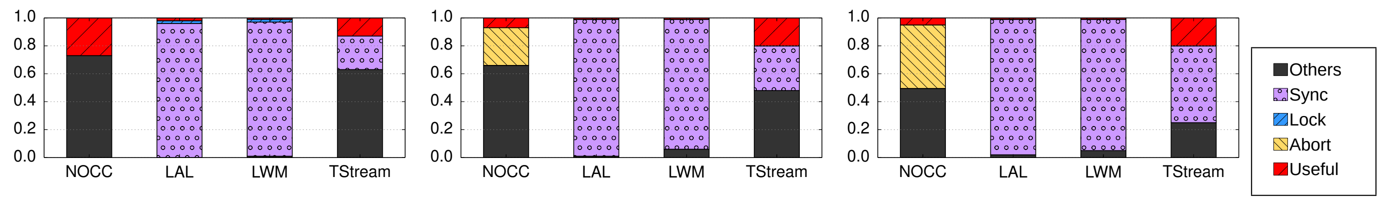
Write-Intensive Workload. We next study a write-intensive workload, where input events only trigger write requests to the shared states. To represent a more realistic scenario, we model the accessing distribution as Zipfian skew, where certain states are more likely to be accessed than others. The amount of skew in the workload is determined by the parameter, *theta*. We use the medium and high contention levels for the transactions' access patterns.

The medium contention results in Figure 10(b) shows a similar trend as in read-only case except that NOCC performs poorly with larger core counts. Figure 11(b) shows that there is a significant increase in management overhead. Under NOCC, transaction needs to hold a copy of records so that it can roll back the changes during aborting. Other order-preserving schemes are permissive in nature and does not involve any transaction aborts in this workload. This is because their lock accusations are performed sequentially due to the lock-ahead mechanism. Therefore, they do not bear such memory-copy overhead. The results of high contention case shown in Figure 10(c) show that only `TStream` is able to further improve its performance beyond 32 cores. The reasons are two folds. First, `TStream` is lock-free and no abort in acquiring locks, higher skew level only potentially brings higher workload unbalancing (i.e., some operation chains may be significant longer than others) to `TStream`. Second, `TStream`'s dynamic work-scheduling



(a) Read-Only (b) Write-Intensive (Medium) (c) Write-Intensive (High)

Figure 10: Throughput comparison for varying types of workload.



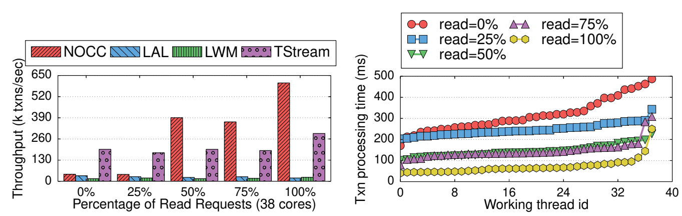
(a) Read-Only (b) Write-Intensive (Medium) (c) Write-Intensive (High)

Figure 11: Runtime breakdown (38 cores) for varying types of workload.

scheme relieves the increased workload unbalance issue. In contrast, the breakdown in Figure 11(c) indicates that NOCC is inhibited by excessive abort and redo. Again, LAL and LWM spend most of their execution time in synchronization for guaranteeing correct lock insertion sequence. LWM performs significantly worse than LAL, due to the additional overhead of accessing the `lwm` counter. Note that, `lwm` is read (at least once) during process and updated once during commit for each write-only transaction.

Read/Write Mixture Workload. An application may issue both read and write operations to the internal states. In our experiment, we vary the percentage of read operations executed by each transaction. Figure 12(a) shows that NOCC performs better with more read operations. The reason is that there is an increasing concurrency with lesser contention as it does not need to preserve ordering property. LAL and LWM are not affected by the read-write ratio because they are dominated by sync time for enforcing ordering. `TStream` also performs similar regardless of read-write ratio, which is mainly due to the its unique asynchronous execution model. On the one hand, the average processing time per transaction per core of `TStream` increases with more write requests, which is shown in Figure 12(b). On the other hand, each read request requires `TStream` to fill up the *return value* of placeholders (see Figure 5 in Section 4.1), which also involves write operations.

Small State Size. The internal state of stream processing (e.g., IoT) can be relatively small compared to traditional relational database. Under such configuration, the chances of transaction contention is very high. We conduct such a study by reducing state size to be only 10k. We use



(a) Throughput (b) Processing time per core

Figure 12: Read/write mixture evaluation.

read/write mixture workload (50% each) in this study. Figure 13 shows that NOCC cannot progress when more than 24 cores are used due to the excessive abort and redo process. Conversely, `TStream` shows similar and sometimes even better performance. The robust of `TStream` under highly contended workloads comes from two aspects. First, under such configuration, many transaction requests are targeting at the same state. The overhead of constructing operation chains are effectively amortized. Second, `TStream`'s dynamic work-scheduling scheme successfully reduces the potential workload unbalancing issue among working threads yielding better overall performance.

6.4 Multi-Partition Analysis

We now study the effect of shared states partitioning. We use a simple hashing strategy to assign the records to partitions based on their primary keys so that each partition stores approximately the same number of records.

As a common issue of all partition based algorithms [33], the performance of PAT is heavily depended on the length

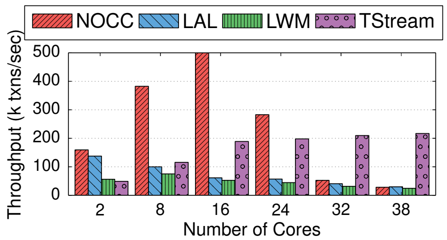
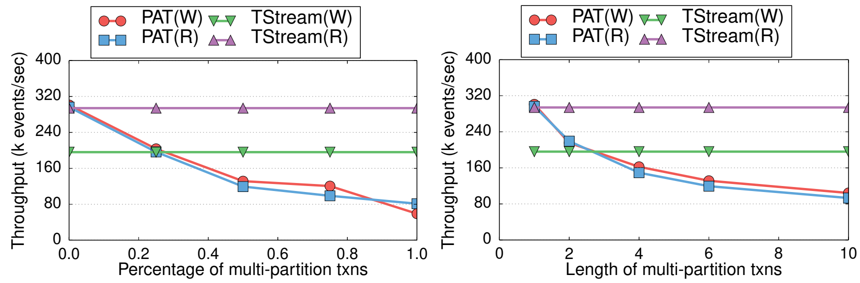


Figure 13: Small state size study (state size=10,000).



(a) Varying Ratio of Multi-partition Txns

(b) Varying Length of Multi-partition Txns

Figure 14: Sensitivity analysis of the PAT scheme for workloads with multi-partition transactions.

and ratio of multi-partition transactions. Hence, we vary both parameters in our experiments. We first set the length of multi-partition to be 6, that is each multi-partition transaction needs to touch 6 different partitions of the shared states. We vary the percentage of multi-partition transactions in the workload. The results are shown in Figure 14(a). As expected, the system's throughput degrades with more multi-partition transactions as they reduce the amount of parallelism. When there are more than 25% of the transactions access multi-partition, PAT scheme performs worse than TStream. We next execute workload with 50% multi-partition transactions, and Figure 14(b) illustrates the results of varying the number of partitions that they access. The system performance drops significantly with muti-partition transactions accessing two or more partitions. In the worst case, PAT falls back to LAL when every transaction needs to access all partitions.

6.5 Use case Studies

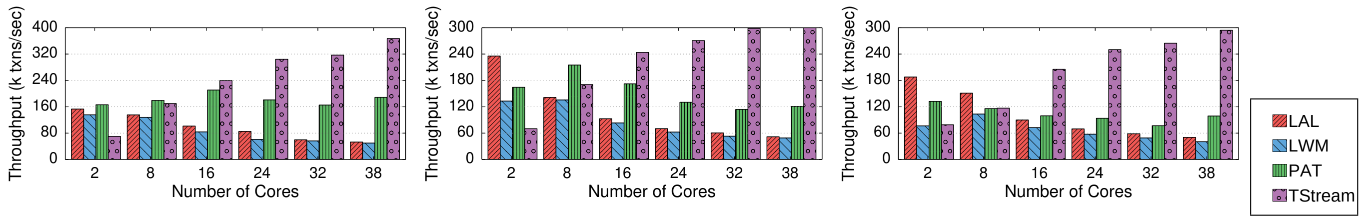
We now study the system performance for two use case workloads assuming each table is pre-partitioned. For this study, we exclude NOCC as it cannot ensure processing correctness. This applies to all other conventional CC protocols without order-preserving guarantee. The results shown in Figure 15 confirm that TStream outperforms all other schemes significantly at large core counts at both use cases, and performs especially better at highly contented workload. Comparing to LAL and LWM [38], TStream achieves higher concurrency while preserving the same consistency properties. The key reason is that TStream is able to explore the hidden parallelism opportunities among transactions at runtime even if they have conflicting operations. Comparing to PAT (i.e., S-Store), TStream does not need to statically partition shared states beforehand. Instead, it dynamically partitions operations inside each transaction at a fine-granularity of operation chain. This allows TStream to be able to explore more parallelism. However, such fine-grained partition also comes with high constant overhead of managing each operation chain. This is also the reason why TStream performs well, sometimes even better under highly contended workload, where management overhead of each operation chain is effectively amortized among more operations. We plan to investigate such issue to further enhance our system in future work.

6.6 Sensitivity Study

The Effect of Watermark Interval. We now study the effect of varying size of watermark interval in TStream, on both latency and throughput. Figure 16 shows that the relationship between system throughput and watermark interval is non-linear. Deciding a suitable batch size can be hence complex. Potential techniques have been studied in previous work [17], and we leave it as future work to enhance our system. Following previous work [17], we define the end-to-end latency of a streaming workload as the duration between the time when an input event enters the system and the time when the results corresponding to that event is generated. The results in Figure 17 show that the processing latency of TStream is comparable to other schemes and can be tuned to be even smaller. This is a highly desired feature as different applications can have different latency requirements, and TStream provides users the opportunity to tune the system flexibly.

Factor Study. Figure 18 shows the relative effectiveness of several optimization techniques of TStream. Changes are added left to right and are cumulative. *Simple* refers to TStream with an synchronous execution mode – it falls back to LAL. *+Asy-execute* enables asynchronous execution model with evaluation push down. *+Dynamic-scheduling* further enables dynamic task scheduling technique to resolve load unbalancing issue. *+Numa-aware* adds the consideration of different NUMA-aware thread and workload placement configurations.

There are three major observations in Figure 18. First, asynchronous state management brings remarkable performance improvement as it avoids unnecessary execution stalls between stream processing and transaction processing. Second, dynamic work scheduling plays a



(a) Use Case 1 (theta=0.6)   (b) Use Case 1 (theta=0.8)   (c) Use Case 2 (theta=0.6)

Figure 15: Performance evaluation of two use case workloads.

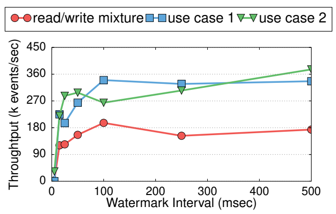
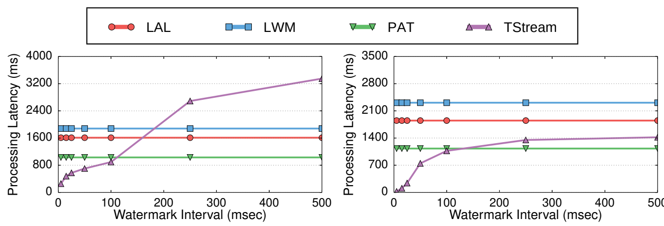


Figure 16: Throughput on varying watermark interval.



(a) Use case 1   (b) Use case 2

Figure 17: 90^{th} Percentile end to end processing latency.

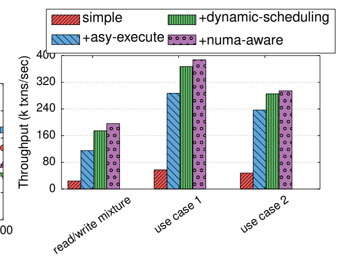


Figure 18: A factor analysis for TStream (theta=0.6).

critical role in reducing workload unbalancing issue under contented workloads. Third, NUMA-awareness brings minor performance improvement in all testing workloads. In TStream, each state is always processed by the same thread (as long as rescheduling is not involved for that state), which greedily minimizes NUMA traffic. Nevertheless, we plan to evaluate the impact of NUMA-awareness with more extensive applications in future work.

7 Related Work

Transactional DSPSs. Most modern DSPSs either do not support transactional state management or have to rely on third-party data storage systems (e.g., [2, 5]), which not only degrades the system performance but also violates the state consistency [16]. Similar motivations have led to a very recent launch of a commercial system, called Ledger [4], developed by *Data Artisans*. It is close-sourced, and we cannot compare our system with it. Wang et al. [38] conducted an early study on the importance of supporting transactional state management in stream processing. They propose a new computing paradigm, called *active complex event processing* (ACEP), to enable complex interactive real-time analytics. Botan et al. [14] presented an *unified transactional model* for streaming applications, called UTM. Recently, MeeHan et al. [31] attempted to fuse OLTP and streaming processing together and developed the S-Store system. Different from previous implementation, TStream's novel asynchronous execution design has shown to achieve much higher throughput and scalability at different types of workloads with limited penalty of higher processing latency.

Concurrency Control. Concurrency control (CC) protocols have been investigated widely in decades [11, 19, 43, 41]. Beyond guaranteeing ACID properties, DSPSs must also provide *order-preserving consistency* – a property that conventional CC protocols are not well-prepared for. Several prior works on transaction chopping and lazy evaluation [39, 19] inspired our design of the operation chain based processing model. The key contribution of our work is the application of transaction decomposition to scale transactional state management for stream processing with order-preserving consistency. In particular, we show that the overhead of maintaining sorted data structure (i.e., the operation chain) can be overcome by the performance gains from improved stream processing concurrency. Several prior works [43, 40, 23] studied the scalability bottlenecks in varies aspects of concurrency control algorithms. Different from these works, our work studied the concurrency control algorithms in the context of DSPSs with different consistency requirements (i.e., order-preserving). TStream is conceptually similar to some deterministic databases [6], which generates a dependency graph that deterministically orders transactions' conflicting record. However, the determinism of TStream is given by input events rather than the system itself. Furthermore, deterministic databases are designed for distributed environment, and the dependency analysis is performed by one thread (or one host) before the actual execution can start. Such dry-run phase is too heavy to be applied in TStream.

Data Management Systems on Multicores. Multicore architectures have brought many research challenges and opportunities for in-memory data management, as outlined in recent surveys [37, 45]. There have been studies on optimizing the instruction cache performance [48, 24], the memory and cache performance [8, 25, 13, 9] and NUMA [29, 30, 22]. To address the needs for a NUMA-aware OLTP system, Porobic et al. [34] proposed "hardware islands", in which NUMA nodes are grouped into logical partitions as islands and communicate through message passing among different islands. We applied the similar idea in `TStream` but it brings only minor improvement in the testing workloads.

8 Conclusion

This paper introduces `TStream` aiming at scaling stream processing with transactional state management on shared-memory multicore architectures. `TStream` achieves high scalability via two key designs including 1) asynchronous state management, which minimizes execution stalls caused by state access operations and 2) operation-chains parallel execution model, which maximize transaction execution concurrency while guaranteeing `ACID+O` properties. We also provide the first comprehensive study of different algorithms for supporting transactional state management in DSPS and results have confirmed the superiority of `TStream`'s designs.

References

[1] Apache flink, `https://flink.apache.org/`, 2018.

[2] Apache smaza, `https://samza.apache.org/learn/documentation/0.7.0/container/state-management.html`, 2018.

[3] Apache storm, `http://storm.apache.org/`, 2018.

[4] Data Artisans Streaming Ledger Serializable ACID Transactions on Streaming Data, `https://www.da-platform.com/streaming-ledger`. 2018.

[5] Stateful stream processing in flink, `https://cwiki.apache.org/confluence/display/FLINK/Stateful+Stream+Processing`, 2018.

[6] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR'05*.

[7] 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 '17, pages 134–145, New York, NY, USA, 2017. ACM.

[8] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[9] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 362–373, April 2013.

[10] P. Bernstein and E. Newcomer. *Principles of Transaction Processing: For the Systems Professional*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[11] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv. 1981*.

[12] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[13] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 54–65, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[14] 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 '12, pages 204–215, New York, NY, USA, 2012. ACM.

[15] 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*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.

[16] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul, K. Tufte, H. Wang, and S. Zdonik. S-store: A streaming newsql system for big velocity applications. *Proc. VLDB Endow.*, 7(13):1633–1636, Aug. 2014.

[17] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 16:1–16:13, New York, NY, USA, 2014. ACM.

[18] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, Nov. 1976.

[19] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 15–26, New York, NY, USA, 2014. ACM.

[20] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making state explicit for imperative big data processing. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 49–60, Philadelphia, PA, 2014. USENIX Association.

[21] J. Ghaderi, S. Shakkottai, and R. Srikant. Scheduling storms and streams in the cloud. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '15, pages 439–440, New York, NY, USA, 2015. ACM.

[22] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. Deployment of query plans on multicores. *Proc. VLDB Endow.*, 8(3):233–244, Nov. 2014.

[23] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, 10(5), Jan. 2017.

[24] S. Harizopoulos and A. Ailamaki. Improving Instruction Cache Performance in OLTP. *ACM Trans. Database Syst*, 2006.

[25] B. He, Q. Luo, and B. Choi. Cache-conscious automata for xml filtering. In *21st International Conference on Data Engineering (ICDE'05)*, pages 878–889, April 2005.

[26] 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. *Proc. VLDB Endow.*, 1(2):1496–1499, Aug. 2008.

[27] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 555–569, New York, NY, USA, 2016. ACM.

[28] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[29] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 743–754, New York, NY, USA, 2014. ACM.

[30] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. M. Lohman. Numa-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[31] 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. *Proc. VLDB Endow.*, 8(13):2134–2145, Sept. 2015.

[32] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin. Streambox: Modern stream processing on a multicore machine. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 617–629, Santa Clara, CA, 2017. USENIX Association.

[33] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 61–72, New York, NY, USA, 2012. ACM.

[34] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. Oltp on hardware islands. *Proceedings of the VLDB Endowment*, 5(11):1447–1458, 2012.

[35] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *Proc. VLDB Endow.*, 7(10):821–832, June 2014.

[36] J. Tan and M. Zhong. An online bidding system (obs) under price match mechanism for commercial procurement. *Applied Mechanics and Materials*, 556-562:6540–6543, 05 2014.

[37] K.-L. Tan, Q. Cai, B. C. Ooi, W.-F. Wong, C. Yao, and H. Zhang. In-memory databases: Challenges and opportunities from software and hardware perspectives. *SIGMOD Rec.*, 44(2):35–40, Aug. 2015.

[38] D. Wang, E. A. Rundensteiner, and R. T. Ellison, III. Active complex event processing over event streams. *Proc. VLDB Endow.*, 4(10):634–645, July 2011.

[39] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1643–1658, New York, NY, USA, 2016. ACM.

[40] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, Mar. 2017.

[41] Y. Wu, C.-Y. Chan, and K.-L. Tan. Transaction healing: Scaling optimistic concurrency control on multicores. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1689–1704, New York, NY, USA, 2016. ACM.

[42] Y. Wu and K. Tan. Chronostream: Elastic stateful stream computation in the cloud. In *2015 IEEE 31st International Conference on Data Engineering*, pages 723–734, April 2015.

[43] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, Nov. 2014.

[44] S. Zeuch, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl. Analyzing efficient stream processing on modern hardware. *Proc. VLDB Endow.*, 12(5):516–530, Jan. 2019.

[45] H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, July 2015.

[46] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze. Revisiting the design of data stream processing systems on multi-core processors. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 659–670, April 2017.

[47] S. Zhang, J. He, A. C. Zhou, and B. He. Briskstream: Scaling Data Stream Processing on Multicore Architectures. To appear in SIGMOD, 2019.

[48] J. Zhou and K. A. Ross. Buffering databse operations for enhanced instruction cache performance. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 191–202. ACM, 2004.