Scaling Transactional State Management in Stream Processing 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

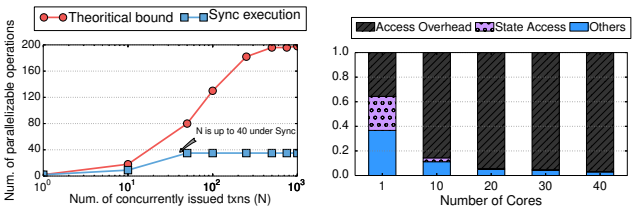
Introducing transactional state management into the data stream processing system (DSPS) has become an emerging research topic. Despite several solutions have been proposed, it is still an open question on how to design efficient transactional state management mechanisms in DSPS on multicore architectures. We introduce *TStream*, a DSPS with efficient transactional state management. Compared to previous works, it guarantees the same consistency properties while judiciously exploits more parallelism opportunities – both within the processing of each input event and among a (tunable) batch of input events. Targeting on multicore architectures, *TStream* is also designed to avoid any centralized contention points. To confirm the effectiveness of our proposal, we compared *TStream* against three prior solutions on a four-socket multicore machine. Our extensive experiment evaluations show that *TStream* yields up to 6.8 times higher throughput comparing with existing approaches with comparable end-to-end processing latency under the same consistency guarantee.

PVLDB Reference Format:
Shuhao Zhang, Yingjun Wu, Feng Zhang, Bingsheng He. Scaling Transactional State Management in Stream Processing on Multicores. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Data stream processing systems (DSPSs) are gaining their popularities in powering modern IoT (Internet-of-Things) and data streaming applications [14, 44, 19, 4, 5, 28]. Recently, several prior works [40, 29, 13, 8, 6] have studied *transactional state management* in stream processing, which allows DSPS to provide transactional consistency of application states. Despite a number of prior solutions have been proposed [40, 13, 29, 15, 8], how to scale transactional state management in stream processing on modern multicore architecture remains an open question.

Efficiently supporting transactional state management is challenging. *First*, an operator may need to manage *shared mutable states* [29], where the same application state may be concurrently accessed (read/write) by multiple instances of the operator (called executors) running in different threads. Based on our analysis of different applications [40, 29, 13, 8], we find that there is a basic three-step procedure of an executor on handling each input event. Specifically, when an executor obtains an input event, it processes (e.g., filter, count) on the event, then optionally accesses to the shared states. All accesses (read/write) triggered by one event are grouped into one *state transaction*, and read/write sets are extracted from the corresponding input event. Finally, it may conduct further process (e.g., aggregate the state values) that may depend on state access results. Any uncoordinated accesses to the same state can cause state inconsistencies, hence concurrent state transaction execution needs to guarantee `ACID` properties [8, 13]. *Second*, more than guaranteeing the `ACID` properties, DSPSs further need to guarantee the state access order follows the input event timestamp order [15] (i.e., `ACID+O`). For example, a read shall never be affected by a write to the same record, which is triggered by an event with a larger timestamp (see our motivation application in Section 2 for detail). As a consequence, simply relying on a third-party DBMS for managing shared states still leads to potential consistency violation due to the lack of support of such ordering guarantee.



(a) Overlooked parallelism opportunities. (b) Serverly contention overhead.

Figure 1: Issues of prior solutions (on a 40-core machine).

Due to the unique requirement of transactional state management, a number of related works have been proposed [40, 8, 13, 6, 29]. However, prior solutions scale poorly on modern multicore architectures. We observe that prior solutions commonly adopt a *synchronous state management* paradigm, where the aforementioned three-step procedure of processing each input event is conducted serially in an executor/thread. By taking toll processing query [9] as an example, Figure 1(a) shows that abundant parallelism opportunities (denoted as *Theoretical bound*)

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

are overlooked under prior solutions (denoted as *Sync execution*) with increasing number of concurrently issued transactions (N). As a state transaction is synchronously processed by an executor, the parallelism opportunities among multiple operations (i.e., read/write to one state) inside the transaction is overlooked. Furthermore, they can only concurrently issue a limited number of transactions (i.e., one executor one transaction), which is constrained by the number of physical cores (i.e., one executor one core). In this example, N is up to 40, which puts a hard limit on their processing concurrency. Similar observations are made in other existing solutions, hence there is a need for a more scalable solution. In Figure 1(b), we further take a partition-based approach *PAT* (the most promising existing algorithm [29]) as an example and evaluate how well it processes under varying available processor resources. We measure the average time spend in each executor on *state access*, which stands for actual time spent in accessing shared states, *access overhead* primarily due to lock acquisition and contention during state access, and *others* including actual processing time on input event and all other system overheads (e.g., context switching, method call). With the increasing number of cores, the access overhead quickly dominates runtime due to the serious contention overhead. The reason is that, in order to guarantee the unique ordering consistency requirement (`ACID+O`), prior approaches compare every lock insertion of each state transaction with a (or a set of) global counter(s) to ensure locks are granted strictly in the timestamp order, which results in centralized contention points and severely degrades system performance.

Witnessing the limitation of prior solutions, we present *TStream* with two key designs. Firstly, *TStream* adopts a novel *asynchronous state management* paradigm which allows an executor to *skip* and *postpone* state transaction and immediately process on more input events. Specifically, it accumulates and process all postponed state transactions periodically, triggered by punctuation signals [39]. This allows stream processing and transaction processing to be efficiently overlapped, and more transactions are concurrently issued to the system with more parallelism opportunities (i.e., N is no longer limited by the number of executors). Secondly, to exploit more parallelism among state transactions of every batch, we propose a novel transaction processing paradigm, namely *operation chains parallel processing*, that aggressively explore parallelism opportunities in the most fine-grained manner. It avoids any centralized locks while guaranteeing the same consistency requirement (i.e., `ACID+O`). Specifically, *TStream* dynamically decomposes all postponed state transactions into atomic state access operations. Then, it partitions operations into different groups sorted by their timestamp, called operation chains. Subsequently, all formed operation chains can be processed concurrently. Put them together, *TStream* achieves a high system concurrency at all times by exploring more parallelism opportunities and avoiding any centralized contention points and hence scales much better than prior solutions.

To confirm *TStream*'s effectiveness, we compared it against four alternative schemes on a high-performance 40-core machine. Our extensive experimental study on four applications show that *TStream* achieves 3.1x∼6.8x higher throughput than best performing existing algorithm with comparable and sometimes even smaller end-to-end processing latency.

Organization. The remainder of this paper is organized as follows. Section 2 covers the background. We summarize prior studies on supporting transactional state management in Section 3. Section 4 discuss the designs and optimizations of *TStream*. Section 5 discusses more implementation details of *TStream*. We report extensive experimental results in Section 6. Section 7 reviews more related work and Section 8 concludes this work.

2. PRELIMINARIES

Different DSPSs can have (slightly) different terminologies and execution models. In this paper, we adopt the execution model from a previous study [7], which has also been used in other DSPSs such as Storm [5], BriskStream [49] We briefly present here for completeness. We summarize terminologies in Table 1. Stream processing continuously processes 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. Our system can be extended to handle the case without monotonically increasing timestamps (see Section 4.3.1). A streaming application contains a sequence of *operators* $O_1, O_2, ..., O_n$, which continuously processes streaming events in a pipelining way [48]. To sustain high input stream ingress rate, each operator can be further executed in multiple *executors* (e.g., Java threads), which handle multiple input events concurrently.

Motivation Application. The usage of shared mutable states in stream processing is important for stream processing and has been studied in several prior works [40, 13, 29, 8]. For illustration, we use a stream application (*Online Bidding (OB)*) depicted in Figure 2 as a running example. The application represents a simplified bidding system [36], where users bid or sell items online. There are three types of trade requests as input events to process. (1) *bid request* reduces the quantity of an item if the bid price is larger or equal to the asking price, otherwise rejected. If the item has an insufficient quantity, the bid request is also rejected. (2) *alter request* modifies prices of a list of items. (3) *top request* increases the quantity of a list of items.

The application can be implemented with the following four streaming operators. `Parser` continuously emits events describing trade 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. `Process` handles the aforementioned three types of requests. We use `Sink` to record the output stream from `Process` to monitor system performance. During the processing of each event, `Process` may need to read/update item table, which contains item id, price and quantity information. The table has to be shared among all executors of `Process` as each request may touch *any* items and executors run concurrently in different threads. As a result, any uncoordinated accesses (i.e., update and read) to the same record can cause state inconsistencies.

Transactional State Management. To relieve users from managing shared states consistency by themselves, DSPS with transactional state management have been proposed [40, 13, 8]. Following previous works [29, 8], we define the set of state accesses triggered by processing of a single input event e_{ts} as one *state transaction*, denoted

Table 1: Summary of Terminologies

Term	Definition
Event	Input stream event with a monotonically increasing timestamp
Punctuation	Special tuple embedded in a data stream that indicates the end of a subset of the stream
Operator	Basic unit of a stream application, continuously process event streams
Executor	Each operator can be carried by multiple executors (i.e., threads)
Shared states	Mutable states that are concurrently accessible by multiple executors of an operator
State transaction	A set of state accesses triggered by processing of a single input event
ACID+O	A consistency property satisfying ACID properties and event timestamp ordering

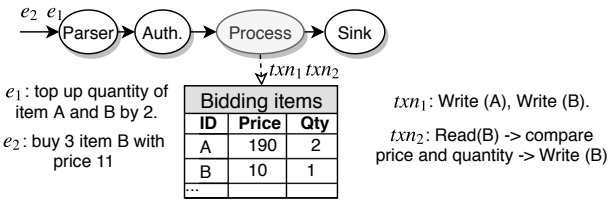


Figure 2: Online Bidding (OB) as a running example.

as txn_{ts}. Timestamp ts of a state transaction is defined as the same as its triggering event. As demonstrated by the previous examples, concurrent accesses to shared state can cause data inconsistency. Previous studies [8, 13, 29] advocate that the execution of multiple state transactions concurrently should satisfy `ACID` properties (due to space limitation, please refer to [8] for a detailed description). Besides `ACID` requirements, there has been an additional requirement of guaranteeing event timestamp ordering during state transaction execution, e.g., bidding request with a smaller timestamp should be processed earlier. Formally, we define a consistency property satisfying both `ACID` properties and event timestamp ordering constraint as `ACID+O` properties. A DSPS under `ACID+O` needs to ensure the state transaction schedule must be conflict equivalent to $txn_1 \prec ... \prec txn_n$. That is, transaction serialization order is given by *external* input event sequence under `ACID+O`. For example, in Figure 2, txn_1 shall be processed before txn_2 such that the quantity of item B is updated to 3, and the bidding request e_2 shall success as there is sufficient quantity and bid price is higher than asking price. Similarly, when multiple requests compete for the same item, whose quantity can only satisfy partial requests, the request with smaller timestamp shall be processed earlier to ensure fairness.

This is very different from conventional concurrency control protocols [10], which serialize transactions in an order that is conflict-equivalent to *any* certain serial schedule, which does not necessarily satisfy `ACID+O`. Thus, traditional concurrency control mechanisms [10] have to be revisited in order to support `ACID+O` in DSPSs.

3. EXISTING SOLUTIONS

Due to the unique requirement of `ACID+O` properties, a number of related works have been proposed [40, 8, 13, 6, 29]. Despite different types of locking schemes were studied, they commonly adopt a *synchronous state management* paradigm, where an executor is assigned to process one

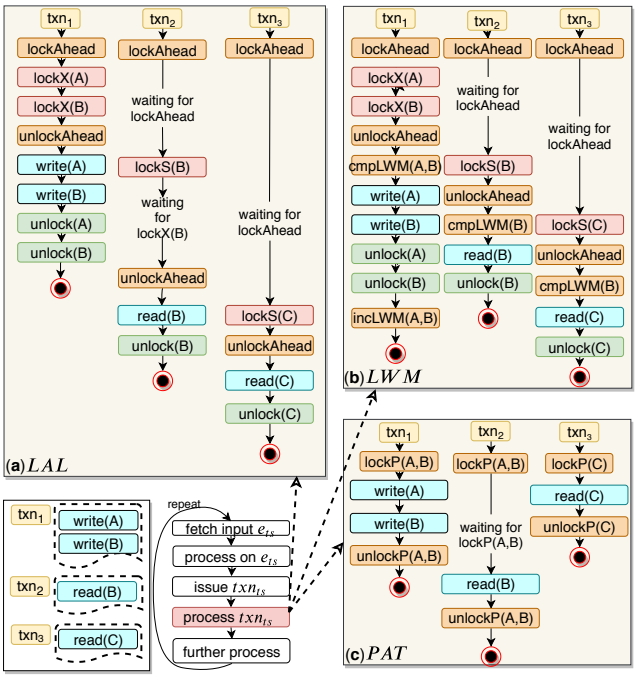


Figure 3: Examples of how existing algorithms process three concurrent state transactions.

input event, and only when all tasks (i.e., stream processing and state transaction processing) of processing that event is completed, the executor is available for the next input event. The workflow overview of an executor under synchronous state management is shown in the bottom left of Figure 3. There is a basic three-step procedure of an executor on handling each input event. For each input event (e_{ts}), an executor performs stream processing (e.g., filter, count, sum) followed by potentially shared states read/update (grouped in a state transaction, txn_{ts}), and further stream processing (e.g., calculate sum of state values) that may be depended on the state transaction. The top right of Figure 3 further illustrates how prior solutions process three concurrently generated state transactions by different locking schemes. We denote a read lock to record R as *lockS(R)*, and a write lock as *lockX(R)*. We use *lockAhead* and *lockP* as their process of ensuring locks are inserted in the correct order following event sequence. We further use *comLWM* and *incLWM* to denote the access and update of `lwm` counter under *LWM* scheme, respectively.

Lock Ahead 2PL (LAL): An earlier study from Wang et al. [40] described a strict two-phase locking (S2PL) based algorithm that allows multiple state transactions to run concurrently while maintaining state consistency. Consider a set of state transactions (txn_1,...,txn_n) generated during stream processing. Before processing on them, every transaction needs to first go through a *LockAhead* process, which compares its timestamp with a centralized global counter. Only the transaction with the lowest timestamp (e.g., txn_1) is allowed to proceed to acquire its locks. Once a transaction obtains all locks it needs, it releases *LockAhead* (i.e., update the global counter) to allow transaction with next lowest timestamp to acquire locks. For example, as shown in Figure 3(a)), *LockAhead* ensures txn_1 to obtain $lockX(A)$ and $lockX(B)$ ahead of txn_2, which guarantees the aforementioned `ACID+O` property. Note that, although

txn_2 is allowed to acquire locks (i.e., *lockS(B)*) after txn_1 releases *LockAhead*, it is blocked by *lockX(B)*. Such blocking, unfortunately, propagates to other transactions (e.g., txn_3), which are still waiting at *LockAhead*.

Low-Water-Mark (LWM): To allow read without blocking by write (e.g., *lockS(B)* of txn_2), Wang et al. [40] proposed to leverage on MVCC protocol [42], where the system maintains multiple versions of each state. However, the conventional MVCC does not guarantee `ACID+O`, and they hence proposed a new algorithm called *low-water-mark (LWM)*. *LWM* leverages a counter (i.e., `lwm`) of each state to guard the ordering constraint. Specifically, transactions that need to access a state need to compare their timestamp with the `lwm` counter of the state. Write lock is permitted only if the transaction's timestamp is equal to `lwm`; while read lock is permitted as long as the transaction's timestamp is larger than `lwm` so that it can read a *readily* version of the state. During commits, a transaction needs to update `lwm` of all its modified states. *LWM* still relies on the *LockAhead* mechanism to ensure locks are inserted following event sequence, which severely restricts system concurrency. Furthermore, although *LWM* brings higher processing concurrency than *LAL*, we found in our experiments that the overhead of accessing and updating *lwm* counters degrades system performance significantly making it perform even worse than *LAL*.

Partition-based approach (PAT): S-Store [29] fuses OLTP and stream processing into one engine. It is built based on H-Store – a shared-nothing deterministic database system [24]. Similar to the way 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 [29] for all shared state accesses, which severely limits system concurrency. We hence extend it to support multiple threads and refer the extended system as *PAT*. Specifically, *PAT* maintains an array of global counters, one for each partition (denoted as *LockP*). Each transaction needs first compare its timestamp with all global counters of its targeting partitions to synchronize the accesses to each partition. It is noteworthy that, despite being partitioned, all executors of an operator may access any partitions of shared states during stream processing, which is different from conventional stream partitioning [25]. Let us assume record A and B are grouped into the same partition and C is grouped differently. As shown in Figure 3(c), the execution of txn_3 (only touches C) will never compete with the other two transactions. In contrast, as txn_1 and txn_2 target at the same partition, their execution must be performed serially guarded by $LockP(A, B)$.

In Summary: Prior approaches have two common scalability limitations. Firstly, the synchronous execution model commonly adopted by prior approaches overlooks many parallelism opportunities – both intra- and inter-transaction parallelism as shown previously in Figure 1(a). Specifically, prior solutions overlook the parallelism opportunity within a state transaction, e.g., Write(A) of txn_1 can be processed in parallel with all other operations. However, as txn_1 is synchronously handled by one executor (i.e., thread), operations of txn_1 are processed serially missing parallelism opportunities. Furthermore, due to synchronous execution, each executor has to finish process all tasks of an input event before processing more input events. As a result, the system can concurrently issue only a limited number of transactions at the same time, which overlooks parallelism opportunities among more transactions. Secondly, the centralized locking schemes of prior solutions unnecessarily block more state transactions than required, which further degrades system performance as shown previously in Figure 1(b). For example, txn_3 is unnecessarily blocked for a long period under *LAL* and *LWM*. Although *PAT* (i.e., S-Store) can potentially reduce such overhead by careful partitioning shared states beforehand, it quickly falls back to *LAL* with more multi-partition transactions (e.g., consider txn_3 also needs to access B) – a common problem for all partition-based approaches [31]. Furthermore, the access of global counter causes excessive cross-core communication overhead, making prior solutions perform poorly at large core counts. There are a few more related studies [13, 8], which are omitted here as they are similarly based on locks but missing details on how they execute state transactions.

4. SYSTEM DESIGN

To address the scalability limitations found in existing solutions, *TStream* adopts an asynchronous state management paradigm, which generally splits the processing of each event into two phases, including a stream processing phase and a transaction processing phase. In the rest of this section, we first give an overview of the two key designs of *TStream* (Section 4.1). Then, we discuss how the stream processing phase, transaction processing phases and the switching between them are designed and optimized (Section 4.2, 4.3). Finally, we discuss design tradeoffs and compare *TStream* with other DSPSs (Section 4.4).

4.1 Overview

Asynchronous State Management. The parallelism opportunities increase with more concurrently issued transactions to the system as shown previously in Figure 1(a). To accumulate more transactions, *TStream* allows an executor to *skip* the issued state transaction, and immediately process subsequent input events, which generates more state transactions. Figure 4(a) illustrates the workflow of an executor under *TStream*, which is explicitly split into two phases. During the stream processing phase, an additional postponing phase is invoked once a state transaction is issued to ensure all postponed transactions (and events) are trackable later. During the transaction processing phase, an additional post-processing phase is invoked after all postponed transactions are evaluated. This is to perform post-operations of the input event that are depended on the transaction evaluation results (i.e., the last step of the three-step procedure discussed in Section 1). The switching between two phases are triggered periodically by punctuation [39] signals so that multiple events are processed consecutively in the stream processing phase, and their postponed transactions are evaluated together with abundant parallelism opportunities during the transaction processing phase.

Operation Chains Parallel Processing. The main idea behind of how *TStream* processes postponed state transactions is that the `ACID+O` properties essentially determine the execution sequences of conflicting operations of each state, where the sequence is given by timestamp. For example, a write operation must precede a read operation

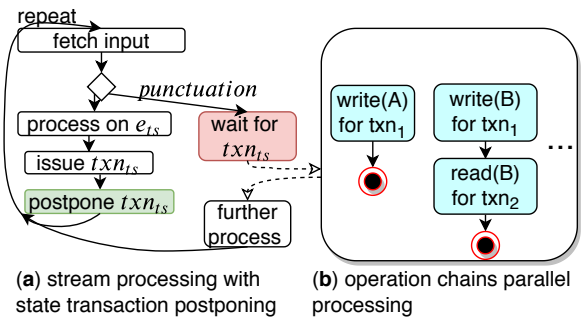


Figure 4: Execution workflow overview of *TStream*.

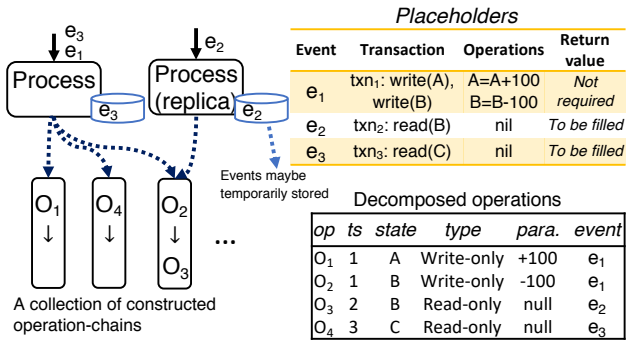


Figure 5: Updating placeholders and constructing operation chains.

of the same state if the write operation is triggered by an earlier input event. Since read/write sets information is already provided to the system via input events, conflicting operations among concurrent transaction can be resolved correctly and efficiently without any locks. This is achieved by decomposing all concurrent transactions into atomic state access operations, each target at one state. Operations targeting at the same state are then grouped together and form a list of operations in the ascending order of their timestamps, called operation chain. Figure 4(b) shows an example where *write(A)*, *write(B)* of txn_1 and read(B) of txn_2 are decomposed and regrouped into two operation chains, which can be processed concurrently without any centralized contention points as there is no cross-state dependency between them. We discuss how to handle cross-state (i.e., cross-chain) dependency shortly later in Section 4.3.2.

4.2 Stream Processing Phase

TStream fetches and processes each event during its stream processing phase. Then, it postpones issued state transactions instead of waiting for state transaction to be evaluated. This postponing process contains two steps, updating placeholders and operation chain construction, described in detail as follows.

4.2.1 Updating Placeholders

A key design decision in *TStream* is to maintain a thread-local structure, called *placeholders*, to track information (e.g., read/write sets) and evaluation results of each postponed transaction. The placeholder is implemented as an auxiliary data structure of each input event and is automatically created when received by the executor. Once a state transaction is issued, read/write sets of it are extracted from the input event. Subsequently, the executor needs to record all necessary information of that transaction into the corresponding event's placeholder. The top of Figure 5 illustrates an example process of postponing three state transactions from two executors of `Process`. After txn_1 is postponed, the executor can continue work on other input events (e.g., e_3). Events may need to be stored temporarily as they may require further process depending on evaluation results of corresponding state transaction. For example, e_2 has to be marked as *unfinished* and stored locally with the executor since it requires the value of state B for further computation.

4.2.2 Operation Chains Construction

After updating placeholders, executors need to construct operation chains. Specifically, each executor decomposes its postponed state transaction into atomic state access operations, and then insert those operations into targeting operation chain. The bottom of Figure 5 illustrates this process involving three state transactions. txn_1 is decomposed into two operations, O_1 and O_2. Each operation is annotated with timestamp (ts) of its original transaction, targeting state ($state$), access type ($type$), and optional parameters (e.g., O_1 is to increase the value of A by 100). Each event is also embedded (by a reference/pointer) with the corresponding operation, which will be tracked during transaction evaluation. O_2 and O_3 are grouped together to form an operation chain as they target at the same state B. As O_2 has a smaller timestamp than O_3, the chain is sorted as $O_2 \rightarrow O_3$. O_1 and O_4 form another two chains as they target at different states.

Intuitively, any concurrent ordered data structure (e.g., self-balancing trees) can be used to implement the operation chain. However, inappropriate implementation can lead to huge overhead in operation chain construction and processing. We consider the following two properties in implementing the operation chain. First, it must allow insertion from multiple executors simultaneously while still guaranteeing operations (of the same chain) are ordered by timestamp. Second, it only requires a sequential look up rather than random searches. We hence adopt ConcurrentSkipListSet [1] due to its much higher insertion performance and simpler design with smaller constant overhead compared to other alternative designs, such as balance trees [34].

4.3 Transaction Processing Phase

During the transaction processing phase, all postponed transactions are actually processed, and placeholders of each event are updated with desired state value. In this section, we first discuss how switching between phases is implemented, then discuss how *TStream* process postponed transactions.

4.3.1 Switching between Phases

TStream adopts *punctuation* [39] as periodic synchronization signal to trigger the actual process of the postponed state transactions. A punctuation is a special type of event guaranteeing that no subsequent input events have a timestamp smaller than any events a prior of it. Punctuation is periodically injected into each executor, and once all executors received punctuation, they are switched from stream processing phase to transaction processing phase. Figure 6 shows an example workflow

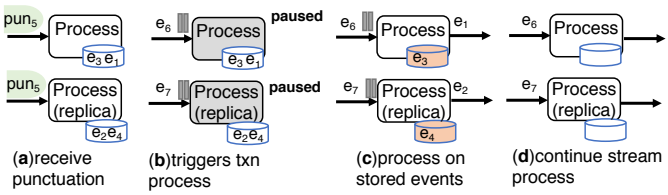


Figure 6: Example workflow of switching between phases.

of switching between phases. (a) punctuations with a timestamp of 5 are injected at every executor of `Process`; (b) all executors are paused and no further input events (e.g., e_6, e_7) are allowed to enter the system. Subsequently, actual transaction processing is started to process all postponed state transactions; (c) when all postponed transactions are processed, executors will be notified to process all stored *unfinished* events, whose placeholders now contain the value of desired states. This step is also called *post-processing phase*; (d) finally, executors are switch back to stream processing phase to process more input events.

To ensure correctness, *all* executors are paused until *all* postponed transactions are evaluated. This is achieved by a *Cyclicbarrier* [2] in *TStream* (please see detailed implementation in Section 5). Such synchronously switching between stream processing and transaction processing introduces synchronization overhead among executors in *TStream*. This can be potentially addressed by adopting an optimistic execution strategy [43]. For example, one can speculatively resume stream processing of an executor without waiting for all operation chains are processed. However, it brings more control complexity to the system to ensure correctness. Due to its considerable complexity, we defer it as future work.

Timestamp Generation. For simplicity, we assign each input event (and punctuation) a unique monotonically increasing timestamp before processing. This is not a limitation of *TStream*, but a limitation of existing techniques since they require to *sequentially* insert locks upon receiving each event. *TStream* only requires punctuation's timestamp to be monotonically increasing in order to progress correctly. Specifically, input events between two subsequent punctuations may arrive arbitrarily with out-of-order timestamp sequence as their issued transactions will be decomposed and sorted automatically during operation chain construction.

4.3.2 Operation Chains Processing

By adopting punctuation, *TStream* has the complete knowledge of all state transactions with a timestamp smaller than the current punctuation since all future events are guaranteed to have larger timestamp than it. In other words, *TStream* only needs to concentrate on improving processing throughput of every batch of state transactions arrived between two consecutive punctuation. As previously discussed, operation chains are constructed during the stream processing phase. Subsequently, there are two cases that we need to consider for processing constructed operation chains.

First, when there is no cross-state dependency in the workload, we can simply sequentially walk (i.e., evaluate) through each operation chain starting from the operation with the smallest timestamp, and different operation chains can be processed concurrently. With a sufficiently large punctuation interval, many operation chains are

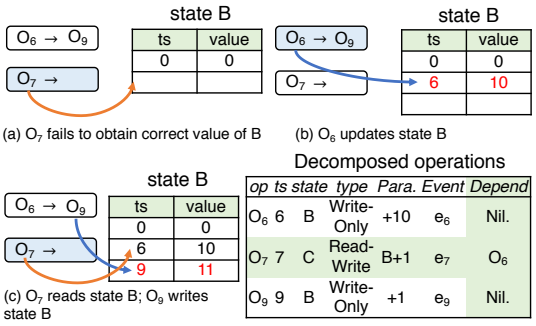


Figure 7: Handling cross-state operation.

formed allowing the system to achieve a high processing concurrency. Furthermore, for write-only and read-modify-write transactions, *TStream* encodes the modification function with the operation itself. During operation chains processing, the modification will be executed without returning the state value to executor reducing round-trip communication overhead. Conversely, if an event requires the state value for further computation, then *TStream* has to write back the state value to its placeholder during transaction processing.

Second, when a cross-state dependency is presented in the workload, we need to first encode dependency information with the operation, then the operation will track the correct version of the dependent state during processing. For example, 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 handle such cross-state 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 staled version of state B violating `ACID+O` properties.

Cross-state Dependency Handling. *TStream* handles such cross-state dependency from two aspects, and an example of handling txn_7 is illustrated in Figure 7. *1) Encoding:* *TStream* encodes dependency information and all update operations of the depended state inside the corresponding operation. For example, O_7's dependency on B is encoded as its parameters as shown in the bottom right of Figure 7. It also needs to record all update operations of the depended state. This can be done by traversing operation chain of depended state. In this example, O_6 is recorded as it has smaller timestamp than O_7. Such an encoding step has to be performed only when punctuation is received by all executors and before actual transaction execution starts. This ensures all dependency information is encoded correctly. *2) Querying:* During transaction execution, one executor/thread will eventually encounter operation with dependency information encoded. It then queries the correct value of the depended state. Figure 7(a) shows that O_7 fails to obtain correct value of state B as O_6 has not been executed. The thread can switch out and evaluate other operation chains. After O_6 is executed and the value of B is updated (Figure 7(b)), O_7 can then continue its process. Note that, *TStream* keeps *multiple versions* of each state updated at different timestamp to allow write without blocking by reading. Therefore, the process of O_7 will not block the process of O_9 (illustrated in Figure 7(c)). After the current batch of transactions is processed, all versions except the latest version are expired and can be safely garbage collected.

Both encoding and querying steps are costly and *TStream* is hence expected to perform better when such cross-state dependency is not presented in the workload. In our experiments, we evaluate *TStream* with applications w/ and w/o cross-state dependency.

4.3.3 Processing Optimizations

NUMA-Aware Processing. Following previous works [33, 32], we consider three different design options for processing operation chains targeting on multi-sockets multicore architectures. *1) Shared-nothing:* In this case, we maintain a task pool of operation chains per core. Essentially, operations are dynamically routed to predefined cores by hash partitioning. Then, only one executor is responsible for processing operation chains of one core. The benefits of such configuration are that it minimizes cross-core/socket communication during execution but it may result in workload unbalance. *2) Shared-everything:* In this case, we maintain a centralized task pool of operation chains, which is shared among all executors to work with. *3) Shared-per-socket:* In this case, we maintain a task pool of operation chains per socket. Executors of the same socket can hence share their workloads, but not share across different sockets.

Work-Stealing. Workloads are shared among multiple executors under shared-everything and shared-per-socket configuration. A simple strategy is to statically assign an equal number of operation chains (as tasks) to every executor. Such static approach may not always achieve good load balancing. Dynamic work-stealing [12] can be applied to achieve better load balancing, where multiple executors (in the same sharing group) continuously fetch and process operation chain as a task from the shared task pool. Such configuration achieves better workload balancing but pays more for cross-core (and cross-socket in case of shared-everything configuration) communication overhead compared to shared-nothing configuration. In our experiments, we evaluate *TStream* with different configurations.

4.4 Discussion

Latency vs. throughput. *TStream* focuses on achieving a reasonable latency level, with high throughput. Compared to existing approaches, *TStream* do not immediately process each issued state transaction. Instead, it processes a batch of state transactions periodically due to the asynchronous state management paradigm. The interval size of two subsequent punctuation (i.e., punctuation interval) hence plays an important role in tuning system throughput and processing latency. Having a large interval, the system needs to wait for a longer period of time to start transaction processing, which increases worst-case processing latency since some events are waiting (i.e., stored with executor) for their issued transactions to be actually processed. Conversely, having a small interval size, the system throughput may drop with insufficient parallelism to amortize synchronization overhead. We evaluate the effect of punctuation interval in our experiments.

System Comparison. We list the comparison to some related systems in Table 2. DSPSs like Storm [5] and Flink [4] provide simple API to express streaming application but scale poorly on modern multicore processors [48]. StreamBox [30], BriskStream [49], and Saber [27] are able to achieve very high throughput and low latency of stream processing, but they are lack of built-in support of transactional state management. SDG [18] manages large mutable state distributedly but does not provide transactional guarantees nor order-preserving consistency. Note that, Ledger [6] recently introduces transactional state management to Flink but is unfortunately close-sourced and little information has been exposed to the public. The closest works to *TStream* are those DSPSs supporting `ACID+O` properties [40, 29], which are however limited at their scalability on modern multicores as we have discussed in Section 3. Thus, we experimentally compare *TStream* with those prior solutions guaranteeing `ACID+O` properties, and omit the comparison with others (e.g., Flink and SDG).

Table 2: Comparisons in related DSPSs

Systems	Storm [5], Flink [4]	BriskStream [49], Saber [27], StreamBox [30]	SDG [18]	LWM [40], PAT(S-Store) [29]	TStream
`ACID+O` properties	✗	✗	✗	✓	✓
Large State	✗	✗	✓	✓	✓
Multicore, manycore	✗	✓	✓	Limited	✓

5. IMPLEMENTATION DETAILS

TStream is built on *BriskStream* [49] – a multicore optimized general purpose data stream processing system. *BriskStream* supports the same APIs of Storm, and we extend it with a transaction processing engine developed based on Cavalia [43] database, which is tightly integrated with *TStream* without cross-process communication. We implement multiple existing algorithms including *LAL*, *LWM* and *PAT* (S-Store) into the transaction processing engine. This allows us to abstract away the implementation details of each approach and concentrate on the algorithm.

Transactional APIs. In *TStream*, we use transaction manager (`TxnManager`) as the interface of the transaction processing engine in each operator that allows multiple executors to access shared mutable states without worrying state consistency violation. This is exemplified by Algorithm 1 that describes how `Process` manages item table under *TStream* as discussed previously in Section 2.

As Line 1, application developer needs to initialize a `TxnManager` as the interface to access shared states. At Line 2, the process is repeatedly involved for every input tuple. At Line 3∼13, the operator consecutively processes each event in the stream processing phase. At Line 4, pre-processing is conducted on each event. At Line 6∼9, when either Read/Write to shared state is called, event's placeholder is updated (Section 4.2.1) and operation chains are constructed (Section 4.2.2) along the way. At Line 11, punctuation signal triggers the switching from stream processing to transaction processing phase (Section 4.3.1), and the details are given in Line 14∼17, where executors need to first synchronize with each other before going to evaluate all constructed operation chains. At Line 17, they also need to be synchronized again to ensure all operation chains are fully processed (Section 4.3.2). At Line 12, post-processing is conducted on all stored events, which now contain their desired state value in their placeholders. Stored events are cleared at Line 13, and then executor returns to the stream processing phase. Both pre- and post-process are user-defined functions. The explicit separation of pre- and post-process is just for illustration purpose and can be automatically analyzed by the system.

6. EVALUATION

Algorithm 1: Code template of `Process` operator of our running example under *TStream* scheme

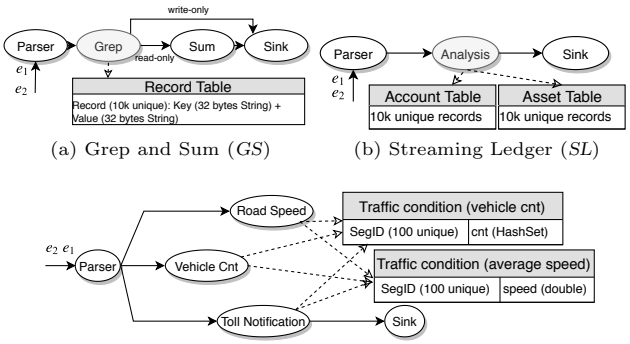
```
Data: TxnManager tm
Data: Configurations cfg
Data: Stored_Events E
// Initilization
1  tm.INITILIZE("ItemTable", cfg);// creates a TxnManager of
     specific algorithm, and initilize shared states
2  foreach input tuple t in input stream do
3      if t is a streaming event then
           /* Stream processing phase                          */
4          PRE_PROCESS(t);// e.g., extract keys of states
             to access from t
5          if t.ReadRequest then
6              tm.READ("ItemTable", t.keys,
                 t.placeholder);// State value will be added
                 to t.placeholder during transaction
                 processing
7              PUSH(t, E);// store ''unfinished'' events
8          else
               /* In case of a read-modify-write request, the
                  modification function is passed as
                  parameter.  Here, we use a write-only as an
                  example, and t.values are passed to update
                  targetting state.                           */
9              tm.WRITE("ItemTable", t.keys, t.values);
10     else
           /* Transaction processing phase                     */
11         START_TXN(t);// t is a punctuation signal
12         POST_PROCESS(E);
13         E.CLEAR();
14 Function START_TXN(t):
       Data: CyclicBarrier barrier
15     barrier.AWAIT();// Ensure all operation chains are
         fully constructed
16     TXN_EVALUATE();// Operation chains parallel
         processing
17     barrier.AWAIT();// Ensure all operation chains are
         fully processed
```

In this section, we discuss a detailed experimental evaluation using four applications. We show that *TStream* manages to better exploit hardware resources compared to the state-of-the-art.

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 4.11.0-rc2. The number of cores devoted to the system and the size of the punctuation interval are system parameters, which can be varied by the system administrator. We vary both parameters in our experiments. We pin each executor on one core and divide the shared states evenly in each executor during initialization. We hence devote 1 to 40 cores to evaluate the system scalability.

Following the previous work [45], we also report how much time is spent on each state transaction. 1) *Useful* is the time that the transaction is really operating on records. 2) *Lock* stands for the total amount of time that a transaction spends due to lock insertion. 3) *RMA* stands for remote memory access overhead that a transaction spends in global counter access (in case of *LAL*, *LWM*, and *PAT*) or in operation chain construction (in case of *TStream*). Index lookup and actual state access also cause remote memory access for all configurations run on multi-sockets. 4) *Sync* is the time that a transaction spends due to synchronization. This is a unique component pay for guaranteeing `ACID+O` properties, where a transaction may need to synchronize among executors per event in *LAL*, *LWM* and *PAT* or synchronize among executors per punctuation in *TStream*. 5) *Others* stands for all other system overheads including index lookup (w/o RMA), context switching, tuple creation, etc.



(a) Grep and Sum (*GS*) (b) Streaming Ledger (*SL*)

(c) Toll Processing (*TP*)

Figure 8: Three more applications in our benchmark workloads.

6.1.1 Benchmark Workloads

There is no standard benchmark for testing transactional state management during stream processing. We hence design our own benchmark including both simple application (*GS*) to be used as microbenchmark and three applications (*OB*, *SL* and *TP*) used in prior works [13, 6].

Grep and Sum (GS): Figure 8(a) shows the topology of *GS* containing four operators. `Parser` continuously feeds input events to `Grep` operator. `Grep` issues a state transaction (read-only or write-only) to access a table of records. If an event triggers a read-only transaction, `Grep` forward the input event with the returned state values to `Sum`; otherwise, it updates the state with value extracted from input event and forwards the input event. `Sum` performs a summation of the returned state values from `Grep`, followed by n times of summation of *system.nanotime*() to simulate a tunable post-process operation, where n is tunable parameter. After `Sum` finishes computation, it emits the computation results as one event to `Sink`. We use `Sink` to record the output stream from `Sum` and `Grep` to monitor system performance. A table of 10k unique records is shared among all executors of `Grep`. Each record has a size of 64 bytes, and every state transaction access 10 records. Despite its simplicity, *GS* shall be applicable to cover a wide range of different workloads by varying parameters such as the Zipfian skew factor of keys (*theta*), read-write ratio of state access and state partition. The post-process complexity (n) can be also varied to vary the stream processing ratio. We vary all those factors in our experiments in Section 6.3 to study different workload configurations.

Streaming Ledger (SL): SL processes events describing wiring money and assets between different user accounts. This application is suggested by a commercial DSPS, namely Streaming Ledger [6]. Detailed descriptions can be found in their white paper [6] and are omitted here for brevity. We implement it with three operators as depicted in Figure 8(b). `Parser` parses received input events and forward to `Analysis`, which updates user accounts (and/or asset records) based on the received events. The process results are further passed to `Sink`. `Sink` reports the processing results to end users. The account and asset tables (each has 10k unique records) are shared among all executors

of `Analysis`. 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 request touches four records) and is two for deposit request.

Online Bidding (OB): We have described *OB* [36] as our running example in Section 2, and we discuss its configurations used in our experiments. `Auth.` and `Process` have a selectivity of one, that is, they always generate one output for each input event. The length of both the alter and the top request is 20 so that each request accesses 20 items. The length of the bid request is one. The price and quantity of each item are randomly initialized before execution and is kept the same among different tests. The ratio of the bid, alter, and top requests is configured as 6:1:1.

Toll Processing (TP): We implement toll processing query (*TP*) from the Linear Road benchmark [9]. *TP* calculates a toll every time a vehicle reports a position in a new segment, in which tolls depend on the level of congestion of the road. The original toll processing query requires to maintain segment statistics (e.g., average speed and count) for every road segment of every expressway for every minute in the last five minutes. To support such semantics, we need to adopt an automatic tuple expiration mechanism in shared state management and refresh states for every minute. For simplification, we defer such time-varying feature as future work and we only store the latest statistics of each road segment. That is, the average speed and count of a road segment are maintained and updated for every vehicle position report regardless of the time evolving. We implement *TP* with five operators as depicted in Figure 8(c). `Parser` parses each input event into position report containing timestamp, a vehicle's current geographic position and speed information. Each position report is broadcast to three downstream operators. `Road Speed` computes the average traffic speed of a road segment and updates the traffic condition information. `Vehicle Cnt` computes the average number of unique vehicles of a road segment, and updates the traffic condition information. `Toll Notification` computes the toll of a vehicle referencing to the traffic speed and number of unique vehicles of the road segment, where the vehicle is. Those three operators need to manage two tables which are shared among all of them (and their executors), one for recording average road speed, and one for recording count of vehicles. It is possible to implement *TP* without utilizing such shared states [35]. However, state consistency has to be left to the application developer to handle – often rely on locks and tuple sorting to ensure event processing sequence [35]. In this work, we focus on improving stream processing performance with transactional state management and omit the case without shared states. we use the dataset from previous work [3], which accesses 100 different segments with a skew factor of around 0.2. Transaction length of *TP* is one for each table, hence there is no multi-partition transaction. However, transactions access each road segment *non-consecutively*, and hence the global counter of each partition need to be increased multiple times after each transaction finishes.

In summary: Our benchmark covers different aspects of application features. First, our applications cover different runtime characteristics. Specifically, *TP* spends 39% of total time (run at a single core) in stream processing, and this ratio is 29% and 22% for *SL* and *OB*, respectively. *GS* spends relatively less time in stream processing (13%), and more time is spent in shared state access. Second, they cover different types of state transactions. Specifically, *GS* has read-only and write-only request. *SL* has write-only and read-modify-write (w/ cross states dependency) request. *OB* has write-only, read-modify-write (w/o cross states dependency) request. *TP* has read-modify-write (w/o cross states dependency) request.

Query Implementation. Thanks to the usage of shared states, *TStream* do not need to rely on key-based stream partitioning [25] as each executor/operator can access any states. We can hence fuse all operators into a single joint operator [46, 22] for all of our testing applications to minimize the impact of cross-operator communication. Input data are simply passed to each executor in a round-robin manner and we use a switch-case statement in the joint operator to handle different types of input events targeting at different original operators. Subsequently, *TStream* allows the joint operator to be scaled to any number of executors without worrying state consistency violation. For more complicated application where operators cannot be fused, we can rely on existing cost-model guided query optimizer [49] to decide suitable parallelism configuration of each operator.

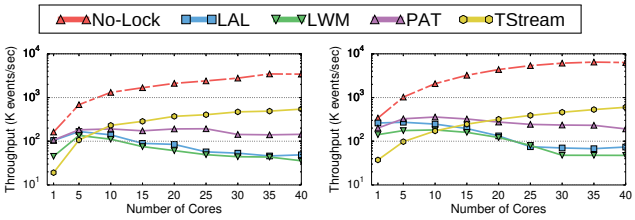
6.1.2 Evaluation Overview

We first show the overall performance comparison of different algorithms on different applications (Section 6.2). Then, by taking *GS* as an example, we further compare different algorithms under varying workload configurations (Section 6.3). We use shared-nothing without work stealing as the default configuration in Section 6.2 and 6.3 under *TStream* scheme. We perform sensitivity study to understand the design trade-off in *TStream* in Section 6.4.

6.2 Overall Performance Comparison

We first compare *TStream* with other schemes on different applications with varying physical resources. Besides three competing schemes, we also examine the system performance when locks are completely removed from *LAL*, which is denoted as No-Lock representing the system performance upper bound. For *GS*, *SL* and *OB*, we set state access skew factor to be 0.6, and the length and ratio of multi-partition transaction is set to 4 and 25%, respectively under *PAT*. That is, each multi-partition transaction needs to access four different partitions, and 25% transactions are multi-partition transactions. We also configure their state accesses to each partition with monotonically increasing timestamp of one so that the global counter of each partition is increased exactly once after each access.

Performance comparison. The throughput comparison results are shown in Figure 9, and there are four major observations. First, *TStream* performs poorly at small core counts. This is expected as it has large constant overhead in operation chain construction. We hence advocate the usage of simpler implementation when there are limited physical resources. Second, *PAT* performs generally better than *LAL* and *LWM* as it avoids blocking when transactions access different partitions. However, *PAT* performs even worse than *LAL* for *TP* although there are no multi-partition transactions. The reason is that transactions access each road segment non-consecutively, and executor needs to increase the global counter of each partition by multiple times (instead of once in case of consecutive access) in order to allow the next



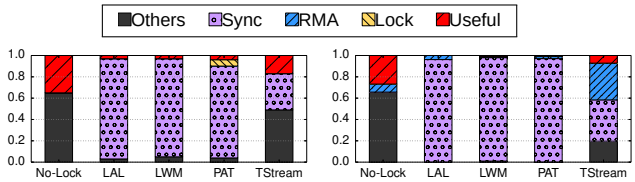
(a) *GS* (50% read requests) (b) *SL*

(c) *OB* (d) *TP*

Figure 9: Throughput (log-scale) comparison of different applications under different execution schemes. There are 25% multi-partition transactions for *GS*, *SL* and *OB*.

transaction to proceed. As a result, excessive access to global counters degrades system performance significantly. Third, *TStream* outperforms all other schemes while preserving the same consistency properties at large core counts for all applications except *SL*. *TStream* successfully exploits more parallelism opportunities thanks to its asynchronous state management paradigm – instead of just looking for parallelism opportunities among up to 40 concurrent transactions, it accumulates more transactions and evaluates them together. However, there is still large room to further improve *TStream* to achieve the performance upper bound indicated by No-Lock scheme. Fourth, there is a significantly increased synchronization overhead due to cross-state dependency handling in *TStream* for *SL*, which prevents *TStream* from scaling well. Prior techniques such as optimistic execution strategy [43] may be adopted to reduce synchronization overhead and further improve system performance under such case. Due to its considerable complexity, we defer it as future work.

Runtime breakdown. We use *TP* as an example to study the runtime breakdown in different algorithms. Similar observations are made in the other three applications. Figure 10 compares the time breakdown when the system is run on single or four sockets. There are two major takeaways. First, No-Lock spends more than 60% of the time in *Others*, which prevents the system to further scale. Our further investigation reveals that the centralized index look-up is the root cause of such performance degradation. We defer the study of more scalable index design in future work and concentrate on concurrent execution control in this work. Second, *Sync* overhead dominates all lock-based algorithms (*LAL*, *LWM*, and *PAT*) regardless of the NUMA effect. Although *LWM* does spend less time in *Sync* compared to *LAL* as read may not be blocked by write resulting in higher processing concurrency, it spends more time in reading and updating the `lwm` variable (grouped in *Others* overhead). Third, NUMA overhead is significant in *TStream* when running on four sockets. Our fine-grained operation-chains based processing paradigm exploits high parallelism opportunities while still guaranteeing the same consistency requirement as prior solutions. Unfortunately, it also brings high communication



(a) Single socket (b) Four sockets

Figure 10: Runtime breakdown per state transaction of TP.

overhead among executors as each executor needs to dynamically partition and regroup decomposed operations into targeting operation chains. Such communication overhead is unavoidable during *operation chain construction* since each executor may issue transaction that touches *arbitrary* states. Nevertheless, as discussed in Section 4.3.3, *TStream* supports multiple NUMA-aware configurations for *operation chain processing*. We study their effectiveness later in Section 6.4.

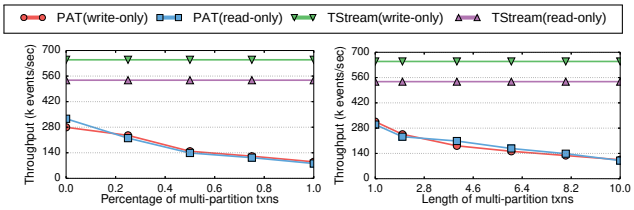
6.3 Workload Sensitivity Study

We now use *GS* as an example to evaluate different algorithms under varying workload configurations in detail.

Multi-partition transaction percentage. We first study the effect of states pre-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 [31], the performance of *PAT* is heavily depended on the length and ratio of multi-sites transactions.

We 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 then vary the percentage of multi-partition transactions in the workload. The results are shown in Figure 11(a). Since *PAT* is specially designed to take advantage of partitioning, it has a much lower overhead for synchronization when no multi-partition is required (i.e., ratio=0%). However, it performs worse than *TStream* even without any multi-portion transaction. This is due to its synchronous execution model, which overlooks parallelism opportunities within the processing of a single transaction/event. Furthermore, as expected, its performance degrades with more multi-partition transactions as they reduce the amount of parallelism. A similar observation can be found in Figure 11(b), where we vary the length of multi-partition transactions and fix its ratio to be 50%. Note that, there is no difference in performance whether or not the workload contains write request under the *PAT* scheme as transactions are always bottlenecked by partition locks. In the following studies, we set the multi-partition ratio to be 50% with a length of 6 under *PAT*.

Read request percentage. We now vary the percentage of events that trigger read request to shared states from 0% (write-only) to 100% (read-only). In this study, we disable stream computation (n=0) of *GS* and focus on evaluating the efficiency of state management. We also set the key skew factor to be 0, and hence the state is accessed with uniform frequency. Figure 12(a) shows the results and there are two major observations. First, varying read/write requests ratio has a minor effect on system performance under prior schemes, *LAL*, *LWM* and *PAT*. This is because their execution runtime is dominated by synchronization overhead



(a) Varying ratio of multi-partition txns (length=6). (b) Varying length of multi-partition txns (ratio=50%).

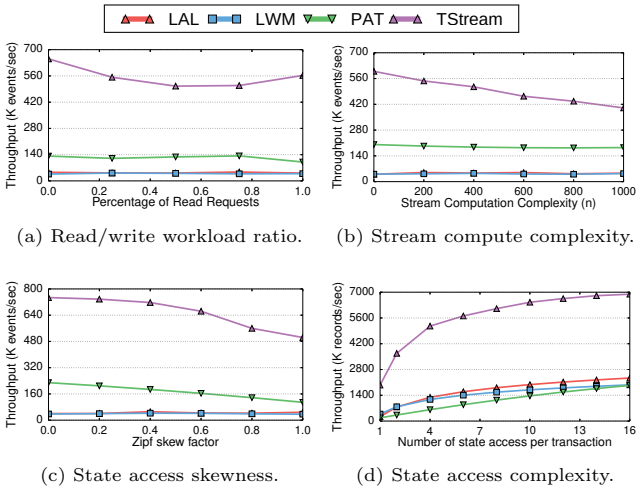
Figure 11: Multi-partition transaction evaluation.

(i.e., due to *LockAhead* and *LockP* process). Second, *TStream* performs generally worse with more read request. This is because the system has to write back the state value to event's (which triggers read request) placeholder after transaction evaluation is finished. In contrast, *TStream* pushes down evaluation and placeholders of events with the write-only request will not be updated and round-trip communication cost is reduced. An interesting point to take note is that *TStream*'s performance increases slightly under read-only workload compared to mixture workload. This is primarily due to the hardware prefetcher, which helps in reducing communication cost. When there are both read and write to shared states, hardware prefetchers are not effective as each prefetch can steal read and or write permissions for shared blocks from other processors, leading to permission thrashing and overall performance degradation [23].

Stream processing complexity. We now vary complexity (n) in `Sum` operator in this test. We configure a read/write request ratio to be zero (read-only) to focus on study the effect stream processing complexity. The results are shown in Figure 12(b). Again, we see that existing approaches perform similarly as their runtime is dominated by synchronization overhead. When state management is heavy (i.e,. n is small), *TStream* performs significantly better than other schemes. This is because *TStream* successfully amortizes the synchronization overhead caused by state management among a batch of state transactions. As expected, its performance drops with larger n due to the increased workload per event.

State access skewness. In this study, we configure a write-only workload to examine how algorithms perform under contented state update. 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*. Figure 12(c) shows that *TStream* achieve high performance even under high skewness. The reason is that *TStream* is still able to discover sufficient parallelism opportunities among a large set of transactions (punctuation interval is set to 500 in this test). *LWM* performs even worse under contented workload as there is more contention on updating `lwm` counter of the same record. *PAT* performs worse with increasing skewness because of the more intensive contention on the same partition lock.

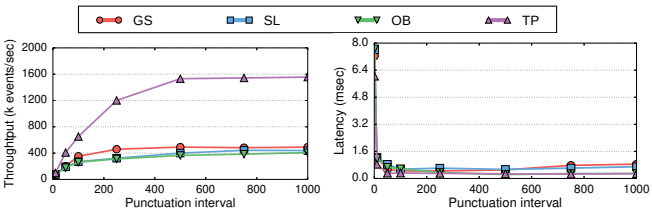
State access complexity. The number of tuples accessed by a transaction (i.e., transaction length) is another factor that impacts scalability. Following previous work [45], we measure the number of tuples accessed per second, rather than transactions completed. In this study, we configure



(a) Read/write workload ratio. (b) Stream compute complexity.

(c) State access skewness. (d) State access complexity.

Figure 12: Varying application workload configurations of *GS* (50% multi-partition with length of 6).



(a) Throughput (b) 99^{th} percentile latency

Figure 13: Effect of varying punctuation interval.

a write-only workload to examine how algorithms perform under the varying length of a transaction from 1 to 16. We also set theta to be 0.6. The results are shown in Figure 12(d). As expected, all schemes achieve higher state access throughput with increasing working set size because costs of processing of one event (except transaction processing) are effectively amortized among more state accesses. However, prior lock-based algorithms scale poorly as they failed to explore the increasing parallelism opportunities within each transaction.

6.4 System Sensitivity Study

In this section, we study the effect of varying punctuation interval (Section 4.3.1) and effect of processing optimizations (Section 4.3.3) in *TStream*.

Varying punctuation interval. The number of transactions to handle between two subsequent punctuation plays a critical role in *TStream*'s performance. Figure 13(a) shows that the performance of *TStream* generally increases with larger punctuation interval. When it is larger than 500, *TStream* achieves its best performance for all applications. It also shows that large punctuation interval is especially beneficial for TP. The reasons are two folds. First, there are only 100 different road segments, and transactions are heavily contented within a small batch of transactions. Second, TP has the highest stream processing ratio, *TStream* achieves a higher stream processing concurrency with large punctuation interval, and the system performance hence increases significantly.

Following the previous work [16], we define the end-to-end latency of a streaming workload as the duration

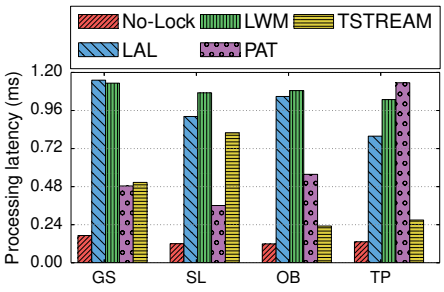


Figure 14: 99^{th} percentile end-to-end processing latency.

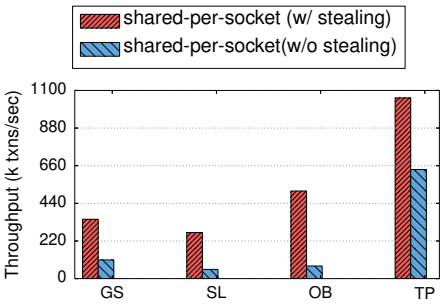


Figure 15: Effect of work-stealing.

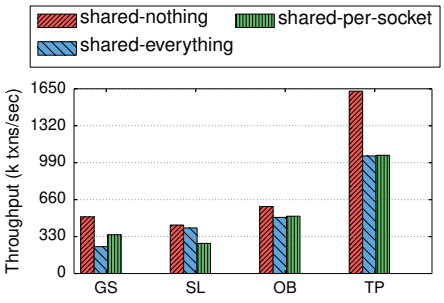


Figure 16: Varying processing configurations.

between the time when an input event enters the system and the time when the results corresponding to that event is generated. Figure 13(b) shows the processing latency of *TStream* under varying punctuation interval. Thanks to the significantly improved throughput, *TStream* achieves very low processing latency. When *TStream* achieves its best performance (around punctuation interval of 500), its processing latency is only around 0.23∼0.56 ms, which is much smaller than many practical use cases [16], hence *TStream* is practically useful. Further increasing punctuation interval, the latency becomes worse as there is no significant throughput improvement. Figure 14 further shows that *TStream* achieves comparable and sometimes even lower processing latency compared to the best existing scheme when punctuation interval is 500.

Effect of processing optimizations. We now compare different operation chains processing configurations of *TStream* including share-nothing, share-everything, and shared-per-socket. Work-stealing can be further enabled in the latter two configurations. By taking shared-per-socket as an example, we show in Figure 15 that work-stealing does significantly improve the system performance, and shall be enabled when the shared configuration is applied. However, Figure 16 shows that *TStream* achieves the best performance for all applications under shared-nothing configuration. This indicates that cross-core and cross-socket communication shall be always avoided for all testing applications. Nevertheless, we plan to investigate more applications that may be more sensitive to workload unbalancing rather than communication overhead.

7. RELATED WORK

We have discussed several prior proposals in Section 3, and we now discuss more related studies.

Transactional DSPSs. Botan et al. [13] presented an *unified transactional model* for streaming applications. However, it is unclear how state transactions are executed in their design of transaction manager. Affetti et al. [8] recently proposed a state consistency model with `ACID+O` properties guaranteed. However, they still rely on locks for guaranteeing state consistency during state transaction execution. Different from the previous implementation, *TStream*'s novel asynchronous state management design has shown to achieve much higher throughput and scalability at different types of applications and workloads. There is also a recent launch of a commercial system, called Streaming Ledger [6] for extending Flink with transactional state management capability. It is close-sourced, and we cannot compare our system with it.

Conventional OLTP Systems. Our work is orthogonal to conventional OLTP systems. Prior works, such as [11, 17, 45, 43, 42, 20] provide highly valuable techniques, mechanisms and execution models but none of them solves the problem we address. Particularly, state transactions in *TStream* are generated by stream operators during their processing of input events, keys of targeting state are extracted from input events, and each transaction bears an external timestamp ordering. Subsequently, *TStream* guarantees `ACID+O` – a property that conventional concurrency control (CC) protocols are not well prepared for. Our operation chain construction and processing are conceptually similar to some deterministic databases [38, 26], which generates a dependency graph that deterministically orders transactions' conflicting access of records. However, the determinism of *TStream* is given by external input events rather than the system itself (i.e., the case of deterministic databases) resulting in different system design and optimization focuses.

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, 47]. The importance of scale-up stream processing is also getting more and more attention recently [48, 27, 30, 49, 46]. *TStream* is built to improve multicore utilization standing on the shoulders of many valuable existing works such as [21, 41, 33]. However, none of the previous work addresses the scalability bottlenecks that *TStream* faces, that is how to scale stream processing under the unique `ACID+O` properties.

8. CONCLUSION

This paper introduces *TStream* aiming at scaling transactional state management in stream processing on shared-memory multicore architectures. *TStream* achieves high scalability via two key designs including 1) asynchronous state management, which allows it to exploit more parallelism opportunities and 2) operation-chains parallel processing, which maximize transaction execution concurrency while guaranteeing `ACID+O` properties without any centralized contention points. To the best of our knowledge, 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.

9. REFERENCES

[1] ConcurrentSkipListSet. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListSet.html.

[2] Cyclicbarrier. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CyclicBarrier.html.

[3] Uppsala University Linear Road Implementations. http://www.it.uu.se/research/group/udbl/lr.html.

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

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

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

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

[8] 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.

[9] 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 Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 480–491. VLDB Endowment, 2004.

[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] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] S. Harizopoulos and A. Ailamaki. A Case for Staged Database Systems. In *CIDR*, 2003.

[22] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, Mar. 2014.

[23] N. D. E. Jerger, E. L. Hill, and M. H. Lipasti. Friendly fire: understanding the effects of multiprocessor prefetches. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 177–188, March 2006.

[24] 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.

[25] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. A holistic view of stream partitioning costs. *Proc. VLDB Endow.*, 10(11):1286–1297, Aug. 2017.

[26] A. Kemper and T. Neumann. Hyper: A hybrid oltp olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206, April 2011.

[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] 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.

[30] 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.

[31] 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.

[32] D. Porobic, E. Liarou, P. Tözün, and A. Ailamaki. Atrapos: Adaptive transaction processing on hardware islands. In *2014 IEEE 30th International Conference on Data Engineering*, pages 688–699, March 2014.

[33] 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.

[34] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449. Springer, 1989.

[35] M. J. Sax and M. Castellanos. Building a Transparent Batching Layer for Storm. 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] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proc. VLDB Endow.*, 3(1-2):70–80, Sept. 2010.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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