Towards Concurrent Stateful Stream Processing on Multicore Processors

Shuhao Zhang[1], Yingjun Wu[2], Feng Zhang[3], Bingsheng He[1]
[1]National University of Singapore, [2]Amazon Web Services, [3]Renmin University of China

***Abstract*—Recent data stream processing systems (DSPSs) can achieve excellent performance when processing large volumes of data under tight latency constraints. However, they sacrifice support for concurrent state access that eases the burden of developing stateful stream applications. Recently, some have proposed managing concurrent state access during stream processing by modeling state accesses as transactions. However, these are realized with locks involving serious contention overhead. Their coarse-grained processing paradigm further magnifies contention issues and tends to poorly utilize modern multicore architectures. This paper introduces *TStream*, a novel DSPS supporting efficient concurrent state access on multicore processors. Transactional semantics is employed like previous work, but scalability is greatly improved due to two novel designs: 1) dual-mode scheduling, which exposes more parallelism opportunities, 2) dynamic restructuring execution, which aggressively exploits the parallelism opportunities from dual-mode scheduling without centralized lock contentions. To validate our proposal, we evaluate *TStream* with a benchmark of four applications on a modern multicore machine. The experimental results show that 1) *TStream* achieves up to 4.8 times higher throughput with similar processing latency compared to the state-of-the-art and 2) unlike prior solutions, *TStream* is highly tolerant of varying application workloads such as key skewness and multi-partition state accesses.**

I. Introduction

The recent advances in data stream processing systems (DSPSs) [1], [2], [3], [4] in terms of performance, elasticity, and scalability have accelerated their adoption in many emerging use cases. Modern stateful DSPSs such as Flink [1], Storm [2], and Heron [3], achieve high performance via disjoint partitioning of application states [5] – often through key-based partitioning [6] so that each execution thread (i.e., executor) maintains a disjoint subset of states and thereby bypass the issue of *concurrent state access*. This type of design can lead to tedious implementation and ineffective performance in many cases (see later in section II-A).

Several recent works propose to support *concurrent state access* in stream processing, where large mutable application states may be concurrently accessed by multiple executors [7], [8]. State consistency is maintained by the system by adopting transactional semantics [7], [9]. Specifically, the set of state accesses triggered by the processing of one input event at one operator is defined as a *state transaction*. Multiple state transactions are concurrently executed using various *concurrency control* mechanisms [7], [10].

Unfortunately, prior implementations are not free of bottlenecks when scaled up due to two reasons: First, they are mostly built on centralized locking schemes, where every transaction has to access a set of monotonically increasing counters to decide if it is allowed to acquire locks of its targeting states. Despite its simplicity, it has serious contention issues and does not properly exploit the underlying multicore nature of modern CPU architectures. Second, they commonly follow a coarse-grained processing paradigm, where an executor must finish all operations of processing one event before the next can begin. This paradigm minimizes context switching overhead but overlooks parallelism opportunities. In particular, the processing of one event may involve multiple conflict-free operations (e.g., stateless computation and multiple accesses to different states). Blocking one state access often unnecessarily blocks all operations of an event in this paradigm, further intensifying contention.

Figure 1 shows the evaluation results of the *PAT* scheme [10], the current state-of-the-art, on the Toll Processing [11] application. We measure the average amount of time spent on (i) *state access*, i.e., time spent accessing states, (ii) *access overhead*, comprising of lock acquisition and blocking due to access contention, and (iii) *others*, including all other operations (excluding state access) and overheads (e.g., context switching). As the number of cores used increases, the overhead of accessing states quickly dominates other operations due to serious contention. Therefore, we need a new solution for scaling concurrent state access in the DSPSs.

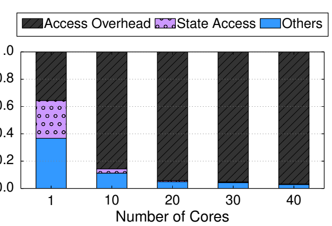


Fig. 1: Severe lock contentions of the *PAT* scheme [10].

This paper presents *TStream*, a novel DSPS supporting efficient concurrent state access in the context of main memory multicore architectures. *TStream* follows previous work [7], [9] of employing transactional semantics to handle concurrent state access but achieves much better scalability. The design of *TStream* is inspired by our careful analysis of existing applications. Stream processing usually consists of a set of operations that are repeated for every input event, and concurrent state access (if applied) often turns out to be a performance bottleneck. This pattern guides us to abstract the processing as a *three-step procedure*: preprocess, state access, and postprocess. While this formulation may appear to limit the flexibility of programming, it unlocks the potential for simple and effective optimization opportunities, which we exploit to improve scalability.

First, based on the three-step procedure, we propose an execution strategy called *Dual-Mode Scheduling*, which exposes more parallelism opportunities. By carefully decoupling the second step (i.e., state access) from the processing logic, *TStream* allows an executor to *postpone* state access and instantly work on other input events without being blocked. Delaying state transactions to the last minute allows them to be processed in batches, enabling further optimizations when accessing state.

Second, we propose a novel state transaction processing mechanism called *Dynamic Restructuring Execution*. Specifically, *TStream* restructures a batch of (postponed) transactions into a collection of sorted lists called *operation chains*. These can be evaluated in parallel without lock contention, significantly relieving contention overhead in concurrent state access.

In summary, we make the following contributions: First, we propose an efficient way of handling concurrent state access during stream processing with two novel designs. Second, we implement the proposed designs as well as several state-of-the-art schemes in a fully functional DSPS [12] optimized for multicore architectures. We then compare them both theoretically and experimentally, revealing the scalability issues of prior solutions. We open the full source code of the system and application benchmarks at https://github.com/Xtra-Computing/briskstream/tree/TStream.

II. PRELIMINARIES

A. Data Stream processing

In this paper, we generally follow the definitions of data stream processing presented in [13], and we briefly recall them for completeness. We summarize the terminology used in this work in Table I. Stream processing continuously processes one or more streams of *events*. Each event (e_{ts}) has a timestamp (ts) that indicates its temporal sequence. A streaming application contains a sequence of *operators* that continuously process streaming events [14]. To sustain a high input stream ingress rate, each operator may be spread across multiple *executors* (e.g., Java threads), which handle multiple input events concurrently through stream partitioning [14]. Operators often need to maintain states during processing for future reference [5]. To avoid state access conflict, the common wisdom is adopting key-based stream partitioning [6] so that each executor maintains a *disjoint subset* of states. Similarly, operators are required to maintain their states exclusively. To illustrate this, we use a simplified toll processing query (*TP*) from the Linear Road Benchmark [11] as an example.

Motivating Example. *TP* calculates the toll every time a vehicle reports its position in a new road segment, in which tolls depend on the level of road congestion. It contains three key operators: 1) `Road Speed (RS)` computes average traffic speed of a road segment; 2) `Vehicle Cnt (VC)` computes the average number of unique vehicles of a road segment; 3) `Toll Notification (TN)` computes the toll of a vehicle based on the traffic speed of and number of unique vehicles on the road segment where the vehicle is.

One common way [15] to implement *TP* is shown in Figure 2 (a), where ovals denote operators and arrows denote data flow between operators. `Parser` parses input events into traffic reports containing <timestamp, vehicle_id, geo_position, speed>, and the computed toll is continuously sent to `Sink` for output. Road congestion status (i.e., speed and count) are *application states* [5], which are maintained for future reference (the dashed arrows in the figure). To avoid state access conflict, a key-based partition scheme is adopted to split the input stream (the blue diamond arrows in the figure). It also prevents operators from concurrently accessing the same state by keeping their states exclusive.

However, such an implementation can be tedious and ineffective. *First*, it requires users to carefully partition and sort the input stream by selecting appropriate keys. In this example, the application needs to ensure that any traffic report is processed only when `TN` receives the *updated* road congestion status from `RS` and `VC`. Prior work [15] embeds tuple buffering and sorting operations (i.e., sort by vehicle id, geo-position and timestamp) inside the `TN` as highlighted in Figure 2 (a). This manual approach is cumbersome and can lead to errors if tuples arrive too late (out of buffering limits). *Second*, the ineffectiveness stems from the duplication of large application states among operators. In this example, states maintained by `RS` and `VC` have to be repeatedly forwarded to `TN`.

B. Concurrent Stateful Stream Processing

Many applications utilizing concurrent state access have been proposed covering various domains (e.g., Health-care [7], IoT [9], and E-commerce [10]). Despite the implementation differences, we identified three common design features from many applications.

F1: Three-step procedure. Each operator can be abstracted as a *three-step procedure*: (1) preprocess input event (e.g. filter invalid input); (2) accesses (shared) application states (e.g., read the road congestion status); finally, (3) perform further processing based on access results (e.g., compute toll based on road congestion status).

F2: Determined read/write sets. Read/write sets of each state access is provided as arguments, which are inferred from the input event. For example, which road segment to access is determined by the corresponding traffic report (i.e., key of state to access is $tuple.geo\_position$).

F3: Deterministic state access sequence. State accesses must strictly follow their triggering event's timestamp sequence. For example, computation of a toll must reference to the exact "current" road congestion status, i.e., the toll should depend on neither *stale* nor *future* road congestion status.

An implementation of *TP* utilizing concurrent state access is illustrated in Figure 2 (b). It contains the same operators, but road congestion information is shared among all operators and their executors. Particularly, congestion status for all road

TABLE I: Summary of Terminologies

Term	Definition
Event (e_{ts})	Input stream event with a timestamp (ts) to indicate its temporal sequence
Punctuation (pun_{ts})	Special tuple embedded in a data stream that indicates the end of a subset of the stream
State transaction (txn_{ts})	A set of state accesses (i.e., read and write to application states) triggered by processing of a single input event.
Correct state transaction schedule (S)	A state transaction schedule S of ($txn_{t1}, txn_{t2}, \ldots txn_{tn}$) is correct if it is conflict equivalent to $txn_{t1} \prec txn_{t2} \prec \ldots \prec txn_{tn}$

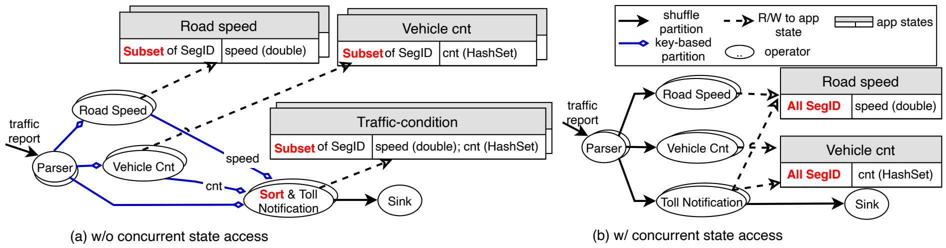


Fig. 2: Implementation of Toll Processing (*TP*).

segments are maintained in two tables. One for the average road speed and the other for the count of unique vehicles. Input events are round-robin shuffled (the solid arrows in the figure) among all executors which concurrently process input events and access to the shared two tables. Such an implementation significantly eases the burden of developing stateful stream application as developers do not need to manually split application states and ensure an exclusive and correctly ordered access among different threads.

Unfortunately, concurrent state access introduces a new challenge of preserving state consistency to DSPSs as multiple threads may concurrently access to the same application state with arbitrary sequence. Previous studies [7], [8], [9], [10] advocate that concurrent state access can be efficiently managed with transactional semantics. We follow prior work [7], [9] and specifically adopt two key definitions.

***Definition** 1 (State Transaction):* The set of state accesses triggered by processing of an event e_{ts} at an operator is defined as one *state transaction*, denoted as txn_{ts}. Timestamp ts of a state transaction is defined as that of its triggering event.

***Definition** 2 (Correct State Transaction Schedule):* A schedule of transactions txn_{t1}, txn_{t2}, ..., txn_{tn} is correct if it is conflict equivalent to $txn_{t1} \prec txn_{t2} \prec \ldots \prec txn_{tn}$. A DSPS ensuring a correct state transaction schedule always guarantees deterministic state access sequence.

C. Existing Solutions Revisited

To ensure a correct schedule of concurrent state transactions, various concurrency control mechanisms have been proposed. In the following, we revisit the representative ones.

*1) **Lock-based approach (LOCK)**:* An earlier study by Wang et al. [7] described a strict two-phase locking (S2PL) -based algorithm that allows multiple state transactions to run concurrently. To maintain a correct schedule, it employs a *lockAhead* process that compares each transaction's timestamp against a monotonically increasing counter to ensure that transaction with the smallest timestamp always obtains locks first, and hence guarantees proper state access sequence. By utilizing determined read/write sets, once a transaction finished inserting its locks, the system can immediately increase the counter to allow next transaction to proceed without waiting for the transaction to finish processing.

*2) **Multiversion-Lock-based approach (MVLK)**:* To relax the rigorous lock incompatibility of the *LOCK* scheme, Wang et al. [7] propose to adopt multiversion concurrency control, where multiple copies of the same application state modified at different timestamps are kept by the system. It further maintains a counter (called `lwm`) of each state to guard the state access order. Specifically, transactions need to compare their timestamp with the corresponding `lwm` counters before proceed. A write is permitted only if the transaction's timestamp is equal to `lwm`; while a read is permitted as long as the transaction's timestamp is larger than `lwm` so that it can read a correct version of the state. During commits, a transaction needs to increase `lwm` of all its modified states.

*3) **Partition-based approach (PAT)**:* S-Store [10] splits application states into multiple disjoint *partitions*, and hence only needs to guard accessing order for transactions targeting the same partition by utilizing determined read/write sets. During execution, each transaction needs to first compare its timestamp with monotonically increasing counters of its targeted partitions (maybe more than one) to ensure that it can proceed to insert locks. It is noteworthy that, despite being partitioned, two transactions can still conflict if their targeted partitions are overlapping. This is fundamentally different from key-based stream partitioning [6].

In Summary: There are two common scalability limitations in prior solutions. First, to ensure schedule correctness, prior approaches compare the timestamp of every state transaction with a set of monotonically increasing counters to ensure that locks are granted in the desired order. Despite its simplicity, such centralized locking schemes can lead to serious contentions, which would severely degrade system performance. Although *PAT* (i.e., S-Store) reduces such overhead when transactions access disjoint state partitions, it quickly devolves to *LOCK* with more multi-partition transactions – a common problem for partition-based approaches [16]. Second, they all adopt a coarse-grained processing paradigm that sequentially evaluates the three-step procedure for each event; an executor (i.e., thread) must complete all operations of one event before starting next. This minimizes potential context switching overhead, but overlooks parallelism opportunities and further intensifying contention.

There are also many other existing concurrency control (CC) schemes [17] that have not been applied to the problem of concurrent state access in stream processing. For example, the timestamp-ordering based (T/O) approach [18], [19] is a popular CC technique that does not rely on locks. However, as they are not designed with awareness of state access order, they are not able to ensure a correct state transaction schedule [20].

III. *TStream* OVERVIEW

In this work, we follow prior work [7], [9] for employing transactional semantics on managing concurrent state access. To better utilize multicore processors, we propose two novel designs. Those designs are largely inspired by the set of common features we identified from our careful analysis of existing applications discussed in Section II-B.

D1: Dual-Mode Scheduling (Exposing Parallelism). We propose an execution strategy called *Dual-Mode Scheduling*, which exposes more parallelism opportunities. Instead of evaluating three steps (Feature **F1**) sequentially for each event as done in the literature, *TStream* decouples the second step and postpones it to be evaluated later. Subsequently, *TStream* involves two modes: 1) the `compute` mode where executors continuously process input events without being blocked due to state access; 2) the `state access` mode where executors collaboratively process a batch of postponed transactions with abundant parallelism opportunities.

D2: Dynamic Restructuring Execution (Exploiting Parallelism). We propose a novel *Dynamic Restructuring Execution* strategy to efficiently evaluate a batch of transactions in the `state access` mode. Leveraging determined read/write sets (Feature **F2**), *TStream* conceptually decomposes each state transaction into multiple operations, each targeting one state. On top of that, with a determined state access sequence (Feature **F3**), *TStream* restructures those operations into timestamp-ordered lists (called operation chains), where one list is tied to one state and evaluated by one thread. With this restructuring, operation chains can be evaluated in parallel, and state access conflict is avoided within the operation chains.

IV. DESIGN DETAILS

In this section, we discuss our designs in detail. We first describe *TStream*'s APIs for users to implement concurrent stateful stream processing applications. Then we discuss the implementation of dual-mode scheduling and dynamic restructuring execution.

A. Programming APIs

In line with many popular DSPSs, *TStream* expresses an application as a DAG (Directed Acyclic Graph) with an API similar to that of Storm [2]. To support concurrent stateful stream processing, *TStream* provides a list of user-implemented and system-provided APIs inside each operator. The former are user implemented based on their application requirements and the latter function as library calls, similar to some existing frameworks [21]. Currently, all APIs are implemented in Java.

User-implemented APIs are summarized in Table II, which requires users to implement an operator as a three-step procedure. We leave full automation of this process for future work. A code template of an operator is shown in Algorithm 1. State transaction is expressed through the `STATE_ACCESS` API, which would be implemented by users using system-provided APIs. Algorithm 2 illustrates an implementation of `STATE_ACCESS` by using `Toll Notification` as an example. All operations (i.e., two in this example) issued from one invocation of `STATE_ACCESS` are subsequently treated as one state transaction. Besides such simple sequential statements, *TStream* also supports conditional statements (i.e., if-else) as well as loops if the loop condition can be determined without accessing to shared states. Please refer to our technical report [20] for more code examples.

Algorithm 1: Code template of an operator

```
boolean dualmode;// flag of dual-mode scheduling
Map cache;// thread-local storage
foreach event e in input stream do
    if e is not punctuation// always true under prior
        schemes
    then
        EventBlotter eb ← PRE_PROCESS (e);// e.g.,
            filter events
        STATE_ACCESS (eb);// issue one state
            transaction
        if dualmode then
            /* stores events whose state access
               is postponed under TStream scheme.
               */
            cache.add(< e, eb >);
        else
            /* evaluates three steps contiguously
               under prior schemes.              */
            POST_PROCESS (< e, eb >);// e.g.,
               computes toll based on obtained
               road statistics
    else
        /* if the event is a punctuation,
           transaction processing can start.   */
        TXN_START()// Triggers mode switching.
        foreach < e, eb > ∈ cache do
            POST_PROCESS (< e, eb >);
```

Algorithm 2: STATE_ACCESS of `Toll Notification`

```
Input: EventBlotter eb
begin
    READ(SpeedTable, eb.ts, eb.geo_pos, eb);// obtain
        average speed of a road segment.
    READ(CountTable, eb.ts, eb.geo_pos, eb);// obtain
        vehicle count of a road segment.
```

System-provided APIs are summarized in Table III `READ`, `WRITE`, and `READ_MODIFY` stand for the atomic operation of a state transaction. For brevity, $table$, $timestamp$, and $EventBlotter$ arguments are omitted in Table III and are shown in Algorithm 2. Key and $Value$ stand for key and new value of the state to access, respectively. opt means that the parameter is optional. Fun stands for a user-defined function such as increment by 1. $CFun$ stands for a user-defined function that determines whether the operation will be applied. Users can implement Fun and $CFun$ by constructing system-provided APIs (e.g., a conditional update depends on a read operation), similar to the way of constructing `STATE_ACCESS`. `TXN_START` is used to indicate mode switching and is only

TABLE II: User-implemented APIs

APIs	Description
EventBlotter **PRE_PROCESS** (Event e)	Implements the pre-process function (e.g., filter). It returns EventBlotter containing parameter values (e.g., read/write sets) extracted from e.
void **STATE_ACCESS** (EventBlotter eb)	Implements the state transaction through constructing system-provided APIs such as $READ$, $WRITE$.
void **POST_PROCESS** (Event e, EventBlotter eb)	Implements the post-process function that is depended on results of state access (stored in EventBlotter).

TABLE III: System-provided APIs

APIs	Description
void **READ** (Key d, EventBlotter eb)	Issues a read request with key of d and store results in eb for further processing (i.e., post-process).
void **WRITE** (Key d, Value v, opt CFun f^*(Key s))	Issues a modify request so that $state(d) \leftarrow v$ if $f^*(s)$ is true or $f^*(s)$ is null. If d!=s, this request involves data dependency.
void **READ_MODIFY** (Key d, Fun f(Key t), opt CFun f^*(Key s))	Issues a read and modify request so that $state(d) \leftarrow f(t)$ if $f^*(s)$ is true or $f^*(s)$ is null.
void **TXN_START** ()	Triggers mode switching to process postponed transactions.

used under *TStream*'s dual-mode scheduling scheme.

B. Dual-Mode Scheduling

As discussed in Section III, *TStream* adopts a nonconventional processing strategy, where the state access step is postponed. There are three key components to support such postponing efficiently and correctly: *1) EventBlotter Maintenance* creates and initializes a thread-local auxiliary data structure called *EventBlotter*, which acts as the data bridge linking the two processing modes; *2) Processing Mode Switching* enables efficient and correct mode switching in *TStream* with punctuation technique; *3) Progress Controller* generates punctuations and assigns timestamps to events and punctuations. *TStream* requires punctuations to contain a monotonically increasing timestamp.

1) EventBlotter Maintenance: A key design decision in *TStream* is to maintain a thread-local auxiliary data structure (implemented as a Java Class), called *EventBlotter*, to track information (e.g., parameter values and processing results) of each postponed transaction. An EventBlotter is created by the system upon exiting `PRE_PROCESS` (i.e., Line 5 of Algorithm 1)). Upon entering `STATE_ACCESS` (i.e., Line 7 of Algorithm 1), *TStream* creates a state transaction with a list of `READ`, `WRITE`, `READ_MODIFY` operations according to users' implementation. As mentioned before, state transaction is not instantly processed under *TStream*'s dual-model scheduling strategy. Instead, those operations are registered to *TStream* to be evaluated later (Section IV-C). Their parameter values (e.g., read/write sets) are stored in the corresponding EventBlotter for future reference during transaction processing. `POST_PROCESS` might be required depending on the result of state accesses. To support this, we store input events and their corresponding EventBlotters in a thread-local map structure (i.e., Line 9 of Algorithm 1), which will be processed after postponed transactions are processed.

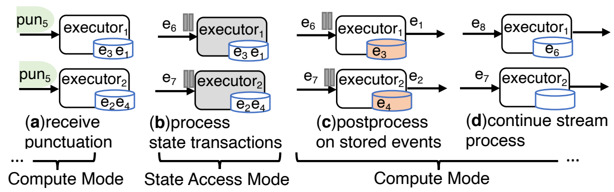


Fig. 3: Example workflow of switching between modes.

2) Processing Mode Switching: *TStream* relies on *punctuation* [22] to periodically switch between two processing modes. A punctuation is a special type of event that guarantees that no subsequent input event will have a smaller timestamp. It is widely used in prior work for out-of-order stream processing [23], [24]. Our usage of punctuation is different from the previous work [23], as we target more fine-grained control at transaction processing rather than event processing. Specifically, input events may be processed in arbitrary order in *TStream*, but their issued transactions must be processed following a correct sequence. A punctuation ensures that any state transaction issued before it should have a smaller timestamp than any one issued after it. This sharply delineates the timestamp boundary of a list of transactions between any two consecutive punctuations and gives *TStream* hints on how to effectively process them.

To ensure the correctness of mode switching, *TStream* artificially adds two barriers (via the *Cyclicbarrier* [25]) to synchronize executors. The first barrier is added after the `TXN_START` is called. This ensures EventBlotter maintenance for all events before the current punctuation is completed. Only when all executors have switched to `state access` mode, can state access begin. The second barrier is added before the `TXN_START` exits. This guarantees the correctness of the postprocessing step as executors do not resume to the `compute` mode until all postponed state accesses are fully processed. By processing transactions in batches, the overhead caused by these barriers will be amortized.

Figure 3 shows an example workflow of switching between modes: (a) executors asynchronously switch to the `state access` mode when they receive punctuation with a timestamp of 5 (i.e., Line 13 of Algorithm 1) and no further input events are allowed to enter the system (e.g., e_6, e_7); (b) subsequently, transaction processing is started (Section IV-C) once all executors are in the `state access` mode; (c) when all postponed transactions are processed, executors are synchronously switched back to the `compute` mode to process (i.e., `POST_PROCESS`) their stored *unfinished* events, whose EventBlotter now contains the value of desired states (i.e., Line 14$\sim$15 of Algorithm 1); finally, (d) executors are asynchronously resumed to process more input events.

3) Progress Controller: Punctuations are periodically broadcast to the input stream of each executor as done in [24]. Punctuations' timestamp must monotonically increase to progress correctly, while events can have arbitrary timestamps as long as they are smaller than the next punctuation. For simplicity, we assign both events and punctuation a monotonically increasing timestamp through the $fetch\&add$ instruction (via the *AtomicInteger* in JDK8). This brings a minor impact on the overall performance as the system's bottleneck is on concurrent state access.

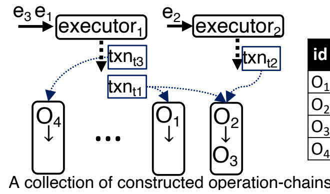
C. Dynamic Restructuring Execution

The key problem in prior solutions (Section II-C) is that all transactions are blocked while the one with the smallest timestamp is acquiring the locks it needs. Such a coarse-grained scheme is simple to realize but introduces significant lock contention overhead. We propose a fine-grained stream transaction execution mechanism, called *dynamic restructuring execution*. Specifically, *TStream* restructures a batch of state transactions (obtained from dual-model scheduling) into a collection of operation chains that can be evaluated in parallel without any lock contentions. It involves two key components: 1) transaction decomposition, which breaks down each transaction into atomic operations, and inserting these into appropriate operation chains during the `compute` mode and 2) transaction processing where the operation chains formed are evaluated in parallel during the `state access` mode.

1) Dynamic Transaction Decomposition: Once an event's EventBlotter is constructed and initialized, the executor is ready to postpone the issued transaction (i.e., Line 7 of Algorithm 1). Conceptually, it decomposes each transaction into multiple state access operations, where each operation targets one application state. Then, it dynamically inserts decomposed operations into ordered lists (called *operation chains*) with each list storing operations targeting one state (e.g., average road speed of one road segment). As the state transaction is expressed by constructing system-provided APIs, the decomposition is naturally achieved by treating one invocation of system-provided APIs (i.e., `READ`, `WRITE`, `READ_MODIFY`) as an operation. For example, two `READ` operations in Algorithm 2 will be inserted into two operation chains as they target two different states from two tables.

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 large overhead in construction and processing. We consider two properties of a suitable data structure. *First*, it must allow insertion from multiple threads simultaneously, while still guaranteeing the order of operations in the same chain. *Second*, it only requires sequential look-up rather than random access during processing. Based on these considerations, we adopt the *ConcurrentSkipList* due to its high insertion performance and small overhead compared to alternative designs, such as self-balancing trees, observed in prior work [26].

Figure 4 illustrates the decomposition process for three transactions. txn_{t1} is decomposed into two operations, O_1 and O_2. Each operation is annotated with timestamp (ts) of its original transaction, targeted state ($state$), access operation ($operation$), and parameters ($para.$) including read/write sets and dependent functions. O_2 and O_3 are inserted into the same operation chain as they target 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 different states. Note that EventBlotters (EBs) are also embedded in the operation (i.e. the last column of the table in Figure 4) so that they can be tracked during transaction processing for recording access results.



Decomposed operations

id	ts	state	operation	para.	EB
O_1	1	A	READ_MODIFY	+10	eb_1
O_2	1	B	WRITE	0	eb_1
O_3	2	B	READ	null	eb_2
O_4	3	C	READ	null	eb_3

Fig. 4: Transaction decomposition example.

2) Parallel Transaction Processing: When executors are all switched to the `state access` mode, they proceed to collaboratively process the formed operation chains. There are two cases that we need to consider.

- *Case 1:* there are no data dependencies among different operation chains. Then, one executor simply sequentially walks (i.e., evaluate) through an operation chain from the top (i.e., operation with the smallest timestamp). All operation chains can be processed in parallel by multiple executors without any contentions.
- *Case 2:* there are data dependencies among different operation chains. For example, a write operation of one state is dependent on a read operation of another state. *TStream* handles data dependencies with a simple yet effective iterative process.

Handling Data Dependency. During transaction decomposition, *TStream* records dependency information of operation chains, e.g., $chain_A$ depends on $chain_B$ if there is at least one operation targeting state A and dependent on state B. This dependency recording process is lightweight (i.e., it simply marks $chain_A$ during operation insertion) without contentions. During transaction processing, *TStream* first process those operation chains with no data dependencies. Then it processes the remaining ones which are dependent on those previously processed. This iterative process continues until all operations are processed. This approach has low overhead at tracking data dependencies (only at operation chain level), but some operations may be processed out-of-order: operations with larger timestamp without dependencies on others may be processed earlier. To handle this issue, *TStream* maintains multiple versions (i.e., updated by operations with different timestamps) of a state during the processing if there are dependencies on it. This ensures that subsequent reads will get the correct version (i.e., not necessarily the latest one) of the targeted states. After the current batch of transactions is processed, all versions of a state except the latest are expired and can be safely garbage collected, restoring all states to having only a single version.

In summary, *TStream* relies on a *mostly single-version* concurrency control (i.e., multiversion of a state is maintained if there are dependencies on it) without any centrally contented locks via two novel designs. However, *TStream* performs the best when there are no data dependencies among operation chains in the workload (e.g., *TP*) as all operation chains can be processed in parallel. In our experiments, we show that *TStream* can still perform better compared to previous solutions when the workload contains a lot of data dependencies (e.g., *SL*) owing to the unlocked parallelism opportunities.

D. System Optimizations

Transaction Batching. *TStream* focuses on achieving a reasonable latency level with high throughput. Compared to the existing approaches, *TStream* does not instantly process each issued state transaction but periodically process batches of state transactions. The interval size of two subsequent punctuations hence plays an important role in tuning system throughput and processing latency. If a large interval is configured, the system waits for a longer period before processing transactions, which increases worst-case processing latency. This is because some events are waiting (i.e., stored on its executor) for their issued transactions to be processed. Conversely, a small interval size might drop system throughput due to insufficient parallelism to amortize synchronization overhead. We will evaluate the effect of the punctuation interval in our experiments.

NUMA-Aware Processing. Following previous work [27], [28], we consider three different design options for processing operation chains on multisocket multicore architectures. *1) Shared-nothing:* we maintain a pool of operation chains per core. Essentially, decomposed operations are dynamically routed to predefined cores by hash partitioning. One executor is responsible for processing all operation chains in one core. This configuration minimizes cross-core/socket communication during execution but it may result in workload imbalance; *2) Shared-everything:* we maintain a centralized pool of operation chains, which is shared among all executors; *3) Shared-per-socket:* we maintain a pool of operation chains per socket. Executors of the same socket can thus share their workloads, but not across different sockets.

Workloads are shared among multiple executors under shared-everything and shared-per-socket configuration. Instead of statically assigning tasks to each executor, dynamic work-stealing [29] can be applied to achieve better load balancing. Specifically, multiple executors (in the same sharing group) continuously fetch and process an operation chain as a task from their shared task pool. Such a configuration achieves better workload balancing but pays more for cross-core (and cross-socket in the case of the shared-everything configuration) communication overhead compared to the shared-nothing configuration. We will evaluate *TStream* with varying NUMA-aware processing configurations in our experiments.

V. Implementation Details

TStream adopts a modular design with two modules: 1) The stream module is based on *BriskStream* [12], a highly optimized general purpose DSPS with an architecture similar to Storm. We extend *BriskStream*'s original APIs as discussed in Section IV-A; 2) The state module is based on the Cavalia [30] database, which implements system-provided APIs for managing state accesses. Our proposed techniques can be generalized to other DSPSs, such as Storm and Flink, by integrating the state module into other DSPSs with minor effort. However, our solution is mainly designed for the shared-memory multicore environment. It might require a system redesign to fully take advantage of *TStream* in a distributed environment such as Flink/Storm [14].

TStream does not rely on key-based partitioning [6] as executors are allowed to access *any* part of application states. This allows *TStream* to fuse [31] operators into a single joint operator to eliminate the impact of cross-operator communication, which is known to be a serious performance bottleneck of DSPSs [14], [32]. For example, `Road Speed`, `Vehicle Cnt`, and `Toll Notification` operators are fused into one joint operator. A switch-case statement is used to invoke the corresponding operator logic for each input event. Subsequently, *TStream* allows this joint operator to be scaled to any number of executors without violating the consistency of state. Input events can be round-robin shuffled among all executors of the joint operator to ensure load-balancing. This further simplifies application development and reduces the complexity of execution plan optimization [12].

VI. Evaluation

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

A. Benchmark Workloads

A benchmark for transactional stream processing is still an open problem. Previous work [7], [9], [10] typically chooses a couple of applications in an ad hoc manner to evaluate their system's performance. For our experiments, we follow the four criteria proposed by Jim Gray [33] and assemble four applications: *Grep and Sum (GS)*, *Streaming Ledger (SL)*, *Online Bidding (OB)*, and *Toll Processing (TP)*.

We briefly describe how our chosen applications achieve the four criteria: 1) Relevance: the applications cover diverse runtime characteristics and types of state access; 2) Portability: we describe the high-level scenario of each application and note that they can be ported easily to other DSPSs supporting concurrent state access; 3) Scalability: the applications chosen can be configured with different sizes; 4) Simplicity: the applications are chosen with simplicity in mind so that the benchmark is understandable.

Our benchmark covers different aspects of application features. *First*, our applications cover varying runtime characteristics. Specifically, when a single core is used, *TP* spends 39% of the total time in `compute` mode, and this ratio is 29% and 22% for *SL* and *OB*, respectively. *GS* spends relatively less time in `compute` mode (13%), and more time in `state access` mode. *Second*, they cover different types of state transactions. Specifically, different combinations of READ, WRITE and READ_MODIFY operations are involved in the issued state transactions from different applications. Furthermore, *SL* has heavy data dependencies when handling transfer requests, i.e., updating one user account requires a read of another user account.

We have described *TP* earlier in Figure 2 (b) in Section II. Now, we describe the remaining applications, *GS*, *SL*, and *OB* including the use case scenario, implementation, and input setups. We use a `Parser` operator to generate and parse input events and feed the remaining operators and a `Sink` operator to measure system performance in all applications. All applications need to maintain shared mutable states among operators, and concurrent state accesses (modelled as state transactions) shall follow a correct schedule. In this work, all transactions in our tested workloads will be successfully processed without abortion.

Grep and Sum (GS): GS represents a synthetic scenario where an application needs to read or update large shared mutable states and subsequently perform a computation based on the obtained state values. `Grep` issues a state transaction to access a list of records for each input event. If an event triggers a state transaction with a list of READ operations, `Grep` forwards the input event with the returned state values to `Sum`; otherwise, it updates the state with a list of WRITE operations and forwards the input event to `Sink` for recording purpose. `Sum` performs a summation of the returned state values from `Grep`. After `Sum` finishes its computation, it emits the result as one event to `Sink`. A table of 10k unique records is shared among all executors of `Grep`. Each record has a size of ∼128 bytes including JVM reference overhead, and each transaction length is 10 (i.e., ten accesses per transaction).

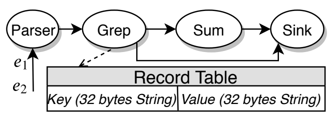


Fig. 5: Grep and Sum (*GS*).

Streaming Ledger (SL): SL is suggested by a recent commercial DSPS, Streaming Ledger [34]. It processes events that involve wiring money and asset between accounts. The detailed descriptions are omitted here for brevity and can be found in the white paper [34]. `Deposit` processes requests that top-up user accounts or assets. `Transfer` processes requests that transfer balances between user accounts and assets. The updating results (success/fail) are passed to `Sink`. The account and asset tables (each containing 10k unique records) are shared among all executors of `Deposit` and `Transfer`. Each record has a size of ∼100 bytes including JVM reference overhead. Transaction length is four for transfer request (i.e., transferring from a pair of account and asset to another pair) and is two for deposit request (i.e., update a pair of account and asset). We set a balanced ratio between transfer and deposit requests (i.e., 50% each) in the input stream.

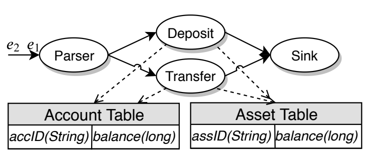


Fig. 6: Streaming Ledger (*SL*).

Online Bidding (OB): OB represents a simplified online bidding system [35]. `Auth` authenticates trade requests and dispatches valid requests for further processing. `Trade` handles three types of requests including (1) *bid request* reduces the quantity of its requested item if the bid price is larger or equal to the asking price and otherwise rejected. If the item has insufficient quantities, the bid request is also rejected. (2) *alter request* modifies the prices of a list of requested items. (3) *top request* increases the quantity of a list of items. The ratio of bid, alter, and top requests is configured as 6:1:1. A table of 10k unique bidding items are shared among all executors of `Trade`. Each record has a size of ∼50 bytes including JVM reference overhead. The transaction length of both the alter and the top request is 20, and that of the bid request is one.

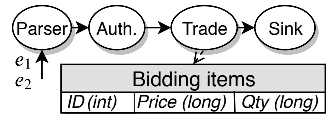


Fig. 7: Online Bidding (*OB*).

Toll Processing (TP): In this work, we focus on evaluating mechanisms to support concurrent state access during stream processing. We hence evaluate the implementation utilizing concurrent state access as illustrated previously in Figure 2(b) and omit the discussion of the conventional implementation. Each record in the road speed table has a size of ∼80 bytes, and record size in the vehicle count table varies depending on the number of items in the HashSet, i.e., $\sim 32 * (2 + |items|)$ bytes. State transactions from `Road Speed` and `Vehicle Cnt` has a length of one (i.e., update one record from one table) and those from `Toll Notification` has a size of two (i.e., read one record from the two tables).

B. Experimental Setup

We conduct all experiments on a 4-socket Intel Xeon E7-4820 server with 128 GB DRAM. The OS kernel is Linux 4.11.0-rc2. Each socket contains ten 1.9GHz cores and 25MB of L3 cache and is connected to the other three sockets via Intel QPI. NUMA characteristics, such as local and inter-socket idle latencies and peak memory bandwidths, are measured with Intel Memory Latency Checker [36]. Specifically, local memory latency (ns) is 142.6 and remote is 327.5, and local memory bandwidth (MB/sec) is 20564.8 and remote is 9944. The number of cores assigned to the system, the size of the punctuation interval and NUMA-aware processing strategies are system parameters that can be varied by users. We vary both parameters in our experiments. We use a punctuation interval of 500 and shared-nothing processing as the default execution configuration. We pin each executor on one core and assign 1 to 40 cores to evaluate the system scalability.

Application states are randomly populated and evenly distributed to each executor before execution and are kept the same among different tests. To present a more realistic scenario, we model the access distribution as Zipfian skew, where certain states are more likely to be accessed than others. For *GS*, *SL*, and *OB*, we set the skew factor to 0.6. For *TP*, we use the datasets from the previous work [14], which accesses 100 different road segments with a skew factor of 0.2. Application states may be partitioned beforehand and a multi-partition transaction will access multiple partitions. Unless explicitly mentioned, we set the length and ratio of multi-partition transactions as 4 and 25%, respectively. That is, each multi-partition transaction will access four different partitions, and 25% of all transactions are multi-partition transactions. `Toll Notification` of *TP* accesses one record from two tables, and it hence always accesses two partitions.

We implement three competing schemes including the lock-based approach (*LOCK*) [7], multiversion-lock-based approach (*MVLK*) [7], and partition-based approach (*PAT*) [10] into *TStream*. Our further investigation validates the efficiency of our reimplementation (details can be found at our technical report [20]). We also examine the system performance when locks are completely removed from the *LOCK* scheme, which is denoted by *No-Lock*, representing an upper bound on the system performance.

Evaluation Overview. We first show the overall performance comparison of different schemes on the benchmark suite (Section VI-C). Next, we provide transaction processing time breakdown for different schemes using *SL* as an example (Section VI-D). Then we evaluate *TStream* under varying workload configurations (Section VI-E). Finally, we perform a sensitivity study of *TStream* in Section VI-F.

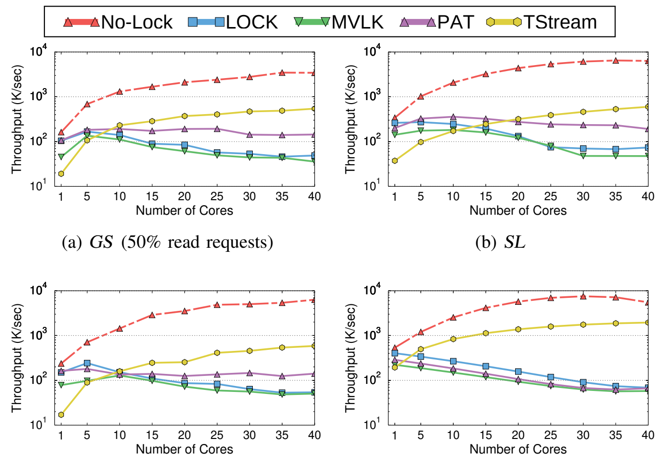
C. Overall Performance Comparison

Finding (1): *TStream* outperforms prior schemes by up to 4.8 times while ensuring a correct transaction schedule for all applications at large core counts.

The comparison results are shown in Figure 8, and there are three major observations. First, *TStream* outperforms the second-best scheme in all applications at large core counts (i.e., 3.8, 1.7, and 3.3 times over *PAT* for *GS*, *SL*, and *OB* respectively, and it outperforms 4.8 times over *LOCK* for *TP*). However, there is still a large room to improve *TStream* to achieve the performance upper bound indicated by *No-Lock*. Second, as expected, *TStream* brings lower performance improvement when the workload has heavy data dependencies (e.g., *SL*). This is because it can only evaluate a subset of operation chains (whose dependencies are resolved) in parallel during each round. Third, *PAT* generally performs better than *LOCK* and *MVLK*, as it avoids blocking when transactions access disjoint partitions. However, *PAT* performs poorly for *TP* because the workload only has 100 unique keys, and transactions are still heavily contented in the same partition. Excessive access to partition locks causes further performance degradation making it perform even worse than *LOCK*. In contrast, *TStream* is still able to exploit parallelism from a batch of transactions.

D. Transaction Processing Time Breakdown

Finding (2): The centralized lock permitting process results in serious contention. Our investigation reveals that prior schemes spend ∼80% of their execution time on synchronization.



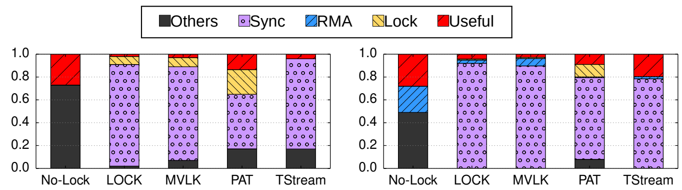
(a) *GS* (50% read requests) (b) *SL*

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

Fig. 8: Throughput (K events per second) comparison of different applications under different schemes.

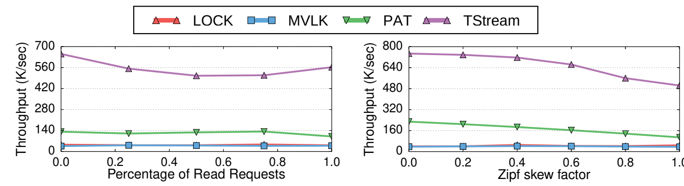
As discussed earlier in Figure 1, Section I, state access overhead quickly dominates runtime. We now use *SL* as a case to further study transaction processing time (including state access and access overhead) breakdown under different schemes. Following the previous work [19], we report how much time is spent on different components in the processing of a state transaction. 1) *Useful*: The time spent on accessing states. 2) *Sync*: The time spent on synchronization. It consists of blocking time before lock insertion is permitted in *LOCK*, *MVLK* and *PAT* or blocking time due to synchronization barriers during mode switching in *TStream*. 3) *Lock*: The total amount of time that a transaction spends inserting locks after it is permitted to do so. 4) *RMA*: The time spent on remote memory access. A thread may remotely access global counters in the case of *LOCK*, *MVLK*, and *PAT*. *TStream* may involve remote access during transaction decomposition as threads need to insert decomposed operation into appropriate operation chains. Actual state access may also cause remote memory access for all schemes run on multi-sockets. 5) *Others*: The time spent for all other operations and system overheads such as index lookup and context switching.

Figure 9 shows the time breakdown when the system is run on a single or four CPU sockets. There are two major takeaways. First, *No-Lock* spends more than 50% of the time on *Others*. Further investigation reveals that index lookup is the root cause of this performance degradation. We defer the study of more scalable index design to future work and concentrate on concurrent execution control in this work. Second, *Sync* overhead dominates all consistency preserving schemes regardless of the effect of NUMA. Although *MVLK* spends less time in *Sync* compared to *LOCK* as read may not be blocked by write, it spends more time in reading and updating the `lwm` variables (grouped under the *Others* overhead). *TStream* shows a high synchronization overhead in *SL* due to heavy data dependencies. This shows that there is a large room for improvement.



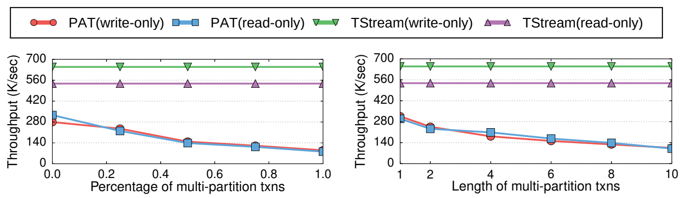
(a) Single socket (10 cores) (b) Four sockets (40 cores)

Fig. 9: Runtime breakdown per state transaction in *SL*.



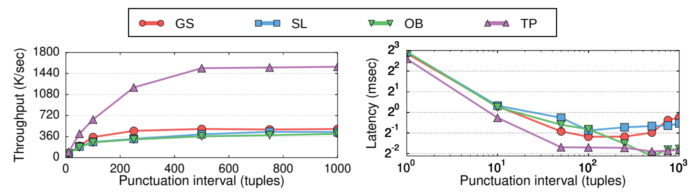
(a) Read/write workload ratio. (b) State access skewness.

Fig. 11: Varying application workload configurations of *GS*.



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

Fig. 10: Multi-partition transaction evaluation.



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

Fig. 12: Effect of varying punctuation interval.

E. Workload Sensitivity Study

Finding (3): The fine-grained design makes *TStream* robust to different workloads. Particularly, it maintains high performance under a) varying ratios and lengths of multi-partition transactions, b) varying read/write ratios, and c) highly skewed access.

We now use *GS* as an example to evaluate the different schemes under varying workload configurations.

Multi-partition Transaction Percentage. We first study the effect of state partitioning. We use a simple hashing strategy to assign states to partitions based on their primary keys so that each partition stores a similar number of states. As a common issue of all partition-based algorithms [16], the performance of *PAT* is heavily dependent on the length and ratio of multi-partition transactions. We first configure each multi-partition transaction to access six different partitions of the application states. We then vary the percentage of multi-partition transactions in the workload. The results are shown in Figure 10 (a). There are two key observations. First, since *PAT* is specially designed to take advantage of partitioning, it has low synchronization overhead when no multi-partition transactions are present (i.e., ratio=0%). However, it performs worse than *TStream* even without any multi-partition transaction as *TStream* can utilize more parallelism opportunities due to its fine-grained execution paradigm. Second, *PAT*'s performance degrades with more multi-partition transactions as it further reduces parallelism opportunities. A similar observation can be found in Figure 10 (b), where we vary the length of multi-partition transactions and fix its ratio to 50%. In the following studies, we set the multi-partition ratio to 50% under *PAT*.

Read Request Percentage. We now vary the percentage of events that trigger read requests to application states from 0% (write-only) to 100% (read-only). In this study, we remove the summation computation from *GS* and focus on evaluating the efficiency of state access. We also set the key skew factor to be 0, and hence states are accessed with uniform frequency. Figure 11 (a) shows the results and there are two major observations. First, varying read/write request ratio has a minor effect on system performance under prior schemes, *LOCK*, *MVLK* and *PAT*. This is because their execution runtime is dominated by synchronization overhead. Second, *TStream* generally performs worse with more read requests as *TStream* has to write the state value to *EventBlotter* of the triggering event (which triggers the read request) during transaction evaluation. An interesting point to take note is that *TStream*'s performance increases slightly under the read-only workload compared to the mixed workload. When there are both reads and writes to the same state, hardware prefetchers are not effective as each prefetch compete for read/write permissions from other cores resulting in permission thrashing as observed in other work [37].

State Access Skewness. In this study, we configure a write-only workload to examine how different schemes perform under contented state updates. Figure 11(b) shows that *TStream* is tolerant to access skewness. Prior schemes perform worse with increasing skewness as there is more intensive contention on the same lock. In contrast, *TStream* achieves high performance even under serious skewness because *TStream* is still able to discover parallelism opportunities among a batch of transactions (a punctuation interval of 500).

F. System Sensitivity Study

Finding (4): *TStream* can be tuned to achieve low processing latency and high throughput. We also find that the shared-nothing NUMA-aware configuration achieves the best performance.

Varying Punctuation Interval. The number of transactions to handle between two consecutive punctuations plays a critical role in *TStream*'s performance. Figure 12 (a) shows

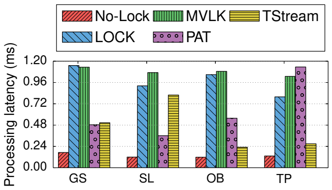


Fig. 13: 99^{th} percentile end-to-end processing latency.

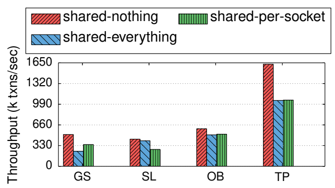


Fig. 14: Varying NUMA-aware configurations.

that the performance of *TStream* generally increases with a larger punctuation interval. It also shows that a large punctuation interval is especially beneficial for *TP* because the workload has only 100 unique segment IDs and transactions are heavily contented at the same state. By allowing more transactions to be accumulated, *TStream* increases parallelism opportunities among more decomposed operations, and its performance hence increases significantly. Figure 12 (b) shows the processing latency of *TStream* with various punctuation intervals. Following the previous work [38], we define the end-to-end processing latency as the duration between the time when an input event enters the system and the time when the result is generated. Thanks to the significantly improved performance, *TStream* achieves very low processing latency. When the punctuation interval is set to 500, its 99^{th}-percentile processing latency is around 0.23$\sim$0.63 ms, which satisfies many existing use cases [38]. It also shows that there is no clear trade-off between throughput and latency under varying punctuation intervals. This is because higher throughput also reduces queuing delays. Latency increases with increasing punctuation interval only when throughput can not be further improved (e.g., *GS* at an interval of 250). Figure 13 further shows that *TStream* (punctuation interval=500) achieves comparable and sometimes even lower processing latency compared to the state-of-the-art. The optimal (e.g., maximum throughput) punctuation interval may be affected by many factors including machine characteristics (e.g., number of cores, size of LLC, and memory), number of unique keys in the workload, state size, tuple size, length of the state transaction, etc. We now obtain the punctuation interval from extensive experiments. Due to its considerable complexity, we leave the estimation of the optimal punctuation interval itself to future work.

Effect of NUMA-aware Optimizations. We now compare different NUMA-aware processing configurations of *TStream* including shared-nothing, shared-everything, and shared-per-socket. Work-stealing can be further enabled in the latter two configurations and our experimental results show that work-stealing significantly improves their throughput by 1.6$\sim$7.0 times. However, Figure 14 shows that *TStream* achieves the best performance for all applications under the shared-nothing configuration. This indicates that cross-core and cross-socket communication during state transaction processing should always be avoided. Nevertheless, we plan to investigate this impact on other applications that may be more sensitive to workload imbalance rather than communication overhead.

VII. Related Work

Concurrent Stateful Stream Processing. We have reviewed some of the related work in Section II and now discuss a few more. Botan et al. [9] presented an *unified transactional model* for streaming applications. Affetti et al. [8] recently proposed a state consistency model for stream processing. Both studies provide the same formal definitions on how mutable application states can be shared among executors during stream processing through transactional semantics, and we have adopted their consistency model. However, their implementations heavily rely on locks to guarantee state consistency. Unless these systems, *TStream*'s novel design has been shown to achieve much higher throughput and scalability with various workloads. There is also a recent commercial system, called Streaming Ledger [34] for extending Flink to support concurrent state access with a goal similar to ours. It is close-sourced, and we can not compare our system with that.

Database Partitioning. Prior work [39], [40] propose to divide the underlying storage into multiple logical partitions, each of which is assigned a single-thread execution engine with exclusive access. Transaction workloads in those databases are partitioned according to the primary key(s) in the root table [41], and the performance can significantly degrade as the ratio of multi-partition transactions increases [27]. S-Store [10] adopts the same technique with extensions to further guarantee state access ordering [42]. The partition-based approach's common drawback is their handling of multi-partition transactions. In contrast, *TStream* decomposes a collection of transactions at runtime and execute the resulting operation chains at high system concurrency.

Program Partitioning. Many have proposed adopting program partitioning and transformation to optimize the performance of transaction processing, such as [43], [44]. TStream deviates from existing techniques such as transaction chopping [45] which are purely static. *TStream* dynamically restructures potentially conflicting operations in a collection of state transactions into independent groups called operation chains which are evaluated in a determined sequence. Transaction-chopping and its many variants such as [46] were proposed in the context of nondeterministic transaction processing, and thus their program partitioning technique does not account for the state access sequence that is necessary for state consistency of stream processing. Although there are some similarities between the periodic transaction processing of *TStream* and lazy transaction evaluation [47], the differences between them are considerably pronounced. For example, *TStream* needs to ensure that transactions are processed following event timestamp sequence, which results in different design challenges and optimization opportunities (e.g., sorted operation chains).

Multicore Architectures. To meet the fast growing performance demand, optimizing stream processing to better utilize hardware resource has been a hot research topic [12],

[14], [32], [48]. We refer more details in our recent survey [49]. *TStream* is built to improve multicore utilization standing on the shoulders of many valuable existing works such as [12], [23], [27]. However, none of the previous work addresses the scalability bottlenecks that *TStream* solves, i.e. how to scale concurrent state access in stream processing with consistency guarantee.

VIII. Conclusion

With the increasing adoption of stream processing in emerging use cases, we believe that an efficient concurrent stateful DSPS becomes more and more desirable. *TStream* demonstrates that efficient concurrent state access during stream processing can be elegantly supported with its novel dual-mode scheduling and dynamic restructuring execution mechanism on modern multicore architectures. In particular, it guarantees state consistency, while judiciously exploits more parallelism opportunities – both within the processing of each input event and among a (tunable) batch of input events. As for future work, an immediate next step is to study how to efficiently handle transaction aborting in *TStream*, e.g., when a state update violates integrity property.

Acknowledgment

We thank Dr. Wentian Guo and Prof. Dinh Tien Tuan Anh for their helps on managing the server. We thank Dr. Benjamin Hong Meng Tan and the anonymous ICDE reviewers for their valuable comments. This work is supported by a MoE AcRF Tier 1 grant (T1 251RES1824) and Tier 2 grant (MOE2017-T2-1-122) in Singapore. Feng Zhang's work is partly supported by the National Natural Science Foundation of China (Grant No. 61732014, 61802412), and Beijing Natural Science Foundation (Grant No. L192027).

References

[1] (2018) Apache flink, https://flink.apache.org/.
[2] (2018) Apache storm, http://storm.apache.org/.
[3] S. Kulkarni and et al., "Twitter heron: Stream processing at scale," in *SIGMOD '15*.
[4] Y. Wu and K. Tan, "Chronostream: Elastic stateful stream computation in the cloud," in *ICDE'15*.
[5] P. Carbone and et al., "State management in apache flink: Consistent stateful distributed stream processing," *Proc. VLDB Endow. 2017*.
[6] N. R. Katsipoulakis and et al., "A holistic view of stream partitioning costs," *Proc. VLDB Endow. 2017*.
[7] D. Wang and et al., "Active complex event processing over event streams," *Proc. VLDB Endow. 2011*.
[8] L. Affetti and et al., "Flowdb: Integrating stream processing and consistent state management," in *DEBS '17*.
[9] I. Botan and et al., "Transactional stream processing," in *EDBT '12*.
[10] J. Meehan and et al., "S-store: Streaming meets transaction processing," *Proc. VLDB Endow. 2015*.
[11] A. Arasu and et al., "Linear road: A stream data management benchmark," in *VLDB '04*.
[12] S. Zhang and et al., "Briskstream: Scaling data stream processing on shared-memory multicore architectures," in *SIGMOD '19*.
[13] D. J. Abadi and et al., "Aurora: A new model and architecture for data stream management," *The VLDB Journal, 2003*.
[14] S. Zhang and et al., "Revisiting the design of data stream processing systems on multi-core processors," in *ICDE'17*.
[15] M. J. Sax and M. Castellanos, "Building a transparent batching layer for storm." HP Labs Technical Report, 2013.
[16] A. Pavlo and et al., "Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems," in *SIGMOD '12*.
[17] P. A. Bernstein and E. Newcomer, *Principles of transaction processing*. Morgan Kaufmann, 2009.
[18] X. Yu and et al., "Tictoc: Time traveling optimistic concurrency control," in *SIGMOD '16*.
[19] X. Yu and et al., "Staring into the abyss: An evaluation of concurrency control with one thousand cores," *Proc. VLDB Endow. 2014*.
[20] S. Zhang, Y. Wu, F. Zhang, and B. He, "Towards concurrent stateful stream processing on multicore processors (technical report)," *arXiv preprint arXiv:1904.03800*, 2019.
[21] B. He and et al., "Mars: a mapreduce framework on graphics processors," in *PACT'08*.
[22] P. A. Tucker and et al., "Exploiting punctuation semantics in continuous data streams," *TKDE'03*.
[23] H. Miao and et al., "Streambox: Modern stream processing on a multicore machine," in *USENIX ATC'17*.
[24] B. Chandramouli and et al., "Trill: A high-performance incremental query processor for diverse analytics," *Proc. VLDB Endow. 2014*.
[25] Cyclicbarrier. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CyclicBarrier.html.
[26] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," in *Workshop on Algorithms and Data Structures, 1989*.
[27] D. Porobic and et al., "Oltp on hardware islands," *Proc. of the VLDB Endow. 2012*.
[28] D. Porobic and et al., "Atrapos: Adaptive transaction processing on hardware islands," in *ICDE'14*.
[29] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM, 1999*.
[30] Y. Wu and et al., "Transaction healing: Scaling optimistic concurrency control on multicores," in *SIGMOD '16*.
[31] M. Hirzel and et al., "A catalog of stream processing optimizations," *ACM Comput. Surv. 2014*.
[32] S. Zeuch and et al., "Analyzing efficient stream processing on modern hardware," *Proc. VLDB Endow. 2019*.
[33] J. Gray, *Benchmark Handbook: For Database and Transaction Processing Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.
[34] "Data Artisans Streaming Ledger Serializable ACID Transactions on Streaming Data, https://www.da-platform.com/streaming-ledger," 2018.
[35] J. Tan and M. Zhong, "An online bidding system (obs) under price match mechanism for commercial procurement," *Applied Mechanics and Materials, 2014*.
[36] (2018) Intel memory latency checker, https://software.intel.com/articles/intelr-memory-latency-checker.
[37] N. D. E. Jerger and et al., "Friendly fire: understanding the effects of multiprocessor prefetches," in *ISPASS'06*.
[38] T. Das and et al., "Adaptive stream processing using dynamic batch sizing," in *SOCC '14*.
[39] A. Thomson and D. J. Abadi, "The case for determinism in database systems," *Proc. VLDB Endow. 2010*.
[40] A. Kemper and T. Neumann, "Hyper: A hybrid oltp olap main memory database system based on virtual memory snapshots," in *ICDE'11*.
[41] M. Stonebraker and et al., "The end of an architectural era: (it's time for a complete rewrite)," in *VLDB '07*.
[42] U. Cetintemel and et al., "S-store: A streaming newsql system for big velocity applications," *Proc. VLDB Endow. 2014*.
[43] A. J. Bernstein and et al., "Concurrency control for step-decomposed transactions," *Inf. Syst. 1999*.
[44] Y. Wu and et al., "Fast failure recovery for main-memory dbmss on multicores," in *SIGMOD '17*.
[45] D. Shasha and et al., "Transaction chopping: Algorithms and performance studies," *ACM Trans. Database Syst. 1995*.
[46] N. Narula and et al., "Phase reconciliation for contended in-memory transactions," in *OSDI'14*.
[47] J. M. Faleiro and et al., "Lazy evaluation of transactions in database systems," in *SIGMOD '14*.
[48] A. Koliousis and et al., "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *SIGMOD '16*.
[49] S. Zhang and et al., "Hardware-conscious stream processing: A survey," *SIGMOD Rec.*, 2019.