MorphStream: Adaptive Scheduling for Scalable Transactional Stream Processing on Multicores

Yancan Mao*
National University of Singapore

Jianjun Zhao†
Huazhong University of Science and Technology

Shuhao Zhang‡
Singapore University of Technology and Design

Haikun Liu§
Huazhong University of Science and Technology

Volker Markl
Technische Universität Berlin

ABSTRACT

Transactional stream processing engines (TSPEs) differ significantly in their designs, but all rely on non-adaptive scheduling strategies for processing concurrent state transactions. Subsequently, none exploit multicore parallelism to its full potential due to complex workload dependencies. This paper introduces *MorphStream*, which adopts a novel approach by decomposing scheduling strategies into three dimensions, and then strives to make the right decision along each dimension, based on analyzing the decision trade-offs under varying workload characteristics. Compared to the state-of-the-art, *MorphStream* achieves up to 3.4 times higher throughput and 69.1% lower processing latency for handling real-world use cases with complex and dynamically changing workload dependencies.

ACM Reference Format:
Yancan Mao, Jianjun Zhao, Shuhao Zhang, Haikun Liu, and Volker Markl. 2022. MorphStream: Adaptive Scheduling for Scalable Transactional Stream Processing on Multicores. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Many emerging stream applications [2, 9, 25, 34] rely on the support of *shared mutable states*, where application states may be concurrently read and modified by multiple threads during stream processing. Those applications are challenging to be supported correctly [11] and/or efficiently [40] in today's mainstream stream processing engines (SPEs), such as Storm [31], Flink [10], and Spark-Streaming [39]. In response, transactional SPEs (TSPEs) have been recently proposed with built-in support of shared mutable states and have received much attention from academia [3, 22, 40] and industry [1]. Different from conventional SPEs, TSPEs adopt transactional semantics during the processing of continuous data streams, where accesses to shared mutable states are modelled as *state transactions*. Subsequently, the concurrent process of state transactions must be scheduled to ensure some form of transactional semantics in TSPEs.

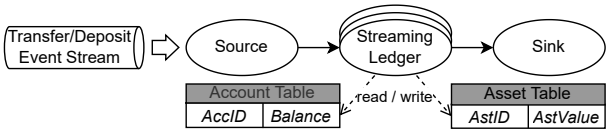


Figure 1: Streaming Ledger (SL) – An example application scenario of TSPEs [1].

Motivating Example. Figure 1 illustrates a running example of a simplified use case that benefits from TSPEs, namely Streaming Ledger (SL), which was suggested by a recently announced commercialized version of Apache Flink [1]. Unlike batch-based ledger systems, it processes a stream of requests involving wiring money and assets among users and outputs the processing results as an output stream to users. Two types of state transactions accessing two tables of shared mutable states are generated during the processing of each input request: (1) a *Transfer* transaction processes a request that transfers balances between user accounts and assets; and (2) a *Deposit* transaction processes requests that top-up user accounts or assets. The processing of all concurrent state transactions needs to ensure transactional semantics. For instance, a state transaction may be aborted due to the violation of a consistency property, such as that the account balance can not become negative. We will demonstrate that Flink, a popular SPE, achieves orders of magnitude lower throughput in handling this use case, compared to any of the existing TSPEs.

The Problem: Non-Adaptive Scheduling Strategies. Existing TSPEs [3, 22, 40] differ significantly in their concurrent execution approaches but all rely on some non-adaptive scheduling strategies. S-Store [22] is a recent TSPE built upon extending H-Store [29] with streaming capabilities. It adopts a coarse-grained processing paradigm (*state partitioning*) by scheduling transactions into pre-split state partitions. It has low context switching overheads and can handle the transaction abort as early as it happens. However,

*Work is done while employed as a visiting student at SUTD.
†Co-first author.
‡Corresponding author, contact at shuhao_zhang@sutd.edu.sg.
§Jianjun Zhao and Haikun Liu are further affiliated with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

as parallelism increases, the intensification of access conflicts among state partitions severely limits the system performance [40]. TStream [40] outperforms S-Store on multicore architectures with a *transaction restructuring* paradigm. In TStream, state access conflicts are avoided as much as possible by decomposing transactions into atomic operations, which can run in parallel. However, it adversely affects the efficient handling of transaction abort and complex data dependencies in the workload. As a result, they often involve large synchronization and unnecessary state transaction aborting overhead. None of the existing TSPEs can maximize performance under different and dynamically changing workload characteristics, such as the varying ratio of aborting transactions and changing state access distribution.

The Solution: *MorphStream*. In this paper, we propose a novel adaptive scheduling strategy for TSPEs by mapping the state transaction scheduling problem to a graph scheduling problem. This allows us (1) to find the best trade-off for selecting a suitable scheduling strategy for a given workload and (2) to employ multiple scheduling strategies concurrently, where each is applied to a subset of workloads. We introduce *MorphStream* that **morph**s multiple scheduling strategies in transactional **stream** processing.

Different from existing TSPEs, *MorphStream* identifies the fine-grained *temporal*, *logical*, and *parametric* dependencies among state access operations of a batch of state transactions. Then, it maps the workloads with dependencies into a *task precedence graph* (TPG), where vertexes map to state access operations, and edges map to fine-grained dependencies among operations. The key point is that, based on the TPG, we can decompose the scheduling strategy into three dimensions of scheduling decisions: 1) *structured* or *non-structured* exploration strategies, 2) *single operation* or *group of operations* as the unit of scheduling, and 3) *lazy* or *eager* abort handling mechanisms. Subsequently, *MorphStream* can adaptively switch to a different scheduling strategy by making suitable scheduling decisions in each dimension. Based on analyzing the decision trade-offs under varying workload characteristics, we propose a heuristic-based decision model that leverages the properties of TPG to guide *MorphStream* to make the correct scheduling decision at runtime.

Compared to existing solutions such as TStream [40] and S-Store [22], *MorphStream* involves a much more complex dependency identification and resolving mechanism. To address this issue, we further design a two-step TPG construction mechanism to create a TPG for every batch of state transactions in parallel with low overheads. To efficiently ensure the correctness during the exploration of the TPG, we propose a stateful TPG implementation, where each vertex in the TPG is further annotated with a finite state machine. *MorphStream* tracks the state transition (e.g., ready to process, executed, or aborted) to guarantee a correct schedule while making dynamic scheduling decisions.

In summary, this work makes the following technical contributions:

- In Section 3, we formally define the *transaction scheduling* problem in TSPEs. We then map the transaction scheduling problem to a graph scheduling problem.
- In Section 4, we discuss how to decompose the scheduling decision into three dimensions, including *exploration strategies*, *scheduling unit granularities*, and *abort handling mechanisms*. We also propose a heuristic-based decision model to make a suitable scheduling decision under varying workload characteristics at runtime.
- To reduce TPG construction overhead and the adaptive scheduling overhead, we further propose parallel TPG construction mechanisms and an efficient stateful TPG implementation (Section 5).
- We experimentally demonstrate that (1) *MorphStream* brings up to 3.4 times higher throughput and 69.1% lower latency for handling real-world use cases, and (2) *MorphStream* is able to select a better-performing scheduling strategy under changing workload characteristics (Section 6).

Table 1: Summary of terminologies.

Type	Notation	Definition
System Specific	txn_{ts}	State Transaction
	O_i	State access operations
	TD	Temporal Dependency
	LD	Logical Dependency
	PD	Parametric Dependency
	TPG	Task Precedence Graph (V, E)
Scheduler Specific	s-explore	Structured Exploration
	ns-explore	Non-Structured Exploration
	f-schedule	Fine-grained Scheduling Unit
	c-schedule	Coarse-grained Scheduling Unit
	e-abort	Eager Aborting
	l-abort	Lazy Aborting
Workload Characteristics	T	Number of Transactions
	l	Transaction Length
	a	Ratio of Aborting Transactions
	θ	State Access Distribution
	r	Number of State Access Per Operation
	C	Computational Complexity of an Operation

2 PRELIMINARIES

In this section, we first discuss some preliminaries of transactional stream processing (TSP). Then, we summarize three types of workload dependencies in TSP applications.

2.1 Transactional Stream Processing

Different from traditional stream processing engines (SPEs), such as Storm [31] and Flink [10], *transactional SPEs (TSPEs)* [1, 4, 5, 9, 12, 16, 21] allow the system to maintain *shared mutable states*, to which multiple execution entities (i.e., threads) can reference and update. Shared mutable states are preallocated in memory. When there is a new state, the preallocated spaces are expanded accordingly before processing. We generally follow the definitions in prior work [22, 40] and briefly recall them for completeness as follows.

Definition 1 (State Access Operation). *A state access is a read or write operation to shared mutable states, denoted as* $O_i = Read_{ts}(k)$ *or* $Write_{ts}(k, v)$. *Timestamp* ts *is defined as the time of its triggering input event, while* k *denotes the state to read or write, and* v *denotes the value to write.*

Note that, *insert* and *delete* are treated as write operations. Essentially, an insert updates the value of the corresponding state, while a delete marks the corresponding state to be invalid. The concurrent accesses (i.e., read and write) to the shared mutable states must satisfy predefined constraints to ensure some form of transactional semantics.

Definition 2 (State Transaction). *The set of state access operations triggered by the processing of one input tuple is defined as one* state transaction, *denoted as* $txn_{ts} = <O_1, O_2, ... O_n>$. *Operations of the same transaction have the same timestamp. For brevity, we use the timestamp ts to differentiate different state transactions.*

In analogy to conventional transaction processing, TSPEs guarantee ACID properties [3]. In addition, TSPEs need to ensure further that dependent state accesses strictly follow their timestamp sequence [3, 11, 22, 40]. Specifically, a correct state transaction schedule can be defined as follows.

Definition 3 (Correct Schedule). *A schedule (S) of state transactions* txn_{t1}, txn_{t2}, ..., txn_{tn} *is correct if it is* ***conflict equivalent*** *to* $txn_{t1} \prec txn_{t2} \prec ... \prec txn_{tn}$, *where* $\prec$ *means that the left operand precedes the right operand.*

2.2 Workload Dependencies

A key objective to scale transactional stream processing is to maximize system concurrency while maintaining a correct schedule. However, it is a nontrivial challenge due to complex inter- and intra-dependencies among state transactions. By carefully analyzing many existing TSPE applications [1, 6, 9, 22, 30, 34, 40], we have summarized workload dependencies into three types, including *temporal*, *logical*, and *parametric*. In the following, we use the Streaming Ledger (SL) as a running example involving three state transactions (txn_1, txn_2, and txn_3) shown in Figure 2 for illustration.

***Temporal Dependency* (TD).** State accesses must strictly follow their event sequence, leading to *temporal dependencies* among operations.

Definition 4. *We define that* O_i *temporally depends on* O_j *if they access the same state, and* O_i *has a larger timestamp.*

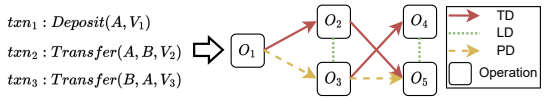
For example, in Figure 2, O_2 temporally depends on O_1 as they access the same record with different timestamps. Similarly, O_4 and O_5 temporally depend on O_3 and O_2, respectively. Note that there is no temporal dependency among operations of the same transaction, as they are triggered by the same input event with the same timestamp.

***Logical Dependency* (LD).** To guarantee ACID, aborting one operation shall lead to aborting all operations of the same transaction. This can be guaranteed during transaction commit by tracking a *logical dependency* among operations of the same transaction (Section 4.3).

Definition 5. *We define that two operations* O_i *and* O_j *logically depend on each other if they belong to the same state transaction.*

For example, in txn_2 and txn_3, the transfer must either change both accounts or neither. Hence, O_3 and O_5 logically depend on O_2 and O_4, and vice versa. If there is sufficient balance in the source account, O_2 (O_4) will subtract the transferred amount from it, and O_3 (O_5) will add the same value to the target account. Otherwise, the entire transaction shall be aborted.

***Parametric Dependency* (PD).** For a write operation $Write(k, v)$, the key k is extracted from the input event [22, 40],



TS	State Access Operation	Function (f)
1	$O_1 = Write_1(A, f_1(V_1))$	$f_1 : Read_1(A) + V_1$
2	$O_2 = Write_2(A, f_2(A, V_2))$	$f_2 : Read_2(A) - V_2$ if $Read_2(A) > V_2$
2	$O_3 = Write_2(B, f_3(B, A, V_2))$	$f_3 : Read_2(B) + V_2$ if $Read_2(A) > V_2$
3	$O_4 = Write_3(B, f_4(B, V_3))$	$f_4 : Read_3(B) - V_3$ if $Read_3(B) > V_3$
3	$O_5 = Write_3(A, f_5(A, B, V_3))$	$f_5 : Read_3(A) + V_3$ if $Read_3(B) > V_3$

Figure 2: Dependencies in Streaming Ledger (SL)

while the value v may depend on the value of a list of states, i.e., $v = f(k_1, k_2, ..., k_m)$, where f is a user-defined function. It implies a *parametric dependency* between two write operations when v of one operation depends on the execution of another operation.

Definition 6. *We define that* $O_i = Write(k_i, v)$, *where* $v = f(k_1, k_2, ..., k_m)$, *parametrically depends on* $O_j = Write(k_j, v')$ *if* $k_j \neq k_i$, $k_j \in k_1, k_2, ..., k_m$, *and* O_i *has a larger timestamp.*

Taking O_3 in txn_2 as an example, it should be noted that whether the write is allowed depends on a user-defined function related to A and V, such as the transferring amount (V) should be less than the balance of the source account (A). Hence, O_3 depends on the state of A, as indicated by $f_3(B, A, V_2)$. Therefore, O_3 parametrically depends on O_1. Similarly, O_5 parametrically depends on O_3.

3 THE MORPHSTREAM APPROACH

In this section, we discuss how we map the state transaction scheduling problem to a graph scheduling problem, and outline the solution overview of *MorphStream*.

3.1 Problem Mapping

Lemma 1. *Processing all operations of a list of state transactions L following the three types of dependencies defined in Definition 4, 5, and 6 guarantees a correct schedule.*

Proof Sketch: According to the conflict serializability theorem [35], *MorphStream* is conflict-serializable if and only if its conflict graph of schedules, where transactions are nodes and conflict relations (i.e., read/write conflicts or data dependencies among transactions) are edges, is always acyclic. This is always true when state transactions are scheduled following three types of dependencies in *MorphStream*. First, the read/write conflicts are captured by TDs and PDs. TDs/PDs are ordered by timestamp sequences, i.e., for any two operation O_i and O_j, if O_j has PDs/TDs on O_i, then the timestamp of O_j is greater than O_i. Thus, a cycle between any two operations will violate the time-ordered property of TDs/PDs, and can not exist in *MorphStream*. Second, LDs are among operations of the same transaction, and will not lead to conflict relations among transactions. *MorphStream* is hence conflict-serializable. Further committing (or aborting) according to LDs guarantees ACID.

Task Dependency Graph (TPG). Motivated by Lemma 1, we map a batch of state transactions to a *task dependency graph (TPG)*. For example, in Figure 2, $O_1 \sim O_5$ and the dependencies among them naturally form a TPG.

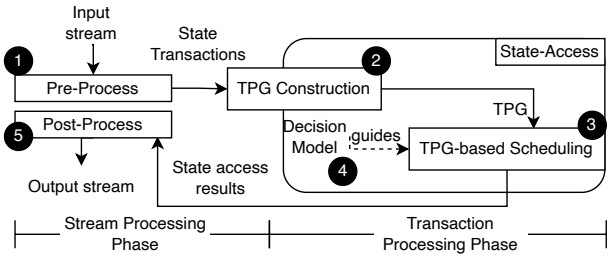


Figure 3: The execution workflow of *MorphStream*.

Definition 7. *Given a list of state transactions* $L = <txn_{t1}, txn_{t2}, ..., txn_{tn}>$*, MorphStream constructs a task precedence graph* $TPG = (V, E)$ *such that there is a one-to-one mapping of one vertex* v_i *in* G *to one operation* $O_j \in txn_{ti}$*, where* $txn_{ti} \in L$*. The three types of dependencies among operations are one-to-one mapped to edges among vertexes, denoted as* $e(v_i, v_j, D)$*, where* $D \in \{\mathsf{TD}, \mathsf{LD}, \mathsf{PD}\}$.

Transaction Scheduling based on the TPG. In analogy to the task graph scheduling problem [24], a vertex v_i may not be able to be scheduled due to unsolved dependencies (i.e., incoming e of v_i). Given Lemma 1, scheduling tasks (i.e., v_i) for concurrent execution according to dependencies (i.e., e_i) ensures that the resulting transaction schedule is correct. Different schedules may lead to different completion times due to 1) varying execution concurrency, 2) varying context switching overhead, and 3) varying wasted computational overheads due to aborts. Hence, the goal of *MorphStream* is to identify a correct schedule that minimizes overall completion time, which can be defined as follows.

Definition 8. *Given a TPG* $G = (V, E)$ *constructed by a list of state transactions* L*, the* ***optimal correct schedule*** *is the schedule of operations (i.e.,* v*) such that the overall completion time is minimized, while the dependency constraints are enforced.*

The task graph scheduling problem, in its original form, is NP-complete [24]. There are many existing heuristic solutions in the literature [14, 18, 19]. However, our transactional context makes the scheduling problem quite different from the well-studied task graph scheduling problem. For example, the generic task scheduling problem [24] assumes a known completion time of each task a priori. In contrast, we can not make such an assumption, as each state access may take a variable processing time due to the PDs with arbitrary user-defined functions. Furthermore, due to the possibility of transaction aborting, the actual execution flow may vary arbitrarily, making the scheduling a difficult (if not impossible) task to optimize with existing scheduling algorithms.

3.2 Solution Overview

Based on our problem mapping, we propose *MorphStream*, a novel TSPE that is able to scale even under highly contended and dynamic workloads. More implementation details are discussed in Section 5.

Programming Model. We follow the common programming model of TSPEs abstracted in the following three steps [40]. These three steps are recursively conducted for every batch of input events. (i) *preprocess* the input events, which will determine the read/write sets of the state transactions. For example, in Figure 1, the system uses *accID* to determine the *account record* in the *Account Table*. Note that, *MorphStream* does not support non-deterministic state transactions, such as those with random keys. We leave the exploration of such features in future. (ii) *state access*, where all state accesses are actually performed. (iii) *postprocess*, where the input event will be further processed according to the access result, and the corresponding output will be generated. As a stream processing system must keep the data moving [28], the output of the aborted transaction is marked with "failed state access" to notify users, who can resubmit requests which become part of new input events.

Execution Workflow. Based on the programming model, the execution workflow of *MorphStream* is depicted in Figure 3. Similar to some prior works [40], *MorphStream* separates transactional stream processing into two phases: stream processing and transaction processing. It periodically switches between the two phases, separated by *punctuations* [32], guaranteeing that no subsequent input event will have a smaller timestamp. During the stream processing phase, ❶ a batch of input events is *preprocessed*, and the generated state transactions are batched to be processed. During the transaction processing phase, the batched *state access* is performed. It relies on punctuation [33] to ensure that input events in subsequent batches have increasing timestamps, while input events within a batch may arrive out-of-order. ❷ *MorphStream* identifies the fine-grained *temporal*, *logical*, and *parametric* dependencies within and among a batch of state transactions to construct the corresponding TPG in parallel. Note that we partition the TPG construction process (Section 5) into two steps during the stream processing phase and the transaction processing phase. Subsequently, ❸ threads schedule operations for concurrent execution according to the constructed TPG (Section 4). ❹ A decision model (Section 4.4) guides *MorphStream* to make the most appropriate scheduling decisions at runtime, considering varying workload characteristics. Upon finishing of transaction execution, final processing results as output stream are generated during ❺ the *postprocess* of input events based on the obtained shared mutable state access results.

4 EXPLORING SCHEDULING DECISIONS

In the following, we discuss how we decompose the scheduling based on the TPG into three dimensions of scheduling decisions, namely exploration strategies (Section 4.1), scheduling unit granularities (Section 4.2), and abort handling mechanisms (Section 4.3). Table 2 summarizes scheduling decisions at each dimension. We also propose a heuristics decision model to make a suitable scheduling decision based on the current workload characteristics at runtime (Section 4.4).

4.1 Exploration Strategies

Threads explore the TPG to pick up operations to process. There are generally two types of exploration strategies, including (1) a structured approach following a depth-first or breadth-first traversal strategy and (2) an unstructured approach based on random traversal with countdown latches. Those exploration strategies involve varying synchronization overheads for workloads with different properties.

4.1.1 Structured Exploration (s-explore). Following a prior work [23], to guide structurally exploring the TPG, we first

Table 2: Scheduling decisions at three dimensions

Dimension	Decision	Pros	Cons
Exploration Strat.	s-explore	Threads can run in parallel with minimum coordination	BFS: Sensitive to workload imbalance / DFS: High memory access overhead
	ns-explore	More parallelism opportunities	Higher message-passing overhead
Scheduling Gran.	f-schedule	Better system scalability	High context switching overhead
	c-schedule	Lower context switching overhead	Less scalable and more sensitive to load imbalance
Abort Handling	e-abort	Less wasted computing efforts	High context switching overhead
	l-abort	Less context switching overhead	More wasted computing efforts

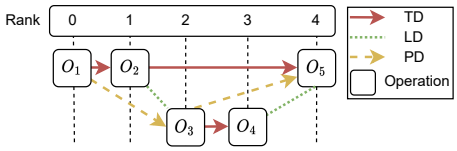


Figure 4: A stratified auxiliary structure of TPG.

generate an auxiliary structure of the TPG as illustrated in Figure 4. First, vertexes are partitioned into subsets such that vertexes are connected by a directed path belonging to different subsets. Second, subsets are assigned rank such that for each edge, the rank of the subset that contains the target of the edge is less than the rank of the subset that contains its source. Subsets with the same rank are said to be in the same stratum, and the most outer stratum is rank 0. Based on the stratified auxiliary structure, two structured exploration methods can be applied to explore the TPG in analogy to the breadth- and depth-first search. In both ways, threads are running in parallel with minimum coordination at runtime.

A) BFS-like Exploration: Like a BFS, a thread begins to explore operations at the most outer stratum of TPG. All threads cooperatively explore operations of the same stratum in parallel. Only when operations at one stratum are all processed can threads proceed to the next stratum for further exploration. A barrier-based synchronization is thus required among threads. The effectiveness of such a BFS-like exploration depends on the workload balance among threads, which is challenging to ensure due to the random completion time of each operation as well as the potential aborts.

B) DFS-like Exploration: For the DFS-like exploration, we first assign an equal number of operations in every stratum to threads. During exploration, a thread may immediately proceed to explore the next stratum once the dependencies of its assigned operations in the next stratum are resolved. In this way, threads cooperate without relying on barriers at each stratum but only on the dependencies of assigned operations. This reduces synchronization overhead, as a thread waits for fewer threads before making progress, but it incurs a higher memory access overhead for checking whether dependencies have been resolved.

4.1.2 Non-Structured Exploration (ns-explore). Instead of structurally exploring the TPG, threads can randomly pick up operations, i.e., those with dependencies resolved. In this strategy, each thread maintains a signal holder, which asynchronously receives and handles countdown latch signals from other threads to manage the resolution of dependencies of the remaining operations.

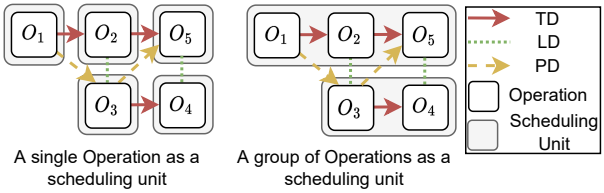


Figure 5: Different scheduling unit granularities.

Specifically, when an operation O_i has been successfully processed, the processing thread immediately sends a countdown signal to all other threads, which can now explore the dependent operations of O_i. Such an exploration strategy resolves the dependencies of operations immediately, leading to more operations being available to schedule. However, as threads have to send countdown latch signals via all directed edges to resolve dependencies, ns-explore hence incurs higher message-passing overhead than s-explore.

4.2 Scheduling Unit Granularities

During exploration, each thread can pick up a single operation or a group of operations as the unit of scheduling, resulting in different performance behaviours.

4.2.1 Fine-grained Scheduling Unit (f-schedule). A thread can schedule a single operation as the scheduling unit, as shown in Figure 5(a). Such a fine-grained scheduling unit leads to good system scalability. First, a small scheduling unit essentially translates to higher parallelism opportunities, as more operations can be concurrently executed. Second, dependency resolution can be also conducted in a fine-grained manner, as whenever an operation has been successfully processed, we can immediately check if its dependent operations have become available for scheduling.

4.2.2 Coarse-grained Scheduling Unit (c-schedule). The drawback of f-schedule is that it incurs significant context-switching overhead. Alternatively, we may schedule a group of operations together. For example, <O_1, O_2, O_5> and <O_3, O_4> form two scheduling units in Figure 5(b). Operations in the same group are sequentially processed following TDs or LDs, when they are grouped by key or timestamp. When all operations in that group have been processed successfully, their subsequent dependent operations become available for scheduling. In this way, the context-switching overhead is amortized over multiple operations. However, this slows down dependency resolution among operations within the same group, making it less scalable and more sensitive to load imbalance among groups.

Circular Dependency Issue. It is noteworthy that the example shown in Figure 5 contains a circular dependency among the two coarse-grained scheduling units (i.e., $O_1 \rightarrow O_3$ and $O_3 \rightarrow O_5$). To address such an issue, *MorphStream* merges circular dependencies into a single scheduling unit. Subsequently, operations with no dependencies (i.e., O_1) are first processed, and the remaining ones are processed when their dependencies have been resolved. This iterative process continues until all operations in the merged scheduling unit have been processed. Such a circular dependency leads to reduced parallelism and will not appear under f-schedule. Hence whether there are cyclic dependencies becomes a key

workload characteristic to be considered when selecting different scheduling unit granularities in *MorphStream*.

4.3 Abort Handling Mechanisms

During execution, each thread may abort transactions *eagerly* or *lazily*, trading off context switching overhead and wasted computing efforts. Specifically, each thread may abort *eagerly* as soon as an operation fails, thereby introducing minimal impact on the ongoing execution of other operations. In contrast, we can also handle aborts *lazily* after the entire TPG has been fully explored to minimize context switching overhead.

4.3.1 Eager Aborting (e-abort). The implementations of e-abort are different under varying exploration strategies.

A) Structured Eager Aborting: When s-explore is adopted, e-abort eagerly handles abort in a stratified way. After all operations of a stratum have been processed, threads start to handle aborts when the processing of any operations fails. It works in two steps. First, threads abort the failed operations and their logically dependent operations. Second, to update the affected operations that would be executed after aborted operations, threads rollback and redo from the most outer stratum containing aborted operations.

B) Non-structured Eager Aborting: When ns-explore is adopted, e-abort handles aborts based on a *coordinator*-based mechanism to ensure that we only need to handle abort once even if multiple operations of the transaction fail. Specifically, we mark the first operation of each state transaction as the *head*. During scheduling, a thread that picks up a *head* operation will further function as a *coordinator* thread for that transaction. Once a thread finds an operation processing failed, it notifies the corresponding *coordinator* thread to start handling the abort of that transaction. When the coordinator receives the failure notification, it enforces abort handling by 1) aborting all operations of the transaction; 2) propagating notifications to other threads, in order to roll back and redo all dependent operations.

4.3.2 Lazy Aborting (l-abort). In contrast to e-abort, threads can record failed operations during their execution but do not immediately handle them. Only when the TPG has been fully explored, i.e., all operations have been picked up and processed (committed or aborted), threads cooperatively abort the failed operations, and those operations that are logically dependent on aborted operations. All aborted operations are removed from the TPG, and *MorphStream* redoes the entire TPG from the beginning. Such an approach is simple to implement and minimizes the context switching overhead, and is adopted by TStream [40]. However, it may need to repeatedly check for transaction abort and perform multiple iterations, until the TPG does not contain any operations to be aborted. This can generally lead to significant wasted computing efforts compared to e-abort.

4.3.3 Rollback based on Multi-Versioning State Storage. To guarantee a correct schedule (Definition 3), we must roll back the states that have been modified by the aborted operations. This is needed under both e-abort and l-abort. We create an additional physical copy of each state whenever it is modified. For each copy of the state, we further annotate a timestamp of the writing operation to specify its version. When an operation is aborted, the associated state will be rolled back to the version with the latest timestamp smaller than the aborted operation. The physical copies can be removed after the current batch of transactions is fully processed (i.e., committed or aborted). It is noteworthy that such multi-versioning state storage is also used to support windowing queries in *MorphStream* (Section 5.4).

Table 3: Workload characteristics to TPG Properties.

Type	S-TPG Prop.	Workload Char.
Vertex	Computation Complexity	C
	Vertex Degree Distribution	$\underset{\sim}{\propto} \theta$
	Ratio of Aborting Vertexes	$\underset{\sim}{\propto} \alpha$
Edge	Number of LDs	$\underset{\sim}{\propto} T * l$
	Number of TDs	$\underset{\sim}{\propto} T * l$
	Number of PDs	$\underset{\sim}{\propto} T * l * r$
	Cyclic Dependency	Correlated to θ, T, l, r

4.4 Heuristics Decision Model

The original task graph scheduling problem is NP-complete [24]. The additional stochasticity (e.g., arbitrary user-defined functions and transaction aborting) in our scheduling context further leads to large solution space. We hence propose a heuristic-based lightweight decision model to guide *MorphStream* to make a proper scheduling decision at runtime. Our decision model is summarized based on our extensive microbenchmark studies and theoretical analysis as summarized in Table 2.

Model Inputs. The decision model takes seven properties of a constructed TPG summarized in Table 3 as inputs. Those properties depend on various *workload characteristics*. *(1) Vertex Computational Complexity* is mapped to the complexity of the user-defined function in the corresponding state access operation (C); *(2) Vertex Degree Distribution* is proportional to the state access distribution of the corresponding operation (θ), where certain states may be more likely to be accessed than others; *(3) Ratio of Aborting Vertexes* is proportional to the ratio of operations that need to be aborted (a), which needs to be profiled and estimated; *(4,5,6) Number of* LDs, TDs, PDs is proportional to the number of transactions arriving during the batch interval (T) and the transaction length (l). The number of PDs is also proportional to the number of state accesses per operation (r) since an operation may need to read multiple states to resolve PDs. *(7) Cyclic Dependency* refers to whether there are cyclic dependencies formed when c-schedule is adopted, and is correlated to θ, T, l, and r. Note that, most properties can be obtained during the construction of TPG, while a few properties, such as the average computational complexity per vertex, need to be instantiated with profiling.

Decision Model. The overview of our decision model is shown in Figure 6. Note that the qualitative remarks of a high, medium and low are relative and the quantitative value depends on actual hardware and workloads. Based on the aforementioned properties of the TPG, our decision model makes the scheduling decision along three dimensions in parallel at runtime.

I) Decision of Exploration Strategies: The model selects the s-explore strategy when two conditions are met at the same time: a) the number of all three types of dependencies is high, and b) the vertex degree distribution is uniform, and the workload is balanced among threads. In such a case, s-explore reduces the context

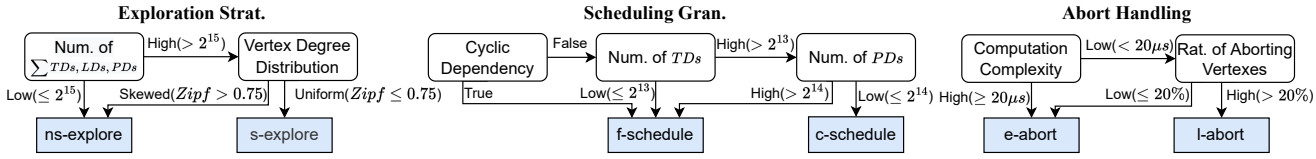


Figure 6: The lightweight decision model. The concrete threshold numbers in brackets are based on our experiments.

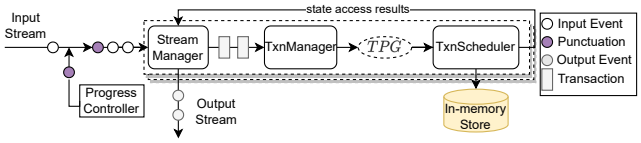


Figure 7: The system architecture of *MorphStream*. The constructed TPG and shared state are stored in memory.

switching overhead during exploration based on the stratified auxiliary structure of the TPG for cooperative exploration. The model selects the ns-explore strategy for the remaining cases, as it can resolve dependencies more flexibly. In particular, ns-explore performs especially better when the distribution of dependencies is skewed.

II) Decision of Scheduling Unit Granularities: The model selects c-schedule when three conditions are met at the same time: a) there is no cyclic dependency among groups of operations, as the overhead for removing cycling dependencies is significant; b) the number of TDs is high, as it leads to a lower context-switching overhead on resolving dependencies; and c) the number of PDs is low, as it leads to fewer dependencies among groups of operations. The model selects f-schedule for the remaining cases to leverage its better scalability.

III) Decision of Abort Handling Mechanisms: The model selects l-abort when two conditions are met at the same time: a) the computation complexity of vertexes is low, as this leads to a low redo overhead; and b) the ratio of aborting vertexes is high, as when many transactions need to be aborted, they can be handled together, reducing overall context switching overhead. The model selects the e-abort mechanism for the remaining cases, as it has minimal impact on the ongoing execution of other operations.

5 IMPLEMENTATION DETAILS

Compared to existing non-adaptive scheduling approaches, much of the additional system overhead of *MorphStream* comes from constructing and exploring the TPG concurrently and correctly, considering that input events may arrive *out-of-order*. In the following, we first discuss the system architecture of *MorphStream*. Next, we discuss the details of the implementation of *MorphStream* that significantly reduces TPG construction and exploration overhead. Finally, we also discuss some limitations of *MorphStream*.

5.1 System Architecture

Figure 7 shows the modularized system architecture of *MorphStream* with four key components: *ProgressController* (PC), *StreamManager* (SM), *TxnManager* (TM), and *TxnScheduler* (TS). The *PC* is a singleton component that assigns monotonically increasing timestamps to its generated punctuations. A simple global counter is used by *PC* to track punctuations. The other three components are instantiated in each thread locally. The *SM* conducts preprocessing and postprocessing for every input event (e). Similar to some prior works [40], state transactions may be issued but not immediately processed during preprocessing. Only when state transactions are processed (i.e., committed or aborted) the associated input events can be postprocessed by the *SM* based on transaction processing results. The *TM* handles dependency resolution among state transactions and inserts decomposed operations to construct a TPG. The *TM* is also involved upon receiving punctuation to refine the constructed TPG with further dependency resolution. We discuss the detailed two-phase TPG construction process shortly later in Section 5.2. The *TS* schedules operations for concurrent execution based on the constructed TPG according to the three dimensions of scheduling decisions as discussed previously in Section 4.

5.2 Parallel TPG Construction

MorphStream needs to construct a TPG for every batch of state transactions with low overhead. The main challenge is to efficiently identify the three types of dependencies (TDs, LDs, PDs), i.e., the edges. Upon arrival, transactions are decomposed into atomic state access operations, which are the vertexes of the TPG, accordingly. During this decomposition, LDs can be identified among operations from the same transaction. However, TDs and PDs can not be identified immediately because the arrival of transactions may be *out-of-order*. To address this issue, we partition the TPG construction process into two steps during the stream processing phase and transaction processing phase, correspondingly. We illustrate in Figure 8 how *MorphStream* constructs a TPG in parallel with the same running example as before.

TPG construction during stream processing phase: Upon arriving, $txn_1 \sim txn_3$ are immediately decomposed into atomic state access operations $O_1 \sim O_5$. LDs are identified among operations from the same transaction according to their statement orders. A TPG G is constructed by inserting operations as vertexes and LDs as edges, correspondingly. To help identify TDs among operations, which may arrive out-of-order, all operations are inserted into key-partitioned *sortedLists* (i.e., a concurrent skip list [15]), where the key is the targeting state of each operation, i.e., state A or B in this example, and sorted by timestamp. To help identify PDs during the next phase, for each write operation $O_i = Write(k, f(k_1, k_2, ...k_n))$, we additionally maintain n "proxy operations" of O_i. Each "proxy operation" is a read operation, denoted as $PO_i^{k_j}$, $k_j \in k_1, k_2, ...k_n$, inserted into *sortedLists* of $k_1, k_2, ...k_n$, correspondingly. For example, for O_3 and O_5 whose write function depend on states A or B, we further insert a "proxy operation" (denoted as PO_3^A and PO_5^B) into the *sortedList* of A and B, correspondingly.

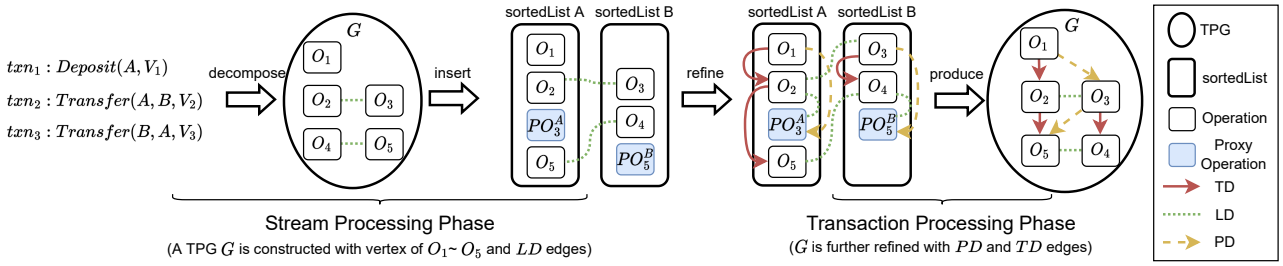
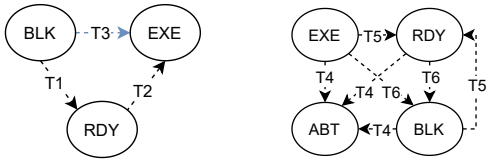


Figure 8: A running example of a TPG construction process involving three state transactions, which may arrive out-of-order.

TPG construction during transaction processing phase: During the transaction processing phase, all further state transactions are blocked until *MorphStream* returns back to the stream processing phase. We can now identify TDs and PDs efficiently with the help of the constructed *sortedList* and "proxy operations" during the stream processing phase. First, TDs can be identified in a straightforward way by iterating through operations inserted into each *sortedList*, i.e., O_1, O_2, O_5 and O_3, O_4. Note that "proxy operations" are not involved in identifying TDs. Second, PDs can be identified according to the inserted "proxy operations". In particular, O_i is parametric dependent on the precedent write operation of "proxy operation" in the *sortedList*. In this example, PDs can be identified among O_1 and PO_3^A and among O_3 and PO_5^B. After identifying all TDs and PDs, we insert them as edges to G, which captures the complex workload dependencies of the current batch of state transactions.

Table 4: State Definition in the TPG.

State	Definition
Blocked (*BLK*)	Operation is not ready to schedule
Ready (*RDY*)	Operation is ready to schedule
Executed (*EXE*)	Operation is successfully processed
Aborted (*ABT*)	Operation is aborted



(a) Transition during execution. **(b) Transition during abortions.**

Figure 9: Six cases of state transition flow of an operation.

5.3 Stateful TPG Management

MorphStream has to explore a TPG efficiently while ensuring transactional semantics. The main challenge is to manage the life cycles of concurrent operations and commit/abort transactions correctly. To this end, *MorphStream* annotates a *finite state machine* in each vertex of the TPG. Each vertex of the TPG can be in one of the four states, summarized in Table 4. **(1) Blocked (*BLK*)**: denotes that a vertex is not ready for scheduling due to unsolved dependencies. **(2) Ready (*RDY*)**: denotes that a vertex is ready for scheduling, as all of its dependencies are resolved. **(3) Executed (*EXE*)**: denotes that a vertex has been processed successfully. **(4) Aborted (*ABT*)**: denotes that a vertex has been aborted either due to failed processing of itself or its dependent vertexes.

MorphStream continuously tracks state transitions in every vertex of TPG to guarantee a correct schedule while making dynamic scheduling decisions. Figure 9 summarizes two scenarios of state transition, including six cases (i.e., **T1**~**T6**). **T1:** When all dependencies of an operation in *BLK* state are resolved, i.e., all of its dependent operations are in *EXE* state, it transits to the *RDY* state and becomes available to schedule. **T2:** After the processing of an operation in *RDY* state succeeds, it transits to *EXE*. **T3:** Operations in *BLK* state may be speculatively scheduled with unsolved dependencies for higher execution concurrency, so it can directly transit to *EXE*. **T4:** The state will be transited to *ABT* from any state when the processing of an operation is failed or its logical dependent operations transit to *ABT*. **T5:** Instead of simply rolling back all affected operations to *BLK* state, we can avoid costly re-exploring of the TPG. Specifically, we can *proactively* check whether the operation is ready to execute immediately after rollback, i.e., when dependent operations transit to *ABT*, the current operation under *EXE* or *BLK* will transit to *RDY* and execute immediately. **T6:** When dependent operations roll back to *RDY*/ *BLK*, the current operation has to roll back to *BLK* because of unsolved dependencies. The abort handling is completed when all operations are processed, after *ABT* operations transit to either *RDY* or *BLK*.

5.4 Limitations

In this paper, we assume there is no system failure at runtime. Guaranteeing fault tolerance without sacrificing low latency and high throughput during normal operation is still an open challenge for TSPEs [4, 22, 40]. This is a challenging problem even in a single node setting because of the non-trivial combination of both transaction and stream-oriented properties in TSPEs. Another major limitation of *MorphStream* is the support for stream window queries. Instead of only accessing the recent states (Def 1), an operation may need to further access a time range of states. Due to its significant complexity, we leave it as future work.

6 EVALUATION

In this section, we conduct a detailed experimental evaluation comparing *MorphStream* to the alternative approaches. We have made the following key observations.

- Our experimental results show that *MorphStream* outperforms conventional SPEs for TSP applications (Section 6.2.1) by orders of magnitude. Because of the adaptive scheduling strategy, *MorphStream* achieves up to 2.2x higher throughput and

69.1% lower latency compared to the state-of-the-art TSPEs (Section 6.2.2 and 6.2.3).
- In Section 6.3, we show that *MorphStream* spends more time on TPG construction and exploration, but largely reduces the overhead of synchronization. A drawback of *MorphStream*, however, is its high memory consumption (about 1.4x times higher) due to the more complex auxiliary data structures.
- We show that no one scheduling strategy can outperform others in all cases (Section 6.4). Each scheduling decision has its own advantages and disadvantages under varying workload characteristics.
- In Section 6.5, we show that *MorphStream* spends up to 2.3x fewer clock ticks and has a lower memory bound than TStream and S-Store. Furthermore, *MorphStream* has much better multicore scalability compared to prior schemes.

6.1 Evaluation Methodology

We conduct all experiments on a dual-socket Intel Xeon Gold 6248R server with 384 GB DRAM. Each socket contains 24 cores of 3.00GHz and 35.75MB of L3 cache. To isolate the impact of NUMA, we use one socket of the server in our experiments. We leave it as a future work to address the NUMA effect [27]. We pin each thread on one core and assign 1 to 24 cores to evaluate the system scalability. The OS kernel is *Linux 4.15.0-118-generic*. We use *JDK 1.8.0_301*, set *-Xmx* and *-Xms* to be 300 GB. We use G1GC as the garbage collector across all the experiments and configure *MorphStream* to not clear temporal objects such as the processed TPGs and multi-versions of states. We show the impact of clean-up and JVM GC in Section 6.3.2.

We use three use cases: Streaming Ledger (SL), Grep&Sum (GS), Toll Processing (TP) from a benchmark proposed by our previous work [40] on evaluating the effectiveness of *MorphStream*. For all these workloads, we follow the original application logic but tweak the configurations to bring more workload dependencies such that we can better expose the issues of existing TSPEs. In particular, we have configured ten times larger sizes of shared mutable states and generated more state transactions accessing overlapped states in our workload settings.

Tuning Workload Characteristics. To better comprehend the system behaviour, we tune the following six workload characteristics. The default workload characteristics and varying ranges are summarized in Table 5. *1). State Access Distribution (θ):* Similar to [40], we modelled the state access distribution as Zipfian skew, and tune the Zipfian factor to vary θ. *2). Ratio of Aborting Transactions (a):* We tune a by artificially adding transactions that violate the consistency property, such as the account balance can not become negative. *3). Transaction Length (l):* We tune l by varying the number of atomic state access operations in one transaction. *4). Computational Complexity of an Operation (C):* We tune C by adding a random delay in each user-defined function (i.e., the f in Definition 6). *5). Number of State Access Per Operation (r):* We vary the ratio of multiple state access operations to tune r. *6). Number of Transactions (T):* We tune T by varying the punctuation interval.

Dynamic Workload Configurations. To evaluate the adaptability of *MorphStream*, we follow the mechanism proposed by Ding et al. [13] to generate dynamic workloads. Due to a large number of tunable workload configurations and their wide range of adjustments, it is difficult to adjust all parameters in one dynamic workload. Therefore, we generate various phases of dynamic workloads by different trends, which determines the parameters we want to tune and how they change over time. For example, in a dynamic workload with an increasing tendency to abort transactions, we will increase the ratio of aborting transactions over time.

Table 5: Workload default configuration.

Workload Char.	SL	GS	TP	Tweaking ranges
θ	0.20	0.20	0.20	0.0~1.0
a	1%	1%	1%	0~90%
l	2 / 4	1	2	1~10
C	10 us	10 us	10 us	0~100 us
r	1 / 2	2	1	1~10
T	10240	10240	40960	5120~81920

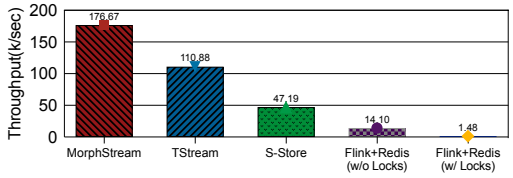


Figure 10: Performance comparison among *MorphStream* and existing systems for running SL on 24 cores.

6.2 Performance Evaluation

In this section, we conduct a series of experiments to confirm *MorphStream*'s superiority compared to the state-of-the-art.

6.2.1 Comparing to Conventional SPEs. In the first experiment, we show that TSPEs significantly outperform conventional SPEs when handling TSP applications. We use the default workload configuration shown in Table 5 in this study. We have implemented SL on Flink-1.10.0. Since the native Flink does not support shared mutable state accesses, we leveraged Redis-6.2.6 with a distributed lock, a common workaround, to store shared mutable states. We deploy a standalone cluster with a single TaskManager configured with 24 slots and set the parallelism of SL to 24. To avoid the OOM exception, we set the TaskManager heap size to 100GB. When locking is disabled (denoted as w/o Locks), execution correctness is not guaranteed in Flink. The detailed workload configuration is shown in Table 5. As shown in Figure 10, *MorphStream* significantly outperforms the two state-of-the-art TSPEs, TStream (1.6x) and S-Store (3.7x), and Flink (up to 117x). It is noteworthy that Flink, a popular conventional SPE, achieves orders of magnitude lower throughput in this application. By disabling locks, its throughput increases but is still far lower than any of the TSPEs. In the following, we hence do not further compare *MorphStream* with Flink.

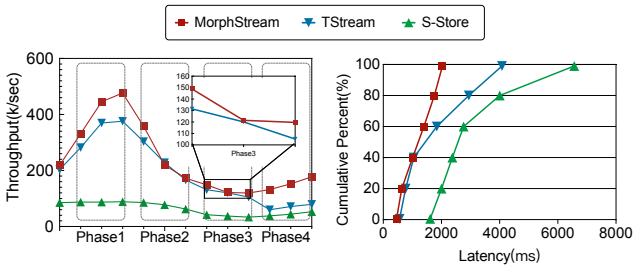
6.2.2 Evaluation on Dynamic Workloads. In this experiment, we show that *MorphStream* can always select a better-performing scheduling strategy under changing workloads, resulting in lower latency and higher throughput compared to state-of-the-art TSPEs. We use SL [2] as the base application and divide the workloads into four phases. Figure 11a and Figure 11b compare the throughput and latency of *MorphStream* against two state-of-the-art TSPEs: S-Store [22] and TStream [40]. We mark each phase in the dynamic

workload using the dotted grey box. In the first phase, a large number of events consisting of *Deposit* transactions arrive, and the state accesses distribution is scattered. As a result, there are lots of LD and TD but few PD. At the same time, the vertex degree distribution is uniform as the state accesses are scattered. As guided by our decision model (Figure 6), *MorphStream* selects the s-explore strategy to resolve a large number of dependencies and selects c-schedule for scheduling since there are fewer PD. *MorphStream* achieves up to 1.27 times higher throughput compared to the second-best. In the second phase, we configure the workload with increasing key skewness over time. Hence, dependencies are gradually contented among a small set of states, which facilitates the resolution of dependencies. As expected, the performance of all approaches drops. *MorphStream* gradually morphs from s-explore to ns-explore strategy to resolve dependencies in a more flexible manner, and constantly outperforms S-Store. In the third phase, we configure the workload with an increasing ratio of *Transfer* transactions so that one of the two types of transactions in SL is called intensively in a short period of time. As the proportion of *Transfer* transactions increases, there are more and more dependencies between scheduling units. Hence, *MorphStream* gradually morphs from c-schedule to f-schedule to reduce the dependency resolution overhead and result in a stable throughput.

There is no transaction abort in the first three phases, and the selection of aborting mechanism in *MorphStream* does not matter. In the fourth phase, we increase the ratio of aborting transactions over time to evaluate the performance of the system under a dynamically changing ratio of aborting transactions. In the beginning, *MorphStream* applies the e-abort mechanism to eagerly abort when the operation fails and morphs to l-abort when aborts are frequent so that transaction aborts can be handled together to reduce context switching overhead. The results show that TStream's performance drops when transaction aborts appear. This is because of the rapidly increasing overhead of redoing the entire batch of transactions. In contrast, *MorphStream* achieves relatively stable performance and is 2.2x to 3.4x higher than other schemes.

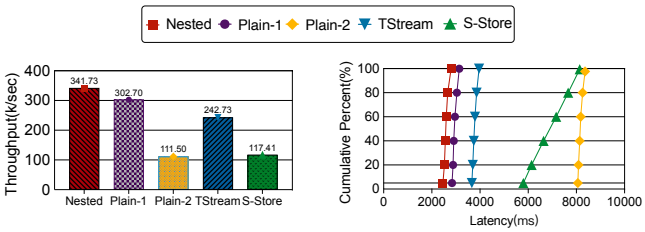
A further key takeaway from Figure 11b is that TStream and S-Store have significantly higher tail latency than *MorphStream*. This is mainly because the scheduling strategies in TStream and S-Store are limited for specific workload characteristics. When workload changes, such as increasing transaction aborts or key skewness, their efficiency drops significantly, resulting in higher processing latency. In contrast, *MorphStream* dynamically morphs the scheduling strategy according to the change of workload characteristics to deal with different situations, thus achieving a constantly lower processing latency.

6.2.3 Evaluation of Multiple Scheduling Strategies. In Toll Processing (TP), the road conditions in different regions can have varying characteristics, which can be divided into multiple groups. TStream [40] and S-Store [22]'s scheduling strategies may work well on one group of state transactions but not on another. In contrast, *MorphStream*'s modular and flexible design allows it to employ multiple scheduling strategies concurrently. For illustration, we configure the TP to contain two groups of state transactions simultaneously. In *group 1*, the state access distribution of state transactions is skewed, and the ratio of



(a) Throughput **(b) Latency**

Figure 11: Evaluation on Dynamic Workload.



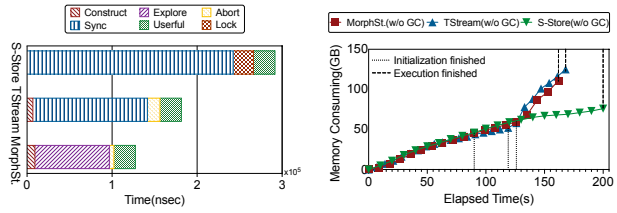
(a) Throughput **(b) Latency**

Figure 12: Single (plain, TStream, S-Store) vs. Multiple (nested) Scheduling Strategies.

aborting transactions is high. In *group 2*, the state distribution of state transactions is uniform and transaction aborts occur rarely. As guided by our decision model (Figure 6), *MorphStream* applies ns-explore, c-schedule, and l-abort for handling transactions of *group 1*, and applies s-explore, c-schedule, and e-abort for handling transactions of *group 2*. We name such a combination of strategies a *nested* configuration.

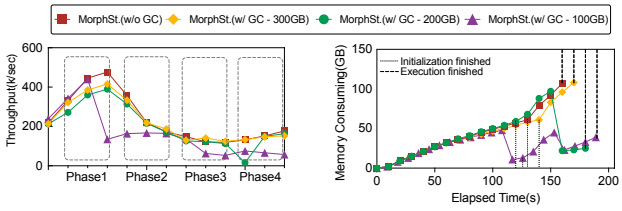
Figure 12a shows the throughput comparison results. We can see that the throughput of the nested configuration is 40.9% higher than TStream and 117% higher than S-Store. To further comprehend the advantage of the nested configuration, we compare it against two plain scheduling strategy: ns-explore, c-schedule, and l-abort (denoted as *plain-1*) and s-explore, c-schedule, and e-abort (denoted as *plain-2*) for handling all transactions from both groups. Unsurprisingly, the throughput of the nested configuration is 1.17× and 2.87× higher than that of each plain scheduling strategy. When the ratio of aborting transactions in *group 1* is high, the *plain-2* is bottlenecked by the frequent context switching overheads. At the same time, as the skewness of state access increases in *group 1*, the workloads become less balanced among threads, hampering the system performance when using s-explore in *plain-2*. As the state distribution of state transactions is uniform and the ratio of aborting transactions is low in *group 2*, the *plain-1* spends more time resolving dependencies and redoing the entire batch of transactions. The *plain-1* performs better than *plain-2* as there are fewer PD and the computation complexity is low, but it is still lower than that of the *nested* setting.

Figure 12b shows the comparison results of end-to-end processing latency. Thanks to the significantly improved performance, *MorphStream* with nested configuration achieves very low processing latency. Note that, S-Store spends more time on synchronization and inserting locks under a highly contended



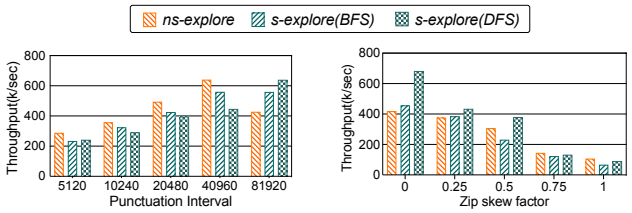
(a) Runtime Breakdown (b) Memory Footprint

Figure 13: System overhead.



(a) Throughput (b) Memory Footprint

Figure 14: Impact of clean-up under varying JVM size.



(a) Varying Punctuation Interval (b) Varying Workload Skewness

Figure 15: Exploration strategy decision.

workload in *group1* because dependent transactions are executed serially, resulting in linearly increasing latency. Under a higher ratio of aborting transactions in *group 1*, *plain-2* spends lots of time achieving fine-grained state rollback because the context-switching overhead and synchronization overhead are huge with s-explore (Table 2), which is why *plain-2* has the highest latency compared to other schemes.

6.3 Overhead

MorphStream achieves adaptive scheduling at the cost of more complex runtime operations such as data structures constructing and exploring available state access operations. These extra operations can negatively impact the system in the following two ways: 1) the complex construction and exploration process may increase the latency of transaction processing, and 2) it will increase the memory consumption of the application due to the auxiliary data structure.

6.3.1 Latency overhead. Following a prior work [40], we show the time breakdown of processing input transactions in the following aspects. 1) *Useful Time* refers to the time spent on doing actual work including accessing shared mutable states and performing user-defined functions. 2) *Sync Time* refers to the time spent on synchronization, including blocking time before lock insertion is permitted or blocking time due to synchronization barriers during mode switching. 3) *Lock Time* refers to the time spent on inserting locks after it is permitted. 4) *Construct Time* refers to the time spent on constructing the auxiliary data structures, e.g., TPG in *MorphStream* and operation chains in TStream. 5) *Explore Time* refers to the time spent on exploring available operations to process. 6) *Abort Time* refers to the wasted computation time due to abort and redos.

Figure 13a shows the time breakdown when the system runs the dynamic workload in Section 6.2. There are three key takeaways. First, although TStream and *MorphStream* spend a significant portion of time during construction (*Construct Time*), they successfully reduce synchronization (*Sync Time*) and lock (*Lock Time*) overhead compared to S-Store. This explains their better performance on multicore processors. Second, TStream has the highest abort time (*Abort Time*) because TStream has to redo the entire batch of transactions when a transaction abort happens. In contrast, S-Store spends little time in abort as it involves little redo of state transactions because dependent transactions are executed serially. Third, we can see that *MorphStream* still spends a significant fraction of time performing exploration (*Explore Time*). This is mainly caused by excessive message-passing among threads. In the future, we plan to investigate more efficient exploration strategies such as prioritizing mechanisms [20] in *MorphStream*.

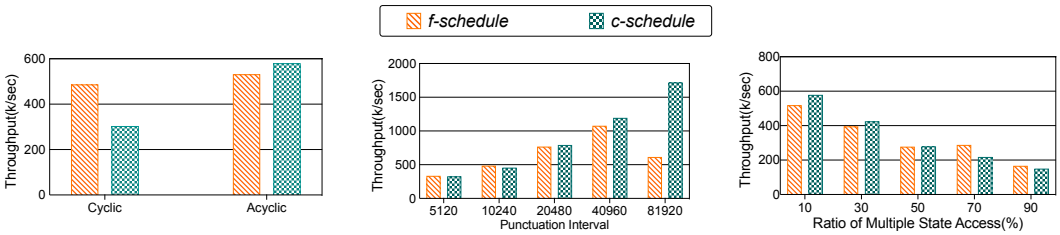
6.3.2 Memory footprint. In this study, we use the dynamic workload in Section 6.2 to evaluate memory footprint. Figure 13b demonstrates the memory consumption of three TSPEs without GC involved. In the initialization stage, the memory consumption of all systems is almost the same. *MorphStream* spends more time during initialization compared to TStream as it needs to initialize more data structures to support adaptive scheduling. During runtime, *MorphStream* and TStream consume a similar amount of memory per batch of state transactions, and both consume much more than S-Store. This is because they construct auxiliary data structures for scheduling, and especially they may maintain multiple physical copies of each state at different timestamps during execution (Section 4.3). Note that, as we have configured *MorphStream* to not clear temporal objects and the JVM size to be large enough (300GB), the total memory usage keeps increasing until execution is finished, during which no GC is triggered.

6.3.3 Clean-up and GC overhead. Figure 14 shows the impact of clean-up under varying JVM size from 100GB to 300GB. We can see that enabling clear temporal objects brings down the performance of *MorphStream* up to 12.8%, and still outperforms TStream and S-Store. In Figure 14(b), the memory usage fluctuates up and down when the JVM size is set to 100GB or 200 GB because the JVM periodically reclaims (GC) the temporary objects in the continued processing of data streams.

6.4 Impact of Scheduling Decisions

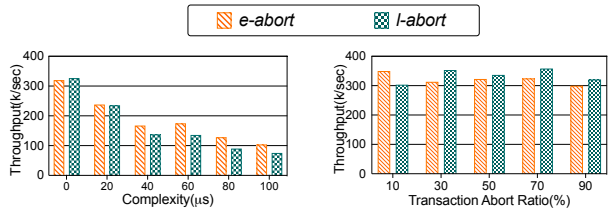
In this section, we evaluate the impact of varying scheduling decisions under different workload characteristics using Grep&Sum (GS) due to its flexibility.

6.4.1 Impact of Exploration Strategies. We first study the impact of different exploration strategies (i.e., ns-explore vs. s-explore). As discussed in Section 4.4, two key factors affect selecting better-performing exploration strategies, namely 1) the *punctuation*



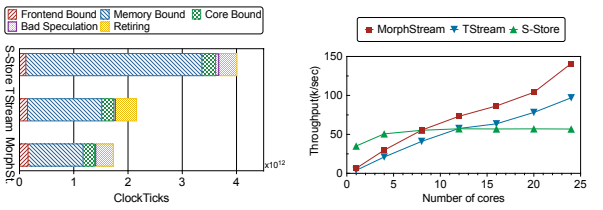
(a) Cyclic/Acyclic (b) Varying Punctuation Interval (c) Varying Ratio of Multi-Accesses

Figure 16: Granularity decision.



(a) Varying Computation Complexity (b) Varying Abort Ratio

Figure 17: Abort handling decision.



(a) Micro-architectural Analysis (b) System Scalability Comparison

Figure 18: Impact of Modern Hardware.

interval and 2) *workload skewness*. Figure 15a shows the effects of selecting different exploration strategies under varying *punctuation interval* and low *workload skewness*. ns-explore works better when *punctuation interval* is low, while s-explore works better when *punctuation interval* is high. This is because there is a linear proportional relationship between the *punctuation interval* and the number of dependencies (TD/PD) of the constructed TPG. When the *punctuation interval* is low, ns-explore resolves the rare dependencies as soon as an operation has been successfully processed, leading to higher system concurrency. s-explore works better otherwise as the notification overhead of the ns-explore approach keeps increasing with more dependencies in the workloads. However, s-explore has a constant construction and synchronization overhead for dependencies resolution. Figure 15b shows the effects of selecting different exploration strategies under varying *workload skewness* and high *punctuation interval*. We can see that s-explore works better when the state accesses are uniformly distributed, i.e., the Zipf skew factor is 0. ns-explore works better when the state accesses are skewed. This is because a skewed workload leads to load unbalance among threads and intensifies the synchronization overhead when s-explore is applied as summarized in Table 2.

6.4.2 Impact of Scheduling Granularities. In this section, we study the effectiveness of different scheduling granularities (i.e., f-schedule v.s. c-schedule), which are affected by the following key workload characteristics, namely *cyclic/acyclic*, *number of state access*, *punctuation interval*, and the *ratio of multi-accesses*. First, Figure 16a shows the results of different scheduling granularities under the workload with or without cyclic dependencies. c-schedule permits better performance when there is no cyclic dependency among the batched scheduling units since each thread can schedule a group of operations together as scheduling unit to amortize the context switching overheads. However, our further experiments reveal that when there is a large number of state access, f-schedule is always better than c-schedule, regardless of whether there are circular dependencies. This is mainly due to the fact that even without circular dependencies, a large number of state accesses will increase the number of PD, causing a significant overhead on resolving the dependencies among operations of the same group. Second, Figure 16b shows how varying *punctuation interval* affect the selection of scheduling unit granularities when there are no cyclic dependencies. We set the number of state accesses to one to avoid the effect of PD, so the punctuation interval only controls the number of TD in the TPG. We can see that c-schedule achieves higher throughput at higher punctuation intervals. When the punctuation interval is high, the large number of TD increases the context-switching overhead in f-schedule, which is why the performance of f-schedule decreases when punctuation interval is large, such as 81920. In contrast, c-schedule schedules the operations in group resulting in lower context-switching overhead on resolving TD compared to the f-schedule. Third, Figure 16c shows that f-schedule works better when the ratio of multiple state access is high, while c-schedule works better when the ratio is low. The ratio of multiple state access controls the number of PD among operations, as we can see that the PD affects the performance of c-schedule significantly. This is mainly because the execution concurrency drops when the number of PD is high.

6.4.3 Impact of Abort Handling Mechanisms. Finally, we study the impact of two abort handling mechanisms (i.e., e-abort v.s. l-abort) mainly affected by the *abort ratio* and the *computation complexity* workload characteristics. Figure 17a shows the comparison results of varying *computation complexity* when the *abort ratio* is high. A lower computation complexity leads to a low redo overhead, and l-abort handles frequent aborts together to reduce the context switching overheads. e-abort is better otherwise, as it makes a minimum impact on the ongoing execution of other operations. Figure 17b shows the results of different abort handling mechanisms

under different abort ratios when the computation complexity is low. The results indicate that as the ratio of aborting transactions increases, l-abort works better. The key reason is that when the computation complexity is low, the context-switching overheads and the synchronization overhead among threads to achieve fine-grained state rollback and redo become the major bottlenecks.

6.5 Impact of Modern Hardware

In this section, we compare *MorphStream* with existing TSPEs on how they interact with modern multicore processors from the modern hardware architecture perspective.

Micro-architectural Analysis. We take SL as an example to show the breakdown of the execution time according to the Intel Manual. Figure 18a compares the time breakdown of different TSPEs. We measure the hardware performance counters through Intel Vtune Profiler during the algorithm execution and compute the top-down metrics. We have three major observations. First, the breakdown results reaffirm our previous analysis that *MorphStream* spends up to 2.3x fewer clock ticks for transaction processing compared to TStream and S-Store, because of its more efficient adaptive scheduling strategies. Second, all three TSPEs are Memory Bounded, i.e., a large proportion of CPU cycles are spent due to memory access instructions: *MorphStream* (58.5%), TStream (63.3%), and S-Store (80.9%). The detailed profiling with Intel Vtune Profiler reveals that it is commonly due to the heavy usage of latches to resolve dependencies among transactions while accessing the shared-mutable state. Both TStream and S-Store have a higher Memory Bound than *MorphStream* due to the higher synchronization cost. Nevertheless, Figure 18a and Figure 13a jointly indicate that *MorphStream* can adopt more efficient exploration strategies to further improve its performance.

Multicore Scalability. Figure 18b shows the scalability comparison among TSPEs, with two major observations. First, *MorphStream* outperforms prior schemes with an increasing number of cores confirming the good scalability of *MorphStream*. However, there is still a large room for further improving *MorphStream* towards linearly scale-up, the reason being that it becomes memory bounded as Figure 18a previously shown. Second, when the number of cores is low, *MorphStream* performs even worse than S-Store due to the large constant overhead of TPG construction process. In a resource constraint setting, existing non-adaptive solutions may be more favoured.

7 RELATED WORK

Unlike key-value stores, database systems, or conventional SPEs, TSPEs such as Streaming-Ledger [1], S-Store [22], FlowDB [3], TStream [40], and *MorphStream* are based on a unique computational model, where each input tuple from data streams may involve multiple keys. Thus the processing of tuples can lead to potentially conflicting shared mutable state accesses. Such a unique system feature has been originally motivated by a list of stream applications [8, 34] and is applied or encouraged to be applied in emerging use case scenarios [1, 21, 25]. Some of the novel design challenges and optimization opportunities of TSPEs have been discussed in previous works [22, 40]. The experimental results shown previously in Figure 10 also confirm that conventional SPEs can not efficiently handle the targeted applications of TSPEs.

Executing each state transaction one by one following the event sequence naturally leads to the correct schedule but seriously limits system concurrency [34]. Recent works have proposed adopting partitioning and decomposition to optimize the performance of transaction processing, such as [7, 26, 36, 38]. Similar ideas have also been adopted in TSPEs. For example, S-Store [22] adopts state partitioning with extensions of guaranteeing state access ordering [11], while TStream [40] adopts transaction decomposition to improve multicore scalability further. However, each existing system is designed with a non-adaptive scheduling strategy and favours a subset of workload characteristics. *MorphStream* deviates from existing solutions. It explores fine-grained workload characteristics of every batch of state transactions. It then makes the correct scheduling based on a decision model to morph the current scheduling strategy into a better-performing strategy.

Despite the large body of research on the scheduling problem in a general context [17, 27, 37], task scheduling for TSPEs presents subtle but unique requirements [14, 18, 19, 24], largely due to the integrated stream processing and transactional contexts [40]. For instance, the scheduling unit can be reconfigured by the system, e.g., by splitting state transactions into operations and regrouping by keys. It is thus difficult (if not impossible) to quantitatively model the objective function of scheduling plans in TSPEs. We hence propose to model the scheduling of TSPEs into a three-dimensional scheduling decision problem and guide it with a heuristic-based decision model. Furthermore, the scheduling overhead is now on the critical path, prohibiting any sophisticated optimization algorithms.

8 CONCLUSION

Transactional stream processing engines (TSPEs) have been increasingly gaining traction. In this work, we show that the scheduling strategies of TSPEs can be decomposed into three dimensions of scheduling decisions, exhibiting trade-offs among execution concurrency, context switching overhead, and wasted computational efforts due to aborts. To this end, we propose a novel TSPE *MorphStream* that is able to morph flexibly among scheduling strategies adapting to dynamically changing workload characteristics. Guided by a lightweight decision model, *MorphStream* can make the correct scheduling decision at runtime with minor overheads, which yields a multi-times performance improvement over the state-of-the-art.

9 ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable comments and thank Aqif Hamid for joining the initial implementation while working on his master's thesis at TU Berlin. This work is supported by the National Research Foundation, Singapore and Infocomm Media Development Authority under its Future Communications Research & Development Programme FCP-SUTD-RG-2022-005, FCP-SUTD-RG-2022-006, and a SUTD Start-up Research Grant (SRT3IS21164). Haikun Liu's work is supported by the National Natural Science Foundation of China under grant No.62072198. Volker Markl's work is supported by the DFG Priority Program (MA4662-5), the German Federal Ministry of Education and Research (BMBF) under grants 01IS18025A (BBDC - Berlin Big Data Center) and 01IS18037A (BIFOLD - Berlin Institute for the Foundations of Learning and Data).

REFERENCES

[1] 2018. Data Artisans Streaming Ledger Serializable ACID Transactions on Streaming Data, https://www.data-artisans.com/blog/serializable-acid-transactions-on-streaming-data. (2018).

[2] (2018). *Serializable ACID Transactions on Streaming Data.* https://www.ververica.com/blog/serializable-acid-transactions-on-streaming-data

[3] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. 2017. FlowDB: Integrating Stream Processing and Consistent State Management. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems* (Barcelona, Spain) *(Debs '17)*. Acm, New York, NY, USA, 134–145. https://doi.org/10.1145/3093742.3093929

[4] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. 2020. TSpoon: Transactions on a stream processor. *J. Parallel and Distrib. Comput.* 140 (2020), 65–79. https://doi.org/10.1016/j.jpdc.2020.03.003

[5] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (June 2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[6] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear Road: A Stream Data Management Benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(Vldb '04)*. VLDB Endowment, 480–491. http://dl.acm.org/citation.cfm?id=1316689.1316732

[7] Arthur J. Bernstein and et al. 1999. Concurrency Control for Step-decomposed Transactions. *Inf. Syst. 1999* 24, 9 (Dec. 1999), 673–698. http://dl.acm.org/citation.cfm?id=337919.337922

[8] Irina Botan, Younggoo Cho, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Kihong Kim, Chulwon Lee, Girish Mundada, Ming-Chien Shan, Nesime Tatbul, Ying Yan, Beomjin Yun, and Jin Zhang. 2009. Design and Implementation of the MaxStream Federated Stream Processing Architecture. (07 2009). https://doi.org/10.1007/978-3-642-14559-9_2

[9] Irina Botan, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul. 2012. Transactional Stream Processing. In *Proceedings of the 15th International Conference on Extending Database Technology* (Berlin, Germany) *(Edbt '12)*. Acm, New York, NY, USA, 204–215. https://doi.org/10.1145/2247596.2247622

[10] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015). http://flink.apache.org/

[11] Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, John Meehan, Andrew Pavlo, Michael Stonebraker, Erik Sutherland, Nesime Tatbul, Kristin Tufte, Hao Wang, and Stanley Zdonik. 2014. S-Store: A Streaming NewSQL System for Big Velocity Applications. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1633–1636. https://doi.org/10.14778/2733004.2733048

[12] Neil Conway. 2008. Cisc 499*: Transactions and data stream processing. *Apr* 6 (2008), 28.

[13] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: a decision support benchmark for workload-driven and traditional database systems. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3376–3388.

[14] Anja Feldmann, Ming-Yang Kao, Jiří Sgall, and Shang-Hua Teng. 1993. Optimal Online Scheduling of Parallel Jobs with Dependencies. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing* (San Diego, California, USA) *(STOC '93)*. Association for Computing Machinery, New York, NY, USA, 642–651. https://doi.org/10.1145/167088.167254

[15] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.

[16] STREAM Group et al. 2003. *STREAM: The Stanford stream data manager.* Technical Report. Stanford InfoLab.

[17] Leslie A Hall, Andreas S Schulz, David B Shmoys, and Joel Wein. 1997. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of operations research* 22, 3 (1997), 513–544.

[18] Edwin SH Hou, Nirwan Ansari, and Hong Ren. 1994. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed systems* 5, 2 (1994), 113–120.

[19] Y.-K. Kwok and I. Ahmad. 1998. Benchmarking the task graph scheduling algorithms. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing.* 531–537. https://doi.org/10.1109/IPPS.1998.669967

[20] Yu-Kwong Kwok and Ishfaq Ahmad. 1999. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)* 31, 4 (1999), 406–471.

[21] John Meehan, Cansu Aslantas, Stan Zdonik, Nesime Tatbul, and Jiang Du. 2017. Data Ingestion for the Connected World.. In *CIDR.*

[22] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. 2015. S-Store: Streaming Meets Transaction Processing. *Proc. VLDB Endow.* 8, 13 (Sept. 2015), 2134–2145. https://doi.org/10.14778/2831360.2831367

[23] Nikola S Nikolov and Alexandre Tarassov. 2006. Graph layering by promotion of nodes. *Discrete Applied Mathematics* 154, 5 (2006), 848–860.

[24] Yves Robert. 2011. *Task Graph Scheduling.* Springer US, Boston, MA, 2013–2025. https://doi.org/10.1007/978-0-387-09766-4_42

[25] Ozlem Ceren Sahin, Pinar Karagoz, and Nesime Tatbul. 2019. Streaming Event Detection in Microblogs: Balancing Accuracy and Performance. In *Web Engineering*, Maxim Bakaev, Flavius Frasincar, and In-Young Ko (Eds.). Springer International Publishing, Cham, 123–138.

[26] Dennis Shasha and et al. 1995. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. Database Syst. 1995* 20, 3 (Sept. 1995), 325–363. https://doi.org/10.1145/211414.211427

[27] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. 2019. Briskstream: Scaling Data Stream Processing on Multicore Architectures. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, Amsterdam, Netherlands, 705–722. https://doi.org/10.1145/3299869.3300067

[28] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 Requirements of Real-time Stream Processing. *SIGMOD Rec.* 34, 4 (Dec. 2005), 42–47. https://doi.org/10.1145/1107499.1107504

[29] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. [n.d.]. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proc VLDB Endow. 2007.*

[30] Jun Tan and Ming Zhong. 2014-05. An Online Bidding System (OBS) under Price Match Mechanism for Commercial Procurement. *Applied Mechanics and Materials* 556-562 (2014-05), 6540–6543. https://doi.org/10.4028/www.scientific.net/AMM.556-562.6540

[31] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data.* 147–156. http://storm.apache.org/

[32] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003), 555–568.

[33] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003-03. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Trans. on Knowl. and Data Eng.* 15, 3 (2003-03), 555–568. https://doi.org/10.1109/tkde.2003.1198390

[34] Di Wang, Elke A. Rundensteiner, and Richard T. Ellison, III. 2011. Active Complex Event Processing over Event Streams. *Proc. VLDB Endow.* 4, 10 (July 2011), 634–645. https://doi.org/10.14778/2021017.2021021

[35] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Elsevier.

[36] Yingjun Wu, Wentian Guo, Chee-Yong Chan, and Kian-Lee Tan. 2017. Fast Failure Recovery for Main-Memory DBMSs on Multicores. In *SIGMOD '17* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 267–281. https://doi.org/10.1145/3035918.3064011

[37] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. 2014. T-Storm: Traffic-Aware Online Scheduling in Storm. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems (Icdcs '14)*. IEEE Computer Society, Washington, DC, USA, 535–544. https://doi.org/10.1109/icdcs.2014.61

[38] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. 2016. Exploiting Single-Threaded Model in Multi-Core In-Memory Systems. *IEEE Transactions on Knowledge and Data Engineering* 28, 10 (2016), 2635–2650. https://doi.org/10.1109/TKDE.2016.2578319

[39] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles.* 423–438.

[40] Shuhao Zhang, Yingjun Wu, Feng Zhang, and Bingsheng He. 2020. Towards concurrent stateful stream processing on multicore processors. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1537–1548.