Online Analytics with Local Operator Rebinding for Simulation Data Stream Processing

Xiaorui Du*, Andrea Piccione†, Adriano Pimpini‡, Stefano Bortoli†, Alessandro Pellegrini§, and Alois Knoll*
*Technical University of Munich
†Huawei Munich Research Center
‡Sapienza, University of Rome
§Tor Vergata University of Rome

***Abstract*—Leveraging multiple threads to process high volumes of simulation data is a prevalent strategy in modern streaming data processing systems. Statically binding operators to specific threads is the most common design employed due to its simplicity in implementation and initial system configuration. However, this approach often fails to effectively account for the inherently dynamic nature of simulation data, potentially leading to inefficient resource utilisation and processing bottlenecks. To address these limitations, we present a novel mechanism for stream-processing operator rebinding that enables lock-free, dynamic workload rebalancing between worker threads. The rebinding is driven by an autonomic policy that captures workload imbalance in the stream-processing pipeline when multiple queries are computed and reacts to it by moving computation around the different threads. We evaluate our proposal using data generated from large-scale traffic simulations on which multiple queries are executed. The volume and organisation of the data we feed to the stream-processing pipeline significantly change over time, providing excellent grounds to evaluate our rebinding policy. The evaluation confirms that the performance of stream processing pipelines can be greatly improved using local operator rebinding.**

***Index Terms*—Stream processing, simulation data processing, non-blocking algorithms**

I. Introduction

Simulation is essential in various fields and is now considered the third pillar of science, alongside theoretical and empirical methods. It has become highly complex, involving extensive models and numerous computationally intensive experiments. Modern simulations can achieve remarkable accuracy, allowing for practical what-if analyses and exploring alternative scenarios through multiple concurrent simulations. However, the increased level of detail in simulations and the scale enabled by distributed parallel computing pose a significant challenge for analysing the results.

The research community is aware that traditional approaches based on aggregating all generated data for post-processing are not viable [1], mainly due to I/O bottlenecks related to storing large amounts of data [2]. Conversely, processing simulation results using stream processing has been shown to be a viable solution [3].

Nevertheless, this approach shifts the performance focus to the stream processing pipeline. If this pipeline cannot cope with the amount of data generated by high-performance simulations, (soft) real-time requirements in the data processing may be violated. This aspect is quite relevant because simulations used for what-if analysis can be coupled with online interactive tools that allow analysts to execute different queries while the simulation is running, to explore specific parts of the simulation-generated data or to focus on alternating parts of the simulated world [4]. At the same time, stream processing applications are subject to changing conditions and fluctuations (e.g., workload, input rates, and environment) at runtime [5] as a result of natural emergent complexity in the simulation execution. Configurations that cannot cope with some quality of services can become suboptimal very quickly [6].

Recent works have tackled this problem by proposing strategies to pre-process part of the data in the simulation engine (the so-called *in-situ* computation, see, e.g., [7]) or relying on a set of distributed nodes (*in-transit* computation, see, e.g., [8]), or a mixture of the two (see, e.g., [9], [10]). These computation paradigms try to cope with the burden of the processing task by selecting the best-suited location to carry out part of the computation and dynamically adapting to changes in the workload [10]. In general, this is a specific instance of *self-adapting computation* [11], a form of *autonomic computing* [12], where the system changes its behaviour in response to input rate and data volumes fluctuations following non-stationary distributions across stream partitions. In stream processing, a self-adaptive computation may change the batch size or the parallelism degree to reach a certain goal of performance or energy efficiency [13].

From the point of view of computing resources, we can now exploit exascale architectures [14], capable of executing in parallel massive computations relying on large amounts of cores available. If orchestrated effectively, this increased processing capability can support the analysis of large-scale simulation data without the need to rely on complex distributed systems, thus possibly also reducing the energy footprint of the pipeline. Single-node stream processing has received limited attention, primarily due to perceived limitations in processing power and memory constraints, but is being recently considered a viable solution [15], [16].

In this paper, we follow this recent path and propose a lock-free [17] operator rebinding mechanism, coupled with an innovative operator rebinding policy, that can effectively sustain the computational complexity of executing multiple stream-processing queries on the data produced by large-scale simulations. Lock-freedom in operator rebinding offers several benefits. When the load on the different operators is detected to have changed significantly, it allows to react to this change without having to stop the current processing activities, resulting in better utilisation of system resources and improved overall throughput. Additionally, it can reduce latency and improve the system's responsiveness, making it well-suited for real-time or near-real-time processing requirements.

We exercised our methodology using various configurations of an agent-based simulation model on top of the CityMoS simulation framework [18], a high-performance microscopic traffic simulator. We chose to focus on microscopic traffic simulation for this study because 1) it allows to model synthetic scenarios that generate both large amounts of data, 2) emergent complexity of chaotic traffic models naturally produce unforeseeable and non-stationary fluctuations, and 3) it is straightforward to support multiple data analytics queries computing continuously traffic metrics with different degrees of complexity. Our experimental evaluation demonstrates that the self-adaptive orchestration of data analytics activities can significantly reduce the time it takes to complete joint simulation and analytics activities.

II. Related Work

In general, operator placement aims to distribute query processing over network nodes to meet system goals effectively, but finding the optimal placement is NP-hard [19]. Heuristics are often employed to achieve near-optimal solutions. A lot of work has been done by the research community in this direction. We refer the reader to [13], [20]–[22] for a comprehensive discussion. Here, we focus on relevant contributions related to two orthogonal aspects we deal with in this paper: how to implement operator migration effectively and how to decide when/what to migrate.

The simplest migration method involves moving the operator to a new host and replaying necessary historical tuples from upstream nodes [23]–[25]. In [26], both the old and new hosts receive the same tuples during migration, allowing a gradual handover and minimising operator downtime. *State shedding* [27] migrates only the most essential partial states and drops less important ones if the total state is too large. Streams can also be redirected from upstream nodes to the new host, which buffers the streams and starts processing them once the state from the old host is received and installed [28], [29]. Other methods send control messages to the old or new host [30], [31]. In [32]–[34], to reduce the migration time, an operator outputs to multiple replicas partitioned by keys allowing to change key distribution. Some proposals extend this approach by avoiding any pause phase during state movement [35], [36].

Checkpoint-assisted algorithms [28], [37]–[41] require sending only minimal state during migration. Priorities on migrating partial states can be introduced to improve responsiveness [42]. Distributed checkpoint replication and incremental checkpointing to speed up state migration are also possible [28], [40].

Compared to all these proposals, we allow for a non-blocking rebinding of the operators. This is thanks to the reliance on a single fat node, which copes well with modern high-performance architectures. Our approach benefits from zero migration downtime; no network copy of the tuples is required, and the balancing of the stream processing workload can be extremely responsive.

Another problem related to operator migration is when and what operators to migrate. These two aspects have a significant performance effect on the overall stream processing system. Migrating too often too many operators may incur additional secondary costs that could jeopardise any expected performance improvement. Three common methods to avoid widespread migrations involve setting a threshold to ensure the new placement's score is significantly better than the current one, triggering migration only when QoS guarantees are violated, and periodically re-evaluating the objective function at high intervals [43]. Interference scores have been used to determine migration needs, moving operators to nodes with less interference [44], though explicit cost-benefit analyses were not performed. Migration mechanisms can be centralized, relying on a coordinator [28], [29], [45], or decentralized, initiated by the operator host [30], [39]. Multiple dependent migrations can be planned and executed sequentially, such as load balancing, where many keys of an operator are moved, and geographically distributed operator graphs, where several operators are migrated [46].

Some works solve the migration problem using Integer Linear Programming (ILP) solvers [19], [47], [48], but these may not scale well as they require a global network view. Operator scaling, however, aims to minimise the number of operator instances while maintaining QoS guarantees [49]. Stream-based overlay networks [50] use heuristic algorithms to minimise network usage by placing nodes in a cost space with latency and load dimensions, where node distances represent communication latency. Heuristic algorithms for load balancing [44] aim to reduce interference impacting stream processing performance by predicting future resource usage and migrating computations to nodes with less interference if a threshold is exceeded. Load balancing can also be achieved using heuristic partitioning algorithms [51], which ensure balanced key-to-server mapping through different approaches. Elastic stream processing platforms [52] use heuristics with predefined thresholds to scale resources up or down based on CPU utilisation.

Accurately defining migration costs, manifested as increased resource consumption or execution degradation, is crucial for correct migration decisions. Most solutions measure migration cost using specific metrics rather than modelling it, focusing on tuple processing performance to model and measure benefits. Few approaches calculate migration costs, but one model predicts tuple latency by considering input rate, migration time, and time before queued events can be processed [53]. The state size is often equated with migration time, typically modelled without detailed calculations, with some solutions migrating multiple operators simultaneously and defining migration time as the maximum time required [19], [47]. In data center-based solutions, operator downtime depends on adaptation type, state size, and round-trip delay [19], while WASP uses state size and link bandwidth to define migration time in wide area networks, solving a min-max problem to minimise the slowest migration [47]. Cost models explicitly defining migration time or cost are also used [26], [54], [55]. State size frequently serves as an objective function in migration decisions [31], [51] and as a constraint to prevent costly solutions [56], sometimes being the sole criterion for load balancing [57], [58]. Control message numbers during migration, affecting total migration time, are crucial [46]. Defining migration cost in terms of time often overlooks other factors, potentially skewing cost assessments. Monetary migration costs distinguish between running the current topology and making changes [59], [60]. In decentralised fog and edge computing, metrics like network usage, bandwidth, and link latency are vital, with overlay networks optimising these for placement and migration decisions [50]. Similar methods reduce the number of migrations to achieve optimal placement [61].

A methodology to reduce interference between stream processing operators uses an interference score based on predicted future packet load [44]. WASP relies on an operator's expected input and output rates [47]. Latency constraints determine migration needs, with systems scaling out or load balancing if constraints are not met, and lower latency constraints increase operational costs [62]. Load balancing solutions use load or tuple performance, minimising variance between nodes, which may require expensive migrations [51]. Resource usage metrics include threads assigned to operators [63] or the CPU queue state [64]. Elasticity-based solutions adjust resources based on workload, scaling out if overloaded [65]. In decentralised fog-based solutions, a cost space model constructs topology based on latency, bandwidth, and load [50]. Many studies minimise migration costs by reducing the number, frequency, or magnitude of migrations [54], [66]. Single-objective optimization [47], [51], [58], [67] or simple additive weighting with multiple objectives [19], [31], [68]–[70] are used, with constraints ensuring acceptable placement quality. Hotspot alleviation reduces migrations while maintaining QoS [71], and relaxation algorithms reduce migrations before optimal placement [61]. Thresholds ensure beneficial migrations triggered by load imbalance or migration benefits exceeding costs [50]. QoS violation adaptability is predicted using linear regression [71] and predictive latency models [65]. Gaussian processes estimate load and latency for scaling decisions [59], and extended Gaussian Processes upper confidence bound algorithm models service capacity [72]. Real-time resource usage prediction employs incremental learning techniques [73], while MPC predicts optimal scaling decisions [36]. Reactive and proactive scaling decisions are used in the Elysium system [54], with reinforcement learning determining scaling timing [69], and operator load predictions considering tuples during a prediction horizon [74]. WASP estimates expected rates to assess load [47]. Combined migration strategies use window state knowledge to schedule minimal state migrations [75], and the Phoebe system predicts workloads for near-optimal scaling using multiple regression and clustering [76]. Finally, also energy consumption for resource-constrained systems can be considered to initiate a migration [77], [78].

We show that very simple policies (e.g., based on greedy algorithms) can deliver good query throughput for single-node stream processing jobs, ensuring timely online simulation data processing.

III. Operator Rebinding

At first glance, a *query* is the unit of computation in a stream processing engine. Nonetheless, queries are not monolithic, opaque pieces of computation; rather, they are described with the aid of some formalism. For instance, employing common Streaming SQL concepts [79], queries can be defined by combining selections, projections, joins, and window aggregation of tuples. More generally, a query describes a computational tree, where leaves are *data sources*, the root is the query output, and non-leaf vertexes are the *operators* that compose the query.

A streaming engine is expected to cope with multiple queries being executed simultaneously. As an optimisation, or with explicit configurations from the user, query trees that share one or more operators may be partially merged. In general, even when shared operators are merged together, the queries submitted to a streaming engine implicitly define a global Directed Acyclic Graph (DAG), which we will refer to as *computation graph*. Therefore, at a high level, the operation of a streaming engine consists of evaluating the computation graph whenever fresh data is available at one of the data sources.

Triggering a computation every time a new tuple is ingested would be severely inefficient; streaming engines commonly employ buffering or tuple batching to achieve good throughput. Indeed, in most implementations, once enough tuples have been collected, a new computational task containing them is spawned and scheduled into the system. Increasing the size of the tuple buffer reduces the scheduling overhead per tuple but, conversely, increases latency. As a result, especially in low-latency scenarios, task scheduling can be a system bottleneck.

In addition, to efficiently use multi-core machines, it is necessary to partition the workload across the available computational resources. In our previous work [80], we deal with large simulation sources that generate considerable amounts of data at each timestep of virtual time. An effective parallelisation strategy in that scenario involves buffering all the tuples for a given timestep and evenly partitioning them across the processing threads. Then, the computation graph can be traversed synchronously by all the threads in a coordinated fashion. This strategy achieves great throughput by exploiting the *data parallelism* of large simulations, but it leads to higher latencies because query computations have to wait for the completion of previous simulation timesteps.

Most commonly, though, streaming engines deal with bursty tuple generation patterns, as could be the case when simulation output is used in interactive applications. For example, users may play with simulation pacing and remove or insert new queries relatively often [81]. In such cases, it is possible to exploit the *task parallelism* exposed by the computation graph, by scheduling computational tasks at the granularity of each vertex, allowing operators to process their incoming tuples concurrently and asynchronously.

The most straightforward way to implement this concurrency scheme is to use a global monolithic scheduler, which can be designed as a priority queue of tasks ordered by their simulation virtual time. The literature offers fairly efficient non-blocking implementations that eliminate the need for mutexes [82]–[84]. However, as we will demonstrate experimentally, this approach is not optimal for interactive applications, even when state-of-the-art data structures are used.

The alternative approach (see, e.g., [85]), which involves having a separate task queue for each thread, can provide better throughput on the same hardware and experience less unpredictability in latency. The multi-task queue design, while mitigating the single queue bottleneck, incurs additional implementation complexities and opens up new problems, such as deciding which tasks are scheduled on which queues. A partial solution to the problem consists of binding operators to processing threads, i.e., scheduling computational tasks meant to be processed by a certain operator i always to a certain thread j, according to some stable mapping.

A. Problem Statement

As mentioned earlier, a DAG can capture the dependencies between multiple operators across various queries. Let $G = (V, E)$, where V is a set of vertices, each $V_i \in V$ representing an operator, and E is the set of directed edges representing the flow of data streams between operators. Let $T = \{T_1, T_2, \dots, T_m\}$ be the set of available threads in a multi-core system.

As a baseline, we consider static binding, where the system maps operators to threads using a function $M_{\text{static}} : V \to T$ and maintains this mapping throughout the entire lifetime of the stream processing application. Static binding is straightforward and efficient regarding setup time and overhead, but heuristic algorithms are necessary since computing the optimal binding is generally an NP-hard problem [19]. In addition, the binding quality depends on the workload exerted by each operator, which is unknown at startup and may change over time. This dynamic aspect emerges from several factors, such as varying load on the machine that runs the streaming engine or the non-linearity of computational requirements of operators with respect to input data rates. This can be particularly relevant when processing the output of large-scale simulations, which can produce terabytes of data [86].

For these reasons, in this paper, we consider dynamic binding, where the system applies an initial operator-to-thread mapping $M_0 : V \to T$, then periodically assesses the computational cost of the operators and applies new mappings $M_i : V \to T$ at runtime to adapt to the changing conditions. Especially when simulations are used as data sources, large workload variance is expected; as a result, achieving decent performance requires performing load balancing of operators fairly frequently. At the same time, we consider scenarios where machines with many cores are utilised to carry out the stream processing activities. Therefore, the dynamic binding policy that we study is specifically tailored to this particular setup.

B. Operator rebinding policy

The dynamic operator rebinding problem is often tackled using heuristic policies based on greedy algorithms. These algorithms are known for their ability to find satisfactory solutions within a reasonable amount of time and computational resources [87].

To introduce the proposed greedy policy, we define the following quantities. First, $tuple\_count$ represents the number of pending tuples an operator must process, defined as:

$$tuple\_count(i) = tuple\_rcv(i) - tuple\_proc(i), \quad (1)$$

where i is the id of an operator, and $tuple\_rcv$ and $tuple\_proc$ are the total number of tuples that operator i has received and processed, respectively.

Second, op_comp is the average time that an operator takes to process the last w tuples, defined as:

$$op\_comp(i) = \frac{\sum_{k=1}^{w} tuple\_comp(i,k)}{w}, \quad (2)$$

where i is the operator id, $tuple\_comp$ represents the time the operator i requires on average to process a tuple, while k represents the tuple id in the latest w tuples.

Third, wl is the workload of a thread, computed as:

$$wl(t) = \sum_{i \in sub\_op(t)} op\_comp(i) \cdot tuple\_count(i), \quad (3)$$

where t is the thread id, and sub_op represents the set of operators bound to thread t.

Finally, wl_avg, the average workload of all threads, is computed as:

$$wl\_avg = \frac{\sum_{t=1}^{thread\_num} wl(t)}{thread\_num} \quad (4)$$

where $thread\_num$ represents the total number of processing threads and $wl(t)$ is the thread workload as defined in Equation (3).

An important aspect related to operator rebinding lies in the amount of housekeeping work that is required to install a new binding. In particular, the rebinding operation must be transparent with respect to the stream processing queries that are being computed. That means that the result of the queries must be the same as if only the M_{static} binding was used, independently of the sequence $M_0, M_1, \dots, M_n$ that are installed during the lifetime of the stream processing application. If this condition is not met, we can consider the query result incorrect *because* of the dynamic binding.

Therefore, the system installing the selected thread-to-binding (which we shall discuss in Section III-C) must ensure that a batch of tuples currently being processed is either completed or replayed at the destination thread, that all buffered tuples are migrated to the destination thread, and that all incoming tuples are redirected to the destination thread. These housekeeping operations can incur significant costs, which may negate any short-term benefits of the new operator binding. If rebinding is computed frequently, as we are considering, the overall performance may degrade to an unacceptable level.

Therefore, we have designed our greedy rebinding policy to incorporate previous binding information and consider preferred thread workloads. We present in Algorithm 1 the pseudocode of the greedy policy to recompute the operator-to-thread binding.

The algorithm first copies the old binding $old\_op\_to\_thread$ to $new\_op\_to\_thread$, then computes op_wl according to Equation (3) and sorts it in ascending order (lines 3–9). The main while loop pops the operator with the smallest workload from op_wl and gets the thread id j it belongs to (lines 11 and 12). Then, we check the workload $wl[j]$; if $wl[j] > avg\_wl$, indicating that the thread is handling a workload larger than the average, we rebind the operator to another thread to achieve a better thread workload balance. The thread k, which has the least workload in wl, will be chosen for the rebinding. After updating the new operator binding in $new\_op\_to\_thread$, we update the current wl for threads k and j accordingly. Specifically, thread k, where the operator is bound to, will increase its workload, while thread j, where the operator is removed from, will decrease its workload (lines 13–17). However, if $wl[j] \leq avg\_wl$, it indicates that the current thread can handle the existing workload, and thus, there is no need to rebind the operator to a different thread.

Algorithm 1 Greedy

```
1:  Input: tuple_count, op_comp, wl, avg_wl, old_op_to_thread
2:  Output: new_op_to_thread
3:  new_op_to_thread ← old_op_to_thread
4:  op_wl ← []
5:  for $i = 0$ to $operator\_num - 1$ do
6:      load = op_comp[i] * tuple_count[i]
7:      op_wl.append(load)
8:  end for
9:  ascending_sort (op_w)
10: while op_wl is not empty do
11:     (i, workload) ← pop first element from op_wl
12:     j ← old_op_to_thread[i]
13:     if wl[j] > avg_wl then
14:         k ← index of least workload in wl
15:         new_op_to_thread[i] = k
16:         wl[k] += workload
17:         wl[j] -= workload
18:     end if
19: end while
```

C. Non-blocking Rebinding

From the implementation's point of view, operator processing can be decoupled from the binding decision-making; in other words, we can assume that an external process periodically evaluates the streaming engine's state and requests that new operator bindings be installed. Installing a new binding clearly requires updating the global operator mapping. In addition, the processing threads need to move the tasks contained in their local scheduler according to the new binding.

To guarantee the correctness of this process, processing threads need to synchronise their rebinding activities. Otherwise, critical data races may cause tasks to be executed out of order or operators to be concurrently activated by multiple threads. For example, let us analyse a system configuration with two processing threads T_0, T_1, and a computation graph $O_0 \rightarrow O_1 \rightarrow O_2$. Let us assume a binding mapping that assigns O_0 to T_0, and O_1, O_2 to T_1. Now, let us assume that T_1 is engaged in a very long computation with operator O_1, but concurrently, a new binding that maps O_1 to T_0 is requested. While processing O_0, T_0 may produce some new tasks for operator O_1 that, according to the new binding, would end up in T_0's queue. T_0 would then try to access and update concurrently O_1, resulting in a race condition.

We note that locking operators is not sufficient to avoid consistency issues. Even if T_1 is busy with a different task during rebinding, T_0 could still update O_1 by executing a task with a later timestamp than another one possibly still present in T_1's queue, resulting in tasks being executed out of order.

The simplest way to avoid these concurrency problems relies on explicit barrier synchronisation. With this approach, whenever a new binding needs to be installed, a global flag is set. Each processing thread periodically checks this flag, and if it is set it waits on a thread barrier; when all the processing threads reach the barrier, rebinding operations can be initiated without fearing concurrent updates of operators. Finally, all threads commit the newly installed binding by waiting on a second thread barrier to avoid resuming operator processing before all threads complete their own rebinding operations.

However, the drawback of this synchronous approach is that processing threads must wait for the slowest one before proceeding with rebinding and before resuming processing after the new binding is applied. We propose an alternative approach that does not require explicit barrier synchronisation.

In our non-blocking implementation, the global binding mapping is maintained in an array of atomic integer variables in which each entry represents the thread identifier to which a specific operator is bound. To apply a new rebinding, the requester process reads the entries $o_0, o_1, \ldots o_n$ of the old binding, comparing them to the new binding entries $n_0, n_1, \ldots n_n$. For every i where $o_i \neq n_i$, the requester schedules a special operator rebind task with the highest priority destined to thread o_i. When this task is processed, thread o_i updates the i-th entry of the global mapping array

TABLE I: Analytics queries—n is the number of agents.

Query	Description
Agent count (Q1)	Read the pre-computed agent number per road. Computational complexity: $\Theta(1)$.
Average speed (Q2)	Average speed of agents. Computational complexity: $\Theta(n)$.
Top K speed (Q3)	Get the speed of the top k fastest agents. Computational complexity: $\Theta(n \log(n))$.

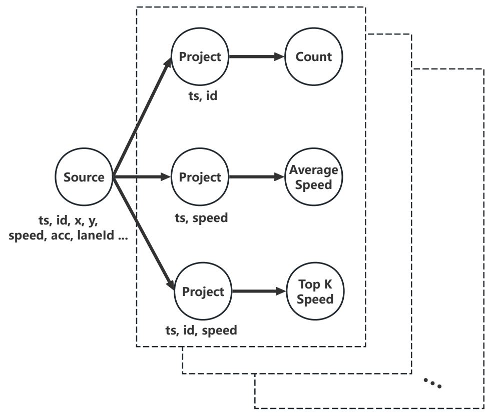


Fig. 1: DAG for 150 queries.

to the value n_i using an atomic store operation.

Clearly, leftover tasks for migrated operators may still be present in the scheduling queues. This case is handled lazily by checking each time a task is extracted whether it is intended for the extracting thread. If not, the task is scheduled to the correct thread; otherwise, processing continues as usual. Finally, to address out-of-order task processing issues, we borrow the idea of TCP sequence numbers: the generating operators assign tasks a monotonically increasing sequence number. Similarly, operators have a receiver sequence number that is increased every time a task is successfully processed. Upon extraction, tasks with higher epoch numbers than their receiver operator are rescheduled so that other tasks may be processed.

With this precaution, the rebinding operation effectively becomes an asynchronous process, where threads simultaneously carry out migrations and operators' processing. In the described system, processing threads never need to block or wait explicitly for the completion of other operations. Nonetheless, we note that the actual progress of streaming engine operations ultimately relies on rebindings to eventually complete task processing.

IV. Experimental Assessment

Similarly to [10], we employ three queries in the experimental assessment; they are listed in Table I. Each query is replicated 50 times, resulting in a DAG composed of 300 operators, as shown in Figure 1. The front dashed box describes the operator representations of three queries.

We use the average latency of queries and query latency distribution for performance evaluation. The average latency is computed as:

$$q\_latency\_avg(t) = \frac{\sum_{i=1}^{q\_num} q\_latency(i,t)}{q\_num}, \quad (5)$$

where $q\_latency(i,t)$ represents the latency of query i after processing all tuples at time t, and q_num represents the total number of queries.

As mentioned, we rely on the CityMoS simulation framework [18] to generate the data processed by the selected queries. We conducted two different experiments to assess the two different proposals set out in this paper. First, we study the non-blocking rebinding when using our greedy policy. This experiment allows us to determine whether non-blocking rebinding can be competitive compared to more classical methods to install a binding. Then, we study whether our greedy policy can outperform classical strategies when complemented with the discussed non-blocking rebinding.

We have run our experiments with Ubuntu 20.04.6 LTS on a machine equipped with Intel Xeon E5-2680 v3 @2.50 GHz CPUs with 12 physical/24 logical cores and 256 GB RAM. Each data point is averaged over 5 runs.

To study the impact of the non-blocking rebinding scheme that we have proposed, we use a simulation scenario with a constant workload, where 120,000 agents travel across Shenzhen, China generated using [88]. For each agent, a tuple $\langle ts, id, x, y, speed, acc, laneId\rangle$ is emitted 25 times in a second of wall clock time and delivered to all 150 queries in the stream processing system. Our greedy rebinding policy is re-evaluated every 2 seconds of wall-clock time to mimic a high-intensity scenario.

Figure 2 depicts the performance results of our experiment, evaluating the average latency of the queries while the simulation data are fed into the stream processing pipeline. The results show that the non-blocking strategy has a consistently lower latency than the blocking strategy. This is expected because, with the blocking rebinding, threads implement a "pause & resume" strategy, amplifying the housekeeping operation's cost. In contrast, the non-blocking design permits each thread to handle operator rebinding independently and on the fly, eliminating the need for thread synchronisation.

To study the behaviour of our rebinding policy, we have compared its performance against a static binding and a multi-threaded scenario in which a single queue is used to extract tuples for the operators to process. The static binding assigns operators to threads using a round-robin assignment at startup. No rebinding is carried out in this scenario. In the single queue scenario, all tuples to process are kept in an Intel TBB concurrent priority queue [89]. No binding policy is applied: threads continuously check the queue and extract tuples to process. In this case, we have also used a traffic model running constantly 220,000 agents in CityMoS, limiting the workload variations to the natural emergent complexity of traffic. This is done with the intention of finding the limits of the processing capacity of the system to simplify the study of

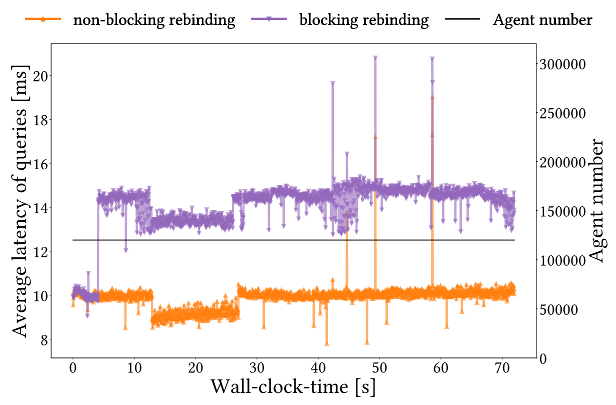
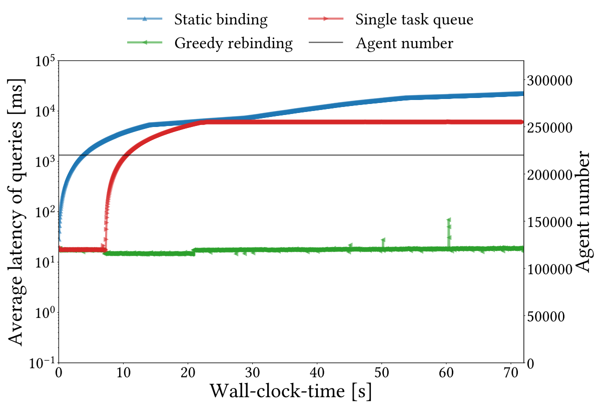
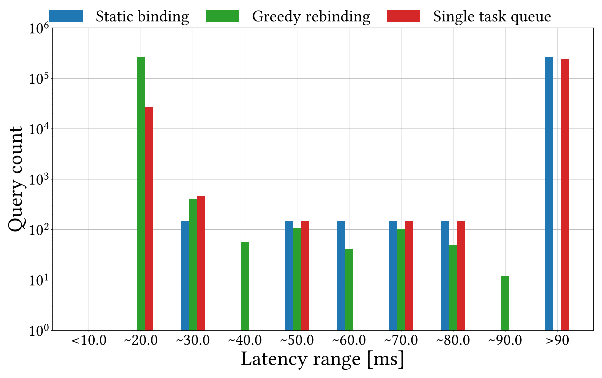


Fig. 2: Non-blocking vs blocking operator rebinding



(a) Average latency of queries



(b) Query latency distribution

Fig. 3: CityMos constant workload (220,000 agents)

the rebinding problem without further considering also natural non-stationarity of traffic distributions in time.

Figure 3a reports the results of this experiment. As can be seen, the average latency increases dramatically with the static operator binding, and eventually it reaches a latency that is unacceptable for real-time data processing. Upon investigation, we discovered that with the static binding, only four threads, on average, actively process tuples, while most remain idle for most of the runtime. This is also expected, as the static policy did not account for each operator's complexity and varying workload, leading to workload imbalances between threads at runtime even with a constant number of agents being processed by the simulator. When the system reaches the processing capacity limits, static binding performance degenerates quickly, highlighting its fragility in complex setups. Our greedy policy consistently maintains a stable and low processing latency finding a balance between static and unnecessary operator rebindings. The single queue design performs well initially, maintaining a low query latency ($< 20ms$) for the first 9 seconds. However, as concurrency costs become dominant, the latency subsequently increases. Figure 3b displays the distribution of query latency. Our greedy policy serves 99.7% of queries within a latency of less than $20ms$ throughout the entire run. Although low-latency queries are observed in other designs or policies, over 90% latency is greater than $90ms$.

To complete our experimental assessment, we studied the behaviour of the approaches considered when dealing with a varying simulation workload. We have configured CityMoS to simulate the road network of Shenzhen in China, simulating 4 hours (an interval representative of a regular morning of a working day) starting at 8:00 am and ending at noon. The maximum number of agents in the road network is 300,000.

The results of this experiment are reported in Figure 4a. From the results, we can see that as the workload increases, the average query latency rises accordingly for all configurations. However, our greedy policy consistently provides the lowest query latency and the latest stable query latency breaking point (after 67 seconds). Under this dynamic workload scenario, both the static assignment and single queue configurations cannot cope with the increased workload very soon.

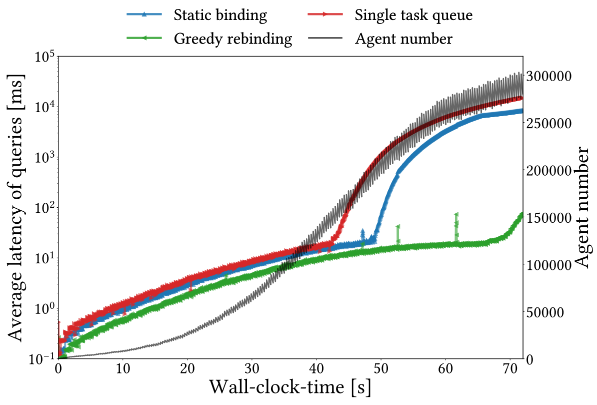
Figure 4b displays the latency distribution of all configurations under this dynamic workload. Our greedy policy consistently outperforms the other configurations across all low-latency ranges, with 57% of queries below 10ms and 96% below 30ms. The static binding and the single queue configurations have over 30% of queries exceeding 90ms latency.

V. Conclusions

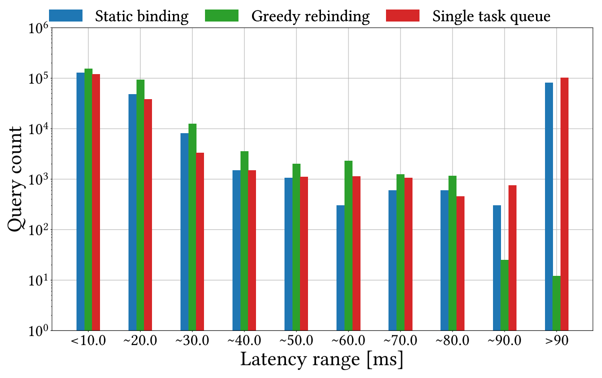
We have proposed a novel non-blocking operator rebinding mechanism to optimise stream processing pipelines for simulation data by dynamically redistributing workloads among worker threads on large-scale computing architectures. We evaluated our approach using data from large-scale traffic simulations, revealing substantial performance improvements. The results show that our dynamic workload management not only achieves lower latency and higher throughput but also ensures better resource utilisation than conventional methods, making it an effective solution for enhancing the efficiency and responsiveness of stream processing systems in near-real-time analytics.

Acknowledgement

This paper has been partially supported by the Italian MUR PRIN 2022 Project: Domain (Grant #2022TSYYKJ) financed by NextGenEu.



(a) Average latency of queries



(b) Query latency distribution

Fig. 4: CityMoS dynamic workload (up to 300,000 agents)

References

[1] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 421–434, Aug. 2015.

[2] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, no. Article 40. New York, NY, USA: Association for Computing Machinery, Nov. 2009, pp. 1–12.

[3] B. Ellis, *Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data*. Hoboken, NJ, USA: John Wiley & Sons, Jun. 2014.

[4] H. J. Yoon and W. Shen, "Simulation-based real-time decision making for manufacturing automation systems: a review," *International journal of manufacturing technology and management*, vol. 8, no. 1/2/3, p. 188, 2006.

[5] H. C. M. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of stream processing: Application design, systems, and analytics*. Cambridge, England: Cambridge University Press, Feb. 2014.

[6] A. Vogel, G. Mencagli, D. Griebler, M. Danelutto, and L. G. Fernandes, "Towards on-the-fly self-adaptation of stream parallel patterns," in *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, Mar. 2021.

[7] K.-L. Ma, C. Wang, H. Yu, and A. Tikhonova, "In-situ processing and visualization for ultrascale simulations," *Journal of physics. Conference series*, vol. 78, no. 1, p. 012043, Jul. 2007.

[8] A. R. Zamani, D. Balouek-Thomert, J. J. Villalobos, I. Rodero, and M. Parashar, "An edge-aware autonomic runtime for data streaming and in-transit processing," *Future generations computer systems: FGCS*, vol. 110, pp. 107–118, Sep. 2020.

[9] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Piscataway, NJ, USA: IEEE, Nov. 2012, pp. 1–9.

[10] X. Du, Z. Meng, A. Siguenza-Torres, A. Knoll, A. Pimpini, A. Piccione, S. Bortoli, and A. Pellegrini, "Autonomic orchestration of in-situ and in-transit data analytics for simulation studies," in *2023 Winter Simulation Conference (WSC)*. IEEE, Dec. 2023.

[11] S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio, "Control-theoretical software adaptation: A systematic literature review," *IEEE transactions on software engineering*, vol. 44, no. 8, pp. 784–810, Aug. 2018.

[12] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[13] A. Vogel, D. Griebler, M. Danelutto, and L. G. Fernandes, "Self-adaptation on parallel stream processing: A systematic review," *Concurrency and computation: practice & experience*, vol. 34, no. 6, p. e6759, Mar. 2022.

[14] J. Dongarra, S. Gottlieb, and W. T. C. Kramer, "Race to exascale," *Computing in science & engineering*, vol. 21, no. 1, pp. 4–5, Jan. 2019.

[15] S. Zeuch, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl, "Analyzing efficient stream processing on modern hardware," *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, vol. 12, no. 5, pp. 516–530, Jan. 2019.

[16] G. Theodorakis, F. Kounelis, P. Pietzuch, and H. Pirk, "Scabbard: single-node fault-tolerant stream processing," *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, vol. 15, no. 2, pp. 361–374, Oct. 2021.

[17] M. Herlihy and N. Shavit, "On the nature of progress," in *Principles of Distributed Systems*, ser. Lecture Notes in Computer Science, A. Fernàndez Anta, G. Lipari, and M. Roy, Eds. Berlin Heidelberg, Germany: Springer International Publishing, 2011, vol. 7109, pp. 313–328.

[18] D. Zehe, S. Nair, A. Knoll, and D. Eckhoff, "Towards citymos: a coupled city-scale mobility simulation framework," in *Proceedings of the 5th GI/ITG KuVS Fachgespräch Inter-Vehicle Communication*, ser. Technical Reports, A. Djanatliev, K.-S. Hielscher, and R. German, Eds. Nuremberg, Germany: Friedrich-Alexander-Universität Erlangen-Nürnberg, 2017, vol. CS-2017-03, pp. 26–28.

[19] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator replication and placement for distributed stream processing systems," *Performance evaluation review*, vol. 44, no. 4, pp. 11–22, May 2017.

[20] D. Foroni, C. Axenie, S. Bortoli, M. Al Hajj Hassan, R. Acker, R. Tudoran, G. Brasche, and Y. Velegrakis, "Moira: A goal-oriented incremental machine learning approach to dynamic resource cost estimation in distributed stream processing systems," in *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*. New York, NY, USA: ACM, Aug. 2018.

[21] C. Qin, H. Eichelberger, and K. Schmid, "Enactment of adaptation in data stream processing with latency implications—a systematic literature review," *Information and software technology*, vol. 111, pp. 1–21, Jul. 2019.

[22] E. Volnes, T. Plagemann, and V. Goebel, "To migrate or not to migrate: An analysis of operator migration in distributed stream processing," *IEEE Communications Surveys & Tutorials*, pp. 1–1, 2023.

[23] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, and M. Völz, "Rollback-recovery without checkpoints in distributed event processing systems," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*. New York, NY, USA: ACM, Jun. 2013.

[24] N. Garg, *Apache Kafka*. Birmingham, England: Packt Publishing, Oct. 2013.

[25] P. Carbone, A. Katsifomodis, S. Ewen, V. Markl, and S. Haridi, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, pp. 28–38, 2015.

[26] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman, "Dynamic plan migration for continuous queries over data streams," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, Jun. 2004.

[27] E. Volnes, T. Plagemann, B. Koldehofe, and V. Goebel, "Travel light: state shedding for efficient operator migration," in *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*. New York, NY, USA: ACM, Jun. 2022.

[28] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl, “Rhino: Efficient management of very large distributed state for stream processing engines,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, Jun. 2020.

[29] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, “StreamCloud: An elastic and scalable data streaming system,” *IEEE transactions on parallel and distributed systems: a publication of the IEEE Computer Society*, vol. 23, no. 12, pp. 2351–2365, Dec. 2012.

[30] T. Repantis and V. Kalogeraki, “Alleviating hot-spots in peer-to-peer stream processing environments,” in *Proceedings of the 5th International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, ser. DBISP2P, Vienna, Austria, 2007.

[31] W. Hummer, P. Leitner, B. Satzger, and S. Dustdar, “Dynamic migration of processing elements for optimized query execution in event-based systems,” in *On the Move to Meaningful Internet Systems: OTM 2011*, ser. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 451–468.

[32] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, “Elastic scaling for data stream processing,” *IEEE transactions on parallel and distributed systems: a publication of the IEEE Computer Society*, vol. 25, no. 6, pp. 1447–1463, Jun. 2014.

[33] V. Cardellini, M. Nardelli, and D. Luzi, “Elastic stateful stream processing in storm,” in *2016 International Conference on High Performance Computing & Simulation (HPCS)*, ser. HPCSIM’16. Piscataway, NJ, USA: IEEE, Jul. 2016, pp. 583–590.

[34] J. Li, C. Pu, Y. Chen, D. Gmach, and D. Milojicic, “Enabling elastic stream processing in shared clusters,” in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, Jun. 2016.

[35] T. De Matteis and G. Mencagli, “Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing,” in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’16. New York, NY, USA: ACM, Feb. 2016, pp. 1–12.

[36] ——, “Proactive elasticity and energy awareness in data stream processing,” *The Journal of systems and software*, vol. 127, pp. 302–319, May 2017.

[37] K. G. S. Madsen and Y. Zhou, “Dynamic resource management in a massively parallel stream processing engine,” in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. New York, NY, USA: ACM, Oct. 2015.

[38] W. Hummer, B. Satzger, and S. Dustdar, “Elastic stream processing in the cloud: Elastic stream processing in the cloud,” *Wiley interdisciplinary reviews. Data mining and knowledge discovery*, vol. 3, no. 5, pp. 333–345, Sep. 2013.

[39] B. Ottenwälder, B. Koldehofe, K. Rothermel, and U. Ramachandran, “MigCEP: operator migration for mobility driven distributed complex event processing,” in *Proceedings of the 7th ACM international conference on Distributed event-based systems*. New York, NY, USA: ACM, Jun. 2013.

[40] Y. Wu and K.-L. Tan, “ChronoStream: Elastic stateful stream computation in the cloud,” in *Proceedings of the 31st International Conference on Data Engineering*, ser. ICDE’15. IEEE, Apr. 2015.

[41] M. Hoffmann, A. Lattuada, and F. McSherry, “Megaphone: latency-conscious state migration for distributed streaming dataflows,” *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, vol. 12, no. 9, pp. 1002–1015, May 2019.

[42] R. Gu, H. Yin, W. Zhong, C. Yuan, and Y. Huang, “Meces: Latency-efficient rescaling via prioritized state migration for stateful distributed stream processing systems,” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 539–556.

[43] G. T. Lakshmanan, Y. Li, and R. Strom, “Placement strategies for internet-scale data stream systems,” *IEEE internet computing*, vol. 12, no. 6, pp. 50–60, Nov. 2008.

[44] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara, “Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams,” *IEEE transactions on parallel and distributed systems: a publication of the IEEE Computer Society*, vol. 28, no. 12, pp. 3553–3569, Dec. 2017.

[45] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, “Flux: an adaptive partitioning operator for continuous query systems,” in *Proceedings 19th International Conference on Data Engineering*, ser. DE ’04. Piscataway, NJ, USA: IEEE, 2004.

[46] M. Luthra, B. Koldehofe, P. Weisenburger, G. Salvaneschi, and R. Arif, “TCEP: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms,” in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. New York, NY, USA: ACM, Jun. 2018.

[47] A. Jonathan, A. Chandra, and J. Weissman, “WASP: Wide-area adaptive stream processing,” in *Proceedings of the 21st International Middleware Conference*. New York, NY, USA: ACM, Dec. 2020.

[48] R. Anand, D. Aggarwal, and V. Kumar, “A comparative analysis of optimization solvers,” *Journal of Statistics and Management Systems*, vol. 20, no. 4, pp. 623–635, Jul. 2017.

[49] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, “Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 783–798.

[50] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, “Network-aware operator placement for stream-processing systems,” in *22nd International Conference on Data Engineering (ICDE’06)*. IEEE, 2006.

[51] B. Gedik, “Partitioning functions for stateful data parallelism in stream processing,” *The VLDB journal: very large data bases: a publication of the VLDB Endowment*, vol. 23, no. 4, pp. 517–539, Aug. 2014.

[52] C. Hochreiner, M. Vogler, S. Schulte, and S. Dustdar, “Elastic stream processing for the internet of things,” in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, Jun. 2016.

[53] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, “Latency-aware elastic scaling for distributed data stream processing systems,” in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, May 2014.

[54] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni, “Elastic symbiotic scaling of operators and resources in stream processing systems,” *IEEE transactions on parallel and distributed systems: a publication of the IEEE Computer Society*, vol. 29, no. 3, pp. 572–585, Mar. 2018.

[55] K. Ma, B. Yang, and Z. Yu, “Optimization of stream-based live data migration strategy in the cloud: Optimization of stream-based live data migration strategy in the cloud,” *Concurrency and computation: practice & experience*, vol. 30, no. 12, p. e4293, Jun. 2018.

[56] K. G. S. Madsen, Y. Zhou, and J. Cao, “Integrative dynamic reconfiguration in a parallel stream processing engine,” in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, Apr. 2017.

[57] J. Fang, R. Zhang, T. Z. J. Fu, Z. Zhang, A. Zhou, and J. Zhu, “Parallel stream processing against workload skewness and variance,” in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. New York, NY, USA: ACM, Jun. 2017.

[58] J. Fang, R. Zhang, T. Z. J. Fu, Z. Zhang, A. Zhou, and X. Zhou, “Distributed stream rebalance for stateful operator under workload variance,” *IEEE transactions on parallel and distributed systems: a publication of the IEEE Computer Society*, vol. 29, no. 10, pp. 2223–2240, Oct. 2018.

[59] N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos, “Elastic complex event processing exploiting prediction,” in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, Oct. 2015.

[60] T. Hiessl, V. Karagiannis, C. Hochreiner, S. Schulte, and M. Nardelli, “Optimal placement of stream processing operators in the fog,” in *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*. IEEE, May 2019.

[61] S. Rizou, F. Durr, and K. Rothermel, “Solving the multi-operator placement problem in large-scale operator networks,” in *2010 Proceedings of 19th International Conference on Computer Communications and Networks*. IEEE, Aug. 2010.

[62] H. Röger, S. Bhowmik, and K. Rothermel, “Combining it all: Cost minimal and low-latency stream processing across distributed heterogeneous infrastructures,” in *Proceedings of the 20th International Middleware Conference*. New York, NY, USA: ACM, Dec. 2019.

[63] J. Xu and B. Palanisamy, “Model-based reinforcement learning for elastic stream processing in edge computing,” in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, Dec. 2021.

[64] D. Sun, S. Gao, X. Liu, X. You, and R. Buyya, "Dynamic redirection of real-time data streams for elastic stream computing," *Future generations computer systems: FGCS*, vol. 112, pp. 193–208, Nov. 2020.

[65] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE, Jun. 2015.

[66] Y. Zhou, K. Aberer, and K.-L. Tan, "Toward massive query optimization in large-scale distributed stream systems," in *Middleware 2008*, ser. Lecture notes in computer science, V. Issarny and R. Schantz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 326–345.

[67] L. I. Wang, T. Z. J. Fu, R. T. B. Ma, M. Winslett, and Z. Zhang, "Elasticutor: Rapid elasticity for realtime stateful stream processing," in *Proceedings of the 2019 International Conference on Management of Data*. New York, NY, USA: ACM, Jun. 2019.

[68] C. Lei and E. A. Rundensteiner, "Robust distributed query processing for streaming data," *ACM transactions on database systems*, vol. 39, no. 2, pp. 1–45, May 2014.

[69] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Decentralized self-adaptation for elastic data stream processing," *Future generations computer systems: FGCS*, vol. 87, pp. 171–185, Oct. 2018.

[70] V. Kakkad, A. E. Santosa, and B. Scholz, "Migrating operator placement for compositional stream graphs," in *Proceedings of the 15th ACM international conference on Modeling, analysis and simulation of wireless and mobile systems*. New York, NY, USA: ACM, Oct. 2012.

[71] T. Repantis and V. Kalogeraki, "Hot-spot prediction and alleviation in distributed stream processing applications," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008.

[72] Y. Liu, H. Xu, and W. C. Lau, "Online resource optimization for elastic stream processing with regret guarantee," in *Proceedings of the 51st International Conference on Parallel Processing*. New York, NY, USA: ACM, Aug. 2022.

[73] C. Wang, X. Meng, Q. Guo, Z. Weng, and C. Yang, "Automating characterization deployment in distributed data stream management systems," *IEEE transactions on knowledge and data engineering*, vol. 29, no. 12, pp. 2669–2681, Dec. 2017.

[74] S. Liu, J. Weng, J. H. Wang, C. An, Y. Zhou, and J. Wang, "An adaptive online scheme for scheduling and resource enforcement in storm," *ACM transactions on networking [a joint publication of the IEEE Communications Society, the IEEE Computer Society, and the ACM with its Special Interest Group on Data Communication]*, vol. 27, no. 4, pp. 1373–1386, Aug. 2019.

[75] M. Lindeberg and T. Plagemann, "A study on migration scheduling in distributed stream processing engines," in *Proceedings of the 23rd International Conference on Distributed Computing and Networking*. New York, NY, USA: ACM, Jan. 2022.

[76] M. K. Geldenhuys, D. Scheinert, O. Kao, and L. Thamsen, "Phoebe: QoS-aware distributed stream processing through anticipating dynamic workloads," in *2022 IEEE International Conference on Web Services (ICWS)*. IEEE, Jul. 2022.

[77] G. Chatzimilioudis, A. Cuzzocrea, D. Gunopulos, and N. Mamoulis, "A novel distributed framework for optimizing query routing trees in wireless sensor networks via optimal operator placement," *Journal of computer and system sciences*, vol. 79, no. 3, pp. 349–368, May 2013.

[78] F. Starks and T. P. Plagemann, "Operator placement for efficient distributed complex event processing in MANETs," in *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, Oct. 2015, pp. 83–90.

[79] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik, "Towards a streaming SQL standard," *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, vol. 1, no. 2, pp. 1379–1390, Aug. 2008.

[80] X. Du, A. Piccione, A. Pimpini, S. Bortoli, A. Knoll, and A. Pellegrini, "HUILLY: A non-blocking ingestion buffer for timestepped simulation analytics," in *Proceedings of the 24th International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid. IEEE, May 2024.

[81] Z. Deng, X. Wu, L. Wang, X. Chen, R. Ranjan, A. Zomaya, and D. Chen, "Parallel processing of dynamic continuous queries over streaming data flows," *IEEE transactions on parallel and distributed systems: a publication of the IEEE Computer Society*, vol. 26, no. 3, pp. 834–846, Mar. 2015.

[82] J. Lindén and B. Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," in *Lecture Notes in Computer Science*, ser. Lecture notes in computer science. Cham: Springer International Publishing, 2013, pp. 206–220.

[83] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, "A non-blocking priority queue for the pending event set," in *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, ser. SIMUTOOLS. Brussels, Belgium: Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST), Aug. 2016, pp. 46–55.

[84] ——, "A conflict-resilient lock-free linearizable calendar queue," *ACM transactions on parallel computing*, vol. 11, no. 1, pp. 1–32, Mar. 2024.

[85] R. Eidenbenz and T. Locher, "Task allocation for distributed stream processing," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. IEEE, Apr. 2016.

[86] H. Lustosa, F. Porto, P. Valduriez, and P. Blanco, "Database system support of simulation data," *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, vol. 9, no. 13, pp. 1329–1340, Sep. 2016.

[87] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., ser. The MIT Press. London, England: MIT Press, Jul. 2009.

[88] Z. Meng, X. Du, P. Sottovia, D. Foroni, C. Axenie, A. Wieder, D. Eckhoff, S. Bortoli, A. Knoll, and C. Sommer, "Topology-preserving simplification of OpenStreetMap network data for large-scale simulation in SUMO," in *Proceedings of the 2022 SUMO User Conference*, vol. 3. Hannover, Germany: TIB Open Publishing, Sep. 2022, pp. 181–197.

[89] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism*. Sebastopol, CA, USA: O'Reilly Media, Jul. 2007.