Received 28 July 2025 | Revised 9 January 2026 | Accepted 17 March 2026

DOI: 10.1002/spe.70064

RESEARCH ARTICLE

Operator Rebinding for Stream Processing on NUMA Machines

Xiaorui Du[1] | Andrea Piccione[2] | Adriano Pimpini[3] | Stefano Bortoli[2] | Alessandro Pellegrini[3] | Alois Knoll[1]

[1]Technical University of Munich, Munich, Germany.
[2]Huawei Munich Research Center, Munich, Germany.
[3]DICII, University of Rome "Tor Vergata", Rome, Italy.

Correspondence
Alessandro Pellegrini, DICII, University of Rome Tor Vergata, Rome, Italy.
Email: a.pellegrini@ing.uniroma2.it

Summary

Modern stream processing engines are increasingly deployed on high-core-count servers with Non-Uniform Memory Access (NUMA) architectures, where the cost of inter-socket memory access poses a significant challenge to achieving low latency and high throughput. Existing approaches to operator placement either rely on static assignments that degrade under workload variations or employ dynamic migrations that incur excessive overhead due to blocking synchronization or global barriers. This paper introduces a lock-free, NUMA-aware operator rebinding mechanism that dynamically reallocates operator tasks across threads with minimal disruption. The mechanism uses an autonomic controller to detect imbalance in per-thread queues and enacts rebinding via control messages and atomic updates, ensuring correctness without stalling execution. A two-level policy is proposed, combining NUMA-level partitioning with intra-node thread-level refinements, triggered by latency thresholds. Extensive experiments using a 300-query urban traffic analytics workload demonstrate that the proposed method achieves non-negligible throughput improvement and reduces latency compared to state-of-the-art static and METIS-based approaches. Furthermore, it reduces latency variance by an order of magnitude, illustrating the importance of fine-grained NUMA-aware scheduling in memory-bound stream processing.

KEYWORDS:
NUMA-aware scheduling, stream processing, operator rebinding, low-latency analytics

1 | INTRODUCTION

Modern stream-processing workloads, such as traffic telemetry, financial tick feeds, and high-frequency scientific experiments, are increasingly executed on single servers whose core counts compete with those of small clusters. These machines are typically multi-socket, allowing memory to be partitioned into Non-Uniform Memory Access (NUMA) domains. In a NUMA machine, an in-memory tuple can be fetched in a few tens of nanoseconds if it resides on the local node or in more than double that time if a request must traverse an interconnect[1]. The asymmetry between local and remote accesses introduces significant pressure for stream processing engines that decompose a query into hundreds of fine-grained operators and schedule their tasks dynamically. Locality requires that each operator executes close to its state, while scalability demands that work migrate freely across all cores.

Early evidence of this pressure emerged in main-memory join algorithms, where placing partitions on different nodes resulted in a throughput reduction of more than 25%[2]. Other studies[3] showed that carefully engineered NUMA-aware join variants could regain the lost performance, but only under a static operator-to-core assignment. Static placement remains attractive for its

simplicity, yet once input rates fluctuate or the queries change (e.g., due to interactive usage of the stream processing pipeline[4]), the placement may degenerate into massive load imbalance, remote-state thrashing, and oscillating latencies, a behaviour that modern low-latency applications cannot tolerate.

Distributed stream-processing systems in cloud environments side-step this issue by rescaling operators across virtual machines. Techniques such as explicit state externalisation[5] and elastic auto-parallelisation[6] relocate operator instances at coarse time scales, but the migration protocols they rely on are heavy-weight: they pin the operator during checkpointing, or depend on locks to serialize state movement. In addition, their design goal is often horizontal elasticity, rather than NUMA locality, so they overlook the latency gap between intra-socket and inter-socket memory.

Lock-freedom has been recognized in the literature[7,8,9] as a possible solution to improve stream processing throughput. For example, IBM Streams[7] introduced a mostly lock-free scheduler that allows worker threads to pull any runnable operator, thus reducing contention and admission overhead. Although the scheduler scales to hundreds of cores, its unconstrained work stealing disregards where an operator's state resides and can amplify cross-node traffic on NUMA hardware. Similarly, other non-blocking works[9] completely ignored the presence of NUMA domains also on large machines, producing operator placements that could be highly suboptimal on modern computing architectures. Recent analytical surveys confirm that neither the distributed migration protocols nor the single-node schedulers in use today combine locality preservation with sub-millisecond adaptation[10].

This paper closes that gap with a lock-free operator rebinding mechanism that is explicitly NUMA-aware. Each operator's tasks are enqueued in a per-thread work list. An autonomic controller monitors queue depths and processing costs, and when it detects a sustained imbalance, it computes a new mapping that minimises both the maximum per-thread backlog and the volume of remote tuple transmission. Rebinding is enacted by injecting lightweight *control tuples* into the affected queues, allowing ongoing tuple processing to proceed uninterrupted, and no global barrier is required. Because the global mapping is held in atomic variables, thread-safe updates never block, and tasks that reach an outdated queue are lazily forwarded to their new owner, preserving correctness without stalls.

We evaluate the mechanism on two dual-socket Xeon platforms, one with 24 physical cores and another with 36 physical cores. Under highly bursty agent-based traffic simulations, the mechanism sustains times higher throughput and shows lower latency than state-of-the-art static, greedy, or METIS-driven[11] operator assignments. More importantly, the latency variance is reduced by an order of magnitude because the operator state rarely crosses the sockets after initial warm-up. These results demonstrate that, for modern memory-bound streaming analytics, fine-grained, lock-free, NUMA-aware rebinding is a viable alternative to both coarse-grained distributed migrations and oblivious single-node schedulers.

The remainder of this paper is structured as follows. In Section 2, we describe the system model of our target stream processing system. Related work is discussed in Section 3. We present our NUMA binding policies in Section 4, while an experimental assessment is provided in Section 5.

2 | SYSTEM MODEL AND ASSUMPTIONS

We consider stream processing pipelines that are executed by a stream processing engine, which is expected to cope with multiple simultaneous queries. These queries are described using the Streaming SQL formalism[12], which combines selections, projections, joins, and window aggregations of tuples. More generally, a query describes a computational tree, where leaves are *data sources*, the root is the query *output*, and non-leaf vertices are the *operators* that compose the query.

Query trees that share one or more operators can be partially merged, either manually or automatically[13]. In general, even when shared operators are combined, the queries sent to the streaming engine implicitly define a global Directed Acyclic Graph (DAG), which we will refer to as the *computation graph*. Thus, at a high level, the operation of a streaming engine involves evaluating the computation graph whenever new data become available from any of the data sources.

Triggering a computation every time a new tuple is ingested would be severely inefficient. We assume the presence of *buffering* or *tuple batching* methods to achieve good throughput, as is typically done by off-the-shelf streaming engines. 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 at the same time, it increases latency. As a result, especially in low-latency scenarios, task scheduling can be the system bottleneck.

We consider data sources that either continuously generate considerable amounts of data[8], or observe bursty tuple generation patterns, as could be the case when simulation output is used in interactive applications[14], where users may play with simulation

pacing, and insert/remove queries relatively often. 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.

In our system model, we assume a decentralized scheduling architecture in which each thread maintains its own task queue, avoiding the single-queue bottleneck and delivering higher throughput with more predictable latency on homogeneous hardware. Although global monolithic schedulers (implemented as virtual-time priority queues with efficient non-blocking, mutex-free designs[15,16,17]) offer a straightforward concurrency scheme, they exhibit suboptimal performance for interactive applications. Clearly, the decentralized design incurs greater implementation complexity, particularly in deciding task–queue assignments. In order to mitigate this complexity, we employ a static binding that maps each operator i to a designated thread j, ensuring that all tasks for operator i are enqueued and processed exclusively on thread j[18].

2.1 | Problem Statement

As mentioned, a DAG captures 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. These operators are executed on a multi-core system with a thread set $T = \{T_1, T_2, \ldots, T_m\}$, where the goal is to assign each operator $V_i \in V$ to a thread $T_j \in T$ through a binding function $M : V \to T$.

A critical design challenge is choosing how to bind operators to threads efficiently to ensure balanced workload distribution and high throughput while minimizing overhead. The design of the binding strategy becomes particularly complex in multi-core NUMA systems, where data locality and thread contention must be carefully managed. A *static binding* approach establishes a fixed operator-to-thread mapping M_{static} before the execution begins and maintains it throughout. While static binding is simple and incurs minimal runtime overhead, it fails to account for dynamic changes in thread workload or system conditions. Thread workloads can be highly variable due to factors such as input data rate fluctuations, non-linear operator processing costs, or external load on the host machine. As a result, static policies often lead to suboptimal performance unless guided by sophisticated heuristics, since finding the optimal assignment is NP-hard[19].

Differently, *dynamic binding* strategies change the mapping $M(t) : V \to T$ over time t, during execution. These approaches adapt to real-time changes in workload, which is especially important in contexts like large-scale scientific simulations that generate data streams with highly variable volumes[20]. However, dynamic binding incurs additional complexity and runtime cost, particularly when frequent re-evaluation of workload and operator costs is needed to maintain balance.

The choice of task queue organization further influences the effectiveness of binding policies. In the centralized queue model, all threads pull tasks from a shared global queue, resulting in implicit load balancing but often suffering from contention and poor NUMA locality. The Per-NUMA node queue model assigns operators to nodes and enables basic load balancing through work stealing. While this approach improves over centralized designs, it still lacks fine-grained thread-level control. In contrast, the per-thread task queue model requires explicit operator-to-thread mappings, offering the highest degree of granularity and optimization potential, albeit at the cost of increased design complexity.

3 | RELATED WORK

The goal of identifying an operator placement is to effectively distribute query processing across network nodes in order to meet system objectives. However, finding the optimal placement has been proven to be NP-hard[19]. To achieve near-optimal solutions, heuristics are commonly used. The research community has made significant contributions in this area. For a comprehensive discussion, we refer the reader to[21,22,23,10].

The simplest migration approach involves moving an operator to a new host and replays upstream tuples[24,25], sometimes duplicating streams on both hosts for gradual handover[26] or employing *state shedding* to transfer only critical partial state[27], while more advanced schemes buffer redirected streams[28] or output to multiple key-partitioned replicas to adjust distribution without pause[6,29]. Checkpoint-assisted algorithms reduce downtime by transferring minimal state using incremental or replicated checkpoints and can prioritize partial migrations for improved responsiveness[30,31]. All costs related to operator migration are addressed by relying on local, non-blocking rebinding on large machines[9], which benefits from zero migration downtime

and incurs no cost for network copying of the tuple. At the same time, any rebinding obtained by the approach in [9] may be sub-optimal, exactly because the machines are typically organized in multiple NUMA zones. We explicitly attack this problem in this article by providing NUMA-aware non-blocking rebinding policies.

Deciding when and which operators to migrate is also an important aspect to consider in the policies, as excessive migrations can cancel performance gains through secondary costs. In the literature, techniques to mitigate these aspects have been identified, such as QoS-triggered actions or coarse-grained re-evaluations of the objective function [32]. Interference metrics have also been used to direct migrations toward less-contended nodes [33]. Coordination may be centralized via a dedicated controller [34] or decentralized, with operators acting autonomously [35]. Sequential planning of dependent migrations remains essential for load balancing and geographically distributed graphs [36].

Integer linear programming has been used for migration decisions [37], but its need for a global network view could hinder the scalability of the approach. Consequently, most systems rely on heuristics, including operator-scaling strategies that minimise instance counts while preserving QoS [38], stream-overlay placement to reduce network traffic [39], interference-aware load balancing that predicts resource usage and migrates computations when thresholds are exceeded [33], partitioning schemes that maintain balanced mappings [40], and elastic frameworks that scale resources using CPU utilization thresholds [41].

Accurate migration decisions require explicit cost models, yet most systems simply monitor tuple-level performance and treat those observations as cost proxies. Few works predict latency analytically, for instance by combining input rate, migration time, and time before queued events can be processed [42]. State size is considered as a surrogate for migration time, and when multiple operators move concurrently, the longest transfer often defines the overall delay [19,37]. More complex models add parameters such as adaptation type, round-trip delay, or link bandwidth and solve min–max formulations to bound the slowest migration [37]. Some approaches also include migration time, control-message volume, or monetary expenditure directly as optimisation objectives [26]. In decentralised fog and edge scenarios, network usage and link latency dominate the cost equation, so overlay-aware heuristics minimise bandwidth and the frequency of migrations [39,43].

Interference-aware scheduling predicts future packet load to assign operators an interference score that guides migration decisions [33], while WASP relies on expected input–output rates for the same purpose [37]. Violated latency constraints are used to trigger scaling or balancing actions whose cost grows as deadlines tighten [44]. Key-partitioning schemes that minimise load variance may still incur expensive state transfers [40], so decentralised fog deployments often embed placement and migration in a latency–bandwidth–load cost space to suppress unnecessary moves [39]. Modern frameworks mitigate migration frequency by penalising churn in multi-objective heuristics [45], while predictive controllers based on model-predictive control or reinforcement learning schedule scaling before overloads materialise [46,47]. Energy consumption can also inform migration under resource constraints [48].

All in all, these proposals do not explicitly consider the performance penalty that a suboptimal assignment on NUMA machines can deliver. Therefore, our proposal is orthogonal and complementary to much of the existing literature on operator assignment for stream processing engines.

4 | NUMA-AWARE BINDING POLICIES

In this section, we present our operator binding policy tailored for NUMA architectures, in which we explicitly rely on a per-thread task queue. The ultimate goal is to illustrate a dynamic policy that can rebind operators at runtime based on the current observed workload, system state, and other relevant runtime metrics. For the sake of clarity, we begin with simpler (static) assignments and then refine these policies to a fully dynamic one. The different policies will be compared experimentally in Section 5, highlighting the performance improvement that a dynamic NUMA-aware operator placement can deliver.

An important aspect related to operator rebinding is the amount of housekeeping work 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 $M_0, M_1, \ldots, M_n$ that are installed during the lifetime of the stream processing application. If this condition is not met, we can consider the query result incorrect due to dynamic binding.

Therefore, the system that installs the selected operator-to-thread binding 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's binding. If rebinding is computed frequently, as we are considering, the overall performance may degrade to an unacceptable level.

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. Additionally, the processing threads must relocate the tasks contained in their local scheduler according to the new binding.

To guarantee the correctness of this process, processing threads need to synchronize 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 operator locking is not sufficient to prevent 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 to indicate this. 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 fear of concurrent updates by the operators. Finally, all threads commit the newly installed binding by waiting on a second thread barrier to avoid resuming operator processing until all threads have completed their own rebinding operations.

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 non-blocking approach that does not require explicit barrier synchronisation.

In our proposal, the global binding mapping is maintained in an array of atomic integer variables, where 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 and compares 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 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.

4.1 | Static Binding

We start our journey with very simple static binding policies. The simplest approach, which we name *static operator binding*, does not take into account any interaction among operators in the stream processing pipeline. With this strategy, each operator in the set V is assigned to some specific thread in T, e.g., using a round-robin strategy, prior to execution. The assignment remains fixed during execution. In other words, the mapping can be expressed by Equation (1). Here, k denotes the index of operator v in the operator set V, and $x_{v_k,i} = 1$ indicates that operator v_k is bound to thread i.

$$x_{v_k,i} = \begin{cases} 1 & \text{if } i = k \bmod |T| \\ 0 & \text{otherwise} \end{cases}. \tag{1}$$

Clearly, this simple assignment does not account for interconnections between operators, let alone differences in operator complexity or runtime dynamics. Therefore, such a naïve policy is expected to behave extremely poorly in practice. To account at least for operator interconnections, we can design a still static binding, which we name *partition binding*, in which the operator set V is divided into fixed partitions based on each operator's location in the DAG, and its incoming/outgoing edges. Each partition groups operators that process data related to the same region or logical subdomain. Let $P = \{P_0, P_1, \ldots, P_{|P|-1}\}$ be the set of such partitions, with each $P_j \subseteq V$ representing a subset of the DAG. These partitions are then assigned to threads in T using a round-robin strategy. The operator-to-thread mapping is defined by Equation (2). Here, $x_{v,i} = 1$ indicates that the operator v from partition P_j is bound to thread i. Compared to static operator binding, this approach improves cache locality and memory reuse by co-locating related operators on the same thread. This can reduce inter-thread communication overhead and enhance performance on NUMA systems.

$$x_{v,i} = \begin{cases} 1 & \text{if } v \in P_j \text{ and } i = j \bmod |T| \\ 0 & \text{otherwise} \end{cases}. \tag{2}$$

Still, this kind of assignment does not consider any possible fluctuations in the workload, which may lead to potential performance penalties that are independent of the queries but depend solely on the amount of data a certain pipeline is processing at a given instant. To cope with this aspect, we must incorporate some form of monitoring to track the amount of tuples that are ingested and processed in the pipeline, and reshuffle the assignment based on the current workload state.

4.2 | Greedy Dynamic Operator Binding

Dynamic binding policies attempt to recompute an assignment based on the current state of the processing pipeline. A simple approach, which has already been explored in the literature[9], is to base the reassignment on greedy algorithms to dynamically determine where to place operators at runtime based on observed workload imbalances.

This approach typically begins with static operator binding in Section 4.1, which is then refined (e.g., at regular intervals) by evaluating operator workloads and identifying imbalances across threads. Operators contributing to overload on their assigned threads are migrated to underloaded threads in order to improve the overall load distribution and reduce query processing latency. Such an approach is illustrated in Algorithm 1. Each operator $v \in V$ is characterized by an estimated processing cost c_v and a pending queue size q_v^{pend}. Its estimated workload w_v is defined by Equation (3). For each thread $i \in T$, the thread workload λ_i is then computed according to Equation (4) as the sum of the workloads of all operators assigned to it, where $V_i \subseteq V$ denotes the set of operators currently mapped to thread i. Finally, the average workload across all threads is given by Equation (5). The policy (Algorithm 1) aims to minimize the maximum thread load variance. It identifies overloaded threads (where $\lambda_j > \overline{\lambda}$) and iteratively reassigns their operators to the least loaded thread $k = \arg\min_{i \in T} \lambda_i$. To minimize migration overhead and avoid instability, operators are processed in ascending order of their workload w_v. If $\lambda_j \leq \overline{\lambda}$, the operator remains on its current thread.

$$w_v = c_v \cdot q_v^{\text{pend}}. \tag{3}$$

$$\lambda_i = \sum_{v \in V_i} w_v, \tag{4}$$

$$\overline{\lambda} = \frac{1}{|T|} \sum_{i \in T} \lambda_i. \tag{5}$$

This policy focuses on mitigating workload imbalance and incrementally updates the mapping with minimal runtime overhead. While it introduces some migration overhead, it effectively adapts to workload variations and improves balance. However, it treats all threads equally, which may result in poor NUMA locality and increased remote memory access costs. Additionally, it does not consider operator locality, which can further impact performance in NUMA environments.

4.3 | Dynamic NUMA-aware Partition Binding

To explicitly account for the performance penalties introduced by operator execution on NUMA architectures, we introduce a runtime-adaptive policy that maintains both structural locality and latency sensitivity. This policy dynamically reassigns operators based on observed workload and query latencies, based on a two-stage optimization method: first, it performs a *partition refinement*, which then undergoes a *dynamic two-level operator rebinding*. The dynamic NUMA-aware policy is triggered only

Algorithm 1 Greedy Dynamic Operator Binding

1: **Input:** V, T, $M_{\text{Op\_old}}$
2: **Output:** $M_{\text{Op\_new}}$
3: $M_{\text{Op\_new}} \leftarrow M_{\text{Op\_old}}$
4: **for all** $v \in V$ **do**
5: Compute the operator workload w_v ▷ Eq. 3
6: **end for**
7: Collect thread workload λ_i for all $i \in T$ ▷ Eq. 4
8: Compute average workload $\overline{\lambda}$ ▷ Eq. 5
9: Sort operators $v \in V$ in ascending order of w_v
10: **for all** operator v in sorted order **do**
11: $j \leftarrow M_{\text{Op\_old}}(v)$
12: **if** $\lambda_j > \overline{\lambda}$ **then**
13: $k \leftarrow \arg\min_{i \in T} \lambda_i$
14: $M_{\text{Op\_new}}(v) \leftarrow k$
15: Update thread workloads
16: **end if**
17: **end for**
18: **return** $M_{\text{Op\_new}}$

when the end-to-end latency of any query exceeds a predefined threshold δ; otherwise, the system continues to use the existing operator-to-thread bindings.

4.3.1 | Partition Refinement

This approach, referred to as *Critical-Path-Based Partition Refinement*, typically starts with the static partition binding described in Section 4.1. Let $P_j \subseteq V$ be a static region-based operator partition, and let $G_j = (P_j, E_j)$ denote the subgraph of the operator DAG corresponding to that partition. The goal is to divide each P_j into smaller sub-partitions $\{S_k\}$ that capture latency-critical paths and topologically connected components to support fine-grained runtime decisions. The algorithm returns two mappings:

$$\Phi : P_j \to \{S_k\}, \quad \Psi : S_k \to \{v\}$$

where Φ maps each partition to its constituent sub-partitions and Ψ maps each sub-partition to its associated operators. The detailed operations are presented in Algorithm 2.

For each partition P_j, the algorithm computes the critical path as follows (lines 6 – 9). First, it estimates the downstream cumulative processing cost $c(v)$ for each operator $v \in P_j$ using ComputeCriticalPathCosts(). Then, it invokes TraceLongestPath() to extract the longest-cost path $C_j \subseteq P_j$. This path is assigned to a new sub-partition S_k, added to $\Psi(S_k)$, and recorded in $\Phi(P_j)$. The remaining operators $R = P_j \setminus C_j$ are decomposed into connected subgroups (lines 11 – 17). An arbitrary operator $v \in R$ is selected, and a connected component $S \subseteq R$ is constructed using ExploreConnectedOperators() (e.g., via DFS). This group is assigned to a new sub-partition S_k, and the mappings Φ and Ψ are updated accordingly. This process repeats until all operators in P_j have been assigned. This decomposition ensures that 1) the most latency-sensitive computation path is explicitly preserved; 2) the remaining operators are grouped according to structural affinity; 3) the resulting sub-partitions reflect execution-aware decomposition of the DAG for fine-grained scheduling decisions.

4.3.2 | Dynamic Two-Level Operator Rebinding

The refined partition serves as the basis for the runtime operator rebinding, which relies on a two-level dynamic rebinding strategy, as outlined in Algorithm 3. The inputs include $M_{\text{P\_old}}$ (the current partition-to-NUMA mapping), $M_{\text{SP\_old}}$ (the current sub-partition-to-thread mapping), $M_{\text{Op\_old}}$ (the current operator-to-thread mapping), and the mappings Φ and Ψ, which originate from the partition refinement stage. The outputs are the updated mappings $M_{\text{P\_new}}$, $M_{\text{SP\_new}}$, and $M_{\text{Op\_new}}$. Here, $M_{\text{Op\_new}}$ provides the final thread assignment for each operator, derived from the sub-partition assignment in $M_{\text{SP\_new}}$. The rebinding

Algorithm 2 Critical-Path-Based Partition Refinement

1: **Input:** Operator partitions $\mathcal{P} = \{P_0, P_1, \dots\}$, each $P_j \subseteq V$
2: **Output:** Mappings: $\Phi : P_j \to \{S_k\}$, $\Psi : S_k \to \{v\}$
3: Initialize: $\Phi \leftarrow \emptyset$, $\Psi \leftarrow \emptyset$, $k \leftarrow 0$
4: **for** each partition $P_j \in \mathcal{P}$ **do**
5: *// Extract critical path*
6: $c(v) \leftarrow$ ComputeCriticalPathCosts(P_j) for all $v \in P_j$
7: $C_j \leftarrow$ TraceLongestPath(c, P_j)
8: $\Psi(S_k) \leftarrow C_j, \quad \Phi(P_j) \leftarrow \Phi(P_j) \cup \{S_k\}$
9: $k \leftarrow k + 1$
10: *// Decompose remaining graph*
11: $R \leftarrow P_j \setminus C_j$
12: **while** $R \neq \emptyset$ **do**
13: Select arbitrary $v \in R$
14: $S \leftarrow$ ExploreConnectedOperators(v, R)
15: $\Psi(S_k) \leftarrow S, \quad \Phi(P_j) \leftarrow \Phi(P_j) \cup \{S_k\}$
16: $R \leftarrow R \setminus S, \quad k \leftarrow k + 1$
17: **end while**
18: **end for**
19: **return** Φ, Ψ

process consists of two parts. The first one, named *NUMA-Level Rebinding* (lines 4–18), explicitly addresses the overhead that can be experienced when tight operator partitions are assigned across different NUMA nodes.

Let $\mathcal{N} = \{n_0, \dots, n_{|\mathcal{N}|-1}\}$ be the set of NUMA nodes. Each node $n \in \mathcal{N}$ maintains a NUMA workload value Λ_n, computed as the sum of thread workloads in that node. The average NUMA workload is computed by Equation (6). Next, for each partition P_j, we define its workload w_{P_j} as the cumulative workload of all operators in its sub-partitions, as given by Equation (7), where the operator workload is $w_v = c_v \cdot q_v^{\text{pend}}$ (Equation (3)). Based on these workloads, partitions are sorted in ascending order of w_{P_j} (line 9). Then, for any partition whose assigned NUMA node $n_{\text{old}} = M_P(P_j)$ is overloaded ($\Lambda_{n_{\text{old}}} > \overline{\Lambda}$), it is reassigned to the least loaded node via Equation (8). After this reassignment, operators in the partition are temporarily assigned to the thread with the least workload in node n_{new}, and the NUMA workloads are updated accordingly using Equation (9). The second part of the rebinding process is the actual *thread-level rebinding* (lines 19–37). Once the partitions are mapped to NUMA nodes, the system performs fine-grained thread-level rebinding of sub-partitions. For each NUMA node n, the thread workload λ_t ($t \in T_n$) is collected, and the average thread workload for the node is computed by Equation (10). Subsequently, the workload of each sub-partition s on NUMA node n is computed by Equation (11) as the sum of its operators' workloads. Sub-partitions are then sorted in ascending order of w_s (line 26). If a sub-partition is currently assigned to an overloaded thread t_{old}, it is reassigned to the least loaded thread via Equation (12). The thread workloads are then updated accordingly using Equation (13). Finally, the mapping $M_{\text{Op\_new}}$ is updated based on $M_{\text{SP\_new}}$, which incorporates both NUMA-level and thread-level load balancing decisions.

$$\overline{\Lambda} = \frac{1}{|\mathcal{N}|} \sum_{n \in \mathcal{N}} \Lambda_n. \tag{6}$$

$$w_{P_j} = \sum_{s \in \Phi(P_j)} \sum_{v \in \Psi(s)} w_v, \tag{7}$$

$$n_{\text{new}} = \arg\min_{n \in \mathcal{N}} \Lambda_n. \tag{8}$$

$$\Lambda_{n_{\text{new}}} \leftarrow \Lambda_{n_{\text{new}}} + w_{P_j}, \quad \Lambda_{n_{\text{old}}} \leftarrow \Lambda_{n_{\text{old}}} - w_{P_j}. \tag{9}$$

$$\overline{\lambda}_n = \frac{1}{|T_n|} \sum_{t \in T_n} \lambda_t. \tag{10}$$

$$w_s = \sum_{v \in \Psi_n(s)} w_v. \tag{11}$$

Algorithm 3 Dynamic Two-Level Operator Rebinding

Input: $M_{\text{P\_old}}$, $M_{\text{SP\_old}}$,$M_{\text{Op\_old}}$, Φ, Ψ
Output: $M_{\text{P\_new}}$, $M_{\text{SP\_new}}$, $M_{\text{Op\_new}}$
Initialize $M_{\text{P\_new}} \leftarrow M_{\text{P\_old}}$, $M_{\text{SP\_new}} \leftarrow M_{\text{SP\_old}}$, $M_{\text{Op\_new}} \leftarrow M_{\text{Op\_old}}$
Collect thread workloads λ_t and compute NUMA workloads: $\Lambda_n = \sum_{t \in T_n} \lambda_t$
Compute average NUMA workload $\overline{\Lambda}$ ▷ Eq. 6
for all $P_j \in \Phi$ **do**
    Compute w_{P_j} ▷ Eq. 7
end for
Sort partitions in ascending order of w_{P_j}
for all P_j in sorted order **do**
    $n_{\text{old}} \leftarrow M_{\text{P\_old}}(P_j)$
    if $\Lambda_{n_{\text{old}}} > \overline{\Lambda}$ **then**
        $n_{\text{new}} \leftarrow \arg\min_n \Lambda_n$ ▷ Eq. 8
        Temporarily assign all operators in P_j to the least-loaded thread in $T_{n_{\text{new}}}$
        Update $M_{\text{P\_new}}(P_j) \leftarrow n_{\text{new}}$
        Update NUMA workloads ▷ Eq. 9
    end if
end for
for all NUMA node $n \in \mathcal{N}$ **do**
    Collect thread workloads λ_t for $t \in T_n$
    Compute average thread workload $\overline{\lambda}_n$ ▷ Eq. 10
    // Retrieve all sub-partitions $\Psi_n$mapped to NUMA node n
    for all $s \in \Psi_n$ **do**
        Compute w_s ▷ Eq. 11
    end for
    Sort sub-partitions in ascending order of w_s
    for all sub-partitions s in sorted order **do**
        $t_{\text{old}} \leftarrow M_{\text{SP\_old}}(s)$
        if $\lambda_{t_{\text{old}}} > \overline{\lambda}_n$ **then**
            $t_{\text{new}} \leftarrow \arg\min_{t \in T_n} \lambda_t$ ▷ Eq. 12
            Assign all $v \in \Psi_n(s)$ to t_{new}
            Update $M_{\text{SP\_new}}(s) \leftarrow t_{\text{new}}$
            Update thread workloads ▷ Eq. 13
            $M_{\text{Op\_new}}(v) \leftarrow t_{\text{new}}$, $\forall v \in \Psi_n(s)$
        end if
    end for
end for
return $M_{\text{P\_new}}$, $M_{\text{SP\_new}}$, $M_{\text{Op\_new}}$

$$t_{\text{new}} = \arg\min_{t \in T_n} \lambda_t. \tag{12}$$

$$\lambda_{t_{\text{new}}} \leftarrow \lambda_{t_{\text{new}}} + w_s, \quad \lambda_{t_{\text{old}}} \leftarrow \lambda_{t_{\text{old}}} - w_s. \tag{13}$$

5 | EXPERIMENTAL EVALUATION

This section presents an empirical evaluation of the proposed *dynamic partition-based operator binding* policy in the context of real-time stream processing using simulation data from CityMoS[49], a high-performance agent-based urban traffic simulator. The

objectives of our evaluation are to examine the performance characteristics of the proposed policy across a range of workload conditions and hardware scales, to analyze its adaptability to changing workload, and to compare it with established operator binding approaches using average latency and latency distribution metrics. To ensure that our proposal is competitive against state-of-the-art binding policies, we also compared our results with assignments based on graph partitioning, which was carried out using the METIS [11] graph partitioning tool.

It is worth noting that we do not include a direct comparison with OS-level scheduling mechanisms, such as the Completely Fair Scheduler (CFS) or Linux AutoNUMA. These mechanisms operate at the granularity of OS threads, whereas our approach schedules fine-grained operators onto a fixed pool of pinned worker threads. Mapping thousands of operators to individual OS threads to leverage OS scheduling would incur prohibitive context-switching overhead and resource contention. Furthermore, OS schedulers lack semantic awareness of the streaming DAG (e.g., data dependencies or operator selectivity), making them unsuitable for the fine-grained, structure-aware optimization required in this context. Instead, we compare against task queue baselines that represent the standard runtime mechanisms where work is distributed dynamically without specific binding logic. We also compare our results against baseline policies based on METIS partitioning, as it is a commonly used approach for dynamic load balancing [50].

5.1 | Baseline Policies based on METIS

As a baseline, we have used METIS to optimize operator placement based on profiling data. Initially, queries are executed using the static policy described in Section 4.1 to collect runtime metrics. Each operator $v \in V$ accumulates its total computation time $comp_v$, which is used as the vertex weight. Similarly, for each edge $u \to v \in E$, the estimated data transfer time is used as the edge weight, computed as $s_{u\to v}/mbw_{u\to v}$, where $s_{u\to v}$ is the amount of data transferred and $mbw_{u\to v}$ is the measured memory bandwidth between the threads hosting operators u and v. For simplicity, $mbw_{u\to v}$ is treated as a constant measured offline using the `lmbench` tool [51]. These weights, along with the DAG topology, are encoded into a METIS-compatible input file `weight.graph`, which includes the number of vertices, number of edges, format flags, and a weighted adjacency list. We then use the `gpmetis` tool from the METIS suite [52] to partition the computation graph based on the constructed weights and topology.

This command partitions the computation graph into $|T|$ balanced subgraphs while minimizing inter-partition communication. Let $\pi : V \to \{0, 1, \ldots, |T| - 1\}$ be the mapping returned by METIS, where $\pi(v)$ denotes the partition (thread) assigned to operator v. The resulting binding is denoted by Equation (14).

$$x_{v,i} = \begin{cases} 1 & \text{if } i = \pi(v) \\ 0 & \text{otherwise} \end{cases}. \tag{14}$$

All operators within the same partition are bound to the same thread, and queries are re-executed using this fixed assignment. Compared to simple static strategies, this method improves load balancing and inter-operator locality by using real workload information and minimizing communication costs. However, the assignment is computed offline and requires prior access to representative workload data, which increases the complexity of the operator binding process. This prerequisite can be impractical or even impossible in real-time data processing scenarios, where the characteristics of the data may not be known in advance. Moreover, since the binding remains fixed during execution, it cannot adapt to runtime workload variations or evolving data patterns, limiting its effectiveness in dynamic environments.

We have also employed a dynamic approach based on METIS, which extends the static METIS approach by continuously applying the partitioning process at runtime. Rather than executing the METIS algorithm once at the end of the profiling phase, the system periodically executes the METIS algorithm using the latest runtime profiling metrics. Specifically, the current computation time $comp_v$ and the edge data transfer $s_{u\to v}$ are collected to construct a new partitioning input.

However, METIS is stateless and does not preserve historical partitioning information. Consequently, consecutive partitioning results can vary significantly, potentially causing excessive operator migrations. To address this, we introduce a remapping step that minimizes disruption between partitioning iterations.

We compute a similarity matrix between the current and previous partition mappings, following a strategy similar to that used by Xu et al [53]. The matrix captures the number of overlapping operators between the new and old partitions. Based on this, we map each new METIS partition to the old partition with which it shares the most operators. This remapping strategy aims to preserve operator-thread locality by mapping each new METIS partition to the old partition with which it shares the most operators. If, after remapping, any operators remain unassigned (which rarely occurs), they are assigned to threads using a simple round-robin strategy.

Let $\pi_t(v)$ be the partition assigned to operator v at time t, and let $\tilde{\pi}_t(v)$ be the remapped result after minimizing migration cost. The operator-thread binding at time t can be expressed by Equation (15).

$$x_{v,i}(t) = \begin{cases} 1 & \text{if } i = \tilde{\pi}_t(v) \\ 0 & \text{otherwise} \end{cases}. \tag{15}$$

For example, consider a system with three threads and six operators $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$. At time t, METIS assigns operators to partitions as follows:

$$\pi_t(v_0) = 0, \quad \pi_t(v_1) = 0, \quad \pi_t(v_2) = 1, \quad \pi_t(v_3) = 2, \quad \pi_t(v_4) = 2, \quad \pi_t(v_5) = 2$$

At time $t + 1$, due to updated workload information, METIS generates a new partitioning:

$$\pi_{t+1}(v_0) = 1, \quad \pi_{t+1}(v_1) = 1, \quad \pi_{t+1}(v_2) = 2, \quad \pi_{t+1}(v_3) = 0, \quad \pi_{t+1}(v_4) = 0, \quad \pi_{t+1}(v_5) = 0$$

To avoid full reassignment of threads, we compute overlaps between new and previous partitions: new partition 0 shares 3 operators with old partition 2, new partition 1 shares 2 operators with old partition 0, and new partition 2 shares 1 operator with old partition 1. Based on maximum overlap, we remap:

$$\tilde{\pi}_{t+1}(v_0) = 1, \quad \tilde{\pi}_{t+1}(v_1) = 1, \quad \tilde{\pi}_{t+1}(v_2) = 2, \quad \tilde{\pi}_{t+1}(v_3) = 0, \quad \tilde{\pi}_{t+1}(v_4) = 0, \quad \tilde{\pi}_{t+1}(v_5) = 0$$

This dynamic binding strategy combines the partitioning accuracy of METIS with adaptability to runtime workload changes. The remapping step preserves operator locality and reduces unnecessary migrations. However, frequent execution of the METIS algorithm is costly and can introduce non-negligible overhead to the system.

5.2 | Queries

We evaluate three representative continuous queries, similar to those in our prior work[54]. We show them in FlinkSQL-like notation below. The evaluation relies on simulation data generated by CityMoS[49] using a traffic scenario for Shenzhen, China, generated by TSMM[55]. Query 1 computes the *average speed* of different types of vehicles in each region. The input stream contains tuples with the schema: $\langle timestamp, vehicle_type, id, x, y, speed, acceleration, latitude, longitude \rangle$. The query applies a region filter via the function `geo_to_region(latitude, longitude)`, projects $\langle timestamp, vehicle\_type, speed \rangle$, and performs a grouped aggregation on `(region_id, vehicle_type)` using a 30-second window with a 1-second slide. Query 2 measures the *number of active vehicles* in each region. After filtering by valid geographic coordinates, the system groups vehicles by `region_id` using `geo_to_region(latitude, longitude)` and computes the count using a 10-second window with a 1-second slide. Query 3 identifies the *top-3 fastest vehicles* for each vehicle type in every region. It filters vehicles by coordinates, projects $\langle timestamp, vehicle_type, speed \rangle$, and ranks vehicles within `(region_id, vehicle_type)` using `ROW_NUMBER()` ordered by descending speed. The query operates over a 20-second window with a 1-second slide, returning vehicles with `rank` ≤ 3. Shenzhen is spatially partitioned into 100 predefined regions based on geolocation data, and each region executes all three queries in parallel. As a result, the overall query workload comprises 300 region-specific queries. These queries are compiled into a single computation graph, forming a Directed Acyclic Graph (DAG) consisting of 1,201 operators, including a single shared source operator, as illustrated in Figure 1.

5.3 | Configurations and Hardware

We assess the performance of different query execution models and operator binding policies using three workload configurations that capture total load variability, spatial imbalance, and temporal fluctuations. In addition, simulation is decoupled from stream processing to eliminate interference from the simulation layer and avoid variability introduced by disk I/O. Accordingly, rather than running simulations live during evaluation, the simulation output is recorded in advance, preloaded into memory, and replayed using a dedicated CPU core. Furthermore, to provide a uniform task scheduling infrastructure across all execution models, we adopt the Intel TBB concurrent priority queue[56] as the underlying task container. By using the same queue implementation for all different runs, we ensure that any observed performance differences are related to execution strategies and binding policies rather than differences in the implementation of task management. Finally, we evaluate each configuration using two key performance metrics. The first is the *average query latency*, calculated by measuring the end-to-end latency for

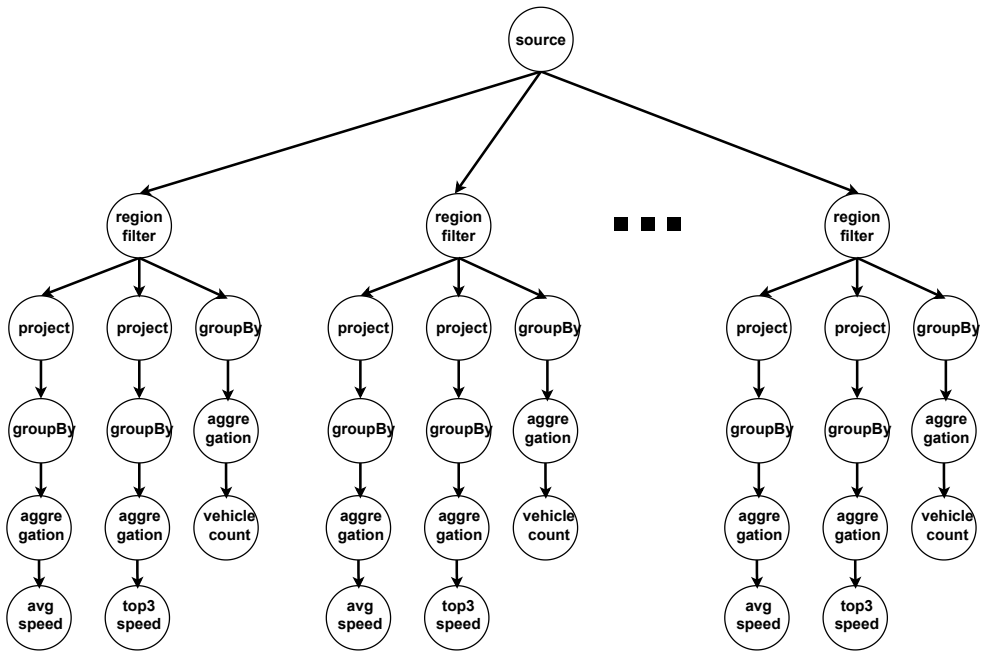


FIGURE 1 DAG for 300 queries.

each query at every simulation step and averaging those values over time. The second is the *latency distribution*, which captures the number of queries falling within predefined latency ranges, providing insights into both tail latency and overall system responsiveness.

All experiments are conducted on servers running Ubuntu 20.04.6 LTS. The evaluation uses two dual-socket NUMA machines with different core counts, reflecting commonly used dual-socket server configurations in single-node stream processing environments. The first machine is equipped with dual-socket Intel Xeon E5-2680 v3 @ 2.50 GHz CPUs. Each socket contains 12 physical cores, resulting in a total of 24 physical cores (48 logical CPUs). The cache hierarchy consists of 768 KiB of L1 cache, 6 MiB of L2 cache, and a shared 60 MiB L3 cache per socket. The second machine features dual-socket Intel Xeon E5-2697 v4 @ 2.3GHz. Each socket contains 18 physical cores, resulting in 36 physical cores (72 logical CPUs). Its cache hierarchy consists of 1.1 MiB of L1 cache, 9 MiB of L2 cache, and a shared 90 MiB L3 cache per socket. For all experiments, hyper-threading is disabled to isolate physical core behavior, and each data point is averaged over 5 runs.

5.4 | Results

In this section, we provide the results of the experimental assessment carried out to compare the behavior of the proposed policies against the baseline policies discussed above. We have used different workload configurations, which are detailed below.

5.4.1 | Workload 1: Static Total Load with Static Partition Load

The first workload represents a static scenario in which both the total number of vehicles and their spatial distribution remain constant over time. This setup reflects stable traffic conditions with minimal mobility. To construct this workload, we fix the total vehicle count and distribute vehicles across regions using a controlled skew. The assignment is based on an initial base count a and an increase ratio r following Equation (16). This results in a gradually increasing load from region 0 to region 99. If the formula does not assign all vehicles exactly due to rounding or count mismatches, the remaining vehicles are distributed across regions in a round-robin manner until all vehicles are placed.

$$\texttt{vehicles}_{\text{region\_id}} = a \cdot (1 + \text{region\_id} \cdot r) \tag{16}$$

Figure 2 shows the performance of different task queue designs and operator binding policies under workload 1 on a dual-socket 24-core platform. Figures 2a and 2b illustrate the average query latency over wall-clock time for 340,000 (with a =

```
SELECT geo_to_region(latitude, longitude) AS region_id,
       vehicle_type,
       HOP_END(ts, INTERVAL '1' SECOND, INTERVAL '30' SECOND) AS window_end,
       AVG(speed) AS avg_speed
FROM (
  SELECT ts, vehicle_type, speed, latitude, longitude
  FROM vehicle_stream
  WHERE latitude IS NOT NULL AND longitude IS NOT NULL
)
GROUP BY geo_to_region(latitude, longitude),
         vehicle_type,
         HOP(ts, INTERVAL '1' SECOND, INTERVAL '30' SECOND);
```

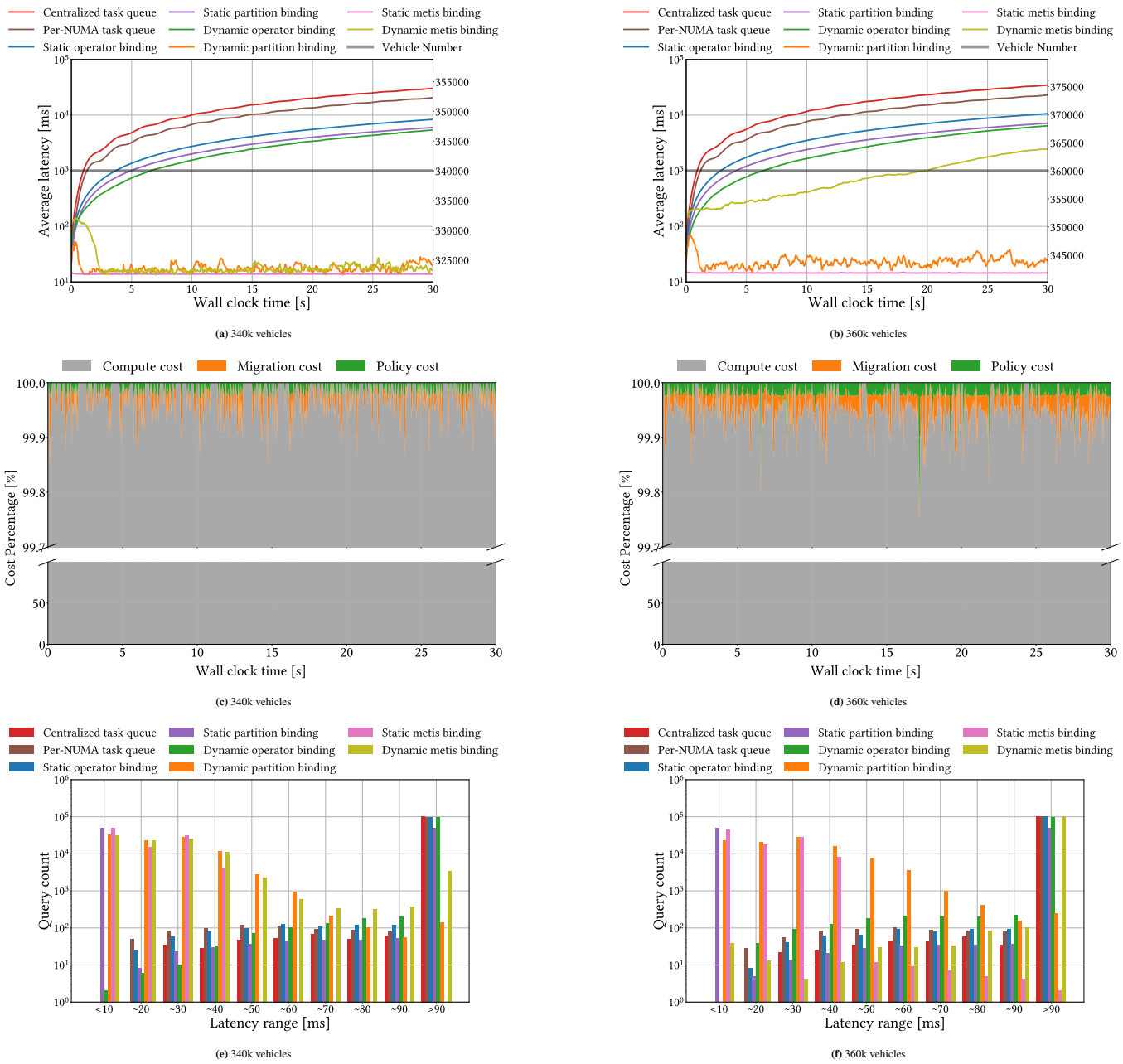
Query 1 Regional average speed

```
SELECT geo_to_region(latitude, longitude) AS region_id,
       HOP_END(ts, INTERVAL '1' SECOND, INTERVAL '10' SECOND) AS window_end,
       COUNT(*) AS vehicle_count
FROM (
  SELECT ts, latitude, longitude
  FROM vehicle_stream
  WHERE latitude IS NOT NULL AND longitude IS NOT NULL
)
GROUP BY geo_to_region(latitude, longitude),
         HOP(ts, INTERVAL '1' SECOND, INTERVAL '10' SECOND);
```

Query 2 Regional vehicle count

```
SELECT *
FROM (
  SELECT *,
         ROW_NUMBER() OVER (
           PARTITION BY region_id, vehicle_type, HOP(ts, INTERVAL '1' SECOND, INTERVAL '20'
               SECOND)
           ORDER BY speed DESC
         ) AS rank
  FROM (
    SELECT
      ts,
      vehicle_type,
      speed,
      geo_to_region(latitude, longitude) AS region_id,
      HOP(ts, INTERVAL '1' SECOND, INTERVAL '20' SECOND) AS window_start
    FROM vehicle_stream
    WHERE latitude IS NOT NULL AND longitude IS NOT NULL
  )
)
WHERE rank <= 3;
```

Query 3 Regional Top-3 speed



(a) 340k vehicles **(b)** 360k vehicles **(c)** 340k vehicles **(d)** 360k vehicles **(e)** 340k vehicles **(f)** 360k vehicles

FIGURE 2 Performance results of different task queue designs and operator binding policies under **workload 1** on a dual-socket **24-core** platform. Columns correspond to vehicle numbers of 340k (with $a = 309, r = 0.2$) and 360k (with $a = 327, r = 0.2$). Rows show (top) average query latency over wall-clock time, (middle) cost breakdown, and (bottom) query latency distribution.

309, $r = 0.2$) and 360,000 (with $a = 327, r = 0.2$) vehicles, respectively. The x-axis denotes the wall-clock time in seconds, while the y-axis (log scale) reports the average query latency in milliseconds.

The centralized task queue exhibits the worst performance, with latency rapidly increasing to tens of seconds due to contention on a single global queue. The per-NUMA task queue performs slightly better by reducing contention and improving locality, but still degrades over time due to persistent contention on shared resources. Within the per-thread queue design, both static operator binding and dynamic operator binding perform poorly. Static binding suffers from severe load imbalance, with more than 40% of the cores remaining underutilized. Although dynamic operator binding adapts to workload changes, it lacks NUMA awareness and inter-operator locality considerations, resulting in increased memory access latency and synchronization overhead.

In contrast, dynamic METIS binding, dynamic partition binding, and static METIS binding show strong performance in the 340, 000-vehicle scenario. Dynamic partition binding maintains low latency by rebalancing operators at the partition level, considering NUMA locality. Static METIS binding generates a high-quality initial operator-to-thread mapping using workload-aware graph partitioning, which remains effective due to the static nature of the workload. However, as shown in Figure 2b, increasing the total vehicle number to 360, 000 reveals further distinctions. Dynamic METIS binding begins to degrade in this high-load setting. This is because METIS only uses accumulated operator computation cost and processed data volume for partitioning, and it does not consider the instantaneous backlog in task queues. Under heavy load, these stale metrics lead to poor partitioning decisions and suboptimal performance. In contrast, under moderate load (Figure 2a), backlogs remain minimal, making the accumulated statistics more representative and yielding effective rebinding.

Notably, static METIS binding slightly outperforms dynamic partition binding in both workloads. This is due to two key factors: first, dynamic policies incur overhead for measuring workload, computing rebindings, and executing operator migrations, all of which become costly at scale. Second, in this workload, the partition-level load remains static over time, meaning that a well-optimized initial mapping can remain effective throughout execution. METIS, as a global graph partitioning algorithm, is particularly well-suited to such scenarios, as it minimizes inter-operator communication and balances computational load without runtime cost.

Figures 2c and 2d illustrate the cost breakdown when using the proposed policy with the non-blocking operator rebinding mechanism in the per-thread task queue design under workload 1. The figure decomposes the total execution cost into computation cost, migration cost, and policy cost over wall-clock time. In both workload settings, the computation cost dominates the overall execution time, accounting for more than 99.8% of the total cost throughout the experiment. This indicates that the additional overhead introduced by runtime rebinding remains negligible compared to the cost of query execution. The migration cost remains consistently low, reflecting the fine-grained and non-blocking nature of the operator rebinding mechanism, which avoids global synchronization and minimizes disruptions to ongoing execution. The cost of the policy is similarly small and stable over time, demonstrating that workload monitoring and rebinding decisions can be performed efficiently without imposing noticeable overhead. Even in the higher-load configuration with 360, 000 vehicles, the combined migration and policy overhead remains below 0.2% of the total execution cost. Additionally, the policy execution and operator migration are not continuously triggered. The rebinding policy is activated only when the query latency exceeds a predefined threshold, which is set to $30ms$ in our cases. Consequently, during periods of stable execution when latency remains below this threshold, neither policy nor migration cost is incurred, as indicated by the intervals in both figures. This event-driven behavior avoids unnecessary rebindings, preserves operator locality, and further reduces runtime overhead.

Figures 2e and 2f show the distribution of query latencies under workload 1. The x-axis represents latency buckets in $10ms$ increments from less than $10ms$ to more than $90ms$, and the y-axis (log scale) indicates the number of queries that fall into each latency range. This figure complements the average latency plot by highlighting overall trends and the consistency of performance across all queries. Dynamic partition binding and static METIS binding exhibit the most favorable latency profiles in both scenarios. In the 340,000 vehicle case (Figure 2e), over 98.5% of queries served by dynamic partition binding complete in under $60ms$, and 99.3% complete within $90ms$. Static METIS binding performs better, with 100% of queries below $60ms$. Dynamic METIS binding, on the other hand, performs acceptably in the lower-load case but already shows signs of degradation, with more queries falling into the higher latency bins compared to the other two. It serves approximately 95.5% of queries under $60ms$ and approximately 96.5% under $90ms$. Under the higher-load configuration (360,000 vehicles in Figure 2f), the distinction becomes clearer. Static METIS and dynamic partition binding remain the only two policies capable of maintaining consistent low-latency performance. Both deliver over 96% of queries within $60ms$ and nearly 100% within $90ms$, showcasing strong scalability. The remaining, including the centralized task queue, the per-NUMA queue, the static operator binding, and the dynamic operator binding, exhibit significantly broader latency distributions, with a significant number of queries falling into the $> 60ms$ and $> 90ms$ bins. These results further highlight their inability to handle imbalanced workloads or to exploit architectural features.

We next evaluate the same set of policies on a larger dual-socket platform with 36 physical cores. This evaluation serves two purposes. First, we assess how observed performance trends change when additional computational resources are available, while keeping the workload identical to the 24-core configuration (Figure 3). Second, we examine system behavior under increased workload intensity on the 36-core platform to evaluate robustness under higher degrees of parallelism and load (Figures 4).

We begin with a configuration that uses the same workload as the 24-core platform. As shown in Figures 3a and 3b, the centralized and per-NUMA task queue designs continue to exhibit rapidly increasing latency, indicating that contention in shared queues remains a dominant bottleneck despite additional cores. Within the per-thread queue design, dynamic operator binding

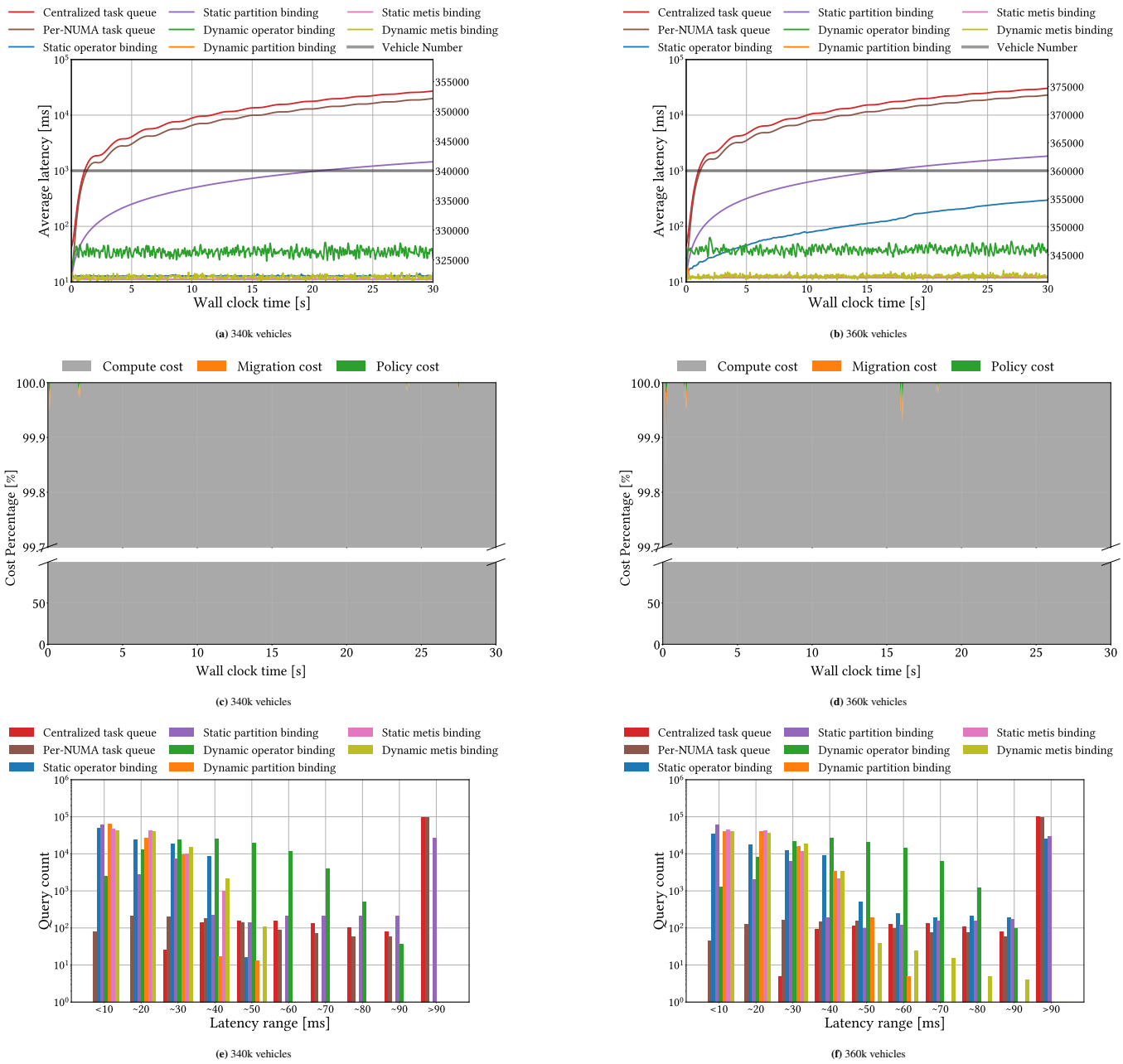
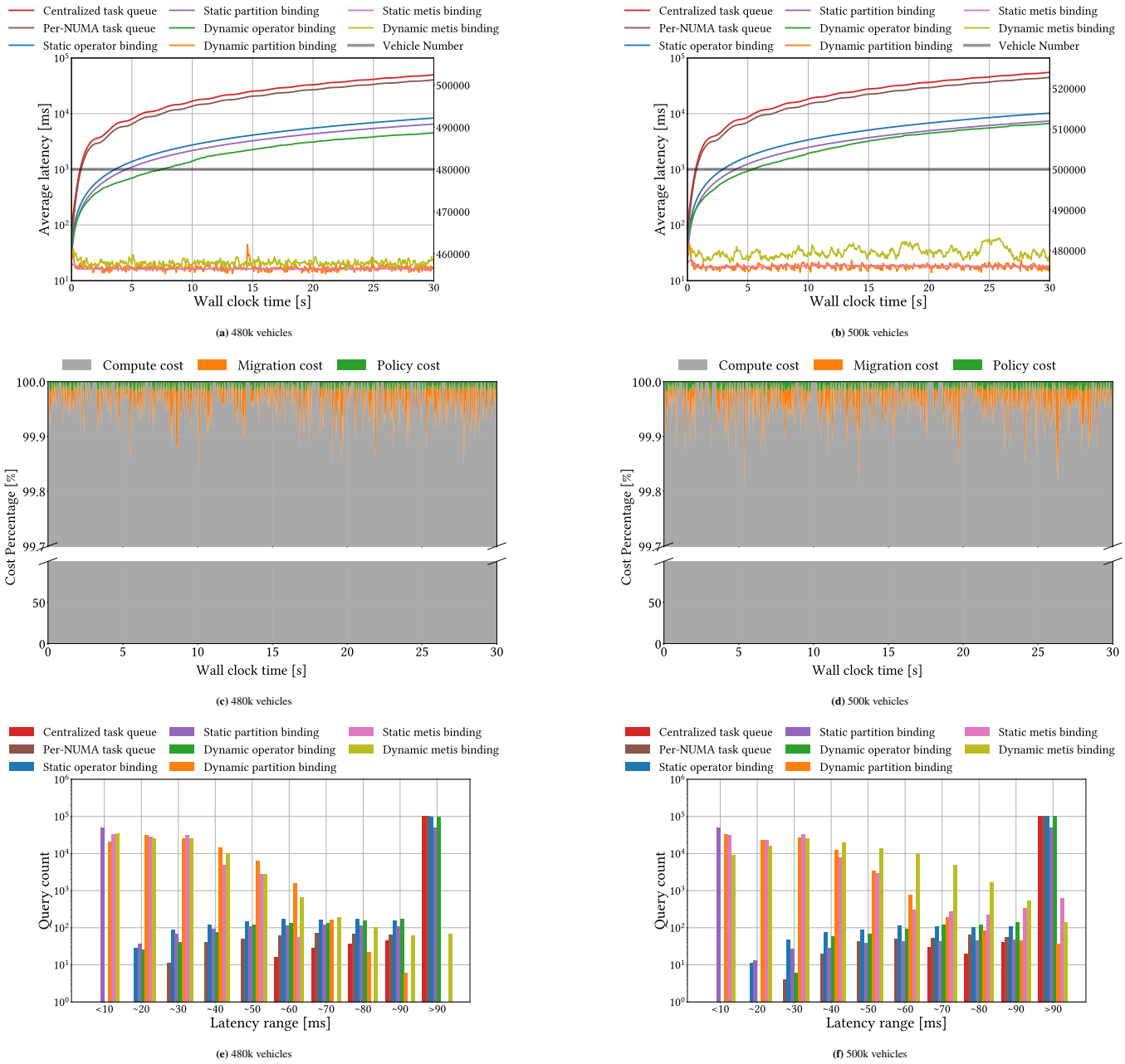


FIGURE 3 Performance results of different task queue designs and operator binding policies under **workload 1** on a dual-socket **36-core** platform. The vehicle numbers are identical to those used in the 24-core platform, enabling a direct comparison of scalability with increasing core counts.

shows improved stability on the 36-core platform, as the additional cores enable more effective load redistribution and prevent the severe degradation observed on the 24-core platform. However, due to its lack of NUMA and inter-operator locality awareness, its performance remains inferior to locality-aware approaches. In contrast, static operator binding performs well under 340,000 vehicle workload, but degrades at 360,000 vehicles, reflecting its inability to adapt to increased per-partition workload intensity. Consistent with the 24-core results, dynamic METIS binding, static METIS binding, and our policy maintain low and stable latency on the 36-core system. Their performance trends and relative ordering remain unchanged, demonstrating robustness as core counts increase.

Figures 3c and 3d present the cost breakdown of our policy on the 36-core platform. With additional computational resources available, most tasks are completed within their latency threshold. As a result, policy execution and operator migration occur less



(a) 480k vehicles **(b)** 500k vehicles **(c)** 480k vehicles **(d)** 500k vehicles **(e)** 480k vehicles **(f)** 500k vehicles

FIGURE 4 Performance results of different task queue designs and operator binding policies under **workload 1** on a dual-socket**36-core** platform, where the vehicle number is increased to assess scalability under higher workload intensity. Columns correspond to vehicle numbers of 480k (with $a = 436, r = 0.2$) and 500k (with $a = 454, r = 0.2$). Rows show (top) average query latency over wall-clock time, (middle) cost breakdown, and (bottom) query latency distribution.

frequently than in the 24-core configuration, reflecting improved headroom rather than changes in policy behavior. The latency distribution on the 36-core platform largely mirrors the trends observed on the 24-core system (Figures 3e and 3f). The designs of the centralized and per-NUMA task queue continue to exhibit broad distributions with significant tail latency, while dynamic operator binding shows a narrower distribution than in the case of 24-cores. Static operator binding remains stable under lower workloads but develops a visible tail at higher loads. In contrast, dynamic METIS binding, dynamic partition binding, and static METIS binding preserve compact latency distributions, consistent with the 24-core results. Figures 4 show the results for 480,000 (with $a = 436, r = 0.2$) and 500,000 (with $a = 454, r = 0.2$) vehicles, respectively. Under increased workload, both the average latency trends and the latency distributions remain largely consistent with those observed on the 24-core platform.

Compared to the lower-load configuration, we observed more frequent policy activation intervals in this experiment, reflecting higher system utilization and increased pressure on execution resources. Nevertheless, the proposed policy continues to adapt effectively, maintaining stable performance despite the increased workload.

5.4.2 | Workload 2: Static Total Load with Dynamic Partition Load

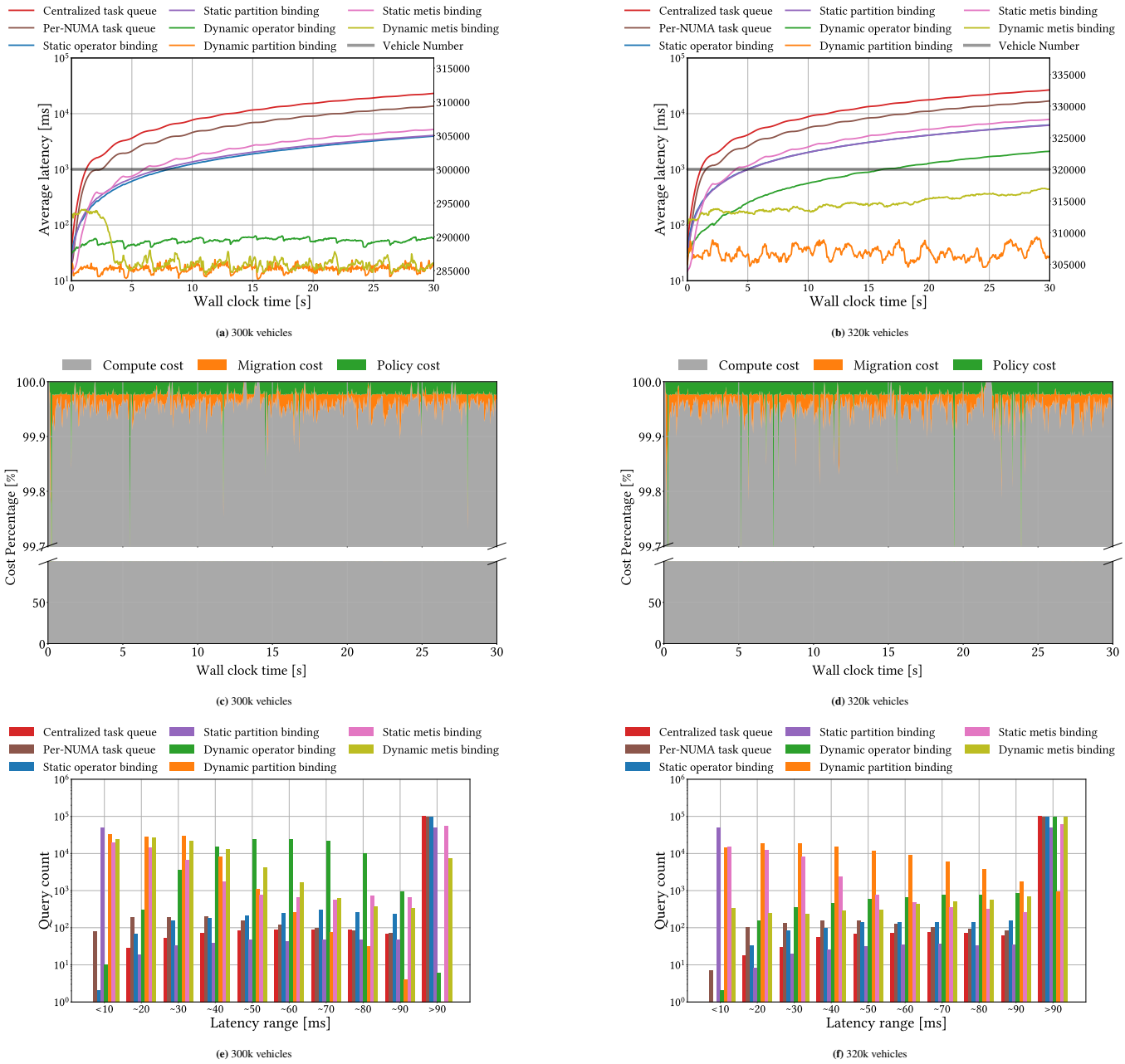
The second workload configuration introduces dynamic spatial load by redistributing vehicles across regions over time, while maintaining a constant total number of vehicles. This setup reflects scenarios in which vehicles frequently cross regional boundaries, resulting in shifting spatial demand. Starting from the same vehicle-to-region distribution as in workload 1, vehicles are cyclically shifted among regions at each timestep, creating evolving regional workloads without changing the global vehicle count. For example, with three partitions and $1,000$ vehicles, the distribution transitions as (250, 350, 400), (400, 250, 350), and (350, 400, 250). This workload allows us to evaluate how effectively different operator binding policies respond to dynamic spatial imbalance while isolating the effects of redistribution from changes in overall workload.

Compared to workload 1, static METIS binding performs noticeably worse in this setting (Figures 5a and 5b). While it was highly effective under static workloads, it fails to handle dynamically shifting regional loads. This is because it computes a one-time operator-to-thread mapping based on the initial workload distribution. As vehicle counts per region fluctuate over time, the static mapping becomes misaligned, causing thread-level load imbalance and significantly higher latency. Without runtime adaptation, static METIS is unable to respond to evolving workload patterns. In contrast, dynamic operator binding shows strong adaptability in the $300,000$-vehicle scenario. Although it lacks NUMA awareness and does not exploit operator locality, it effectively balances computational load across threads, allowing this policy to perform better than several static alternatives. Dynamic METIS binding outperforms dynamic operator binding even further. It considers both computation load and operator locality when computing rebindings at runtime. Its average latency is very close to that of dynamic partition binding in the $300,000$-vehicle case. However, the latency line shows more noticeable oscillations due to the overhead and coarseness of global partitioning operations, which may lag or overcorrect when the workload shifts rapidly. Dynamic partition binding delivers the best performance in both configurations. Especially in the $320,000$-vehicle scenario, it consistently maintains low and stable latency. Our policy stands out because it is lightweight and responsive. Not only does it consider operator computational cost, but it also incorporates task queue backlog and data locality into its decision-making. These combined factors allow it to quickly and accurately adjust bindings to maintain balanced execution with minimal overhead.

The cost breakdown results (Figures 5c and 5d) closely mirror those observed in the workload 1. In both configurations, the computation cost dominates the execution time, accounting for more than 99.7% of the total cost. The overhead introduced by policy execution and operator migration remains negligible, consistently below 0.3%, confirming the efficiency of the proposed rebinding mechanism even under dynamic workloads. The query latency distribution (Figures 5e and 5f) shows that under both scenarios, dynamic partition binding maintains a strong lead with the majority of queries falling in the low-latency bins ($< 90ms$), even as the total number of vehicles increases. The distribution remains tight and stable, confirming its ability to quickly adapt to changing regional workloads with minimal overhead.

We next repeat workload 2 on the dual-socket 36-core platform. As shown in Figures 6a and 6b, with increased computational resources, static operator binding is able to maintain low latency in this setting. Further inspection indicates that, on the 36-core system, the workload is relatively well balanced across threads, allowing the static mapping to remain effective. Nevertheless, the proposed approach continues to achieve the best overall performance in both configurations. Policy activation remains infrequent, as query latency stays below the activation threshold for most of the execution (Figures 6c and 6d). As reflected in the latency distributions (Figures 6e and 6f), several policies exhibit favorable latency profiles in this low-stress regime. Consequently, under this setting the benefits of dynamic rebinding are less pronounced, since the workload intensity is insufficient to trigger frequent rebindings and most queries complete within the latency threshold.

Then we increase the vehicle counts to 420,000 and 430,000 on the 36-core platform (Figure 7). Under these conditions, only dynamic partition binding maintains stable and low latency, while dynamic METIS binding degrades and eventually breaks at 430,000 vehicles. Cost breakdown and latency distributions (Figures 7c - 7f) confirm negligible overhead and stable tail behavior. Overall, the observed trends are consistent with the 24-core results, demonstrating the robustness of the proposed policy under higher load and increased parallelism.



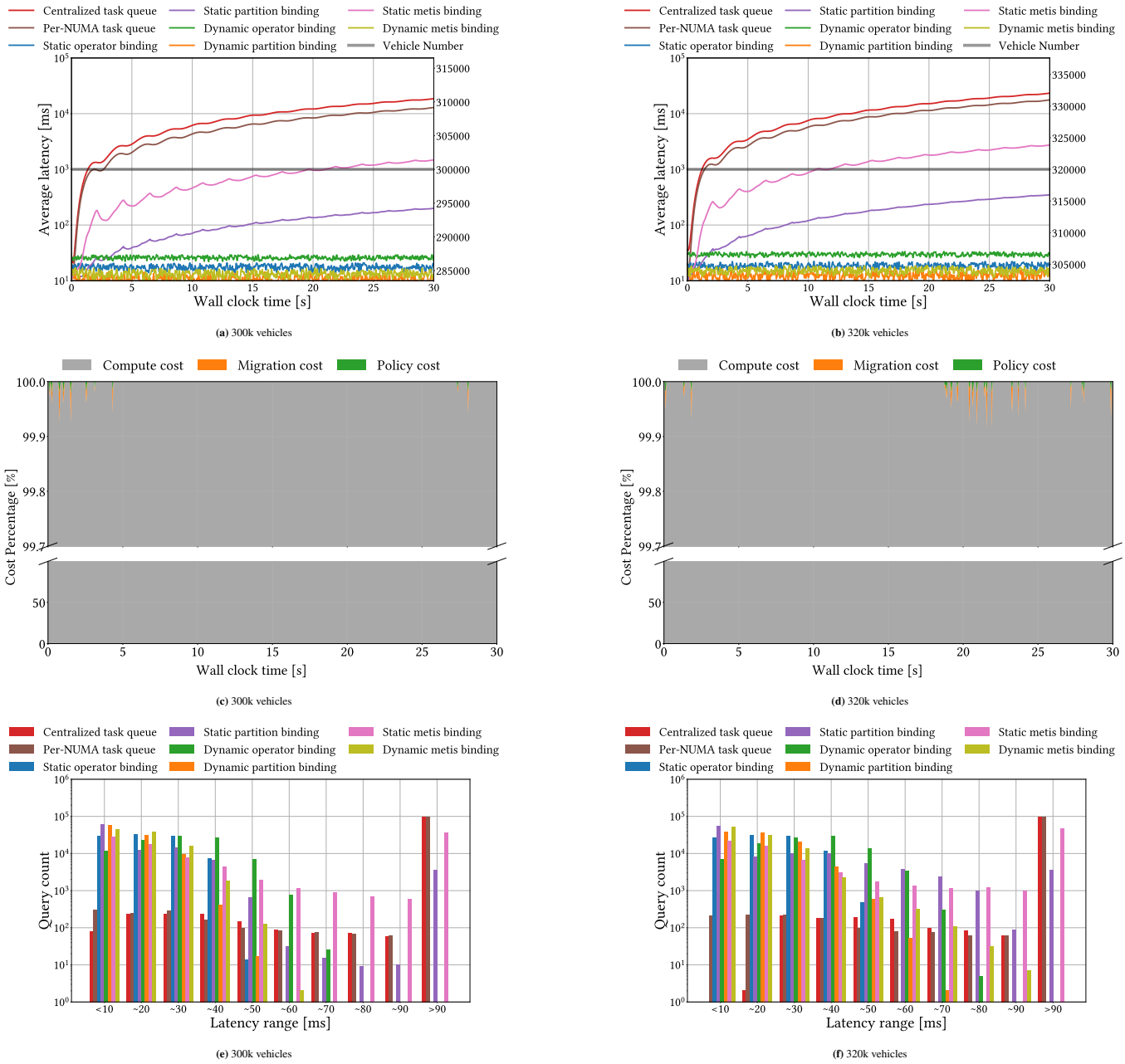
(a) 300k vehicles **(b)** 320k vehicles **(c)** 300k vehicles **(d)** 320k vehicles **(e)** 300k vehicles **(f)** 320k vehicles

FIGURE 5 Performance results of different task queue designs and operator binding policies under **workload 2** on a dual-socket **24-core** platform. Columns correspond to vehicle numbers of 300k (with $a = 275, r = 0.2$) and 320k (with $a = 293, r = 0.2$). Rows show (top) average query latency over wall-clock time, (middle) cost breakdown, and (bottom) query latency distribution.

5.4.3 | Workload 3: Dynamic Total Load with Dynamic Partition Load

In this configuration, we replay rush-hour traffic data for Shenzhen, China, where both the total number of vehicles and their distribution across regions vary over time. This results in a constantly changing workload intensity, making it the most challenging of the three workloads.

Figures 8a and 8b show simulations peaking at 350,000 and 400,000 vehicles, respectively. In both cases, the gray curve indicates the instantaneous vehicle count and highlights rapid workload fluctuations over time. At the beginning of execution, when the vehicle count is low, all execution models and operator binding policies are able to maintain low query latency. As the vehicle count increases, performance divergence becomes apparent. The centralized task queue design degrades first, followed



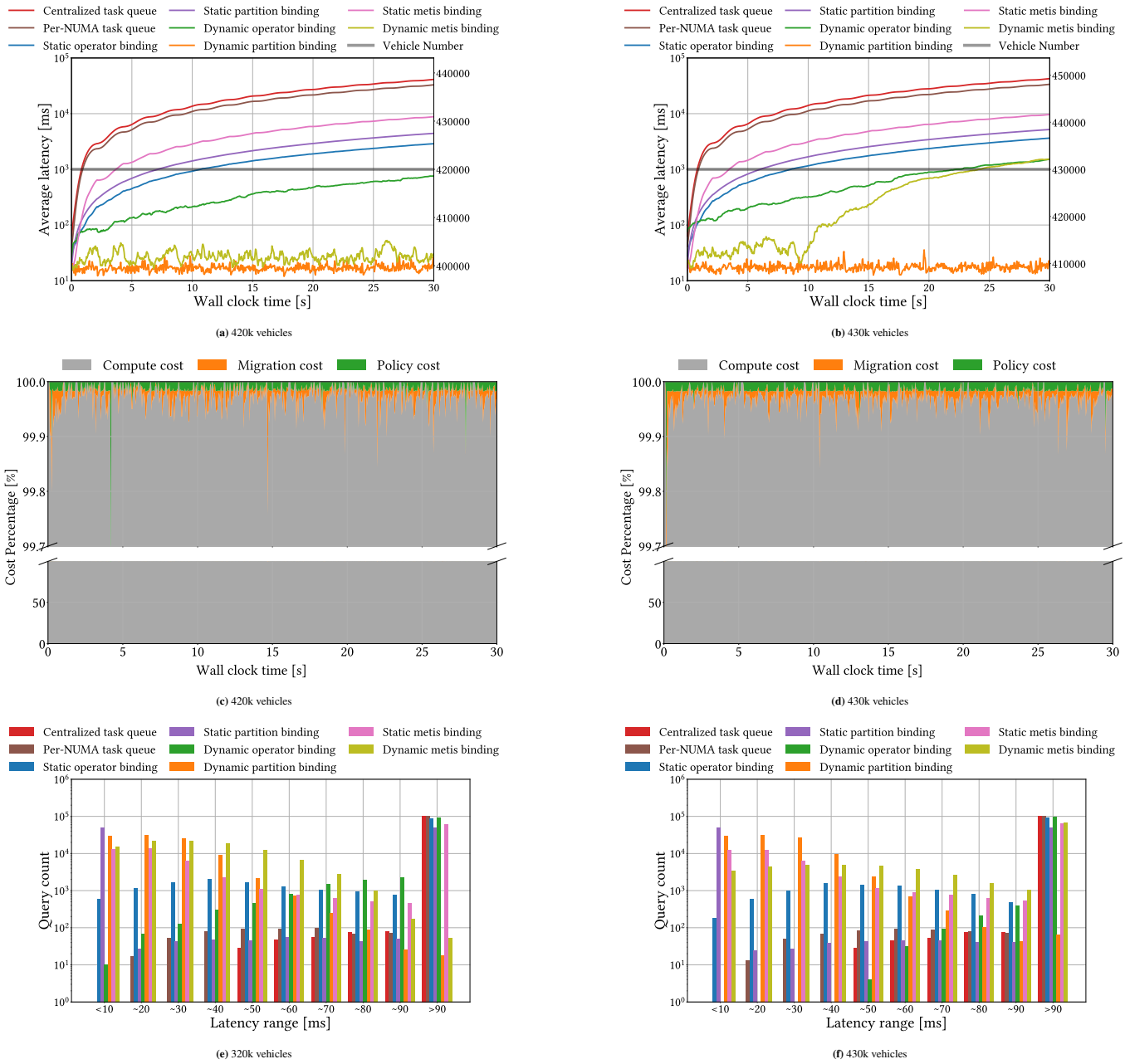
(a) 300k vehicles (b) 320k vehicles

(c) 300k vehicles (d) 320k vehicles

(e) 300k vehicles (f) 320k vehicles

FIGURE 6 Performance results of different task queue designs and operator binding policies under **workload 2** on a dual-socket **36-core** platform. The vehicle numbers are identical to those used in the 24-core platform, enabling a direct comparison of scalability with increasing core counts.

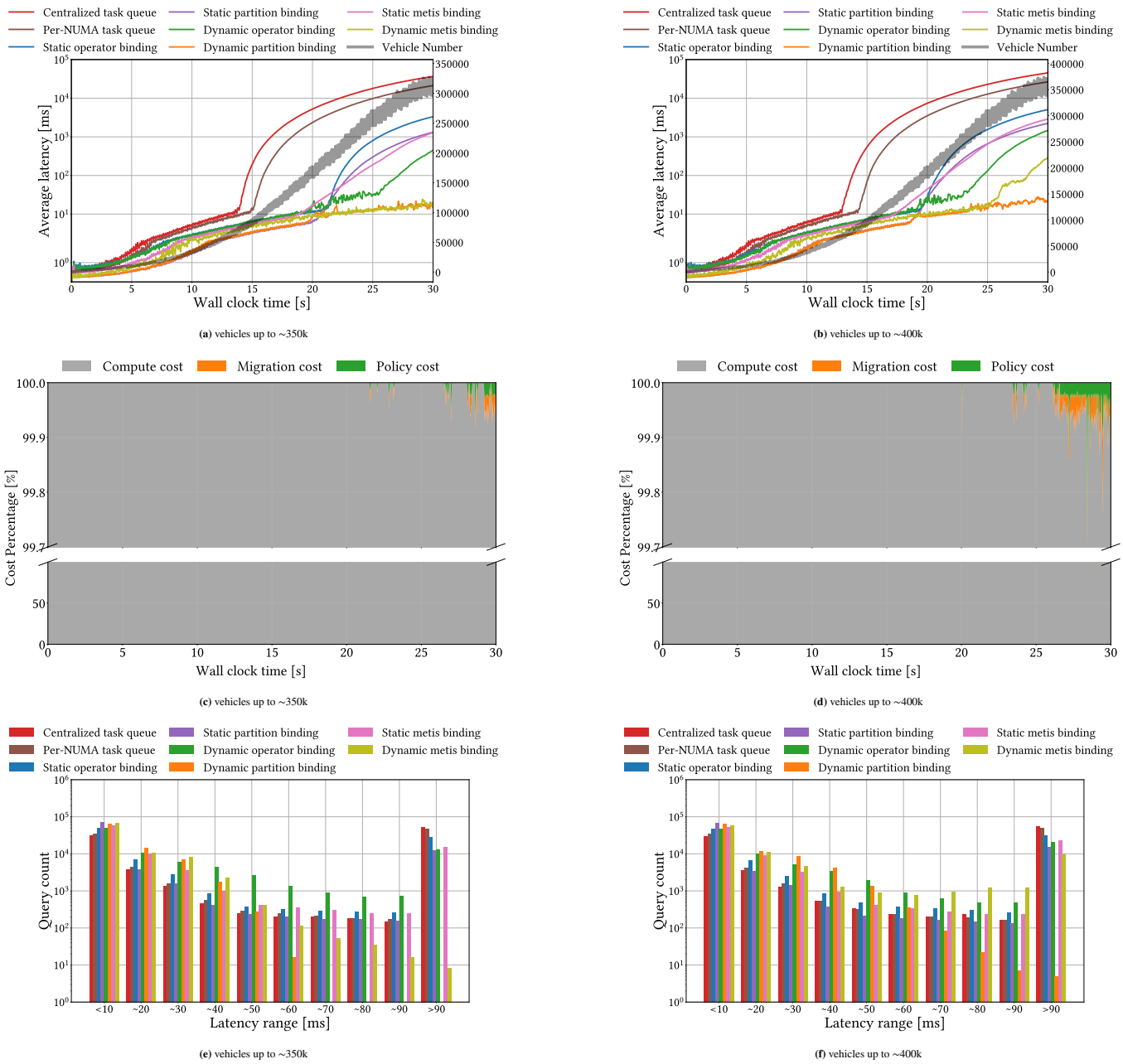
by the per-NUMA task queue, which is expected given their reliance on shared task queues and susceptibility to contention under increasing parallelism. Static binding strategies, including static operator binding and static METIS binding, also degrade significantly as the workload evolves, since fixed mappings quickly become misaligned with the continuously changing spatial demand. Compared to workloads 1 and 2, this effect is amplified in workload 3 due to the combined variation in overall load and persistent redistribution of vehicles across regions. Among adaptive approaches, dynamic operator binding shows moderate adaptability in the early phase, but begins to degrade as vehicle counts exceed 150, 000. Its inability to account for NUMA locality and lack of awareness of the operator graph structure lead to rising latency. Dynamic METIS binding performs better by capturing both load and locality during runtime rebalancing in Figure 8a. However, its performance still fluctuates significantly as the workload increases, as shown in Figure 8b. This instability stems from the coarse-grained and costly nature of its global



(a) 420k vehicles **(b)** 430k vehicles **(c)** 420k vehicles **(d)** 430k vehicles **(e)** 320k vehicles **(f)** 430k vehicles

FIGURE 7 Performance results of different task queue designs and operator binding policies under **workload 2** on a dual-socket**36-core** platform, where the vehicle number is increased to assess scalability under higher workload intensity. Columns correspond to vehicle numbers of 420k (with $a = 381, r = 0.2$)and 430k(with $a = 390, r = 0.2$). Rows show (top) average query latency over wall-clock time, (middle) cost breakdown, and (bottom) query latency distribution.

partitioning, which can lag behind rapid workload shifts. In addition, it does not consider the backlog in task queues, leading to suboptimal partitioning decisions. Dynamic partition binding again stands out as the best-performing strategy across both vehicle scenarios. It maintains sub-50ms average latency up to approximately 200, 000 vehicles and scales smoothly as the load grows beyond that. Even as the system approaches 400, 000 vehicles in Figure 8b, it remains the most stable and least-latency configuration. Its lightweight, fine-grained design allows it to respond quickly to changes in both workload volume and spatial distribution. By incorporating operator computation cost, task queue backlog, and data locality, it achieves a more holistic and responsive rebalancing approach.



(a) vehicles up to ~350k

(b) vehicles up to ~400k

(c) vehicles up to ~350k

(d) vehicles up to ~400k

(e) vehicles up to ~350k

(f) vehicles up to ~400k

FIGURE 8 Performance results of different task queue designs and operator binding policies under **workload 3** on a dual-socket **24-core** platform. Columns correspond to increasing vehicle counts up to ~350k and ~400k, respectively. Rows show (top) average query latency over wall-clock time, (middle) cost breakdown, and (bottom) query latency distribution. **workload 3** represents rush-hour traffic traces with continuously varying total workload and spatial distribution, introducing both temporal and spatial non-stationarity.

Figures 8c and 8d show the cost breakdown for workload 3. At the beginning of execution, the workload is light, and the query latency remains below the activation threshold of 30ms. Consequently, the rebinding policy is inactive, and execution cost is dominated by query computation. As workload intensity increases and latency rises, the policy is activated to rebalance operator placement. This activation occurs more frequently in the higher-load scenario with a peak of 400,000 vehicles, reflecting greater system pressure and faster workload fluctuations. Despite more frequent rebindings, the combined policy and migration overhead remains negligible (below 0.3%), demonstrating that the proposed approach can adapt efficiently to highly dynamic workloads while maintaining low runtime overhead.

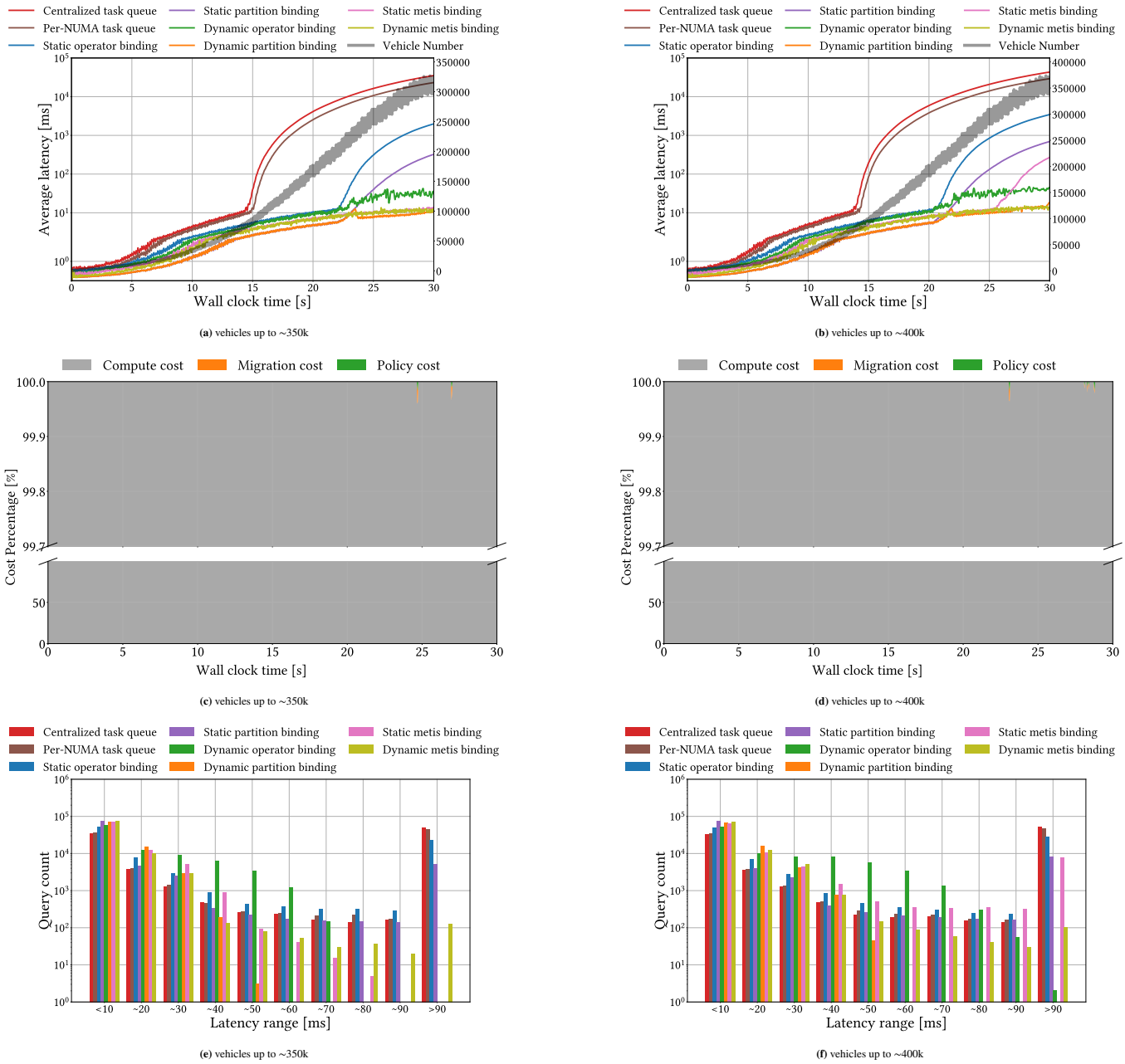
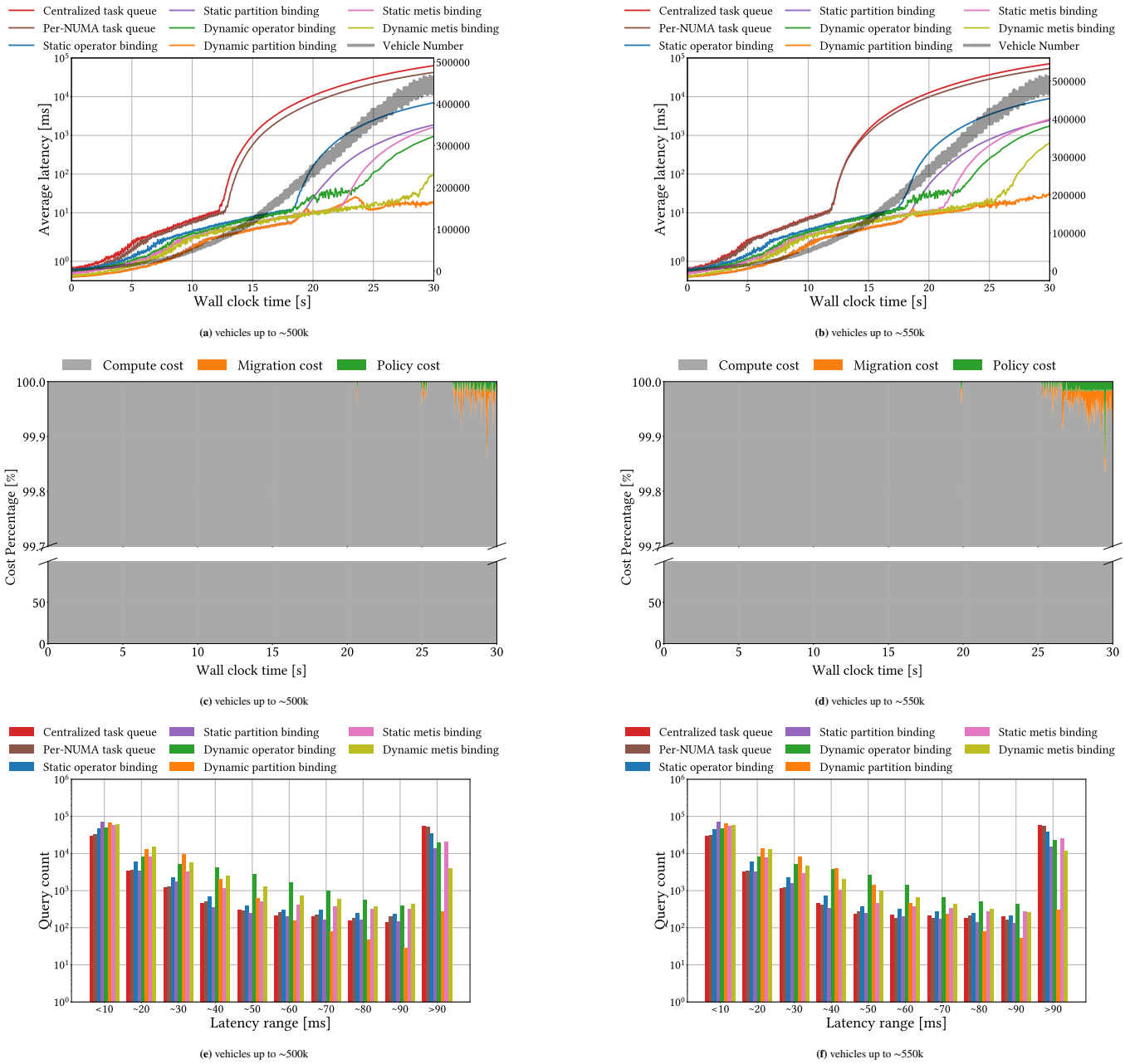


FIGURE 9 Performance results of different task queue designs and operator binding policies under **workload 3** on a dual-socket **36-core** platform. The vehicle numbers are identical to those used in the 24-core platform, enabling a direct comparison of scalability with increasing core counts.

Figures 8e and 8f show the query latency distribution. Dynamic partition binding again achieves the most favorable profile, keeping most queries below 60*ms* and showing minimal tail latency, even under highly dynamic conditions. Dynamic METIS binding maintains comparable performance in lower latency ranges but continues to suffer from a broader distribution beyond 60*ms*, similar to what was observed in workload 2. Dynamic operator binding remains moderately effective, but again incurs higher tail latency. Static strategies, including static METIS and static operator binding, result in a large number of queries falling in the $> 90ms$ bin, further reflecting their limited adaptability to shifting workloads, as also seen in earlier scenarios.

Figures 9 show the results by running the same workload configuration on the 36-core platform. With additional computational resources available, dynamic operator binding exhibits improved stability and no longer breaks in either workload setting. Static METIS binding also benefits from the increased core count, showing lower latency and a later onset of degradation compared to



(a) vehicles up to ~500k

(b) vehicles up to ~550k

(c) vehicles up to ~500k

(d) vehicles up to ~550k

(e) vehicles up to ~500k

(f) vehicles up to ~550k

FIGURE 10 Performance results of different task queue designs and operator binding policies under **workload 3** on a dual-socket**36-core** platform, where the vehicle number is increased to assess scalability under higher workload intensity. Columns correspond to vehicle numbers up to ~500k and ~550k. Rows show (top) average query latency over wall-clock time, (middle) cost breakdown, and (bottom) query latency distribution.

the 24-core results. The cost breakdown in this setting remains consistent with earlier observations, with query latency staying below the activation threshold for most of the execution, and the proposed policy is therefore rarely triggered.

Stability and Worst-Case Behavior

A critical concern in dynamic scheduling is the system's behavior under rapid workload oscillations that exceed the controller's reaction speed. A potential failure mode in this context is *thrashing*, where the scheduler continuously rebinds operators based on stale state, consuming resources without improving locality. Our approach mitigates this via two mechanisms: first, the latency threshold δ acts as a hysteresis filter, preventing reactions to negligible jitter. Second, the lock-free migration mechanism ensures

that even in a worst-case scenario of frequent rebinding, the overhead remains bounded and low (< 0.3% in our experiments), preventing the throughput collapse observed in blocking approaches, such as Dynamic METIS. Regarding jitter, the latency distribution tails (Figures 8e and 8f) confirm that our fine-grained, localized adjustments prevent the massive latency spikes associated with global repartitioning, effectively containing worst-case jitter even under the rapid oscillations of Workload 3.

When we further increase the workload on the 36-core platform, with peak vehicle counts of 500,000 and 550,000, we observe performance trends (Figure 10) consistent with those seen in the corresponding experiments on the 24-core system (Figure 8). Among all evaluated approaches, only dynamic partition binding sustains stable execution, achieving the lowest latency and the most controlled tail behavior despite the substantially higher load.

5.4.4 | Summary

Across all workloads, the evaluation we have carried out shows that execution models with fine-grained control (per-thread queues) combined with adaptive, locality-aware binding policies perform best. In particular, our **dynamic partition binding** stands out for its lightweight design, responsiveness to both load and data locality, and consistent low-latency performance. While dynamic METIS is competitive in moderate conditions, it struggles under heavier loads due to its coarse, global rebalancing. Static strategies, though efficient in fixed workloads, fail to cope with runtime variability. These results highlight the importance of runtime adaptability and architectural awareness in designing scalable, low-latency stream processing systems.

6 | CONCLUSIONS

We have proposed a 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.

ACKNOWLEDGEMENTS

This paper has been partially supported by the European Union—Next Generation EU, Mission 4, Component 2, CUP E53D23008200006 (Domain).

References

1. Drepper Ulrich. *What Every Programmer Should Know About Memory.* : Red Hat, Inc.; 2007.
2. Lang Harald, Leis Viktor, Albutiu Martina-Cezara, Neumann Thomas, Kemper Alfons. Massively parallel NUMA-aware hash joins. In: *In Memory Data Management and Analysis*. Lecture notes in computer science. Cham: Springer International Publishing 2015 (pp. 3–14).
3. Balkesen Cagri, Teubner Jens, Alonso Gustavo, Ozsu M Tamer. Main-memory hash joins on modern processor architectures. *IEEE transactions on knowledge and data engineering.* 2015;27(7):1754–1766.
4. Ghoting A, Parthasarathy S. Facilitating interactive distributed data stream processing and mining. In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings*. IEEE 2004 (pp. 86).
5. Castro Fernandez Raul, Migliavacca Matteo, Kalyvianaki Evangelia, Pietzuch Peter. Integrating scale out and fault tolerance in stream processing using operator state management. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM 2013.
6. Gedik Bugra, Schneider Scott, Hirzel Martin, Wu Kun-Lung. Elastic scaling for data stream processing. *IEEE transactions on parallel and distributed systems: a publication of the IEEE Computer Society.* 2014;25(6):1447–1463.

7. Schneider Scott, Wu Kun-Lung. Low-synchronization, mostly lock-free, elastic scheduling for streaming runtimes. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM 2017.

8. Du Xiaorui, Piccione Andrea, Pimpini Adriano, Bortoli Stefano, Knoll Alois, Pellegrini Alessandro. HUILLY: A Non-Blocking Ingestion Buffer for Timestepped Simulation Analytics. In: *Proceedings of the 24th International Symposium on Cluster, Cloud and Grid Computing*. CCGrid. IEEE 2024.

9. Du Xiaorui, Piccione Andrea, Pimpini Adriano, Bortoli Stefano, Pellegrini Alessandro, Knoll Alois. Online analytics with local operator rebinding for simulation data stream processing. In: *2024 28th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE 2024 (pp. 64–73).

10. Volnes Espen, Plagemann Thomas, Goebel Vera. To migrate or not to migrate: An analysis of operator migration in distributed stream processing. *IEEE Communications Surveys & Tutorials.* 2023:1–1.

11. Karypis George, Kumar Vipin. Multilevel algorithms for multi-constraint graph partitioning. In: *Proceedings of the 1998 IEEE/ACM High Performance Networking and Computing Conference*. SC'98. Piscataway, NJ, USA: IEEE 1998.

12. Jain Namit, Mishra Shailendra, Srinivasan Anand, et al. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases.* 2008;1(2):1379–1390.

13. Zeuch Steffen, Chatziliadis Xenofon, Chaudhary Ankit, et al. NebulaStream: Data management for the internet of Things. *Datenbank-Spektrum : Zeitschrift fur Datenbanktechnologie : Organ der Fachgruppe Datenbanken der Gesellschaft fur Informatik e.V.* 2022;22(2):131–141.

14. Deng Ze, Wu Xiaomin, Wang Lizhe, et al. Parallel processing of dynamic continuous queries over streaming data flows. *IEEE transactions on parallel and distributed systems: a publication of the IEEE Computer Society.* 2015;26(3):834–846.

15. Lindén Jonatan, Jonsson Bengt. A skiplist-based concurrent priority queue with minimal memory contention. In: *Lecture Notes in Computer Science*. Lecture notes in computer science. Cham: Springer International Publishing 2013 (pp. 206–220).

16. Marotta Romolo, Ianni Mauro, Pellegrini Alessandro, Quaglia Francesco. A Non-Blocking Priority Queue for the Pending Event Set. In: *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*. SIMUTOOLS. Brussels, Belgium: Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST) 2016 (pp. 46–55).

17. Marotta Romolo, Ianni Mauro, Pellegrini Alessandro, Quaglia Francesco. A conflict-resilient lock-free linearizable calendar queue. *ACM transactions on parallel computing.* 2024;11(1):1–32.

18. Eidenbenz Raphael, Locher Thomas. Task allocation for distributed stream processing. In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. IEEE 2016.

19. Cardellini Valeria, Grassi Vincenzo, Lo Presti Francesco, Nardelli Matteo. Optimal operator replication and placement for distributed stream processing systems. *Performance evaluation review.* 2017;44(4):11–22.

20. Lustosa Hermano, Porto Fabio, Valduriez Patrick, Blanco Pablo. Database system support of simulation data. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases.* 2016;9(13):1329–1340.

21. Foroni Daniele, Axenie Cristian, Bortoli Stefano, et al. 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 2018.

22. Qin Cui, Eichelberger Holger, Schmid Klaus. Enactment of adaptation in data stream processing with latency implications—A systematic literature review. *Information and software technology.* 2019;111:1–21.

23. Vogel Adriano, Griebler Dalvan, Danelutto Marco, Fernandes Luiz Gustavo. Self-adaptation on parallel stream processing: A systematic review. *Concurrency and computation: practice & experience.* 2022;34(6):e6759.

24. Koldehofe Boris, Mayer Ruben, Ramachandran Umakishore, Rothermel Kurt, Völz Marco. 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 2013.

25. Carbone Paris, Katsifomodis Asterios, Ewen Stephen, Markl Volker, Haridi Seif. Apache flink: Stream and batch processing in a single engine. *Bulletin of the Technical Committee on Data Engineering*. 2015;38(4):28–38.

26. Zhu Yali, Rundensteiner Elke A, Heineman George T. 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 2004.

27. Volnes Espen, Plagemann Thomas, Koldehofe Boris, Goebel Vera. 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 2022.

28. Del Monte Bonaventura, Zeuch Steffen, Rabl Tilmann, Markl Volker. 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 2020.

29. De Matteis Tiziano, Mencagli Gabriele. 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*. PPoPP '16. New York, NY, USA: ACM 2016 (pp. 1–12).

30. Madsen Kasper Grud Skat, Zhou Yongluan. 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 2015.

31. Gu Rong, Yin Han, Zhong Weichang, Yuan Chunfeng, Huang Yihua. Meces: Latency-efficient rescaling via prioritized state migration for stateful distributed stream processing systems. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Santa Clara, CA, USA: USENIX Association 2022 (pp. 539–556).

32. Lakshmanan Geetika T, Li Ying, Strom Rob. Placement strategies for internet-scale data stream systems. *IEEE internet computing*. 2008;12(6):50–60.

33. Buddhika Thilina, Stern Ryan, Lindburg Kira, Ericson Kathleen, Pallickara Shrideep. 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*. 2017;28(12):3553–3569.

34. Shah M A, Hellerstein J M, Chandrasekaran Sirish, Franklin M J. Flux: an adaptive partitioning operator for continuous query systems. In: *Proceedings 19th International Conference on Data Engineering*. DE '04. Piscataway, NJ, USA: IEEE 2004.

35. Ottenwälder Beate, Koldehofe Boris, Rothermel Kurt, Ramachandran Umakishore. 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 2013.

36. Luthra Manisha, Koldehofe Boris, Weisenburger Pascal, Salvaneschi Guido, Arif Raheel. 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 2018.

37. Jonathan Albert, Chandra Abhishek, Weissman Jon. WASP: Wide-area adaptive stream processing. In: *Proceedings of the 21st International Middleware Conference*. New York, NY, USA: ACM 2020.

38. Kalavri Vasiliki, Liagouris John, Hoffmann Moritz, Dimitrova Desislava, Forshaw Matthew, Roscoe Timothy. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. OSDI '18. Santa Clara, CA, USA: USENIX Association 2018 (pp. 783–798).

39. Pietzuch P, Ledlie J, Shneidman J, Roussopoulos M, Welsh M, Seltzer M. Network-aware operator placement for stream-processing systems. In: *22nd International Conference on Data Engineering (ICDE'06)*. IEEE 2006.

40. Gedik Buğra. Partitioning functions for stateful data parallelism in stream processing. *The VLDB journal: very large data bases: a publication of the VLDB Endowment.* 2014;23(4):517–539.

41. Hochreiner Christoph, Vogler Michael, Schulte Stefan, Dustdar Schahram. Elastic stream processing for the internet of things. In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE 2016.

42. Heinze Thomas, Jerzak Zbigniew, Hackenbroich Gregor, Fetzer Christof. 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 2014.

43. Rizou Stamatia, Durr Frank, Rothermel Kurt. Solving the multi-operator placement problem in large-scale operator networks. In: *2010 Proceedings of 19th International Conference on Computer Communications and Networks*. IEEE 2010.

44. Röger Henriette, Bhowmik Sukanya, Rothermel Kurt. 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 2019.

45. Lombardi Federico, Aniello Leonardo, Bonomi Silvia, Querzoni Leonardo. 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.* 2018;29(3):572–585.

46. De Matteis Tiziano, Mencagli Gabriele. Proactive elasticity and energy awareness in data stream processing. *The Journal of systems and software.* 2017;127:302–319.

47. Cardellini Valeria, Lo Presti Francesco, Nardelli Matteo, Russo Russo Gabriele. Decentralized self-adaptation for elastic Data Stream Processing. *Future generations computer systems: FGCS.* 2018;87:171–185.

48. Chatzimilioudis Georgios, Cuzzocrea Alfredo, Gunopulos Dimitrios, Mamoulis Nikos. A novel distributed framework for optimizing query routing trees in wireless sensor networks via optimal operator placement. *Journal of computer and system sciences.* 2013;79(3):349–368.

49. Zehe Daniel, Nair Suraj, Knoll Alois, Eckhoff David. Towards citymos: a coupled city-scale mobility simulation framework. In: Djanatliev Anatoli, Hielscher Kai-Steffen, German Reinhard, eds. *Proceedings of the 5th GI/ITG KuVS Fachgespräch Inter-Vehicle Communication*. Technical Reports, vol. CS-2017-03: Nuremberg, Germany: Friedrich-Alexander-Universität Erlangen-Nürnberg 2017 (pp. 26–28).

50. Schloegel Kirk, Karypis George, Kumar Vipin. A New Algorithm for Multi-objective Graph Partitioning. In: Patrick Amestoy, Philippe Berger, Michel Daydé, et al. , eds. *Euro-Par'99 Parallel Processing*. Lecture notes in computer science, vol. 1685: Berlin, Heidelberg: Springer Berlin Heidelberg 1999 (pp. 322–331).

51. Staelin Carl. Lmbench: An extensible micro-benchmark suite. *Software: practice & experience.* 2005;35(11):1079–1105.

52. Karypis George, Kumar Vipin. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices.* 97-061: Department of Computer Science and Engineering, University of Minnesota; 1997.

53. Xu Yadong, Cai Wentong, Aydt Heiko, Lees Michael. Efficient graph-based dynamic load-balancing for parallel large-scale agent-based traffic simulation. In: *Proceedings of the Winter Simulation Conference 2014*. IEEE 2014.

54. Du Xiaorui, Meng Zhuoxiao, Siguenza-Torres Anibal, et al. Autonomic orchestration of in-situ and in-transit data analytics for simulation studies. In: *2023 Winter Simulation Conference (WSC)*. IEEE 2023.

55. Meng Zhuoxiao, Du Xiaorui, Sottovia Paolo, et al. Topology-Preserving Simplification of OpenStreetMap Network Data for Large-scale Simulation in SUMO. In: *Proceedings of the 2022 SUMO User Conference*. Hannover, Germany: TIB Open Publishing 2022 (pp. 181–197).

56. Reinders James. *Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism*. Sebastopol, CA, USA: O'Reilly Media; 2007.

How to cite this article: Xiaorui Du, Andrea Piccione, Adriano Pimpini, Stefano Bortoli, Alessandro Pellegrini, et al. (2026), Operator Rebinding for Stream Processing on NUMA Machines, *Software: Practice and Experience*, *2026;ABC:1–2*. https://doi.org/10.1002/spe.70064