Processing Distributed Transactions in a Predefined Order

Pavan Poudel
Kent State University
Kent, Ohio
ppoudel@cs.kent.edu

Shishir Rai
Kent State University
Kent, Ohio
srai@cs.kent.edu

Gokarna Sharma
Kent State University
Kent, Ohio
sharma@cs.kent.edu

ABSTRACT

Consider distributed transactional memory systems where transactions residing at nodes of a communication graph operate on shared, mobile objects. A transaction requests the objects it needs, executes once those objects have been assembled, and then forwards those objects to other waiting transactions. We study the *predefined order scheduling* problem of committing transactions according to their priorities. This problem naturally arises in areas, such as loop parallelization and state-machine-based computing, where producing executions equivalent to a priority order is needed to satisfy certain properties. Specifically, we study predefined order scheduling considering two performance metrics fundamental to any distributed system: (i) *execution time* - total time to commit all the transactions and (ii) *communication cost* - the total distance messages travel. We design scheduling algorithms that are simultaneously efficient for both the metrics and rigorously evaluate them through several benchmarks on random and grid graphs, validating their efficiency. To the best of our knowledge, this is the first study of predefined order scheduling in distributed systems.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**; • **Theory of computation** → **Scheduling algorithms**; **Parallel algorithms**; **Online algorithms**.

KEYWORDS

Distributed systems; transactional memory; scheduling; predefined order; execution time; communication cost; conflicts

ACM Reference Format:
Pavan Poudel, Shishir Rai, and Gokarna Sharma. 2021. Processing Distributed Transactions in a Predefined Order. In *International Conference on Distributed Computing and Networking 2021 (ICDCN '21), January 5–8, 2021, Nara, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3427796.3427819

1 INTRODUCTION

Concurrent processes (threads) need to synchronize to avoid introducing inconsistencies while accessing shared data objects. Traditional synchronization mechanisms such as locks and barriers have well-known downsides, including deadlock, priority inversion, reliance on programmer conventions, and vulnerability to failure or delay. *Transactional memory* (TM) [14, 30] has emerged as an attractive alternative. Using TM, program code is split into *transactions*, blocks of code that appear to execute atomically. Transactions are executed *speculatively*: synchronization conflicts (or failures) may cause an executing transaction to *abort*: its effects are rolled back and the transaction is restarted. In the absence of conflicts (or failures), a transaction typically *commits*, causing its effects to become visible to all threads. Several commercial processors support TM, for example, (Intel) Haswell [17], (IBM) Blue Gene/Q [22], (IBM) zEnterprise EC12 [22], and (IBM) Power8 [8].

TM has been studied heavily in the past for *shared memory multi-core* systems where processing cores operate on a single shared memory and the latency to perform a memory access is the same for all the processors. However, the shared memory multi-core paradigm is shifting towards many-core to non-uniform memory access (NUMA) to more general distributed networked architectures, where the latency to perform a memory access varies depending on the processor in which the thread executes and the physical segment of memory that stores the requested memory location. The recent focus has been on how to support TM in *distributed shared memory* systems, incorporating memory access latency into analysis and evaluation. Some proposals include TM^{2}C for many-core systems [12], Nemo for NUMA systems [21], the systems in [3, 19] for clusters, the system in [10] for GPUs, and Hyflow [31] for distributed systems.

TM is beneficial in distributed computing platforms where the data is spread across the participating nodes. For example, distributed data centers can use TM to simplify the burden of distributed synchronization and provide more reliable and efficient concurrent program execution while accessing data in possibly remote compute nodes. The distributed TM designed on top of such systems needs to execute the transactions effectively by taking into consideration the system's infrastructure. Especially, the network structure can play a crucial role in the performance of the distributed TM, since the data that the transactions access has to be reached across the network in a timely manner.

In this paper, we study the *predefined order scheduling* problem, denoted as PredOrderScheduling, in a n-node connected, undirected, and weighted graph G, where each node denotes a processing node and each edge denotes a communication link between processing nodes. There are a set of w shared objects $\mathcal{S} := \{S_1, S_2, \ldots, S_w\}$ that reside on the nodes of G. We consider the *data-flow* model [15], where transactions execute at nodes where they were mapped (i.e., immobile) but the shared objects move to transactions (at graph nodes) needing them. PredOrderScheduling is defined as follows. Consider a set of transactions $\mathcal{T} := \{T(v_1, age_1), T(v_2, age_2), \ldots\}$ mapped (arbitrarily) to the

ICDCN '21, January 5–8, 2021, Nara, Japan

ACM ISBN 978-1-4503-8933-4/21/01…$15.00
https://doi.org/10.1145/3427796.3427819

nodes of G, i.e., $T(v_i, age_i)$ is mapped to node v_i of G. Each transaction $T(v_i, age_i)$ accesses (reads/writes) an arbitrary subset of w shared objects from $\mathcal{S}$. The transactions in $\mathcal{T}$ must commit in the order of the parameter *age*, i.e., transaction $T(v_i, age_i)$ commits only after all transactions $T(v_j, age_j)$ with $age_j < age_i$ have been committed. In $T(v_i, age_i)$, age_i provides a predefined commit order (or priority) to $T(v_i, age_i)$ among transactions in $\mathcal{T}$; age_i is an externally provided parameter (details in Section 2).

PredOrderScheduling naturally arises in applications where producing executions equivalent to a predefined order is needed to satisfy/guarantee certain properties. Example applications include speculative loop parallelization and distributed computation using state machine approach [24]. In loop parallelization [25], loops designed to run sequentially are parallelized by executing their operations concurrently using TM. Providing an order matching the sequential one is fundamental to enforce equivalent semantics for both the parallel and sequential code. Regarding state machine approach [16], many distributed systems order tasks before executing them to guarantee that a single state machine abstraction always evolves consistently on distinct nodes, e.g., Paxos [18].

This is the first time PredOrderScheduling is considered in distributed systems. Previously, PredOrderScheduling has been studied only in shared memory multi-core systems [11, 24] where execution time is the only metric of interest. Nevertheless, [11, 24] do not provide provable guarantees on execution time, i.e., they focused on empirical studies. Moreover, the techniques developed in [11, 24] do not extend for PredOrderScheduling in a distributed setting as they have no latency consideration.

In this paper, we design scheduling algorithms for PredOrderScheduling in the *synchronous* data-flow model [6, 7] where time is divided into discrete steps and aim to optimize two performance metrics that are fundamental to any distributed system: (i) *execution time* – the total time to execute and commit all the transactions, and (ii) *communication cost* – the total distance messages travel during execution. The messages are of two types: (a) the messages involved in moving the shared objects (read/written by transactions) to the transactions needing them and (ii) the messages involved in informing the commit status of transactions. A transaction's execution terminates as soon as it commits. That transaction may have started earlier but blocked while assembling the required objects.

We provide both offline and dynamic (online) algorithms to compute conflict-free schedules. The schedule determines the time step when each transaction executes and commits. After a transaction commits, it forwards its objects to any next requesting transactions in the execution order. Typically, an object is sent along a shortest path (or close to it), implying that the transfer time depends on the distance in G between the sender and receiver nodes. Execution time depends on the objects' traversal times, inter-transaction data dependencies, and commit of higher priority transactions.

We measure the efficiency of the designed scheduling algorithms from a widely-studied notion of *competitiveness* – the ratio of total time (total communication cost) for a designed algorithm to the shortest time (minimum communication cost) achievable by any scheduling algorithm. The goal is to make the competitive ratio as small as possible (the best possible is 1 which is hard to achieve).

Contributions. We have the following five contributions.

- For the offline version with complete knowledge of all transactions, their priorities, and the shared objects they need, we provide a scheduling algorithm that achieves simultaneously optimal execution time and communication cost in any graph G, i.e., the algorithm is 1-competitive (**Section 3**).
- For the offline version with only knowledge on the transactions and their priorities (but not the shared objects), we provide a scheduling algorithm that is $O(\log^2 n)$-competitive for both execution time and communication cost in any graph G. The competitive ratio becomes $O(1)$ on special graphs, such as doubling dimension graphs (**Section 4**).
- For the dynamic (online) version with transactions arriving over time, we provide a scheduling algorithm that is $O(D)$-competitive for both execution time and communication cost in any graph G, where D is the diameter of G (**Section 5**).
- For *relaxed* PredOrderScheduling where transactions at different nodes can commit out of order provided that they do not conflict with any lower aged transaction, we design algorithms achieving the same competitive bounds as in the above three non-relaxed counterparts (**Section 6**).
- We implement and rigorously evaluate the designed algorithms through 3 micro-benchmarks and 3 complex STAMP benchmarks on random and grid graphs (**Section 7**).

Techniques. The first offline algorithm (with complete knowledge on transactions, their priorities, and the objects they access) works using *object tours* for each object connecting the mapped nodes of transactions using that object in the priority order through the shortest paths in G. The lowest aged transaction using that object is connected to the owner node of that object again through a shortest path in G. A transaction can then execute when it has all needed objects assembled at its mapped node and the transaction before it in the priority order has been committed. We show that this method is optimal for both time and communication.

The second offline algorithm (only knowledge on transactions and their priorities but not the shared objects) exploits the concepts behind well-studied *distributed directory protocols* [15, 28] designed for synchronizing access to shared objects, adapted appropriately to send commit messages and shared objects to the nodes that need them, and shows that they allow to control execution time and communication cost simultaneously. In particular, the algorithm guarantees that the commit message from current transaction to the next in the order is sent with cost only a $O(\log^2 n)$ factor away from the best possible shortest path distance in any graph G (which can be improved to a $O(1)$ factor on a special graph G). It is also guaranteed that when the commit message is received from the previous transaction, the shared objects accessed by the current transaction have already been assembled to its mapped node, so that the current transaction can execute and commit immediately.

These techniques are then extended to the dynamic (online) version appropriately so that the designed algorithm can provide $O(D)$-competitive ratios for both time and communication even without knowing transaction priorities beforehand. It turned out that the directory protocol ideas (based on the hierarchy of clusters partitioning of G) used in the $O(\log^2 n)$-competitive algorithm above could only provide $O(D \log^2 n)$-competitive bounds for the

dynamic version. Therefore, the dynamic algorithm uses the directory protocol ideas (based on the spanning tree of G) so that the $O(\log^2 n)$ can be removed from the competitive bound. The above techniques are then extended to handle both offline and dynamic versions of *relaxed* PredOrderScheduling, achieving the same competitive bounds as in the algorithms for non-relaxed versions.

Related Work. The most closely related works to ours are [11, 24] in shared-memory multi-core systems. Gonzalez-Mesa *et al.* [11] introduced and studied PredOrderScheduling and Saad *et al.* [24] presented three improved algorithms and evaluated them. We study this problem in this paper for the very first time in distributed shared memory systems, where the goal is to minimize execution time and communication cost. The goal in [11, 24] is to minimize only the execution time and they did not provide formal competitive analysis of execution time like we do here.

The other previous works [15, 28, 33] in distributed shared memory systems focused on minimizing only the communication cost. The execution time minimization is first considered by Zhang *et al.* [33]. Busch *et al.* [4] considered for the first time minimizing both the execution time and communication cost. They showed that it is impossible to simultaneously minimize execution time and communication cost for all the scheduling problem instances in arbitrary G even in the offline setting, i.e., minimizing execution time implies high communication cost (and vice-versa). Then they provided offline algorithms optimizing individually execution time or communication cost. Busch *et al.* [5] considered transaction scheduling in special topologies (e.g., grid, line, clique, star, hypercube, butterfly, and cluster) and provided offline algorithms that minimize simultaneously execution time and communication cost. Later, Poudel and Sharma [23] provided an evaluation framework for processing transactions in distributed system following [5]. Recently, Busch *et al.* [7] provided dynamic (online) algorithms.

We consider in this paper the data-flow model. There is another model called the *control-flow* model [26], in which objects are immobile and transactions either move to the network nodes where the required objects reside, or invoke remote procedure calls. It is believed that the data-flow model provides a clear abstraction and is more amenable to control both time and communication [15]. Most of the previous studies on scheduling with provable bounds in distributed systems, e.g., [5–7], used the data-flow model.

(Equal-priority) transaction scheduling in general is widely-studied in shared memory multi-core systems. Several scheduling algorithms with provable upper and lower bounds, and impossibility results were given [1, 13, 27], besides several other scheduling algorithms that were evaluated only experimentally [32]. These scheduling algorithms and ideas developed, however, are not suitable for distributed shared memory systems as they do not deal with a crucial performance metric, communication cost.

Roadmap. We discuss model and some preliminaries in Section 2. We present our five contributions in Sections 3–7. We give concluding remarks in Section 8. Pseudocodes and some proofs are omitted due to space constraints.

2 MODEL AND PRELIMINARIES

Graph. We consider a distributed system $G = (V, E, \mathrm{w})$ of n nodes (representing processing nodes) $V = \{v_1, v_2, \ldots, v_n\}$, edges (representing communication links between nodes) $E \subseteq V \times V$, and edge weight function $\mathrm{w} : E \rightarrow \mathbb{Z}^+$. A *path* p in G is a sequence of nodes (with respective edges between adjacent nodes) with length$(p) = \sum_{e \in p} \mathrm{w}(e)$. We assume that G is connected. Let dist(u, v) denote the shortest path length (distance) between two nodes u and v. The communication links are *bidirectional* – the messages can be sent in both directions. It is also assumed that both the nodes and links are non-faulty and the links deliver messages in a FIFO order. The messages can be of any size and any number of messages can traverse an edge at any time, i.e., no bandwidth restriction. The *diameter* D is the maximum shortest path distance between any two nodes in G. The *k-neighborhood* of a node $u \in G$ is the set of nodes which are at distance $\leq k$ from u.

Communication Model. We consider the synchronous communication model where time is divided into discrete steps such that at each time step a node receives messages, performs a local computation, and then transmits messages to adjacent nodes [5–7]. For an edge $e = (u, v) \in E$, it takes $\mathrm{w}(e)$ time steps to transfer a message *msg* from u to v (and vice-versa); the *communication cost* contributed by *msg* is $\mathrm{w}(e)$. We assume that the message size is sufficient to convey the information about an object over the network. In realistic settings, the costs to transfer objects over the interconnection links may vary depending on the object's size. In this case, our bounds are affected only by the size factor (maximum object size divided by the size of the object that can be transferred over any link $e = (u, v) \in E$ of weight $\mathrm{w}(e)$ in $\mathrm{w}(e)$ time steps).

Transactions. Let $\mathcal{S} = \{S_1, S_2, \ldots, S_w\}$ denote the total w shared objects available in G. Each object has some value which can be read/written. The shared objects reside at nodes and are mobile, i.e., an object can move from one graph node to another. The node of G where an object S_i is currently positioned is called the *owner* of S_i, denoted as *owner*(S_i). Initially, the shared objects may be distributed arbitrarily at the nodes of G. A transaction $T(v_i, age_i)$ is an atomic block of code mapped and executed at node v_i which requires a set of objects $\mathcal{S}(T(v_i, age_i)) \subseteq \mathcal{S}$ and has priority age_i. Transaction $T(v_i, age_i)$ may modify some objects during execution while others remain unchanged. To simplify the analysis, we assume that each object has a single copy (for both read/write) and resides at one node at any time. We assume that each node runs a single thread and issues transactions sequentially, i.e., a node can issue a new transaction only after its previous transaction commits.

Transaction Execution and Conflicts. Transaction $T(v_i, age_i)$ can finish execution and commit only after all its required objects are gathered at v_i and it is the first among the not-yet-committed transactions in the predefined order. An execution of $T(v_i, age_i)$ ends with either *commit* (success) or *abort* (failure). Two transactions $T(v_i, age_i)$ and $T(v_j, age_j)$ *conflict* if they request (at least) a same shared object and at least one of them wants to modify that object (write a new value in it). A conflict is handled by aborting one of the transactions, or deferring the access of one transaction until the other commits. We assume that an aborted transaction *restarts* immediately from its beginning. A *scheduling algorithm* $\mathcal{A}$ determines the time that objects move from one node to another and the time that transactions execute. The transaction's computation time is one time step once it has the required objects.

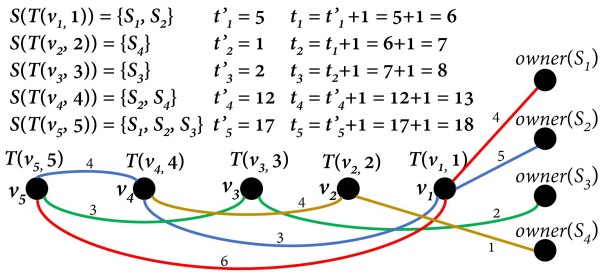

Figure 1: An illustration of the time step t_i in which transaction $T(v_i, age_i)$ using shared objects $\mathcal{S}(T(v_i, age_i))$ executes and commits in Off-Opt. There are 4 shared objects $S_1, \ldots, S_4$, initially at their owner nodes. The object tour $Object\_Tour(S_j)$ for each object S_j is also shown. The length of $Object\_Tour(S_j)$ is the communication cost of S_j. The total execution time is $t_4 = 18$ and the total communication cost is the sum of the object tours of $S_1, \ldots, S_4$.

Performance Metrics. We consider two performance metrics fundamental to any distributed system, namely execution time and communication cost. Let $\mathcal{E}$ be an execution schedule based on scheduling algorithm $\mathcal{A}$. The metrics can be defined as follows.

Definition 2.1 (Execution Time). For a set of transactions $\mathcal{T}$ in G, the time of an execution $\mathcal{E}$ is the time elapsed until the last transaction finishes its execution in $\mathcal{E}$. The execution time of scheduling algorithm $\mathcal{A}$ is the maximum time over all possible executions for $\mathcal{T}$.

Definition 2.2 (Communication Cost). For a set of transactions $\mathcal{T}$ in G, the communication cost of an execution $\mathcal{E}$ is the sum of the distances messages travel during $\mathcal{E}$. The communication cost of $\mathcal{A}$ is the maximum cost over all possible executions for $\mathcal{T}$.

The Scheduling Problem. Each transaction $T(v_i, age_i)$ is assigned age, age_i, before it is being activated, and the age signifies the transaction commit order [11, 24]. Following [11, 24], parameter *age* is supposed to satisfy the following three properties: (i) *unique* – no two transactions in $\mathcal{T}$ can have the same age, (ii) *non-modifiable* – does not change even if $T(v_i, age_i)$ is aborted multiple times, and (iii) *externally determined* – an external parameter that does not depend on the execution of concurrent transactions. Formally,

Definition 2.3 (PredOrderScheduling). Given a set of transactions $\mathcal{T} := \{T(v_1, age_1), T(v_2, age_2), \ldots\}$ mapped (arbitrarily) to the nodes of a distributed system G, commit transactions in $\mathcal{T}$ in the increasing order of the parameter age.

A *relaxed* PredOrderScheduling where transactions don't need to execute in the increasing order of *age* is studied in Section 6.

3 OFFLINE OPTIMAL-TIME-AND-COMMUNICATION ALGORITHM

We study here offline PredOrderScheduling with complete knowledge on the problem – the transactions in $\mathcal{T}$, their priorities, the shared objects transactions need, the initial positions of shared objects in $\mathcal{S}$. We provide a polynomial-time algorithm Off-Opt that achieves optimal execution time and communication cost i.e., Off-Opt is 1-competitive in both time and communication.

Algorithm Off-Opt. Off-Opt constructs an *object tour*, denoted as $Object\_Tour(S_j)$, for each object $S_j \in \mathcal{S}$ starting from its initial owner node visiting all the transactions in $\mathcal{T}$ that access it in the order of their priorities. $Object\_Tour(S_j)$ for an object $S_j \in \mathcal{S}$ is constructed as follows. Let $\mathcal{T}(S_j) \subseteq \mathcal{T}$ be the set of transactions in $\mathcal{T}$ that access object S_j (for read/write). Order the transactions in $\mathcal{T}(S_j)$ according to their ages, starting from the lowest aged transaction to the highest. Let $\mathcal{T}(S_j) := \{T(v_1(S_j), age_1(S_j)), T(v_2(S_j), age_2(S_j)), \ldots\}$ be the set after the ordering. Let $owner(S_j)$ be the owner node of object S_j. Connect $owner(S_j)$ with the mapped node $v_1(S_j)$ of the lowest aged transaction $T(v_1(S_j), age_1(S_j)) \in \mathcal{T}(S_j)$ through the shortest path $\text{dist}(v_1(S_j), owner(S_j))$ in G. Then, connect the node $v_1(S_j)$ of $T(v_1(S_j), age_1(S_j)) \in \mathcal{T}(S_j)$ with the second lowest aged transaction $T(v_2(S_j), age_2(S_j)) \in \mathcal{T}(S_j)$ through the shortest path $\text{dist}(v_1(S_j), v_2(S_j))$ in G. Repeat this process until the highest aged transaction in $\mathcal{T}(S_j)$ is connected to the second highest aged transaction in $\mathcal{T}(S_j)$. Therefore, $Object\_Tour(S_j) := \{owner(S_j), v_1(S_j), v_2(S_j), \ldots\}$, with each pair of successive nodes are connected through the shortest paths in G.

The object tours help transactions to decide on the time steps they can execute and commit. Off-Opt decides on the time step t_i transaction $T(v_i, age_i)$ can execute and commit as follows. Let $\mathcal{S}(T(v_i, age_i)) \subseteq \mathcal{S}$ be the set of objects required by $T(v_i, age_i)$. The lowest aged transaction $T(v_1, age_1) \in \mathcal{T}$ can execute and commit at time step $t_1 = t'_1 + 1$, where

$$t'_1 = \max_{S_j \in \mathcal{S}(T(v_1, age_1))} \text{dist}(v_1, owner(S_j))$$

such that by time step t'_1 the lowest aged transaction $T(v_1, age_1)$ assembles each object $S_j \in \mathcal{S}(T(v_1, age_1))$ at v_1.

For simplicity in exposition, let $T(v_{i-1}, age_{i-1})$ be the transaction in $\mathcal{T}$ that (immediately) precedes $T(v_i, age_i)$ in the *age* order. If there does not exist $T(v_{i-1}, age_{i-1})$, then v_{i-1} is the owner node $owner(S_j)$, i.e., $owner(S_j)$ precedes v_i and $T(v_i, age_i)$ is in fact $T(v_1, age_1)$. Let t_{i-1} be the time step at which $T(v_{i-1}, age_{i-1})$ has been committed. Pick an object $S_j \in \mathcal{S}(T(v_i, age_i))$. Consider the object tour for S_j, $Object\_Tour(S_j)$. Let $T(v_{prev(S_j)}, age_{prev(S_j)})$ be the transaction that used S_j before $T(v_i, age_i)$[1]. Let $t_{prev(S_j)}$ be the time step at which $T(v_{prev(S_j)}, age_{prev(S_j)})$ has been committed. Off-Opt decides that

$$t_i = \begin{cases} t_{i-1} + 1, & \text{if } t'_i < t_{i-1} \\ t'_i + 1, & \text{otherwise,} \end{cases}$$

where

$$t'_i = \max_{S_j \in \mathcal{S}(T(v_i, age_i))} \begin{cases} \text{dist}(v_i, v_{prev(S_j)}) + \\ t_{prev(S_j)}, & \text{if } v_{prev(S_j)} \neq owner(S_j) \\ \\ \text{dist}(v_i, owner(S_j)), & \text{otherwise.} \end{cases}$$

Initially, each object S_j moves from its owner node $owner(S_j)$ to the first transaction in the order that requests it following $Object\_Tour(S_j)$. From that node it moves towards the next node in

[1] $T(v_{prev(S_j)}, age_{prev(S_j)})$ may not necessarily be $T(v_{i-1}, age_{i-1})$; $T(v_{prev(S_j)}, age_{prev(S_j)})$ is $T(v_{i-1}, age_{i-1})$ if and only if $S_j \in \mathcal{S}(T(v_{i-1}, age_{i-1}))$, otherwise it is some other transaction in $\mathcal{T}$.

$Object\_Tour(S_j)$ as soon as the transaction on that node commits. The algorithm terminates as soon as the highest aged transaction $T(v_h, age_h) \in \mathcal{T}$ executes and commits at time step t_h. Object S_j stops moving after it reaches the last node in $Object\_Tour(S_j)$. Fig. 1 illustrates ideas behind Off-Opt.

Analysis of Off-Opt**.** We analyze Off-Opt for correctness, execution time, and communication cost bounds.

Theorem 3.1. Off-Opt *correctly solves* PredOrderScheduling.

Proof. Consider any transaction $T(v_i, age_i) \in \mathcal{T}$. Consider another transaction $T(v_{i-1}, age_{i-1}) \in \mathcal{T}$ that immediately precedes $T(v_i, age_i)$ in the age order. In Off-Opt, $T(v_i, age_i)$ does not commit before $T(v_{i-1}, age_{i-1})$ since $t_i \geq t_{i-1} + 1$. Furthermore, $T(v_i, age_i)$ does not execute until all objects it requires assemble at its mapped node v_i which happens by time t'_i. If $t'_i < t_{i-1}$, then $t_i = t_{i-1} + 1$ satisfies $t_i > t_{i-1}$, otherwise $t_i = t'_i + 1$ satisfies that $t_i > t_{i-1}$. Therefore, Off-Opt correctly solves PredOrderScheduling. □

Theorem 3.2. Off-Opt *achieves optimal execution time.*

Theorem 3.3. Off-Opt *achieves optimal communication cost.*

Theorem 3.4. Off-Opt *computes the execution schedule in polynomial time.*

4 OFFLINE EFFICIENT-TIME-AND-COMMUNICATION ALGORITHM

We study here offline PredOrderScheduling with a priori knowledge only on the transactions and their priorities. The transactions are not known about the shared objects they access until runtime. We provide a polynomial-time algorithm Off-Eff that simultaneously achieves execution time and communication cost within a $O(\log^2 n)$ factor from optimal, i.e., Off-Eff is $O(\log^2 n)$-competitive in both metrics. On special graphs, it becomes $O(1)$-competitive.

High Level Idea Behind Off-Eff**.** The idea is to use the *directory protocols* developed for synchronizing access to shared objects distributed over a network, for example, [2, 9, 15, 28], so that the communication cost in moving the objects to the mapped nodes can be controlled forming a *distributed object tour* per object based on transactions that request that object. We then control execution time by forming a *distributed transaction queue* of transactions in $\mathcal{T}$. We achieve the following two properties for both the queues:

(i) Construction of a *distributed transaction queue* satisfying predefined order of transactions in $\mathcal{T}$ mapped initially to the nodes of G. The commit messages are then sent to the successor transactions following the paths in their respective transaction tours.
(ii) Construction of a *distributed object tour* for each object $\mathcal{S}_j \in \mathcal{S}$, satisfying the predefined order of the transactions that access that object. The objects are then forwarded to the transactions accessing them following the paths in their respective object tours.

We will show that executing transactions following the distributed transaction tour controls the execution time of (executing and) committing all the transactions. We will also show that when a transaction $T(v_i, age_i)$ receives a commit message from the previous transaction, all the objects in the set $\mathcal{S}(T(v_i, age_i))$ are already assembled at v_i. These ideas altogether make Off-Eff simultaneously execution time and communication cost efficient.

We discuss in the next three subsections a notion of *overlay tree* $\mathcal{OT}$ construction and the construction of distributed transaction queue and object tours on top of $\mathcal{OT}$. We will then finally describe the details of algorithm Off-Eff and present its analysis.

4.1 Overlay Tree Construction

There are two different well-known approaches in constructing overlay tree $\mathcal{OT}$ on the graph G: (i) use a *spanning tree* of G or (ii) use a *hierarchy of clusters* on G. The spanning-tree-based approach was used in directory protocols [2, 9] and the hierarchy-of-clusters-based approach was used in directory protocols [15, 28, 29].

Both of these approaches work for Off-Eff. One major distinction is that hierarchy-of-clusters-based overlay trees are more suitable to control communication costs (and hence the execution time) compared to the spanning-tree-based overlay trees. Therefore, we discuss the construction of hierarchy-of-clusters-based overlay tree $\mathcal{OT}$ as follows. In a high level, divide the graph G into a hierarchy of clusters with $H_1 = \lceil \log D \rceil + 1$ layers such that the clusters sizes grow exponentially. A *cluster* is a subset of nodes, and its diameter is the maximum distance between any two nodes. The diameter of each cluster at layer ℓ, where $0 \leq \ell < H_1$, is no more than $f(\ell)$, for some function f, and each node participates in no more than $g(\ell)$ clusters at layer ℓ, for some other function g. Moreover, for each node u in G, there is a cluster at layer ℓ such that the $(2^\ell - 1)$-neighborhood of u is contained in that cluster.

There are known cluster hierarchy construction algorithms, such as a *hierarchical sparse cover* of G that gives a cluster hierarchy $\mathcal{Z}$ with H_1 layers with $f(\ell) = O(\ell \log n)$ and $g(\ell) = O(\log n)$. This construction was used in the Spiral directory protocol due to Sharma *et al.* [28] in which each layer ℓ is decomposed into $H_2 = O(\log n)$ sub-layers of clusters, such that a node participates in all the sub-layers of a layer but in a different cluster within each sub-layer, i.e., at each layer ℓ a node u participates in $g(\ell) = O(\log n)$ clusters. In the construction of [28], a node in each cluster is designated as the *leader* of the cluster. Connecting the leaders of the clusters in the subsequent levels gives $\mathcal{OT}$. An *upward path* $p(u)$ for each node $u \in G$ is built by visiting leader nodes in all the clusters that u belongs to starting from layer 0 (the bottom layer in $\mathcal{Z}$) up to layer H_1 (the top layer in $\mathcal{Z}$). Within each layer, H_2 sub-layers are visited by $p(u)$ according to the order of their sub-layer labels. In fact [28] already constructs upward paths to be able to run their algorithm Spiral. The upward path $p(u)$ visits two subsequent leaders using shortest paths in G between them. Lets say two paths *intersect* if they have a common node. Using this definition, two upward paths intersect at layer i if they visit the same leader at layer i. The lemmas below are satisfied in the construction of [28].

Lemma 4.1. *For any two nodes $u, v \in G$, their upward paths $p(u)$ and $p(v)$ intersect at layer* $\min\{H_1, \lceil \log(\text{dist}(u, v)) \rceil + 1\}$.

Lemma 4.2. *For any upward path $p(u)$ for any node $u \in G$ from the bottom layer upto layer ℓ (and any sub-layer in layer ℓ),* $\text{length}(p(u)) \leq O(2^\ell \log^2 n)$.

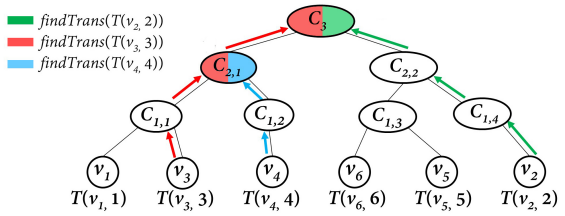

Figure 2: Illustration of construction of $DistTransQueue(\mathcal{T})$. The message $findTrans(T(v_3, 3))$ from node v_3 meets $findTrans(T(v_4, 4))$ from node v_4 at layer 2 cluster $C_{2,1}$ (denoted as red|blue). Similarly, $findTrans(T(v_2, 2))$ from node v_2 meets $findTrans(T(v_3, 3))$ from node v_3 at layer 3 cluster C_3 (denoted as red|green). Now, transaction $T(v_3, 3)$ knows the previous ($T(v_2, 2)$) and next ($T(v_4, 4)$) transactions. $T(v_3, 3)$ executes after receiving commit message from $T(v_2, 2)$ and it sends a commit message to $T(v_4, 4)$ as soon as it commits.

4.2 Distributed Transaction Queue Construction

Denote the distributed transaction queue by $DistTransQueue(\mathcal{T})$ for the transactions in $\mathcal{T}$. The goal is to order the transactions in $DistTransQueue(\mathcal{T})$ so that they satisfy the predefined order. Let the transactions in $\mathcal{T}$ in the increasing order of age are denoted as $\mathcal{T} := \{T(v_1, age_1), T(v_2, age_2), \ldots, T(v_{i-1}, age_{i-1}), T(v_i, age_i), \ldots\}$. Consider transaction $T(v_i, age_i) \in \mathcal{T}$. Transaction $T(v_i, age_i)$ sends $findTrans(T(v_i, age_i))$ message in its upward path $p(v_i)$ in $\mathcal{OT}$. The goal of sending the $findTrans(T(v_i, age_i))$ message is to find the messages $findTrans(T(v_{i-1}, age_{i-1}))$ and $findTrans(T(v_{i+1}, age_{i+1}))$ at some leader node at some level in $\mathcal{OT}$. If $findTrans(T(v_i, age_i))$ meets $findTrans(T(v_{i-1}, age_{i-1}))$ at some layer ℓ but not $findTrans(T(v_{i+1}, age_{i+1}))$, it continues moving upward in $p(v_i)$ until it meets $findTrans(T(v_{i+1}, age_{i+1}))$ at some layer $\ell' \geq \ell$ and vice-versa. After it meets both $findTrans(T(v_{i-1}, age_{i-1}))$ and $findTrans(T(v_{i+1}, age_{i+1}))$, then $findTrans(T(v_i, age_i))$ stops. After such meeting happens for all $findTrans(T(v_{i-1}, age_{i-1}))$, $findTrans(T(v_i, age_i))$, and $findTrans(T(v_{i+1}, age_{i+1}))$, $2 \leq i \leq n-1$, the construction of $DistTransQueue(\mathcal{T})$ is essentially finished. Fig. 2 illustrates the construction of $DistTransQueue(\mathcal{T})$ on top of the overlay tree $\mathcal{OT}$ constructed in the previous subsection.

4.3 Distributed Object Tour Construction

A distributed object tour, denoted as $DistObjTour(S_j)$, is constructed for each object $S_j \in \mathcal{S}$, i.e., there will be total w different object tours for w shared objects in $\mathcal{S}$. $DistObjTour(S_j)$ is constructed as follows. Let $\mathcal{T}(S_j) \subseteq \mathcal{T}$ be the set of transactions in $\mathcal{T}$ that access object S_j (for read/write). Each of the graph nodes (with transactions) accessing S_j send $findObj(T(v_i, age_i), S_j)$ message in $\mathcal{OT}$ following the upward path $p(v_i)$. As in $findTrans(T(v_i, age_i))$, the goal of $findObj(T(v_i, age_i), S_j)$ is to find messages $findObj(T(v_{prev(S_j)}, age_{prev(S_j)}), S_j)$ and $findObj(T(v_{next(S_j)}, age_{next(S_j)}), S_j)$, where $T(v_{prev(S_j)}, age_{prev(S_j)})$ and $T(v_{next(S_j)}, age_{next(S_j)})$ are the transactions in $\mathcal{T}(S_j)$ such that they are ordered immediately before and after T_i in $\mathcal{T}(S_j)$ in the age order. Message $findObj(T(v_i, age_i), S_j)$ stops as soon as it meets both $findObj(T(v_{prev(S_j)}, age_{prev(S_j)}), S_j)$ and $findObj(T(v_{next(S_j)}, age_{next(S_j)}), S_j)$ at some layer ℓ in $\mathcal{OT}$. After the meeting happens for $findObj(T(v_{prev(S_j)}, age_{prev(S_j)}), S_j)$, $findObj(T(v_i, age_i), S_j)$, and $findObj(T(v_{next(S_j)}, age_{next(S_j)}), S_j)$, $DistObjTour(S_j)$ is essentially constructed by v_i for S_j.

Let v_κ be the leader node at layer κ where $findObj(T(v_i, age_i), S_j)$ met $findObj(T(v_{prev(S_j)}, age_{prev(S_j)}), S_j)$. Object S_j can be forwarded to $T(v_i, age_i)$ by $T(v_{prev(S_j)}, age_{prev(S_j)})$ by first sending the object upward in $p(v_{prev(S_j)})$ up to v_κ and then sending the object downward in $p(v_i)$ from v_κ up to node v_i.

There is one last thing to consider while constructing $DistObjTour(S_j)$. There is a special case while constructing $DistObjTour(S_j)$, such that $owner(S_j)$ may not be at the mapped node v_1 of the lowest aged transaction $T(v_1, age_1)$. In this case, node $owner(S_j)$ sends a special $find(T(v_1, age_1), S_j)$ message following $p(owner(S_j))$. Message $find(T(v_1, age_1), S_j)$ stops as soon as it meets message $findObj(T(v_1, age_1), S_j)$.

4.4 Algorithm Off-Eff and Its Analysis

Off-Eff works on top of $\mathcal{OT}$. Off-Eff first constructs distributed transaction queue and distributed object tours, $DistTransQueue(\mathcal{T})$ and $DistObjTour(S_j)$, respectively. When the queue/tour construction finishes, there will be a single queue $DistTransQueue(\mathcal{T})$ and w tours $DistObjTour(S_j)$ for w objects in $\mathcal{S}$.

After that, the lowest aged transaction $T(v_1, age_1)$ executes and commits as soon as all the shared objects in $\mathcal{S}(T(v_1, age_1))$ are gathered at v_1. Each object $S_j \in \mathcal{S}(T(v_1, age_1))$ not already at v_1 move to v_1 following its $DistObjTour(S_j)$ from its $owner(S_j)$. Suppose $T(v_1, age_1)$ commits at time step t_1. $T(v_1, age_1)$ sends $commit(T(v_1, age_1))$ message and each object $S_j \in \mathcal{S}(T(v_1, age_1))$ in the upward path $p(v_1)$ in $\mathcal{OT}$. The commit message and objects in $\mathcal{S}(T(v_1, age_1)$ required by $T(v_2, age_2)$ reach node v_2 following their paths in $\mathcal{OT}$. As soon as $T(v_2, age_2)$ executes and commits at time step $t_2 > t_1$, then $commit(T(v_2, age_2))$ is sent to $T(v_3, age_3)$ and objects in $\mathcal{S}(T(v_2, age_2))$ move to the next nodes following their individual $DistObjTour(.)$. Off-Eff terminates as soon as the highest aged transaction $T(v_h, age_h) \in \mathcal{T}$ executes and commits at some time step t_h.

We now analyze Off-Eff for correctness, execution time, and communication cost bounds.

Lemma 4.3. *Consider two consecutive transactions $T(v_{i-1}, age_{i-1})$, $T(v_i, age_i) \in \mathcal{T}$ in the age order. If $\text{dist}(v_{i-1}, v_i) \leq 2^\ell$, then messages $findTrans(T(v_i, age_i))$ and $findTrans(T(v_{i-1}, age_{i-1}))$ meet at layer ℓ of $\mathcal{OT}$.*

We have the similar lemma for $findObj(.,.)$ messages.

Lemma 4.4. *If $\text{dist}(v_i, v_j) \leq 2^\ell$ for any two consecutive transactions $T(v_{prev(S_j)}, age_{prev(S_j)}), T(v_i, age_i) \in \mathcal{T}$ in the age order in accessing an object $S_j \in \mathcal{S}$, then the messages $findObj(T(v_{prev(S_j)}, age_{prev(S_j)}), S_j)$ and $findObj(T(v_i, age_i), S_j)$ meet at layer ℓ of $\mathcal{OT}$.*

The following observation is also immediate.

Observation 1. *After the meeting of the messages $findTrans(T(v_i, age_i))$ and $findTrans(T(v_{i+1}, age_{i+1}))$ in some layer $\ell \leq H_1$, transaction $T(v_i, age_i)$ knows how to reach the mapped node v_{i+1} of transaction $T(v_{i+1}, age_{i+1})$, following the upward path $p(v_i)$ up to layer $0 \leq \ell \leq H_1$ and then the upward path $p(v_{i+1})$ in opposite direction from layer ℓ to layer 0.*

A similar observation applies to $findObj(T(v_i, age_i), S_j)$ messages so that the object S_j can move to $v_{next(S_j)}$ following upward path $p(v_i)$ up to layer ℓ and then upward path $p(v_{next(S_j)})$ in opposite direction from layer ℓ to layer 0.

Now, we provide following lemma that is crucial for execution in Off-Eff.

Lemma 4.5. *If a transaction $T(v_i, age_i) \in \mathcal{T}$ receives $commit(T(v_{i-1}, age_{i-1})$ message at time step $t_i - 1$, then all the objects in $\mathcal{S}(T(v_i, age_i)) \subseteq \mathcal{S}$ are gathered at v_i by time step $t_i - 1$.*

Lemma 4.6. Off-Eff *is $O(\log^2 n)$-competitive in execution time.*

Proof. We have from Lemma 4.3 that if two consecutive transactions $T(v_{i-1}, age_{i-1}), T(v_i, age_i) \in \mathcal{T}$ in the age order are $\text{dist}(v_{i-1}, v_i) \leq 2^\ell$ apart in G, then $findTrans(T(v_{i-1}, age_{i-1}))$ and $findTrans(T(v_i, age_i))$ messages meet at layer ℓ in $\mathcal{OT}$ following the upward paths $p(v_{i-1}), p(v_i)$. The length of the paths $p(v_{i-1}), p(v_i)$ up to layer ℓ is $c \cdot 2^\ell \cdot \log^2 n$ (Lemma 4.2). Therefore, after $T(v_{i-1}, age_{i-1})$ commits at time step t_{i-1}, $commit(T(v_{i-1}, age_{i-1}))$ reaches $T(v_i, age_i)$ (following the upward path $p(v_{i-1})$ up to layer ℓ and following the upward path $p(v_i)$ in the opposite direction from layer ℓ up to later 0 at time step

$$t_i' = t_{i-1} + 2 \cdot c \cdot 2^\ell \log^2 n = t_{i-1} + 2 \cdot c \cdot \text{dist}(v_{i-1}, v_i) \log^2 n.$$

We have from Lemma 4.5 that all the shared objects in $\mathcal{S}(T(v_i, age_i))$ are already gathered at v_i by time step t_i'. Therefore, transaction $T(v_i, age_i)$ can execute and commit at time step $t_i = t_i' + 1$. This process repeats in Off-Eff for each consecutive pairs of transactions in $\mathcal{T}$. Therefore, the total execution time in Off-Eff is bounded by

$$t_{ALG} \leq \max_{S_j \in \mathcal{S}(T(v_1, age_1))} 2 \cdot c \cdot \text{dist}(v_1, owner(S_j)) \log^2 n + \sum_{i=2}^{|\mathcal{T}|} 2 \cdot c \cdot \text{dist}(v_{i-1}, v_i) \log^2 n,$$

where $\text{dist}(v_1, owner(S_j))$ is the shortest path distance between the mapped node v_1 of transaction $T(v_1, age_1)$ and the owner of object $S_j \in \mathcal{S}(T(v_1, age_1))$.

We now establish lower bound on execution time. A transaction $T(v_i, age_i)$ cannot execute until it receives the commit message from transaction $T(v_{i-1}, age_{i-1})$. Therefore, the optimal execution time must be at least $\geq \sum_{i=2}^{|\mathcal{T}|} \text{dist}(v_{i-1}, v_i)$. Moreover, the lowest aged transaction $T(v_1, age_1)$ cannot commit until it receives all the objects in $\mathcal{S}(T(v_1, age_1))$ from their owner nodes. The time for this is at least

$$\geq \max_{S_j \in \mathcal{S}(T(v_1, age_1))} \text{dist}(v_1, owner(S_j)).$$

Therefore, the optimal execution time t_{OPT} is at least

$$t_{OPT} \geq \max_{S_j \in \mathcal{S}(T(v_1, age_1))} \text{dist}(v_1, owner(S_j)) + \sum_{i=2}^{|\mathcal{T}|} \text{dist}(v_{i-1}, v_i).$$

Thus, the competitive ratio of Off-Eff in execution time is bounded by $\frac{t_{ALG}}{t_{OPT}} \leq 2c \log^2 n = O(\log^2 n)$. □

Lemma 4.7. Off-Eff *is $O(\log^2 n)$-competitive in communication cost.*

Theorem 4.8. Off-Eff *computes the execution schedule in polynomial time.*

Proof. The overlay tree $\mathcal{OT}$ construction takes polynomial time [28]. The upward paths can then be computed in polynomial time. Furthermore, the construction of transaction and object queues takes polynomial time. □

Remark: Bounds on Special Graphs. Off-Eff can achieve better execution time and communication cost bounds for doubling dimension metrics, including both constant-dimensional Euclidean and growth-restricted metrics. Particularly, for doubling-dimension graphs, we can use the overlay tree $\mathcal{OT}$ construction of [15], where it is guaranteed that $\text{length}(p(u)) \leq O(2^\ell)$ for any upward path $p(u)$ of node u until layer ℓ in $\mathcal{OT}$ (the improvement is an $O(\log^2 n)$ factor compared to Lemma 4.2). Everything else remains the same.

Theorem 4.9. *In doubling dimension metrics, including constant-dimensional Euclidean and growth-restricted metrics,* Off-Eff *is $O(1)$-competitive in both execution time and communication cost.*

5 DYNAMIC SCHEDULING

We study here dynamic PredOrderScheduling with no a priori knowledge on transactions (including age and the objects they access) and the initial positions of objects. A transaction only knows its *age* and the objects it needs when it arrives. In contrast to offline PredOrderScheduling where all transactions in $\mathcal{T}$ arrive before execution starts, in the dynamic PredOrderScheduling transactions in $\mathcal{T}$ may arrive over time. The arrival may be arbitrary in the sense that at any time, there may be zero, one, or more transactions arriving to the system. We present an algorithm Dyn that is $O(D)$-competitive in both execution time and communication cost.

Algorithm Dyn**.** Dyn works on top of a spanning tree-based overlay tree, which we denote as $\mathcal{OT}_{ST}$. The reason of using $\mathcal{OT}_{ST}$ instead of the hierarchy of clusters-based overlay tree $\mathcal{OT}$ used in Section 4 is that $\mathcal{OT}_{ST}$ helps to reduce the competitive ratio bounds by an $O(\log^2 n)$ factor compared to the ratio bounds obtained using $\mathcal{OT}$, i.e., using $\mathcal{OT}$, Dyn can achieve $O(D \log^2 n)$ competitive ratio, whereas with $\mathcal{OT}_{ST}$, it achieves $O(D)$ competitive ratio.

Let v_{root} be the root node of $\mathcal{OT}_{ST}$; v_{root} is in fact the root node of the spanning tree ST used as $\mathcal{OT}_{ST}$. The upward path $p(v)$ can be defined in $\mathcal{OT}_{ST}$ connecting the parent nodes in ST from node v up to the root v_{root}. As soon as the execution starts, each shared object $S_j \in \mathcal{S}$ moves to the v_{root} following the upward path $p(owner(S_j))$ from its initial position. Furthermore, as soon as a transaction $T(v_i, age_i)$ arrives, it sends $findTrans(T(v_i, age_i))$ message to the root v_{root} following the upward path $p(v_i)$. $T(v_i, age_i)$ also sends $findObj(T(v_i, age_i), S_j)$ message to v_{root} for each $S_j \in \mathcal{S}(T(v_i, age_i))$ that it accesses, again following the upward path $p(v_i)$.

At any time step t, the root node v_{root} has a set of pending transactions $\mathcal{T}_t(v_{root})$ as follows. Suppose $T(v_i, age_i)$ arrives at node v_i at time step t_i''. When v_{root} receives $findTrans(T(v_i, age_i))$ at time

Transaction	$T(v_1, 1)$	$T(v_2, 2)$	$T(v_3, 3)$	$T(v_4, 4)$	$T(v_5, 5)$
Denoted as	T_1	T_2	T_3	T_4	T_5
Arrival time step	0	11	7	5	0

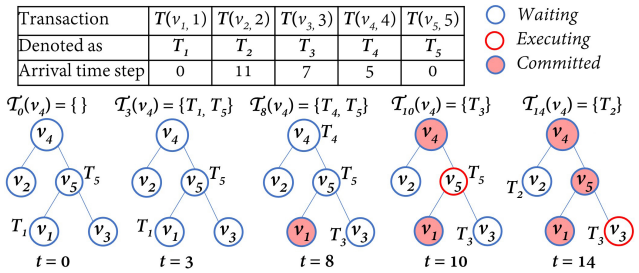

Figure 3: Illustration of pending transactions $\mathcal{T}_t(v_{root})$ at v_{root} at any time step $t \geq 0$ in Algorithm Dyn. At $t = 2$ and $t = 3$, $findTrans(*)$ messages from transactions $T(v_1, 1)$ and $T(v_5, 5)$ (denoted as T_1 and T_5, respectively) reach the root node v_4. T_1, T_5 will be added to $\mathcal{T}_t(v_4)$ at time step $t_{arrival}+D = 0+3 = 3$. T_4 will be added to $\mathcal{T}_t(v_4)$ at time step $5+D = 5+3 = 8$. Similarly, T_2, T_3 will be added at time steps 14, 10, respectively. T_1 executes and commits at $t = 6$. v_4 receives commit message from T_1 at $t = 8$ and by that time, T_4 has already been added to $\mathcal{T}_t(v_4)$, then T_4 executes and commits at $t = 9$. T_5 executes at $t = 10$ and so on.

$t_{root} \geq t_i''$, then it includes $T(v_i, age_i)$ at time step $t''' = t_i'' + D$, where D is the diameter of graph G. It might be the case that $T(v_i, age_i)$ arrives at time step $< t_i'' + D$. In this case, $T(v_i, age_i)$ simply waits at v_{root} to be included in $\mathcal{T}_t(v_{root})$. At any time step t, v_{root} makes scheduling decision among the transactions in $\mathcal{T}_t(v_{root})$ as follows. Let $\mathcal{S}_t(v_{root}) \subseteq \mathcal{S}$ be the set of objects that are at v_{root} at time step t. Let $T_t(v_s, age_s) \in \mathcal{T}_t(v_{root})$ be the lowest aged transaction in $\mathcal{T}_t(v_{root})$ at time t. If all the objects in $\mathcal{S}(T_t(v_s, age_s))$ are on v_{root} at time t, v_{root} will send all the objects to go to node v_s following $p(v_s)$ in the opposite direction from v_{root} to v_s. Suppose the objects in $\mathcal{S}(T_t(v_s, age_s))$ reach v_s at time $t_s' > t$. $T_t(v_s, age_s)$ executes and commits at time step $t_s = t_s' + 1$. After the commit, v_s sends the objects in $\mathcal{S}(T_t(v_s, age_s))$ back to v_{root} following the upward path $p(v_s)$. Node v_s also sends $commit(T_t(v_s, age_s))$ message to v_{root} following $p(v_s)$. As soon as v_{root} receives $commit(T_t(v_s, age_s))$, it can then schedule the smallest aged transaction among the ones now in $\mathcal{T}_t(v_{root})$ (which does not include $T_t(v_s, age_s)$). Fig. 3 illustrates the ideas behind Dyn.

Analysis of Dyn. We analyze Dyn considering two cases: (i) transaction arriving at time step t has age smaller than any transaction arriving at time step $t' > t$ and (ii) transaction arriving at time step t might not have age smaller than any transaction arriving at time step $t' > t$. Interestingly, in both the cases, the competitive ratio bounds remain the same. The only difference is on the guarantees on the commit order of transactions. We first analyze case (i).

Lemma 5.1. *Take any time step t. Suppose the transaction priorities depend on their arrival time. Let $T(v_j, age_j)$ be a transaction that arrives at t. Consider a transaction $T(v_i, age_i)$ that arrives at time step $t_i < t$.* Dyn *commits $T(v_i, age_i)$ before $T(v_j, age_j)$.*

Theorem 5.2. Dyn *is $O(D)$-competitive in execution time.*

Proof. Consider any transaction $T(v_i, age_i) \in \mathcal{T}$. Consider the time step t_{i-1} in which the previous transaction (in the age order) in $\mathcal{T}$ commits. By $t_{i-1} + D$ time steps, v_{root} receives all the objects used by that transaction plus the commit message. After that, the objects in $\mathcal{S}(T(v_i, age_i))$ can reach node v_i from v_{root} in at most $t_{i-1} + 2D$ time steps. $T(v_i, age_i)$ can then execute and commit at time step $t_i = t_{i-1} + 2D + 1$. The lowest aged transaction takes at most $t_1 = 2D + 1$ time steps to commit since it needs to get the objects from their owner nodes; the objects first reach to v_{root} and then to the transactions' mapped node. Therefore, total time in Dyn to commit all the transactions in $\mathcal{T}$ is $(|\mathcal{T}| - 1) \cdot (2D + 1)$. The optimal time to commit all the transactions in $\mathcal{T}$ is $|\mathcal{T}|$ time steps since they have to commit in the age order. Therefore, Dyn is $O(D)$-competitive in execution time. □

Theorem 5.3. Dyn *is $O(D)$-competitive in communication cost.*

Proof. We prove that Dyn is $O(D)$-competitive for any object $S_j \in \mathcal{S}$. This immediately applies that it is in overall $O(D)$-competitive in communication cost considering the collective costs due to all w objects. Consider $\mathcal{T}(S_j) := \{T(v_1(S_j), age_1(S_j)), T(v_2(S_j), age_2(S_j)), \ldots\} \subseteq \mathcal{T}$ be the transactions that use S_j listed in the increasing order of their ages. S_j needs to traverse starting from $owner(S_j)$ to all the transactions in $\mathcal{T}(S_j)$. After reaching one transaction, it will go back to v_{root} before going to the next transaction in $\mathcal{T}(S_j)$. Therefore, the communication cost for S_j is $|\mathcal{T}(S_j)| \cdot 2D$. Since S_j has to be reached to $|\mathcal{T}(S_j)|$ different nodes of G, the optimal cost is at least $|\mathcal{T}(S_j)|$. Therefore, Dyn is $O(D)$-competitive in communication cost. □

We now analyze case (ii). It is easy to see that the time and communication bounds remain the same as in case (i). Therefore, we prove the following guarantee on the commit order.

Lemma 5.4. *Suppose the transaction priorities do not depend on their arrival time.* Dyn *commits transactions in the increasing age order of $\mathcal{T}_t(v_{root})$, the set of transactions at v_{root} at t.*

Proof. Since ages of transactions are unique, at any time t, there can be an ordering of the transactions in $\mathcal{T}(v_{root})$. Among those transactions, at time step t, v_{root} picks among them the lowest aged one to execute and commit within next $D + 1$ time steps. □

6 EXTENSION: RELAXED SCHEDULING

The *relaxed* PredOrderScheduling problem can be defined as follows: Transactions can commit out of order if they do not conflict with any lower aged transactions. This is in contrast to PredOrderScheduling, where even if a transaction does not conflict with any lower aged transaction, it has to wait until all lower aged transactions commit. Therefore, *relaxed* PredOrderScheduling achieves better concurrency compared to PredOrderScheduling.

Relaxed Off-Opt. We study offline *relaxed* PredOrderScheduling as in Section 3. We present a polynomial-time algorithm R-Off-Opt extending (the polynomial-time) algorithm Off-Opt and achieve optimal execution time and communication cost.

Algorithm R-Off-Opt is as follows. Use the notations and concepts as in algorithm Off-Opt. For each transaction $\mathcal{T}(v_i, age_i) \in \mathcal{T}$, compute t_i', and execute and commit $\mathcal{T}(v_i, age_i)$ at time step $t_i = t_i' + 1$. Each object $S_j \in \mathcal{S}(\mathcal{T}(v_i, age_i))$ starts traversing its respective object tour $Object\_Tour(S_j)$ at the end of time step t_i to reach to the next transaction that uses it.

Theorem 6.1. *Algorithm* R-Off-Opt *correctly solves* offline relaxed PredOrderScheduling, *and achieves optimal execution time and communication cost.*

Relaxed Off-Eff**.** We study here offline *relaxed* PredOrderScheduling as in Section 4. We present a polynomial-time algorithm R-Off-Eff extending the polynomial-time algorithm Off-Eff and achieve $O(\log^2 n)$-competitiveness on both execution time and communication cost.

R-Off-Eff modifies Off-Eff as follows. R-Off-Eff does not need to rely on the transaction queue $DistTransQueue(\mathcal{T})$ and hence $findTrans(T(v_i, age_i))$ messages are not needed. Furthermore, $commit(T(v_i, age_i))$ message are not needed. Therefore, R-Off-Eff works based only on $DistObjTour(S_j)$ for each object $S_j \in \mathcal{S}$. The construction of $DistObjTour(S_j)$ for each $S_j \in \mathcal{S}$ is similar to Off-Eff.

Consider a transaction $T(v_i, age_i)$ that requires objects $\mathcal{S}(T(v_i, age_i)) \subseteq \mathcal{S}$. Let t_i' be the time step at which all the objects in $\mathcal{S}(T(v_i, age_i))$ are assembled at node v_i. Transaction $T(v_i, age_i)$ executes and commits at time step $t_i = t_i' + 1$. The objects in $\mathcal{S}(T(v_i, age_i))$ are then forwarded to their next transactions following their respective queues at the end of time step t_i.

Theorem 6.2. R-Off-Eff *correctly solves offline* relaxed PredOrderScheduling, *and is* $O(\log^2 n)$*-competitive in both execution time and communication cost.*

Relaxed Dyn**.** We study here dynamic *relaxed* PredOrderScheduling. We present an algorithm R-Dyn that achieves $O(D)$-competitiveness on both execution time and communication cost. R-Dyn is a modified version of Dyn (Section 5). Consider algorithm Dyn. Let $\mathcal{T}_t(v_{root})$ be the set of pending transactions at v_{root} at time step t. Node v_{root} asks all the transactions in $\mathcal{T}_t(v_{root})$ that do not share the objects to execute and commit, if all the shared objects needed by those transactions are available to v_{root}. We again have two cases as in Section 5. For case (i), we have the following.

Lemma 6.3. *Take any time step t. Suppose the transaction priorities depend on their arrival time. Let $T(v_j, age_j)$ be a transaction that arrives at t that uses object S_j. Consider a transaction $T(v_i, age_i)$ that arrives at time step $t_i < t$ that uses S_j.* Dyn *commits $T(v_i, age_i)$ before $T(v_j, age_j)$.*

We prove the following guarantee of R-Dyn for case (ii).

Lemma 6.4. *Suppose the transaction priorities do not depend on their arrival time.* R-Dyn *commits transactions in the increasing age order of $DistObjTour(S_j)$ at any time step t.*

7 EVALUATION

We have implemented Off-Opt, Off-Eff, and Dyn and evaluated them using a set of micro- and complex benchmarks. The implementation was done in Java. The experiments were performed on an Intel Core i7-7700K processor with 32 GB RAM, simulating two different communication graphs, namely *random* and *grid*. The total number of nodes are varied from 64 to 1024 for each graph. The total number of shared objects, transactions, and the transaction sizes vary based on a benchmark. The results presented are the average of 10 runs.

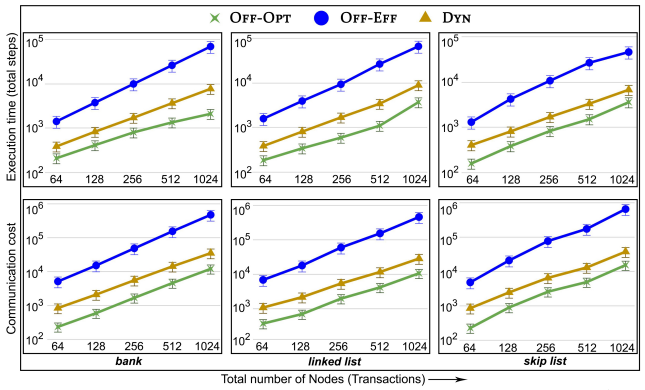

Figure 4: Time and communication of the algorithms (log scale) against three micro-benchmarks on random graphs.

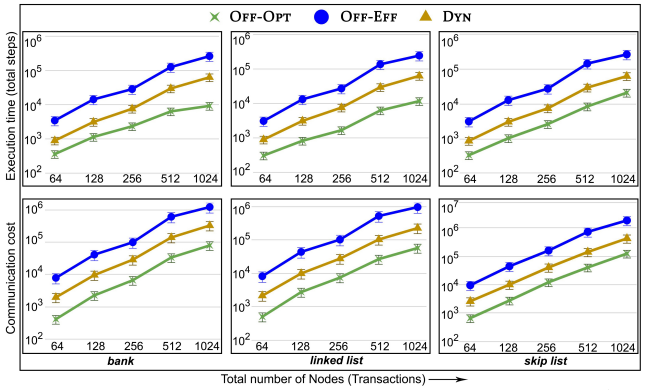

Figure 5: Time and communication of the algorithms (log scale) against three micro-benchmarks on grid graphs.

In the experiments, execution time is measured as the number of time steps taken to execute all the transactions. Communication cost is measured as the total distance the objects (and commit notifications) traverse in the respective communication graphs.

Results on Micro-benchmarks. We experimented Off-Opt, Off-Eff, and Dyn against three micro-benchmarks *bank*, *linked list*, and *skip list*. Figs. 4 and 5 provide the results in random and grid graphs, respectively.

Results on STAMP benchmarks. We experimented the algorithms against *intruder*, *genome*, and *vacation* out of 8 benchmarks available in the STAMP benchmark suite [20]. Figs. 6 and 7 provide the results in random and grid graphs, respectively.

Result Discussion. Both the execution time and communication cost results for Off-Opt are the minimum possible. Therefore, the results of Off-Eff and Dyn can be compared with the Off-Opt results. The results for Off-Eff are substantially better than the theoretical $O(\log^2 n)$ factor compared to the time and communication in Off-Eff. The trend can also be seen when we compare Dyn results with Off-Opt results. The results for Dyn are again orders of magnitude better than the theoretical $O(D)$ factor compared to the time and communication in Off-Opt. Moreover, Dyn has less execution time and less communication cost than the Off-Eff. This is because of $D < \log^2 n$ in the experiment. Therefore, the experimental results validate the efficacy of our algorithms for solving PredOrderScheduling in real application setting.

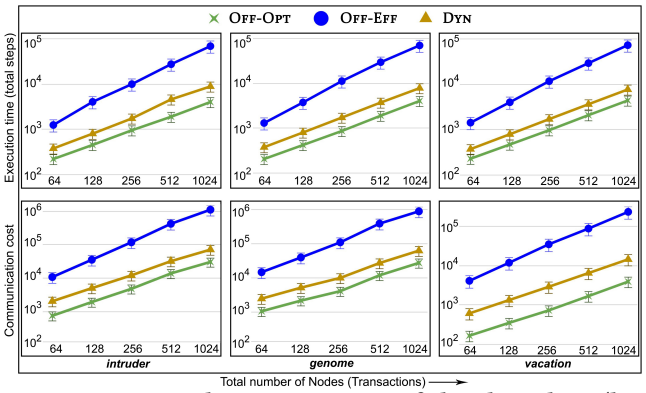

Figure 6: Time and communication of the algorithms (log scale) against three STAMP benchmarks on random graphs.

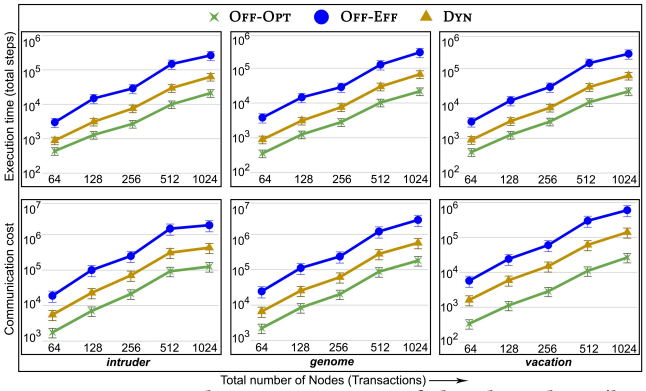

Figure 7: Time and communication of the algorithms (log scale) against three STAMP benchmarks on grid graphs.

8 CONCLUDING REMARKS

In this paper, we have studied for the very first time the predefined order scheduling problem of committing transactions according to their priorities in the data-flow model of transaction execution in distributed systems, minimizing simultaneously two fundamental performance metrics, namely execution time and communication cost. This problem finds applications in many areas, such as loop parallelization and state-machine-based distributed computing. We have provided a range of algorithms considering this problem in the offline and dynamic online settings as well as considering the relaxed version of the problem. Our results (for both relaxed and non-relaxed versions) are optimal in the offline setting with complete knowledge to poly-log competitive in the offline setting with partial knowledge to diameter competitive in the dynamic online setting. The experimental results validated the efficacy of the solutions in a set of simple micro-benchmarks and complex STAMP benchmarks. In future, it will be interesting to deploy the algorithms in real distributed system(s) and measure the wall clock results.

ACKNOWLEDGMENTS

This work is supported by the NSF grant CCF-1936450.

REFERENCES

[1] Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. 2010. Transactional Contention Management as a Non-Clairvoyant Scheduling Problem. *Algorithmica* 57, 1 (2010), 44–61.

[2] Hagit Attiya, Vincent Gramoli, and Alessia Milani. 2015. Directory Protocols for Distributed Transactional Memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*. 367–391.

[3] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. 2008. Software transactional memory for large scale clusters. In *PPoPP*. 247–258.

[4] Costas Busch, Maurice Herlihy, Miroslav Popovic, and Gokarna Sharma. 2015. Impossibility Results for Distributed Transactional Memory. In *PODC*. 207–215.

[5] Costas Busch, Maurice Herlihy, Miroslav Popovic, and Gokarna Sharma. 2017. Fast Scheduling in Distributed Transactional Memory. In *SPAA*. ACM, 173–182.

[6] Costas Busch, Maurice Herlihy, Miroslav Popovic, and Gokarna Sharma. 2018. Time-communication impossibility results for distributed transactional memory. *Distributed Computing* 31, 6 (2018), 471–487.

[7] Costas Busch, Maurice Herlihy, Miroslav Popovic, and Gokarna Sharma. 2020. Dynamic Scheduling in Distributed Transactional Memory. In *IPDPS*. 874–883.

[8] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Q. Le. 2013. Robust architectural support for transactional memory in the power architecture. In *ISCA*. 225–236.

[9] Michael J. Demmer and Maurice Herlihy. 1998. The Arrow Distributed Directory Protocol. In *DISC*. 119–133.

[10] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. 2011. Hardware transactional memory for GPU architectures. In *MICRO*. 296–307.

[11] Miguel A. Gonzalez-Mesa, Eladio Gutiérrez, Emilio L. Zapata, and Oscar G. Plata. 2014. Effective Transactional Memory Execution Management for Improved Concurrency. *TACO* 11, 3 (2014), 24:1–24:27.

[12] Vincent Gramoli, Rachid Guerraoui, and Vasileios Trigonakis. 2018. TM^{2}C: a software transactional memory for many-cores. *Distributed Computing* 31, 5 (2018), 367–388.

[13] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. 2005. Toward a Theory of Transactional Contention Managers. In *PODC*. 258–264.

[14] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-free Data Structures. In *ISCA*. 289–300.

[15] Maurice Herlihy and Ye Sun. 2007. Distributed transactional memory for metric-space networks. *Distributed Computing* 20, 3 (2007), 195–208.

[16] Sachin Hirve, Roberto Palmieri, and Binoy Ravindran. 2014. Archie: A Speculative Replicated Transactional System. In *Middleware*. 265–276.

[17] Intel. 2012. http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell.

[18] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.

[19] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. 2006. Exploiting Distributed Version Concurrency in a Transactional Memory Cluster. In *PPoPP*. 198–208.

[20] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC*. 35–46.

[21] Mohamed Mohamedin, Sebastiano Peluso, Masoomeh Javidi Kishi, Ahmed Hassan, and Roberto Palmieri. 2018. Nemo: NUMA-aware Concurrency Control for Scalable Transactional Memory. In *ICPP*. 38:1–38:10.

[22] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. 2015. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *ISCA*. 144–157.

[23] Pavan Poudel and Gokarna Sharma. 2020. GraphTM: An Efficient Framework for Supporting Transactional Memory in a Distributed Environment. In *ICDCN*. 11:1–11:10.

[24] Mohamed M. Saad, Masoomeh Javidi Kishi, Shihao Jing, Sandeep Hans, and Roberto Palmieri. 2019. Processing transactions in a predefined order. In *PPOPP*. 120–132.

[25] Mohamed M. Saad, Roberto Palmieri, and Binoy Ravindran. 2018. Lerna: Parallelizing Dependent Loops Using Speculation. In *SYSTOR*. 37–48.

[26] Mohamed M. Saad and Binoy Ravindran. 2011. Snake: Control Flow Distributed Software Transactional Memory. In *SSS*. 238–252.

[27] Gokarna Sharma and Costas Busch. 2012. A Competitive Analysis for Balanced Transactional Memory Workloads. *Algorithmica* 63, 1-2 (2012), 296–322.

[28] Gokarna Sharma and Costas Busch. 2014. Distributed Transactional Memory for General Networks. *Distrib. Comput.* 27, 5 (2014), 329–362.

[29] Gokarna Sharma and Costas Busch. 2015. A load balanced directory for distributed shared memory objects. *J. Parallel Distrib. Comput.* 78 (2015), 6–24.

[30] Nir Shavit and Dan Touitou. 1997. Software Transactional Memory. *Distrib. Comput.* 10, 2 (1997), 99–116.

[31] Alexandru Turcu, Binoy Ravindran, and Roberto Palmieri. 2013. Hyflow2: A High Performance Distributed Transactional Memory Framework in Scala. In *PPPJ*. 79–88.

[32] Richard M. Yoo and Hsien-Hsin S. Lee. 2008. Adaptive transaction scheduling for transactional memory systems. In *SPAA*. 169–178.

[33] Bo Zhang, Binoy Ravindran, and Roberto Palmieri. 2014. Distributed Transactional Contention Management as the Traveling Salesman Problem. In *SIROCCO*. 54–67.