Ordered Scheduling in Control-flow Distributed Transactional Memory

Pavan Poudel[1], Shishir Rai[2], Swapnil Guragain[2], and Gokarna Sharma[2][0000−0002−4930−4609]

[1] ATGWORK, Norcross, Georgia, USA
poudelpavan@gmail.com
[2] Kent State University, Kent, Ohio, USA
{srai,sguragai,gsharma2}@kent.edu

Abstract. Consider the *control-flow* model of transaction execution in a distributed system modeled as a communication graph where shared objects positioned at nodes of the graph are immobile but the transactions accessing the objects send requests to the nodes where objects are located to read/write those objects. The control-flow model offers benefits to applications in which the movement of shared objects is costly due to their sizes and security purposes. In this paper, we study the *ordered scheduling* problem of committing *dependent* transactions according to their predefined priorities in this model. The considered 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 ordered 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 traversed in accessing required shared objects. We design scheduling algorithms that are individually or simultaneously efficient for both the metrics and rigorously evaluate them through several benchmarks on random and grid graphs, validating their efficiency. To our best knowledge, this is the first study of ordered scheduling in the control-flow model of transaction execution.

1 Introduction

Concurrent processes (threads) need to synchronize to avoid introducing inconsistencies while accessing shared data objects. Traditional mechanisms of locks and barriers have well-known downsides, including deadlock, priority inversion, reliance on programmer conventions, and vulnerability to failure or delay. *Transactional memory* (TM) [16,37] 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, e.g., Intel's Haswell [22] and IBM's Blue Gene/Q [14], zEnterprise EC12 [27], and Power8 [8].

TM has been studied extensively for *multiprocessors*, where processors operate on a single shared memory and the latency to access (read/write) shared memory is the same

(and negligible) for each processor. However, recently, the computing trend is shifting toward *distributed multiprocessors*, where the memory access latency varies depending on the processor in which the thread executes and the physical segment of memory that stores the requested memory location. Therefore, the recent research focus is on how to support TM in distributed multiprocessors. Some proposals in this direction include TM^{2}C [13], NEMO [26], cluster-TM [3,24], GPU-TM [10], and HYFLOW [38].

TM is beneficial in distributed systems where data is spread across multiple nodes. For example, distributed data centers can use TM to simplify the burden of distributed synchronization and provide more reliable and efficient program execution while accessing data from remote nodes. *Distributed TM* (DTM) designed for such systems need to execute transactions effectively by taking into consideration the system's infrastructure. The network structure can play a crucial role in the DTM performance, since the data transactions access has to be reached across the network in a timely manner.

In this paper, we study ordered scheduling (ORDS) problem in distributed multiprocessors. We model distributed multiprocessors as an n-node connected, undirected, and weighted graph G, where each node denotes a processor and each edge denotes a communication link between processors. A set of w shared objects $\mathcal{S} := \{S_1, S_2, \ldots, S_w\}$ reside on the (possibly different) nodes of G. We consider the *control-flow* model [33], where objects are immobile but transactions send access requests to the nodes the required objects are located. Consider a set $\mathcal{T} := \{T(v_1, age_1), T(v_2, age_2), \ldots\}$ of transactions mapped (arbitrarily) to the nodes of G with each $T(v_i, age_i)$ accessing an arbitrary subset of the shared objects $\mathcal{S}(T(v_i, age_i)) \subseteq \mathcal{S}$, where age is an externally provided parameter that is unique for each transaction providing a priority order. We say transaction $T(v_i, age_i)$ is *dependent* on $T(v_j, age_j), age_j < age_i$, if at least an object read/write by $T(v_i, age_i)$ is being written by $T(v_j, age_j)$. The ORDS problem is to commit the dependent transactions in the age order. For example, transaction $T(v_i, age_i)$ that depends on $T(v_j, age_j), age_j < age_i$, commits only after $T(v_j, age_j)$ has been committed. Non-dependent transactions can execute and commit in parallel.

ORDS naturally arises in applications where producing (dependent) executions equivalent to a priority order is needed to satisfy/guarantee certain properties. Example applications include speculative loop parallelization and distributed computation using state machine approach [31]. In loop parallelization [32], 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 [19], 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 [23].

ORDS has been studied heavily in multiprocessors [11,31] where execution time is the only metric of interest. However, those studies focused on empirical studies and they do not extend to distributed multiprocessors as they do not consider latency. Recently, Poudel *et al.* [29] studied for the first time the ORDS problem in a distributed multiprocessor. However, they considered the *data-flow* model where transactions are immobile but the objects are mobile. Since the data-flow model is direct opposite of the control-flow model, the contributions in [29] do not apply to the control-flow model.

Contributions. In this paper, we design ORDS scheduling algorithms in the control-flow model and establish complementary results compared to [29] . We consider the *synchronous* communication model [6,7] where time is divided into discrete steps. We optimize two performance metrics: (i) *execution time* – the total time to execute and commit all the transactions, and (ii) *communication cost* – the total distance messages travel to access shared objects. A transaction's execution finishes as soon as it commits. The presented algorithms determine the time step when each transaction executes and commits. We measure the efficiency using a widely-studied notion of *competitiveness* – the ratio of total time (communication cost) for a designed algorithm to the minimum time (communication cost) achievable by an optimal scheduling algorithm.

Specifically, we have the following five contributions:

1. We provide an impossibility result showing that the optimal execution time and optimal communication cost can not be achieved simultaneously. (**Section 3**)
2. For the offline version, we provide two algorithms, one with optimal execution time and another with 2-competitive on communication cost. (**Section 4**)
3. For the partial dynamic version with the knowledge of transactions and their priorities but not the shared objects, we provide an $O(\log^2 n)$–competitive algorithm for both execution time and communication cost. (**Section 5**)
4. For the fully dynamic version with transactions arriving over time, we provide an $O(D)$–competitive algorithm for both execution time and communication cost, where D is the diameter of the graph G. (**Section 6**)
5. We implement and rigorously evaluate the designed algorithms through micro-benchmarks and complex STAMP benchmarks on random and grid graphs, which validate the efficiency of the designed algorithms. (**Section 7**)

Techniques. For the offline version, the optimal time algorithm sends access requests in parallel following the shortest paths in G. The 2-competitive communication cost algorithm sends (combined) access requests through a minimum Steiner tree that connects the graph nodes containing the required objects.

In the partial dynamic version (with the knowledge of transactions and their priorities but not the shared objects), the proposed algorithm exploits the concept of *distributed directory protocols* [17,35]. Particularly, the directory protocol technique based on the hierarchical partitioning of the graph into clusters is used. This technique guarantees that the object access cost for a transaction is within an $O(\log^2 n)$ factor from the cost of minimum Steiner tree for that transaction. The directory protocol technique is then extended to the dynamic version guaranteeing $O(D)$-competitiveness without knowing transactions and their priorities a priori. This bound is interesting since the hierarchical partitioning technique used in the partial dynamic version is shown to only provide $O(D \log^2 n)$-competitive bound for the fully dynamic version. Therefore, the dynamic algorithm uses the directory protocol running on a spanning tree.

Related Work. Gonzalez-Mesa *et al.* [11] introduced the ORDS problem for multiprocessors and Saad *et al.* [31] presented three improved algorithms and evaluated them through empirical studies. Transaction scheduling with no predefined ordering is widely-studied in multiprocessors providing provable upper and lower bounds, and impossibility results [1,34], besides several other scheduling algorithms that were only

evaluated experimentally [39]. The multiprocessor ideas are not suitable for distributed multiprocessors as they do not deal with a crucial metric, communication cost.

Many previous studies on transaction scheduling in distributed multiprocessors, e.g., [2,4,5,6,7,35,36], considered the data-flow model. The papers [17,35,40] focused on minimizing communication cost. Execution time minimization is considered by Zhang *et al.* [40]. Busch *et al.* [4] considered minimizing both execution time and communication cost. Busch *et al.* [5] considered special topologies (e.g., grid, line, clique, star, hypercube, butterfly, and cluster) and provided offline algorithms minimizing execution time and communication cost. Recently, Busch *et al.* [7] provided dynamic (online) algorithms. However, all these works have no predefined ordering requirement.

Some papers considered the hybrid model that combines data-flow with control-flow. Hendler *et al.* [15] studied a lease based hybrid DTM which dynamically determines whether to migrate transactions to the nodes that own the leases or to demand the acquisition of these leases by the node that originated the transaction. Palmieri *et al.* [28] presented a comparative study of data-flow versus control-flow models.

2 Model and Preliminaries

Graph. We consider a distributed multiprocessor $G = (V, E, \mathfrak{w})$ 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 $\mathfrak{w} : E \rightarrow \mathbb{Z}^+$. A *path* p in G is a sequence of nodes (with respective edges between adjacent nodes) with $\mathsf{length}(p) = \sum_{e \in p} \mathfrak{w}(e)$. We assume that G is connected and $\mathsf{dist}(u, v)$ denotes the shortest path length (distance) between two nodes $u, v \in G$. The *diameter* $D := \max_{u,v \in G} \mathsf{dist}(u, v)$, the maximum shortest path distance between two nodes $u, v \in G$. The communication links are *bidirectional* – messages can be sent in both directions. Both the nodes and links are non-faulty and the links deliver messages in FIFO order. There is no bandwidth restriction on the edges, i.e., the messages can be of any size and any number of messages can traverse an edge at any time. 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 [5,6,7]. For an edge $e = (u, v) \in E$, it takes $\mathfrak{w}(e)$ time steps to transfer a message msg from u to v (and vice-versa); the *communication cost* contributed by msg is $\mathfrak{w}(e)$.

Transactions. Let $\mathcal{S} = \{S_1, S_2, \ldots, S_w\}$ denote the w shared objects residing on nodes of G. Each object has some value which can be read/written. The node of G where an object S_i is currently positioned is called the *owner* of S_i, denoted as $owner(S_i)$. A transaction $T(v_i, age_i)$ is an atomic block of code mapped 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. To simplify the analysis, we assume that each object has a single copy (for both read/write). We assume that each node runs a single thread and issues transactions sequentially.

Control-flow Model. The model works in two steps:

i. **Object Access Phase:** Transaction $T(v_i, age_i)$ sends access request to the owner node of each object $S_i \in \mathcal{S}(T(v_i, age_i))$ and the owner node of S_i replies back a

success or *failure* message to v_i. A success message for S_i means that $T(v_i, age_i)$ was able to read/write S_i, whereas a failure message means denied access.

ii. **Validation Phase:** If transaction $T(v_i, age_i)$ receives *success* message from owner node of each $S_i \in \mathcal{S}(T(v_i, age_i))$, then it commits. If $T(v_i, age_i)$ receives at least a *failure* message, then it either aborts or waits.

Transaction Execution and Conflicts. For an access request received for S_j from $T(v_i, age_i)$, $owner(S_j)$ handles that request by allowing $T(v_i, age_i)$ to read or write (update) S_j and replies a *success* message back to v_i. If $owner(S_j)$ receives two access requests for object S_j at the same time and at least one of them is a write request, *conflict* is said to be occurred between transactions accessing S_j. $owner(S_j)$ handles such type of simultaneous access requests by denying at least one request. In case $owner(S_j)$ denies the access request, it replies a *failure* message back to node v_i.

Performance Metrics. Let $\mathcal{E}$ be an execution schedule following an algorithm $\mathcal{A}$.

Definition 1 (*Execution Time).* *For a set of transactions $\mathcal{T}$, the total time for $\mathcal{E}$ is the time elapsed until the last transaction finishes its execution in $\mathcal{E}$. The execution time of algorithm $\mathcal{A}$ is the maximum time over all possible executions for $\mathcal{T}$.*

Definition 2 (*Communication Cost).* *For a set of transactions $\mathcal{T}$, the communication cost of $\mathcal{E}$ 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 ORDS **Problem.** Each transaction $T(v_i, age_i)$ is assigned age, age_i, before it is activated, and the age signifies the transaction commit order under dependencies. Following [11,29,31], parameter age is (i) *unique* – no two transactions can have the same age, (ii) *non-modifiable* – it never changes once assigned, and (iii) *externally determined* – it does not depend on transaction execution.

For transaction $T(v_i, age_i)$, $\mathcal{S}(T(v_i, age_i)) := write(\mathcal{S}(T(v_i, age_i))) \cup read(\mathcal{S}(T(v_i, age_i)))$. We say $T(v_i, age_i)$ is *dependent* on $T(v_j, age_j), age_j < age_i$, if $(write(\mathcal{S}(T(v_i, age_i))) \cap \mathcal{S}(T(v_j, age_j)) \neq \emptyset) \vee (read(\mathcal{S}(T(v_i, age_i))) \cap write(\mathcal{S}(T(v_j, age_j))) \neq \emptyset)$. I.e., at least an object read/write by $T(v_i, age_i)$ is being written by $T(v_j, age_j)$. $T(v_i, age_i)$, if dependent on $T(v_j, age_j)$, can commit only after $T(v_j, age_j)$ commits. Formally,

Definition 3 (*The* ORDS ***problem).* *Given a set of transactions $\mathcal{T} := \{T(v_1, age_1), T(v_2, age_2), \ldots\}$ mapped (arbitrarily) to the nodes of G, commit dependent transactions in $\mathcal{T}$ in the increasing order of age in the control-flow model.*

3 Impossibility Result

Consider a star graph G as shown in Fig. 1 with eight rays going out from the center node. Let there be three nodes on each ray (except the center node). Additionally, let the end nodes of consecutive rays are connected. Suppose there are six objects a, b, c, d, e, and f positioned on six consecutive end nodes, and a transaction T is mapped at the center node and it requests all six objects. All edges have unit weight.

Theorem 1. *There are transaction scheduling instances for which execution time and communication cost cannot be minimized simultaneously in the control-flow model.*

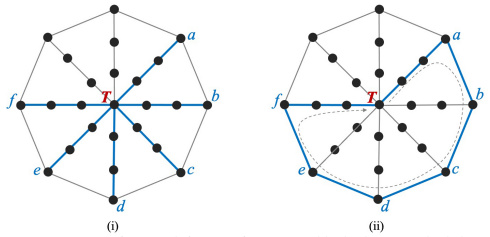

Fig. 1: **(i)** Transaction T accessing objects in parallel through blue colored paths, **(ii)** T accessing objects sequentially again through blue colored paths.

Proof. When transaction T sends object access requests in parallel, they can be reached in 3 steps. In next 3 steps, T gets reply messages from all the object nodes, and in one additional step, it can execute and commit. This gives optimal execution time of 7 steps. However, total communication cost becomes 36 (3 steps to reach requests to objects and 3 steps to receive replies back). Alternatively, let T sends the object access request (combined) first to a and then to b, c, d, e, f in order. Moreover, the reply from a is also sent together with the (combined) request towards b and so on with others. Thus, when the request reaches f, the replies from a, b, c, d, e also reach there. Now, from f, all the replies traverse the ray connecting f and T. The communication cost becomes 11 steps, which is optimal. Total execution time becomes $11+1 = 12$, which is sub-optimal. □

4 Offline Algorithms

In this section, we study the offline version of the ORDS problem. We present two algorithms, one called OFFEXEC that achieves optimal execution time and another called OFFCOMM that is 2-competitive on communication cost.

Execution Time Algorithm OFFEXEC**.** OFFEXEC accesses required objects for each transaction in parallel. All transactions in $\mathcal{T}$ are initiated at time step $t = 0$. Therefore, at $t = 0$, all the transactions in $\mathcal{T}$ send requests to access the required objects to the respective owner nodes following the shortest paths. Each owner node then replies *success* message for every request (after performing the read/write operation) respecting the age order and dependency of the transactions at corresponding owner node.

For transaction $T(v_i, age_i)$ at node v_i, let $\mathcal{S}(T(v_i, age_i)) \subseteq \mathcal{S}$ be the set of objects it needs. $T(v_i, age_i)$ sends corresponding access requests to $owner(S_j)$ of each object $S_j \in \mathcal{S}(T(v_i, age_i))$ following the shortest path from v_i to $owner(S_j)$. After the access request reaches $owner(S_j)$, $owner(S_j)$ sends *success* message back to v_i as soon as $T(v_i, age_i)$ is able to read/write that object respecting the age order. Specifically, there can be two cases: (i) There is no $T(v_k, age_k), age_k < age_i$, in $\mathcal{T}$ which also wants to access S_j, then $owner(S_j)$ immediately sends *success* message back to v_i (ii) There is another transaction $T(v_k, age_k), age_k < age_i$, in $\mathcal{T}$ that conflicts with $T(v_i, age_i)$ while accessing S_j, then $owner(S_j)$ sends *success* message to v_k first and to v_i in the next time step. When v_i receives *success* messages from all $owner(S_j)$, $T(v_i, age_i)$ finishes its execution and commits.

Let $t_i^{S_j}$ be the time step at which $owner(S_j)$ of object $S_j \in S(T(v_i, age_i))$ replies *success* message back to node v_i corresponding to the request sent by $T(v_i, age_i)$. Then, $t_i^{S_j} = \max\{t^{S_j}_{prev(T(v_i,age_i))} + 1, \mathsf{dist}(v_i, owner(S_j))\}$, where $t^{S_j}_{prev(T(v_i,age_i))}$ is the time step at which $owner(S_j)$ replies to the dependent transaction of $T(v_i, age_i)$ that

is immediately previous to $T(v_i, age_i)$ in the age order. For the lowest aged transaction $T(v_1, age_1)$, $t_1^{S_j} = \mathsf{dist}(v_1, owner(S_j))$.

Let CT_i be the time step at which transaction $T(v_i, age_i) \in \mathcal{T}$ commits. Then,

$$CT_i = \begin{cases} CT_{prev(T(v_i,age_i))} + 1, & \text{if } t'_i < CT_{prev(T(v_i,age_i))} \\ t'_i + 1, & \text{otherwise.} \end{cases}$$

where $CT_{prev(T(v_i,age_i))}$ is the time at which the transaction dependent to $T(v_i, age_i)$ that is immediately previous to $T(v_i, age_i)$ in the age order commits and

$$t'_i = \max_{S_j \in \mathcal{S}(T(v_i,age_i))} (t_i^{S_j} + \mathsf{dist}(v_i, owner(S_j))).$$

For the lowest aged transaction $T(v_1, age_1)$, $CT_1 = \max_{S_j \in \mathcal{S}(T(v_1,age_1))} 2 \cdot \mathsf{dist}(v_1, owner(S_j)) + 1$.

Theorem 2. OFFEXEC *achieves optimal execution time.*

Proof. The execution time depends on two factors. First, how long does a transaction take to access required objects and second, when does each transaction commit? In OFFEXEC, each transaction accesses required object using the shortest path in G which is thus optimal. Now, we need to show that each transaction commits at the earliest possible time. First, let there is no conflict between any transactions in $\mathcal{T}$. Then all the transactions can access required objects in parallel and as soon as each transaction receives success messages from the owner nodes of each required object, it can commit. The total execution time becomes

$$\max_{T(v_i,age_i) \in \mathcal{T}} \left\{ \max_{S_j \in \mathcal{S}(T(v_1,age_1))} 2 \cdot \mathsf{dist}(v_i, owner(S_j)) + 1 \right\}$$

which is optimal.

Now, let there are conflicts between transactions in $\mathcal{T}$ when accessing objects. Let $\mathcal{T} = \{T(v_1, age_1), T(v_2, age_2), \ldots, T(v_n, age_n)\}$ be the set of transactions. Let a dependency graph $H = (V_H, E_H)$ holds the dependency between the conflicting transactions where the nodes V_H represent transactions in $\mathcal{T}$ and the directed edges E_H represent dependencies between the transactions. The edge $(T(v_i, age_i), T(v_j, age_j)) \in E_H$, where $age_i < age_j$, represents a dependency between $T(v_i, age_i)$ and $T(v_j, age_j)$ such that $T(v_j, age_j)$ can commit only after $T(v_i, age_i)$ commits. The ORDS problem requires the dependent transactions to commit in their age order. The diameter D_H of H provides the longest chain of dependent transactions and the total execution time of any optimal algorithm will be the time required by all the transactions that belong to D_H to commit. During the execution of OFFEXEC, for each transaction $T(v_i, age_i)$, if there is no any dependent transaction in H or all the dependent transactions in H have already been committed, then $T(v_i, age_i)$ can commit as soon as it receives success messages from the owner nodes of all required objects. Note that both, object access requests and success messages, are sent through the shortest paths in G. When the highest age transaction that belongs to D_H of H commits, OFFEXEC finishes. Hence, the total execution time is optimal. □

Theorem 3. OFFEXEC *is k-competitive in communication cost, where k is the maximum number of shared objects accessed by a transaction in $\mathcal{T}$.*

Communication Cost Algorithm OFFCOMM**.** In OFFCOMM, we convert the execution of each transaction to a Minimum Steiner Tree (MST) [20,21]. Steiner trees have been extensively studied in the context of weighted graphs [12]. Given a graph $G = (V, E)$ and a subset $P \subseteq V$, a *Steiner tree* spans through P. The Steiner tree problem in our case is to find a Steiner tree that connects all the vertices of P with the minimum possible total weight. Computing MST is known to be NP-Hard. We follow the algorithm of Takahashi and Matsuyama [18] which provides $2(1 - 1/|P|)$–approximation for MST. The algorithm of [18] constructs a Steiner tree as follows:

- Start from a participant node in P.
- Find the next participant that is closest to the current tree.
- Join the closest participant to the closest node of the tree.
- Repeat until all nodes in P are connected.

Now, we discuss how MST is constructed for each transaction in $\mathcal{T}$. Let $\mathcal{S}(T(v_i, age_i)) \subseteq \mathcal{S}$ be the set of objects required by a transaction $T(v_i, age_i) \in \mathcal{T}$. Let $P_i \subseteq V$ contains node v_i and the owner node of each object $S_j \in \mathcal{S}(T(v_i, age_i))$ (i.e., $P_i := (\forall_{S_j \in \mathcal{S}(T(v_i, age_i))} owner(S_j)) \cup v_i$). Now, the problem is to find a MST that connects the nodes in P_i which is constructed by following the algorithm of [18] and is denoted as MST_i. Then, $T(v_i, age_i)$ sends object access requests in MST_i. The total message cost incurred by transaction $T(v_i, age_i)$ is $2.|MST_i|$. That means, messages visit each edge of MST_i exactly twice, one for sending access request and the other for receiving reply (*success* or *failure*) message from each owner node.

Instead of sending requests individually to access the objects in $S(T(v_i, age_i))$, $T(v_i, age_i)$ sends them collectively in MST_i. Each neighboring node recursively sends the request to the next neighbor in MST_i until the request reaches all the owner nodes of the required objects. To be specific, if $v_p, v_q \in MST_i$ be any two owner nodes of objects which share a common path from v_i up to some intermediate node v_s, then the requests to v_p and v_q from v_i are sent collectively up to v_s as a single message. The request is then divided into two at v_s and they are forwarded separately towards v_p and v_q. When all the access requests reach respective owner nodes, the reply messages are collected in the opposite direction. Here, each intermediate node which had initially sent access requests to the neighboring nodes later collects the reply messages from those neighboring nodes and returns them collectively to the ancestor node. When v_i receives reply messages from all the neighboring nodes in MST_i, $T(v_i, age_i)$ commits (provided that all the reply messages are *success* messages).

The OFFCOMM algorithm works as follows. It produces a conflict-free execution schedule. At time step $t = 0$, each transaction $T(v_i, age_i)$ sends access requests to required objects following its corresponding MST_i. When the access request reaches $owner(S_j)$, $owner(S_j)$ sends *success* message back to v_i as soon as $T(v_i, age_i)$ is able to read/write that object respecting the age order of the dependent transactions. Let $\mathsf{dist}_{MST_i}(v_i, v_j)$ represents the distance between nodes v_i and v_j following the shortest path in MST_i. Then, for each $T(v_i, age_i) \in \mathcal{T}$, $owner(S_j)$ of each $S_j \in S(T(v_i, age_i))$ replies success message to v_i at time step: $t_i^{S_j} = \max\{t_{prev(T(v_i, age_i))}^{S_j} + 1, \mathsf{dist}_{MST_i}(v_i, owner(S_j))\}$, where $t_{prev(T(v_i, age_i))}^{S_j}$ is the time step at which $owner(S_j)$ replies to the dependent transaction of $T(v_i, age_i)$ that is immediately previous to $T(v_i, age_i)$ in the age order.

The commit time step CT_i for each $T(v_i, age_i)$ is:

$$CT_i = \begin{cases} CT_{prev(T(v_i,age_i))} + 1, & \text{if } t'_i < CT_{prev(T(v_i,age_i))} \\ t'_i + 1, & \text{otherwise.} \end{cases}$$

where $CT_{prev(T(v_i,age_i))}$ is the time at which the transaction dependent to $T(v_i, age_i)$ that is immediately previous to $T(v_i, age_i)$ in the age order commits and $t'_i = \max_{S_j \in \mathcal{S}(T(v_i,age_i))}(t_i^{S_j} + \mathsf{dist}_{MST_i}(v_i, owner(S_j)))$.

Theorem 4. OFFCOMM *is 2-competitive in communication cost.*

Proof. Let MST_i be the minimum cost Steiner tree constructed for transaction $T(v_i, age_i)$ in OFFCOMM. Let $\mathsf{dist}_{MST_i}(v_x, v_y)$ be the shortest path distance between v_x and v_y in MST_i. If $\mathsf{dist}(v_x, v_y)$ be the shortest path distance in G, then we have: $\mathsf{dist}_{MST_i}(v_x, v_y) \leq 2 \cdot \mathsf{dist}(v_x, v_y)$. Since OFFCOMM follows the shortest paths in respective MSTs for accessing required objects, the communication cost $C_{T(v_i,age_i)}$ of executing each transaction $T(v_i, age_i) \in \mathcal{T}$ is: $C_{T(v_i,age_i)} = 2 \cdot C_{opt}^{T(v_i,age_i)}$, where $C_{opt}^{T(v_i,age_i)}$ is the cost of any optimal communication algorithm for executing $T(v_i, age_i)$ that accesses required objects following the shortest paths in G. If C_{total} and C_{opt} be the total communication costs of OFFCOMM and any optimal algorithm, respectively, such that $C_{opt} = \sum_{T \in \mathcal{T}} C_{opt}^T$, then, $C_{total} = \sum_{T \in \mathcal{T}} C_T = \sum_{T \in \mathcal{T}} 2 \cdot C_{opt}^T = 2 \cdot C_{opt}$. □

Theorem 5. OFFCOMM *is r-competitive in execution time, where r is the maximum stretch of MST computed for each transaction in $\mathcal{T}$ which is given by:*

$$r = \max_{T(v_i,age_i) \in \mathcal{T}} \left\{ \max_{S_j \in \mathcal{S}(T(v_i,age_i))} \frac{\mathsf{dist}_{MST}(v_i, owner(S_j))}{\mathsf{dist}(v_i, owner(S_j))} \right\}.$$

5 Partial Dynamic Algorithm

Here we study the partial dynamic version of the ORDS problem, where a priori knowledge on transactions and their priorities is available, but not the shared objects they access and their locations. All transactions arrive at time $t = 0$. Thus, the following two tasks are additional to the offline version:

i. Determine the owner nodes of all the shared objects that a transaction requests.
ii. Determine the node where the next transaction in the commit order is located and the path to reach that node.

We present an efficient algorithm PARTDYN using distributed directory protocol technique [2,9,17,30,35]. We compute two distributed queues, the first helps transactions accessing required objects and the second helps sending commit messages to the next dependent transaction in age order. The first is called *distributed object queue* where *object access tours* are constructed for each transaction. The second is called *distributed transaction queue* that satisfies the commit order of transactions. Each transaction sends commit message to the next transaction in order following the path in its respective transaction tour in the distributed transaction queue. We use the hierarchy-of-clusters-based overlay tree ($\mathcal{OT}$) (discussed next) for the computation of both queues.

Overlay Tree $\mathcal{OT}$ Construction. The well-known approaches for $\mathcal{OT}$ construction are based on either a *spanning tree* or a *hierarchy of clusters* on G. The spanning tree was used in directory protocols [9,2] and the hierarchy of clusters was used in directory protocols [17,35,36].

Both approaches work, however, 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, in the following, we discuss the construction of hierarchy-of-clusters-based overlay tree $\mathcal{OT}$. 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 (i.e., $2^\ell, 0 \leq \ell \leq H_1$). 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 algorithms, such as a *hierarchical sparse cover* of G, that give a cluster hierarchy $\mathcal{Z}$ of H_1 layers with $f(\ell) = O(\ell \log n)$ and $g(\ell) = O(\log n)$. This construction was used in the directory protocol, SPIRAL, by Sharma *et al.* [35], where additionally, 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. Suppose 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. 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 [35].

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

Lemma 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),* $\mathsf{length}(p(u)) \leq O(2^\ell \log^2 n)$.

Computing Distributed Transaction Queue. We denote the distributed transaction queue by $DTQueue(\mathcal{T})$. To construct $DTQueue(\mathcal{T})$, each transaction $T(v_i, age_i)$ sends a $findT(T(v_i, age_i))$ message in its upward path $p(v_i)$ in $\mathcal{OT}$. The $findT(T(v_i, age_i))$ message contains information about the required objects by $T(v_i, age_i)$ and moves upward until it meets the similar messages sent by it's previous and next conflicting transactions in age order. When two messages $findT(T(v_i, age_i))$ and $findT(T(v_j, age_j))$ meet at some node v_k, it can easily be found that whether $T(v_i, age_i)$ and $T(v_j, age_j)$ conflict with each other or not by looking at the information of required objects for each of them. When such meetings happen for all $findT(prev(T(v_i, age_i))$, $findT(T(v_i, age_i))$, and $findT(next(T(v_i, age_i)))$, $1 \leq i \leq n$, the computation of $DTQueue(\mathcal{T})$ is completed.

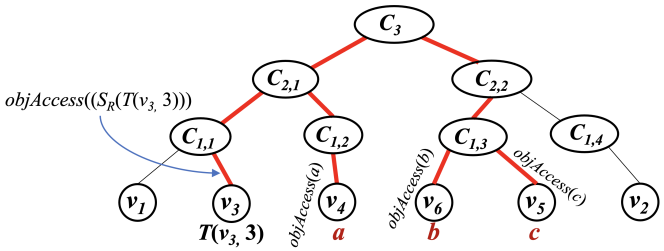

Fig. 2: Illustration of computation of distributed object queue for transaction $T(v_3, 3)$ requiring objects (a, b, c). $T(v_3, 3)$ sends $objAccess(S_R(T(v_3, 3)))$ message in its upward path to cluster $C_{1,1}$ which recursively sends it to $C_{2,1}$. $C_{2,1}$ contains the owner node of object a (i.e., v_4), thus sends $objAccess(a)$ message to v_4. Then, after removing a from $S_R(T(v_3, 3))$, $C_{2,1}$ sends $objAccess(S_R(T(v_3, 3)))$ message to cluster C_3. C_3 sends the message downward until the requests reach nodes v_5 and v_6. Later, all three nodes v_4, v_5, and v_6 reply *success* messages which are combined at clusters $C_{1,3}$ and $C_{2,1}$, and finally reach node v_3. Then $T(v_3, 3)$ commits. The edges traversed by the messages are highlighted in red.

The upward paths $p(v_i)$ and $p(v_j)$ for the two consecutive dependent transactions $T(v_i, age_i)$ and $T(v_j, age_j)$ intersect at some node v_k at some layer $l > 0$. Transaction $T(v_i, age_i)$ sends a commit message to $T(v_j, age_j)$ by first sending it upward in $p(v_i)$ up to v_k and then sending the message downward in $p(v_j)$ from v_k up to node v_j. The following theorem follows from the hierarchy of clusters based $\mathcal{OT}$.

Theorem 6. *If d is the shortest path distance between nodes $v_i, v_j \in G$, then the distance between v_i, v_j following the upward paths $p(v_i)$ and $p(v_j)$ in $\mathcal{OT}$ is $O(d \cdot \log^2 n)$.*

Computing Distributed Object Queues. Distributed object queue for each transaction $T(v_i, age_i) \in \mathcal{T}$ is denoted as $DOQueue(T(v_i, age_i))$. $DOQueue(T(v_i, age_i))$ contains object tour(s) to access the object(s) requested by $T(v_i, age_i)$.

$DOQueue(T(v_i, age_i))$ is constructed as follows. Let $\mathcal{S}_R(T(v_i, age_i)) \subseteq \mathcal{S}(T(v_i, age_i))$ be the set of objects required by $T(v_i, age_i)$ that are not present on v_i. $T(v_i, age_i)$ sends $objAccess(S_R(T(v_i, age_i)))$ message in its upward path $p(v_i)$. Let at some level $l > 0$, $objAccess(\mathcal{S}_R(T(v_i, age_i)))$ reaches a cluster with node v_j that contains an object $S_j \in \mathcal{S}_R(T(v_i, age_i))$. Then the leader of the cluster (say v_l) forwards $objAccess(S_j)$ to the node v_j downward in the path $p(v_j)$. The leader also removes object S_j from $\mathcal{S}_R(T(v_i, age_i))$ and forwards $objAccess(\mathcal{S}_R(T(v_i, age_i)))$ message upward in the path $p(v_i)$ if $\mathcal{S}_R(T(v_i, age_i))$ is not empty. This process continues until $\mathcal{S}_R(T(v_i, age_i))$ becomes empty and by that time, the computation of $DOQueue(T(v_i, age_i))$ is completed.

Later, during the execution of $T(v_i, age_i)$, when the object access request $objAccess(S_j)$ reaches the owner node of S_j, $owner(S_j)$, $T(v_i, age_i)$ performs read or write operation on S_j. After the read or write operation is completed, v_j replies a *success* message back following the previous path in the opposite direction (i.e., upward from v_j to the leader node v_l in $p(v_j)$). Each leader node when receives reply messages from the owner nodes of objects, combines them into a single message and sends it back downward in the path $p(v_i)$ to node v_i. The leader node waits to combine the reply message until it receives reply messages from all the paths that it has sent previously the access requests. Fig. 2 illustrates this idea.

Algorithm PARTDYN**.** PARTDYN starts with computing distributed object queues $DOQueue(T(v_i, age_i))$ for each transaction $T(v_i, age_i) \in \mathcal{T}$ and distributed transaction queue $DTQueue(\mathcal{T})$. $DOQueue(T(v_i, age_i))$ contains object tours to access all the required objects in $\mathcal{S}(T(v_i, age_i))$.

All the transactions that do not depend on any lower aged transactions start execution at time $t = 0$. $T(v_1, age_1)$ starts at $t = 0$ and sends object access requests recursively following object tours in $DOQueue(T(v_1, age_1))$. Then, for each object $S_j \in \mathcal{S}(T(v_1, age_1))$, $objAccess(S_j)$ reaches the owner node $owner(S_j)$. $T(v_1, age_1)$ performs read or write operation on all S_j and a *success* message from each $owner(S_j)$ is replied back following the object tours in the backward direction. $T(v_1, age_1)$ commits after it receives *success* messages from all the owner nodes of required objects (possibly in combined form). Let $T(v_1, age_1)$ commits at time step $t_1 > 0$. $T(v_1, age_1)$ sends commit message $commit(T(v_1, age_1))$ to the next conflicting transaction in age order $next(T(v_1, age_1)) = T(v_k, age_k), age_k > age_i$, by following upward paths in $DTQueue(\mathcal{T})$. When $T(v_k, age_k)$ receives commit messages from all the dependent transactions, $T(v_k, age_k)$ executes and commits at time step $t_k > t_1$ and sends $commit(T(v_k, age_k))$ message to $next(T(v_k, age_k))$. The process continues until the highest aged transaction $T(v_h, age_h)$ commits at some time step t_h.

Theorem 7. PARTDYN *is* $O(\log^2 n)$*-competitive in both execution time and communication cost.*

6 Fully Dynamic Algorithm

Here, we study ORDS with no a priori knowledge on transactions, their priorities, the shared objects they access, and their initial locations. Additionally, transactions arrive at different nodes of G arbitrarily over time. Once a transaction arrives at some node v_i, it knows the priority (i.e., age) of that transaction and the objects needed by it. We present an algorithm DYN that achieves $O(D)$ competitive ratio in both execution time and communication cost. Algorithm DYN works on top of a spanning-tree-based overlay tree, denoted as OT_{ST}. Let v_{root} be the root node of $\mathcal{OT}_{ST}$. For any node v, the upward path $p(v)$ in $\mathcal{OT}_{ST}$ is the path obtained by connecting the parent nodes in ST from node v up to the root v_{root}. DYN executes in two phases:

- **Phase 1 – Object Advertisement** in which each node of graph G is advertised with the locations of all the objects.
- **Phase 2 – Transaction Execution** in which transactions are executed and committed according to age order.

Phase 1 – Object Advertisement. The object advertisement phase makes each node of G know the locations of all the shared objects. Later, when a transaction at node v_i needs some object S_j, v_i can forward object access request to the owner node of that object. The ownership of each object is advertised in the form of a hash map where each key-value pair represents $(objID, nodeID)$, where $objID$ is the ID of an object located at node $v \in V$ and $nodeID$ is the ID of v.

Execution starts from leaf nodes of $\mathcal{OT}_{ST}$. Each leaf node v_l sends a hash map $(objID, nodeID)$. If v_l contains no object, v_l sends an empty hash map. Also, if v_l contains more than one object, it sends a hash map with multiple key-value pairs. When

a parent node v_{p1} receives hash maps from all its child nodes, v_{p1} merges those into a single hash map and appends new key-val pair(s) if it contains any object(s). The updated hash map is then sent upward to the next parent node v_{p2}. v_{p2} again merges all hash maps into a single one after receiving from all the child nodes. This process is repeated until the current node is the root v_{root}. When v_{root} receives hash maps from all of its child nodes, it merges them into a single hash map and replies back the updated hash map to all the child nodes recursively. This phase ends when all the leaf nodes receive updated hash map containing all $(objID, nodeID)$ pairs.

Lemma 3. *Phase 1 finishes in $O(D)$ time steps with communication cost $O(n)$.*

Phase 2 – Transaction Execution. Let H be the height of $\mathcal{OT}_{ST}$, $H \leq D$. As soon as transaction $T(v_i, age_i)$ is initiated, it sends an arrival message $T_{arrival}(T(v_i, age_i), t_i)$ to v_{root} following the upward path $p(v_i)$, where t_i is the time step at which $T(v_i, age_i)$ arrives at node v_i. Let $\mathcal{T}_t(v_{root})$ be a list maintained by v_{root} which contains the information of pending transactions at time step t sorted by arrival time. The arrival message $T_{arrival}(T(v_i, age_i), t_i)$ sent from node v_i reaches v_{root} in $\leq H$ time steps. Thus, when v_{root} receives a transaction arrival message $T_{arrival}(T(v_i, age_i), t_i)$ at some time step $t_r \geq t_i$, it includes $T(v_i, age_i)$ in $\mathcal{T}_t(v_{root})$ at time step $t'_i = t_i + H$.

Let $T(v_x, age_x) \in \mathcal{T}_t(v_{root})$ be the lowest aged transaction in $\mathcal{T}_t(v_{root})$ at time t. v_{root} sends $startExec(T(v_x, age_x))$ message to node v_x to execute $T(v_x, age_x)$. $T(v_x, age_x)$ sends object access requests to the owner nodes of $\mathcal{S}(T(v_x, age_x))$. When $T(v_x, age_x)$ successfully accesses all the required objects in $\mathcal{S}(T(v_x, age_x))$, it commits and sends a commit message to v_{root}. After that, v_{root} removes $T(v_x, age_x)$ from $\mathcal{T}_t(v_{root})$ and schedules next conflicting transaction in the age order to execute. Note that, v_{root} can schedule multiple transactions together which are not dependent on any lower aged transactions or receive commit messages from all the dependent transactions during the execution. Phase 2 finishes when all transactions in $\mathcal{T}$ commit.

Lemma 4. *In Phase 2, each transaction finishes its execution in $O(D)$ time steps with communication cost $O(D)$-competitive.*

Combining Lemmas 3 and 4, we have,

Theorem 8. DYN *is $O(D)$-competitive in both execution time and communication cost.*

Proof. DYN executes in two phases, Phase 1 and Phase 2, sequentially. Phase 1 finishes in $O(D)$ time steps. In Phase 2, each transaction in $\mathcal{T}$ spends $O(D)$ time steps to execute and commit. So, for all n transactions in $\mathcal{T}$, it takes $O(n \cdot D)$ time steps to execute and commit. In total, both Phase 1 and Phase 2 of DYN end in $O(D) + O(n \cdot D) = O(n \cdot D)$ time steps. Since, transactions need to follow the age order to commit, any optimal algorithm requires at least $O(n)$ time steps to execute and commit. Hence, DYN is $O(D)$-competitive in execution time. The same analysis works to show $O(D)$-competitive in communication cost. □

7 Evaluation

We have implemented OFFEXEC, OFFCOMM, PARTDYN, and DYN and evaluated them using a set of micro- and complex benchmarks. The experiments were performed on an

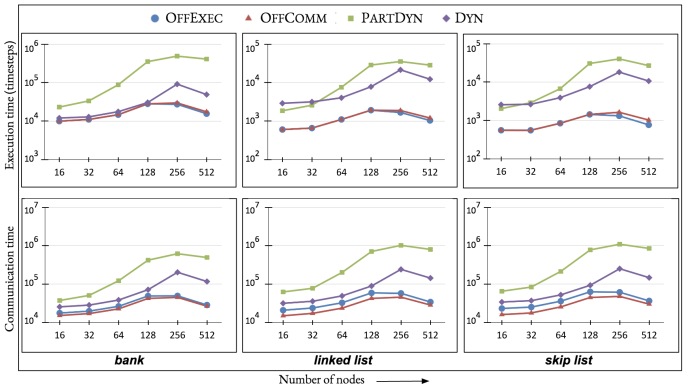

Fig. 3: Time and communication (log scale) in micro-benchmarks on random graphs.

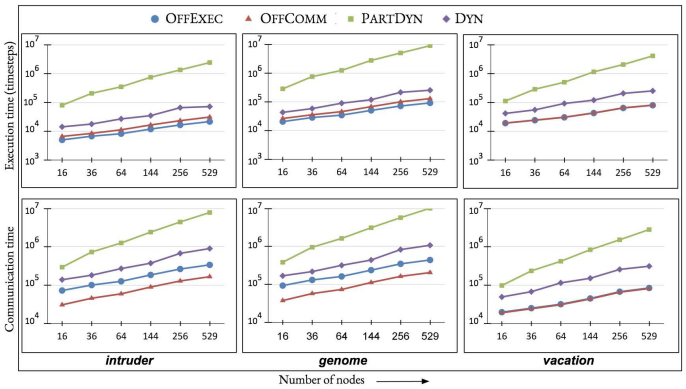

Fig. 4: Time and communication (log scale) in STAMP benchmarks on grid graphs.

Intel Core i7-7700K processor with 32 GB RAM, simulating two different communication graphs, namely *random* and *grid* whose diameters range from 3 to 6 and 6 to 44, respectively. The results presented are the average of 10 runs.

Results on micro-benchmarks: We experimented against three micro-benchmarks *bank, linked list*, and *skip list*. Fig. 3 provides the results in random graph.

Results on STAMP benchmarks: We experimented against *intruder*, *genome*, and *vacation* benchmarks from STAMP [25]. Fig. 4 provides the results in grid graph.

Results Discussion. For both random and grid graphs, OFFEXEC has the minimum execution time (which is optimal) in all the benchmarks. The execution time for OFFCOMM is higher than OFFEXEC but always within factor 2 of optimal. Similarly, in all the benchmarks, OFFCOMM has the minimum communication cost, which is with in factor of 2 from optimal. The experimental results in all the benchmarks show that the execution time of PARTDYN is always within $O(\log^2 n)$ factor compared to OFFEXEC. Moreover, the execution time in DYN is always within $O(D)$ factor. The communication cost results follow the same pattern. In fact, the results are substantially better than the theoretical bounds for both PARTDYN and DYN. In all the results, we can see that DYN has less execution time and less communication cost than PARTDYN. This is because of $D < \log^2 n$ in the experiment.

8 Concluding Remarks

In this paper, we have studied the ordered scheduling problem of committing transactions according to their predefined priorities in the control-flow distributed transactional memory, minimizing execution time and communication cost. The control-flow model is important because in many applications, the movement of data is costly due to its size and security purposes. We have provided a range of algorithms considering this problem in the offline and dynamic settings. As a future work, it will be interesting to deploy the algorithms in real distributed system(s) and measure the wall clock results.

Acknowledgements This research was supported by National Science Foundation under Grant No. CAREER CNS-2045597.

References

1. Attiya, H., Epstein, L., Shachnai, H., Tamir, T.: Transactional contention management as a non-clairvoyant scheduling problem. Algorithmica **57**(1), 44–61 (2010)
2. Attiya, H., Gramoli, V., Milani, A.: Directory protocols for distributed transactional memory. In: Transactional Memory. Foundations, Algorithms, Tools, and Applications, pp. 367–391 (2015)
3. Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: PPoPP. pp. 247–258 (2008)
4. Busch, C., Herlihy, M., Popovic, M., Sharma, G.: Impossibility results for distributed transactional memory. In: PODC. pp. 207–215 (2015)
5. Busch, C., Herlihy, M., Popovic, M., Sharma, G.: Fast scheduling in distributed transactional memory. In: SPAA. pp. 173–182. ACM (2017)
6. Busch, C., Herlihy, M., Popovic, M., Sharma, G.: Time-communication impossibility results for distributed transactional memory. Distributed Computing **31**(6), 471–487 (2018)
7. Busch, C., Herlihy, M., Popovic, M., Sharma, G.: Dynamic scheduling in distributed transactional memory. In: IPDPS. pp. 874–883 (2020)
8. Cain, H.W., Michael, M.M., Frey, B., May, C., Williams, D., Le, H.Q.: Robust architectural support for transactional memory in the power architecture. In: ISCA. pp. 225–236 (2013)
9. Demmer, M.J., Herlihy, M.: The arrow distributed directory protocol. In: DISC. pp. 119–133 (1998)
10. Fung, W.W.L., Singh, I., Brownsword, A., Aamodt, T.M.: Hardware transactional memory for gpu architectures. In: MICRO. pp. 296–307 (2011)
11. Gonzalez-Mesa, M.A., Gutiérrez, E., Zapata, E.L., Plata, O.G.: Effective transactional memory execution management for improved concurrency. TACO **11**(3), 24:1–24:27 (2014)
12. Gouveia, L.E.N., Magnanti, T.L.: Network flow models for designing diameter-constrained minimum-spanning and steiner trees. Networks **41**(3), 159–173 (2003)
13. Gramoli, V., Guerraoui, R., Trigonakis, V.: Tm^2c: a software transactional memory for many-cores. Distributed Computing **31**(5), 367–388 (2018)
14. Haring, R., Ohmacht, M., Fox, T., Gschwind, M., Satterfield, D., Sugavanam, K., Coteus, P., Heidelberger, P., Blumrich, M., Wisniewski, R., Gara, A., Chiu, G., Boyle, P., Chist, N., Kim, C.: The ibm blue gene/q compute chip. IEEE Micro **32**(2), 48–60 (2012)
15. Hendler, D., Naiman, A., Peluso, S., Quaglia, F., Romano, P., Suissa, A.: Exploiting locality in lease-based replicated transactional memory via task migration. In: DISC. pp. 121–133 (2013)

16. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA. pp. 289–300 (1993)
17. Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. Distributed Computing **20**(3), 195–208 (2007)
18. Hiromitsu, T., Akira, M.: An approximate solution for the steiner problem in graphs. MATH. JAP.; JPN; DA. 1980; VOL. 24; NO 6; PP. 573-577; BIBL. 9 REF. (1980)
19. Hirve, S., Palmieri, R., Ravindran, B.: Archie: A speculative replicated transactional system. In: Middleware. pp. 265–276 (2014)
20. Hwang, F.K.: On steiner minimal trees with rectilinear distance. SIAM Journal on Applied Mathematics **30**(1), 104–114 (1976), http://www.jstor.org/stable/2100587
21. Hwang, F., Richards, D., Winter, P.: The Steiner Tree Problem. ISSN, Elsevier Science (1992), https://books.google.com/books?id=-_yKbY3X_jUC
22. Intel: http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell (2012)
23. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
24. Manassiev, K., Mihailescu, M., Amza, C.: Exploiting distributed version concurrency in a transactional memory cluster. In: PPoPP. pp. 198–208 (2006)
25. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: stanford transactional applications for multi-processing. In: IISWC. pp. 35–46 (2008)
26. Mohamedin, M., Peluso, S., Kishi, M.J., Hassan, A., Palmieri, R.: Nemo: Numa-aware concurrency control for scalable transactional memory. In: ICPP. pp. 38:1–38:10 (2018)
27. Nakaike, T., Odaira, R., Gaudet, M., Michael, M.M., Tomari, H.: Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and POWER8. In: ISCA. pp. 144–157 (2015)
28. Palmieri, R., Peluso, S., Ravindran, B.: Transaction execution models in partially replicated transactional memory: The case for data-flow and control-flow. In: Transactional Memory, pp. 341–366 (2015)
29. Poudel, P., Rai, S., Sharma, G.: Processing distributed transactions in a predefined order. In: ICDCN. p. 215–224. ACM (2021)
30. Rai, S., Sharma, G., Busch, C., Herlihy, M.: Load balanced distributed directories. Information and Computation p. 104700 (2021)
31. Saad, M.M., Kishi, M.J., Jing, S., Hans, S., Palmieri, R.: Processing transactions in a predefined order. In: PPOPP. pp. 120–132 (2019)
32. Saad, M.M., Palmieri, R., Ravindran, B.: Lerna: Parallelizing dependent loops using speculation. In: SYSTOR. pp. 37–48 (2018)
33. Saad, M.M., Ravindran, B.: Snake: Control flow distributed software transactional memory. In: SSS. pp. 238–252 (2011)
34. Sharma, G., Busch, C.: A competitive analysis for balanced transactional memory workloads. Algorithmica **63**(1-2), 296–322 (2012)
35. Sharma, G., Busch, C.: Distributed transactional memory for general networks. Distrib. Comput. **27**(5), 329–362 (2014)
36. Sharma, G., Busch, C.: A load balanced directory for distributed shared memory objects. J. Parallel Distrib. Comput. **78**, 6–24 (2015)
37. Shavit, N., Touitou, D.: Software transactional memory. Distrib. Comput. **10**(2), 99–116 (1997)
38. Turcu, A., Ravindran, B., Palmieri, R.: Hyflow2: A high performance distributed transactional memory framework in scala. In: PPPJ. pp. 79–88 (2013)
39. Yoo, R.M., Lee, H.S.: Adaptive transaction scheduling for transactional memory systems. In: SPAA. pp. 169–178 (2008)
40. Zhang, B., Ravindran, B., Palmieri, R.: Distributed transactional contention management as the traveling salesman problem. In: SIROCCO. pp. 54–67 (2014)