Dynamic Scheduling in Distributed Transactional Memory

Costas Busch*, Maurice Herlihy†, Miroslav Popovic‡, Gokarna Sharma¶
*Louisiana State University, Baton Rouge, LA 70803, USA, *busch@csc.lsu.edu*
†Brown University, Providence, RI 02912, USA, *herlihy@cs.brown.edu*
‡University of Novi Sad, Novi Sad, Serbia, *miroslav.popovic@rt-rk.uns.ac.rs*
¶Kent State University, Kent, OH 44242, USA, *sharma@cs.kent.edu*

***Abstract*—We investigate scheduling algorithms for 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 sends the objects to other waiting transactions. We study scheduling algorithms with provable performance guarantees. Previously, only the *offline batch scheduling* setting was considered in the literature where transactions and the objects they access are known a priori. Minimizing execution time, even for the offline batch scheduling, is known to be NP-hard for arbitrary communication graphs. In this paper, we analyze for the very first time scheduling algorithms in the *online dynamic scheduling* setting where transactions and the objects they access are not known a priori and the transactions may arrive online over time. We provide efficient and near-optimal execution time schedules for dynamic scheduling in many specialized network architectures. The core of our technique is a method to convert offline schedules to online. We first describe a centralized scheduler which we then adapt it to a purely distributed scheduler. To our knowledge, these are the first attempts to obtain provably efficient online execution schedules for distributed transactional memory.**

***Index Terms*—Transactional memory, distributed systems, execution time, data-flow model, dynamic scheduling**

I. Introduction

Concurrent processes (threads) need to synchronize to avoid introducing inconsistencies in 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* [16, 29] (TM) has emerged as an alternative. Using TM, concurrent code is split into *transactions*, blocks of code that appear to execute atomically with respect to one another. Transactions are executed *speculatively*: synchronization conflicts or failures may cause an executing transaction to *abort* with its effects rolled back and then the transaction is restarted. In the absence of conflicts or failures, a transaction typically *commits*, causing its effects to become visible to all threads.

TM has fundamentally changed the way programming is done in multi-core computers both from theoretical and practical aspect and it is an active area of research in both academia and industry [1]. Several commercial processors provide direct hardware support for TM such as (Intel) Haswell [18], (IBM) Blue Gene/Q [15], (IBM) zEnterprise EC12 [25], and (IBM) Power8 [6]. There are proposals for adapting TM to clusters of GPUs [3, 12, 19]. TM is predicated to be widely used in distributed systems, going beyond GPUs and clusters.

Here, we consider TM in distributed networked systems which are widely available now-a-days and there is a growing interest in implementing TM in them [3, 17, 23, 28, 30]. In a distributed TM, there is an underlying network modeled as a weighted graph G. Each transaction resides at a node of G and requires one or more shared objects for read or write. Particularly, we consider a *data-flow* model of transaction execution [17, 28], in which each transaction executes at a node, but data objects are mobile and move to the nodes (transactions) that need them. A transaction initially requests the objects it needs, and executes only after it has assembled them. When the transaction commits, it releases its objects, possibly forwarding them to other waiting transactions.

Execution time is a fundamental metric for any computing system. In a multi-core TM, execute time is primarily dominated by the costs of handling data conflicts. In contrast, in a distributed TM, execution time is dominated by the costs of moving objects from one transaction to another. The goal of a *transaction scheduling algorithm* (sometimes called *contention management*) is to minimize delays caused by data conflicts and data movement.

We consider a synchronous model where time is divided into discrete steps [5]. At any time step, a node may perform three actions: (1) it may receive objects from adjacent nodes, (2) it executes any transaction that has assembled its required objects, and (3) it may forward objects to adjacent nodes. A transaction's execution step models when it commits, i.e., that transaction may have started earlier, but may have been blocked while assembling the objects it needed.

In this paper, we consider for the very first time to analyze the *online dynamic scheduling* setting where transactions and the objects accessed by them are not known a priori. In addition, transactions may arrive online and continuously over time. This departs significantly from the literature which studied and analyzed only the *offline batch scheduling* setting [4] where all transactions were known at beginning of time.

We provide online algorithms to compute conflict-free execution time schedules. Each node issues one transaction at a time requesting to access one or more shared objects. Each object is generated at some node and moves to the transactions that request it. The objective is to minimize the total execution time (makespan) until all transactions complete. The execution schedule determines the time steps when each transaction executes and commits. After a transaction commits, it forwards its objects to the respective next requesting transactions in the execution order that depend on these objects. Typically, an

object is sent along a shortest path, implying that the transfer time depends on the distance in G between the sender and receiver nodes. Hence, the execution time depends on both the objects' traversal times and inter-transaction dependencies.

It is known [5], through a reduction from vertex coloring, that for batch problems determining the shortest execution time in arbitrary communication graphs is NP-hard, and even hard to approximate within a sub-linear factor of n, the number of nodes in G. This hardness result also applies to online problems which are more general than batch problems. Therefore, we study online scheduling algorithms from a widely-studied notion of *competitiveness* – the ratio of total execution time produced by a designed algorithm to the shortest execution time achievable by any optimal offline algorithm. The goal is to make the competitive ratio as small as possible (the best possible is 1 which is hard to achieve).

Contributions. Since the online competitive ratio is hard to approximate in arbitrary networks, we focus on various specific distributed computing architectures: Clique, Hypercube, Butterfly, Grid, Line, Cluster, and Star. All these are popular topologies for a variety of applications in multiprocessors, networks-on-chip, rack-scale or cluster-scale distributed systems [11, 24, 26].

We consider scheduling problems where each node holds a single transaction, and each transaction requests up to k arbitrary objects. The following upper bounds on the competitive ratios assume that there is a centralized scheduler that decides the execution schedule. These schedules can also be computed by a decentralized scheduler which affects by a poly-log factor overhead the competitive ratios.

- *Clique:* in a clique (complete graph) of n nodes we give an online schedule which is $O(k)$ competitive.
- *Hypercube, Butterly, Grids:* in a hypercube with n nodes we give an online schedule which is $O(k \log n)$ competitive. Same bound holds also for the butterly and $\log n$-dimensional grids.
- *Line:* in a line graph of n nodes we give an online schedule which is $O(\log^3 n)$ competitive. Note that for the line graph the competitiveness does not depend on k.
- *Cluster:* we consider a cluster graph which consists of cliques with β nodes each connected to each other through bridge edges of weight $\gamma \geq \beta$. We show that there is a schedule which is $O(\min(k\beta, \log_c^k m) \cdot \log^3(n\gamma))$-competitive for some constant c.
- *Star:* in the star graph topology there is a central node that connects to rays each consisting of β nodes. We obtain a schedule which is $O(\log \beta \cdot \min(k\beta, \log_c^k m) \cdot \log^3 n)$ competitive, for some constant c.

Challenges and Techniques. We use two main techniques to produce the dynamic schedules which are suitable for different network topologies. The first is a direct approach that is suitable to small diameter graphs, while the second is an indirect approach that converts offline batch schedules to online dynamic schedules.

The first technique is constructing *online greedy schedules* which is based on continuously calculating a valid vertex coloring on the (dynamic) dependency graph of the transactions. Since the optimal chromatic number is hard to approximate in arbitrary graphs, this technique can be effective in specialized network graphs which have low diameter. Our competitive bounds on the clique, hypercube, butterfly, and $\log n$-dimensional grid are obtained from this technique.

The second technique is more general and it converts arbitrary offline batch schedules to online dynamic schedules. Thus, known batch scheduling results can be adapted to the online setting. The impact to the approximation of the offline schedule is a $O(\log^3(nD))$ factor delay, where D is the graph diameter. This is higher overhead than the direct method above but the technique can be used to graphs with larger diameter. With this approach the offline schedules by Busch *et al.* [4] are converted to online schedules for the line, cluster, and star topologies, which can be graphs with large diameter.

The conversion of offline to online is achieved by repeatedly dividing the transactions into *buckets*, where each bucket can be processed using a batch scheduler. When a transaction is generated it is assigned to one of the buckets according to its current dependencies and remains there until scheduled. The buckets have different levels according to the scheduling time for the batch problem of the transactions within. The bucket B_i, at level i, corresponds to a batch problem that takes up to 2^i time steps to execute its transactions. The bucket B_i is processed periodically every 2^i time steps. In this way, the periodically accumulated batch problems in the buckets are scheduled sequentially producing an online schedule.

The main benefit of the buckets is that transactions with small number of dependencies are able to make progress faster. Namely, a transaction with small number of dependencies with other transactions will be inserted in a lower level bucket which is processed more frequently than higher level buckets. In the analysis we show that if a batch scheduling algorithm has approximation ratio $b_{\mathcal{A}}$ then the competitive ratio of the online schedule is $O(b_{\mathcal{A}} \log^3(nD))$ competitive. The $O(\log^3(nD))$ factor is a penalty for the bucket separation that guarantees there is no interference between bucket levels.

When we present the basic bucket scheduling algorithm in Section IV we assume for simplicity of presentation that it is implemented by a central authority that has instant knowledge about all the current transactions and objects. In Section V we present a decentralized version of the bucket algorithm which is based on a sparse cover decomposition of the graph that gives a $O(b_{\mathcal{A}} \log^9(nD))$ competitive schedule.

Related Work. The most closely related work to ours is due to Busch *et al.* [4], where they provided efficient execution time schedules for offline batch scheduling on the data-flow model in specialized graphs likely to arise in practice: Clique, Line, Grid, Cluster, Hypercube, Butterfly, and Star. The algorithms presented were (near-)optimal. They also provided a non-trivial lower bound on execution time, improving significantly on the trivial TSP lower bound. However, the upper bound techniques of Busch *et al.* [4] do not apply to dynamic schedul-

ing. In the lower bound, there is a scheduling problem on the grid, with 2 objects per transaction, where every schedule must have execution time $\Omega(n^{1/40}/\log n)$ factor away from the optimal TSP tour length of any object. The same lower bound holds also for trees. These lower bounds also hold for the dynamic setting, since the batch problems are a special execution scenario case.

In another work, Busch *et al.* [5] considered the problem of minimizing both the execution time and *communication cost* (the total distance travelled by all the objects in G) simultaneously for transaction scheduling for distributed TMs under the data-flow model. They showed that it is impossible to simultaneously minimize execution time and communication cost, that is, minimizing execution time implies high communication cost (and vice-versa). They then provided efficient algorithms minimizing either execution time or communication cost individually in arbitrary communication graphs, where the result for execution time is sub-optimal due to the known inapproximability of the problem. All these results were established in the offline batch scheduling setting. They also sketch an approach for dynamic scheduling based on graph coloring, but that approach does not provide any competitive ratio bounds for the execution time as we do here.

There have also been previous works [17, 28, 30] on the data-flow model of distributed TMs which focused on minimizing only the communication cost for scheduling problem instances with only a single shared object. Kim and Ravindran [20] provided communication cost bounds for special workloads and problem instances with multiple shared objects. The execution time minimization is considered by Zhang *et al.* [30]. where they use TSP tours for object paths, which for arbitrary graphs can lead to significantly sub-optimal results according to the non-trivial lower bound in [4].

Several papers [3, 8, 23] presented techniques to implement distributed TMs. However, they either use global lock, serialization lease, or commit-time broadcasting technique which may not scale well with the size of the network. Moreover, several other papers studied distributed TMs employing replication and multi-versioning [21, 23].

Both offline batch and dynamic transaction scheduling problems were widely studied in multi-core systems without involving a communication network. Several scheduling algorithms with provable upper and lower bounds, and impossibility results were given [2, 10, 13, 27].

Finally, Deng *et al.* [9] study the problem of parallel execution of dynamically generated data-dependent graphs modeled as directed acyclic graphs (DAGs) with the same analytical method of competitive ratio used in this paper. However, our work is not directly comparable with [9] because of the fundamental differences in the respective system models considered. Our paper considers concurrent transactions (atomic blocks) based on a data-flow model where mobile copies of variables are distributed to nodes in the network and transactions are fixed to nodes, whereas [9] assumes a distributed memory model and execution of DAGs on parallel computers with no atomic blocks of operations on shared variables within the DAGs. Other several differences are as follows: (i) the dependency graphs in our paper are not restricted to DAGs, i.e., the dependency graphs may be arbitrary; (ii) the dependency graphs are not completely unknown in [9], whereas our paper makes no such assumption on the dependency graphs, i.e., they may be completely unknown and be of any kind (tree or graph); (iii) [9] makes assumptions on whether the number of processors are fixed or depend on the size of the input DAG, and the algorithms they present depend on these assumptions, whereas our paper makes no such assumption on the number of processors and our results apply to any system with any number of processors; (iv) [9] models communication latency through a fixed parameter τ representing average communication latency between any two processors, whereas our paper models communication latency through the shortest path distances between processors, exhibiting different system characteristics.

Roadmap. In Section II we give the model and preliminaries. We give an online greedy algorithm in Section III. We present the bucket algorithm in Section IV, and discuss its decentralized version in Section V. We conclude in Section VI. Due to space limitations some of the proofs have been omitted.

II. Model and Preliminaries

Consider a weighted graph $G = (V, E, w)$, with nodes V, edges E and an edge weight function $w : E \to \mathcal{Z}^+$. Let D be the diameter of the graph, which is the maximum length of any shortest path between any pair of nodes. We consider a synchronous communication model where all actions occur at discrete time steps. For an edge $e \in E$ it takes $w(e)$ time steps to send a message between its nodes.

Transactions are generated continuously over time. We assume that a transaction T is generated and resides in some node of G and requests a set of objects $O(T)$ for read or write. Transaction T executes once it acquires all the objects in $O(T)$. For simplicity, we assume that the transaction executes instantly at the time step that it gathered all the objects. Thus, all delays in out model are due to communication. A transaction is considered live until the time step it executes.

We assume that an object is created at some time step at some node of G by a transaction. At any time t, an object o_i either has been acquired by some transaction (the object resides at the node of the transaction), or the object o_i is in transit in the graph from one transaction to another. If a transaction T acquired an object and then T finished execution, then the object will remain at the node of T until some other transaction requests it and the scheduler moves the object to that transaction.

At any time t the *latest transaction* of an object o_i, denoted $L_t(o_i)$, is the transaction T that holds the object o_i at time t, or if there is no such T (i.e. the object is in transit), it is the last transaction that acquired (or generated) o_i before t. Let $L_t(T) = \bigcup_{o_i \in O(T)} L_t(o_i)$ denote the set of all last transactions of the objects of transaction T at time t.

Let $\mathcal{T}_t$ denote the set of live transactions at time t. Let $L(\mathcal{T}_t) = \bigcup_{T \in \mathcal{T}_t} L_t(T)$ be the set of latest transactions for the objects to be used by the transactions in $\mathcal{T}_t$.

In an online execution schedule S each transaction is executed at some designated time step. Consider a transaction T generated at time t. Suppose that the transaction executes at time $t_T > t$ in schedule S. The *execution duration* of T in schedule S is the time difference $t_T - t$. Let t^* denote the optimal time duration it takes to execute all the transactions in $\mathcal{T}_t$, given the execution times of the transactions in $L(\mathcal{T}_t)$. The competitive ratio for S at time t is $r_S(t) = \max_{T \in \mathcal{T}_t}((t_T - t)/t^*)$. The competitive ratio for S is $r_S = \sup_t r_S(t)$.

Definition 1 (Algorithm competitive ratio): For online scheduling algorithm $\mathcal{A}$, the competitive ratio $r_{\mathcal{A}}$ is the maximum competitive ratio over all possible execution schedules S that it produces, $r_{\mathcal{A}} = \sup_S r_S$. (We also say that A is $r_{\mathcal{A}}$-competitive.)

In all of our scheduling algorithms the execution times for the new transactions are not affecting the previously scheduled transactions. With this feature we can still manage to obtain good competitive competitive ratios. Such scheduling algorithms can be appealing in practice because future events are not affecting the currently scheduled transactions.

III. Online Greedy Schedule

We describe a generic scheduling algorithm which can be applied for arbitrary graphs. The algorithm is near optimal for special but interesting cases of small diameter graphs. The basic idea is to give a greedy coloring for the conflict dependency graph H for the transactions, where colors will be translated to execution times. The challenge is that the dependency graph changes over time and the greedy schedule is constrained by the transactions that have already been scheduled. We start with some basic results on graph coloring.

A. Weighted Graph Coloring

Consider a weighted graph $H = (V_H, E_H)$ with a weight function $w : E_H \rightarrow \mathcal{Z}^+$ on its edges. For each $v \in V_H$ let $N(v)$ denote the neighborhood of v which is the set of adjacent nodes of v in H. The *degree* of v is $\Delta(v) = |N(v)|$. The *weighted degree* of v denoted $\Gamma(v) = \sum_{u \in N(v)} w((u, v))$, is the sum of the weights of the edges adjacent to v in H.

A valid coloring of H is an assignment of integer values (colors) to the nodes of H, $c : V_H \rightarrow \mathcal{Z}^*$, such that for any two adjacent nodes their respective colors differ by at least the weights of their edges. Namely, for any $(u, v) \in E_H$,

$$|c(u) - c(v)| \geq w((u, v)). \tag{1}$$

A *partial coloring* of H is an assignment of colors to a subset of its vertices. A partial coloring is valid, as long as Equation 1 is satisfied for each pair of nodes that have received a color. We continue with a result that assigns a valid color to a node assuming that some other nodes may have already received a color.

Lemma 1: Given an arbitrary valid coloring of a set $V' \subseteq V_H$, any node $v \in V_H \setminus V'$ can be assigned a valid color $c(v) \leq 2\Gamma(v) - \Delta(v)$.

We can obtain an improved version of Lemma 1 for the case were all edge weights are equal.

Lemma 2: If all edges have the same weight β, and given an arbitrary valid coloring of a set $V' \subseteq V_H$, such that for each $u \in V'$, $c(u) = k_u\beta$, for some constant $k_u \geq 0$, then each node $v \in V_H \setminus V'$, can be assigned a valid color $c(v)$ such that $c(v) = k_v\beta$, for some constant $k_v \geq 1$, and $c(v) \leq \Gamma(v)$.

B. Coloring-based Schedule

Let $\mathcal{T}_t^g \subseteq \mathcal{T}_t$ denote the newly generated transactions at time t. In the greedy schedule, all the newly generated transactions $\mathcal{T}_t^g$ get immediately assigned (at time t) a designated execution time which remains unchanged thereafter. The challenge is to schedule the newly generated transactions $\mathcal{T}_t^g$ based on the existing schedules of the previously generated transactions. Moreover, some objects might be in transit at time t complicating the scheduling. Below we describe how to resolve these scheduling challenges by creating an appropriate dependency graph for the transactions at time t. The dependency graph will be then colored to provide the resulting schedules for the transactions in $\mathcal{T}_t^g$.

a) Dependency Graphs: At time t an object o_i is either in transit or it resides at the node of the latest transaction $L_t(o_i)$. If the object o_i is in transit then assume that at time t it resides at some node $v_t(o_i)$ along a shortest path connecting $L_t(o_i)$ to the next node that requested o_i. In case at time t the object o_i is in transit along an edge (u, v) (already left u going to v), then we will assume in the analysis that $v_t(o_i)$ is an artificial node with an edge connecting it to v with weight equal to the time remaining to reach v. Assume also for the sake of analysis that $v_t(o_i)$ has a temporary transaction T that uses o_i and executes at time t.

Let $Z_t(o_i)$ denote the current transaction that holds the object at time t, which is either $L_t(o_i)$ or the transaction in $v_t(o_i)$. Denote by $Z_t(T) = \bigcup_{o_i \in O(T)} Z_t(o_i)$ the set of current transactions that hold the objects of T at time t. Let $Z(\mathcal{T}_t) = \bigcup_{T \in \mathcal{T}_t} Z_t(T)$.

Consider the live transactions $\mathcal{T}_t$ at time t. Two transactions $T_1, T_2 \in \mathcal{T}_t$ *conflict* if $O(T_1) \cap O(T_2) \neq \emptyset$. The *conflict set* of a transaction $T \in \mathcal{T}_t$ at time t, denoted $C_t(T)$, is the set of live transactions in $\mathcal{T}_t$ that it conflicts with. The *extended conflict set* of a transaction T at time t, denoted $C'_t(T)$, includes $C_t(T)$ and all the current transactions of the objects in $O(T)$, namely, $C'_t(T) = C_t(T) \cup Z_t(T)$.

The *dependency graph* H_t of transactions at time t, represents the conflicts of the transactions at time t. Each node in H_t corresponds to a live transaction, that is $V(H_t) = \mathcal{T}_t$. Each edge in H_t corresponds to a conflict between two transactions, such that $(T_1, T_2) \in E(H_t)$ if $T_2 \in C_t(T_1)$ (and symmetrically, $T_1 \in C_t(T_2)$). The graph H_t is actually weighted, such that the weight of an edge represents the distance between the respective transactions in G.

Let $\mathcal{T}'_t = \mathcal{T}_t \cup Z(\mathcal{T}_t)$ denote the *extended set* of live transactions at time t. We can define the *extended dependency graph* H'_t with respect to the extended conflict sets of transactions. The set of nodes in H'_t are all the extended live transactions $\mathcal{T}'_t$

Algorithm 1: Online Greedy Schedule

foreach *time step* t **do**
- Let $\mathcal{T}_t^g$ denote the transactions generated at time t;
- Let H'_t be the extended conflict graph of the transactions in $\mathcal{T}'_t$ at time t;
- Transactions that have already been scheduled ($\mathcal{T}'_t \setminus \mathcal{T}_t^g$) are assumed to have a color in H'_t equal to their execution time minus t;
- Assign greedily a color $c(T)$ to each transaction $T \in \mathcal{T}_t^g$ by repeatedly applying Lemma 1 to uncolored transactions of H'_t;
- Each transaction $T \in \mathcal{T}_t^g$ is scheduled to execute at time $t + c(T)$;

at time t namely, $V(H'_t) = \mathcal{T}'_t$, and set of edges in H'_t is such that $(T_1, T_2) \in E(H'_t)$ if $T_2 \in C'_t(T_1)$ (and symmetrically, $T_1 \in C'_t(T_2)$).

b) Greedy Scheduling: A *valid coloring* of a set of transactions $S \subseteq V(H_t)$ (or $S \subseteq V(H'_t)$) assigns a unique positive integer (color) to each transaction in S such that any two adjacent transactions in S receive colors which differ by at least the weight of the incident edge that connects them in H_t (or H'_t). A valid coloring can translate to an execution schedule such that the colors assigned to the transactions correspond to the distinct time steps that transactions execute. Since at time t some objects may be in transit, a coloring of H_t is not adequate to provide a realistic schedule without knowing the current positions of the objects. For this reason, we consider a valid coloring based on the extended dependency graph H'_t which includes the current transactions for the objects that hold the objects at time t, including the temporary transactions for the objects that are in transit at time t.

Algorithm 1 has the details of the greedy scheduling algorithm. The main objective is to give a valid coloring to all the newly generated transactions in $\mathcal{T}_t^g$ assuming the existing schedules of the previously generated transactions and the current positions of the objects. Since each transaction in $\mathcal{T}'_t \setminus \mathcal{T}_t^g$ has already been scheduled, it is assumed to have a color equal to its scheduled execution time minus t (the remaining time until execution of the transaction). As a consequence, those transactions in $\mathcal{T}'_t \setminus \mathcal{T}_t^g$ that are executing at time t get color 0.

The algorithm applies repeatedly Lemma 1 on the extended dependency graph H'_t to assign a color to each transaction of $\mathcal{T}_t^g$. The color is then translated to an execution time by adding the current time t. The objects move to the next scheduled transactions that use them by following shortest paths in G.

We continue with an analysis of Algorithm 1. Let $\Delta'_t(T_i)$ and $\Gamma'_t(T_i)$ denote the respective regular and weighted degree of a transaction T_i in extended dependency graph H'_t.

Theorem 1 (Greedy online schedule): There is an execution schedule such that each transaction T_i generated at time t executes by time $t + 2\Gamma'_t(T_i) - \Delta'_t(T_i)$.

Proof. Given an arbitrary valid coloring of the transactions in $T'_t \setminus \mathcal{T}_t^g$ in H'_t, by repeatedly applying Lemma 1 to uncolored transactions of H'_t, we obtain a valid coloring of the newly generated transactions $\mathcal{T}_t^g$ in H'_t such that each transaction $T_i \in \mathcal{T}_t^g$ can receive a color at most $2\Gamma'_t(T_i) - \Delta'_t(T_i)$.

We can sort the transactions that request any specific object according to their respective $t+c$ where t is the generation time of the transaction and c is its respective color. An object will move from node to node in ascending time order. According to this schedule, each object in $O(T_i)$ will reach T_i no later than time $t + c$, since the valid coloring gives enough time for each object to be transferred to T_i by that time. Therefore, a transaction T_i will execute by time no later than $t + c \leq t + 2\Gamma'_t(T) - \Delta'_t(T)$. □

In Algorithm 1 if we use Lemma 2 instead of Lemma 1, we can obtain an improved result for the case where all edges in G have the same weight β. Here, we can execute the transactions at time steps which are multiples of β.

Theorem 2 (Schedule for uniform weights): If all the edges of the graph G have the same weight, then there is an execution schedule such that each transaction T_i generated at time t executes by time $t + \Gamma'_t(T_i)$.

Note that for each time step t, the sequential run time complexity of Algorithm 1 is polynomial to the size of the extended dependency graph H'_t at time t. Specifically, it is $O(n' + m' \log n')$ where n' and m' are the respective number of nodes (transactions) and edges of H'_t. This is because it applies Lemma 1 for each of the n' nodes. Each node with k neighbors in H'_t can be processed by first sorting the neighbors according to their current colors (requires $O(k \log k)$ steps), and then checks if the new color choices conflict with each neighbor node (requires $O(k)$ steps), which gives $O(k \log k)$ steps in total per node. Since $k \leq n'$ and $\sum_{v \in V(H'_t)} \Delta(v) = 2m'$, we get $O(n' + m' \log n')$ steps for processing all nodes. However, according to the execution model in Section II, all these sequential steps are subsumed within a single time step t of the concurrent execution, since the network communication delay is more detrimental to the overall execution time.

We continue to apply Algorithm 1 in several special case graphs, such as the complete graph, hypercube and butterfly.

C. Complete Graph

a) Scheduling Problem: Consider a unweighted complete graph (clique) G with n nodes where every node is connected to every other node with an edge of weight 1. Every node holds one transaction. Each transaction requests an arbitrary set of k objects. Once a transaction completes execution, the node of the transaction issues in the next step a new transaction requesting an arbitrary set of k objects. The process repeats.

b) Algorithm and Analysis: We use the greedy schedule of Algorithm 1 (with Lemma 2 instead of Lemma 1). Consider a transaction T that is generated at time t. From Theorem 2, T can execute by time $t + \Gamma'_t(T)$. Suppose that T uses objects $O(T) = o_{i_1}, \ldots, o_{i_k}$. Suppose also that each object o_{i_j} is used by l_{i_j} transactions in $\mathcal{T}'_t$. Then, since the edge weight is $\beta = 1$, the weighted degree in the extended dependency graph is $\Gamma'_t(T) \leq \sum_{j=1}^{k} l_{i_j} \leq k l_{\max}$, where $l_{\max} = \max_j l_{i_j}$. Thus, T will execute no later than $t + k l_{\max}$.

The transactions scheduled at time t are not affected by the transactions generated at later time. For the transactions

generated at time t, the execution time to execute all of them is at least $l_{\max}$, since at least so many transactions at time t request some object, and that object has to be transferred to all of them. Therefore, we obtain the following result:

Theorem 3 (Complete graph): In the complete graph the greedy online schedule has competitive ratio $O(k)$.

D. Hypercube and Related Graphs

In a hypercube graph [22] with n nodes any pair of nodes is connected with a path of length at most $\log n$ edges (logarithm is base 2). Thus, the hypercube graph can be represented as a complete graph with n nodes where the weight of any edge ranges between 1 and $\log n$. Assume for simplicity that the weights in the complete graph are all set to the worst case uniform value $\beta = \log n$. If we apply Algorithm 1 and Theorem 2, we get that each transaction generated at time T executes by time $t + \Gamma'_t(T)$. Since the edge weight is $\beta = \log n$, for k distinct objects the weighted degree is $\Gamma'_t(T) \leq \beta \sum_{j=1}^{k} l_{i_j} \leq \beta k l_{\max}$, where $l_{\max} = \max_j l_{i_j}$. Thus, T will execute no later than time $t + \beta k l_{\max}$. Since $l_{\max}$ is a lower bound on the execution time, we get that the resulting execution schedule has competitive ratio $O(\beta k) = O(k \log n)$.

The same result applies to other networks where the maximum distance between nodes is bounded by $\beta = O(\log n)$, as for example in butterfly networks [22] and $\log n$-dimensional grids [7].

Note that Theorem 2 is useful for getting upper bounds for the worst case scenario, but Algorithm 1 with Theorem 1 can give better execution schedule when used in practice.

E. Simple Centralized Online Scheduler

The online greedy schedules described above are obtained assuming a central authority with instant knowledge about the current positions of all the transactions and objects in the system. However, in reality such a centralized authority may not exist since transactions and objects are created in a distributed manner independent of each other.

Since all the above graphs have small diameter, $O(\log n)$, a simple remedy is to have a designated node in G to collect all the information as new transactions are generated and objects move. With this, each actual time step of the execution can be simulated with $O(\log n)$ time steps which is enough time to collect the information in the designated node and then decide on the actions of the next step of the execution. Thus, all the upper bounds on the execution schedule can be scaled with a $O(\log n)$ factor, proportional to the graph diameter.

Later we will give a more elegant decentralized solution in Section V with smaller dependence of the graph diameter. Thus, that solution is more suitable for larger diameter graphs. Since it involves a hierarchical decomposition of the graph, that approach's overhead to the overall schedule is more significant by a higher order poly-log factor. Therefore, it does not benefit significantly the low diameter graphs which we considered in this section.

IV. Online Bucket Schedule

We continue with an approach that converts offline scheduling algorithms to online. Consider an arbitrary offline batch scheduling algorithm $\mathcal{A}$ which can schedule any given set of transactions in graph G. We will convert $\mathcal{A}$ to an online algorithm using buckets of transactions. Before we describe the actual conversion to online, we need to perform two simple modifications to $\mathcal{A}$ on the way it operates to batch problems which leave unaffected its asymptotic performance.

A. Basic Modifications

The first basic modification of Algorithm $\mathcal{A}$ is to allow it to operate even if some of the batch transactions under consideration have already been scheduled. This can be easily achieved by computing an execution schedule for the currently unscheduled transactions with algorithm $\mathcal{A}$ which is appended at the end of the schedule of the already scheduled transactions. This does not alter the execution times of already scheduled transactions. In the worst-case the execution time of all the transactions doubles, leaving unaffected the asymptotic performance of $\mathcal{A}$.

For any set of transactions X, where some transactions in X may have already a determined schedule, let $F_{\mathcal{A}}(X)$ denote the time to execute all the transactions X using algorithm $\mathcal{A}$.

In the second basic transformation of $\mathcal{A}$ we will require that it has the following *suffix property* for batch execution schedules: for any set of transactions X the execution schedule S by $\mathcal{A}$ is such that every suffix S' with respective transactions X' execute in time $F_{\mathcal{A}}(X')$ assuming that the initial positions of the objects when S' starts are those from the last transactions executed in the prefix of S before S'.

If a batch schedule S does not satisfy the suffix property, then it can be easily modified to satisfy the suffix property by repeatedly applying algorithm $\mathcal{A}$ to any suffix that violates the property, starting from the longest suffix until there is no smaller suffix that violates the property anymore.

The purpose of the first modification to $\mathcal{A}$ is that in the online schedules some of the transactions have already been scheduled while we attempt to schedule a new set of transactions. The purpose of the second modification to $\mathcal{A}$ is that suffixes of schedules of previously executed transactions are affecting the performance of the schedules for the new transactions. With these two basic modifications we will be able to bound the performance of the online algorithm with respect to the performance of the offline algorithm $\mathcal{A}$.

B. The Bucket Algorithm

We are now ready to give the transformation of the offline algorithm $\mathcal{A}$ to an online algorithm. The details appear in Algorithm 2. We assume that the basic modifications described earlier have already applied to $\mathcal{A}$.

We will use buckets to store the newly generated transactions at time t, $\mathcal{T}_t^g \subseteq \mathcal{T}_t$, until their execution schedules are determined. Let $\mathcal{T}_t^s \subseteq \mathcal{T}_t$ denote the set of transactions whose schedule has already been determined at time t (the

Algorithm 2: Online Bucket Schedule

Consider a batch scheduling algorithm $\mathcal{A}$;
Assume disjoint buckets B_i at levels $i \geq 0$, such that bucket B_i activates every 2^i time steps;
foreach *time step* t **do**
  Each $T \in \mathcal{T}_t^g$ will be inserted into the bucket B_i with smallest index i such that $F_{\mathcal{A}}(\mathcal{T}_t^s \cup B_i \cup \{T\}) \leq 2^i$;
  if B_i *activates at time* t **then**
    All transactions in B_i get scheduled using algorithm $\mathcal{A}$ with transactions $\mathcal{T}_t^s \cup B_i$;
    The schedule does not modify the execution times of the already scheduled transactions in $\mathcal{T}_t^s$;
    B_i becomes empty and all its transactions are inserted in $\mathcal{T}_t^s$;

transactions in $\mathcal{T}_t^s$ may execute at t or later according to their scheduled execution time).

A *bucket* B_i at *level* i, where $i \geq 0$, is a set of unscheduled transactions which are expected to execute in at most 2^i time steps. At time t each transaction $T \in \mathcal{T}_t^g$ will be inserted into the bucket B_i with smallest index i such that $F_{\mathcal{A}}(\mathcal{T}_t^s \cup B_i \cup \{T\}) \leq 2^i$. That is, T is inserted into the smallest index bucket B_i that does not increase the offline execution time of that bucket beyond its limit, given the fixed execution times of the already scheduled transactions in $\mathcal{T}_t^s$.

Buckets get activated periodically so that the transactions within get actual execution times. Bucket B_i gets activated every 2^i time steps. Once B_i activates, say at time t', all transactions in B_i get scheduled using algorithm $\mathcal{A}$ on the set $B_i \cup \mathcal{T}_{t'}^s$. The generated schedule does not alter the execution times of the already scheduled transactions in $\mathcal{T}_{t'}^s$. Then, we remove the transactions from B_i (which becomes an empty bucket), since the transactions in B_i are considered scheduled and become part of $\mathcal{T}_{t'}^s$.

If multiple buckets get activated simultaneously at time t', then we first schedule the transactions of the lower level buckets. Thus, when higher level buckets are scheduled at t' they assume that the lower level bucket transactions are already scheduled as members of $\mathcal{T}_{t'}^s$.

We give a few remarks about this algorithm. The maximum bucket level is bounded and determined by the diameter and number of nodes in G (see Lemma 3). The activation times of different levels are not required to be aligned.

C. Analysis of Bucket Algorithm

We continue with an analysis of Algorithm 2. We first prove two lemmas that lead to the main result in Theorem 4.

Lemma 3: The level of a bucket is at most $\log(nD) + 1$.

Proof. The worst execution schedule is when the transactions execute sequentially. At any time step, there can be no more than n scheduled transactions (one from each node). The maximum distance in G between any pair of transactions is no more than the graph diameter D. Thus, the worst in time schedule will execute the transactions in sequence requiring at most time nD. Hence, the maximum bucket level will not exceed $\lceil \log(nD) \rceil \leq \log(nD) + 1$ (log is base 2). □

Lemma 4: Any transaction $T \in \mathcal{T}_t^g$ (generated at time t) which is inserted into bucket B_i will be executed by time $t + (i+1)2^{i+2}$.

Proof. We prove this by induction on k, the level of bucket B_k. For the basis case the level is $k = 0$. The bucket B_0 gets activated at every time step. Thus, the bucket B_0 gets activated at time t (i.e. instantly). Since for $T \in B_0$ it was determined that $F_{\mathcal{A}}(\mathcal{T}_t^s \cup B_0 \cup \{T\}) \leq 2^i = 1$, the transaction T will execute by time $t + 1 = t + 2^i \leq t + (i+1)2^{i+2}$ ($i = 0$).

Suppose that the claim holds for all $k <= i$. We consider now the case $k = i + 1$. The bucket B_{i+1} gets activated no later than time $t' = t + 2^{i+1}$. Let $B_{i+1,t}$ denote the contents of B_{i+1} at time t (after new transactions were inserted in B_{i+1} at time t). Consider the latest transaction T'' that was inserted into B_{i+1} at some time t'', $t \leq t'' \leq t'$. It must be that $F_{\mathcal{A}}(\mathcal{T}_{t''}^s \cup B_{i+1,t''}) \leq 2^{i+1}$.

Suppose that no additional transactions had their execution schedule determined between t'' and t' (that is, $\mathcal{T}_{\hat{t}}^s \subseteq \mathcal{T}_{t'}^s$, for $t'' \leq \hat{t} \leq t'$). Since $\mathcal{T}_{t'}^s \cup B_{i+1,t'} \subseteq \mathcal{T}_{t''}^s \cup B_{i+1,t''}$, $F_{\mathcal{A}}(\mathcal{T}_{t'}^s \cup B_{i+1,t'}) \leq F_{\mathcal{A}}(\mathcal{T}_{t''}^s \cup B_{i+1,t''}) \leq 2^{i+1}$. Thus, in this case all the transactions in B_{i+1} can be executed no later than

$$t' + 2^{i+1}. \tag{2}$$

However, between t'' and t', some other buckets may get activated causing a set of new transactions, say transactions A, having their schedules determined. According to the algorithm, these additional transactions must have been from buckets at level i or lower. In the worst case, from induction hypothesis, those buckets are activated the latest at time t' (before B_{i+1}), with the latest execution time of any transaction from those buckets $t' + (i+1)2^{i+2}$. To accommodate the transactions in A the schedule of B_{i+1} may be shifted by time,

$$2(t' - t'' + (i+1)2^{i+2}), \tag{3}$$

which is enough time to allow the shared objects used by the transactions in A to move to the positions where they execute (according to their scheduled execution times) and then return back to the original positions they were at time t''.

Therefore, combining Equations 2 and 3 the transactions in B_{i+1} will execute no later than

$$\begin{aligned} & t' + 2^{i+1} + 2(t' - t'' + (i+1)2^{i+2}) \\ \leq\ & t + 2^{i+1} + 2^{i+1} + 2(2^{i+1} + (i+1)2^{i+2}) \\ \leq\ & t + (i+2)2^{i+3}. \end{aligned}$$

□

Theorem 4 (Bucket schedule competitiveness): The online schedule has competitive ratio $O(b_{\mathcal{A}} \log^3(nD))$, where $b_{\mathcal{A}}$ is the approximation ratio of offline algorithm $\mathcal{A}$.

Proof. Consider some arbitrary time t where the live transactions are $\mathcal{T}_t = \mathcal{T}_t^s \cup \mathcal{T}_t^g$. If the transactions in $\mathcal{T}_t^g$ were scheduled alone by algorithm $\mathcal{A}$ then their execution schedule time would be within $b_{\mathcal{A}}$ from optimal. However, the newly generated transactions $\mathcal{T}_t^g$ are going to be scheduled based on the restrictions imposed by the already scheduled transactions

$\mathcal{T}_t^s$. In the worst case scenario, the newly generated transactions $\mathcal{T}_t^g$ will execute after the transactions in $\mathcal{T}_t^s$. Therefore, we need to estimate how long it will take to execute the last transactions in $\mathcal{T}_t^s$ in order to determine when the transactions in $\mathcal{T}_t^g$ will execute, and hence determine the duration of the whole schedule.

The transactions in $\mathcal{T}_t^s$ have been generated from various levels at previous buckets. Consider a specific level i that has been used to add transactions in $\mathcal{T}_t^s$. From Lemma 4, any such transaction was generated no earlier than $t-(i+1)2^{i+2}$. During that period the number of level i buckets that have been activated is at most $(i+1)2^{i+2}/2^i = 4(i+1)$, since a level i bucket activates every 2^i time steps.

According to Lemma 4, the execution time schedule of bucket B_i is within a factor $O(ib_{\mathcal{A}})$ from the optimal time of the transactions within B_i, since the 2^i is a $O(b_{\mathcal{A}})$ approximation of the optimal schedule, by the definition of the bucket level. Since from Lemma 3 the maximum level is at most $\log(nD)+1$, we have that the execution time of any bucket is within approximation factor $\zeta = O(b_{\mathcal{A}}\log(nD))$ from optimal. However, this approximation factor is assuming the current scheduled buckets.

Let ξ be the total number of buckets that contribute to $\mathcal{T}_t^s$. From Lemma 3 and since each level contributes at most $4(i+1)$ activated buckets in $\mathcal{T}_t^s$ we get:

$$\xi \leq \sum_{i=0}^{\log(nD)+1} 4(i+1) = O(\log^2(nD)).$$

The ξ buckets can be ordered according to when they get activated. Let's examine the execution time of the transactions in the ξ buckets with respect to starting time t.

For the first of the ξ buckets, the respective transactions in $\mathcal{T}_t^s$ must execute within factor of ζ from optimal (after t). The reason is that their predetermined schedule corresponds to a suffix (starting from t) of the original execution schedule when the bucket was activated. Furthermore, algorithm $\mathcal{A}$ has the suffix property such that any suffix of its schedule has execution time within $b_{\mathcal{A}}$ from optimal. Moreover, the schedule of $\mathcal{A}$ is only shifted in time to give ζ total approximation time due to other dependencies, as we analyzed in Lemma 4.

Similarly, for the second of the ξ buckets, the respective transactions in $\mathcal{T}_t^s$ must execute within time proportional to a factor of $b_{\mathcal{A}}$ from optimal starting from t if they were running alone. Thus, the combined time of the first two of the ξ buckets execute in time 2ζ from optimal. Generalizing, the execution time of the ξ buckets is within $\xi\zeta$ from optimal.

With a similar argument, each of the corresponding buckets of T_t^g executes within time which is a factor $\xi\zeta+\zeta$ from optimal, where $\xi\zeta$ is the approximation factor contributed from the previous buckets, while the additional ζ term is the approximation from the newly generated transactions. Thus, the total execution time of the transactions in $\mathcal{T}_t$ is within a factor $(\xi+1)\zeta = O(b_{\mathcal{A}}\log^3(nD))$ from optimal. □

D. Applications to Specialized Graphs

We will apply Algorithm 2 to several cases of network topologies to convert batch scheduling algorithms to online scheduling algorithms. Busch *et al.* [4] give offline scheduling algorithms for a variety of specialized graphs including the line, cluster, and star. For these batch scheduling problems, there are w objects, each node generates at most one transaction, and each transaction requests an arbitrary set of k objects out of w. When we convert these offline algorithms to online we obtain the following results.

- *Line:* A line graph is a set of n ordered nodes so that each node has an edge of weight 1 connecting it to the next node in order. It is shown [4] that there is an offline schedule in the line graph which is within a factor $b_{\mathcal{A}} = O(1)$ from optimal (asymptotically optimal). From Theorem 4, since $D = O(n)$, we obtain an online scheduling algorithm which is $O(b_{\mathcal{A}}\log^3(nD)) = O(\log^3 n)$ competitive.
- *Cluster:* A cluster graph consists of α cliques with β nodes each; where edges in the clique have weight 1. Each clique has a designated bridge node. The bridge nodes from different clusters connect to each other with edges of weight $\gamma \geq \beta$. It is shown [4] that there is an offline algorithm which gives a schedule with factor $b_{\mathcal{A}} = O(\min(k\beta, \log_c^k m))$ approximation from the optimal, for some constant c, where $m = \max(n, w)$. From Theorem 4, since $D = O(n+\gamma)$, we obtain an online scheduling algorithm which is $O(b_{\mathcal{A}}\log^3(nD)) = O(\min(k\beta, \log_c^k m)\cdot\log^3(n\gamma))$ competitive.
- *Star:* In the star graph there is a central node that connects to α rays each consisting of β nodes; all edges have weight 1. It is shown [4] that there is an offline algorithm which gives a schedule with factor $b_{\mathcal{A}} = O(\log\beta\cdot\min(k\beta, \log_c^k m))$ approximation from optimal, for a constant c. From Theorem 4, since $D = O(n)$, we obtain an online scheduling algorithm which is $O(b_{\mathcal{A}}\log^3(nD)) = O(\log\beta\cdot\min(k\beta, \log_c^k m)\cdot\log^3 n)$ competitive.

Note that the cluster and star batch scheduling algorithms used above are in fact randomized. In the case that the bad event occurs for a bucket of not getting a batch schedule with the specified bound (with small probability), then we repeat the offline algorithm for that bucket until we successfully obtain a batch schedule. Thus, the online schedules remain feasible.

The sequential run time complexity of Algorithm 2 in calculating an execution schedule at time step t, depends on the sequential complexity of algorithm $\mathcal{A}$ for each bucket level with a small additional overhead since there are only $O(\log(nD))$ levels of buckets. The offline batch scheduling algorithms we used above have polynomial time complexity in computing the schedules [4]. Thus, Algorithm 2, has polynomial time complexity in calculating the respective online schedules for the network cases we discussed above. However, according to our execution model these sequential calculations are subsumed within a single time step t of the concurrent exe-

cution, since the communication delay is the main contributing factor in the concurrent execution time.

V. Distributed Bucket Approach

The online algorithms we presented earlier use a central authority with knowledge about all current transactions and objects. Here, we discuss a distributed approach which allows the online schedule to be computed in a decentralized manner.

The distributed approach is based on the online bucket scheduling algorithm described in Section IV but adapted appropriately to work in the distributed setting. In the adapted version of the bucket algorithm, the buckets of different levels are split among various nodes of the graph G in what we call *partial buckets*. To facilitate the distributed scheduling algorithm, we use a hierarchy of clusters where designated leader nodes at the clusters will hold the partial buckets.

Cluster Decomposition. Divide the graph G into a hierarchy of clusters with $H_1 = \lceil \log D \rceil + 1$ layers, where D is the diameter of G (logarithms are base 2). A cluster is a subset of the nodes, and its diameter is the maximum distance between any two nodes (we use the *weak diameter* of the cluster where distances between nodes in the cluster are measured with respect to G and not within the subgraph induced by the cluster). 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 (the k-neighborhood is the set of nodes which are distance at most k from u; the 0-neighborhood is u itself).

There are cluster constructions [14, 28] which give a hierarchy with H_1 layers where $f(\ell) = O(\ell \log n)$, and $g(\ell) = O(\log n)$, known as a *hierarchical sparse cover* of G. (These constructions have the *strong diameter* property, where distances are measured within the induced subgraph in the clusters, but these still serve our purpose.)

There is actually a hierarchical sparse cover construction [28] where each layer ℓ is decomposed into at most $H_2 = O(\log n)$ sub-layers of clusters, where each sub-layer is a partition of G. Thus, a node participates in all the sub-layers of a layer but possibly in a different cluster within each sub-layer. We use such a sparse cover in our algorithm.

The *height* h of a cluster which is at layer h_1 and sub-layer h_2 is the pair $h = (h_1, h_2)$. Heights are ordered lexicographically. One node in each cluster is designated as the *leader* of the cluster.

At each layer ℓ a node u participates in H_2 clusters, one in each sub-layer. At least one of those clusters at layer ℓ contains the $(2^\ell - 1)$-neighborhood of u, and we pick one of them to be the *home cluster* of u at layer ℓ. Thus, a node has H_1 home clusters in total, one at each layer.

To facilitate the actions of the distributed transaction scheduling, each shared memory object o_i carries with it some information to assist the transaction scheduling. The object o_i carries the information of all the transaction locations (node addresses) that will use it, and also which of these transactions have already been scheduled or not. In addition, the object o_i has information about the bucket locations (node addresses) that have transactions that use it.

Algorithm 3: Distributed Bucket Schedule

1 **foreach** *new transaction* T **do**
2 Transaction T discovers the current positions of its objects in G; say the furthest is x away from T;
3 Each of the objects of T informs the transaction T about other conflicting transactions (scheduled and unscheduled);
4 Let y be the maximum of x and the distance to the furthest conflicting transaction from T;
5 T picks the lowest layer in the cluster hierarchy that has a cluster whose home node includes the y-neighborhood of T;
6 T reports to the leader v of the chosen home cluster; then, v places T into a partial i-bucket where i is determined by the transactions reported to v;
7 When the bucket of T gets activated, all the objects of T are informed about the execution schedule;

Distributed Bucket Algorithm. Algorithm 3 has the transaction actions of the distributed bucket approach. The bucket B_i is split into possibly multiple *partial i-buckets*. The union of the transactions in the partial i-buckets make the whole buckets B_i. The partial i-buckets may appear into multiple layers (or sub-layers) of the hierarchy. In fact, there could be multiple i-buckets (for the same i) in each sub-layer. The partial i-buckets are hosted at leader nodes of the clusters. All of the partial-i buckets get activated at the same time step every 2^i time steps, that corresponds to the activation time of B_i.

Consider a transaction T generated at time t at a node v such that T uses a set of objects. Transaction T (i.e. the process that executes T in v) looks to find all conflicting transactions with T that have already been scheduled. It also looks for buckets that have transactions conflicting with T. It collects this information from the objects, as described in Algorithm 3, which help to determine into which bucket T will be inserted.

The objects could be moving from node to node while the transactions execute. We can track objects in transit by reaching the node that the object departs from. While a transaction in v tries to discover the current locations of the objects that it uses, the objects may drift further away from v making it difficult to be discovered. For this reason, we require that an object moves at a slower pace than the discovery request messages travel. We can achieve this by artificially having an object travel over a unit weight link at a rate of two time steps, instead of one, that is, halving the speed that the object moves. In this way, if at time t an object is at distance d from v, then it will be discovered by time $2d$ at most.

Analysis of Distributed Algorithm. We continue with the analysis of Algorithm 3. Conflicting transactions with T could be reporting to different clusters at various levels due to their own dependencies, which could be caused from a different set of objects than those used by T.

Lemma 5: For any two live conflicting transactions T and T' it cannot be that neither detect each other before they report to their home clusters.

Lemma 6: If T reports at a home cluster C then there is no other conflicting transaction to T that reports in a cluster at the same sub-layer as that of C.

Proof. Suppose that there is a conflicting transaction T' that reports to a cluster at the same sub-layer to T. From Lemma 5, T is aware of T' or vice-versa (which is treated similarly).

If T is aware of T', then the cluster C must contain T' (by the way T picks the cluster it reports to, which contains all the transactions that T knows it conflicts with); hence, T' must have also reported to C, due to the sub-layer being a partition of G. Symmetrically, if T' is aware of T, then the cluster that T' has reported to must be the same with the cluster of T. □

From Lemma 6 we obtain the following result.

Corollary 1: In a sub-layer, any two partial i-buckets do not have transactions that conflict with each other.

Lemma 7: The maximum height in the cluster hierarchy that a partial i-bucket can appear to is $(i+1, H_2 - 1)$.

Define $M = \max(H_1, H_2, L)+3$, where L is the maximum number of distinct bucket levels, which according to Lemma 3 is $\log(nD) + 1$. Note that $M = O(\log(nD))$. Next are adaptations of Lemma 4 and Theorem 4 in the distributed setting.

Lemma 8: In the distributed setting, any transaction $T \in \mathcal{T}_t^g$ (generated at time t) which is inserted into a partial i-bucket $\overline{B}$ that resides at height (j, k) will be executed by time $t + (iM^2 + jM + k + 3)2^{i+3}$.

Theorem 5 (Distributed bucket competitiveness): In the distributed setting, the online schedule has competitive ratio $O(b_{\mathcal{A}} \log^9(nD))$, where $b_{\mathcal{A}}$ is the approximation ratio of offline algorithm $\mathcal{A}$.

Algorithm 3 also has polynomial time complexity as it adapts Algorithm 2 with polynomial time complexity.

VI. Concluding Remarks

We have presented efficient execution time schedules in the online dynamic scheduling setting on the data-flow model of distributed transactional memory. Our results are the first known attempts to obtain provably efficient online execution schedules for distributed transactional memory.

There are some open questions. It would be interesting to examine the impact of congestion, and the case where network links may also have bounded capacity. Furthermore, it would also be interesting to evaluate our algorithm against different application benchmarks in a practical setting.

Acknowledgments: M. Popovic is supported by the Ministry of Education, Science and Technology Development of Republic of Serbia Grant III-44009-2. G. Sharma is supported by the National Science Foundation grant CCF-1936450.

References

[1] M. K. Aguilera, D. Malkhi, K. Marzullo, A. Panconesi, A. Pelc, and R. Wattenhofer. Announcing the 2012 edsger w. dijkstra prize in distributed computing. *SIGARCH Computer Architecture News*, 40(4):1–2, 2012.

[2] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. *Algorithmica*, 57(1):44–61, 2010.

[3] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP*, pages 247–258, 2008.

[4] C. Busch, M. Herlihy, M. Popovic, and G. Sharma. Fast scheduling in distributed transactional memory. In *SPAA*, pages 173–182, 2017.

[5] C. Busch, M. Herlihy, M. Popovic, and G. Sharma. Time-communication impossibility results for distributed transactional memory. *Distributed Computing*, 31(6):471–487, Nov 2018.

[6] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Q. Le. Robust architectural support for transactional memory in the power architecture. In *ISCA*, pages 225–236, 2013.

[7] M. Y. Chan. Embedding of d-dimensional grids into optimal hypercubes. In *SPAA*, pages 52–57, 1989.

[8] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2stm: Dependable distributed software transactional memory. In *PRDC*, pages 307–313, 2009.

[9] X. Deng, E. Koutsoupias, and P. D. MacKenzie. Competitive implementation of parallel programs. *Algorithmica*, 23(1):14–30, 1999.

[10] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *PODC*, pages 7–16, 2009.

[11] M. Dubois, M. Annavaram, and P. Stenstrm. *Parallel Computer Organization and Design*. Cambridge University Press, New York, NY, USA, 2012.

[12] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt. Hardware transactional memory for gpu architectures. In *MICRO*, pages 296–307, 2011.

[13] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC*, pages 258–264, 2005.

[14] A. Gupta, M. T. Hajiaghayi, and H. Räcke. Oblivious network design. In *SODA*, pages 970–979, 2006.

[15] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G. Chiu, P. Boyle, N. Chist, and C. Kim. The ibm blue gene/q compute chip. *IEEE Micro*, 32(2):48–60, 2012.

[16] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[17] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.

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

[19] S. Irving, S. Chen, L. Peng, C. Busch, M. Herlihy, and C. J. Michael. CUDA-DTM: distributed transactional memory for GPU clusters. In *NETYS*, pages 183–199, 2019.

[20] J. Kim and B. Ravindran. On transactional scheduling in distributed transactional memory systems. In *SSS*, pages 347–361, 2010.

[21] J. Kim and B. Ravindran. Scheduling transactions in replicated distributed software transactional memory. In *CCGrid*, pages 227–234, 2013.

[22] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[23] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP*, pages 198–208, 2006.

[24] M. Michalewicz, L. Orlowski, and Y. Deng. Creating interconnect topologies by algorithmic edge removal: Mod and smod graphs. *Supercomput. Front. Innov.: Int. J.*, 2(4):16–47, Mar. 2015.

[25] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and POWER8. In *ISCA*, pages 144–157, 2015.

[26] S. Pasricha and N. Dutt. *On-Chip Communication Architectures: System on Chip Interconnect*. Morgan Kaufmann Publishers Inc., 2008.

[27] G. Sharma and C. Busch. Window-based greedy contention management for transactional memory: Theory and practice. *Distrib. Comput.*, 25(3):225–248, 2012.

[28] G. Sharma and C. Busch. Distributed transactional memory for general networks. *Distrib. Comput.*, 27(5):329–362, 2014.

[29] N. Shavit and D. Touitou. Software transactional memory. *Distrib. Comput.*, 10(2):99–116, 1997.

[30] B. Zhang, B. Ravindran, and R. Palmieri. Distributed transactional contention management as the traveling salesman problem. In *SIROCCO*, pages 54–67, 2014.