# On Transactional Scheduling in Distributed Transactional Memory Systems

Junwhan Kim and Binoy Ravindran

ECE Department, Virginia Tech, Blacksburg, VA, 24061
Email: {junwhan, binoy}@vt.edu

**Abstract.** We present a distributed transactional memory (TM) scheduler called *Bi-interval* that optimizes the execution order of transactional operations to minimize conflicts. *Bi-interval* categorizes concurrent requests for a shared object into read and write intervals to maximize the parallelism of reading transactions. This allows an object to be simultaneously sent to nodes of reading transactions (in a data flow TM model), improving transactional makespan. We show that *Bi-interval* improves the makespan competitive ratio of the Relay distributed TM cache coherence protocol to $O(\log(n))$ for the worst-case and $\Theta \log(n-k)$ for the average-case, for $n$ nodes and $k$ reading transactions. Our implementation studies confirm *Bi-interval*'s throughput improvement by as much as 200% $\sim$ 30%, over cache-coherence protocol-only distributed TM.

**Keywords:** Transactional Memory, Transactional Scheduling, Distributed Systems, Distributed Cache-Coherence

## 1 Introduction

Transactional memory (TM) is an alternative synchronization model for shared in-memory data objects that promises to alleviate difficulties with lock-based synchronization (e.g., lack of compositionality, deadlocks, lock convoying). A transaction is a sequence of operations, performed by a single thread, for reading and writing shared objects. Two transactions *conflict* if they access the same object and one access is a write. When that happens, a contention manager (CM) is typically used to resolve the conflict. The CM resolves conflicts by deciding which transactions to abort and aborting them, allowing only one transaction to proceed, and thereby ensures atomicity. Aborted transactions are retried, often immediately. Thus, in the contention management model, a transaction ends by either committing (i.e., its operations take effect), or by aborting (i.e., its operations have no effect). Efficient contention management ensures transactional *progress*—i.e., at any given time, there exists at least one transaction that proceeds to commit without interruption [20]. TM for multiprocessors has been proposed in hardware [11], in software [12], and in hardware/software combination [17].

A complimentary approach for dealing with transactional conflicts is *transactional scheduling*. Broadly, a transactional scheduler determines the ordering of transactions so that conflicts are either avoided altogether or minimized. This includes serializing transaction executions to avoid conflicts based on transactions' predicted read/write access sets [8] or collision probability [7]. In addition, conflicts can be minimized by

carefully deciding when a transaction that is aborted due to a conflict is resumed [1, 7], or when a transaction that is stalled due to potential for an immediate conflict is later dispatched [22]. Note that, while contention management is oblivious to transactional operations, scheduling is operation-aware, and uses that information to avoid/minimize conflicts. Scheduling is not intended as a replacement for contention management; a CM is (often) needed and scheduling seeks to enhance TM performance.

Distributed TM promises to alleviate difficulties with lock-based distributed synchronization [13, 4, 18, 16, 24]. Several distributed TM models are possible. In the data-flow model [13], which we also consider, object performance bottlenecks can be reduced by migrating objects to the invoking transactional node and exploiting locality. Moreover, if an object is shared by a group of geographically-close clients that are far from the object's home, moving the object to the clients can reduce communication costs. Such a data flow model requires a *distributed cache-coherence protocol*, which locates an object's latest cached copy, and moves a copy to the requesting transaction, while guaranteeing one writable copy. Of course, CM is also needed. When an object is attempted to be migrated, it may be in use. Thus, a CM must mediate object access conflicts. Past distributed TM efforts present cache coherence protocols (e.g., Ballistic [13], LAC [24], Relay [23]) and often use a globally consistent CM (e.g., Greedy [9]).

We consider distributed transactional scheduling to enhance distributed TM performance. We present a novel distributed transactional scheduler called *Bi-interval* that optimizes the execution order of transactional operations to minimize conflicts. We focus on read-only and read-dominated workloads (i.e., those with only early-write operations), which are common transactional workloads [10]. *Bi-interval* categorizes concurrent requests for a shared object into read and write intervals to maximize the parallelism of reading transactions. This reduces conflicts between reading transactions, reducing transactional execution times. Further, it allows an object to be simultaneously sent to nodes of reading transactions, thereby reducing the total object traveling time.

We evaluate *Bi-interval* by its makespan competitive ratio—i.e., the ratio of *Bi-interval*'s makespan (the last completion time for a given set of transactions) to the makespan of an optimal transactional scheduler. We show that *Bi-interval* improves the makespan competitive ratio of the Relay cache coherence protocol with the Greedy CM from $O(n)$ [23] to $O(log(n))$, for $n$ nodes. Also, *Bi-interval* yields an average-case makespan competitive ratio of $\Theta(log(n-k))$, for $k$ reading transactions.

We implement *Bi-interval* in a distributed TM implementation constructed using the RSTM package [5]. Our experimental studies reveal that *Bi-interval* improves transactional throughput of Relay by as much as 188% and that of LAC protocols by as much as 200%. In the worst-case (i.e., without any reading transaction), *Bi-interval* improves throughput of Relay and LAC protocols (with the Greedy CM) by as much as 30%.

Thus, the paper's contribution is the *Bi-interval* transactional scheduler. To the best of our knowledge, this is the first ever transactional scheduler for distributed TM.

The rest of the paper is organized as follows. We review past and related work in Section 2. We describe our system model and definitions in Section 3. Section 4 describes the *Bi-interval* scheduler, analyzes its performance, and gives a procedural description. We discuss *Bi-interval*'s implementation and report experimental evaluation in Section 5. The paper concludes in Section 6.

## 2 Related Work

Past works on distributed transactional memory include [4, 13, 18, 16, 24]. In [18], the authors present a page-level distributed concurrency control algorithm, which maintains several distributed versions of the same data item. In [4], the authors decompose a set of existing cache-coherent TM designs into a set of design choices, and select a combination of such choices to support TM for commodity clusters. Three distributed cache-coherence protocols are compared in [16] based on benchmarks for clusters.

In [13], Herlihy and Sun present a distributed cache-coherence protocol, called Ballistic, for metric-space networks, where the communication cost between nodes form a metric. Ballistic models the cache-coherence problem as a distributed queuing problem, due to the fundamental similarities between the two problems, and directly uses an existing distributed queuing protocol, the Arrow protocol [6], for managing transactional contention. Since distributed queuing protocols, including Arrow, do not consider contention between transactions, Ballistic suffers from a worst-case queue length of $O(n^2)$ for $n$ transactions requesting the same object. Further, its hierarchical structure degrades its scalability—e.g., whenever a node joins or departs the network, the whole structure has to be rebuilt. These drawbacks are overcome in the Relay protocol [23], which reduces the worst-case queue length by considering transactional contention, and improves scalability by using a peer-to-peer structure.

Zhang and Ravindran present a class of location-aware distributed cache-coherence (or LAC) protocols in [24]. For LAC protocols, the node which is "closer" to the object (in terms of the communication cost) always locates the object earlier. When working with the Greedy CM, LAC protocols improve the makespan competitive ratio.

None of these efforts consider transactional scheduling. However, scheduling has been explored in a number of multiprocessor TM efforts [8, 1, 22, 7, 3]. In [8], Dragojević *et. al.* describe an approach that schedules transactions based on their predicted read/write access sets. They show that such a scheduler can be 2-competitive with an optimal scheduler, and design a prediction-based scheduler that dynamically serializes transactions based on the predicted access sets. In [1], Ansari *et. al.* discuss the Steal-On-Abort transaction scheduler, which queues an aborted transaction behind the non-aborted transaction, and thereby prevent the two transactions from conflicting again (which they likely would, if the aborted transaction is immediately restarted).

Yoo and Lee present the Adaptive Transaction Scheduler (ATS) [22] that adaptively controls the number of concurrent transactions based on the contention intensity: when the intensity is below a threshold, the transaction begins normally; otherwise, the transaction stalls and do not begin until dispatched by the scheduler. Dolev *et. al.* present the CAR-STM scheduling approach [7], which uses per-core transaction queues and serializes conflicting transactions by aborting one and queueing it on the other's queue, preventing future conflicts. CAR-STM pre-assigns transactions with high collision probability (application-described) to the same core, and thereby minimizes conflicts.

Attiya and Milani present the BIMODAL scheduler [3], targeting read-dominated and bimodal (i.e., those with only early-write and read-only) workloads. BIMODAL alternates between "writing epochs" and "reading epochs" during which writing and reading transactions are given priority, respectively, ensuring greater concurrency for

reading transactions. BIMODAL is shown to significantly outperform its makespan competitive ratio in read-dominated workloads, and has an $O(s)$ competitive ratio.

Our work is inspired by the BIMODAL scheduler. The main idea of our work is also to build a read interval, which is an ordered set of reading transactions to simultaneously visit requesting nodes of those reading transactions. However, there is a fundamental trade-off between building a read interval and moving an object. If an object visits only read-requesting nodes, the object moving time may become larger. On the other hand, if an object visits in the order of the nearest node, we may not fully exploit the concurrency of reading transactions. Thus, we focus on how to build the read interval, exploiting this trade-off. Note that this tradeoff does not occur for BIMODAL.

## 3 Preliminaries

We consider Herlihy and Sun's data-flow TM model [13]. In this model, transactions are immobile, but objects move from node to node. A CM module is responsible for mediating between conflicting accesses to avoid deadlocks and livelocks. We use the Greedy CM which satisfies the work conserving [2] and pending commit [9] properties.

When a transaction attempts to access an object, the cache-coherence protocol locates the current cached copy of the object, moves it to the requesting node's cache, and invalidates the old copy (e.g., [23, 24]). If no conflict occurs, the protocol is responsible for locating the object for the requesting nodes. Whenever a conflict occurs, the CM aborts the transaction with the lower priority. The aborted transaction is enqueued to prevent it from concurrently executing again. When the current transaction commits, the transactional scheduler should decide in what order the enqueued transactions should execute. We assume that the same scheduler is embedded in all nodes for consistent scheduling. We only consider read and write operations in transactions: a transaction that only reads objects is called a reading transaction; otherwise, it is a writing transaction.

Similar to [13], we consider a metric-space network where the communication costs between nodes form a metric. We assume a complete undirected graph $G = (V, E)$, where $\mid V \mid = n$. The cost of an edge $e(i, j)$ is measured by the communication delay of the shortest path between two nodes $i$ and $j$. We use $d_G(i, j)$ to denote the cost of $e(i, j)$ in $G$. Thus, $d_G(i, j)$ forms the metric of $G$.

A node $v$ has to execute a transaction $T$, which is a sequence of operations on the objects $R_1$, $R_2$,... $R_s$, where $s \geq 1$. Since each transaction is invoked on an individual node, we use $v_{T_j}$ to denote the node that invokes the transaction $T_j$. We define $V_T = \{$ $v_{T_1}$, $v_{T_2}$, ... $v_{T_n}$ $\}$ indicating the set of nodes requesting the same object. We use $T_i \prec T_j$ to represent that transaction $T_j$ is issued a higher priority than $T_i$ by the Greedy CM. We use $\tau_j$ to denote the duration of a transaction's execution on node $j$.

**Definition 1** *A scheduler A is conservative if it aborts at least one transaction in every conflict.*

**Definition 2 (Makespan)** *Given a scheduler $A$, $makespan_i(A)$ is the time that $A$ needs to complete all the transactions in $V_{T_n}^{R_i}$ which require accesses to an object $R_i$.*

We define two types of makespans: (1) *traveling makespan*($\mathrm{makespan_i^d(A)}$), which is the total communication delay to move an object; and (2) *execution makespan*($\mathrm{makespan_A^\tau(A)}$), which is the time duration of transactions' executions including all aborted transactions.

**Definition 3 (Competitive Ratio)** *The competitive ratio (CR) of a scheduler $A$ for $V_{T_n}^{R_i}$ is $\frac{makespan_i(A)}{makespan_i(OPT)}$, where OPT is the optimal scheduler.*

**Definition 4 (Object Moving Time)** *In a given graph $G$, the object moving cost $\eta_G^A(u, V)$ is the total communication delay for visiting each node from node $u$ holding an object to all nodes in $V$, under scheduler $A$.*

We now present bounds on transactional aborts.

**Lemma 1.** *Given $n$ transactions, in the worst-case, the number of aborts of a CM in $V_{T_n}^{R_i}$ is $n^2$.*

*Proof.* Since the CM is assumed to be work-conserving, there exists at least one transaction in $V_{T_n}^{R_i}$ that will execute uninterruptedly until it commits. A transaction $T$ can be aborted by another transaction in $V_{T_n}^{R_i}$. In the worst-case, $\lambda_{CM} = \sum_{m=1}^{n-1} m \leq n^2$.

**Lemma 2.** *Given $n$ transactions, in the worst-case, the number of aborts of a conservative scheduler in $V_{T_n}^{R_i}$ is $n-1$.*

*Proof.* By Lemma 1, we know that a transaction $T$ can be aborted as many times as the number of elements of $V_{T_n}^{R_i}$. In the worst-case, the number of aborts of $T$ is $n-1$. Thus, a scheduler enqueues the requests that are aborted and an object moves along the requesting nodes. Hence, in the worst-case, $\lambda_{Scheduler} = \sum_{m=1}^{n-1} 1 = n-1$.

We now investigate the makespan for all requests for object $R_i$ by an optimal off-line scheduler.

**Lemma 3.** *The makespans of the optimal off-line scheduler are bounded as:*

$$makespan_i^d(OPT) \geq \min d_G(v_T, V_{T_{n-1}}^{R_i}), makespan_i^\tau(OPT) \geq \sum_{m=1}^{n-1} \tau_m$$

*Proof.* Suppose that $n$ concurrent requests are invoked for the same object $R_i$ and $n$-1 transactions are aborted in the worst-case. The optimal moving makespan is the summation of the minimum paths for $R_i$ to visit each requested nodes. $R_i$ starts moving from $v_T$ to each node that belongs to $V_{T_{n-1}}^{R_i}$ according to the shortest paths. Once $R_i$ visits a node, the node holds it during $\tau_m$ for processing.

**Lemma 4.** *The makespans of scheduler A are bounded as:*

$$makespan_i^d(A) \leq \max \eta_G^A(v_T, V_{T_{n-1}}^{R_i}), makespan_i^\tau(A) \leq \sum_{m=1}^{n-1} \tau_m$$

*Proof.* If $n-1$ requests arrive before the completion of a transaction at $v_T$, conflicts occur at $v_T$, which currently is $R_i$'s holding node. The scheduler builds a list of requested nodes to visit. Once a transaction is aborted, object moving and execution times are determined using the number of nodes that would be used in the worst case scenario.

## 4 The *Bi-interval* Scheduler

*Bi-interval* is similar to the BIMODAL scheduler [3] in that it categorizes requests into read and write intervals. When a request of transaction $T_2$ on node 2 arrives at node 1, there are two possible cases: 1) if the transaction $T_1$ on node 1 has committed, then the object will be moved to node 2; 2) if $T_1$ has not committed yet, then a conflict occurs. For the latter case, two sub-cases are possible. If the queue of aborted transactions is empty, the (Greedy) CM aborts the newer transaction [9]. If $T_1 \prec T_2$, $T_1$ is aborted. To handle a conflict between a reading and a writing transaction, a reading transaction is aborted to concurrently process it with other reading transactions in the future. On the other hand, if there are aborted transactions in the queue, $T_1$ and the aborted transactions will be scheduled by node 1. If $T_1$ has been previously scheduled, we use a pessimistic concurrency control strategy [21]: $T_2$ is aborted and waits until the aborted transactions are completed. Otherwise, we use the Greedy CM.

When a transaction commits and the aborted transactions wait in the queue, *Bi-interval* starts building the read and write intervals.

*Write Interval*: The scheduler finds the nearest requesting node. If the node has requested an object for a writing transaction, a write interval starts and the scheduler keeps searching for the next nearest node. When a requesting node for a reading transaction is found, the write interval ends and a read interval starts. The object is visited according to the chain of writing transactions in serial order.

*Read Interval*: When the scheduler finds the nearest node that has requested an object for a reading transaction, a read interval starts. The scheduler keeps searching for the next nearest node to build the read interval. If a requesting node for a writing transaction is found, the scheduler keeps checking the nearest node until a node requesting the object for a reading transaction appears again. If it appears, the read requesting node is joined to the read interval, meaning that the previously found requesting node(s) for a writing transaction is(are) scheduled behind the read interval. Instead of giving up the benefit of shorter object traveling times by visiting the nearest node, we achieve the alternative benefit of increased concurrency between reading transactions. Before the joining procedure to extend the read interval, the scheduler computes the trade-off between these benefits. If scheduling is completed, the object is simultaneously sent to all requesting nodes involved in the read interval. If no node for a reading transaction appears, another write interval is created.

There are two purposes for building a read interval through scheduling. First, the total execution time decreases due to the concurrent execution of reading transactions. Second, an object is simultaneously sent to some requesting nodes for reading transactions. Thus, the total traveling time in the network decreases. We now illustrate *Bi-interval*'s scheduling process with an example.

Figure 1 shows an example of a five-node network. Node 3, where the conflicts occurs, is responsible for scheduling four requests from nodes 1, 2, 4, and 5. If all requests involve write operations, node 3 schedules them as the following scheduled list: ④ → ⑤ → ① → ②. If only nodes 2 and 4 have reading transactions, node 3 yields the following scheduled list: ④ → ② → ① → ⑤. Node 3 simultaneously sends a copy of the object to nodes 4 and 2. Once the copy arrives, nodes 4 and 2 process it. After processing the object, node 4 sends a signal to node 2 letting it know about

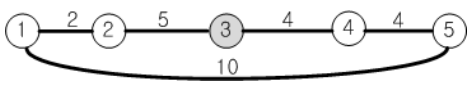



**Fig. 1.** A Five-Node Network Example for Bi-interval's Illustration

the object availability. At this time, node 2 is processing or has finished processing the object. After processing it, node 2 sends the object to node 1. The makespan is improved only when $min(\tau_3, \tau_4) > \eta_G(3, 2)$. In the meantime, if other conflicts occur while the object is being processed along the scheduled list, aborted transactions (due to the conflicts) are scheduled behind the list to ensure *progress* based on pessimistic concurrency control. This means that those aborted transactions will be handled after processing the scheduled list.

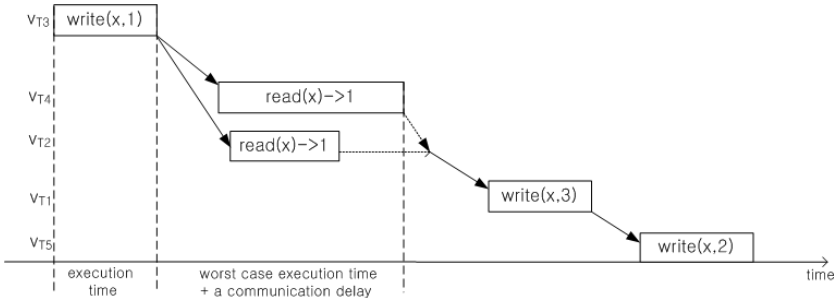



**Fig. 2.** An Example of Exploiting Parallelism in a Read Interval

Figure 2 shows an example of the parallelism in a read interval. Even though the object is simultaneously sent to nodes 4 and 2, it may not arrive at the same time. Due to different communication delays of the object and different execution times of each transaction, nodes 4 and 2 may complete their transactions at different times. According to the scheduled order, node 4 sends a signal to node 2 and node 2 immediately sends the object to node 1. Thus, the total makespan at nodes 4 and 2 includes only the worst-case execution time plus the object moving time. However, the communication delay between nodes 4 and 1 takes longer because node 2 is not the nearest node of node 4.

### 4.1 Algorithm Description

*Bi-interval* starts finding the set of consecutive nearest nodes using a nearest neighbor algorithm. $V_{T_m}^{R_i}$ denotes the set of nodes for reading transactions to obtain an object

$R_i$, where $m \geq 1$. $V_{T_w}^{R_i}$ denotes the set of nodes for writing transactions to obtain $R_i$. Suppose that the scheduler found the ordered set of $V_{T_m}^{R_i}$ and $V_{T_w}^{R_i}$ as nearest nodes.

$$\eta_G(V_{T_m}^{R_i}, v_{m+1}) - \eta_G(V_{T_m}^{R_i}, V_{T_w}^{R_i}) < \tau_\omega \tag{1}$$

When a request for a reading transaction from node $v_{m+1}$ appears after $V_{T_w}^{R_i}$ is found, $V_{T_w}^{R_i}$ is switched to $v_{m+1}$ in the condition of Equation 1 to extend the size of $V_{T_m}^{R_i}$. Equation 1 shows the condition for a reading request $v_{m+1}$ to move to the previous read interval if the difference between the delay from $V_{T_m}^{R_i}$ to $v_{m+1}$ and from $V_{T_m}^{R_i}$ to $V_{T_w}^{R_i}$ is less than or equal to $\tau_\omega$, where $\tau_\omega$ is the worst-case execution time of a reading transaction, and $1 \leq m \leq k$.

$$\min_{1 \leq I_r \leq k} (\tau_\omega \cdot I_r + \sum_{m=1}^{n-k} \tau_m + \eta_G(v_T, \bar{V}_T^{R_i})) \tag{2}$$

Here, $I_r$ is the number of read intervals, $k$ is the number of reading transactions, $v_T \in V_T^{R_i}$, and $\bar{V}_T^{R_i} = \{ v_{T_1}, v_{T_2}, \dots v_{T_{n+I_r-k}} \}$.

Equation 2 expresses the minimization of makespan for the execution and object traveling time, which is *Bi-interval*'s main objective:

**Algorithm 1**: Algorithm of *Bi-interval*

**Input**: $V_{T_n}^{R_i} = \{ v_{T_1}, v_{T_2}, \cdots v_{T_n} \}$
**Output**: $L_T^{R_i}$ /* Scheduled List */
1 $W_T^{R_i} \leftarrow \varnothing$; $L_T^{R_i} \leftarrow \varnothing$;
2 $p \leftarrow$NULL; $q \leftarrow$NULL
3 **repeat**
4  $p$=*FindNearestNode*($V_{T_n}^{R_i}$);
5  **if** *$p$ is a reading request* **then**
6  **if** *$q$ is a writing request and DetermineTotal($p$,$W_T^{R_i}$) is not OK* **then**
7  A write interval is confirmed;
8  $L_T^{R_i} \leftarrow L_T^{R_i} \cup W_T^{R_i}$;
9  $W_T^{R_i} \leftarrow \varnothing$;
10  **else**
11  $V_{T_n}^{R_i} \leftarrow V_{T_n}^{R_i} \setminus \{ p \}$;
12  $L_T^{R_i} \leftarrow L_T^{R_i} \cup \{ p \}$;
13  **else**
14  $V_{T_n}^{R_i} \leftarrow V_{T_n}^{R_i} \setminus \{ p \}$;
15  $W_T^{R_i} \leftarrow W_T^{R_i} \cup \{ p \}$;
16  $q = p$;
17 **until** $V_{T_n}^{R_i}$ *is* $\varnothing$ ;
18 **Boolean** *DetermineTotal*($p$,$W_T^{R_i}$)
19 **if** *delay*($L_T^{R_i}$, $p$)-*delay*($L_T^{R_i}$, $W_T^{R_i}$)$< \tau_{worst}$ **then**
20  **return** *OK*;
21 **return** *Not OK*;

Algorithm 1 shows a detailed description of *Bi-interval* based on the nearest neighbor problem [15], which is known to be an NP-complete problem. Algorithm 1 is invoked when a transaction is committed and aborted requests are accumulated to be scheduled. In order to solve Equation 2, we consider a greedy approach, where at each stage of the algorithm, the link with the shortest delay is taken.

In order to visit $k$ requesting nodes, the path from a starting node to visit $k$ nodes in $V_{T_k}^{R_i}$ is selected. The set of $L_T^{R_i}$ is initiated, and the last remaining element is returned as a result. If the nearest node is found and it is a request for a read operation, the algorithm checks if a read interval has been started. If a read interval was previously started, the *DetermineTotal* function is called. If it returns *OK*, the read requesting node is joined to the previous read interval. Otherwise, a new read interval is created. Note that if a new read interval is started, it means that a write interval is confirmed because the *DetermineTotal* function is called only if a read requesting node is found as the nearest node right after a write requesting node is found in a queue.

The $FindNearestNode$ function finds the smallest delay from node $v_T$ to a node in the set of $V_{T_k}^{R_i}$. Whenever the $FindNearestNode$ function returns a requesting node $p$ for a reading transaction after finding a requesting node $q$ for a writing transaction, Algorithm 1 has to check whether a write interval is created (i.e., $L_T^{R_i} \leftarrow L_T^{R_i} \cup W_T^{R_i}$) by comparing the delay corresponding to the total execution times and communication delay. The time and message complexity of Algorithm 1 is $O(n^2)$.

### 4.2 Competitive Ratio Analysis

We focus on the analysis of execution and traveling makespan competitive ratios.

**Theorem 1.** *Bi-interval's execution makespan competitive ratio is 1+*$\frac{I_r}{n-k+1}$.

*Proof.* The optimal off-line algorithm concurrently executes all reading transactions. So, *Bi-interval*'s optimal execution makespan ($\mathrm{makespan}_{\mathrm{i}}^{\tau}(\mathrm{OPT})$) is $\sum_{m=1}^{n-k+1} \tau_m$.

$$CR_{Biinterval}^{\tau} \leq \frac{\tau_\omega \cdot I_r + \sum_{m=1}^{n-k+1} \tau_m}{\sum_{m=1}^{n-k+1} \tau_m} \approx \frac{I_r + n - k + 1}{n - k + 1}$$

**Theorem 2.** *Bi-interval's traveling makespan competitive ratio is* $\log(n + I_r - k - 1)$.

*Proof. Bi-interval* follows the nearest neighbor path to visit each node in the scheduling list. We define the *stretch* of a transactional scheduler as the maximum ratio of the moving time to the communication delay—i.e., $Stretch_\eta(v_T, V_{T_{n-1}}^{R_i}) = \max \frac{\eta_G(v_T, V_{T_{n-1}}^{R_i})}{d_G(v_T, V_{T_{n-1}}^{R_i})}$ $\leq \frac{1}{2}\log(n-1) + \frac{1}{2}$ from [19]. Hence, $CR_{Biinterval}^{d} \leq \log(n + I_r - k - 1)$.

**Theorem 3.** *The total worst-case competitive ratio* $CR_{Biinterval}^{Worst}$ *of Bi-interval for multiple objects is* $O(\log(n))$.

*Proof.* In the worst-case, $I_r = k$. This means that there are no consecutive read intervals. Thus, $\mathrm{makespan}_{\mathrm{OPT}}$ and $\mathrm{makespan}_{\mathrm{Biinterval}}$ satisfy the following, respectively:

$$makespan_{OPT} = \sum_{m=1}^{n-k+1} \tau_m + \min d_G(v_T, V_{T_{n-k+1}}^{R_i}) \tag{3}$$

$$makespan_{Biinterval} = \sum_{m=1}^{n-1} \tau_m + \log(n-1) \max d_G(v_T, V_{T_{n-1}}^{R_i}) \tag{4}$$

Hence, $CR_{Biinterval}^{Worst} \leq \log(n-1)$.

We now focus on the case $I_r < k$.

**Theorem 4.** *When $I_r < k$,* Bi-interval *improves the traveling makespan* $(\text{makespan}_{\text{i}}^{\text{d}}(\text{Biinterval}))$ *as much as* $O(|\log(1 - (\frac{k-I_r}{n-1})|)$.

*Proof.*

$$\begin{aligned} \max \frac{\eta_G(v_T, V_{T_{n+I_r-k-1}}^{R_i})}{d_G(v_T, V_{T_{n-1}}^{R_i})} &= \max \Big( \frac{\eta_G(v_T, V_{T_{n-1}}^{R_i})}{d_G(v_T, V_{T_{n-1}}^{R_i})} + \frac{\varepsilon}{d_G(v_T, V_{T_{n-1}})} \Big) \\ &\leq \frac{1}{2} \log(n-k+I_r-1) + \frac{1}{2} \end{aligned} \tag{5}$$

When $I_r < k$, a read interval has at least two reading transactions. We are interested in the difference between $\eta_G(v_T, V_{T_{n-1}}^{R_i})$ and $\eta_G(v_T, V_{T_{n+I_r-k-1}}^{R_i})$. Thus, we define $\varepsilon$ as the difference between two $\eta_G$ values.

$$\max \frac{\varepsilon}{d_G(v_T, V_{T_{n-1}})} \leq \frac{1}{2} \log(\frac{n-k+I_r-1}{n-1}) \tag{6}$$

In (6), due to $I_r < k$, $\frac{n-k+I_r-1}{n-1} < 1$. *Bi-interval* is invoked after conflicts occur, so $n \neq k$. Hence, $\varepsilon$ is a negative value, improving the traveling makespan.

The average-case analysis (or, probabilistic analysis) is largely a way to avoid some of the pessimistic predictions of complexity theory. *Bi-interval* improves the competitive ratio when $I_r < k$. This improvement depends on the size of $I_r$—i.e., how many reading transactions are consecutively arranged. We are interested in the size of $I_r$ when there are $k$ reading transactions. We analyze the expected size of $I_r$ using probabilistic analysis. We assume that $k$ reading transactions are not consecutively arranged (i.e., $k \geq 2$) when $n$ requests are arranged according to the nearest neighbor algorithm. We define a probability of actions taken for a given distance and execution time. The action indicates the satisfaction for the inequality of Equation 1.

**Theorem 5.** *The expected number of read intervals $E(I_r)$ of Bi-interval is* $\log(k)$.

*Proof.* The distribution used in the proof of Theorem 5 is an independent uniform distribution. $p$ denotes the probability for $k$ reading transactions to be consecutively arranged.

$$\begin{aligned} E(I_r) &= \int_{p=0}^{1} \sum_{I_r=1}^{k} \binom{k}{I_r} \cdot p^k (1-p)^{k-I_r} dp \\ &= \sum_{I_r=1}^{k} \Big( \frac{k!}{I_r! \cdot (k-I_r)!} \int_{p=0}^{1} p^k (1-p)^{k-I_r} dp \Big) \\ &\approx \sum_{I_r=1}^{k} \frac{k!}{I_r!} \cdot \frac{k!}{(2k-I_r+1)!} \approx \log(k) \end{aligned} \tag{7}$$

We derive Equation 7 using the beta integral.

**Theorem 6.** *Bi-interval 's total average-case competitive ratio ($CR^{Average}_{Biinterval}$) is* $\Theta(\log(n-k))$.

*Proof.* We define $CR^{m}_{Biinterval}$ as the competitive ratio of node $m$. $CR^{Average}_{Biinterval}$ is defined as the sum of $CR^{m}_{Biinterval}$ of $n + E(I_r) - k + 1$ nodes.

$$CR^{Average}_{Biinterval} \leq \sum_{m=1}^{n+E(I_r)-k+1} CR^{m}_{Biinterval}$$
$$\leq \log(n + E(I_r) - k + 1) \approx \log(n - k)$$

Since $E(I_r)$ is smaller than $k$, $CR^{Average}_{Biinterval} = \Theta(\log(n - k))$.

## 5 Implementation and Experimental Evaluation

We implemented *Bi-interval* in an experimental distributed TM implementation, which was built using the RSTM package [5]. Figure 3 shows the architecture of our distributed TM implementation. As a C++ TM library, RSTM provides a template that returns a transaction-enabled wrapper object. We implemented a distributed database repository that is connected to the template for handling transactional objects. The architecture consists of two parts: local TM and remote TM. Algorithm 2 gives detailed descriptions of these parts. When an object is needed, the local TM is invoked in the requesting node and the remote TM is invoked in the object holding node.

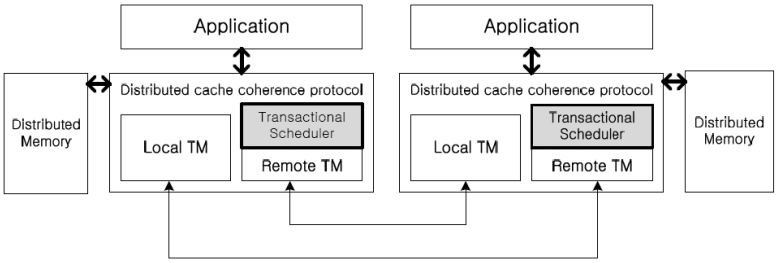



**Fig. 3.** Architecture of Experimental Distributed TM System

*Experimental Evaluation.* The purpose of our experimental evaluation is to measure the transactional throughput when the *Bi-interval* scheduler is used to augment a distributed cache-coherence-based TM. We implemented the LAC and Relay cache coherence protocols, which can be augmented with *Bi-interval*. We also included the classical RPC and distributed shared memory (DSM-L) models, both lock-based, in our experiments, as baselines. RPC and DSM-L are based on a client-server architecture in

**Algorithm 2**: Algorithm of Local and Remote TM

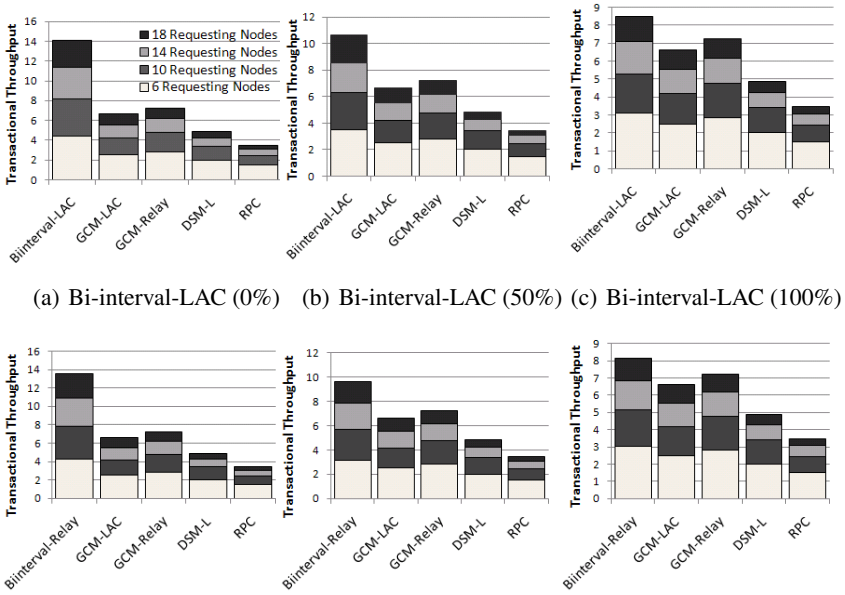
```
1  Local TM
2  if a requested object is in the local memory
   and it is validated then
3      return the object;
4  else
5      send a request message;
6  if the response is not an abort message then
7      wait for the requested object during a
       timer t
8  if t is expired then
9      resend a request message;
10 Remote TM
11 if a requested object is validated then
12     invalidate and send the object
13 else
14     a conflict is detected.
15     invoke a CM.
16     enqueue the aborted request;
17 if a transaction is committed then
18     invoke the scheduler algorithm;
```

which a server has to hold shared objects, and clients request the object from the server. In contrast, in distributed TM, a shared object is distributed for each node. The LAC and Relay protocols include a procedure to find a node that holds or will hold an object. However, for fair comparison, we assume that all nodes know the location of the shared objects. Each node runs varying number of transactions, which write to and/or read from the objects. We used an 18-node testbed in the experimental study.

Two types of transactions were used in the experiments: insertion for a writing transaction and lookup for a reading transaction in a red-black tree. We measured the transactional throughput—i.e., the number of completed (committed) transactions per second under an increasing number of requesting nodes, for the different schemes. Since the throughput varies depending on the nodes' locations, we measured the total time to complete all transactions for each requesting node and computed the average number of committed transactions per second. We also varied the number of writing transactions. In the plots, 100% means that all requesting nodes are involved in writing transactions.

Figure 4 shows the transactional throughput under 6, 10, 14, and 18 nodes requesting an object. (In all the figures, we abbreviate the Greedy CM as GCM.) We observe that Bi-interval-LAC and GCM-LAC, and Bi-interval-Relay and GCM-Relay exhibit the same behavior under no conflicts. However, once a conflict occurs, an aborted transaction is not aborted again in Bi-interval-LAC (Relay). In GCM-LAC and GCM-Relay, the requesting nodes that have been aborted request an object again, so that increases the communication delay. Unless a request arrives at the object holding node right after a transaction is committed, the up-to-date copy of the object has to wait until the request arrives. However, the object holding node with *Bi-interval* immediately sends the copy to the requesting node right after it commits. The purpose of the Relay protocol is to reduce the number of aborts, so GCM-Relay has better performance than GCM-LAC. If no conflict occurs, Relay performs better than LAC. However, when a conflict occurs, Bi-interval-LAC performs better than Bi-interval-Relay due to the advantage of the parallelism of the reading transactions involved in the aborts. Bi-interval-LAC (0%) boosts the performance to the maximum possible transactional throughput.

It is interesting to observe the throughput difference between Bi-interval-LAC (100%) and Bi-interval-Relay (100%) shown in Figures 4(c) and 4(f), respectively. The cause




(a) Bi-interval-LAC (0%) (b) Bi-interval-LAC (50%) (c) Bi-interval-LAC (100%)

(d) Bi-interval-Relay (0%) (e) Bi-interval-Relay (50%) (f) Bi-interval-Relay (100%)

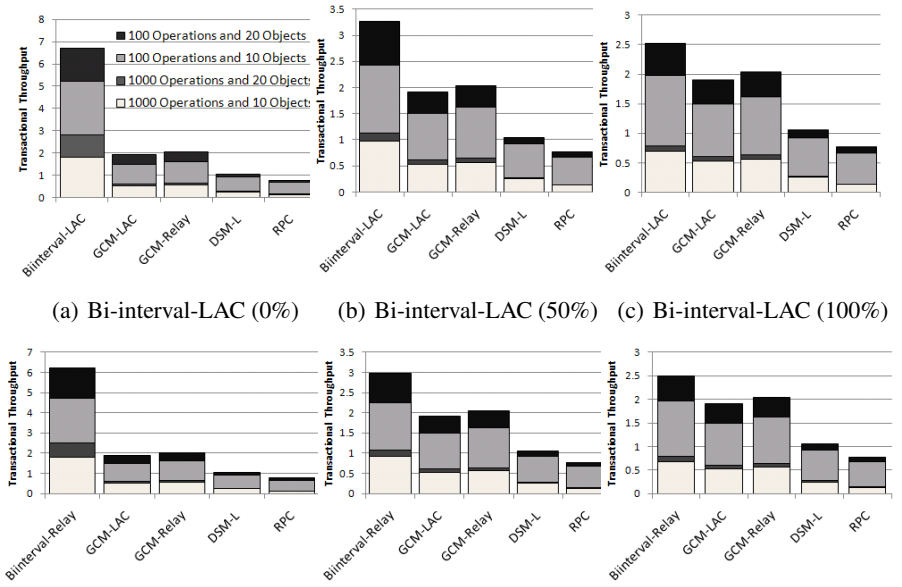
**Fig. 4.** Transactional Throughput Under Increasing Requesting Nodes/Single Object

of the throughput deterioration of Bi-interval-Relay (100%) is the high object moving delay. Even though transactions are aborted less in Relay, the copy of an object moves along a fixed spanning tree, which may not be the shortest path. Bi-interval-LAC, which has a relatively higher number of aborts, achieves the nearest node for aborted transactions.

In DSM-L and RPC, if an object is invalidated, they block new requests to protect the previous request. Specifically, Bi-interval-LAC and Relay (0%) simultaneously send all requesting nodes involved in aborted transactions to the object. Thus, they significantly outperform the other schemes.

We now turn our attention to throughput when a transaction uses multiple objects, and performs increasing number of object operations, causing longer transaction execution times. We measured throughput under 10 and 20 objects and 1000 and 100 operations. The objects are not related to each other, but the transaction has to use them together, so a transaction's execution time is longer. Figure 5 shows the results. GCM-LAC and GCM-Relay suffer from large number of aborts due to increasing number of objects and operations. They show greater throughput degradation from the number of aborts than that under shorter transactions. We observe a maximum improvement of 30% for Bi-interval under 100% updates.

Due to space restrictions, additional results are omitted; they can be found in [14].




(a) Bi-interval-LAC (0%) (b) Bi-interval-LAC (50%) (c) Bi-interval-LAC (100%)

(d) Bi-interval-Relay (0%) (e) Bi-interval-Relay (50%) (f) Bi-interval-Relay (100%)

**Fig. 5.** Transactional Throughput Under Multiple Objects and Increasing Operations

## 6 Conclusions

Our work shows that the idea of grouping concurrent requests into read and write intervals to exploit concurrency of reading transactions — originally developed in BIMODAL for multiprocessor TM — can also be successfully exploited for distributed TM. Doing so poses a fundamental trade-off, however, one between object moving times and concurrency of reading transactions. *Bi-interval*'s design shows how this trade-off can be exploited towards optimizing transactional throughput.

Several directions for further work exist. First, we do not consider link or node failures. Second, *Bi-interval* does not support nested transactions. Additionally, we assume that transactional objects are unlinked. If objects are part of a linked structure (e.g., graph), scalable cache-coherence protocols and schedulers must be designed.

## References

1. Mohammad Ansari et al. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In André Seznec, Joel S. Emer, et al., editors, *HiPEAC*, volume 5409 of *LNCS*, pages 4–18. Springer, 2009.
2. Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *PODC '06*, pages 308–315, 2006.
3. Hagit Attiya and Alessia Milani. Transactional scheduling for read-dominated workloads. In *OPODIS '09*, pages 3–17, Berlin, Heidelberg, 2009. Springer-Verlag.

4. Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP '08*, pages 247–258, 2008.
5. Luke Dalessandro, Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Capabilities and limitations of library-based software transactional memory in c++. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*. Portland, OR, Aug 2007.
6. Michael J. Demmer and Maurice Herlihy. The arrow distributed directory protocol. In *DISC '98*, pages 119–133, 1998.
7. Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *PODC '08*, pages 125–134, 2008.
8. Aleksandar Dragojević, Rachid Guerraoui, et al. Preventing versus curing: avoiding conflicts in transactional memories. In *PODC '09*, pages 7–16, 2009.
9. Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC '05*, pages 258–264, 2005.
10. Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.
11. Lance Hammond, Vicky Wong, et al. Transactional memory coherence and consistency. *SIGARCH Comput. Archit. News*, 32(2):102, 2004.
12. Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03*, page 522, 2003.
13. Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.
14. Junwhan Kim and Binoy Ravindran. On transactional scheduling in distributed transactional memory systems. Available at `http://www.real-time.ece.vt.edu/SSS-2010-TR.pdf`, 2010. Technical Report, Virginia Tech, Real-Time Lab.
15. Jon Kleinberg and Eva Tardos. Algorithm design, 2005.
16. Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. DiSTM: A software transactional memory framework for clusters. In *ICPP '08*, pages 51–58, Washington, DC, USA, 2008. IEEE Computer Society.
17. Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *PPoPP '06*, pages 209–220, 2006.
18. Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP '06*, pages 198–208, Mar 2006.
19. Daniel J. Rosenkrantz, Richard Edwin Stearns, and Philip M. Lewis II. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6(3):563–581, 1977.
20. William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05*, pages 240–248, 2005.
21. Nehir Sonmez, Tim Harris, Adrian Cristal, Osman S. Unsal, and Mateo Valero. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. *Parallel and Distributed Processing Symposium, International*, 0:1–10, 2009.
22. Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA '08*, pages 169–178, 2008.
23. Bo Zhang and Binoy Ravindran. Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In *OPODIS '09*, pages 48–53, 2009.
24. Bo Zhang and Binoy Ravindran. Location-aware cache-coherence protocols for distributed transactional contention management in metric-space networks. In *SRDS '09*, pages 268–277, 2009.