Relay: A Cache-Coherence Protocol for Distributed Transactional Memory

Bo Zhang and Binoy Ravindran

ECE Dept., Virginia Tech
Blacksburg VA 24061, USA
{alexzbzb,binoy}@vt.edu

Abstract. Transactional memory is an alternative programming model for managing contention in accessing shared in-memory data objects. Distributed transactional memory promises to alleviate difficulties with lock-based (distributed) synchronization and object performance bottlenecks in distributed systems. The design of the cache-coherence protocol is critical to the performance of distributed transactional memory systems. We evaluate the performance of a cache-coherence protocol by measuring its worst-case competitive ratio — i.e., the ratio of its makespan to the makespan of the optimal cache-coherence protocol. We establish the upper bound of the competitive ratio and show that it is determined by the worst-case number of abortions, maximum locating stretch, and maximum moving stretch of the protocol — the first such result. We present the Relay protocol, a novel cache-coherence protocol, which optimizes these values, and evaluate its performance. We show that Relay's competitive ratio is significantly improved by a factor of $O(N_i)$ for N_i transactions requesting the same object when compared against past distributed queuing protocols.

1 Introduction

Conventional synchronization methods based on locks and condition variables are inherently error-prone. For example, with fine-grained locking, where each component of a data structure is protected by a lock, programmers must acquire only necessary and sufficient locks to obtain maximum concurrency without compromising safety, and must avoid deadlocks when acquiring multiple locks. Furthermore, lock-based code is non-composable. For example, atomically moving an element from one hash table to another using those hash tables' pre-existing atomic methods (e.g., insert, delete) is not possible in a straightfoward manner. For these and other reasons, lock-based concurrent code is difficult to reason about, program, and maintain [11].

Transactional memory (TM) is an alternative synchronization model (for shared in-memory data objects) that promises to alleviate the difficulties with lock-based synchronization. A transaction is an explicitly delimited sequence of steps that is executed atomically by a single thread. Transactions read and write shared objects. Two transactions *conflict* if they access the same object and one access is a write. The transactional approach to *contention management* [12] guarantees atomicity by ensuring that whenever a conflict occurs, only one of the transactions involved can proceed. Thus, a transaction ends by either committing (i.e., its operations take effect), or by aborting (i.e., its operations have no effect). If a transaction aborts, it is typically retried until it commits. Transactional memory API for multiprocessors have been proposed in hardware [8], in software [10, 16], and in hardware/software combination [4].

In this paper, we focus on distributed transactional memory. We are motivated by the difficulties of lock-based synchronization that plague distributed control-flow programming models such as RPCs. For example, distributed deadlocks (e.g., due to RPCs that become remotely blocked on each other) and livelocks are unavoidable for such solutions. Furthermore, in the RPC model, an object can become a "hot spot," and thus a performance bottleneck. In the data-flow distributed TM model of [13] (that we also consider), such bottlenecks can be reduced by exploiting locality: move the object to nodes. 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. Distributed (data-flow) TM can therefore alleviate these difficulties, where distributed transactional conflicts and object inconsistencies are resolved through distributed contention managers and cache-coherence protocols, respectively. See [13] for an excellent discussion on these issues.

Different TM models for distributed systems have been proposed in the past. In [13], Herlihy and Sun identify three competing models: the control flow model, in which data objects are typically immobile, and computations move from node to node via RPCs; the data flow model, where transactions are immobile (running on a single node) and data objects move from node to node; and the hybrid model, where data objects are migrated depending on an array of heuristics such as size and locality. These transactional models make different trade-offs. Past work on multiprocessors [12] suggest that the data flow model can provide better performance than the control flow model on exploiting locality, reducing communication overhead, and supporting fine-grained synchronization.

Distributed TM differs from multiprocessor TM in two key aspects. First, multiprocessor TM designs extend built-in cache-coherence protocols that are already supported in modern multiprocessor architectures. Distributed systems with nodes linked by communication networks typically do not come with such built-in protocols. A distributed cache-coherence protocol has to be therefore designed. When a transaction attempts to access an object, the cache-coherence protocol must locate the current cached copy of the object, and move it to the requesting node's cache, invalidating the old copy. For example, in [13], Herlihy and Sun present a *Ballistic* cache-coherence protocol based on hierarchical clustering for tracking and moving up-to-date copies of cached objects.

Secondly, the communication costs for distributed cache-coherence protocols to locate the copy of an object in distributed systems are orders of magnitude larger than that in multiprocessors and are often non-negligible. Such costs are typically determined by the different physical locations of nodes that invoke transactions, as well as that of the performance of the cache-coherence protocol used. These costs directly affect the system performance.

In this paper, we focus on the design of a cache-coherence protocol to minimize its worst-case competitive ratio, which is the ratio of its makespan (the last completion time for a given set of transactions) to the makespan of an optimal cache-coherence protocol. We first establish the upper bound of the competitive ratio and show that it is determined by the worst-case number of abortions, maximum locating stretch, and maximum moving stretch of the protocol. The design of a cache-coherence protocol should therefore minimize these values.

Past works on transactional memory in distributed systems include [2], [13], and [14]. In [14], the authors present a page-level distributed concurrency control algorithm, which maintains several distributed versions of the same data item. In [2], 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. None of these works present theoretical analysis of the fundamental properties of TM for distributed systems, such as performance upper bounds of cache-coherence protocols, which is our focus.

In [13], Herlihy and Sun present a Ballistic distributed cache-coherence protocol in a metric-space network where the communication costs between nodes form a metric. The protocol's performance is evaluated by measuring its *stretch*, which is the ratio of the protocol's communication cost for obtaining a cached copy of an object to that of the optimal communication cost. The Ballistic protocol mainly suffers from two drawbacks. First, it employs an existing distributed queuing protocol, which does not consider the contention between two transactions, and the worst-case queue length, which is $O(N_i^2)$ for N_i transactions requesting the same object. Second, its hierarchical structure degrades its scalability — e.g., whenever a node joins or departs, the whole structure has to be rebuilt.

Our cache-coherence protocol design is motivated by the distributed queuing problem [9]. Similar to this problem, distributed TM must also synchronize accesses to mobile objects in a network. Hence, the principles of management of distributed queues also apply to distributed TM: the transaction requests have to be ordered in a queue and each transaction needs to know the location of its successor in the queue so that it knows where to forward the object to. However, most traditional distributed queuing protocols often do not consider the contention between transactions: an abortion of a transaction increases the length of the queue since the transaction has to be restarted. We show that for the arrow protocol [15], which is a simple distributed queuing protocol, the worst-case number of total abortions is $O(N_i^2)$ for N_i transactions requesting the same object. We improve this bound in our cache-coherence protocol design by a factor of $O(N_i)$.

We present a novel cache-coherence protocol, called the Relay protocol. Similar to the arrow protocol, the Relay protocol works on a network spanning tree. Hence, its maximum locating stretch and maximum moving stretch are determined by the maximum stretch of the underlying spanning tree. The Relay protocol efficiently reduces the worst-case number of total abortions to $O(N_i)$. As a result, we show that the protocol has a better worst-case competitive ratio than the arrow protocol by a factor of $O(N_i)$.

Thus, the paper's contribution is twofold. First, we identify the three factors that critically affect the performance of a cache-coherence protocol. Second, we present the Relay protocol, illustrate how these factors are optimized in its design, and show that its worst-case competitive ratio is better than that of the arrow protocol by a factor of $O(N_i)$. To the best of our knowledge, these are the first such results.

The rest of the paper is organized as follows. We present our system model and formulate our problem in Section 2. We analyze the general case of the competitive ratio and establish its upper bound in Section 3. Section 4 presents the Relay protocol. The paper concludes in Section 5.

2 System Model and Problem Description

Network Model. We consider Herlihy and Sun's metric-space network model [13]. Let $G = (V, E, d)$ be a weighted connected graph, where $|V| = n$ and d is a function that maps E to the set of positive real numbers. Specifically, we use $d(u, v)$ to denote the communication cost of the edge $e(u, v)$. For two nodes u and v in V, let $\text{dist}_G(u, v)$ denote the *distance* between them in G, i.e., the length of a shortest path between u and v.

We assume that the proposed Relay protocol runs on a $fixed$-rooted spanning tree of G. Given a spanning tree T of G, we define the distance in T between a pair of two nodes, u and v, to be the sum of the lengths of the edges on the unique path in T between u and v. Now, we define the *stretch* of u and v in T with respect to G as: $\text{str}_{T,G}(u, v) = \frac{\text{dist}_T(u,v)}{\text{dist}_G(u,v)}$.

Let the *maximum stretch* of T with respect to G be denoted as: $\text{max-str}(T, G) = \max_{u,v\in V}\{\text{str}_{T,G}(u, v)\}$.

When there is no ambiguity, we omit G, for convenience, i.e., we say $\text{str}_T(u, v)$ and $\text{max-str}(T)$. the graph G is clear from the context, we simply write $\text{str}_T(u, v)$ and $\text{max-str}(T)$. We define the *diameter* of G as: $\text{Diam} = \max_{u,v,x,y\in V}\{\frac{\text{dist}_G(u,v)}{\text{dist}_G(x,y)}\}$.

Transaction Model. We are given a set of $m \geq 1$ transactions $T_1, \ldots, T_m$ and a set of $s \geq 1$ objects $R_1, \ldots, R_s$. Since each transaction is invoked on an individual node, we use v_{T_i} to denote the node that invokes the transaction T_i, and $V_T = \{v_{T_1}, \ldots, v_{T_m}\}$. We use $T_i \prec T_j$ to represent that transaction T_i is issued a higher priority than T_j by the contention manager (see the distributed transactional memory model).

Each transaction is a sequence of actions, each of which is an access to a single object. Each transaction T_j requires the use of $R_i(T_j)$ units of object R_i for one of its actions. If T_j updates R_i, i.e., a write operation, then $R_i(T_j) = 1$. If it reads R_i without updating, then $R_i(T_j) = \frac{1}{n}$, i.e., the object can be read by all nodes in the network simultaneously. When $R_i(T_j) + R_i(T_k) > 1$, T_j and T_k conflict at R_i. We use $v_{R_i}^0$ to denote the node that holds R_i at the start of the system, and $v_{R_i}^j$ to denote the j^{th} node that fetches R_i. We denote the set of nodes that requires the use of the same object R_i as $V_T^{R_i} := \{v_{T_j} | R_i(T_j) \geq 0, j = 1, \ldots, m\}$.

An execution of a transaction T_j is a sequence of *timed actions*. Generally, there are four action types that may be taken by a single transaction: *write*, *read*, *commit*, and *abort*. When a transaction is started on a node, a cache-coherence protocol is invoked to locate the current copy of the object in the network and fetch it. The transaction then starts its action sequence and may perform local computations (not involving access to objects) between consecutive actions. A transaction completes either with a commit or an abort. The duration of transaction T_j running locally (without taking into account the time for fetching objects) is denoted by τ_i.

Distributed Transactional Memory Model. We consider Herlihy and Sun's data-flow model [13] to support the transactional memory API in a distributed system. In this model, transactions are immobile (running at a single node), but objects move from node to node. Transactional synchronization is optimistic: a transaction commits only if no other transaction has executed a conflicting access. A *contention manager* module is responsible for mediating between conflicting accesses to avoid deadlocks and livelocks. The core of this design is an efficient *distributed cache-coherence* protocol. A distributed transactional memory system uses a distributed cache-coherence protocol for read/write operations. When a transaction attempts to access an object, the cache-coherence protocol must locate the current cached copy of the object, move it to the requesting node's cache, and invalidate the old copy.

Each node is assumed to have a *transactional memory proxy* module that provides interfaces to the application and to proxies at other nodes. This module performs the following functions:

— *Data Object Management*: An application informs the proxy to open an object when it starts a transaction. The proxy is responsible for fetching a copy of the object requested by the transaction, either from its local cache or from other nodes. When the transaction requests to commit, the proxy checks whether any object opened by the transaction has been modified by other transactions. If not, the proxy makes the transaction's tentative changes to the object permanent; otherwise discards them.

— *Cache-Coherence Protocol Invocation*: The proxy is responsible for invoking a cache-coherence protocol when needed. When a new data object is created in the local cache, the proxy invokes the cache-coherence protocol to *publish* it in the network. When an object is requested by a read access and is not in the local cache, the proxy invokes the cache-coherence protocol to *look-up* the object and fetch a read-only copy. If it is a write request, the proxy invokes the cache-coherence protocol to *move* the object to its local cache.

— *Contention Management*: When a transaction requests for an object that is currently used by an active local transaction, the proxy can either abort the local transaction and make the object available, or it can postpone a response to give the local transaction a chance to commit. This decision is made by a globally consistent contention management policy that avoids deadlocks and livelocks. An efficient contention management policy should guarantee progress—i.e., at any given time, there exists at least one transaction that proceeds to commit without interruption. For example, the Greedy contention manager [7] guarantees that the transaction with the highest priority can be executed without interruption, using a globally consistent priority policy that issues priorities to transactions.

Problem Statement. We evaluate the performance of a distributed transactional memory system by measuring its *makespan*. Given a set of transactions accessing a set of objects under a contention manager A and a cache-coherence protocol C, makespan(A, C) denotes the duration that the given set of transactions are successfully executed under the contention manager A and cache-coherence protocol C. We assume a fixed contention manager A, which satisfies the *work conserving* [1] and *pending commit* [7] properties.

Definition 1 *A contention manager is* work conserving *if it always lets a maximal set of non-conflicting transactions to run.*

Definition 2 *A contention manager obeys the* pending commit *property if, at any given time, some running transaction will execute uninterrupted until it commits.*

For example, as shown in [1], the Greedy manager satisfies both properties.

We use makespan(A, Opt) to denote the makespan of the optimal cache-coherence protocol with respect to A. We evaluate the performance of a cache-coherence protocol C with respect to A by measuring its *competitive ratio*:

Definition 3 (Competitive Ratio) $CR(A, C) = \frac{makespan(A,C)}{makespan(A,\textsc{Opt})}$.

When there is no ambiguity on the contention manager used, for convenience, we drop it from the notations—i.e., we simply write makespan(C), makespan(Opt), and CR(C).

Our goal is to design a cache-coherence protocol C to minimize its competitive ratio.

3 Competitive Ratio Analysis

We first analyze the makespan of the optimal cache-coherence protocol makespan(Opt). Let the makespan of a set of transactions which require accesses to an object R_i, be denoted as makespan$_i$. It is composed of three parts:

(1) Traveling Makespan (makespan$_i^d$): the total communication cost for R_i to travel in the network.
(2) Execution Makespan (makespan$_i^\tau$): the duration of transactions' executions involving R_i, including all successful and aborted executions; and
(3) Waiting Makespan (makespan$_i^w$): the time that R_i waits for a transaction request.

Generally, a cache-coherence protocol performs two functions: 1) locating the up-to-date copy of the object and 2) moving it in the network to meet transactions' requests. We define these costs as follows:

Definition 4 (Locating Cost) *In a given graph G, the locating cost $\delta^C(u, v)$ is the communication cost for a transaction request invoked by node u to travel in the network, to successfully locate an object held by node v, under a cache-coherence protocol C.*

Definition 5 (Moving Cost) *In a given graph G, the moving cost $\zeta^C(u, v)$ is the communication cost for an object held by node u to travel in the network to node v, which invokes a transaction request of the object, under a cache-coherence protocol C.*

For the set of nodes $V_T^{R_i}$ that invoke transactions with requests for object R_i, we build a *complete* graph $G_i = (V_i, E_i, d_i)$, where $V_i = \{V_T^{R_i} \bigcup v_{R_i}^0\}$ and $d_i(u, v) = \text{dist}_G(u, v)$. We use $H(G_i, v_{R_i}^0, v_{T_j})$ to denote the cost of the *minimum-cost Hamiltonian path* that visits each node from $v_{R_i}^0$ to v_{T_j} exactly once. Now, we have:

Theorem 1

$$makespan_i^d(\textsc{Opt}) \geq \min_{v_{T_j} \in V_T^{R_i}} H(G_i, v_{R_i}^0, v_{T_j}), \quad makespan_i^\tau(\textsc{Opt}) \geq \sum_{v_{T_j} \in V_T^{R_i}} \tau_j$$

$$makespan_i^w(\textsc{Opt}) \geq \sum_{v_{T_j} \in V_T^{R_i}} \min_{v_{T_k} \in \{V_T^{R_i} \bigcup v_{R_i}^0\}} dist_G(v_{T_k}, v_{T_j})$$

Proof. The execution of the given set of transactions with the minimum makespan schedules each transaction exactly once, which implies that R_i only has to visit each node in $V_T^{R_i}$ once. In this case, the node travels along a Hamiltonian path in G_i starting from $v_{R_i}^0$. Hence, we can lower-bound the traveling makespan by the cost of the minimum-cost Hamiltonian path and the execution makespan by the sum of τ_j. For the optimal cache-coherence protocol, each object is located via the shortest path. The theorem follows.

Let $\lambda_C^*(j)$ denote T_j's worst-case number of abortions under cache-coherence protocol C and $\lambda_C(j) = \lambda_C^*(j) + 1$. Let Λ_C^* denote the worst-case number of total transactions' abortions under C and $\Lambda_C = \Lambda_C^* + N_i$. Generally, we have the following theorem for a cache-coherence protocol C:

Theorem 2 $makespan_i^d(C) \leq \Lambda_C \cdot \max_{v_{T_j} \in V_T^{R_i}} \{\max_{v_{T_k} \in \{V_T^{R_i} \bigcup v_{R_i}^0\}} \zeta^C(v_{T_j}, v_{T_k})\}$

$$makespan_i^\tau(C) \leq \sum_{v_{T_j} \in V_T^{R_i}} \{\lambda_C(j) \cdot \tau_j\}, \quad makespan_i^w(C) \leq \Lambda_C \cdot \max_{v_{T_j} \in V_T^{R_i}} \{ \max_{v_{T_k} \in \{V_T^{R_i} \bigcup v_{R_i}^0\}} \delta^C(v_{T_j}, v_{T_k})\}$$

Hence, we have the following relationship of $\mathrm{CR}_i(C)$, the competitive ratio of cache-coherence protocol C for transactions requesting object R_i:

$$\mathrm{CR}_i^d(C) \leq \frac{\Lambda_C \cdot \max_{v_{T_j} \in V_T^{R_i}} \{\max_{v_{T_k} \in \{V_T^{R_i} \bigcup v_{R_i}^0\}} \zeta^C(v_{T_j}, v_{T_k})\}}{\sum_{v_{T_j} \in V_T^{R_i}} \{\min_{v_{T_k} \in \{V_T^{R_i} \bigcup v_{R_i}^0\}} \mathrm{dist}_G(v_{T_j}, v_{T_k})\}}, \quad \mathrm{CR}_i^\tau(C) \leq \max_{v_{T_j} \in V_T^{R_i}} \lambda_C(j)$$

$$\mathrm{CR}_i^w(C) \leq \frac{\Lambda_C \cdot \max_{v_{T_j} \in V_T^{R_i}} \{\max_{v_{T_k} \in \{V_T^{R_i} \bigcup v_{R_i}^0\}} \delta^C(v_{T_j}, v_{T_k})\}}{\sum_{v_{T_j} \in V_T^{R_i}} \{\min_{v_{T_k} \in \{V_T^{R_i} \bigcup v_{R_i}^0\}} \mathrm{dist}_G(v_{T_j}, v_{T_k})\}}$$

We define the *locating stretch* and *moving* stretch of u and v under cache-coherence protocol C as: $\mathrm{Str}_C^\delta(u,v) = \frac{\delta^C(u,v)}{\mathrm{dist}_G(u,v)}$ and $\mathrm{Str}_C^\zeta(u,v) = \frac{\zeta^C(u,v)}{\mathrm{dist}_G(u,v)}$. Let the *maximum locating stretch* and *maximum moving stretch* with respect to C be denoted, respectively, as: $\text{max-Str}_C^\delta = \max_{u,v \in V}\{\frac{\delta^C(u,v)}{\mathrm{dist}_G(u,v)}\}$ and $\text{max-Str}_C^\zeta = \max_{u,v \in V}\{\frac{\zeta^C(u,v)}{\mathrm{dist}_G(u,v)}\}$. Let $N_i = |V_T^{R_i}|$, i.e, N_i represents the number of transactions that request access to object R_i. Now we have the following theorem:

Theorem 3 $CR_i(C) \leq \max\{\max_{v_{T_j} \in V_T^{R_i}} \lambda_C(j), \frac{\Lambda_C}{N_i} \cdot \max\{\textit{max-Str}_C^\zeta, \textit{max-Str}_C^\delta\} \cdot \textit{Diam}\}$

Remarks: Theorem 3 gives the upper bound of the competitive ratio of cache-coherence protocol C. Clearly, the design of a cache-coherence protocol should therefore focus on minimizing its worst-case number of abortions, maximum locating stretch, and maximum moving stretch.

4 The Relay Protocol

Rationale. Our work is motivated by the arrow protocol of Raymond [15], which is a simple *distributed queuing* protocol based on path reversal on a network spanning tree. Distributed queuing is a fundamental problem in the management of synchronization accesses to mobile objects in a network. When multiple nodes in the network request an object concurrently, the requests must be queued in some order, and the object travels from one node to another down the queue. To manage such a distributed queue, an efficient distributed queuing protocol must solve two problems: a) how to order the requests from different nodes into a single queue; and b) how to provide the necessary information to nodes such that each node knows the location of its successor in the queue and the object can be forwarded down the queue. Note that the protocol is "distributed" in the sense that no single node needs to have the global knowledge of the queue. Each node only has to know its successor in the queue and forward the object to it.

The arrow protocol runs on a fixed spanning tree T of G. Each node v keeps an "arrow" or a pointer $p(v)$ to itself or to one of its neighbors in T. If $p(v) = v$, then v is the tail of the queue, i.e., the next request should be forwarded to v. In this case, the node v is defined as a *"sink"*. Clearly, at any time, there exists only one sink for each object. If $p(v) = u$, then $p(v)$ only knows the "direction" of the tail of the queue and the request is forwarded following that direction.

The protocol works based on path reversal. When an object is created by a node, the arrows are initialized such that following the arrows from any node leads to the object. A node v requests the object by sending a *find* message to $p(v)$. When a node u receives a find message from its neighbor w, there are two possible cases: a) if $p(u) \neq u$, then it forwards the find message to $p(u)$ and flips $p(u)$ to point to w; and b) if $p(u) = u$, then the find message has arrived at the tail of

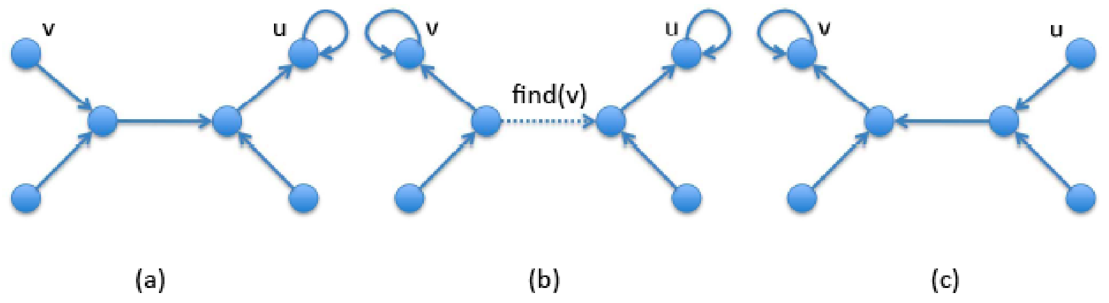

Fig. 1. The Arrow Protocol

the queue. The object will move to v after it arrives at u, and $p(u)$ is also flipped to point to w. See Figure 1 for an example of the arrow protocol. We only give an informal description here and more details can be found in [5].

The arrow protocol is attractive as a candidate cache-coherence protocol. In the context of distributed transactional memory, nodes request access to mobile objects in the network. Hence, a cache-coherence protocol has to be able to arrange the requests to be ordered in a queue. If we directly apply the arrow protocol as a cache-coherence protocol, we can immediately have the following relationships: max-$\text{Str}^{\delta}_{arrow}$ = max-str(T) and max-$\text{Str}^{\zeta}_{arrow} = 1$. Hence, the maximum locating stretch of the arrow protocol is the maximum stretch of the underlying spanning tree, and the maximum moving stretch of the arrow protocol is 1 — i.e., the object can be directly moved via the shortest path.

However, the difference between the arrow protocol and a cache-coherence protocol is that the arrow protocol does not consider the contention between two transactions. Hence, the arrow protocol is not able to reduce the worst-case number of abortions $\lambda_C(j)$. We have the following theorem:

Theorem 4 $\max_{v_{T_j} \in V_T^{R_i}} \lambda_{arrow}(j) \leq N_i, \quad \Lambda_{arrow} \leq \frac{N_i(N_i+1)}{2}$

Proof. Note that we assume a contention manager with work conserving and pending commit properties. Hence, we know that at any time, there exists at least one transaction in $V_T^{R_i}$ that will execute uninterruptedly until it commits. Let T^*_{arrow} denote the transaction with the maximum worst-case number of abortions in $V_T^{R_i}$ under the arrow protocol. We first prove that, for each time that T^*_{arrow} is aborted by another transaction in $V_T^{R_i}$, at least one transaction in $V_T^{R_i}$ will commit before T^*_{arrow} is aborted again.

Assuming that T^*_{arrow} is aborted by another transaction T^{head}_{arrow}, we have $T^{head}_{arrow} \prec T^*_{arrow}$. According to the arrow protocol, the "arrows" are now pointed to the tail of the queue, which is the latest transaction requesting the object. Transaction T^*_{arrow}'s new request will be forwarded to the tail of the queue. We now focus on the set of transactions between T^{head}_{arrow} and T^*_{arrow} in the queue, denoted by S. Let T' be the transaction with the highest priority in S. If $T' \prec T^{head}_{arrow}$, then T' will commit before it forwards the object down the queue. Otherwise, T^{head}_{arrow} will commit. In both cases, at least one transaction will commit before the object is forwarded to T^*_{arrow} again.

Now, it is easy to prove that $\max_{v_{T_j} \in V_T^{R_i}} \lambda_{arrow}(j) \leq N_i$, as for each time that T^*_{arrow} is aborted, at least one transaction in $V_T^{R_i}$ commits. By induction, we can further prove that the second-maximum worst-case number of abortions in $V_T^{R_i}$ is at most $N_i - 1$, the third-maximum worst-case number of abortions is at most $N_i - 2$, etc. The theorem follows. We now have the following corollary:

Corollary 1 $CR_i(arrow) \leq \frac{(N_i+1)}{2} \cdot \textit{max-str}(T) \cdot \textit{Diam}$

Hence, a new cache-coherence protocol should focus on minimizing the number transaction abortions to improve the upper-bound of the competitive ratio.

Protocol Description. We design a novel cache-coherence protocol, called the Relay protocol, based on a fixed spanning tree T on G. The Relay protocol inherits the advantages of the arrow protocol and significantly reduces the number of transaction abortions by a factor of $O(N_i)$. We start with an informal description of the protocol.

The protocol is initialized in the same way as the arrow protocol. The node where the object resides is selected to be the tail of the queue. Each node $v \in V$ maintains a pointer $p(v)$ and is initialized so that following the pointers from any node leads to the tail.

After the initialization, the protocol works as follows. To request the object, a transaction T_p invoked by node v sends a find message to node $p(v)$. Note that $p(v)$ is not modified when a find message is forwarded. If a node w between v and the tail of the queue receives a find message, it simply forwards the find message to $p(v)$. At the end, the find message will be forwarded to the tail of the queue without changing any pointers.

The find message from v keeps a *path vector* $\boldsymbol{r}(v)$ to record the path it travels. Each node receiving the find message from v appends its ID to $\boldsymbol{r}(v)$. When the find message arrives at the tail of the queue, the vector $\boldsymbol{r}(v)$ records the path from v to the tail. Suppose the tail of the queue x receives a find message from node v. Now, there are two possible cases: a) if the transaction T_x on x has committed, then the object will be moved to p; and b) if the transaction T_x has not committed, the contention manager will compare the priorities of T_x and T_p. We discuss this scenario case by case.

- Case 1: If $T_p \prec T_x$, then T_x is aborted and the object will be moved to p. Node p stores a field $next(p) = T_x$ after receiving the object.
- Case 2: If $T_x \prec T_p$, then T_p will be postponed to let T_x commit. Node x stores a field $next(x) = T_p$. Node x may receive multiple find messages since the pointers are not changed before the object is moved. Suppose it receives another find message from node u. If $T_u \prec T_x$, then it falls into Case 1. If $T_x \prec T_u$, then the contention manager compares the priorities of $next(x)$ (in this case it is T_p) and T_u. If $T_p \prec T_u$, then the find message from u is forwarded to p. If $T_u \prec T_p$, then u sets $next(x)$ to T_u and forwards the find message from p to u.

The key idea of the Relay protocol is the way it updates the pointers and path vectors to make those operations feasible. When the object is available at node x, it will be moved to $next(x)$ via the path from x to $next(x)$ of the spanning tree T. The problem is, how does x learn that path so that the object can be correctly moved? Note that the Relay protocol uses path vectors to record the path. Suppose that x moves an object to v. The Relay protocol keeps a *route vector* $\boldsymbol{route}$ at x which records the path from v to x by copying the path vector $\boldsymbol{r}(v)$ after the find message from v arrives. Hence, node x is able to move the object by following the reverse path saved in $\boldsymbol{route}$.

An important part of the protocol is the way it updates the path vector. Since a find message from v may be forwarded to several destinations before the object is moved to v, the path that the find message travels to the last destination may not be the shortest path in the spanning tree T, since some nodes may be visited multiple times. Since there is only one path in a spanning tree between two nodes such that each node in the path is visited exactly once, the path vector is updated in the following way. When the find message from v is forwarded to a node w, it checks the last two elements in $\boldsymbol{r}(v)$, say $\boldsymbol{r}(v)[max-1]$ and $\boldsymbol{r}(v)[max]$. If $\boldsymbol{r}(v)[max-1] \neq \boldsymbol{r}(v)[max]$, then w is added to the path vector by setting $\boldsymbol{r}(v)[max+1] = w$. If $\boldsymbol{r}(v)[max-1] = \boldsymbol{r}(v)[max]$, then it checks $\boldsymbol{r}(v)[max-2]$. If $\boldsymbol{r}(v)[max-2] \neq w$, then $\boldsymbol{r}(v)[max]$ is removed and w is added to $\boldsymbol{r}_v$. If $\boldsymbol{r}(v)[max-2] = w$, then a loop forms in $\boldsymbol{r}(v)$ by $\{\boldsymbol{r}(v)[max-2], \boldsymbol{r}(v)[max-1], \boldsymbol{r}(v)[max], w\}$. In this case, both $\boldsymbol{r}(v)[max-1]$ and $\boldsymbol{r}(v)[max]$ are removed and w is added to $\boldsymbol{r}(v)$. The new updated path vector will not contain $\boldsymbol{r}(v)[max-1]$ or $\boldsymbol{r}(v)[max]$ any more. See Figure 2 for an example of the path vector updating process of the Relay protocol.

The pointers are updated when the object is moved. Suppose that node x moves the object to node v, node x sends a *move* message with the object to $move(x).\boldsymbol{route}[max]$. Meanwhile,

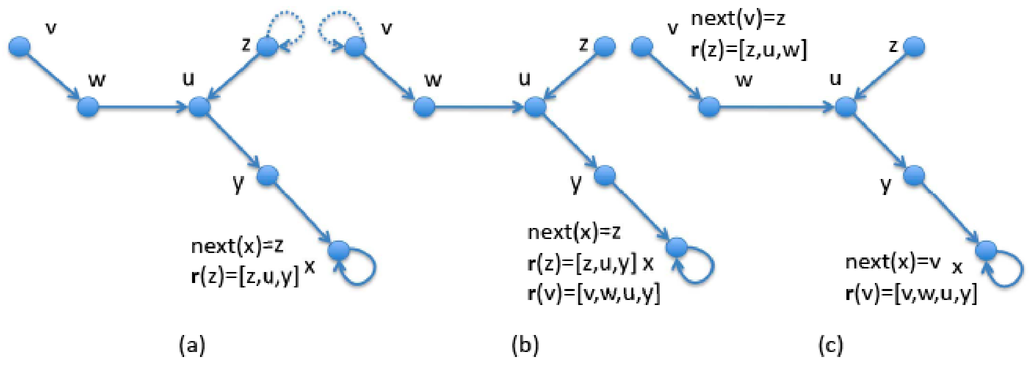

Fig. 2. Updating of the Path Vector of the Relay Protocol: $T_x \prec T_v \prec T_z$

node x sets $p(x)$ to $move(x).\boldsymbol{route}[max]$. Suppose a node u receives a move message from one of its neighbors. It updates $move(x).\boldsymbol{route}$ by removing $move(x).\boldsymbol{route}[max]$ and sends the object to the new $move(x).\boldsymbol{route}[max]$, setting $p(u) = move(x).\boldsymbol{route}[max]$. Finally, when the object arrives at v, $p(v)$ is set to v and all pointers are updated. Such operations guarantee that at any time, there exists only one sink in the network, and, from any node, following the direction of its pointer leads to the sink.

We now describe the protocol formally.

`Variables.` We use the following variables:

1. $p(v)$ is v's pointer pointing to a neighbor on the tree or to itself.
2. $\boldsymbol{r}(v)$ is a vector recording the path from v to the node it resides.
3. $\boldsymbol{route}$ is a vector to store the route that the message follows.
4. $next(v)$ stores v's successor in the queue.

`Messages.` There are three types of messages:

1. Node v sends a $publish$ message to publish the object.
2. Node v sends a $find(v)$ message to request the object.
3. Node v sends a $move(v)$ message with the object when the object is moved from v.

`Operations.` All operations are of the form: (event) followed by (actions). The following operations are performed on node v. Note that $\boldsymbol{r}(v)[max]$ or $\boldsymbol{route}[max]$ always point to the last element of the referred vector.

```
1. Event: Object $R_i$ is created by $v$. /* publish a new object */
   a Set $p(v) \leftarrow v$. /* set the pointer to itself */
   b Probe all neighbors of node $v$: $Neighbor(v)$ on $T$.
   c Send $publish(R_i)$ to $Neighbor(v)$.
2. Event: Receive $publish(R_i)$ from $u$. /* receive a publish message: set the pointer */
   a Set $p(v) \leftarrow u$.
   b Probe all neighbors of node $v$: $Neighbor(v)$ on $T$.
   c Forward $publish(R_i)$ to $\{Neighbor(v) \backslash \{u\}\}$.
3. Event: Transaction $T_v$ requests the object $R_i$. /* invoke a transaction request */
   a Set $find(v).\boldsymbol{route} \leftarrow null$ /* route vector reserved for future use */
   b Set $find(v).\boldsymbol{r}(v)[1] \leftarrow v$. /* initialize the path vector */
   c Send $find(v)$ to $p(v)$.
4. Event: Receive $find(w)$ message from $u$.
   a If $p(v) = v$ then /* $v$ holds the object */
      i If $R_i$ is idle then /* move the object */
         A Set $move(v).\boldsymbol{route} \leftarrow find(w).\boldsymbol{r}(w)$. /* set the route vector */
```

```
        B Send $move(v)$ and object $R_i$ to $move(v).\boldsymbol{route}[max]$. /* $move(v).\boldsymbol{route}[max] = u$ */
        C Set $p(v) \leftarrow move(v).\boldsymbol{route}[max]$. /* update the pointer */
      ii Else if $T_w \prec T_v$ then /* the local contention manager compares priorities */
        A Abort $T_v$. /* $T_v$ is aborted by $T_w$ */
        B Move the object following actions in 4aiA — 4aiC.
        C Set $find(v).\boldsymbol{route} \leftarrow find(w).\boldsymbol{r}(w)$. /* $T_v$ is restarted immediately and initializes the
          find message */
        D Set $find(v).\boldsymbol{r}(v)[1] \leftarrow v$. /* initialize the path vector */
        E Send $find(v)$ to $find(v).\boldsymbol{route}[max]$. /* send a find message to $w$ following the route
          vector */
      iii Else if $next(v) = null$ then /* $T_v \prec T_w$ */
        A Set $next(v) \leftarrow w$ /* $w$ is queued after $v$ */
        B Set $next(v).\boldsymbol{route} \leftarrow find(w).\boldsymbol{r}(w)$. /* save the route vector */
      iv Else if $T_w \prec T_{next(v)}$ then
        A Set $find(next(v)).\boldsymbol{route} \leftarrow find(w).\boldsymbol{r}(w)$.
        B Forward $find(next(v))$ to $find(next(v)).\boldsymbol{route}[max]$. /* forward $find(next(v))$ to $w$
          following the route vector */
        C Set $next(v) \leftarrow w$ /* $w$ is queued after $v$ */
        D Set $next(v).\boldsymbol{route} \leftarrow find(w).\boldsymbol{r}(w)$. /* save the route vector */
      v Else, /* $T_{next(v)} \prec T_w$ */
        A Set $find(w).\boldsymbol{route} \leftarrow find(next(v)).\boldsymbol{route}$. /* set the route vector to forward $find(w)$
          to node $next(v)$ */
        B Forward $find(w)$ to $find(w).\boldsymbol{route}[max]$.
    b Else if $find(w).\boldsymbol{route} = null$ then /* forward the message to the tail of the queue following
      pointers */
      i Set $find(v).\boldsymbol{r}(v)[max + 1] \leftarrow v$. /* node $v$ is added to the path vector */
      ii Forward $find(w)$ to $p(v)$.
    c Else if $|find(w).\boldsymbol{route}| = 1$ then follow actions in 4aiii — 4av. /* $v$ is the destination to
      forward $find(w)$ following the route vector and $T_v \prec T_w$ */
    d Else if $find(w).\boldsymbol{r}(w)[max-1] \neq find(w).\boldsymbol{r}(w)[max]$ /* the find message comes from the tail
      of the queue */
      i Set $find(w).\boldsymbol{r}(w)[max + 1] \leftarrow v$. /* node $v$ is added to the path vector */
      ii Remove $find(w).\boldsymbol{route}[max]$. /* $find(w).\boldsymbol{route}[max] = v$ before removing */
      iii Forward $find(w)$ to $find(w).\boldsymbol{route}[max]$.
    e Else if $find(w).\boldsymbol{r}(w)[max - 2] \neq v$ /* $find(w).\boldsymbol{r}(w)[max - 1]$ is the intersection node of two
      paths $find(w).\boldsymbol{route}$ and $find(w).\boldsymbol{r}(w)$ */
      i Remove $find(w).\boldsymbol{r}(w)[max]$. /* $find(w).\boldsymbol{r}(u)[max] = find(w).\boldsymbol{r}(w)[max - 1]$ before re-
        moving it */
      ii Set $find(w).\boldsymbol{r}(w)[max + 1] \leftarrow v$. /* node $v$ is added to the path vector */
      iii Remove $find(w).\boldsymbol{route}[max]$. /* $find(w).\boldsymbol{route}[max] = v$ before removing */
      iv Forward $find(w)$ to $find(w).\boldsymbol{route}[max]$.
    f Else, /* a loop forms */
      i Remove $find(w).\boldsymbol{r}(w)[max-1]$ and $find(w).\boldsymbol{r}(w)[max]$. /* $find(w).\boldsymbol{r}(w)[max] = find(w).\boldsymbol{r}(w)[max-$
        $1]$ before removing */
      ii Set $find(w).\boldsymbol{r}(w)[max + 1] \leftarrow v$. /* node $v$ is added to the path vector */
      iii Remove $find(w).\boldsymbol{route}[max]$. /* $find(w).\boldsymbol{route}[max] = v$ before removing */
      iv Forward $find(w)$ to $find(w).\boldsymbol{route}[max]$.
5. Event: Receive $move(w)$ message and object $R_i$ from $u$
  a If $|move(w).\boldsymbol{route}| = 1$ then /* $v$ is the destination */
    i Set $p(v) \leftarrow v$. /* update the pointer */
  b Else, /* forward the move message and the object */
```

```
        i Remove $move(w).\boldsymbol{route}[max]$. /* $move(w).\boldsymbol{route}[max] = v$ before removing */
        ii Forward $move(w)$ and object $R_i$ to $move(w).\boldsymbol{route}[max]$.
        iii Set $p(v) \leftarrow move(w).\boldsymbol{route}[max]$. /* update the pointer */
6. Event: Transaction $T_v$ on $v$ commits and $R_i(T_v) \neq 0$
    a if $next(v) \neq null$ then /* move the object */
        i Set $move(v).\boldsymbol{route} \leftarrow next(v).\boldsymbol{route}$. /* set the route vector */
        ii Send $move(v)$ and object $R_i$ to $move(v).\boldsymbol{route}[max]$.
        iii Set $p(v) \leftarrow move(v).\boldsymbol{route}[max]$.
    b Else, /* wait for transaction requests */
```

Protocol Analysis. The correctness of the protocol can be proved from the protocol description. The pointers are only "flipped" when the object is moved, which guarantees that at any time there is only one sink in the network and following the pointer from any node leads to the sink. The key to proving the correctness of the protocol is that find and move messages are forwarded along the correct path on T. As explained in the protocol description, we use path vectors and route vectors to record paths. As long as they are correctly updated, a find or a move message can be forwarded along the unique path on T to its destination.

We now focus on the performance of the Relay protocol, which we measure through its competitive ratio. We can directly derive the following relationships from the protocol description:

$$\text{max-Str}^{\delta}_{Relay} = \text{max-Str}^{\zeta}_{Relay} = \text{max-str}(T), \tag{1}$$

since the object is located and moved via a unique path on T. To illustrate the advantage of the Relay protocol on reducing the number of abortions, we have the following theorem:

Theorem 5 $\max_{v_{T_j} \in V_T^{R_i}} \lambda_{Relay}(j) \leq N_i, \quad \Lambda_{Relay} \leq 2N_i - 1$

Proof. The first part of the theorem can be proved following the same way as that of Theorem 4. To prove the second part, we first order the set of transactions in the priority order such that $\{T_1 \prec T_2 \prec \ldots \prec T_{N_i}\}$. Suppose a transaction T_v is aborted by another transaction. In this case, T_v is restarted immediately and a find message is sent to its predecessor on the queue. Finally, a node w keeps a variable $next(w) = v$. In other words, for each time that a node is aborted, a successor link *next* between two nodes is established. Now, assume the next abortion occurs and a successor link $next(w') = v'$ is established. If $T_w \prec \{T_{w'}\, or\ T_{v'}\} \prec T_v$, we say that these two links are *joint*; otherwise we say that they are *disjoint*. We can prove that, if $next(w)$ and $next(w')$ are joint, at least one transaction in $\{T_w, \ldots, T_v\}$ has committed. Hence, there are only two outcomes for an abortion: at least one transaction's commit or a successor link disjoint to other successor links established. Hence, we just need at most $N_i - 1$ abortions to let N_i transactions commit or establish a chain of links among all transactions (since they are disjoint). For the latter case, no more abortion will occur since the object is moved following that chain. The theorem follows.

From Equation 1 and Theorem 5, we have the following corollary:

Corollary 2 $CR_i(Relay) \leq \max\{N_i, 2\textit{max-str}(T) \cdot \textit{Diam}\}$

Thus, the Relay protocol successfully improves the competitive ratio by reducing the number of total transactions' abortions.

5 Concluding Remarks

We conclude that the worst-case performance of a cache-coherence protocol is determined by its worst-case number of abortions, maximum locating stretch, and maximum moving stretch. Compared with the traditional distributed queuing problem, the design of a cache-coherence protocol must take into account the contention between two transactions because transaction abortions increase the length of the queue. Motivated by a distributed queuing protocol with excellent performance, the arrow protocol, we show that its worst-case number of total abortions is $O(N_i^2)$ for N_i transactions requesting the same object. Based on this protocol, we design the Relay protocol which reduces the worst-case number of total abortions to $O(N_i)$. Meanwhile, the Relay protocol inherits the advantage of the arrow protocol—i.e., the maximum locating stretch and moving stretch are exactly the maximum stretch of the underlying spanning tree. As a result, the Relay protocol yields a better competitive ratio.

We show that the worst-case performance of the Relay protocol is determined by the maximum stretch of the underlying spanning tree. Hence, choosing a good spanning tree for the protocol is an important problem. The problem of finding a spanning tree that minimizes max-str(T) is referred to as the *Minimum Max-Stretch spanning Tree* (or MMST) problem and is known to be NP-hard [3]. Emek and Peleg [6] presents an $O(\log n)$-approximation algorithm for this problem for unweighted graphs.

The Relay protocol is designed to support multiple objects. Since the protocol is totally distributed (all nodes are of the same importance in the protocol), it avoids significantly overloading some nodes in the network. There are several directions for future work. Fault-tolerance is an important issue. Similar to [17], a self-stabilizing algorithm can also be designed for the Relay protocol. We assume a bounded communication cost between nodes and evaluate the worst-case performance in this paper. Studying the average-case performance of cache-coherence protocols in a network with stochastic behavior of message loss and delay will be an interesting future direction.

References

1. Hagit Attiya, Leah Epstein, Hadas Shachnai, Tami Tamir: Transactional contention management as a non-clairvoyant scheduling problem. In PODC '06, 308–315
2. R. L. Boccino, V. S. Adve, B. L. Chamberlain: Software Transactional Memory for Large Scale Clusters. In PPoPP'08, 247–258
3. Cai, Leizhen, Corneil, Derek G.: Tree Spanners. SIAM J. Discret. Math., 8(3): 359–387 (1995)
4. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In ASPLOS'06, 336–346
5. Demmer, Michael J., Herlihy, Maurice: The Arrow Distributed Directory Protocol. In DISC '98, 119–133
6. Emek, Yuval, Peleg, David: Approximating Minimum Max-Stretch spanning Trees on unweighted graphs. In SODA '04, 261–270
7. Rachid Guerraoui, Maurice Herlihy, Bastian Pochon: Toward a theory of transactional contention managers. In PODC '05, 258–264
8. Lance Hammond, Vicky Wong, Mike Chen, Ben Hertzberg, Brian D. Carlstrom, John D. Davis, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, Kunle Olukotun: Transactional Memory Coherence and Consistency. In ISCA'04, 102–113
9. Herlihy, Maurice, Tirthapura, Srikanta, Wattenhofer, Roger: Competitive concurrent distributed queuing. In PODC '01, 127–133
10. Maurice Herlihy, Victor Luchangco, Mark Moir: Obstruction-free Synchronization: Double-ended Queues as an Example. In ICDCS'03, 522–529
11. Herlihy, Maurice, Luchangco, Victor, Moir, Mark: A flexible framework for implementing software transactional memory. In OOPSLA '06, 253–262
12. Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, III: Software transactional memory for dynamic-sized data structures. In PODC '03, 92–101
13. Maurice Herlihy, Ye Sun: Distributed Transactional Memory for Metric-space Networks. Distributed Computing, 20(3): 195–208 (2007)
14. K. Manassiev, M. Mihailescu, C. Amza: Exploiting Distributed Version Concurrency in a Transactional Memory Cluster. In PPoPP'06, 198–208
15. Raymond, Kerry: A tree-based algorithm for distributed mutual exclusion. ACM Trans. Comput. Syst., 7(1): 61–77 (1989)
16. N. Shavit, D. Touitou: Software Transactional Memory. In PODC '95, 204–213
17. Srikanta Tirthapura, Maurice Herlihy: Self-Stabilizing Distributed Queuing. IEEE Transactions on Parallel and Distributed Systems, 17(7): 646–655 (2006)