Distributed Computing Column 54
Transactional Memory: Models and Algorithms

Jennifer L. Welch
Department of Computer Science and Engineering
Texas A&M University, College Station, TX 77843-3112, USA
welch@cse.tamu.edu

This issue's column consists of a review article by Gokarna Sharma and Costas Busch on models and algorithms for transactional memory (TM), with particular emphasis on scheduling. With the ever-growing popularity of TM, this is a timely topic. The authors cover three main models. First, work on transaction scheduling algorithms for tightly-coupled systems are surveyed. Second, distributed networks systems are considered; the new aspect is how to find the shared objects efficiently and provide consistency of the objects after transactions terminate. Third, related results for non-uniform memory access systems are surveyed, with emphasis on how to provide consistency in a load-balanced way. The article closes with a discussion of future directions.

You might want to check out previous coverage of TM in this Column, dating back to 2008. In March of that year, the entire column was devoted to the topic, in the context of multicore systems. The first four Workshops on the Theory of Transactional Memory (WTTM) are reviewed in the December 2009, December 2010, March 2012, and December 2012 issues. The latter issue also covers the awarding of the 2012 Dijkstra Prize for the work by Herlihy, Moss, Shavit and Touitou on TM.

Many thanks to Gokarna and Costas for their contribution!

Call for contributions: I welcome suggestions for material to include in this column, including news, reviews, open problems, tutorials and surveys, either exposing the community to new and interesting topics, or providing new insight on well-studied topics by organizing them in new ways.

Transactional Memory: Models and Algorithms

Gokarna Sharma
Louisiana State University
Baton Rouge, LA, USA
gokarna@csc.lsu.edu

Costas Busch
Louisiana State University
Baton Rouge, LA, USA
busch@csc.lsu.edu

1 Introduction

Modern multicore architectures enable the concurrent execution of an unprecedented number of threads. This gives rise to the opportunity for extreme performance and the complex challenge of synchronization. Conventional lock-based synchronization has several drawbacks which limits the parallelism offered by multicore architectures. Coarse-grained locks do not scale. Fine-grained locks are difficult to program correctly because locks are generally not composable. Transactional memory (TM) [45, 81] provides an alternative synchronization mechanism that is non-blocking, composable, and easier to write than lock-based code [64]. TM-based synchronization has recently been included in IBM's Blue Gene/Q [39, 84] and Intel's Haswell processors [20]. TM is predicted to be widely used in future processors, possibly even GPUs [32, 86]. In the research community, several TM implementations (hardware, software, and hybrid) have been proposed and studied, e.g., [16, 24, 26, 30, 31, 43, 44, 60]. The TM book by Harris *et al.* [40] provides an excellent overview of the design and implementation of TM systems up to early spring 2010.

TM operates in a way similar to database transactions, and aggregates a sequence of shared resource accesses (reads or writes) that should be executed atomically (by a single thread) in a fundamental module called *transaction*. A transaction is a piece of code that executes a series of reads and writes to shared memory. These reads and writes logically occur as a transaction at a single instance in time; intermediate states are not visible to other (successful) transactions. TM increases parallelism as no threads need to wait for access to a shared resource and different threads can simultaneously modify disjoint parts of a data structure that would normally be protected under the same lock. A transaction ends either by committing, in which case all of the updates take effect, or by aborting, in which case no update is effective. Each program thread generates a sequence of transactions. Transactions of the same thread execute sequentially by following the program execution flow. However, transactions of different threads may *conflict* when they attempt to access the same shared memory resources. The advantage of TM is that if there are no conflicts

between transactions then the threads continue execution without delays that would have been caused unnecessarily if locking mechanisms were used. Thus, TM can be viewed as an *optimistic* synchronization mechanism [44].

Transaction conflicts are detected using conflict detection mechanisms [83]. If a transaction T discovers that it conflicts with another transaction T' (because they access a shared resource), then T has the following three choices: (i) it can give T' a chance to finish and commit by T aborting itself; (ii) it can proceed and commit by forcing T' to abort; the aborted transaction T' then retries immediately again until it eventually commits; or (iii) it can wait (or back off) for a short period of time and retry the conflicting access again. In other words, a *conflict handling mechanism* decides which transactions should continue and which transactions should abort and try again until they eventually commit. This decision process leads us to the *transaction scheduling* problem. Typically, this transaction scheduling problem is *online* in the sense that transaction conflicts are not known a priori.

To solve the transaction scheduling problem efficiently, each transaction consults with the *contention manager* module of the TM system for which choices to make. DSTM [44] is the first software TM (STM) implementation that uses a contention manager as an independent module to resolve conflicts between transactions and ensure *progress* – some useful work in done in each time step of the execution. A major challenge in guaranteeing progress through contention managers is to devise a scheduling algorithm which ensures that all transactions commit in the shortest possible time. Given a set of transactions, a central optimization metric in the literature, e.g. [6, 8, 34, 36, 68, 70], is to minimize the *makespan* which is defined as the duration from the start of the execution schedule, i.e., the time when the first transaction is issued, until all transactions commit. In a dynamic scenario where transactions are issued continuously, the makespan translates to the *throughput*, measured as the number of committed transactions per unit of time.

Since it is projected that a processor chip will have a large number of cores, it is important to design TM systems which scale gracefully with the variability of the system sizes and complexities. To achieve this goal, it is desirable to devise scheduling algorithms which have both good theoretical asymptotic behavior and also exhibit good practical performance. Provable formal properties help to better understand worst-case and average-case scenarios and determine the scalability potential of the system. It is also equally important to design scheduling algorithms with good performance for various reasonable practical execution scenarios. We proceed by models and metrics that capture the performance evaluation of scalable TM systems.

TM Models: TM has been studied mainly in three system models that we describe below. The main distinction between these models is the variation of the intra-core communication cost in accessing shared memory locations. The communication cost can be symmetric, asymmetric, or partially symmetric. These types of communication cost models are appropriate to cover tightly-coupled systems, larger scale distributed systems, and their combinations.

- *Tightly-coupled systems (symmetric communication):* This model represents the most common scenario where multiple cores reside in the same chip. The shared memory access mechanism is implemented through a multi-level cache coherence algorithm (see left of Fig. 1). The cost of accessing shared resources is symmetric (uniform) across different processors.

- *Distributed networked systems (asymmetric communication):* This model represents the scenario of completely decentralized distributed shared memory where processors are connected

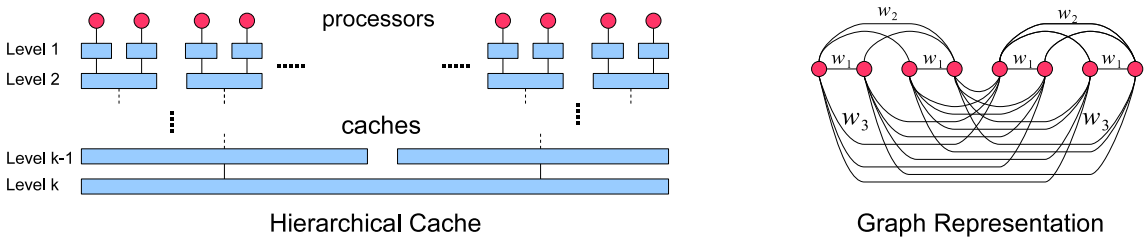


Figure 1: Left: a hierarchical multilevel cache; Right: a processor communication graph.

through a large-scale message passing system. Typically, the network is represented as a graph where the processors are nodes and links are weighted edges (see right of Fig. 1). The distance between processor nodes plays a significant role in the communication cost which is typically asymmetric among different network nodes. Note that this model is general enough to also include the uniform case of the tightly-coupled systems. This model is also suitable to model transaction scheduling scenarios that arise in cloud computing systems and heterogenous architectures.

- *Non-uniform memory access systems (NUMA, partially symmetric communication):* This model is a bridge between tightly-coupled systems and distributed systems described above. It represents a set of multiprocessors communicating through a small scale interconnection network. The interconnection network has a regular structure such as a grid (mesh), hypercube, butterfly, etc.; see Fig. 6 for a 3-dimensional multicore processor grid. Such network topologies have been extensively studied in the literature [55] and have predictable performance guarantees in terms of communication efficiency. There are two levels of communication: local (symmetric) communication within cores of the same processor and larger-scale (asymmetric) communication between different processors in different areas of the network topology. High performance multiprocessors are typically organized with such an architecture, e.g. [1, 19, 48], and their efficiency is vital for scientific applications.

Performance Evaluation Metrics: We focus on the following metrics that are used for evaluating the formal and experimental performance of transaction scheduling algorithms in the aforementioned TM models. These metrics include time, communication cost, and load.

- *Makespan:* It measures the commit duration for the last transaction in a given input set of transactions. This is a typical performance metric in transaction scheduling. In a dynamic setting, the makespan translates to throughput. A primary goal for a transaction scheduling algorithm (i.e., a contention manager) is to minimize the makespan.

- *Communication cost:* It concerns distributed network TM models, and measures the number of messages sent on network links for scheduling the transactions. This metric relates to the total utilization of the distributed system resources, and it translates to the time and energy performance of the distributed transaction scheduling.

- *Load balancing:* This is particularly relevant for distributed and NUMA models, and it concerns the load of the network edges and nodes that is involved in fulfilling requests for the

shared objects. Load balancing is important when energy and resource utilization needs to be minimized.

We now outline what topics are covered and how this column is organized. Different models and algorithmic topics are covered depending on the TM models (tightly-coupled, distributed networked, or NUMA) as described below.

Overview and Organization: We survey in this column the work that has been done in the literature in the design, development, and analysis of transaction scheduling algorithms in typical tightly-coupled systems and in newly-considered distributed and NUMA systems. In particular, we consider the previous upper bound, lower bound, impossibility, and experimental results, and provide high level details of our results. We discuss results for tightly-coupled systems in Section 3. We then discuss results for distributed networked systems in Section 4. In distributed networked systems, transactions are scheduled and conflicts are resolved in each network node using a globally-consistent scheduling algorithm similar to the one that is designed for TM implementation in tightly-coupled systems. Therefore, the focus of transaction scheduling work for distributed networked systems in the literature is on how to find the shared objects needed by a transaction efficiently from the remote nodes and provide consistency of the objects after transactions commit and abort. We then talk about transaction scheduling problem in NUMA systems in Section 5. For NUMA systems, we talk about providing consistency to the objects in a load balanced way. Consistency algorithms are important because the makespan of the transaction scheduling problem in distributed networked and NUMA systems relies on how efficiently objects are moved to the nodes that need them (or how efficiently transactions are moved to the nodes where the required object reside). We then outline future research directions and conclude in Section 6. Note that there has been a significant volume of work in the literature related to TM and we intend to cover only the work on scheduling algorithms.

2 Transaction Scheduling Problem

Consider a set of $M \geq 1$ transactions $\mathcal{T} = \{T_1, T_2, \ldots, T_M\}$, one transaction each in M different threads $\mathcal{P} = \{P_1, \ldots, P_M\}$, and a set of $s \geq 1$ shared resources $\mathcal{R} = \{R_1, R_2, \ldots, R_s\}$. Since there is only one transaction in each thread, we call this problem the *one-shot* transaction scheduling problem. Each transaction is a sequence of actions (or operations) that is either a read or a write to some shared resource $R_i, 1 \leq i \leq s$. The sequence of actions in a transaction must be atomic: all actions of a transaction are guaranteed to either completely occur or have no effects at all. A transaction after it is issued and starts execution, it completes either with a *commit* or an *abort*. A transaction is *pending* after its first action until its last action; it takes no further actions after a commit or an abort. A pending transaction can restart multiple times until it eventually commits. The first action of a transaction must be a read or a write and its last action is either a commit or an abort.

Concurrent write-write actions or read-write actions to same shared objects by two or more transactions cause conflicts between transactions. If a transaction conflicts then it either aborts or it may commit and force all other conflicting transactions to abort. In *eager conflict management* TM systems, conflicts are resolved as soon as they are detected, whereas in *lazy conflict management* TM systems, conflict detection and resolution process is deferred to the end of a transaction. An

execution schedule is called *greedy* if a transaction aborts due to conflicts it then immediately restarts its execution and attempts to commit again.

Each transaction $T_i \in \mathcal{T}$ has execution time duration $\tau_i > 0$. The execution time is the total number of discrete time steps that the transaction requires to commit uninterrupted from the moment it starts. Let $\tau_{\max} := \max_i \tau_i$ be the execution time of the longest transaction, and $\tau_{\min} := \min_i \tau_i$ be the execution time of the shortest transaction. A resource can be read in parallel by arbitrarily many transactions. A transaction is called *read-only* if it only reads the shared resources, otherwise it is a *writing* transaction.

The goal is to come up with a scheduling algorithm $\mathcal{A}$ for the transactions in the set $\mathcal{T}$ such that $makespan(\mathcal{A}, \mathcal{T})$ is minimized. Here $makespan(\mathcal{A}, \mathcal{T})$ denotes the completion time of all the transactions under $\mathcal{A}$, that is the latest time at which any transaction in $\mathcal{T}$ commits. The makespan of $\mathcal{A}$ for $\mathcal{T}$ can be compared to the makespan $makespan(\textsc{opt}, \mathcal{T})$ of the optimal offline scheduling algorithm OPT for $\mathcal{T}$ to obtain the *competitive ratio* of $\mathcal{A}$ for $\mathcal{T}$. The competitive ratio of $\mathcal{A}$ for any workload $\mathcal{T}$ is $\max_{\mathcal{T}} \frac{makespan(\mathcal{A},\mathcal{T})}{makespan(\textsc{opt},\mathcal{T})}$ which is the maximum over all $\mathcal{T}$. Note that the makespan and the competitive ratio primarily depend on the *workload* − the set of transactions, along with their arrival times, execution time duration, and resources they read and modify [8]. Therefore, the one-shot model described above is general enough to extend to different variations introducing some restrictions; we will discuss some of them in Section 3.

Let $\mathcal{R}(T_i)$ denote the set of resources used by a transaction T_i. We have that $\mathcal{R}(T_i) = \mathcal{R}_w(T_i) \cup \mathcal{R}_r(T_i)$, where $\mathcal{R}_w(T_i)$ are the resources written by T_i and $\mathcal{R}_r(T_i)$ are the resources read by T_i. Two transactions T_i and T_j *conflict* if at least one of them writes on a common resource, i.e., $\exists R$ such that $R \in (\mathcal{R}_w(T_i) \cap \mathcal{R}(T_j)) \cup (\mathcal{R}(T_i) \cap \mathcal{R}_w(T_i))$. We can now define the *conflict graph* for a set of transactions which models any transaction scheduling problem. In the conflict graph, each node represents a transaction and each edge represents a conflict in accessing resources between the adjacent transactions. Formally, for any set of transactions $\mathcal{T}$, the conflict graph $\mathcal{G}(\mathcal{T}) = (V, E)$ has as nodes the transactions, $V = \mathcal{T}$, and $(T_i, T_j) \in E$ for any two transactions T_i, T_j that conflict. We now provide the definition of pending commit property.

Definition 1 (pending commit property [36])**.** *A transaction scheduling algorithm obeys the* pending commit property *if, whenever there are pending transactions, some running transaction T will execute uninterrupted until it commits.*

The above description of the transaction scheduling problem applies to tightly-coupled systems. In distributed networked systems, transactions in $\mathcal{T}$ are in the network nodes or processors (one transaction each in M different processors). It is assumed that there is a shared memory which is split among the processors. Each processor has its own cache, where copies of *objects* (individual entries at the shared resources) reside. A transaction may consist of multiple shared objects. When a transaction running at a processor node issues a read or write operation for a shared memory location, the data object at that location is loaded into the processor-local cache. Some of the shared objects needed by a transaction may be in the shared memory of the node which is executing that transaction and some of the shared objects may be in the shared memory of other nodes. To be able to execute the transaction, either the shared objects in other nodes need to be moved to the node where the transaction is currently executing or the transaction needs to be moved to the node where the shared object needed by that transaction currently resides. This decision depends on the implementation technique used. In a *data-flow* implementation [46], transactions are immobile and objects are moved to nodes that need them. In a *control-flow* implementation

[65], objects are immobile and transactions are moved to the nodes when objects reside. In a hybrid implementation, what to move, transactions or objects, is determined using some criteria minimizing some performance metric.

In NUMA systems, the cost of accessing shared resources is asymmetric across different processors (symmetric communication within the cores of the same processor and asymmetric communication between different processors), in contrast to tightly-coupled systems where the cost is assumed to be symmetric.

3 Tightly-coupled Systems

Tightly-coupled systems represent the typical scenario of a multicore chip with multilevel cache organization, where the lower level caches are distinct to each processor, while the highest level cache is common to all the cores in the chip (see left of Fig. 1). Communication costs between the processors are symmetric. We start with describing the related work in the literature, and then present high level details of our work on obtaining new efficient scheduling algorithms and their experimental evaluations.

Most of the algorithms proposed in the literature [3, 27, 35, 44, 61, 68, 69, 87] for the transaction scheduling problem (Section 2) have been assessed only experimentally by using specific benchmarks. Guerraoui *et al.* [36] were the first to develop a scheduling algorithm which exhibits non-trivial provable worst-case guarantees along with good practical performance. Their Greedy scheduling algorithm decides in favor of old transactions using timestamps and achieves $\mathcal{O}(s^2)$ competitive ratio in comparison to the *optimal* off-line scheduling algorithm for n concurrent transactions that share s resources, and at the same time has good empirical performance. They experimented with Greedy in DSTM [44] using the list and red-black tree benchmarks and concluded that it achieves performance comparable to other scheduling algorithms like Polka [68] and Aggressive [69] along with its provable worst-case guarantees. Later, Guerraoui *et al.* [34] studied the impact of transaction failures on transaction scheduling. They presented the algorithm FT-Greedy and proved an $\mathcal{O}(k \cdot s^2)$ competitive ratio when some running transaction may *fail* at most k times and then eventually commits.

Algorithm	**Model**	**Competitive ratio**	**Deterministic/ Randomized**	**Assumptions**
Serializer [27], ATS [87]	One-shot	$\Theta(\min\{s, M\})$ [6, 28]	Deterministic	-
Polka [68], SizeMatters [61]	One-shot	$\Omega(\min\{s, M\})$ [6, 70]	Deterministic	-
Restart [28], SoA [3]	One-shot	$\Theta(\min\{s, M\})$ [6, 8]	Deterministic	-
Greedy [36]	One-shot	$\mathcal{O}(s^2)$ [36]	Deterministic	Unit length transactions
FTGreedy [34]	One-shot	$\mathcal{O}(k \cdot s^2)$ [34]	Deterministic	Transactions can fail
Greedy [36]	One-shot	$\Theta(s)$ [6]	Deterministic	-
FTGreedy [34]	One-shot	$\Theta(k \cdot s)$ [6]	Deterministic	Transactions can fail
Phases [6]	One-shot	$\mathcal{O}(\max\{s, k \log k\})$[6]	Randomized	Unit length transactions
RandomizedRounds [70]	One-shot	$\mathcal{O}(C \cdot \log M)$ [70]	Randomized	Equal length transactions
CommitRounds [70]	One-shot	$\mathcal{O}(\min\{s, M\})$ [6, 70]	Deterministic	Equal length transactions
Bimodal [8]	One-shot	$\Theta(s)$ [8]	Deterministic	Bimodal workloads
Clairvoyant [73]	One-shot	$\mathcal{O}(\sqrt{s})$ [73]	Deterministic	Balanced workloads
Non-Clairvoyant [73]	One-shot	$\mathcal{O}(\sqrt{s} \cdot \log M)$ [73]	Randomized	Balanced workloads
Clairvoyant [73]	One-shot	$\mathcal{O}(k \cdot \sqrt{s})$ [73]	Deterministic	Transactions can fail
Non-Clairvoyant [73]	One-shot	$\mathcal{O}(k \cdot \sqrt{s} \cdot \log M)$ [73]	Randomized	Transactions can fail
Offline-Greedy [75]	Window	$\mathcal{O}(s + \log(MN))$ [75]	Randomized	Equal length transactions; conflict graph is known
Online-Greedy [75]	Window	$\mathcal{O}(s \cdot \log(MN) + \log^2(MN))$ [75]	Randomized	Equal length transactions; conflict graph is not known
Offline-Greedy [75]	Window	$\mathcal{O}(k \cdot (s + \log(MN)))$ [75]	Randomized	Transactions can fail
Online-Greedy [75]	Window	$\mathcal{O}(k \cdot (s \cdot \log(MN) + \log^2(MN)))$ [75]	Randomized	Transactions can fail

Table 1: Comparison of transaction scheduling algorithms, where C denotes the number of conflicts and it can be as much as the number of shared resources s, k denotes the number of times a transaction can fail, M denotes the number of different threads (or cores), and N denotes the number of transactions in each thread. The assumptions of the algorithms for the failure-free case are applied also to their versions for transaction failures.

Several other algorithms have also been proposed for the efficient transaction scheduling and the performance of some of them has been analyzed formally [6, 8, 28, 70]. The detailed comparison of the results and their properties are listed in Table 1. Attiya *et al.* [6] improved the competitive ratio of Greedy to $\mathcal{O}(s)$ and of FTGreedy to $\mathcal{O}(k \cdot s)$, and proved a matching lower bound of $\Omega(s)$ ($\Omega(k \cdot s)$ when transactions may fail) for any deterministic *work-conserving* algorithm which schedules as many transactions as possible (by choosing a maximal independent set of transactions at each time step). They also gave a randomized scheduling algorithm Phases that achieves $\mathcal{O}(\max\{s, k \log k\})$ competitive ratio for the special case of unit length transactions in which a transaction may fail

at most k times before it eventually commits. Schneider and Wattenhofer [70] proposed an algorithm, called RandomizedRounds, which produces a $\mathcal{O}(C \cdot \log M)$-competitive schedule with high probability, for the transaction scheduling problem with C conflicts (assuming unit delays for transactions). They also gave a deterministic algorithm CommitRounds with $\mathcal{O}(\min\{s, M\})$ competitive ratio. Later, Attiya *et al.* [8] proposed a $\Theta(s)$-competitive algorithm for the one-shot scheduling problem in bimodal workloads. A workload is called *bimodal* if it contains only early-write and read-only transactions; a transaction is called *early-write* if the time from its first write access until its completion is at least half of its duration [8]. Hasenfratz *et al.* [41] studied different schedulers to adapt the load in STM systems based on contention. The model in [6, 8] is *non-clairvoyant* in the sense that it requires no prior knowledge about the transactions while they are executed. The model used in [70] is based on the *degree of a transaction* (i.e., neighborhood size) in the conflict graph $\mathcal{G}(\mathcal{T})$ of transactions.

Schneider and Wattenhofer [70] proved that the scheduling algorithms Polka [68] and SizeMatters [61] are $\Omega(M)$-competitive. Attiya and Milani [8] showed that Steal-on-Abort (SoA) [3] and Serializer [27] algorithms are $\Omega(M)$-competitive. Moreover, Dragojević *et al.* [28] proved that Serializer [27] and adaptive transaction scheduling (ATS) [87] algorithms are $\mathcal{O}(M)$-competitive. Attiya *et al.* [6] proved that every deterministic scheduling algorithm is $\Omega(s)$-competitive. Combining all these results, we obtain the bounds listed in Table 1 for these algorithms.

We provided novel techniques and bounds in [71–73, 75, 80] for the formal performance analysis of transaction scheduling algorithms with respect to the makespan in tightly-coupled systems. At the same time we have evaluated the performance of the scheduling algorithms experimentally for other performance metrics, such as performance throughout, as well. We provide two main scheduling models, the *balanced workload* model and the *window-based execution* model. Both of these models aim at improving the previous formal bounds relating the makespan performance of TM to the number of resources. In the balanced workload model, we give sub-linear bounds with respect to the number of resources for a simple restricted version of the one-shot scheduling problem of Section 2. In the window-based model, we actually give an alternative bound based on the metric of conflict number C which may be smaller than the number of resources for an extended version of the one-shot problem of Section 2.

3.1 Balanced Workloads

We considered the transaction scheduling problem in the context of balanced workloads in [71, 73]. A workload is called balanced if the number of write operations a transaction performs is a constant fraction of the total number of read and write operations of that transaction. We considered the one-shot model of $M \geq 1$ transactions (Section 2) with two additional fairly minimalistic assumptions, which we call the *balanced transaction scheduling* problem (the bimodal scenario of Attiya and Milani [8] also uses some restrictions on when a writing transaction can actually write). Specially, the two assumptions are that each transaction T_i should know (i) its execution time duration τ_i, and (ii) the number of shared resources it accesses. Let $\mathcal{R}(T_i)$ denote the set of resources used by a transaction T_i. We can write $\mathcal{R}(T_i) = \mathcal{R}_w(T_i) \cup \mathcal{R}_r(T_i)$, where $\mathcal{R}_w(T_i)$ are the resources which are to be written by T_i and $R_r(T_i)$ are the resources to be read by T_i. The *balancing ratio* of T_i is given as $\beta(T_i) = \frac{|\mathcal{R}_w(T_i)|}{|\mathcal{R}(T_i)|}$. For a read-only transaction $\beta(T_i) = 0$; for a writing transaction $\frac{1}{s} \leq \beta(T_i) \leq 1$, where s is the number of shared resources. The global balancing ratio can be defined as $\beta := \min_{(T_i \in \mathcal{T}) \wedge (|\mathcal{R}_w(T_i)| > 0)} \beta(T_i)$, which is the minimum of all $\beta(T_i)$. A workload (set of

Algorithm 1: A generic balanced transaction scheduling algorithm

Input: A set $\mathcal{T}$ of $M \geq 1$ transactions with global balancing ratio β;
Output: A greedy execution schedule for the transactions in $\mathcal{T}$;

1 Divide time into discrete time steps ;
2 Divide writing transactions into ℓ groups $A_0, A_1, \ldots, A_{\ell-1}$ according to their execution time durations such that A_i contains transactions with $\tau \in [2^i \cdot \tau_{\min}, (2^{i+1} \cdot \tau_{\min} - 1)]$; put read-only transactions in a special group B;
3 Divide each group A_i again into $\kappa = \lceil \log s \rceil + 1$ subgroups $A_i^0, A_i^1, \ldots, A_i^{\kappa-1}$ according to the number of shared resources they access such that A_i^j contains transactions with $\mathcal{R}(T_i) \in [2^j, 2^{j+1} - 1]$;
4 Order the groups and subgroups such that $A_i^j < A_k^l$ if $i < k$ or $i = k \wedge j < l$; assign special group B the highest order;
5 Resolve conflicts, if any, among transactions in groups and subgroups, at each time step based on some conflict resolution mechanism;

transactions) $\mathcal{T}$ is called *balanced* if $\beta = \Theta(1)$.

We give an outline of the scheduling approach to solve the balanced transaction scheduling problem in Algorithm 1. Let $\ell = \lceil \log(\frac{\tau_{\max}}{\tau_{\min}}) \rceil + 1$. The algorithm works by dividing the transactions into groups and subgroups and ordering them as mentioned in Lines 2–4 of Algorithm 1 and resolving conflicts, if any, that arise in each time step of the execution using an appropriate conflict resolution mechanism. We presented in [71, 73] two different versions of Algorithm 1 based on two different conflict resolution mechanisms applied in Line 5 of Algorithm 1.

The first version, called Clairvoyant, uses an idea of assigning a priority level to the pending transactions at any time step t which determines which transactions commit and which transactions abort. The priority level is either *high* (1) or *low* (0). Therefore, transactions in lower order subgroups (groups) have always higher priority than higher order subgroups (groups). In conflicts, high priority transactions (transactions in lower order groups and subgroups) abort low priority transactions (transactions in higher order groups and subgroups) and conflicts between transactions of the same priority level are resolved arbitrarily by computing a maximal independent set in the conflict graph of pending transactions. This decision is made in every time step of the execution. The separation of transactions in groups and subgroups helps in resolving the transaction conflicts efficiently so that tight competitive ratio bounds can be obtained.

The second version, called Non-Clairvoyant, uses an idea of resolving the conflicts in every time step based on the ordering of the groups and subgroups similar to Clairvoyant such that lower order subgroups (groups) have always higher priority than higher order subgroups (groups). However, when transactions in the same subgroup conflict, in contrast to the maximal independent set approach used in Clairvoyant, conflicts are resolved according to discrete random priority numbers. A transaction T_i, as soon as it starts execution, chooses a discrete priority number $r(T_i)$ uniformly at random in the interval $[1, M]$ and the transaction with small priority number wins at the time of conflict [70]. When a transaction aborts and restarts again, it does not retain the previous priority number as used in Greedy but chooses a discrete priority number again uniformly at random from the interval $[1, M]$.

Theorem 3.1. *For any balanced workload with $\beta = \Theta(1)$ and when $\ell = \mathcal{O}(1)$, Algorithm 1 has competitive ratio $\mathcal{O}(\sqrt{s} \cdot \mathbb{C}(M))$, where $\mathbb{C}(M)$ is a factor that depends on the conflict resolution*

mechanism used in Line 5 of the algorithm.

Proof (sketch). We start with the highlevel overview of the proof and later present details of the competitive ratio computation. The idea is to first consider the competitive ratio for a subgroup A_i^j and then extend it for the competitive ratio of a group A_i. After the competitive ratio for a group is multiplied by the number of groups ℓ, this will give the desired result for balanced transactions with almost equal time durations.

For transactions in a subgroup A_i^j, we obtain two different competitive ratios. The first competitive ratio is obtained through the upper and lower bounds based on the degree of a transaction in the conflict graph $\mathcal{G}(A_i^j)$ of the transactions in that subgroup. The upper bound is obtained by multiplying the maximum number of transactions that write to a resource and the maximum number of resources needed by any transaction in A_i^j. The lower bound is given by the maximum number of transactions in A_i^j that write to a resource because they conflict in accessing a shared resource and they need to be serialized to successfully commit in the shortest possible time.

The second competitive ratio is obtained through the upper and lower bounds based on the pending commit property (Definition 1) and balancing ratio properties for the transactions in that subgroup. The pending commit property provides us the upper bound as $|A_i^j|$ transactions in A_i^j are executed sequentially in the worst-case. The balancing ratio property provides us the lower bound as it determines the minimum number of transactions in A_i^j that conflict with each other while accessing a particular resource. These minimum number of transactions need to be serialized to commit them in the shortest possible time.

We now provide the details on the computation of these competitive ratios. For the competitive ratio based on the degree of a transaction, let γ be the maximum number of transactions in A_i^j that write to any resource in $\mathcal{R}$. All these γ transactions in A_i^j have to be serialized because they all conflict with each other in accessing the common resource; this gives the lower bound of $\gamma \cdot \tau_{\min}^j$. Here $\tau_{\min}^j$ denotes the execution time of the shortest transaction in A_i^j. The upper bound cannot be more than $(s_{\max}^j \cdot \gamma + 1) \cdot \tau_{\max}^j \cdot \mathbb{C}(M)$, where $s_{\max}^j$ is the maximum number of resources needed by any transaction $T \in A_i^j$, $\tau_{\max}^j$ is the execution time of the longest transaction in A_i^j, and $\mathbb{C}(M)$ is a factor that depends on the conflict resolution mechanism used in Line 5 of Algorithm 1 (we give the precise value of $\mathbb{C}(M)$ later).

For the other competitive ratio, the upper bound is at most $|A_i^j| \cdot \tau_{\max}^j \cdot \mathbb{C}(M)$ due to the pending commit property which, in the worst-case, serializes the transactions of A_i^j. The corresponding lower bound is at least $\frac{|A_i^j| \cdot \beta \cdot s_{\max}^j}{2 \cdot s} \cdot \tau_{\min}^j$ using the balancing ratio property. This is because, according to the definition of balancing ratio, each transaction $T \in A_i^j$ accesses at least $|\mathcal{R}_w(T)| \geq \beta \cdot s_{\max}^j/2$ resources. Therefore, the minimum number of transactions in A_i^j that access a particular resource $R \in \mathcal{R}$ is at least the ratio of the sum of $|\mathcal{R}_w(T)|$ among all $T \in A_i^j$ to the total number of resources s; all these transactions of A_i^j accessing R should be serialized because they conflict with each other. The balancing ratio property is very useful here, otherwise this lower bound claim would not have been possible to achieve.

As $\tau_{\max}^j/\tau_{\min}^j \leq 2$ for the transactions in any subgroup A_i^j, we obtain two independent competitive ratios for A_i^j combining respective upper and lower bounds. Combining these two competitive ratios, we obtain $4 \cdot \min\left\{s_{\max}^j, \frac{s/\beta}{s_{\max}^j}\right\} \cdot \mathbb{C}(M)$ for the competitive ratio of transactions in A_i^j. This competitive ratio bound forms a bitonic sequence of κ competitive ratios for κ subgroups of transactions (Line 3 of Algorithm 1) in any group A_i, and this sequence of competitive ratios has a single

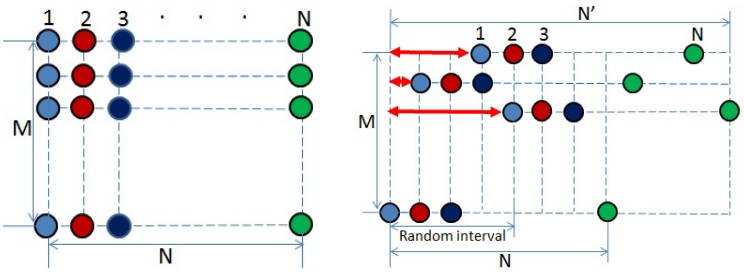


Figure 2: Execution window model for TM. Left: before execution; Right: after execution.

pick at the subgroup A_i^y with $y = \log(s/\beta)/2$. Therefore, summing the sequence of κ competitive ratios, we obtain the competitive ratio of $\mathcal{O}(\sqrt{s/\beta} \cdot \mathbb{C}(M))$ for group A_i consisting of κ subgroups. For ℓ groups, we have $\mathcal{O}(\ell \cdot \sqrt{s/\beta} \cdot \mathbb{C}(M))$. □

Consider the conflict graph $\mathcal{G}(\mathcal{T})$ of a balanced transaction scheduling problem with a set $\mathcal{T}$ of M transactions. Let $d_{T,max}$ be the maximum number of transactions that conflict with some transaction $T \in \mathcal{T}$. The factor $\mathbb{C}(M)$ measures the maximum ratio of the total execution time of the transactions in $\mathcal{T}$ using some conflict resolution mechanism and the execution time obtained serializing $d_{T,max}$ transactions. As shown in [71, 73], the factor $\mathbb{C}(M)$ due to the conflict resolution mechanism based on maximal independent set of pending transactions used in Clairvoyant is $\mathcal{O}(1)$ and the conflict resolution mechanism based on discrete priority numbers used in Non-Clairvoyant is $\mathcal{O}(\log M)$ with high probability. Therefore, Algorithm 1 is $\mathcal{O}(\sqrt{s})$-competitive in the first case but it needs to know a priori the conflict graph of transactions at each time step of execution to resolve conflicts. Algorithm 1 is $\mathcal{O}(\sqrt{s} \cdot \log M)$-competitive, with high probability, in the second case, without the explicit knowledge of the conflicts. In other words, we removed the necessity of the global conflict knowledge in Non-Clairvoyant in the expense of $\mathcal{O}(\log M)$ increase in the competitive ratio, with high probability. When transactions in $\mathcal{T}$ can fail at most k times, competitive ratio bounds of our algorithms increase proportionally with k. Therefore, Algorithm 1 is the first sub-linear algorithm for a simple restricted version of the one-shot scheduling problem given in Section 2.

The balanced version of the one-shot scheduling problem leads us to obtain the inapproximability result for the transaction scheduling problem using a reduction from the graph coloring problem given in [50].

Theorem 3.2. *Unless NP ⊆ ZPP, there does not exist a polynomial time algorithm for every instance with $\beta = 1$ and $\ell = 1$ of the balanced transaction scheduling problem that can achieve competitive ratio smaller than $\mathcal{O}\left((\sqrt{s})^{1-\epsilon}\right)$ for any constant $\epsilon > 0$,*

This implies that the $\mathcal{O}(\sqrt{s})$ bound given above for $\beta = \Theta(1)$ and $\ell = \mathcal{O}(1)$ is very close to optimal as ϵ approaches 0.

3.2 Window-Based Workloads

We considered the transaction scheduling problem for $M \times N$ *execution windows of transactions* with $M \geq 1$ threads and N transactions per thread (see left of Fig. 2) in [75, 80]. Each window W consists of a set of transactions $\mathfrak{T}(W) = \{(T_{11}, \ldots, T_{1N}), (T_{21}, \ldots, T_{2N}), \ldots, (T_{M1}, \ldots, T_{MN})\}$,

where each thread P_i issues N transactions $T_{i1}, \ldots, T_{iN}$ in sequence, so that T_{ij} is issued as soon as $T_{i(j-1)}$ has committed. This departs from the one-shot model given in Section 2 where $N = 1$ (one transaction per thread). However, similar to the one-shot model, transactions share a set of $s \geq 1$ shared resources $\mathcal{R}$, i.e. $\mathcal{R}(T_i)$ denotes the resources read or written by T_i. For the purpose of analysis, we assume that all transactions have the same execution time duration $\tau = \tau_{ij}$ and this time does not change over time.

This execution window model is useful in many real-world scenarios. Consider for an example the scenario in which each thread needs to execute a job that comprises of many transactions. These transactions will be executed one after another on the processor core where the thread is running and the thread tries to execute T_i as soon as T_{i-1} has finished execution and successfully commit. Note that jobs in different threads may have different number of transactions. This execution window model can successfully handle such scenarios as well.

Assuming that each transaction conflicts with at most C other transactions inside the $M \times N$ window, a trivial greedy contention manager can schedule them within τCN time. Our goal is to solve this window transaction scheduling problem minimizing the makespan. Algorithm 2 is an outline of the solution to this window transaction scheduling problem. The algorithm works by dividing time into time frames, denoted as F_{ij}, of size $\Phi = \mathcal{O}(\tau \cdot \log(MN) \cdot \mathbb{C}(M, N))$ time steps based on the values of τ, M, and N, where $\mathbb{C}(M, N)$ is a factor that depends on the conflict resolution mechanism used in Line 4 of Algorithm 2 and measures the maximum ratio of the execution times of the transactions inside each frame similar to the maximum ratio of the execution times of the transactions inside a subgroup in balanced workloads (we give precise value of $\mathbb{C}(M, N)$ later). The time frames are designed in such a way that transactions that become high priority in the beginning of any time frame finish execution and commit before the end of the time frame. To guarantee this property, we assign a random interval q_i between 0 to $C/\log(MN) - 1$ frames uniformly at random to each thread $i, 1 \leq i \leq M$. During q_i, all transactions T_{ij} of the thread i are in low priority (Line 2 of Algorithm 2). Here C plays the important role in determining how much randomization is needed in the beginning of the execution. After that transactions in each thread i become high priority one after another in the beginning of every successive frame and remain in high priority until they commit (Line 3 of Algorithm 2). In other words, the first transaction in thread i becomes high priority in the beginning of first frame after the random interval for thread i, the second transaction becomes high priority in the beginning of the second frame after the random interval for thread i, and so on. We guarantee in Algorithm 2 that at most $\mathcal{O}(\log(MN))$ transactions (from all the threads) become high priority, with high probability, at the beginning of any frame such that they will be committed before the end of that time frame. The use of randomization interval q_i helps us to minimize the transaction conflicts by forcing each transaction to become high priority in an appropriate randomly selected frame so that transaction will be executed in a way that is shown in right of Fig. 2.

Theorem 3.3. *For any execution window W with equal length transactions, Algorithm 2 has competitive ratio $\mathcal{O}((s + \log(MN)) \cdot \mathbb{C}(M, N))$, with high probability, where $\mathbb{C}(M, N)$ is a factor due to the conflict resolution mechanism used in Line 4 of the algorithm.*

Proof (sketch). Note that Algorithm 2 needs total $N' = (C/\log(MN) + N) \cdot \mathcal{O}(\tau \cdot \log(MN) \cdot \mathbb{C}(M, N)) = \mathcal{O}(\tau \cdot (C + N\log(MN)) \cdot \mathbb{C}(M, N))$ time steps to finish all the transactions. This is because we have up to $C/\log(MN)$ frames that are used in randomization interval and N frames after that for N transactions in any thread with $\Phi = \mathcal{O}(\tau \cdot \log(MN) \cdot \mathbb{C}(M, N))$ being the size of each

Algorithm 2: A generic window transaction scheduling algorithm

Input: An $M \times N$ window of transactions W with $M \geq 1$ threads of N transactions, where C is the maximum number of transactions in W that any transaction conflicts with;

Output: A greedy execution schedule for the transactions in W;

1 Divide time into poly-logarithmic (with respect to M, N, C) time duration frames;
2 Assign a random interval of $q_i \in [0, C/\ln(MN) - 1]$ frames to each thread i, during which all transactions T_{ij} of the thread are in low priority;
3 After interval q_i finishes, each transaction T_{ij} becomes high priority in frame $F_{ij} = q_i + (j-1)$;
4 Resolve conflicts, if any, among transactions at each time step based on some conflict resolution mechanism;

frame. Moreover, the use of randomized intervals guarantees that $\mathcal{O}(\log(MN))$ transactions become high priority in the beginning of each frame, with high probability, and they commit before the frame duration expires. Let $\gamma_{\max} := \max_{1 \leq j \leq s} \gamma(R_j)$, where $\gamma(R_j)$ denotes the number of transactions that write to resource R_j. Let $\lambda_{\max} := \max_i \lambda(T_i)$, where $\lambda(T_i) = |\{R : R \in \mathcal{R}(T_i) \wedge \gamma(R) \geq 1\}|$ denotes the number of resources that cause conflicts to any transaction $T_i \in \mathfrak{T}(W)$. We can relate C with $\gamma_{\max}$ and $\lambda_{\max}$ through the relation $\gamma_{\max} - 1 \leq C \leq \lambda_{\max} \cdot \gamma_{\max}$. We can relate the optimal time to execute all the transactions in $\mathfrak{T}(W)$ to τ, $\gamma_{\max}$, and N such that the optimal time is at least $\tau \cdot \max\{\gamma_{\max}, N\}$. Moreover, we can show that $\lambda_{\max} \leq s$. Combining all these results, we obtain the desired $\mathcal{O}((s + \log(MN)) \cdot \mathbb{C}(M, N))$ competitive ratio, with high probability. □

We presented in [75, 80] three different variants of Algorithm 2 based on three different conflict resolution mechanisms (Line 4 of Algorithm 2). In order to resolve conflicts within a frame in the first variant called Offline-Greedy, a maximal independent set is chosen, similar to Clairvoyant, from the conflict subgraph induced by the high priority transactions. The effect of this approach is that $\mathcal{O}(\log(MN))$ transactions that become high priority in any frame F_{ij} finish execution (and commit) in $\mathcal{O}(\log(MN))$ time steps, with high probability, such that $\mathbb{C}(M, N) = \mathcal{O}(1)$. Therefore, the competitive ratio for Offline-Greedy is $\mathcal{O}((s + \log(MN))$, with high probability. The second variant called Online-Greedy is only a $\mathcal{O}(\log(MN))$ factor worse compared to Offline-Greedy, but does not require knowledge of the conflict graph (due to the use of discrete random priorities). Nevertheless, it still relies on knowledge of C in the window. One transaction finishes in $\mathcal{O}(\log(MN))$ time steps, with high probability, in Online-Greedy, requiring $\mathcal{O}(\log^2(MN))$ time steps for $\mathcal{O}(\log(MN))$ transaction in a frame due to the conflict resolution mechanism used within a frame that is based on discrete random priority numbers similar to Non-Clairvoyant. Therefore, $\mathbb{C}(M, N) = \mathcal{O}(\log(MN))$ for Online-Greedy. The third variant called Adaptive-Greedy is an alternative algorithm which dynamically guesses C. The benefit is that these algorithms exhibit worst-case competitive ratios of almost $\mathcal{O}(s)$ for $M \times N$ windows of transactions in contrast to previous algorithms which achieve $\mathcal{O}(s)$ competitive ratio only for the one-shot model of Section 2. When transactions in $\mathfrak{T}(W)$ can fail at most k times, competitive ratio bounds of our algorithms increase proportionally with k.

Experiments: We performed experimental evaluation of several variants of Algorithm 2 in [72, 75]. In the experiments, we considered Algorithm 2 variants for eager conflict management systems (eager conflict management systems handle conflicts as soon as they are detected) and performed the evaluation in DSTM2 [43], an eager STM system. We used two simple benchmarks: the linked list (List) and red-black tree (RBTree), and a sophisticated benchmark vacation (Vacation)

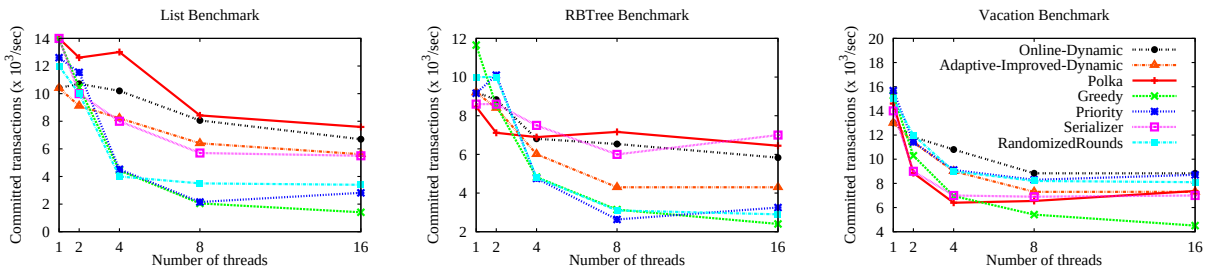


Figure 3: Comparison of performance throughput results in List, RBTree, and Vacation benchmarks in high contention. Higher is better.

from the STAMP suite [18, 43, 44]. These benchmarks are configured to generate large amounts of transactional conflicts (i.e., high contention scenario). The throughput results of several algorithms are given in Fig. 3. We ran these specific experiments for 10 seconds using up to 16 threads and the data plotted are the average of 6 experiments.

We did not use the Offline-Greedy algorithm in evaluation because it resolves conflicts based on the complete knowledge of the conflict graph in every time step of execution, an assumption which may not be practical. In the experiments, Online-Dynamic is a new improved version of Online-Greedy algorithm where the frames are dynamically contracted or expanded based on the amount of contention inside the frame. Adaptive-Improved-Dynamic is the variant of Adaptive-Greedy where frames are dynamically contracted or expanded similar to Online-Dynamic. Performance variance is generally minimal between the two best performing strategies Online-Dynamic and Adaptive-Improved-Dynamic. Moreover, we compared our window-based algorithms with Polka [68] (the published overall best contention manager, but does not have the formal analysis of the upper bound), Greedy [36] (the first contention manager which has non-trivial theoretical provable properties), Priority [68] (a simple static priority-based contention manager), Serializer [27] (a serialization based scheduler), and RandomizedRounds [70] (a random priority-based contention manager). The conclusion from Fig. 3 is that Algorithm 2 variants improve throughput over Greedy and achieve throughput that is comparable to Polka in almost all the benchmarks that we considered. The evaluation shows the benefits of our algorithms in practice along with their non-trivial theoretical guarantees.

The transaction scheduling problem is also studied in several other papers, e.g. [9, 11–13, 58, 67, 82], for TM implementations in both hardware and software. However, they do not provide the formal analysis and the performance of their techniques is evaluated through benchmarks only.

4 Distributed Networked Systems

Distributed networked systems represent the scenario of completely decentralized distributed shared memory where processors are placed in a network which communicate through a message passing environment. Here, the network is represented with an arbitrary weighted graph $G = (V, E, w)$, where V is the set of nodes (machines), E is the set edges (interconnection links between machines), and w is a weight function in E which reflects physical distances and delays. This model is more abstract than the hierarchical multilevel cache, because the network could be any arbitrary topology not restricted to any specific multiprocessor architecture. Thus, it models distributed networks over

large areas. To solve the transaction scheduling problem in distributed systems, nodes need to use a transaction scheduling algorithm to resolve conflicts that arise while executing transactions. To support transaction scheduling satisfying atomicity, each node is enriched with a *transactional memory proxy* module that interacts with the local node and also with the proxies at other nodes [46]. The proxy module is asked to open the shared object when it is needed for reading or writing by a transaction. The proxy module checks whether the object is at the local cache, otherwise it calls an appropriate algorithm to get that object from the node that has it. At the commit time of a transaction, proxy checks whether any object that is read and written by that transaction was not invalidated by other transactions that are committed from other nodes. If that is the case, the proxy asks the transaction to abort, otherwise it allows the transaction to commit. The aborted transactions restart their execution and try to commit again.

When the proxy module of a node receives a request (from a remote node) for the shared object that is at the local node, it checks whether a local pending transaction using it. If the object is in use, the proxy can give the object to the requester aborting the local transaction or delay the response for a while so that local transaction can commit. This decision is done through the scheduling algorithm used in the nodes.

Several researchers [14, 23, 53, 59] presented techniques to implement TM in distributed networked systems. Manassiev *et al.* [59] presented the lazy conflict detection and handling algorithm based on global lock. Kotselidis *et al.* [53] presented the serialization/multiple lease based algorithm. Bocchino *et al.* [14] and Couceiro *et al.* [14, 23] presented the commit-time broadcasting based algorithm. Control-flow based distributed TM implementation is studied by Saad and Ravindran [65]. Romano *et al.* [63] discussed the use of the TM programming model in the context of the cloud computing paradigm and posed several open problems. Kim and Ravindran [51] studied transaction scheduling in replicated *data-flow* based distributed TM systems. Saad and Ravindran [65] provided a Java framework implementation, called HyFlow, for distributed TM systems. Recently, Hendler *et al.* [42] studied a lease based hybrid distributed software transactional memory implementation which dynamically determines whether to migrate transactions to the nodes that own the leases, or to demand the acquisition of these leases by the node that originated the transaction.

As transactions are scheduled and conflicts are resolved using a scheduling algorithm in each network node, the focus in the TM implementation in distributed networked systems is on how to find the shared objects needed by transactions efficiently from the remote nodes and provide the *consistency* of the objects after transactions commit and abort. These previous algorithms [14, 23, 53, 59, 63] essentially try to provide consistency of the shared objects. However, they either use global lock, serialization lease, or commit-time broadcasting technique which do not scale well with the size of the network [7]. Moreover, they do not provide the formal analysis of the cost incurred by their algorithms to support distributed transaction scheduling and the performance of these techniques are evaluated through experiments only. Thus, it is of great importance to design consistency algorithms that scale well with the size, complexity, and network kind of the distributed systems, and also provide reasonable theoretical and empirical performance. We provide an overview of the work on designing scalable consistency algorithms for supporting TM in distributed networked systems.

Algorithm	stretch			Network	Runs on
	sequential	one-shot	dynamic		
Arrow [25]	$\mathcal{O}(S_{ST})$ [25]	$\mathcal{O}(S_{ST} \cdot \log l)$ [47]	$\mathcal{O}(S_{ST} \cdot \log D)$ [54]	General	Spanning tree
Relay [88]	$\mathcal{O}(S_{ST})$ [88]	$\mathcal{O}(S_{ST} \cdot \log l)$ [47]	$\mathcal{O}(S_{ST} \cdot \log D)$ [89]	General	Spanning tree
Combine [7]	$\mathcal{O}(S_{OT})$ [7]	$\mathcal{O}(S_{OT} \cdot \log l)$ [47]	$\mathcal{O}(S_{OT} \cdot \log D)$ [54]	General	Overlay tree
Combine [7]	$\mathcal{O}(\log D)$ [76]	$\mathcal{O}(\log D)$ [76]	$\mathcal{O}(\log D)$ [76]	Constant doubling	Hierarchical directory (independent sets)
Ballistic [46]	$\mathcal{O}(\log D)$ [46]	$\mathcal{O}(\log D)$ [46]	$\mathcal{O}(\log D)$ [76]	Constant doubling	Hierarchical directory (independent sets)
Spiral [79]	$\mathcal{O}(\log^2 n \cdot \min\{\log n, \log D\})$ [79]	$\mathcal{O}(\log^2 n \cdot \min\{\log n, \log D\})$ [79]	$\mathcal{O}(\log^2 n \cdot \min\{\log n, \log D\})$ [76]	General	Hierarchical directory (sparse covers)

Table 2: Comparison of consistency algorithms, where $S_{ST} = \mathcal{O}(D)$ is the stretch of the spanning tree, $S_{OT} = \mathcal{O}(D)$ is the stretch of the overlay tree, $l \leq n$ is the number of *move* operations in one-shot executions, n is the number of nodes, and D is the diameter of the network.

Herlihy and Sun [46] proposed Ballistic consistency algorithm. This algorithm is *hierarchial*: network nodes are organized as clusters at different levels. They evaluated the formal performance of Ballistic by its *stretch* (i.e., the competitive ratio on distances): each time a node issues a request for a remote shared object, compute the ratio of the algorithm's communication cost for that request to the optimal communication cost for that request. The optimal communication cost is computed based on the shortest path distances between the requesting node and the node in which the request finds that object. In *constant doubling* networks, their algorithm achieves amortized $\mathcal{O}(\log D)$ stretch, where D is the diameter of the constant doubling network for non-overlapping (i.e., sequential) requests to locate and move a cached copy of an object from one node to another. In this algorithm, concurrent requests are synchronized by *path reversal*: when two requests meet at some intermediate node, the second request is diverted behind the first request.

The Arrow algorithm [25] originally designed for the distributed queuing problem can also be used as the consistency algorithm for TM in distributed systems. Zhang and Ravindran [88] proposed the Relay consistency algorithm. Both Arrow and Relay run on a *spanning tree*. In Relay, the pointers lead to the node that is currently holding the object and the pointers are changed only after the object moves from one node to another, like the tree-based mutual exclusion algorithm of Raymond [62]. Relay has stretch $\mathcal{O}(S_{ST})$ in sequential executions, where S_{ST} is the stretch of the pre-selected spanning tree ST. They also showed that Relay efficiently reduces the worst-case number of total abortions of transactions to $\mathcal{O}(M)$ in comparison to using Arrow [25, 62], which has an $\mathcal{O}(M^2)$ for M transactions requesting the same object. Recently, Attiya *et al.* [7] proposed Combine, which runs on an *overlay tree*, whose leaves are the computing nodes of the system. They claimed that Combine avoids *race conditions* (missing one concurrent request by another) of Ballistic and Relay by combining requests that overtake each other as they pass through the same node. Combine exhibits the stretch $\mathcal{O}(S_{OT})$ in sequential executions, where S_{OT} is the stretch of the embedded overlay tree OT. The stretch of Arrow, Relay and Combine may be as much as the diameter of the network. Kim and Ravindran [52] proposed a technique that improves the stretch

of Relay to $\mathcal{O}(\log n)$ in bimodal workloads in the worst-case and $\Theta(\log(n-m))$ in the average-case, for n nodes and m reading transactions. Table 2 summarizes the properties of the consistency algorithms in all possible (sequential, one-shot, and dynamic) execution scenarios. In *sequential* executions, object requests do not overlap with each other, whereas object requests are issued at the same time in *one-shot* executions and no further requests occur. Object requests are issued in arbitrary moments of time in *dynamic* executions.

4.1 Directory-Based Approach

We presented Spiral, a novel consistency algorithm, in [79]. Spiral was designed for the *data-flow* implementation of TM in distributed networked systems. Spiral supports three basic operations to provide consistency of the shared objects: (i) *publish*, allowing a shared object ξ to be inserted in the directory so that other nodes can find it; (ii) *lookup*, providing a read-only copy of the object ξ to the requesting node; (iii) *move*, allowing the requesting node to write the object ξ locally after the node gets it. The algorithm runs on a *directory* built upon a hierarchical cluster construction we describe below.

There are $h+1 = \mathcal{O}(\log D)$ cluster levels in Spiral such that cluster diameters increase exponentially (see Fig. 4), where D is the diameter of the network. In each cluster one node is chosen to act as a leader which is used to communicate with different level clusters. Clusters may overlap and the same node may act as a leader in multiple levels. At the bottom level (level 0) each cluster consists of an individual node, while at the top level (level h) there is a single cluster for the whole graph with a special leader node called *root*. Only the bottom level nodes can issue requests for the shared objects, while the nodes in higher levels are used to propagate the requests in the graph. The clusters are built using *sparse covers* obtained from a hierarchical partitioning in Gupta *et al.* [37]. We obtain the directory by organizing the clusters in a *logical tree* structure connecting the limited subset of leader nodes of consecutive levels starting from the lowest level and ending at the top level.

We provide algorithm description below considering only one shared object; to support multiple objects, the directory can be replicated appropriately. Given a shared object the algorithm maintains a *directory path* in the logical tree structure (i.e., directory). A directory path is a directed path from the root node to the bottom-level node that owns the shared object. This path follows (i.e., visits) the leader nodes in every level. The directory path is updated whenever the object moves from one node to another. In order to get access to the object, each bottom level node uses a *spiral path* to intersect the directory path and then reach the object. The spiral path of a node $u \in G$ is built by visiting upward the parent leader nodes in all the clusters that the node u belongs to starting from the bottom level up to the top level. The name of the algorithm (Spiral) is inspired by the path formed in the directory which slowly unwinds outwards while it visits cluster leaders of higher levels which are possibly farther away from the origin node. Algorithm 3 gives an overview of the approach.

A *publish* operation from any node $u \in V$ is served by following the spiral path up to the root and making each parent cluster leaders point toward child cluster leaders in its way (Lines 3, 4 of Algorithm 3). A *lookup* operation is served by first following the spiral path upward until it finds a downward (i.e., directory) path and then reaching the node that has the object following the downward pointers (Lines 5–7 of Algorithm 3). A *move* operation is also served similar to *lookup* but while going upward following the spiral path it sets downward pointers and removes the existing pointers while going downward following the downward path (Lines 8–10 of Algorithm

Algorithm 3: A generic directory-based consistency algorithm for TM in distributed systems

Initialization:

Given a network graph G, develop a directory $\mathcal{Z}$ (a logical tree structure) based on some hierarchical graph partitioning method and assign each cluster a leader;

Publish an object ξ at some node v:

Until a publish message from v reaches the root node of $\mathcal{Z}$, visit all the parent clusters (i.e. higher level clusters) in the spiral path of v towards root making parent cluster leaders point toward child cluster leaders;

Lookup from node u for an object ξ at node v:

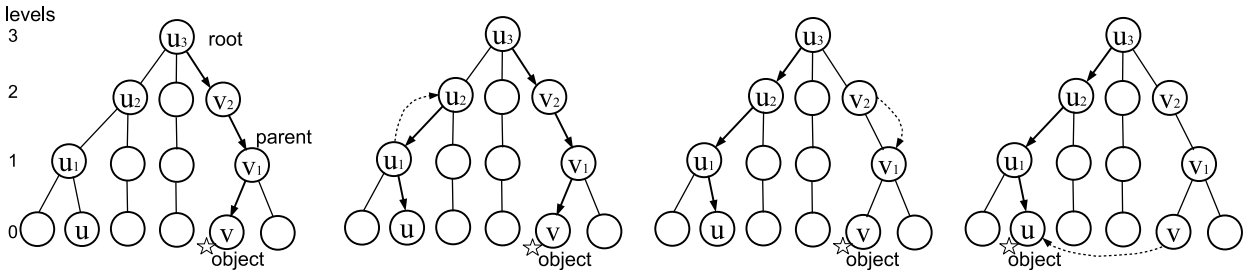
Until a lookup message from u finds a downward path in $\mathcal{Z}$, visit all the parent clusters in the path of u towards the root;

When the downward path is found, go to v following the pointers and send the read-only copy of ξ to u;

Move from node u for an object ξ at node v:

Until a move message from u finds a downward path in $\mathcal{Z}$, visit all the parent clusters in the path of u towards the root making parent cluster leaders point to child cluster leaders;

When the downward path is found, go to v following the path but removing its pointers and send the writable copy of ξ to u invalidating ξ from v and its read-only copies from other nodes;



((a)) Initially, node v publishes the object ((b)) The request continues up phase ((c)) The request continues down phase ((d)) Object is moved directly from v to u

Figure 4: Illustration of Spiral algorithm for a move request issued by node u for the object at node v.

3). An execution example with a *move* request from node u for the shared object at node v is depicted in Fig. 4. The *move* request from u goes upward following the spiral path of u until it finds the downward pointer at u_3, setting downward pointers in its way (Figs. 4(a) and 4(b)), after that it follows the leaders v_2, v_1, v in the directory path to reach v, removing the existing downward pointers in its way (Fig. 4(c)). The object is then moved to u following some shortest path in G (Fig. 4(d)).

Spiral is suitable for arbitrary network topologies. We measure the performance of Spiral by stretch (competitive ratio on distances) similar to previous algorithms [7, 25, 46, 88] by comparing Spiral's communication cost for an operation (resp. for a set of operations) to the optimal communication cost for that operation (resp. for that set of operations). We proved the following theorem for Spiral; details can be found in [76, 77]. For the analysis, we consider the dynamic execution of any arbitrary set of *move* operations that covers all possible execution scenarios including sequential and one-shot executions

Theorem 4.1. Spiral *has* $\mathcal{O}(\log^2 n \cdot \min\{\log n, \log D\})$ *stretch in any execution (sequential, one-shot, or dynamic) for any arbitrary set of* move *operations.*

Proof (sketch). We can prove that, in Spiral, any spiral path or directory path from any leaf node to any level i leader node has length $\mathcal{O}(2^i \cdot \log^2 n)$. Recall that, in Spiral, each *move* request is served finding a downward path at a cluster leader at some level. Therefore, for any set of *move* operations, we can count the number of requests that reach any level $i, 1 \leq i \leq h$, in the directory following their spiral paths upward before they find corresponding downward paths at that level. The upper bound in communication cost for that level is given by summing up the spiral path length of all the operations that reach to that level. We can also obtain a corresponding lower bound for the requests that reach that level. The idea is that if the spiral paths of any two *move* requests r_i and r_j issued, respectively, by the leaf nodes u and v intersect at level i, then the distance between them in G must be at least 2^i, since otherwise their spiral paths would intersect at level $i-1$ or lower. We say that two spiral paths *intersect* at level i if they visit the same leader node at level i (not necessary at the same time). Now combining the upper and lower bound costs for all the levels we obtain the desired bound of $\mathcal{O}(\log^2 n \cdot \log D)$. The $\log^2 n$ factor comes from the ratio of spiral (directory) path lengths of intersected requests with their corresponding shortest paths in G. The $\mathcal{O}(\log D)$ factor comes from the number of levels in the directory, as we take as overall lower bound the maximum lower bound cost among any level. Note that the factor of $\mathcal{O}(\log D)$ in the stretch is very loose. Nevertheless, it is still good when $D < n^{\mathcal{O}(1)}$. For $D > n^{\mathcal{O}(1)}$, we can do a tight analysis of the lower bound to obtain $\mathcal{O}(\log n)$ instead of $\mathcal{O}(\log D)$ in the stretch bound. Combining these two results, we obtain the desired *move* stretch of $\mathcal{O}(\log^2 n \cdot \min\{\log n, \log D\})$. □

For any *publish* operation, we can prove the total cost of $\mathcal{O}(D \cdot \log^2 n)$ to serve it. Moreover, we can prove $\mathcal{O}(\log^4 n)$ stretch for any *lookup* operation in Spiral algorithm. We obtain this without considering the request sequences as we did for move operations. Our approach of using *shortcut pointers* makes this possible; details can be found in [79]. Moreover, race conditions are avoided by appropriately ordering the overlapping clusters of the same level of the directory.

To the best of our knowledge, Spiral is the first consistency algorithm for TM in distributed systems that achieves poly-log approximation for stretch in general networks. Previous approaches, Arrow [25], Relay [88], Combine [7], and Ballistic [46], were only for either specific network topologies or they do not scale well in arbitrary network topologies. For example, Ballistic is only suitable for doubling-dimension metrics, which is not general enough to cover other network topologies; further, the spanning tree approach of Relay [88] does not perform well on trees that do not preserve the shortest path metric, as for example, in ring networks.

Experiments: Experimental results for Spiral in real world networks for the sequential and dynamic execution of 10 to 10,000 *move* operations are given in Fig. 5. For the experimentation we generated random networks of different sizes adapting the Erdős-Rényi model [29] (we report here the results from the networks with 512 nodes only), where a graph $G(n, \rho)$ is constructed connecting nodes randomly such the each edge is included in G with probability $0 < \rho < 1$ independent from every other edge. The graphs we use in the experiments are generated setting $\rho = 0.5$. The weight of each edge is also chosen at random, independently from the weight of every other edge, from 1 to 10. Spiral in the figure denotes the results in a sequential execution and Spiral(t) denotes the results in a dynamic execution in which a new *move* request is generated in every t steps from a randomly chosen node in the graph. The results show that Spiral performs slightly better in

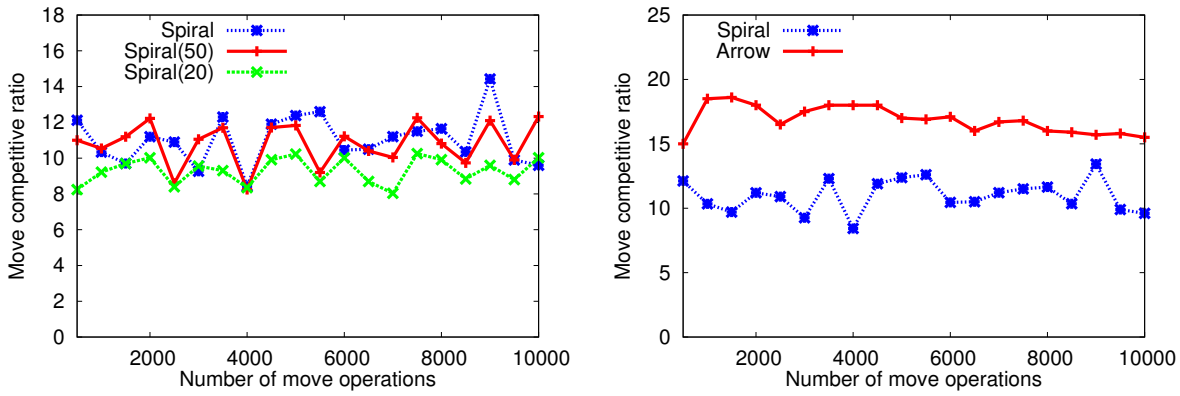


Figure 5: Performance of Spiral (Left) and its comparison to Arrow (Right) for up to 10,000 *move* operations in a network of 512 nodes. Lower is better. Spiral(t) denotes a dynamic execution where a new *move* operation is generated in every t time steps.

terms of communication cost when there are large number of active *move* requests at each step of the execution. This is because the paths used by requests become less fragmented when many requests are issued concurrently which in turn minimizes the communication cost [78]. We also compare the performance of Spiral with the performance of Arrow. This comparison is interesting because Arrow uses a spanning tree that is different from the hierarchical structure used by Spiral. The results (right of Fig. 5) show that Spiral achieves *move* competitive ratio that is $1.1 - 1.57$ times better in comparison to Arrow. This comparable performance is due to the fact the random network model allows for low cost spanning tree. The performance difference is significant when we consider the execution in special networks, e.g. ring. Several other performance comparisons results are reported in [78].

5 Non-Uniform Memory Access Systems

Multicore processor architectures provide interfaces that enable multicore chips to connect with each other through high speed interconnect communication links, in order to form larger size multiprocessor systems. An example is the Intel QuickPath Interconnect (QPI) [21] which is implemented in the Intel Pentium i7 Nehalem multicore architecture [22]. Fig. 6 illustrates an example organization of an interconnect multiprocessor system in a 3-dimensional grid. Such large scale architectures are suitable for high performance distributed and parallel computing. In IBM Blue Gene/L 65,000 nodes are interconnected as a $64 \times 32 \times 32$ 3-dimensional mesh or torus [1]. Recently, IBM Blue Gene/Q integrated a 5-dimensional torus [19]. Moreover, Cray XT5 [48] is also based on a similar multiprocessor organization. These configurations are known as Non-Uniform Memory Access (NUMA) systems where the shared memory is distributed among various processors. There are various ways to ensure that the caches of the cores are coherent, such as snoopy bus algorithm, or a distributed directory organization. An important characteristic is the *NUMA factor* which is related to the difference in latency for accessing data from a local memory location as opposed to a non-local one.

Wang *et al.* [85] evaluated several STM implementations on a big SMP machine that uses cache coherent NUMA (ccNUMA) architecture. They concluded that latencies due to remote

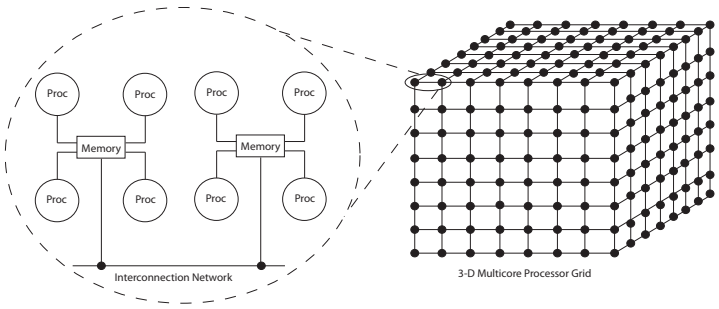


Figure 6: Multi-processor system with high speed interconnect (i.e., Intel QPI [21]).

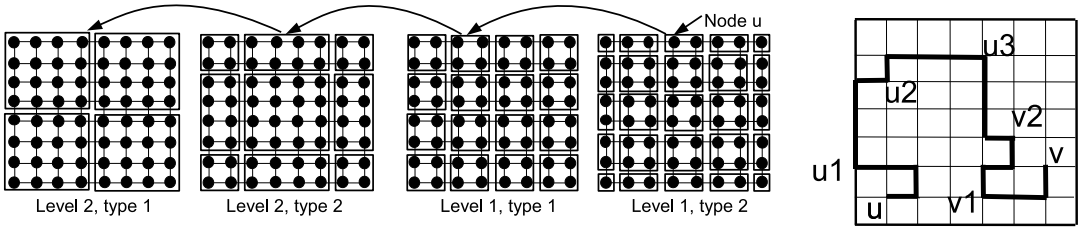


Figure 7: Left: illustration of mesh decomposition for the $2^3 \times 2^3$ 2-dimensional mesh. The decomposition for level 0 and level 3 is not shown; in level 0, every node is a submesh by itself, and in level 3 (top level) the whole mesh itself is a submesh. The arrows show the parent clusters of a particular node u; Right: move path in the 2-dimensional mesh by MultiBend for the *move* request shown in Fig. 4.

memory accesses is the key factor that influences STM performance. Lu *et al.* [56] proposed a latency-based scheduling algorithm with a forecasting-based conflict prevention method to improve the TM performance in NUMA systems. Kotselidis *et al.* [53] studied how to exploit STM on clusters. They concluded that the performance depends on network congestion. Blagodurov *et al.* [10] provided a case for NUMA-aware scheduling on multicore systems. However, they did not consider implementing transactions. Calciu *et al.* [17] designed a family of reader-writer lock algorithms tailored to NUMA architectures, extending the existing lock algorithms designed for UMA architectures.

For NUMA architectures, we are interested in minimizing the communication cost, makespan, and also the network load while executing transactions. In this direction, we give an algorithm that minimizes simultaneously the communication cost and the network load in accessing the memory locations of the shared objects. We leave the problem of scheduling transactions minimizing makespan in NUMA architectures as an open problem.

5.1 Load-Balanced Approach

In [74] we considered the problem of implementing a communication efficient consistency algorithm on grid network topologies which at the same time achieves load balancing (minimizes maximum node and edge utilization) of the network nodes and edges. This load balancing is beneficial because the network congestion can affect the overall performance of the algorithm and sometimes it is a

major bottleneck. For achieving simultaneously low communication cost and low congestion (i.e., load balancing), we applied techniques from *oblivious routing* [15] on d-dimensional grid network topologies, with near optimal congestion while maintaining small stretch (competitive ratio on distances). In particular, we combined an oblivious routing algorithm approach with the Spiral algorithm (Section 4) to obtain the desired algorithm with poly-log approximation in stretch and poly-log approximation in congestion (with respect to optimal edge congestion). In small (constant) degree graphs, low edge congestion implies also low node congestion.

The algorithm MultiBend presented in [74] demonstrates that such a construction with dual optimization in grids is feasible. MultiBend is based on an appropriate hierarchical clustering of 2-dimensional mesh networks, where each mesh of a specific size is decomposed into two types of submeshes (clusters), type-1 and type-2 (see Fig. 7). The type-1 submeshes are obtained by partitioning the mesh recursively into 4 submeshes by dividing each side m_i of the mesh by 2 until there is only one node in each submesh. Therefore, there are $\mathcal{O}(\log n)$ levels of type-1 submeshes with sizes increasing by a factor of 2 between consecutive levels starting from the lowest level, i.e., level 0 submeshes are the individual nodes of the mesh. The type-2 submeshes are obtained by taking the type-1 sumbeshes of that level and shifting them by $-m_i/2$ simultaneously in both dimensions. Some of the shifted submeshes are entirely with in the mesh and the remaining of the shifted submeshes are partially overlapped with the original mesh. For the partially overlapped submeshes, we keep only their intersection with the original mesh. These two types of submeshes make it possible to serve the *move* and *lookup* operations more quickly, without increasing the congestion. (Using only the type-1 submeshes may be significantly expensive while serving the *move* and *lookup* operations and the use of type-2 submeshes is instrumental in controlling the stretch while maintaining low congestion.) We obtain a directory by organizing the type-1 and type-2 submeshes of each level in a logical tree structure similar to the one we described for Spiral algorithm. Based on the directory we can further define paths for accessing a shared object that take now the form of *multi-bend* paths (Fig. 7). From this construction, we obtained $\mathcal{O}(\log n)$ approximation on both load and stretch which are almost-optimal results; compare them with the lower bounds due to Alon *et al.* [2] for the stretch and Maggs *et al.* [57] for the congestion.

The construction for a 2-dimensional mesh can be extended to a d-dimensional mesh with $\mathcal{O}(d)$ different types of submeshes. Both the stretch and the load approximation become $\mathcal{O}(2^d \cdot \log n)$ using this construction approach which is excessively high for a large d. Therefore, the alternative construction with $\mathcal{O}(d)$ different types of submeshes alleviates the problem such that the stretch becomes $\mathcal{O}(d \cdot \log n)$ and the load approximation becomes $\mathcal{O}(d^2 \cdot \log n)$. For a fixed d, these are also constant factors from the optimal [2, 57]. Similar to the Spiral algorithm, the routing paths when requesting an object are formed through the hierarchy of submeshes. However, to achieve load balancing, the leader nodes in clusters are changed every time the leader is accessed. We summarize the results below.

Theorem 5.1. *In d-dimensional grid networks,* MultiBend *has the stretch $\mathcal{O}(d \cdot \log n)$ and the load (congestion) approximation $\mathcal{O}(d^2 \cdot \log n)$ with high probability.*

Experiments: We performed experiments in a 16×16 nodes 2-dimensional grid network to see how the theoretical properties of MultiBend translate in practice. We compared the performance of MultiBend with existing algorithms Arrow [25] and Ballistic [46]. The *move* and *lookup* competitive ratios for 100 to 1,000 sequential operations are given in Fig. 8. The performance of Ballistic is

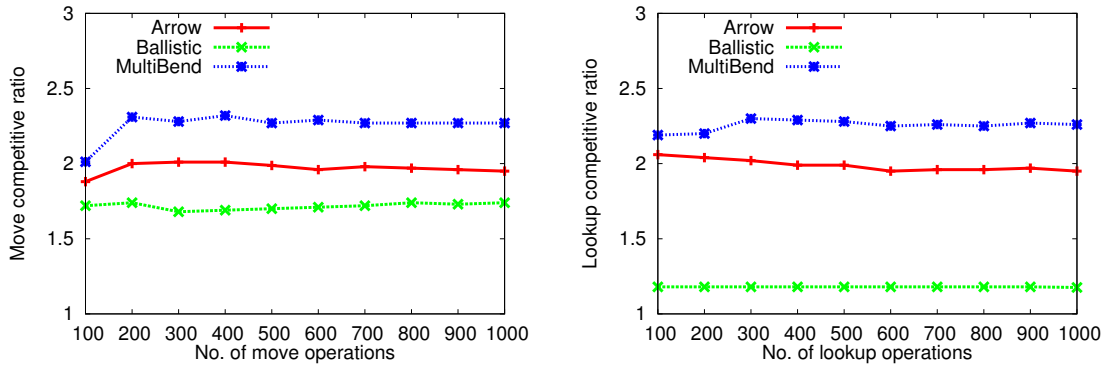


Figure 8: The stretch comparison for up to 1000 *move* operations (Left) and *lookup* operations (Right). Lower is better.

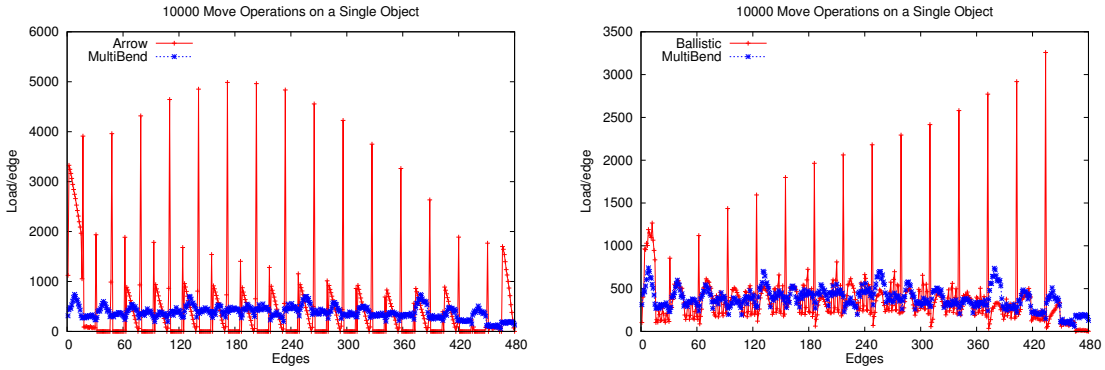


Figure 9: The load comparison for 10,000 *move* operations on a single object: Left: Arrow and MultiBend (the worst load per edge: Arrow 4986 at edge 172 and MultiBend 742 at edge 8); Right: Ballistic and MultiBend (the worst load per edge: Ballistic 4986 at edge 172 and MultiBend 742 at edge 8). The vertical lines show the number of times any edge is used by the algorithms while serving 10,000 *move* operations.

slightly better than the performance of MultiBend because we do not consider the move-parent set (leaders of limited number of clusters in each level that Ballistic visits in its directory) probing cost of Ballistic. Arrow performs better due to nice neighbor growth and connection properties of the grid network topology used in the experimentation. However, as shown in Fig. 9, MultiBend is the only algorithm that balances the load of the network edges. In Arrow and Ballistic, the load on some edges is proportional to the number of edges.

6 Future Directions and Concluding Remarks

Tightly-coupled Systems: A natural direction is to investigate the transaction scheduling problem for a combination of window-based model with the balanced workload model to achieve competitive ratio close to $\mathcal{O}(\sqrt{s})$ for windows of transactions. The significance of this is that the previous bounds in the literature considered only one-shot problems and do not generalize well in

the window-based model. For example, the bound of $\mathcal{O}(s)$ from [6] in the one-shot model becomes $\mathcal{O}(s \cdot N)$ in the window-based model.

Another natural direction is to determine what is the smallest balancing ratio (number of writes vs total number of reads and writes) that can maintain the same formal bounds in the balanced workload model. Moreover, it is interesting to investigate special cases of transaction conflict graphs which can enable makespan competitive ratios asymptotically smaller than $\mathcal{O}(\sqrt{s})$. Such graphs can represent interesting access patterns to shared resources. It is also interesting to consider scheduling algorithms for *mini-transactions* – simple atomic operations on a small number of locations [5].

Further, it is interesting to investigate how is the performance affected when we take into account the latency to access shared variables. Different processors may have different access times to the shared variables because they may reside in different levels of the memory hierarchy and in different caches. This affects both the lower and upper bounds of the makespan analysis. To properly model the access time variance one idea is to consider a weighted conflict graph and derive new lower and upper bounds on the makespan that take into account the edge weights of the graph. Moreover, it is interesting to design and analyze transaction scheduling algorithm using different performance metrics such as throughput, average response time, aborts per commit ratio, etc. Experimental evaluations for (combinations of) these metrics appeared in several papers, e.g. [4, 28, 71]. It is interesting also to experimentally evaluate these newly designed scheduling algorithms and existing algorithms [4, 27, 36, 87].

Distributed Networked Systems: Ballistic, Relay, and Combine (and also Spiral) have all been analyzed for a single shared object only. Thus, a natural extension is to handle multiple shared objects $\xi_1, \ldots, \xi_k$. In order to handle the case of multiple objects, one idea is to follow a universal TSP (traveling salesperson problem) approach [33, 38, 49]. A *universal TSP approach* computes a TSP tour Q for all the nodes in the network by going through all the nodes in some specific order. Now if we need to visit subset S of nodes inducing a sub-tour, the TSP tour Q approximates the optimal tour for S (in the induced subgraph) within a factor of $\mathcal{O}(\log^4 n / \log\log n)$ [49]. For each shared object ξ_i, we can then compute an approximate TSP tour for the object which visits all the nodes that have transactions that request the object (e.g. for *move*, namely, write operations, or for *lookup*). The TSP tour for the object is obtained by visiting in sequence the involved nodes in the universal TSP tour. For a transaction to execute in a node v, all the objects of the transaction must appear in v. Once all the objects appear in v, the transaction executes and then each object moves to the next node according to the order specified in their respective tours. Eventually, all the transactions will execute. The use of a universal TSP guarantees that there will be no deadlocks. In addition, the total communication cost of this approach will be close to optimal (poly-log approximation), assuming that the TSP tours of the objects are good approximations (typically, poly-log approximations). This idea will also be helpful in analyzing makespan.

For the experimental evaluation, it will be interesting to extend the HyFlow framework [66] − a Java framework implementation for STM in distributed systems − by including the aforementioned distributed directory algorithms. Moreover, it will be interesting to extend the STAMP benchmarks that are originally designed for tightly-coupled TM systems to support distributed implementations of the TM systems.

Non-uniform Memory Access Systems: For TM implementation in NUMA architectures, it will be interesting to explore load and distance competitive ratio bounds of MultiBend for the

case of d-dimensional networks with uneven dimensions. Moreover, it will be interesting to extend MultiBend for dynamic networks where nodes join and leave the network over time and make it fault-tolerant. This extension for dynamic networks also applies to algorithms designed for distributed networked systems (as existing algorithms can not handle dynamic networks). Moreover, the problem of incorporating the consistency algorithms in a full-fledged distributed TM system remains as an important open problem.

Acknowledgements

The authors would like to thank the editor Jennifer L. Welch for many thoughtful comments and suggestions on the earlier draft of this article.

References

[1] Narasimha R. Adiga, Matthias A. Blumrich, Dong Chen, Paul Coteus, Alan Gara, Mark Giampapa, Philip Heidelberger, Sarabjeet Singh, Burkhard D. Steinmacher-Burow, Todd Takken, Mickey Tsao, and Pavlos Vranas. Blue gene/l torus interconnection network. *IBM J. Res. Dev.*, 49(2-3):265–276, 2005.

[2] Noga Alon, Gil Kalai, Moty Ricklin, and Larry J. Stockmeyer. Lower bounds on the competitive ratio for mobile user tracking and distributed job scheduling. *Theor. Comput. Sci.*, 130(1):175–201, 1994.

[3] Mohammad Ansari, Christos Kotselidis, Mikel Lujan, Chris Kirkham, and Ian Watson. On the performance of contention managers for complex transactional memory benchmarks. In *ISPDC*, pages 83–90, 2009.

[4] Mohammad Ansari, Mikel Lujn, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *HiPEAC*, pages 4–18, 2009.

[5] Hagit Attiya. The inherent complexity of transactional memory and what to do about it. In *PODC*, pages 1–5, 2010.

[6] Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. Transactional contention management asanon-clairvoyant scheduling problem. *Algorithmica*, 57(1):44–61, 2010.

[7] Hagit Attiya, Vincent Gramoli, and Alessia Milani. A provably starvation-free distributed directory protocol. In *SSS*, pages 405–419, 2010.

[8] Hagit Attiya and Alessia Milani. Transactional scheduling for read-dominated workloads. *J. Parallel Distrib. Comput.*, 72(10):1386–1396, 2012.

[9] Tongxin Bai, Xipeng Shen, Chengliang Zhang, William N. Scherer III, Chen Ding, and Michael L. Scott. A key-based adaptive transactional memory executor. In *IPDPS*, pages 1–8, 2007.

[10] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *USENIXATC*, pages 1–1, 2011.

[11] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. Proactive transaction scheduling for contention management. In *MICRO*, pages 156–167, 2009.

[12] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. Bloom filter guided transaction scheduling. In *HPCA*, pages 75–86, 2011.

[13] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *ISCA*, pages 127–138, 2008.

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

[15] Costas Busch, Malik Magdon-Ismail, and Jing Xi. Optimal oblivious path selection on the mesh. *IEEE Trans. Comput.*, 57(5):660–671, 2008.

[16] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

[17] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *PPoPP*, pages 157–166, 2013.

[18] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, pages 35–46, 2008.

[19] Dong Chen, Noel Eisley, Philip Heidelberger, Sameer Kumar, Amith Mamidala, Fabrizio Petrini, Robert Senger, Yutaka Sugawara, Robert Walkup, Burkhard Steinmacher-Burow, Anamitra Choudhury, Yogish Sabharwal, Swati Singhal, and Jeffrey J. Parker. Looking under the hood of the ibm blue gene/q network. In *SC*, pages 69:1–69:12, 2012.

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

[21] Intel Corporation. A first look at the intel quickpath interconnect. `http://www.intel.com/intelpress/files/A_First_Look_at_the_Intel(r)_QuickPath_Interconnect.pdf`.

[22] Intel Corporation. Who moved the goal posts? the rapidly changing world of cpus. `http://software.intel.com/en-us/articles/who-moved-the-goal-posts-the-rapidly-changing-world-of-cpus/`.

[23] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues. D2stm: Dependable distributed software transactional memory. In *PRDC*, pages 307–313, 2009.

[24] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: Streamlining stm by abolishing ownership records. In *PPoPP*, pages 67–78, 2010.

[25] Michael J. Demmer and Maurice P. Herlihy. The arrow distributed directory protocol. *DISC*, pages 119–133, 1998.

[26] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *DISC*, pages 194–208, 2006.

[27] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *PODC*, pages 125–134, 2008.

[28] Aleksandar Dragojević, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *PODC*, pages 7–16, 2009.

[29] Paul Erdős and Alfréd Rényi. On random graphs I. *Publ. Math. Debrecen*, 6:290–297, 1959.

[30] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Trans. Parallel Distrib. Syst.*, 21(12):1793–1807, 2010.

[31] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, pages 237–246, 2008.

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

[33] Igor Gorodezky, Robert D. Kleinberg, David B. Shmoys, and Gwen Spencer. Improved lower bounds for the universal and a priori tsp. In *APPROX/RANDOM*, pages 178–191, 2010.

[34] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust Contention Management in Software Transactional Memory. In *SCOOL*, pages 1–8, 2005.

[35] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *DISC*, pages 303–323. 2005.

[36] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC*, pages 258–264, 2005.

[37] Anupam Gupta, Mohammad T. Hajiaghayi, and Harald Räcke. Oblivious network design. In *SODA*, pages 970–979, 2006.

[38] Mohammad T. Hajiaghayi, Robert Kleinberg, and Tom Leighton. Improved lower and upper bounds for universal tsp in planar metrics. In *SODA*, pages 649–658, 2006.

[39] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, Alan Gara, George Chiu, Peter Boyle, Norman Chist, and Changhoan Kim. The ibm blue gene/q compute chip. *IEEE Micro*, 32(2):48–60, 2012.

[40] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2Nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[41] D. Hasenfratz, J. Schneider, and R. Wattenhofer. Transactional memory: How to perform load adaption in a simple and distributed manner. In *HPCS*, pages 163 –170, 2010.

[42] Danny Hendler, Alex Naiman, Sebastiano Peluso, Francesco Quaglia, Paolo Romano, and Adi Suissa. Exploiting locality in lease-based replicated transactional memory via task migration. In *DISC*, pages 121–133, 2013.

[43] Maurice Herlihy, Fabian Kuhn, Srikanta Tirthapura, and Roger Wattenhofer. Dynamic analysis of the arrow distributed protocol. *Theor. Comp. Syst.*, 39(6):875–901, 2006.

[44] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

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

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

[47] Maurice Herlihy, Srikanta Tirthapura, and Rogert Wattenhofer. Competitive concurrent distributed queuing. In *PODC*, pages 127–133, 2001.

[48] Cray Inc. Cray xt^{TM} system overview. `http://docs.cray.com/books/S-2423-22/S-2423-22.pdf`.

[49] Lujun Jia, Guolong Lin, Guevara Noubir, Rajmohan Rajaraman, and Ravi Sundaram. Universal approximations for tsp, steiner tree, and set cover. In *STOC*, pages 386–395, 2005.

[50] S. Khot. Improved inapproximability results for maxclique, chromatic number and approximate graph coloring. In *FOCS*, pages 600–609, 2001.

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

[52] Junwhan Kim and Binoy Ravindran. On transactional scheduling in distributed transactional memory ystems. In *SSS*, pages 347–361, 2010.

[53] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Lujn, Chris Kirkham, and Ian Watson. Distm: A software transactional memory framework for clusters. In *ICPP*, pages 51–58, 2008.

[54] Fabian Kuhn and Rogert Wattenhofer. Dynamic analysis of the arrow distributed protocol. In *SPAA*, pages 294–301, 2004.

[55] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1991.

[56] Kai Lu, Ruibo Wang, and Xicheng Lu. Brief announcement: Numa-aware transactional memory. In *PODC*, pages 69–70, 2010.

[57] B. Maggs, F. Meyer auf der Heide, B. Voecking, and M. Westermann. Exploiting locality for data management in systems of limited bandwidth. In *FOCS*, pages 284–293, 1997.

[58] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *PPoPP*, pages 79–90, 2010.

[59] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP*, pages 198–208, 2006.

[60] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT*, pages 1–11, 2006.

[61] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. Metatm/txlinux: Transactional memory for an operating system. *IEEE Micro*, 28(1):42–51, 2008.

[62] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1):61–77, 1989.

[63] Paolo Romano, Luis Rodrigues, Nuno Carvalho, and Joäo Cachopo. Cloud-tm: harnessing the cloud with distributed transactional memories. *SIGOPS Oper. Syst. Rev.*, 44(2):1–6, 2010.

[64] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *PPoPP*, pages 47–56, 2010.

[65] Mohamed M. Saad and Binoy Ravindran. Snake: control flow distributed software transactional memory. In *SSS*, pages 238–252, 2011.

[66] Mohamed M. Saad and Binoy Ravindran. Supporting stm in distributed systems: Mechanisms and a java framework. In *TRANSACT*, pages 1–9. 2011.

[67] David Sainz and Hagit Attiya. Relstm: A proactive transactional memory scheduler. In *TRANSACT*, pages 1–8, 2013.

[68] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, pages 240–248, 2005.

[69] William N. Scherer III and Michael L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs (CSJP)*, St. John's, NL, Canada, Jul 2004.

[70] Johannes Schneider and Roger Wattenhofer. Bounds on contention management algorithms. *Theor. Comput. Sci.*, 412(32):4151–4160, 2011.

[71] Gokarna Sharma and Costas Busch. A competitive analysis for balanced transactional memory workloads. In *OPODIS*, pages 348–363, 2010.

[72] Gokarna Sharma and Costas Busch. On the performance of window-based contention managers for transactional memory. In *APDCM*, pages 559–568, 2011.

[73] Gokarna Sharma and Costas Busch. A competitive analysis for balanced transactional memory workloads. *Algorithmica*, 63(1-2):296–322, 2012.

[74] Gokarna Sharma and Costas Busch. Towards load balanced distributed transactional memory. In *Euro-Par*, pages 403–414, 2012.

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

[76] Gokarna Sharma and Costas Busch. An analysis framework for distributed hierarchical directories. *Algorithmica*, Preprint:1–32, 2013.

[77] Gokarna Sharma and Costas Busch. An analysis framework for distributed hierarchical directories. In *ICDCN*, pages 378–392, 2013.

[78] Gokarna Sharma and Costas Busch. Distributed transactional memory for general networks. *Distrib. Comput.*, Preprint:1–34, 2014.

[79] Gokarna Sharma, Costas Busch, and Srinivas Srinivasagopalan. Distributed transactional memory for general networks. In *IPDPS*, pages 1045–1056, 2012.

[80] Gokarna Sharma, Brett Estrade, and Costas Busch. Window-based greedy contention management for transactional memory. In *DISC*, pages 64–78, 2010.

[81] Nir Shavit and Dan Touitou. Software transactional memory. *Distrib. Comput.*, 10(2):99–116, 1997.

[82] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP*, pages 141–150, 2009.

[83] Michael F. Spear, Virendra J. Marathe, William N. Scherer, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *DISC*, pages 179–193, 2006.

[84] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of blue gene/q hardware support for transactional memories. In *PACT*, pages 127–136, 2012.

[85] Ruibo Wang, Kai Lu, and Xicheng Lu. Investigating transactional memory performance on ccnuma machines. In *HPDC*, pages 67–68, 2009.

[86] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, and Depei Qian. Software transactional memory for gpu architectures. In *CGO*, pages 1:1–1:10, 2014.

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

[88] Bo Zhang and Binoy Ravindran. Relay: A cache-coherence protocol for distributed transactional memory. In *OPODIS*, pages 48–53, 2009.

[89] Bo Zhang and Binoy Ravindran. Dynamic analysis of the relay cache-coherence protocol for distributed transactional memory. In *IPDPS*, pages 1–11, 2010.