Investigating Software Transactional Memory on Clusters

Christos Kotselidis Mohammad Ansari Kimberly Jarvis Mikel Lujan
Chris Kirkham Ian Watson
School of Computer Science
The University of Manchester
Oxford Road, M13 9PL, Manchester
{kotselidis, ansari, jarvis, mikel, chris, watson}@cs.manchester.ac.uk

Abstract

Traditional parallel programming models achieve synchronization with error-prone and complex-to-debug constructs such as locks and barriers. Transactional Memory (TM) is a promising new parallel programming abstraction that replaces conventional locks with critical sections expressed as transactions. Most TM research has focused on single address space parallel machines, leaving the area of distributed systems unexplored. In this paper we introduce a flexible Java Software TM (STM) to enable evaluation and prototyping of TM protocols on clusters. Our STM builds on top of the ProActive framework and has as an underlying transactional engine the state-of-the-art DSTM2. It does not rely on software or hardware distributed shared memory for the execution. This follows the transactional semantics at object granularity level and its feasibility is evaluated with non-trivial TM-specific benchmarks.

1. Introduction

The advent of multicore chips provides huge incentives for the development of easy-to-use parallel programming models. In this direction and borrowing from the success in databases, Transactional Memory (TM) [9] is gathering momentum. Until now, synchronization in parallel applications has been achieved by the use of mutually exclusive locks and barriers. The problems of using such mechanism are the limited scalability, unless fine-grained locking is implemented, and the difficulty in programming. TM promises to address these problems by replacing these synchronization mechanism with atomic regions executed transactionally. These transactions are speculatively executed in parallel with the read and write operations generating an associated read and write set. At some stage of the transaction a validation phase reveals any conflicts with (non-empty intersections with the read set and write set of) other concurrently executing transactions. In that stage, a conflict resolution policy is employed to determine whether the transaction is aborted or delayed.

The research community has developed numerous TM systems divided into Software (STMs) [7, 5] and Hardware (HTMs) [4, 1] or Hybrid Software/Hardware (HyTMs) [3] solutions to understand TM behavior and how to implement it. The majority of these TM systems, however, focus on shared-memory parallel architectures leaving unexplored the domain of distributed systems. Limited research has been conducted on utilizing TM on clusters. The only related published work that tackles distributed execution is Distributed MultiVersioning (DMV) [7]. Transactional execution is achieved by modifying the underlying software Distributed Shared Memory system (DSM) [8]. The role of the DSM is to provide a shared memory view among the nodes of the cluster.

Our framework, the Distributed Dynamic Software Transactional Memory System (DDSTM), differs significantly from DMV in the sense that it does not rely on any DSM mechanism to achieve memory coherence. DDSTM employs transactional semantics at object granularity instead of page granularity as DMV. The contributions of this paper are:

- The first Java-based distributed STM system.
- The first distributed STM that does not rely on any underlying software or hardware memory consistency mechanism. Memory is distributed across the nodes of the cluster and the TM system is responsible for maintaining memory coherence.
- The first evaluation of distributed TM behaviour using non-trivial TM-specific benchmarks that have recently emerged in the literature.

The current DDSTM has been built for flexibility not for performance. At this stage it is important to have a platform

to establish feasibility and on which to quickly prototype different transactional protocols, contention managers, etc. appropriate for cluster-based TM execution.Remote communication across nodes is based on the ProActive framework [2] while the state-of-the-art DSTM2 [5] STM implementation executes transactions locally on the nodes. DDSTM is evaluated on a 32 processor cluster with the 3 benchmarks from the STAMP suite and a complex TM application which implements Lee's routing algorithm for circuit boards. This evaluation provides a first insight into transactional execution on clusters.

The remainder of the paper is organized as follows: Section 2 describes the core components of DDSTM. Section 3 describes the implemented distributed transactional protocol. Section 4 describes the platform as well as the benchmarks used to evaluate our system. Section 5 presents the experimental results and Section 6 summarizes the paper.

2. Distributed Dynamic Software Transactional Memory

DDSTM is written entirely in Java and it builds on two core components: the underlying transactional execution engine and the remote communication system. As a transactional engine we have adopted an extended version of DSTM2 [5]. The remote communication is based on the ProActive Grid framework [2] which is a high level API for Java RMI. The following subsections describe further how these components are integrated.

2.1. DSTM2 Transactional Engine

DSTM2 [5, 6] is a software TM system entirely written in Java which supports transactional execution for dynamically-sized data structures. All transactions are executed speculatively. When a transaction attempts to modify an object, instead of modifying directly the actual object, a cloned version of the object is used and kept private until it is safe for the transaction to commit. The commit phase follows a validation phase where any conflicts are detected and resolved. The validation phase reveals any conflicts (write-after-write or read-after-write). Upon conflict detection, a contention manager is consulted in order to resolve the conflict by aborting or delaying one or more of the conflicting transactions. After the validation phase finishes the "winning" transaction can safely commit; making public its changes (replacing shared objects with their respective modified clone objects). DSTM2 employs an obstruction-free synchronization policy which guarantees forward progress as any halted threads do not prevent active threads from making progress. However, the obstruction-free synchronization policy [6] does not prevent active threads from causing livelocks.

We selected DSTM2 because it offers a wide array of contention managers and atomic factories (mechanisms that describe how transactions are synchronized). Furthermore, it allows the user to plug-in custom managers and factories making it an ideal platform for experimentation purposes. Due to space limitation, we cannot expand more on the details DSTM2. Further information can be retrieved from [5].

2.1.1 Extensions

We have added distributed functionality to DSTM2 with minimal changes to its architecture. The two main modifications regard the way transactions commit and the way objects are identified amongst the nodes of the cluster.

Initially in DSTM2, the commit stage included two phases: the validation() and the commit() phases. Upon validation, potential conflicts are discovered and if none, the transactions attempt to commit by CompareAndSwaping (CASing) their status flags from ACTIVE to COMMIT. In that scheme an extra step has been added which validates the transaction against the transactions executing on remote nodes. The remoteValidate() function broadcasts each transaction's read and write sets (these sets contain the objects' identifiers (see next paragraph) read/written from the transactions) to the rest of the nodes. The objects contained in the read and write sets are validated against the objects contained in the write and read sets of the transactions currently executing on the remote nodes. There are two feasible results of conflict detection. Either the incoming transaction (the transaction that broadcasts its sets in order to be validated) is aborted (remoteValidate() returns false) or all conflicting local transactions (the transactions of the remote node which are validated against the incoming one) of the node are aborted (remoteValidate() returns true).

Each node of the cluster runs an instance of the Java Virtual Machine (JVM). Transactional objects reside on each node and each node holds a copy of the working transactional dataset (objects accessed transactionally from threads). Storing the hashcodes of objects in the transactions' read and write sets and comparing them between distributed transactions does not guarantee correct validation. The fact that objects are created on different JVMs results in the creation of different hashcodes. Therefore a transactional object A at node 1 may have a different hashcode at node 2 using the default Java hashcode implementation. Consequently, if a transaction writes that object and in turn attempts to validate its write set against transactions running on other nodes, the system must ensure that object A will be identified as the same. To achieve that, an indexing technique had to be added. Each entry in a transaction's read/write set has a pair of its hashcode and its index in the

data structure used as well as the value of the object read or written. Validation is achieved by checking the values of the pairs, which map indexes with hashcodes, of the incoming transaction against those of the residing local transactions of a node ensuring correct validation.

2.2. Remote Communication

Remote communication enables distributed execution of transactions over the multiple instances of DSTM2 running on each node of the cluster. The remote communication layer is entirely written in Java as it is based on the ProActive framework [2]. The key concept of the ProActive framework is the notion of "Active Objects". Each active object has its own thread of execution and can be distributed over the network. Based on this primitive, each node has a number of active objects serving various requests. The architecture used at the first implementation of DDSTM is master centric with the master node coordinating the execution on the cluster. Upon bootstrap, a VM is created on every node, including the master node. DDSTM begins execution on the main thread (master node) creating the necessary structures on the remaining nodes. The main thread on the master node is an active object in order to serve requests from the worker nodes. The VM on each worker node has two active objects. The first one (an instance of the `DDSTMClient` class) is the main thread which coordinates the execution on the node. In addition it is responsible for updating the worker node's datasets upon a transaction's commit, maintaining consistency among the various copies of the datasets residing on the cluster. The second one residing on each of the worker nodes is the `Provalidator` active object. The role of the `ProValidator` object is to accept transactions from other nodes wanting to be validated against the transactions running on that node. Figure 1 depicts DDSTM's architecture.

3. A Centralized Distributed Transactional Protocol

The master node initializes the worker nodes and their active threads. In turn, the worker nodes initialize the worker threads of each node and wait for incoming tasks to execute. After the tasks from the master are enqueued on the worker nodes, Fig. 1 (1), threads begin executing them. Each thread executes one transaction at a time. Transactions or threads do not spawn or migrate over multiple nodes. Local conflicts (i.e. between transactions executed within the same node) are resolved by DSTM2's underlying mechanisms. When a transaction attempts to commit, it first has to validate itself against the locally executed transactions, then against all distributed transactions of the cluster, and finally commit. We selected the Transactional Coherence and Consistency (TCC) [4] validation protocol as it performs lazy validation (each transaction attempts to validate its read/write set only once; during an arbitration phase before the commit stage). The aim is to minimize the expensive broadcasting action for validation purposes – only once at commit. To maintain coherence, transactions acquire a "ticket" (global serialization number) from the master node before they start remote validation. This "ticket" assists in the conflict resolution policy adopted in the first version of DDSTM. We have adopted the policy of the oldest-commit-first (oldest in terms of remote validation time - which transaction attempts to remotely validate its read/write sets first).

Upon remote validation, Fig. 1 (2), a transaction's read and write set (incoming transaction) are compared against the read and write sets of the transactions executed on a remote node (local transactions) resulting in three possible scenarios:

1. **There is no conflict** — No transaction is aborted and the `remoteValidation()` method returns true, so the caller can proceed committing its transaction.

2. **There is a conflict with a "younger" local transaction** — In that case, a conflict is detected against a transaction which has a greater "ticket" number than the incoming transaction. That means that the incoming transaction has acquired the ticket before the transaction it is validated against (local transaction of the remote node). The local transaction is considered to be "younger" and therefore it is aborted. Instead of aborting the transaction immediately, we store its id in a temporary buffer. Each transaction with a larger "ticket" will be stored in the buffer and will be aborted only when the validation of the incoming transaction has finished and there has been no conflict with a local transaction with a smaller "ticket" number. The validation phase is performed serially. Each transaction's read/write sets are validated serially against the read/write sets of other transactions. Furthermore, all transactions which attempt to be validated against the transactions running on another node are queuing up and each one performs the `remoteValidate()` function serially. Therefore, there might be the case where the first transaction to be validated against is younger while the second one is older. If we were to abort the first transaction immediately then the incoming transaction could be aborted by the second (older) transaction and, hence, we would have falsely aborted the first one.
   If the validation phase finishes and the incoming transaction has not been aborted by any older one on the remote node, the transactions stored in the temporary buffer are aborted and the `remoteValidation()` method returns true so the caller can proceed in com-

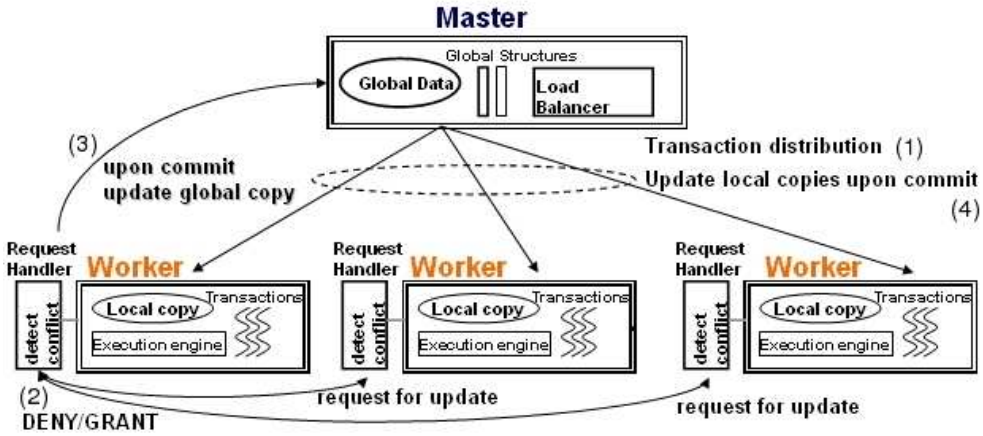

Figure 1. DDSTM's architecture.

mitting its transaction.

3. **There is a conflict with an "older" transaction** — In that case, a conflict is detected against a transaction which has a smaller "ticket" number than the incoming transaction, i.e. it is older and should abort the incoming transaction. In that case, the `remoteValidate()` function returns false and the buffer with the transactions to be aborted is released.

If the transaction successfully passes the validation phases, it attempts to commit locally and then globally, Fig. 1 (3). After having committed locally, all the transactions running at the same node can access the up-to-date copy of the dataset. On the other hand any other transactions running on a different node may access "dirty" values. Therefore in order to preserve memory consistency, the committed transaction must make its changes available to the rest of the nodes. There may be various ways of achieving that. In our first implementation we adopted a master-centric, eager approach. The committed transaction updates the global dataset at the master node and in turn the master node eagerly forces the remaining nodes to fetch the new dataset (4). Upon fetching the new data, an eager validation phase takes place aborting any conflicting transactions. The role of this eager validation phase is to discover if any transactions at any node have started executing before the commited transaction makes its changes visible to the remaining of the nodes. If any transaction has read any object modified by the committed transaction it is aborted and re-executed after the node gets a consistent view of the data.

4. Experimental Platform

The hardware platform used in our experimentation is a cluster with 40 cores in total. We use five nodes, the master node and four others. The master node has 2 dual-core AMD Opteron CPUs at 2.4GHz with 8GB of RAM. All the remaining worker nodes are 4-way nodes with 4 dual core AMD Opterons at 2.4GHz with 16GB of RAM each. Each worker node has 8 cores and thus a maximum of 8 threads (excluding the threads of the "active objects") are spawned (to keep thread-switching to a minimum). By using the cluster's four nodes we create from 1 to 8 threads per run utilizing in total from 4 (one thread per node) to 32 (8 threads per node) execution threads. All the nodes run OpenSuse 10.1 and Sun Java6 build 1.6.0-b105 with maximum heap size set to 8GB.

Common microbenchmarks used for TM systems' evaluation on CMPs are not adequate for evaluating a distributed TM system because they are not computationally intensive and the majority of the time spent in executing transactions would be spent on remote requests. To confirm our assumptions a List benchmark has been tested and it spent 99% of its execution time on remote requests resulting in poor performance.

Benchmarks for TM systems are still few. Lock-based parallel applications translated to transactional ones do not seem adequate. Complex benchmarks for stretching TM systems have only recently emerged in the literature. In order to evaluate our system, the benchmarks of the STAMP benchmark suite [3] as well as Lee's transactional routing algorithm [10] have been ported to our system. In total four benchmarks have been used to evaluate our system: Lee's algorithm, KMeans, Vacation and Genome.

Lee's algorithm is the classic algorithm of laying routes on a circuit board. Each thread attempts to lay a route on the mainboard. Conflicts occur when two threads try to write the same point on the circuit. A real mainboard configuration of 1506 routes is used the evaluation.

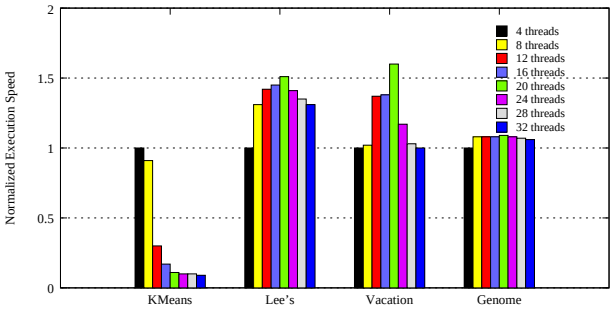

Figure 2. DDSTM's normalized performance results.

KMeans is a clustering algorithm where a number of objects with numerous attributes are partitioned into a number of clusters. Conflicts occur when two threads attempt to insert objects into the same partition. Varying the number of partitions affects the amount of contention.

Vacation is a simulator of an enterprise server. It is similar to the SpecJBB benchmark. Several threads acting as clients try to book, view, edit their records while performing actions of renting cars or booking flights or hotel rooms.

Genome performs gene sequencing from randomly generated segments. Conflicts occur when different threads try to use the same segment during the segment matching phase.

Concerning the transactional engine, we used the Priority contention manager (older always commit) and the obstruction freedom synchronization policy (in regards to local transaction execution). DSTM2 offers a wide range of contention managers, but as a first step we decided to be consistent with the conflict resolution policy used during the validation of distributed transactions (older always commit).

5. Experimental Results

This section contains the results of evaluating DDSTM against the mentioned benchmarks. The results shown are the averages of ten iterations. Figure 2 illustrates the normalized execution time over the increasing number of threads with respect to executing with one thread per node.

A first observation that can be drawn from the results is the limited scalability. The best performance is observed on Lee's and Vacation benchmarks. The maximum speed-up achieved overall is 60% for Vacation and 51% for Lee's-both at 20 threads (5 threads per node excluding the "active" threads). In general, all benchmarks except KMeans gain in performance up to a certain point. After they reach their peak, performance starts degrading.

Regarding Genome, any improvement of its performance is limited to 9%. To understand where the time is being spent, we split it down in Figure 3. The time spent at each step of a transaction's lifetime has been measured and averaged in order to separate the amount of execution time spent for local actions (local validation and commit) from that for remote operations (remote validation and commit). Finally, execution time contains the pure time a transaction spent in computation.

The cause of the deterioration in the performance of KMeans by up to 90% is the high abort rate among transactions. This causes transactions to be invalidated in a high rate and consequently be serialized resulting in poor performance.

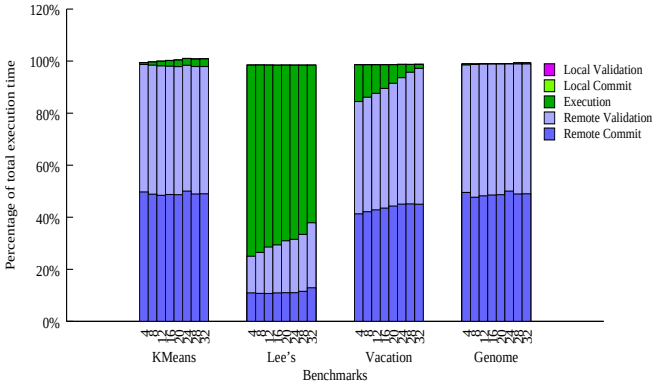

Figure 3. Percentages of time spent on transcations' stages.

As shown in Figure 3 Lee's is the only benchmark to perform heavy computations. In general all benchmarks from the STAMP suite utilize "atomic" data structures and parallelism is gained from performing concurrent actions on them. Inserting, deleting or looking up elements from a hash table or a list are performed in parallel and the level of contention is determined by the sizes of these structures as well as the number of concurrent threads accessing them. Lee's benchmark threads perform heavy computations along with the transactional access of the data structures used in it. We can see that while executing with 4 threads (one thread per node) the percentage of time spent in executing is 75% dropping gradually to 60% when executing with 32 threads (8 threads per node). The reason behind this is that when executing with more threads, a transaction must spent more time in remote validation. However, we still gain in performance in Lee's benchmark.

On the contrary, the performance of the STAMP benchmarks is dominated by network traffic and remote requests.

Therefore performance, especially in Genome is identical no matter the number of threads used. On the other hand because of the high contention KMeans' performance drops dramatically while Vacation's performance improves due to low contention. If remote requests' percentages are aggregated, we can notice that over 95% of the time is spent on remote requests. Therefore we do not gain much by executing non-computational intensive benchmarks on the cluster.

Currently, remote validation is performed serially. All transactions that want to be validated against a remote node's transaction are queuing up in a buffer and are validated serially. Furthermore, upon commit, when transactions attempt to update the global copy of the dataset again they are queuing up. These two bottlenecks are also responsible for the limited scalability. Trying to achieve a memory consistent system as a first version of DDSTM, we have been conservative in some of our design choices.

6. Conclusions

Research on TM for shared memory chip multiprocessors has been ongoing for some years. Clusters, being a core part of high performance computing and enterprise applications, remain open for evaluation with TM. To this end, the flexible Java-based DDSTM has been designed and implemented.

DDSTM has been evaluated with the most complex and established TM-specific benchmarks existing in the literature giving a first insight into transactional execution on distributed systems. Our first experimental results underline the important role remote communication plays in transactions' validation phases. Aggregate remote requests may vary from 25% to 99.9% influencing dramatically the performance of the system. Computationally intensive applications with loosely coupled datasets seem to benefit most from distributed transactional execution. For example, in Lee's benchmark we observe a 50% time improvement. Non-computationally-intensive benchmarks however seem not to benefit at all as their time totally depends on the network traffic and the serialization bottleneck points upon remote validation and commit.

Concerning future work, the two bottlenecks we have already spotted will be tackled in order to enhance DDSTM's performance. Having achieved a flexible and stable underlying platform for TM on clusters, we can start experiments to optimize it. Lazy and optimistic validation techniques will also be employed and compared with the current eager ones in order to identify suitable configurations.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, 2005.

[2] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, 2006.

[3] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.

[4] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. *ISCA*, 2004.

[5] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. *OOPSLA*, pages 253–262, 2006.

[6] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, 2003.

[7] M. M. Kaloian Manassiev and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 198–208, 2006.

[8] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[9] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[10] I. Watson, C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *Proceedings of the Sixteenth International Conference on Parallel Architectures and Compilation Techniques PACT-2007*, September 2007.