Clustering JVMs with Software Transactional Memory Support

Christos Kotselidis*, Mikel Luján*, Mohammad Ansari†, Konstantinos Malakasis*,
Behram Kahn*, Chris Kirkham* and Ian Watson*
*School of Computer Science, The University of Manchester, Manchester UK
{kotselidis, mikel.lujan, malakasis, kahn, kirkham, watson}@cs.manchester.ac.uk
†Department of Computer Science, Umm Al-Qura University, Makkah, Saudi Arabia
mmansari@uqu.edu.sa

***Abstract*—Affordable transparent clustering solutions to scale non-HPC applications on commodity clusters (such as Terracotta) are emerging for Java Virtual Machines (JVMs). Working in this direction, we propose the Anaconda framework as a research platform to investigate the role Transactional Memory (TM) can play in this domain. Anaconda is a software transactional memory framework that supports clustering of multiple off-the-shelf JVMs on commodity clusters.**

The main focus of Anaconda is to investigate the implementation of Java synchronization primitives on clusters by relying on Transactional Memory. The traditional lock based Java primitives are replaced by memory transactions and the framework is responsible for ensuring transactional coherence.

The contribution of this paper is to investigate which kind of TM coherency protocol can be used in this domain and compare the Anaconda framework against the state-of-the-art Terracotta clustering technology. Furthermore, Anaconda tracks TM conflicts at object granularity and provides distributed object replication and caching mechanisms. It supports existing TM coherence protocols while adding a novel decentralized protocol.

The performance evaluation compares Anaconda against three existing TM protocols. Two of these are centralized, while the other is decentralized. In addition, we compare Anaconda against lock-based (coarse, medium grain) implementations of the benchmarks running on Terracotta. Anaconda's performance varies amongst benchmarks, outperforming by 40 to 70% existing TM protocols. Compared to Terracotta, Anaconda exhibits from 19x speedup to 10x slowdown depending on the benchmark's characteristics.

***Keywords*-Distributed Software Transactional Memory; Java Virtual Machine Clustering; Multithreading; Parallel Computing**

I. Introduction

Developing concurrent lock-based applications is known to be challenging, which often leads to problems such as deadlocks, data races, and convoying. These problems can be difficult to debug and thus developing and maintaining such concurrent applications can be time consuming.

Transactional Memory (TM) [1] has gathered momentum as an alternative model to lock-based parallel programming. Borrowing from database transactions, TM requires developers to label blocks of code that access shared data as transactions. In exchange, TM guarantees that a transactional block will execute atomically, and in isolation from other transactions, leaving the program state consistent. There are two major functions required to implement transactions on top of a multi-threaded multi-core system. The first is the ability to handle both committed and uncommitted data while a transaction is executing (memory versioning). The second is the detection of interference between transactions (conflict detection) and the control of which threads are allowed to commit their state and which threads need to be aborted and restarted (contention management). This is based on an observation of the intersection between the read and write sets of concurrent transactions. Research in TM has developed several TM systems divided into Software TM (STMs), Hardware (HTMs) and Hybrid Software/Hardware (HyTMs) [1] in order to understand TM behavior and reduce the overhead of TM runtime checking. The majority of these TM systems, however, focus on shared-memory parallel architectures resulting in limited work in the cluster domain [2], [3]. Deploying TM on clusters is attractive in clustering Java Virtual Machines (JVMs) scenarios, as

exemplified by Terracotta [4]. The execution behavior of TM on clusters differs significantly from that on multi-core processors, particularly due to the expensive inter-node communication messages that may need to be exchanged. This increases the challenge in developing a high performance TM system for clusters: not only must the intra-node TM overheads be minimized, similar to existing shared-memory TM systems, but also the inter-node TM coherence protocol must be designed to ensure it does not degrade scalability. Compared with traditional distributed shared memory, TM has in its favor a less demanding consistency semantics by only requiring it at commit points. TM has been investigated as both a programming abstraction, normally relying on the `atomic` construct, and as a means of implementing lock elision [5].

Our recent publication has described the Distributed Software Transactional Memory (DiSTM) [3], a flexible research platform addressing STM for clusters. DiSTM is designed and built to easily prototype different TM components and TM coherence protocols. Its evaluation compared two centralized and one decentralized TM coherence protocols. The analysis suggested that the centralized protocols performed better under high contention, as they reduced network congestion, while the decentralized protocol resulted in better performance under low contention as it avoided the bottlenecks of the centralized model. Nonetheless, the decentralized protocol was only marginally better under low contention. All the TM coherence protocols operated on object-granularity with eager local conflict resolution and lazy remote conflict resolution.

In this paper we introduce and evaluate Anaconda, a clustering JVM solution which integrates a novel decentralized TM coherence protocol. Both DiSTM and Anaconda are built on top of off-the-shelf JVMs and track TM conflicts at object granularity. However, only Anaconda provides distributed object replication and caching mechanisms.

The objective of the *Anaconda* TM coherence protocol is to minimize network traffic by employing a lazy local, lazy remote conflict detection, and lazy object versioning. As a contention manager policy with Anaconda we have selected "older transaction commits first" with a distributed unsynchronized means of generating unique timestamps. The distributed structures employed by Anaconda are the Transactional Object Buffer (TOB), which maintains per transaction meta-data, and the Transactional Object Cache (TOC) which maintains per node locally shared meta-data. The meta-data stored at the TOC describe the ownership of objects as well as which nodes have copies of a given object. The TOC can be viewed as a directory that indicates where the different copies are for an object, while the TOB represents a transaction's readset and writeset. Finally, the Anaconda TM protocol follows an update-upon-commit approach (i.e. all the cached versions of an object are updated upon commit).

The performance evaluation compares Anaconda with those three TM protocols evaluated in DiSTM. In addition, we compare Anaconda against coarse grain locking and medium grain locking implementations of benchmarks running on Terracotta. Anaconda's performance varies amongst benchmarks, outperforming by 40 to 70% DiSTM's protocols. Compared to Terracotta, Anaconda exhibits from 19x speedup to 10x slowdown depending on the benchmark's characteristics.

The structure of this paper is as follows. Section II describes related work in the area of TM for distributed systems. Section III describes the architecture of Anaconda, including object replication and caching mechanisms. Section IV describes the Anaconda decentralized coherence protocol. Section V presents the experimental platform used. Section VI presents the evaluation of Anaconda, and Section VII concludes the paper.

II. Related Work

Limited research has been carried out in executing TM on distributed systems. Existing work includes Distributed MultiVersioning (DMV) [6], Cluster-STM [2], and DiSTM [3]. DMV executes transactions on top of a Distributed Shared Memory (DSM) system [7] through modifications to its software-based shared memory layer. The DSM provides a shared memory view to all the processors in the cluster. Cluster-STM is a prototype implementation with limited support for transactional execution. The limitation arises from only being able to execute one transactional thread per node of the cluster. Therefore, it cannot take full advantage of nodes that are, e.g., dual or quad-core systems. DiSTM differs from DMV as it does not rely on a DSM layer to achieve coherence. DiSTM improves upon Cluster-STM as it resolves conflicts between a transaction and any other transactions executing locally or remotely, i.e. both local and remote validation. DiSTM

provides transactional semantics at object granularity, whereas DMV provides page granularity and ClusterSTM provides word granularity. DiSTM is modular and flexible for experimentation purposes.

In addition to TM research literature, new parallel programming languages are emerging to enable efficient parallel programming on clusters. The *Partitioned Global Address Space* (PGAS) languages such as UPC [8], Titanium [9], X10 [10], Chapel [11] and Fortress [12] allow parallel programming while providing a global address space. Some of the *PGAS* languages (such as X10, Fortress and Chapel) include TM-like *atomic* constructs without currently containing any underlying distributed TM system. The implementation of these constructs is undefined and is subject to future research.

Transactions are being investigated in distributed environments with Sinfonia [13]. Sinfonia uses mini-transactions as a replacement for message-passing in cluster file servers. Java Virtual Machines have spawned distributed solutions such as Terracotta [4] for enterprise applications, but these rely on locks for synchronization amongst threads. Anaconda provides an alternative distributed JVM platform based on transactional execution.

Finally, in the domain of distributed JVMs there have been some prototype implementations such as cJVM [14], dJVM [15] and Jessica [16], [17]. The first two are in early prototype stage whereas Jessica mainly focuses on thread migration techniques leaving the synchronization mechanisms unexplored.

III. Anaconda Architecture

Anaconda implements STM on clusters using one JVM per-node. Each node is a multi-core chip, running an off-the-shelf JVM, executing multiple threads. Anaconda ensures accesses to shared objects are performed safely by performing runtime checks at the intra-node and the inter-node level. An execution thread may access shared objects that may be located locally or on a remote node. Anaconda consists of three key components: the transactional runtime, the inter-node communication and the object caching and replication. They are described in detail below.

A. Transactional Runtime

Each node of the system has its own instance of a TM runtime that employs a TM coherence protocol to validate, commit or abort local or remote transactions. The Anaconda protocol has a unified mechanism for local and remote coherence while using object cache and replication structures (Section IV-A).

Transactional granularity is maintained at the object level. Transactional classes are declared as *@atomic* annotated Java interfaces. The user has to declare only the getter and setter methods of the transactional objects. The *@atomic* interfaces are bytecode-rewritten upon bootstrap in order to enable transactional functionality; similar to [18]. Furthermore, the preferred TM coherence protocol is defined as a plug-in and is being bytecode-engineered during this stage. A side effect is that Anaconda offers strong isolation as the bytecode rewritten objects will throw a Java *NullPointerException* upon a thread's attempt to access them outside the scope of a transaction.

B. Remote Communication

The communication amongst the nodes is achieved by the use of the ProActive [19] framework (high level API wrapper of RMI). The remote requests are invoked on the *active objects*. The *active objects* have their own thread of execution, can be distributed, and constitute the basic building blocks of the ProActive framework. Each node in the Anaconda framework has a number of active objects serving various requests. Those requests have been decoupled and logically assembled in different active objects in order to avoid bottlenecks. Generally, active objects serve one request at a time and hence congestion may occur. The decoupling of the remote requests in the Anaconda framework resulted in the creation of three active objects per node. Furthermore, a remote method invocation can either be synchronous or asynchronous. Depending on the protocol stage, Anaconda can utilize either synchronous or asynchronous requests to achieve higher performance.

C. Object Caching and Replication

Each transactional object in the cluster has a unique identification number (OID). The generation of OIDs is hidden underneath the collection classes provided by Anaconda. In addition, each object has a parent node identification number (NID) which is the node that first created that object. The OID is encapsulated as a field in every transactional object. This is done automatically to all objects declared as transactional (by *@atomic*). Transactional objects are simple serializable

POJOs (Plain Old Java Objects) that can be replicated and cached. Finally, each executed transaction has a global unique identifier TID which is the concatenation of a timestamp (assigned at the beginning of the transaction), the ID of the transaction's executing thread (threadID), and the NID. As described later, the TID is used for conflict resolution and by concatenating these three fields each transaction's TID is guaranteed to be unique.

Each node maintains a set of helper data structures when the Anaconda coherence protocol is employed: the Transactional Object Cache (TOC) and the Transactional Object Buffer (TOB). Each node maintains a single TOC that is shared by all threads executing on that node. The TOC provides caching functionality of remotely fetched transactional objects. Furthermore, it maintains book-keeping information of executing transactions. Figure 1 illustrates a TOC. The first field is the OID that maps to a particular entry. The second entry is the NID of the home node of the object assigned the OID. Unless the NID of an entry in the TOC is equal to that node, they are cached copies of objects residing on other nodes. Maintaining the NID of the objects can assist in identifying the home owners of every object cached. The Cache field maintains a list of all the nodes that have requested and retrieved a copy of an object (and consequently stored it in their caches). This information is used to assist conflict resolution (Section IV-A). The Lock TID field is a lock associated with each entry and it is acquired during transactions' commit stage. Finally, the Local TID field is a list of all the local transactions currently accessing this object.

The second data structure, the TOB, is maintained per transaction but is visible amongst transactions (Figure 2). After accessing an object for a write operation, a cloned copy of the object residing in the TOC is created and stored in the TOB. Thereafter read operations will be redirected to the cloned object version. The TOB actually serves the role of maintaining transactions' book-keeping information.

D. Distributed Atomic Collection Classes

Anaconda provides various collection classes for distribution. Currently, the classes provided are distributed arrays, distributed single objects and distributed hashmaps. The distributed arrays can be either declared to be cached as a whole to all nodes or to be partitioned amongst them. The partitioning can be achieved in various configurable ways such as horizontal, vertical or blocked.

OID	NID	Cache	Lock TID	Local TIDs
A,pA	1	15,16	9	54,35
		...		

Figure 1. Transactional Object Cache (TOC) structure

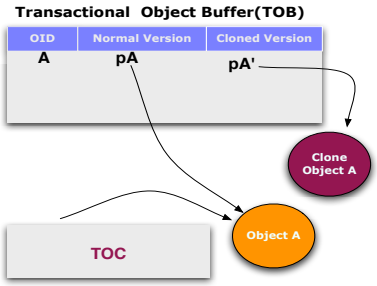

Figure 2. Transactional Object Buffer (TOB) structure

IV. Anaconda TM Coherence Protocol

This section describes the commit and abort procedures of the protocol in action by illustrating the TM interactions. The protocol uses a lazy versioning of modified objects and lazy local/remote validation. The lazy remote validation follows a pessimistic approach in which, if a transaction has detected a possible conflict, it will abort rather than wait for confirmation that the other conflicting transactions have successfully committed.

The Anaconda protocol has a three phase commit stage. As a transaction progresses through the phases it becomes more certain that there are no conflicts. During the first phase, the transaction interacts with the local TOC to ensure exclusive access to each object in its writeset. In the second phase, the objects in the writeset are multicast to those nodes that have cached copies of them. The idea is not to update yet, but to validate against the remote TOC. In the third final phase, the transaction cannot be aborted by any other transaction and can commit the new values stored in its local TOB.

A. Commit Process

Phase 1: Lock acquisition During this phase a transaction is required to acquire the locks for each

object in its writeset. The writeset of each transaction is processed and the objects contained in it are grouped according to their home nodes. Batch requests are sent to each node starting from the local node. This is done in order to save remote requests upon failed local lock acquisition. The response from the remote TOCs to each request contains a list of the nodes having a cached copy of any object in the writeset. Note that this phase is liable to run into deadlock if implemented naively, as the gathering of all object locks is no longer an atomic operation. A discussion of the rules used to overcome this issue is included in Section IV-C.

Phase 2: Validation phase Having successfully completed Phase 1, a transaction has the list of nodes which contain remotely cached versions of the objects that are part of its writeset. Thus, the modified objects (i.e., the OIDs as well as the new values) are multicast to the nodes of the list. Upon arrival in a remote node, a validation phase then takes place aborting any remote conflicting transactions. The validation phase concerns only the transactions pointed to by the Local TID field part of a remote affected TOC. When a conflict is detected, the TIDs are compared in order to decide which transaction should be aborted using the "older-commit-first" policy (i.e., the transaction with the larger TID is aborted). In order to enhance performance of the validation phase, bloom filters are utilized to encode the read-set of the transactions. In this way, we try to minimize the validation phase time as it is a blocking request against the transaction that performs this phase as well as the transactions queued waiting their turn to validate against this node. If the validating transaction is aborted it revokes any locks (if it had acquired any) and removes its TID from any entry in the TOC.

Phase 3: Update Objects Having successfully completed Phase 2, the transaction cannot be aborted by any other transactions and therefore can safely proceed in swapping or updating the old objects with the new ones. This can be done safely as the committing transaction still holds the locks of the objects being updated not allowing any other transactions to fetch and cache, neither read from nor write to them. Any such request will result in a negative acknowledgment by the TOC. The requesting transaction will continue to retry until it gets aborted or until the committing transaction releases the lock. Upon an object update the TOC is responsible for updating all the cached copies of this object. This can be done in different ways (invalidate vs. update protocol). In the invalidate protocol, the transactions have to discover by themselves any potentially stale object and consequently abort themselves, while in the update protocol the system eagerly patches all the cached values and eagerly aborts any conflicting transactions. In Anaconda we currently employ the eager-patching approach but we also plan to incorporate the invalidation protocol for comparative evaluation. Consequently, the new patches are being sent to the nodes that hold cached copies of the objects. In those nodes receiving the patches, the local executing transactions that are accessing those objects are validated against the incoming writeset. Any local conflicting transactions are aborted. Finally, the updating transaction revokes all locks and if this phase takes place at the node where the transaction belongs, the TID of the transaction is removed from any entries in the TOC.

B. Simple Commit Example

Figure 3 demonstrates a simple commit example of two transactions, T1 and T2 executing on Nodes 1 and 2 respectively. For clarity, in the example we assume that Nodes 1 and 2 are the home nodes of objects A and B respectively. In this example T1 commits successfully while T2 is aborted and subsequently restarted. In Step 1, T1 attempts to read object A with OID(A). As OID(A) is not present in T1's TOB, T1 checks if OID(A) exists in the current node by inquiring at TOC 1. After the successful inquiry at TOC 1, T1 stores a reference of OID(A) in its readset (marking the Bloom filter). Furthermore, T1 adds itself to OID(A) Local TIDs entry in the TOC. The same procedure takes place upon T2's effort to load object B. If the objects were not residing in the nodes, a remote request for fetching the object would be sent across the cluster.

In Step 2, T1 speculatively writes to object A by creating a private cloned copy pA' of object A and storing its reference in T1's TOB. At the same time, T2 reads object A by fetching it from Node 1. This request will add Node 1 to object's A Cached field in TOC 1. In this way the home node of any object is aware of any remote nodes that have fetched a copy of a particular object. This information will be used later to maintain TM coherence between the various cached copies of objects residing on different nodes.

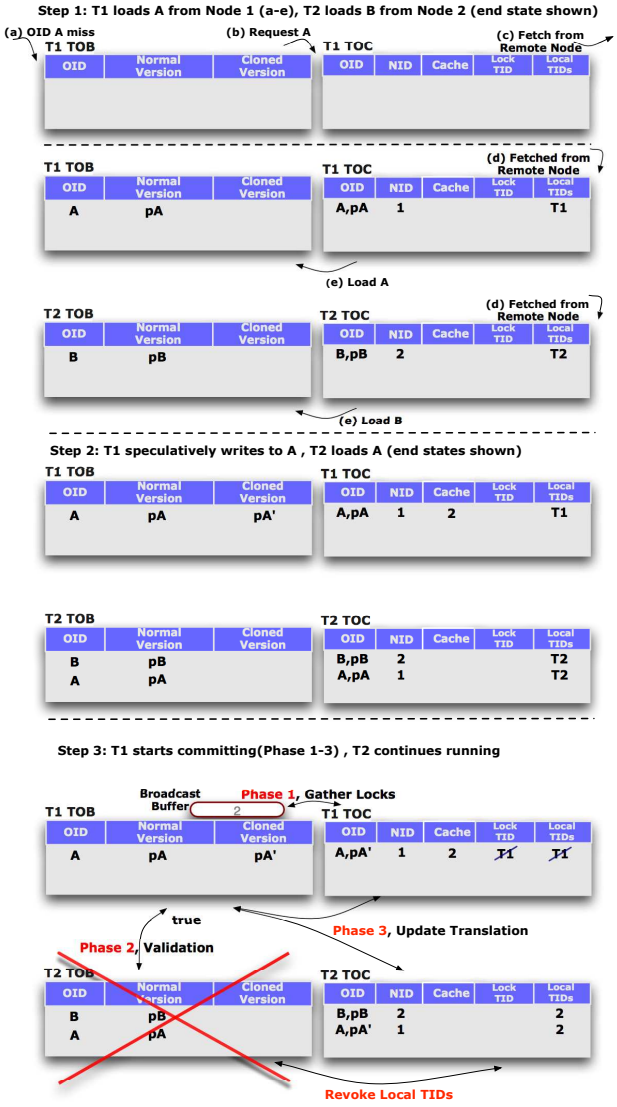

Figure 3. Simple Commit Example

In Step 3, T1 starts its three stage commit phase while T2 continues execution. During phase 1, T1 requests the lock for object A. Although a translation of OID(A) exists in both TOCs, the lock acquisition will take place only at the home node of OID(A). In this way, T1 gets the lock by adding itself to the Lock TID field for TOC 1. Any potential lock request to this particular lock will cause the contention manager to be invoked and one of the two transactions will be aborted. TOC 1 will respond to T1's request for OID(A) lock by sending back the list of the nodes that have a cached copy of OID(A). Hence, the return node list for T1 will contain Node 2 as it has a cached copy of object A. Upon successful acquisition of all locks, T1 proceeds to Phase 2. Phase 2 entails the multicast of T1's writeset to the node list received from Phase 1. A validation step will abort either any conflicting local transactions or T1. In this example T2 is aborted by T1 and acknowledges the verification. Upon receiving positive replies from phase 2, T1 proceeds in phase 3 by CompareAndSwapping (CASing) its status from ACTIVE to UPDATING. If the update of the flag is successful, no other transaction can abort T1 and it moves to the final phase. In Phase 3, T1 sends an update-objects request to the same nodes to which it multicast in Phase 2 (note that the objects themselves were already sent in Phase 2). When Node 2 receives the requests it replaces the old objects with the new ones and performs a validation check only to the transactions contained in the Local TID field of its OID(x) mapping contained in the updating dataset. In the meantime, the aborted transaction T2 releases any locks acquired, if any. Finally, both transactions revoke their TIDs for the corresponding Local TID fields of their TOCs.

C. Contention Cases

There are phases within the protocol where contention can occur. This section describes them and explains how the contention is resolved.

Lock Acquisition Contention phase The first phase where contention can occur is during commit phase 1. During this phase transactions are required to gather all locks sufficient to cover all modified objects and the locks are gathered in the order in which they appear in the TOB. It is possible that multiple transactions requiring multiple locks can enter a scenario similar to the deadlock achieved in the dining philosophers problem. A typical scenario might be: T1 holds lock for object A, needs lock for object B. T2 holds lock for object B, needs lock for object A. In such a scenario when T1 requests the lock for object B, the TOC containing that lock forwards a message to the owner (T2) informing it that the lock must be revoked, because T1 has a higher priority. T2 will release the lock and abort.

Multicast Contention During commit phase 3 it is possible that a transaction will multicast its intent to modify an object contained in another transaction's readset. If this causes a violation, i.e. it is not a false conflict, a contention manager will be invoked in order

to resolve the conflict. Anaconda allows the plug-in of different contention managers. After the contention manager invocation, only one transaction will continue in committing while the other will be aborted.

TOC trimming Upon successful lock acquisition of a transaction in the commit stage, the number of nodes to which the writeset of the committing transaction is multicast is determined by the number of nodes that have a cached version of the objects. The TOC has the responsibility to multicast the committing transaction's writeset to the corresponding nodes and the extra validation step will reveal any conflicts. Furthermore, the TOCs can grow large, slowing down any operations on them. The aforementioned two problems can be easily tackled by periodically trimming the TOC, i.e. removing records that have not been accessed lately.

V. Experimental Platform

A. Hardware

The hardware platform used for experiments is a cluster of 4 nodes. Each node has 4 dual core AMD Opteron processors at 2.4GHz. We run a maximum of 8 threads per node (excluding the threads of the "active objects"). For the centralized experiments one extra master node is used. The network interconnect is Gigabit ethernet. Each experiment creates from 1 to 8 threads per node (i.e. always utilizes all four nodes), and thus experiments use from 4 threads (1 thread per node) to a maximum of 32 threads (8 threads per node). All the nodes run OpenSuse 10.1, Sun Java 6 build 1.6.0-b105 with maximum heap size set to 8GB and Terracotta 2.7.3.

B. Benchmarks

We have selected three different benchmarks which exercise different parts of a TM system. Table I presents general characteristics of these benchmarks.

LeeTM offers long transactions and low contention. KMeans is placed at the other end of the spectrum with very short transactions and high levels of conflicts (aborts). GlifeTM also provides short transactions but with low levels of conflicts instead. More specifically:

LeeTM [20], [21] implements the classic Lee's algorithm for laying routes on a circuit board. Each transaction attempts to lay a route on the board. Conflicts occur when two transactions try to write the same cell in the circuit board. A real circuit of 1506 routes is used in the evaluation. LeeTM algorithm is also included in the STAMP benchmark suite [22] as *Labyrinth*, although using random configurations. Furthermore, the configuration of LeeTM used, employs the *early-release* [23] optimization that reduces the contention [24].

KMeans [22] is a clustering algorithm where a number of objects with numerous attributes are partitioned into a number of clusters. Conflicts occur when two transactions attempt to insert objects into the same cluster. Varying the number of clusters affects the amount of contention. Two configurations of KMeans are used (KmeansHigh, KMeansLow). KMeansHigh attempts to cluster the objects into 20 clusters resulting in high contention. By contrast, KMeansLow attempts to cluster the objects into 40 clusters resulting in lower contention. Both configurations cluster 10000 objects of 12 attributes.

GLifeTM [25] is a cellular automaton which applies the rules of Conway's Game of Life. Conflicts occur when two transactions try to modify concurrently the same cell of the grid. Parameters used: columns:100, rows:100, generations:10.

C. Protocols

The protocols used for the comparative evaluation of *Anaconda* are the TCC, Serialization Lease and Multiple Leases protocols of DiSTM as well as coarse and/or medium grain lock based implementations of the benchmarks running on the state-of-the-art clustering software Terracotta:

TCC performs eager local and lazy remote validation of transactions that attempt to commit. Each committing transaction, broadcasts its read/write sets only once, during an arbitration phase before committing. All other transactions executed concurrently compare their read/write sets with those of the committing transaction and if a conflict is detected, one of the conflicting transactions aborts in order for the other to commit safely (Contention Manager invocation).

Serialization Lease The logic behind that protocol is the use of a lease in order to serialize the transactions' commits over the network. In this way, the expensive broadcasting of transactions' read/write sets for validation purposes can be avoided. The lease acquisition takes place after a successful local validation of a transaction. In turn, the lease release takes place after the transaction, that owns the lease, commits.

Configuration Name	Application	Configuration
LeeTM	Lee with early release	early_release:true, input_file:mainboard, 600x600x2 circuit with 1506 transactions
KMeansHigh	KMeans with high contention	min_clusters:20, max_clusters:20, threshold:0.05, input_file:random10000_12
KMeansLow	KMeans with low contention	min_clusters:40, max_clusters:40, threshold:0.05, threshold:0.05, input_file:random10000_12
GLifeTM	Game of Life	grid size:100x100, generations:10

Table I
BENCHMARKS' PARAMETERS

After that, it is the system's responsibility to assign the lease to the next waiting transaction.

Multiple Leases In contrast to the serialization-lease scheme, in the multiple-leases scheme multiple leases can be assigned to committing transactions. To ensure correctness, an extra validation step is performed upon acquiring the leases.

Lock-based LeeTM and GLifeTM have both coarse and medium grain locking implementations while KMeans has only a coarse grain locking implementation. In the coarse-grain locking configuration, all shared data structures are guarded by distributed locks. In the medium-grain locking implementations (LeeTM and GLifeTM), the shared data structures (distributed arrays) have been partitioned in blocks guarded by distinct locks. In addition, measures to avoid deadlocks and to ensure correctness have been taken.

The evaluation of GLifeTM includes only the *Anaconda* and the lock-based implementations. We omitted direct comparison with DiSTM due to the fact that the internals of the benchmark are different influencing the results to a great extent.

Figure 4 presents the graphs with the execution times of LeeTM, KMeans and GLifeTM over the *Anaconda*, TCC, Serialization Lease, Multiple Leases and the Terracotta lock-based implementations. The reported execution times of the experiments are the averages of 10 runs.

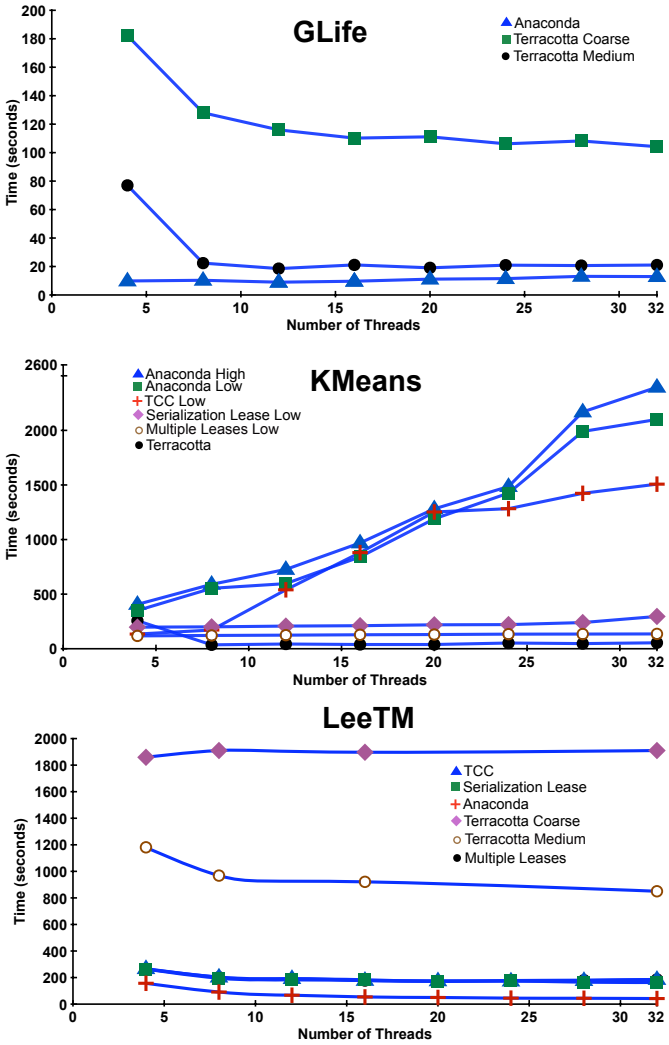

Figure 4. Benchmark's execution times

VI. EVALUATION

Concerning LeeTM, *Anaconda* outperforms all protocols. More specifically, it is: 19x faster than Terracotta coarse-grain locking, 10x faster than Terracotta medium-grain locking, and 40% to 70% faster than the TCC, Serialization Lease and Multiple Leases protocols. Transactions in LeeTM, as Table III shows for Anaconda, spent the majority of their time in computations (63 to 75%). This suggests that LeeTM can benefit from distributed execution as it spends small percentages of time in remote requests. *Anaconda* scales similarly to the rest of the TM coherence

protocols, suggesting that it has significantly lower transactional overhead. TCC, a decentralized protocol, gives similar performance to the lease-based protocols, which are centralized. The locking implementations ported to Terracotta perform poorly due to two reasons: a) serialized execution, as each attempt to lay a route (by reading/writing from/to grid cells) results in either acquiring a lock for the whole grid or acquiring locks over grid partitions, and b) the overhead of the Terracotta distribution mechanisms. The transactions of LeeTM are fairly large [24] and therefore every access to a grid cell results in memory coherence actions taken by the Terracotta infrastructure. On the other hand, in the *Anaconda* protocol due to lazy validation and the smaller sizes of the transactions (*early-release*) these costs are minimized, resulting in better performance.

In KMeans, *Anaconda* along with TCC are significantly slower than the lock-based and the leases protocols (5x to 10x). In high contention scenarios the centralized protocols outperform the decentralized ones. No matter the configuration used in KMeans (KMeansHigh, KMeansLow), the number of aborts still remains significantly high as shown in Table VIII. This is due to a single atomic counter (globalDelta) of KMeans which performs checks over the specified threshold. This object is shared among all threads executing on the cluster. In combination with the fact that KMeans' transactions are really small as shown in Table VII, transactions spend the majority of their time in remote requests, Table II. This, in turn, adds traffic over the network slowing down the execution.

	Number of Threads							
	4	8	12	16	20	24	28	32
Avg % Execution	3	3	2	2	4	3	3	2
Avg % Lock Acquisitions	21	22	23	24	26	27	26	26
Avg % Validation Phase	40	40	42	43	45	46	47	46
Avg % Updating Objects	36	35	33	33	25	24	24	26

Table II
KMEANSLOW EXECUTION TIME PERCENTAGES BREAKDOWN INTO TRANSACTION STAGES

Anaconda on average performs worse than the TCC protocol while the lock-based Terracotta port is faster (2x) than the lease-based protocols. The small size of KMean's transactions do not have a great impact on Terracotta. In GLifeTM, *Anaconda* gains approximately 70% in execution time over 32 threads while it is 5x slower than the lock-based Terracotta programs. This is due to having both a small number of aborts (Table V) and small transactions (Table IV). Terracotta does not scale while increasing threads, but the approximately constant execution time is faster than *Anaconda*. The transactional overhead of *Anaconda* over small transactions is comparatively higher than Terracotta. As shown in Table III, LeeTM is a computationally intensive benchmark with execution percentages varying from 63 to 75%. As the number of threads increases the percentage of time spent for remote requests increases due to network traffic. Table VI shows the average time spent per transaction (Tx Total Time). This amount of time is approximately spent in: a) computation time of the transaction (Tx Execution Time), and b) time spent during the commit stage (remote messages). While adding more threads, the transaction's total times are increasing in both ways. The execution time increases due to increased CPU utilization while the commit time increases due to increased network traffic.

	Number of Threads							
	4	8	12	16	20	24	28	32
Avg % Execution	75	74	72	71	70	68	66	63
Avg % Lock Acquisitions	12	13	13	14	14	14	15	15
Avg % Validation Phase	7	7	8	8	9	10	10	11
Avg % Updating Objects	6	6	7	7	7	8	9	11

Table III
LEETM EXECUTION TIME PERCENTAGES BREAKDOWN INTO TRANSACTION STAGES

Tables II and VII show the percentages and actual time spent in KMeansLow transactions at the Anaconda protocol. Transactions spend the majority of their time in remote requests (over 96%) because of the small size of transactions (0.5-3.6ms). In combination with the shared counter (globalDelta), the number of aborts increases dramatically while increasing the number of threads (from 91K to 712k over an average of 46K commits). This results in the poor performance of the decentralized protocols compared to the centralized ones as well as the lock based implementations. Tables IV shows the times spent in GLifeTM transactions at the Anaconda protocol. Transactions are very small (4.8-8.8ms.) and spend the majority of their time in remote requests, similar to KMeansLow. In contrast to KMeansLow, the number of aborts is small varying from 790 to 4953 over a constant amount of 10K commits.

	Number of Threads							
	4	8	12	16	20	24	28	32
Number of Commits	100000	100000	100000	100000	100000	100000	100000	100000
Number of Aborts	790	1966	3007	3592	3565	4078	4363	4953

Table V
ANACONDA GLIFETM NUMBER OF COMMITS AND ABORTS.

	Number of Threads							
	4	8	12	16	20	24	28	32
Avg. Tx Total Time	349.4	396.2	465.17	506.9	560.2	582.5	721.7	768.2
Avg. Tx Execution Time	294.4	315.3	341.2	374.4	368.4	378.7	409.7	428.4
Avg. Tx Commit Time	52.7	78.3	119.9	124.2	187.3	195.5	296.7	330.8

Table VI
ANACONDA LEETM TRANSACTIONS' EXECUTION TIMES (MS).

	Number of Threads							
	4	8	12	16	20	24	28	32
Number of Commits	53376	46704	40032	33360	46704	40032	45587	43675
Number of Aborts	91434	183432	256588	299466	556121	590376	630089	712987

Table VIII
ANACONDA KMEANSLOW NUMBER OF COMMITS AND ABORTS.

	Number of Threads							
	4	8	12	16	20	24	28	32
Avg. Tx Total Time	4.8	6.4	7.6	8.2	8.2	8.3	8.5	8.8
Avg. Tx Execution Time	0.1	0.1	0.2	0.2	0.3	0.3	0.4	0.4
Avg. Tx Commit Time	4.7	6.3	7.4	8	7.9	8	8.1	8.4

Table IV
ANACONDA GLIFETM TRANSACTIONS' EXECUTION TIMES (MS).

	Number of Threads							
	4	8	12	16	20	24	28	32
Avg. Tx Total Time	13.7	19.5	24.9	31.3	38	44.2	49.3	53.4
Avg. Tx Exec Time	0.5	0.9	1.3	1.8	2.4	2.7	3	3.6
Avg. Tx Com Time	13.2	18.6	23.6	29.5	35.6	41.5	46	49.8

Table VII
ANACONDA KMEANSLOW TRANSACTIONS' EXECUTION TIMES (MS).

Overall, *Anaconda* improves performance over existing decentralized distributed TM coherence protocols on low contention workloads. Concerning high contention workloads, the centralized protocols perform better than the decentralized ones as they first impose lower transactional overhead and second they minimize the number of aborts by serializing the transactions. Concerning the lock-based distributed Terracotta programs, they perform poorly when long transactions are present. On the contrary, they outperform all protocols when short transactions are present in low contention scenarios.

VII. CONCLUSION

In this paper we have presented and evaluated Anaconda, a JVM clustering solution which integrates a novel decentralized TM coherence protocol. Anaconda is a software TM system designed and built to easily prototype different TM components and coherence protocols on clusters. It is built on top of off-the-shelf JVMs and tracks TM conflicts at object granularity. Finally, Anaconda provides distributed object replication and caching mechanisms.

The objective of this new TM coherence protocol is to minimize network traffic by employing a lazy local and lazy remote conflict detection and lazy object versioning. As a contention manager policy with Anaconda we have selected "older transaction commits first" with a distributed means of generating unique timestamps.

The performance evaluation has compared Anaconda with three TM protocols evaluated in DiSTM. Two of these are centralized, while the other is decentralized. In addition, we have compared Anaconda

against a coarse grain and a medium grain locking implementation of the benchmarks running on Terracotta.

The results indicate that Anaconda's performance depends on the application's contention as well as the transactions' sizes. In LeeTM, Anaconda is the winner outperforming by 40 to 70% the previous TM protocols. Furthermore, it exhibits up to 19x speedup over Terracotta coarse and medium grain locking implementations. On the other hand, in both KMeans and GLifeTM Anaconda is being outperformed by the rest of the protocols. KMeans, due to high contention, favors centralized protocols and therefore the lease-based protocols outperform the decentralized Anaconda and TCC protocols. Furthermore, due to high contention, even the coarse grain locking implementation outperforms the lease-based protocols by removing the transactional overheads. In GLifeTM, although Anaconda scales and improves performance up to 70%, again it is outperformed by the medium and coarse grain locking implementations (executing with Terracotta). The small size of the transactions, despite the low contention, exposes the transactional overheads compared to locks.

ACKNOWLEDGMENT

Dr. Mikel Luján is supported by a Royal Society University Research Fellowship.

REFERENCES

[1] J. R. Larus and R. Rajwar, "Transactional Memory." Morgan and Claypool, 2006.

[2] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain, "Software Transactional Memory for large scale Clusters," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008.

[3] C. Kotselidis, M. Ansari, K. Jarvis, M. Lujan, C. Kirkham, and I. Watson, "DiSTM: A Software Transactional Memory Framework for Clusters," in *ICPP '08: Proceedings of the 37th International Conference on Parallel Processing*, 2008.

[4] J. Bonér and E. Kuleshov, "Clustering the Java virtual machine using aspect-oriented programming," in *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, 2007.

[5] R. Rajwar and J. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *Microarchitecture, IEEE/ACM International Symposium on*. IEEE Computer Society, 2001.

[6] K. Manassiev, M. Mihailescu, and C. Amza, "Exploiting distributed version concurrency in a transactional memory cluster," in *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[7] P. Keleher, A. L. Cox, H. Dwarkadas, and W. Zwaenepoel, "Treadmarks: Distributed shared memory on standard workstations and operating systems," in *Proceedings of the 1994 Winter Usenix Conference*, 1994.

[8] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, "UPC: Distributed shared-memory programming." Wiley-Interscience, 2005.

[9] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance Java dialect," in *Concurrency: Practice and Experience*, 1998.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2005.

[11] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The cascade high productivity language," in *HIPS 2004: 9th International Workshop on High-Level Programming Models and Supportive Environments*, 2004.

[12] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt, *The Fortress language specification version 1.0. Technical report, SUN Microsystems*, 2008.

[13] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: A new paradigm for building scalable distributed systems," in *SOSP '07: Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[14] Y. Aridor, M. Factor, and A. Teperman, "cJVM: A Single System Image of a JVM on a Cluster," in *ICPP '09: Proceedings of the 28th International Conference on Parallel Processing*, 1999.

[15] J. N. Zigman and R. Sankaranarayana, "Designing a distributed jvm on a cluster," in *In Proceedings of the 17th European Simulation Multiconference*, 2003.

[16] T. Kielmann, P. Hatcher, L. Bouge, and H. E. Bal, "Enabling Java for High-Performance Computing: Exploiting Distributed Shared Memory and Remote Method Invocation," in *Communications of the ACM*, 2001.

[17] M. J. M. Ma, C.-L. Wang, and F. C. M. Lau, "Jessica: Java-enabled single-system-image computing architecture," in *Journal of Parallel and Distributed Computing*, 2000.

[18] M. Herlihy, V. Luchangco, and M. Moir, "A flexible framework for implementing software transactional memory," in *OOPSLA '06: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, 2006.

[19] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici, "Programming, deploying, composing, for the grid," in *Grid Computing: Software Environments and Tools*, 2006.

[20] I. Watson, C. Kirkham, and M. Luján, "A study of a transactional parallel routing algorithm," in *PACT '07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, 2007.

[21] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson, "Lee-TM: A non-trivial benchmark for transactional memory," in *ICA3PP '08: Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing*, 2008.

[22] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[23] M. Herlihy, V. Luchangco, M. Moir, and I. William Scherer, "Software transactional memory for dynamic-sized data structures," in *PODC '03: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, 2003.

[24] M. Ansari, K. Jarvis, C. Kotselidis, M. Luján, C. Kirkham, and I. Watson, "Profiling transactional memory applications," in *PDP '09: Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-based Processing*, 2009.

[25] E. Berlekamp, J. H. Conway, and R. K. Guy, "Winning ways for your mathematical plays." New York: Academic Press, 1982.