DiSTM: A Software Transactional Memory Framework for Clusters

Christos Kotselidis, Mohammad Ansari, Kim Jarvis
Mikel Luján, Chris Kirkham, and Ian Watson
School of Computer Science
The University of Manchester, UK
{kotselidis, ansari, jarvis, mikel, chris, watson}@cs.manchester.ac.uk

Abstract

While Transactional Memory (TM) research on shared-memory chip multiprocessors has been flourishing over the last years, limited research has been conducted in the cluster domain. In this paper, we introduce a research platform for exploiting software TM on clusters. The Distributed Software Transactional Memory (DiSTM) system has been designed for easy prototyping of TM coherence protocols and it does not rely on a software or hardware implementation of distributed shared memory. Three TM coherence protocols have been implemented and evaluated with established TM benchmarks. The decentralized Transactional Coherence and Consistency protocol has been compared against two centralized protocols that utilize leases. Results indicate that depending on network congestion and amount of contention different protocols perform better.

1. Introduction

The advent of Chip MultiProcessors (CMPs) provides great incentives for the development of easy-to-use parallel programming models. In this direction, and borrowing from the success in databases, Transactional Memory (TM) [18] is gathering momentum. Until now, synchronization in parallel applications has been achieved by the use of mutually exclusive locks and barriers. The drawbacks of using such mechanisms are the limited scalability (coarse-grain locking), and the cost of program complexity (fine-grain locking). TM promises to address these problems by replacing these synchronization mechanisms with atomic regions executed transactionally. These transactions are speculatively executed in parallel with the read and write operations generating associated read and write sets which are managed by a runtime layer. At some stage of the transaction's lifecycle a validation phase reveals any conflicts (non-empty intersections with the read and write sets) with other concurrently executing transactions. At that stage, a contention management policy is invoked to determine which transactions are aborted or delayed.

The research community has developed numerous TM systems divided into Software (STMs) [19, 14], Hardware (HTMs) [13, 3] and Hybrid Software/Hardware (HyTMs) [9] in order to understand TM behavior. The majority of these TM systems, however, focus on shared-memory parallel architectures leaving unexplored the domain of clusters. The execution behavior of TM on clusters differs significantly from that on shared-memory CMPs due to the expensive communication messages exchanged amongst the nodes.

The Distributed Software Transactional Memory (DiSTM) system is built on top of the state-of-the-art DSTM2 [14] transactional engine and the ProActive framework [5]. In DiSTM, each node can execute multiple transactions according to the number of threads currently running on that node of the cluster. The number of threads is user-defined allowing the installation of the system on single-processor node clusters as well as on multi-processor node clusters. DiSTM also allows the plug-in of different contention managers for local and remote validation.

Our previous work [17] introduced the first version of DiSTM and some initial evaluation results on TM execution on clusters. We enhance the first version of DiSTM by implementing and evaluating three different TM coherence protocols. The decentralized Transactional Coherence and Consistency (TCC) [13] protocol is compared with two centralized TM coherence protocols based on the concept of leases [12]. The first lease protocol allows only one transaction at a time to commit, and make its changes globally visible, therefore serializing the transactions over the network. The second lease protocol allows multiple transactions at a time to acquire the lease and therefore multiple transactions to commit concurrently.

These TM coherence protocols are evaluated on a 40 processor cluster with three benchmarks from the STAMP suite [9], Lee-TM (a complex TM application which implements Lee's routing algorithm for circuit boards)

[20, 4], and GLife-TM (a transactional version of Conway's Game of Life [6]).

The remainder of the paper is organized as follows: Section 2 introduces background and related work, and Section 3 describes the core components of DiSTM. Section 4 describes the implemented TM coherence protocols. Section 5 describes the platform as well as the benchmarks used in the evaluation. Section 6 presents the evaluation results, while Section 7 summarizes the paper.

2. Related Work

TM research has focused on shared-memory CMPs. The only related published work that tackles distributed TM execution are Distributed MultiVersioning (DMV) [19] and Cluster-STM [7]. In DMV, transactional execution is achieved by modifying the underlying software Distributed Shared Memory system (DSM) [16]. The role of the DSM is to provide a shared memory view among the nodes of the cluster. Cluster-STM is a prototype implementation in which one thread is being executed on each processor. Our framework, the Distributed Software Transactional Memory (DiSTM) system, differs significantly from DMV as it does not rely on a DSM mechanism to achieve memory coherence. Furthermore, the problems of local and remote conflicts are tackled in DiSTM as each node of the cluster is a CMP allowing the execution of multiple concurrent threads. DiSTM also employs transactional semantics at object granularity instead of page granularity as DMV or word granularity as Cluster-STM while being more modular and flexible for experimentation purposes.

Besides the research conducted on the area of TM, new parallel programming languages are being developed to enable efficient parallel programming on clusters. The so-called *Partitioned Global Address Space (PGAS)* languages such as UPC [11], Titanium [21] and the emerging ones X10 [10] and Fortress [2] allow parallel programming while providing a global address space. Some of the *PGAS* languages include the *atomic* construct without currently containing any underlying distributed TM system. The transactional implementation of these constructs is not yet defined as it is still a subject of research.

Transactions have already started being investigated on distributed environments with Sinfonia [1]. Sinfonia utilize "minitransactions" which replace message-passing protocols on applications such as cluster file servers. In the domain of Java Virtual Machines (JVMs), distributed solutions for enterprise applications already exist (e.g. Terracota [8]), but still rely on locks for synchronization among threads. DiSTM can assist in this domain by providing a research platform for investigating transactional execution on distributed JVMs.

The work presented in this paper enables experimentation and investigation of the transactional semantics of these structures.

3. Distributed Software Transactional Memory

DiSTM is written entirely in Java and it builds on two core components: an underlying transactional execution engine and a remote communication system. As a transactional engine we have extended DSTM2 [14]. The remote communication is based on the ProActive framework [5] which is a high level API for Java RMI. The following subsections describe how these components are integrated.

3.1. DSTM2 Transactional Engine

DSTM2 [14] is a Java STM which supports transactional execution for dynamically-sized data structures on shared-memory architectures. All transactions are executed speculatively. When a transaction attempts to modify an object, instead of directly modifying 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 (write-after-write or read-after-write) are detected and resolved. Upon conflict detection, a contention manager is consulted 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 cloned objects). DSTM2 employs an obstruction-free synchronization policy [15] which guarantees forward progress as any halted threads do not prevent active threads from making progress. However, the obstruction-free synchronization policy does not prevent active threads from causing livelocks.

We selected DSTM2 because it offers a wide array of contention managers. Furthermore, it allows the user to plug-in custom contention managers making it an ideal platform for experimentation purposes. Further information can be found in [14].

3.1.1 Extensions to DSTM2

Distributed functionality was added 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 Compare-AndSwaping (CASing) their status flags from ACTIVE to COMMIT. To that scheme an extra step has been added. The role of this extra step is to guarantee transactional coherence on the cluster. The actions taking place at this step depend on which TM coherence protocol is chosen. For example, if the TCC protocol is used, then at this step the transaction's read/write sets are broadcast to the nodes of the cluster in order to be validated against the concurrently running remote transactions. On the contrary, if a lease scheme is employed, the transaction attempts to acquire a lease before it commits.

No matter which protocol is used, after finishing this step, the transaction is aware of whether it is safe for it to commit or not, based on the returned value.

Each node of the cluster maintains a cached version of the transactional dataset. Transactions running on each node read/write from/to the cached dataset. DiSTM is responsible for keeping the various cached datasets consistent. For brevity, details of how objects are identified amongst the nodes are omitted, but can be found in [17].

3.2. Remote Communication

The remote communication layer is written in Java and relies on the ProActive framework [5]. The key concept of the ProActive framework is *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. Depending on the TM coherence protocol, various active objects are created on the master and the worker nodes. However, the number of active objects that constitute the skeleton of DiSTM are constant. Upon bootstrap, a JVM is created on every node, including the master node. DiSTM begins execution on the main thread (instance of the `MAIN` active object) on the master node by creating the necessary structures on the remaining nodes. The JVM on each worker node has an instance of the `DiSTMClient` class (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 remaining active objects of DiSTM differ for each protocol, and are presented along with their implementations (next section).

4. Distributed TM Coherence Protocols

This section describes the distributed TM coherence protocols implemented in DiSTM. The first subsection discusses an equivalent to the decentralized TCC protocol [13], while the second subsection describes two novel centralized protocols that utilize leases [12].

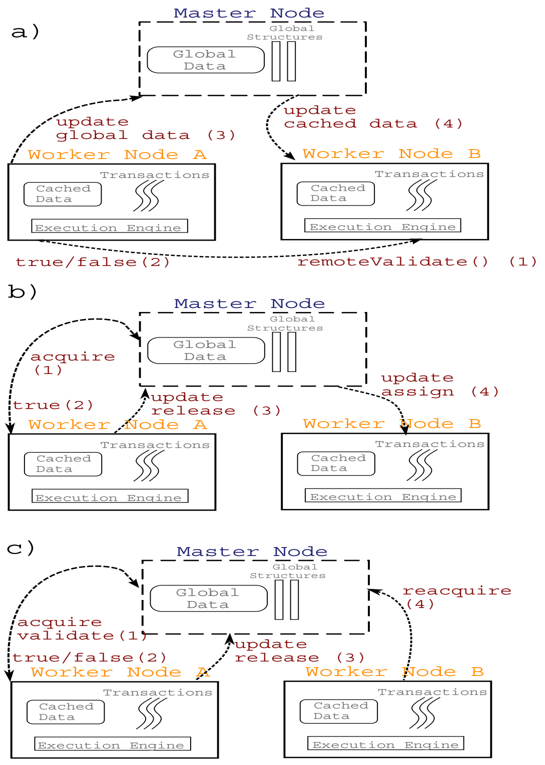

Figure 1. Distributed TM coherence protocols: a) TCC, b) Serialization lease, c) Multiple leases.

4.1 TCC

TCC performs lazy validation of transactions that attempt to commit. Each transaction that wishes to commit, 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. TCC, initially designed for cache-coherent architectures, has been adopted as a decentralized TM coherence protocol for DiSTM. In order to maintain an ordering of the transactions on the cluster, a "ticketing" mechanism has been employed. Each transaction before broadcasting its read/write sets, in order to be validated against transactions running on remote nodes, acquires a "ticket" (global serialization number) from the master node. The role of the "ticket" is to assist the contention manager upon a conflict detection between transactions running on different nodes. The contention management policy adopted is the oldest-commit-first policy (oldest in terms of remote validation time – which transaction attempts to remotely validate its read/write sets first). Upon remote validation, see Fig. 1a) Step 1, a transaction's read and write sets (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 `remoteValidate()` method returns true, so the caller can commit safely.

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 local transaction has acquired the ticket after the transaction it is validated against (incoming transaction). The local transaction is considered to be "younger" and therefore has to be aborted. Instead of aborting the local transaction immediately, its id is stored in a temporary queue. Each transaction with a greater "ticket" (than the incoming transaction) will be stored in the queue. The decision of whether to abort the transactions in the queue takes place when the incoming transaction has been validated against all local transactions. When the validation of the incoming transaction has finished and there has been no conflict with a local transaction with a smaller "ticket" number (older) the transactions stored in the queue are aborted. This is due to serial validation. 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 may be a 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 unnecessarily aborted the first one.
   If the incoming transaction conflicts only with "younger" locally executed transactions, the `remoteValidate()` method returns true so the caller can proceed in committing 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. the local transaction is older and should abort the incoming younger transaction. Consequently, the `remoteValidate()` function returns false and the queue holding the conflicting younger local transactions (if there are any) is released.

The commit phase follows the validation phase. A transaction that has passed the validation phase successfully must make its changes visible both locally (at the node it is running) and globally (to the rest of the nodes of the cluster), Fig. 1a) Step 3. TM coherence is assured by a master-centric, eager approach. After a transaction has made its changes visible locally, it updates the global dataset kept at the master node. In turn, the master node eagerly updates all the cached datasets on the rest of the nodes of the cluster Fig. 1a) Step 4. Upon updating the cached datasets, a validation phase occurs which invalidates the transactions that have read "dirty" values of the cached dataset. Any transactions discovered during the invalidation phase, are re-executed after the node gets a consistent view of the data.

4.2 Lease-based TM Coherence Protocols

When utilizing leases on the network, two additional active objects are created on the master node. The active objects are assigned the roles of acquiring and releasing the leases. Depending on which of the following schemes is utilized, the lease handlers behave differently.

4.2.1 Serialization Lease

The role of the lease is to serialize the transactions' commits over the network and therefore to avoid the expensive broadcasting of transactions' read/write sets for validation purposes. Each transaction that passes the local validation phase, attempts to acquire the lease from the master node Fig. 1b) Step 1. If no other transaction has acquired the lease, the incoming transaction acquires the lease. Any other transaction that tries to acquire the lease, after the transaction from worker node A has acquired it, will block and wait its turn to commit after adding itself to a queue kept at the master node. The role of the queue is to maintain the order of the transactions waiting to acquire the lease. When the transaction (lease owner) of worker node A commits, it updates the global data at the master node and then releases the lease Fig. 1b) Step 2. Before the lease is released, the cached datasets of the worker nodes are updated with the new values of the committed transaction. A validation phase while fetching the new data aborts any conflicting local transactions. After the lease is released, the master node retrieves the next transaction from the queue and attempts to assign the lease to it Fig. 1b) Step 4. If the transaction has not been aborted (in the process of updating its cached dataset), it acquires the lease and proceeds in committing. On the contrary, if the transaction has been aborted, the master node attempts to assign the lease to the next transaction retrieved from the queue.

The advantage of the serialization lease is the minimization of the broadcasting messages exchanged in TCC. The disadvantages are that transactions block waiting to be assigned the lease, that there may be attempts to assign the lease to aborted transactions and the bottlenecks created upon acquiring and releasing the lease at the master node.

4.2.2 Multiple Leases

In this scheme, multiple leases are assigned for transactions that attempt to commit. After a transaction passes the local validation phase, it attempts to acquire a lease from the master node, Fig. 1c) Step 1. Unlike the previous lease scheme where only one lease was assigned at a time, in this scheme multiple leases are available for transactions. When a transaction attempts to acquire a lease, a validation phase is performed at the master node. Each transaction that attempts to acquire a lease is validated against each transaction (stored in the pool) that currently owns a lease. If there is no conflict, the transaction acquires a lease and proceeds in committing after adding itself to the pool of transactions that hold a lease. If a conflict is discovered, the transaction aborts and restarts. Upon successful commit, Fig. 1c) Step 3, the transaction updates the global data at the master node and in turn the master node updates the cached data at the worker nodes. The transactions that unsuccessfully attempted to acquire a lease (and have been aborted) will be re-executed and consequently try to acquire a lease during their commit stage. There is no limit in the number of leases that can be assigned on the cluster. As long as no conflicts are detected at the master node leases can be assigned. However, controlling the number of leases could be a way to control the network traffic on the cluster. The advantages of this scheme is that multiple transactions can commit concurrently. The disadvantages are the fact that an extra validation step has been added to the master node as well as the bottlenecks created upon acquiring and releasing the leases.

5. Experimental Platform

5.1. Hardware

The hardware platform used in our experiments is a cluster with 40 cores residing on five nodes: the master node and four others while the network interconnection is a Gigabit ethernet. The master node has 2 dual-core AMD Opteron CPUs at 2.4GHz with 8GB of RAM. All the remaining worker nodes are 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 Java 6 build 1.6.0-b105 with maximum heap size set to 8GB.

5.2. Benchmarks

Complex benchmarks for TM systems have only recently emerged in the literature. In order to evaluate our system, the benchmarks of the STAMP benchmark suite ver 0.9.5 [9] as well as Lee's transactional routing algorithm [20, 4] have been ported to our system while GLife-TM has been implemented from scratch. In total five benchmarks have been used to evaluate our system: Lee-TM, GLife-TM, KMeans, Vacation and Genome.

Lee-TM implements the classic 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 configuration of 1506 routes is used in the evaluation. Parameters used: input_file:mainboard.
KMeans 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. Parameters used: clusters:40, threshold:0.05, input_file:random10000_12.
Vacation is a simulator of an enterprise server. It is similar to the SpecJBB benchmark. Several threads acting as clients try to book, view, and edit their records while performing actions, such as renting cars, booking flights or hotel rooms. Parameters used: relations:40, percent_of_relations_queries:80, queries_per_transaction:1, number_of_transactions:2430.
Genome performs gene sequencing from randomly generated segments. Conflicts occur when different transactions try to use the same segment during the segment matching phase. Parameters used: gene_length:1834, segment_length:24, num_segments:19430.
GLife-TM 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:600, rows:600, generations:10.

Concerning the transactional engine, we used the Priority contention manager (older always commit) and the obstruction freedom synchronization policy (for local transaction execution).

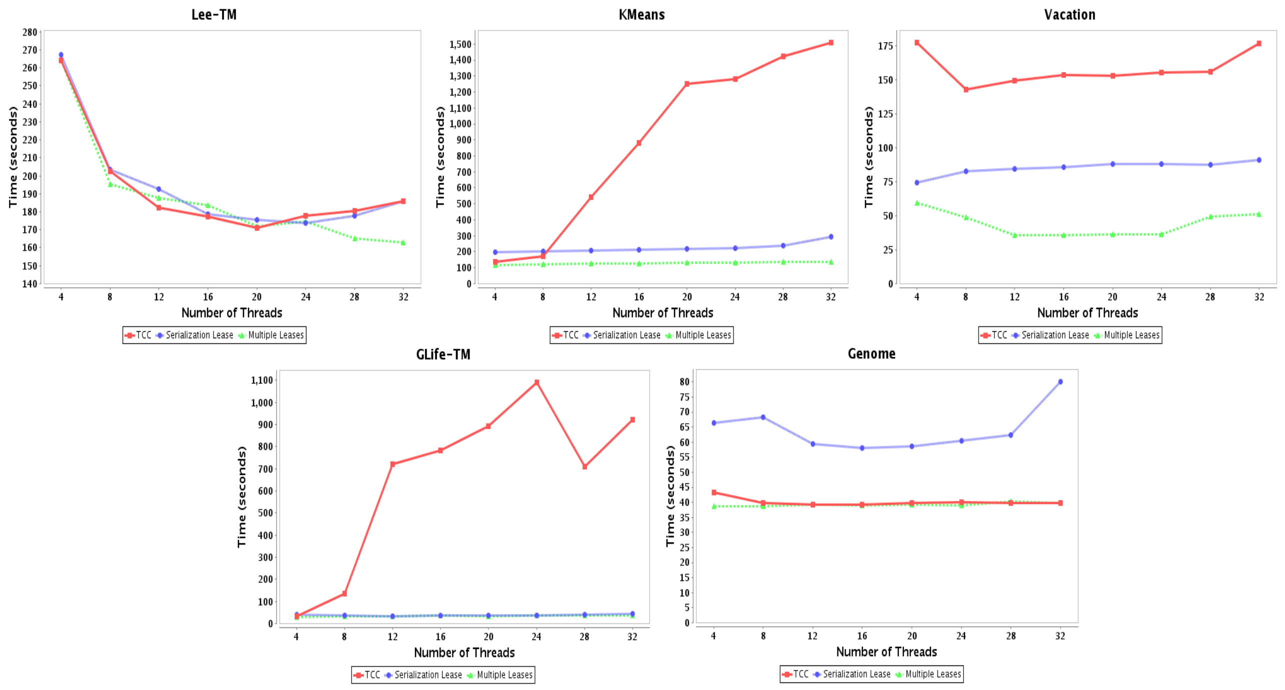

Figure 2. Benchmarks' execution times comparing the three TM coherence protocols.

6. Evaluation

This section contains the results of evaluating the TM coherence protocols with the described benchmarks. The results shown are the averages of ten iterations. Figure 2 illustrates the execution time over an increasing number of threads with respect to executing with one thread per node. For example, the four-threads point in the x-axis shows the execution time using one thread per cluster node. Similarly, the 16-threads points presents the execution time using 4 threads per cluster node.

Concerning Lee-TM, all three protocols exhibit similar behavior as more threads are added. For TCC, there is a peak at 20 threads (maximum speed-up 35%) beyond which, performance degrades. This is due to the fact that the performance benefit of executing with more threads is outweighed by the cost of validating remotely more transactions on the cluster. When the serialization lease is utilized, all commits are serialized on the network. While at the beginning a slight speed-up is observed, after a certain point (24 threads), performance starts degrading. This is due to more transactions being invalidated when the worker nodes update their cached datasets. With multiple leases a constant speed-up is observed (maximum 38% at 32 threads). The performance gained by committing more transactions in parallel seems to overcome the cost of the additional validation step added at the master node.

Concerning KMeans, we can observe that no speed-up is gained in any configuration. The cause of the deterioration in the performance of KMeans is the high abort rate among transactions resulting in limited parallelism (objects are attempted to be clustered in 40 partition resulting from 45% (4 threads) to 5500% (32 threads) abort rate). Compared with Lee-TM, transactions in KMeans are 480 times smaller in execution time and therefore spend the majority of their time trying either validating themselves against remote transactions (TCC) or acquiring leases (lease schemes). The time spent at these stages are orders of magnitude larger than pure execution time resulting in local transactions being invalidated at a high rate. When the TCC protocol is employed, we can see that the network traffic has a severe impact on performance. On the other hand, if the serialization lease is utilized, all transactions are serialized on the network and each one waits its turn to commit. This results in an approximately steady performance no matter the number of threads used. A slight deterioration is observed when we add threads because more transactions are being invalidated when the cached data are updated. This causes the bottleneck (while trying to acquire a lease) on the master node to become greater and have an impact on performance. When multiple leases are used, the validation phase at the master node invalidates transactions at a high rate allowing only a small number of transactions to commit in parallel. Hence, the execution time is identical in all configurations.

When leases are utilized in Vacation, great performance improvement (compared to TCC) is observed. Vacation exhibits, a stable transaction abort rate (below 30% at worst case) and thus the remote validation requests in TCC outweighed the performance benefit of parallel execution. On the contrary, when the serialization lease is used performance remains stable, with a slight increase in execution time. Multiple leases outperform the serialization lease and show an execution time pattern similar to TCC (but with approximately two times lower execution time).

GLife-TM exhibits fluctuating execution time when TCC is used. This is a benchmark specific issue concerning scheduling. In GLife-TM each thread is assigned the calculation of the next generation of a partition of the cell grid. Conflicts occur when a thread reads cell values from a partition which another thread has modified (read/write conflict). Therefore, the transactions abort rate is highly connected with the location of the conflicting partitions. The deterioration in performance is accompanied by high abort rate (from 2380% at 4 threads to 28825% at 24 threads) because neighboring partitions have been scheduled on the same node. On the contrary, when leases are used the execution time is significantly lower than TCC because the high invalidation rate is avoided by the transactions' serialization.

Genome is the only benchmark where TCC outperforms in all cases the serialization lease and also has identical execution time with multiple leases. This is due to the fact that Genome has a very low transaction abort rate (less than 1%), and therefore transactions can commit in parallel at a high rate.

The two main conclusions that can be drawn are the limited scalability for the given benchmarks and the fact that TCC is usually outperformed by the lease-based protocols. The only exceptions are Lee-TM and Genome where TCC performs better than or similarly to both lease schemes. Furthermore Lee-TM shows the most significant performance improvement by 35% (TCC) and 38% (Multiple Leases). To gain a better understanding we split down the execution time of transactions in Fig. 3. The time spent at each step of a transaction's lifetime has been measured, using the Java timing utilities, and averaged in order to separate the amount of execution time spent for local actions (local validation and commit) from that of remote operations (remote validation/lock acquisition or release and commit). Finally, the section of the bars labeled as "computation" represents the execution time of a transaction spent in real computation. As depicted in Figure 3, Lee-TM is the only computationally intensive benchmark spending from 62% to 74% of its time performing computations. Thus, the transactions are spread in the time spectrum and the remote validation does not have a great impact on performance. By contrast, all other benchmarks perform light computations resulting in spending the majority of their time in remote

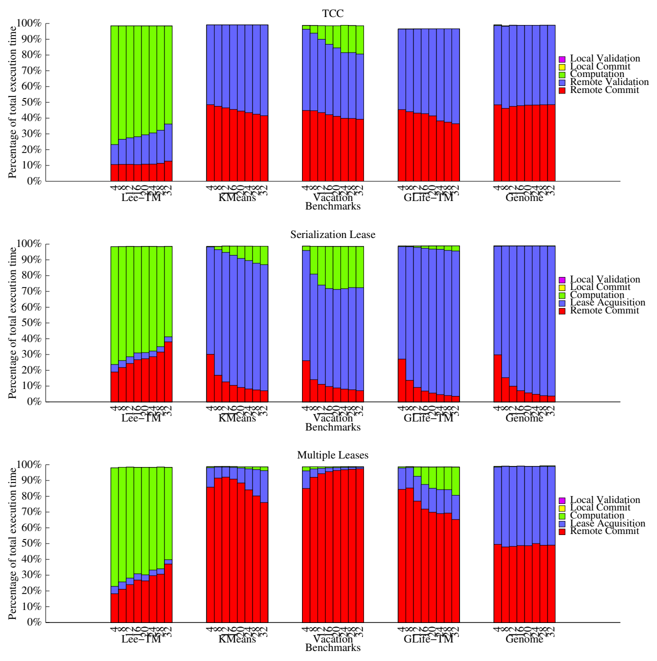

Figure 3. Breakdown by transaction stage of the execution times.

requests. Comparing the serialization lease graph with the multiple lease graph we can observe that the remote commit and lock acquisition percentages are almost switched for the non-computationally intensive benchmarks. This is due to the bottleneck created upon commit at the multiple lease scheme. This bottleneck does not exist in the serialization lease scheme as only one transaction at a time can commit. However, in the serialization scheme, transactions spend the majority of their time trying to acquire a lease.

To conclude, the results indicate that the selection of a TM coherence protocol is determined by two factors. The first factor is the transaction abort rate (in Genome, TCC outperforms the serialization lease) while the second factor is how computationally intensive the application, with respect to the remote operations of the TM coherence protocols, is (in Lee-TM TCC sometimes outperforms or performs similarly to the lease schemes).

7. Summary

TM research for shared-memory CMPs has flourished in recent years. Clusters, being a core part of high performance computing, remain open for evaluation with TM. To this end, the flexible research platform DiSTM has been designed and implemented.

This paper has evaluated three distributed TM coherence protocols on DiSTM with complex TM benchmarks existing in the literature. The results underline the important role remote communication plays in transactions' validation and commit phases. Furthermore, depending on the contention of the application, different TM coherence protocols perform better. When contention is high, and transactions are serialized on the network, it is more effective to use a centralized protocol, such as a lease scheme, in order to avoid network congestion. On the other hand, when transactions spend most of their time on computations and little in communication, a decentralized memory coherence protocol seems to perform better as we avoid the bottlenecks caused by lease acquisition and release.

References

[1] 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.

[2] 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.

[3] 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*, 2005.

[4] 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.

[5] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. Programming, deploying, composing, for the grid. Springer-Verlag, 2006.

[6] E. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways for your mathematical plays*. Academic Press, New York, 1982.

[7] 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.

[8] 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.

[9] 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 *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[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] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2005.

[12] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP '89: Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 1989.

[13] 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. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[14] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006.

[15] 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 22nd Annual Symposium on Principles of Distributed Computing*, 2003.

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

[17] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Investigating software transactional memory on clusters. In *Proceedings of the 10th International Workshop on Java and Components for Parallelism, Distribution and Concurrency*, 2008.

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

[19] 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.

[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 Architecture and Compilation Techniques*, 2007.

[21] 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. *Concurrency: Practice and Experience*, 1998.