Concurrent Programming Using Transactional Memory With Parallel Nesting

David Granchinho
Under supervision of João Pedro Barreto and Paulo Ferreira
Dep. Engenharia Informática, IST, Lisbon, Portugal

October 14, 2011

Abstract

Transactional Memory (TM) is a promising technique that simplifies parallel programming through a simple abstraction: the memory transaction. Yet, TM implementations available today don't fully explore the potential parallelism available: they either disallow nested transactions to run in parallel or they introduce overheads that make them not scalable to deeper nesting depths. For this reason there are no applications that expose parallel nesting which makes the scientific community doubt about the relevance of parallel nesting.

This work helps to shed a new light on parallel nesting through the means of simulation of executions of a modified application that exposes parallel nesting. For this work, a simulator of a TM system featuring an efficient support to parallel nesting was built. The Vacation application of the STAMP benchmark suite was modified in order to expose parallel nesting.

Our main results show that parallel nesting can improve performance even in high contention scenarios, and show that the modified versions of known contention mangers such as Karma and Eruption achieve better performances by taking into account the relations between nested transactions.

Keywords: Transactional Memory, Parallel Nesting, Simulation, Contention Managers

1 Introduction

Transactional Memory (TM) is seen today as a powerful paradigm for developing concurrent applications [1–3] and particularly Software Transactional Memory systems (STM) have enjoyed an increase in popularity. This paradigm makes programming concurrent applications easier by relieving the programmer from the responsibility of dealing with concurrency control. As such, the programmer simply has to identify sequences of instructions as transactions that access and modify shared objects in an atomic and isolated way. By allowing this, TM reduces the number of bugs and, consequently, reduces the software development time.

In order to fully explore the potential of this paradigm and take advantage of the multiprocessing capabilities of today's processors, it is relevant to support the parallel execution of nested transactions. Yet, this feature is still largely unexplored mostly due to the complexity inherent to efficiently support parallel nesting [4] and also because of the lack of applications exposing parallel nesting. Because of this, there's still no consensus among the scientific community about the importance of parallel nesting and about the research directions on the development of TM implementations that efficiently support parallel nesting [4–6].

In order to give further insight into the problem of efficiently supporting parallel nesting, this work studies the performance of nested parallelism in scenarios with different levels of contention, studies different conflict detection and management techniques, identifies its limitations, propose alternatives and evaluates the different techniques through simulated executions of an application exposing parallel nesting. An adapted version of the Vacation application of the STAMP benchmark suite [7] is used. This application is adapted so as to expose parallel nesting.

Because this work focuses on the parallel execution of subtransactions, and in order to help to fill the gap on the development of distributed and replicated STMs with the objective of enhancing both

dependability and performance [8], and also in order to gain further insight on parallel nesting, this work also proposes and evaluates a new protocol of distributed parallel nesting, whose objective is to increase the performance of a distributed STM by allowing nested transactions to run in parallel in different nodes of the network.

In summary, this work aims to answer the following three questions: *Q1*: Can parallel nesting improve application performance? *Q2*: Are traditional contention managers [9] adequate to parallel nesting? *Q3*: Does parallel nesting make sense in a distributed environment?

The remaining of this paper is organized as follows. Section 2 presents and discusses related work. Section 3 describes the simulator built for this work, the different contention management techniques that were tested and also the proposed protocol for distributed parallel nesting. Section 4 presents the results of the experimental study. Finally, Section 5 concludes this paper.

2 Related Work

Recently, several TM systems that support parallel nesting have already been proposed, namely NesTM [6], NePalTM [10], and PNSTM [4]. NePalTM was the first TM system to support some type of parallel nesting. It does so by allowing parallel threads to be forked inside a transaction. It allows those threads to run in parallel as long as they don't create any subtransactions. If any of the subthreads create subtransactions, NePalTM enforces the sibling transactions to run in mutual exclusion. For this reason, NePalTM does not support fully-parallel nesting.

Unlike NePalTM, NesTM theorically supports parallel nesting with unbounded nesting depths. However, its transaction handling overheads (beginning, committing and detecting conflicts) grow linearly with nesting depth. Also, this system features late conflict detection between transactions which forces it to do work that grows proportionally with the nesting depth of the committing transaction [4, 5]. For these reasons, NesTM is an adequate solution for low nesting depths only.

Unlike both NePalTM and NesTM, PNSTM truly supports parallel nesting with an unbounded nesting depth, featuring low overheads that are independent of nesting depth. However, this system has the severe limitation of treating every transaction as a write-only transaction. As such, this system doesn't allow multiple concurrent transactions to read the same object.

The area of distributed STMs is still a largely unexplored one. Distributed STMs are designed to give the semantics of the TM paradigm in a distributed context so as to allow replication and synchronization among the nodes of the system. To do so, distributed STMs require message exchanging between the nodes whenever a transaction tries to commit, which imposes great overheads. As such, most of the work done in this area is about studying new ways to reduce the impact of the message exchanging [8]. D^2STM [8] is one such system. This system uses an instance of the JVSTM [11] on each node, and uses an *Atomic Broadcast* service [12] to exchange messages between nodes. This service is reliable and guarantees *Total Order* between the exchanged messages. In order to maintain coherence and synchronization between nodes, this system uses a *Bloom Filter*-based certification protocol (BFC) (built on top of the underlying Atomic Broadcast service) which takes advantage of the space-efficient Bloom Filter-based encoding to reduce the size of the messages and thus reducing the synchronization overhead. Because this system is built on top of the JVSTM, it doesn't feature any kind of nesting of transactions.

In order to test different TM implementations, several TM benchmarks have already been proposed. These benchmarks range from trivial applications such as *red-black trees* and *skiplists* [2, 3] that feature simple data structures and short transactions with few accesses, to more complex real-world applications that generate longer and more complex transactions such as *Atomic Quake* [13] and *FénixEDU* [14] and to comprehensive benchmark suites such as STAMP [7], Lee-TM [15] and STMBench7 [16]. Atomic Quake is an adapted transactional version of a multiplayer game server. This application generates both long and short transactions with long and short *read sets* and *write sets*, with nested transactions that reach up to 9 levels of depth. Even though this application features nesting and has enough complexity that its probably possible to expose parallel nesting, its complexity and size make it undesirable for simulated runs. FénixEDU is a web application used by Instituto Superior Técnico where it provides critical management services to both students and teachers. This application relies on a distributed STM for transactionally manipulating the in-memory state of its application server. STAMP is a benchmark suite specifically designed for evaluating TM systems. This benchmark is composed by 8 distinct applications and different workloads. The applications are representative of different scientific areas such as engineering, graphical computation and learning.

Even though this benchmark covers a wide area of applications, none of them exposes nesting as this benchmark was initially created to evaluate TM systems that at the time didn't support nesting. However, due to its wide variety of workloads and transaction complexity, this benchmark has enough complexity to explore both sequential nesting and parallel nesting. Also, this benchmark features configurations appropriate to simulations [7]. Lee-TM features long and realistic workloads and is based on the algorithm of *circuit routing* of Lee [15]. This benchmark doesn't expose nesting and is less versatile than other benchmarks. It's also important to notice that one of the applications composing the STAMP benchmark suite (*labyrinth*) uses the same algorithm as Lee-TM and thus produces an identical workload. Finally, STMBench7 features complex and realistic workloads and a wide variety of operations: from small operations with only a few reads, to longer and more complex operations that write several objects. The workloads vary from read-only workloads to write-dominated workloads. Just like the other benchmarks, STMBench7 doesn't expose nesting of transactions, but in [10], a modified version of this benchmark is used to evaluate parallel nesting on NePalTM.

3 Simulator

For the purpose of this work, we built a simulator capable of simulating program executions featuring parallel nesting. The simulator is composed of two main components. The first is a profiling component that produces information about an execution of a given client application. The profiling component was built on top of one of the tools of the Valgrind framework [17], a framework for Dynamic Binary Instrumentation. This component outputs information about each executed block to an XML file. For each block, this component outputs the number of instructions (which is used as the execution cost of this block), read and write accesses. Because of the way this information will be processed later, this component is only used for sequential client programs (i.e. programs that don't explicitly create threads). In order to later simulate thread creation and transaction creation, this component detects the usage of special keywords on the client code such as `LIBDJG_THREADFORK()` and void `LIBDJG_ENDTHREAD()` for delimiting some thread's code. Similar keywords are used for delimiting transactions, parallel transactions and work done on distinct nodes (for simulations of distributed environments). The usage of these keywords is also outputted to the XML file so that threads and transactions can be simulated on the simulation component.

The execution information in the XML file is used by the second component which is the simulation component. The simulation component (implemented in Java) uses the information from the profiling component and other input parameters in order to simulate executions of the client application as if it were using an underlying TM system characterized by the given input parameters. The input parameters specify: the execution type (local or distributed); the conflict detector (optimistic or pessimistic); the conflict manager (section 3.1); the number of processor cores; the network latency (for distributed STM simulation); and the number of nodes of the network (for distributed STM simulation). The simulation algorithm is divided in two phases: the construction phase and the execution phase. The construction phase parses the input XML file and creates an execution tree which represents the threads created during the execution and the relationship between them (e.g. which thread created the other thread), and also a transaction execution tree that represents relationships between transactions. During this phase, the execution blocks are associated to their respective thread and transaction (or to non-transactional code). The execution phase uses the information from the construction phase and simulates the execution by adding each block one at a time to the processor's core executing the corresponding thread. By adding a block to a core, the core's clock is incremented by the value of the block's cost. Every time a new thread is created, it is stolen by one of the cores and starts the execution. During the simulation, if a conflict is detected between two transactions, it is resolved by the specified conflict manager. In the end of the simulation, the simulator outputs the execution time and the number of aborted transactions. The execution time is calculated from the value of the clock of the core that finished last.

3.1 Contention Managers

To date, several contention managers have already been defined [9, 18, 19]. The Aggressive contention manager always aborts the conflicting transaction (victim transaction) in case of a conflict. The Polite contention manager exponentially backs off for a fixed number of attempts, eventually aborting the conflicting transaction. The Randomized contention manager either aborts the victim

transaction with some probability P or waits for it with 1 - P probability. The Greedy contention manager associates a timestamp to a transaction when it starts. This contention manager establishes priorities by giving a higher priority to an older transaction. In case of conflict, if the victim transaction is waiting for another transaction it is aborted regardless of it's priority. Otherwise, the attacking transaction always waits for a conflicting transaction with higher priority and always aborts a conflicting transaction with lower priority. The Karma contention manager increases the priority of a transaction whenever the transaction successfully another block. When two transactions are in conflict, the attacking transaction makes a number of attempts equal to the difference among priorities of both transactions with a constant backoff time between attempts. The Eruption contention manager is similar to Karma, except that when a conflict occurs, if the victim transaction has a higher priority than the attacking one, the priority of the attacking transaction is added to the priority of the victim transaction, and then sends the attacking transaction to sleep. This work also features the Default contention manager which is similar to the greedy contention manager, except it uses an execution cost instead of a timestamp, and it always aborts either the victim transaction or the attacking transaction. In other words, it never sends any transaction to sleep.

This work proposes modifications on both Karma and Eruption contention managers in order to make them aware of the ancestor-descendent relationship between transactions and sub-transactions. Thus we propose the following modifications to both contention managers: when a sub-transaction is created, it gets it's parent priority value. When a sub-transaction commits, it also adds the increment on it's priority level to it's parent transaction.

The Aggressive and Default contention managers are tested using both optimistic and pessimistic conflict detectors. The other contention managers are only tested using the pessimistic conflict detector. The optimistic conflict detector only does conflict detection at commit time, while the pessimistic conflict detector does conflict detection every time a new shared object is accessed.

3.2 Distributed Parallel Nesting Protocol

This work proposes a protocol for distributed parallel nesting that allows the parallel execution of nested sub-transactions from the same family on distinct nodes of a distributed and replicated STM.

In a regular distributed STM such as D^{2}STM [8], message exchanging between the nodes is required whenever a transaction tries to commit, in order to guarantee coherence and synchronization among the nodes of the system. Using it's protocol, if this STM supported nesting of transactions, it would only support local nesting: A sub-transaction could only execute on the same node as it's parent transaction (local-only nesting). In order to enable distributed parallel nesting, the proposed protocol requires additional message exchanging steps: First, a message is sent from the original node to the host node containing the necessary information to execute the sub-transaction. The sub-transaction is then executed on the host node and goes through conflict detection on that node. After successfully finishing, a message is sent from the host node to the original node containing the sub-transaction's read and write sets. Back on the original node, the sub-transaction's read and write sets are added to its parent transaction. If there are conflicts between the sub-transaction and any other local transactions, the victim transactions are aborted (i.e. the returning sub-transaction has always maximum priority). In order to identify sub-transactions that can be run in parallel and distributed, the keywords `DPARALLELSTART` and `PARALLELEND` must be used on the client code to identify it.

4 Evaluation

In this work we modify the Vacation application of the STAMP benchmark suite [7] so as to expose parallel nesting. This application implements a travel reservation system powered by a non-distributed database. The workload consists of several client threads interacting with the database via the system's transaction manager. There are four different possible tasks: *Make Reservation* - the client checks the price of n items, and reserves a few of them; *Delete Customer* - The total cost of a customer's reservations is computed and then the customer is removed from the system; *Add to Item Tables* - Add n new items for reservation; *Remove from Item Tables* - Remove n new items for reservation. The distribution of the tasks and the percentage of the total number of relations queried can be adjusted in order to change the workload and the contention level. For the purpose of evaluating TM systems, each task is encapsulated by a transaction [7]. In order to expose parallel nesting, we subdivided each transaction into sub-transactions as follows: *Make Reservation* - each price

query is a distinct sub-transaction. They can be run in parallel; *Delete Customer* - The total cost of a customer's reservations is computed in a sub-transaction and then the customer is removed from the system in another sub-transaction; *Add/Remove to/from Item Tables* - Each addition/removal is a distinct sub-transaction, and they can all run in parallel.

In order to obtain answers to the questions *Q1* and *Q2*, we simulated centralized executions with a processor with 4 cores. In the experimental tests we varied the number of client threads from 1 to 4 and varied the contention levels. We compared executions featuring parallel nesting with executions featuring only sequential nesting and flat nesting. Also, we compared every contention manager described in section 3.1.

In order to obtain answers to *Q3*, we compared the performance of the distributed parallel nesting protocol (only 2% of the total number of transactions are distributed sub-transactions) with the performance of local-only executions: local flat nesting, local sequential nesting and local parallel nesting. To do this, we simulated a distributed execution where the client threads were spread through multiple working nodes, each with 4 cores, and the network latency is 0.008s from [20]. The number of client threads was varied from 1 to 4 in each node and the number of working nodes went from 2 to 5. In the simulations, there is always one non-working node that initially has no work. This node will only be used when distributed parallel nesting is allowed.

4.1 Results

4.1.1 *Q1*: Performance of Parallel Nesting

Figure 1 shows the speedup curves (normalized to a sequential execution) for some of the contention manageres in a scenario of high contention, and also their espective number of aborts. The results show that, overall, parallel nesting outperforms sequential (and flat) nesting for a low number (1 or 2) client threads. However for 3 or 4 threads, the sequential nesting achieves better performance levels. With 3 or 4 client threads taking full (or almost full) advantage of the 4 cores, there are no gains in parallelism when parallel nesting is supported. By trying to run more transactions at the same time, parallel nesting generates higher levels of contention, and thus higher numbers of aborts which translates into lower performance when it's not possible for the processor to execute more work in parallel.

4.1.2 *Q2*: Contention Management with Parallel Nesting

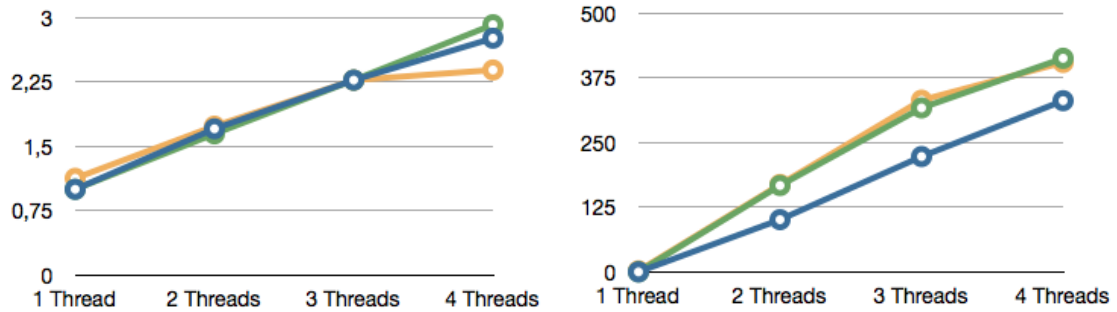
Figure 2 shows the speedup curves for all the contention managers when supporting parallel nesting, for two different contention levels. In lower contention contention scenario in figure 2(a), the contention managers achieve closer performance results than in the higher contention scenario represented in figure 2(b). Overall, Karma and Eruption contention managers achieve the best performances. The results also show that for higher contention scenarios, the modified versions of Karma and Eruption consistently achieve the best performances.

4.1.3 *Q3*: Distributed Parallel Nesting

Figure 3(a) shows the speedup curve of the distributed nested parallelism execution normalized to the local-only flat nested execution. The results show that the local-only executions are similar to each other and outperform the distributed nested parallelism execution. Figure 3(b) shows that distributed parallel nesting generates a number of aborts that is similar to the local-only executions. As such, the underperformance of this protocol is due to the overheads of the extra message exchanging steps outweigh the gain of parallel processing.

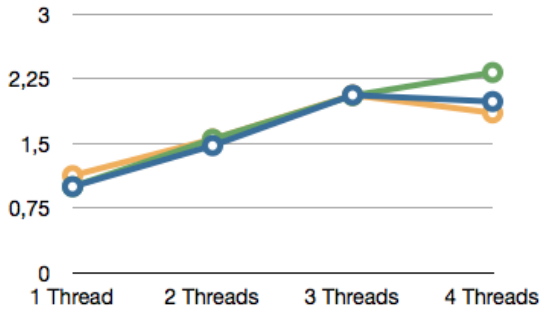
5 Conclusions

In the context of transactional memory, parallel nesting is still largely unexplored mostly due to the difficulties of supporting it efficiently [4]. To tackle this issue, this work aimed at answering the following three questions: *Q1*: Can parallel nesting improve application performance? *Q2*: Are traditional contention managers [9] adequate to parallel nesting? *Q3*: Does parallel nesting make sense in a distributed environment? Overall, the results show that parallel nesting can improve performance

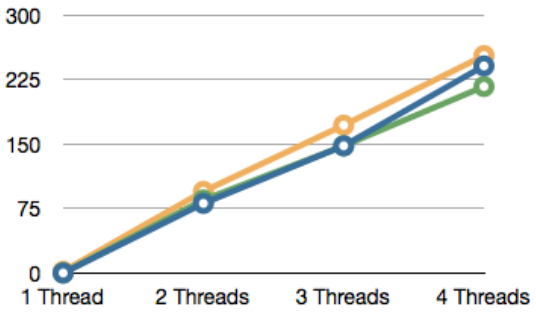


(a) Optimistic Default

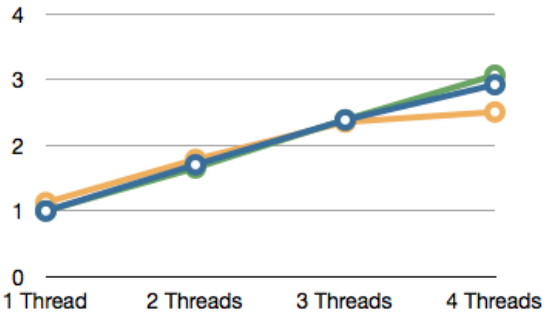
(b) Optimistic Default Aborts



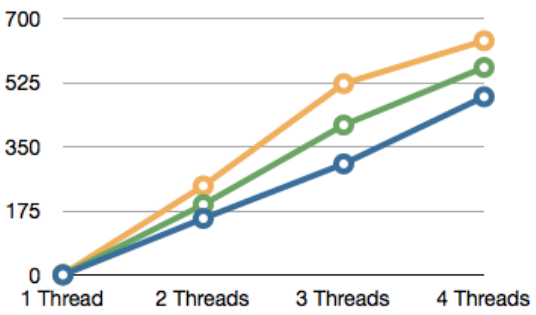
(c) Optimistic Aggressive



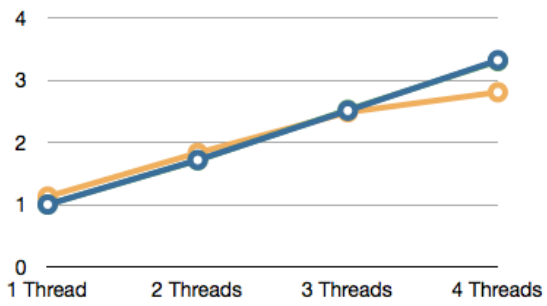
(d) Optimistic Aggressive Aborts



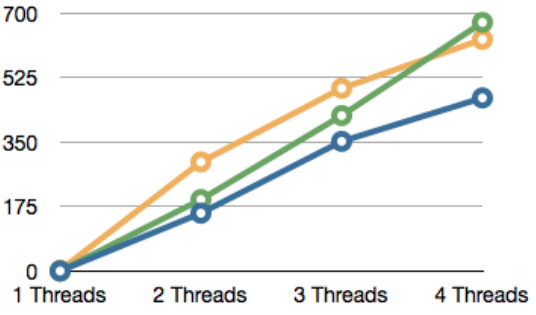
(e) Pessimistic Greedy



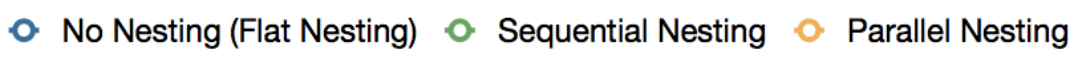
(f) Pessimistic Greedy Aborts



(g) Pessimistic Karma

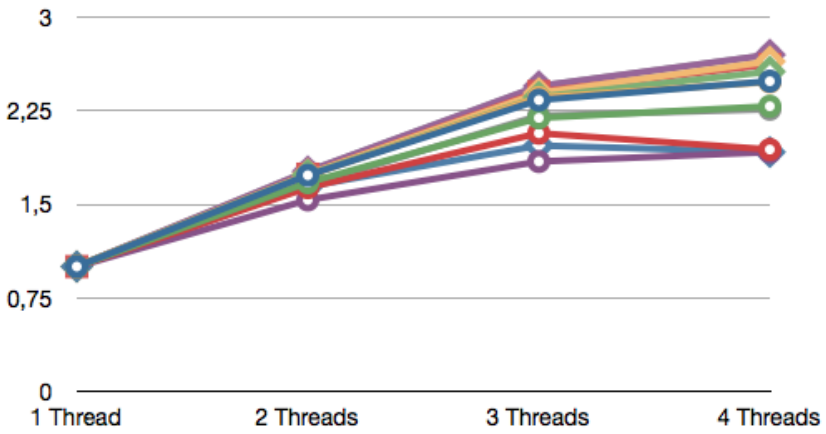


(h) Pessimistic Karma Aborts

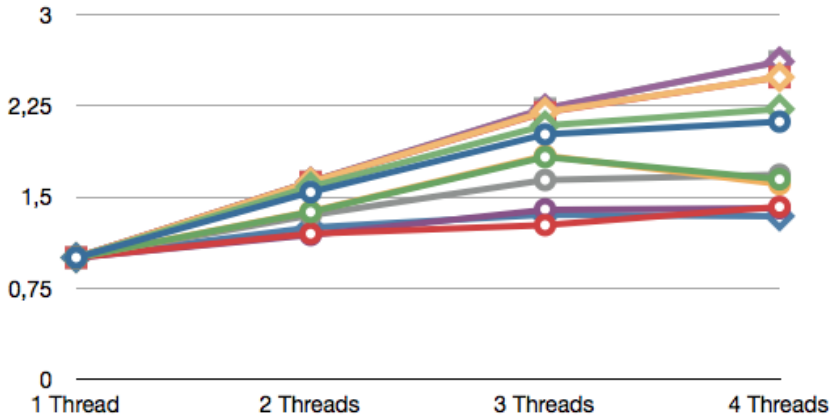


(i) Legend

Figure 1: Speedup and respective abort number for the different type of executions normalized to sequential execution code (Optimistic version of Default and Aggressive, and also Greedy and Karma)



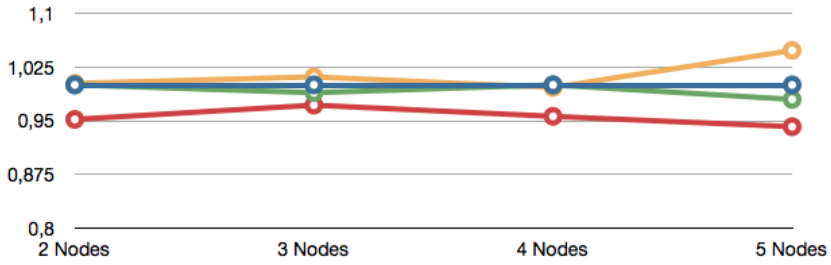
(a) High Level of Contention



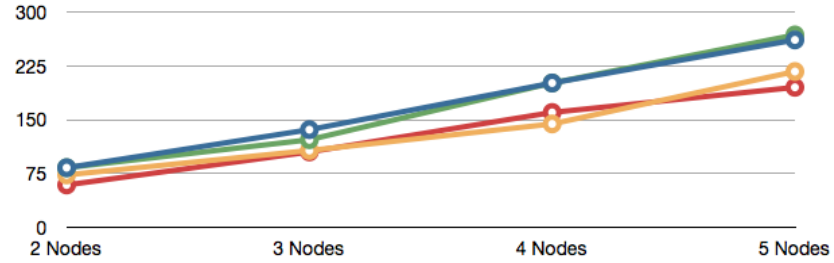
(b) Very High Level of Contention

(c) Legend

Figure 2: Speedup of Parallel Nesting normalized to sequential code for the different contention managers



(a) Speedup of distributed parallel nesting normalized to local nesting-only executions



(b) Number aborts of the different types of distributed executions

(c) Legend

Figure 3: Distributed execution with a variable number of nodes

in scenarios where it is still possible to obtain more parallelism. However, the results also show that parallel nesting tends to generate a higher abort rate and because of this, it's important for a TM system be abort friendly in order to efficiently support parallel nesting. The results for the different contention managers show that the modified versions of Karma and Eruption contention managers achieve the best performances overall when considering parallel nesting. This result shows that traditional contention managers should be adapted so as to efficiently support parallel nesting. Finally, the results also show that distributed parallel nesting is deeply affected by the overheads introduced by the extra communication steps needed, and because of this, it is outperformed by traditional distributed STM executions.

References

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, vol. 21, pp. 289–300, May 1993. [Online]. Available: http://doi.acm.org/10.1145/173682.165164

[2] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, "Software transactional memory for dynamic-sized data structures," in *PODC '03: Proc. 22nd ACM Symposium on Principles of Distributed Computing*, Jul 2003, pp. 92–101.

[3] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, 2006, pp. 194–208.

[4] J. a. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapalka, "Leveraging parallel nesting in transactional memory," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/1693453.1693466

[5] K. Agrawal, J. T. Fineman, and J. Sukha, "Nested parallelism in transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 163–174. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345232

[6] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun, "Implementing and evaluating nested parallel transactions in software transactional memory," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 253–262. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810528

[7] 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.

[8] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2stm: Dependable distributed software transactional memory," in *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, ser. PRDC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 307–313. [Online]. Available: http://dx.doi.org/10.1109/PRDC.2009.55

[9] W. N. Scherer, III and M. L. Scott, "Advanced contention management for dynamic software transactional memory," in *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, ser. PODC '05. New York, NY, USA: ACM, 2005, pp. 240–248. [Online]. Available: http://doi.acm.org/10.1145/1073814.1073861

[10] H. Volos, A. Welc, A.-R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy, "Nepaltm: Design and implementation of nested parallelism for transactional memory systems," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 123–147. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_7

[11] S. M. Fernandes and J. Cachopo, "A scalable and efficient commit algorithm for the jvstm," in *5th ACM SIGPLAN Workshop on Transactional Computing*. ACM, april 2010.

[12] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, pp. 372–421, December 2004. [Online]. Available: http://doi.acm.org/10.1145/1041680.1041682

[13] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero, "Atomic quake: using transactional memory in an interactive multiplayer game server," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 25–34. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504183

[14] N. Carvalho, J. a. Cachopo, L. Rodrigues, and A. R. Silva, "Versioned transactional shared memory for the fénixedu web application," in *Proceedings of the 2nd workshop on Dependable distributed data management*, ser. SDDDM '08. New York, NY, USA: ACM, 2008, pp. 15–18. [Online]. Available: http://doi.acm.org/10.1145/1435523.1435526

[15] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis, "Lee-tm: A non-trivial benchmark suite for transactional memory," in *Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing*, ser. ICA3PP '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 196–207. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69501-1_21

[16] R. Guerraoui, M. Kapalka, and J. Vitek, "Stmbench7: a benchmark for software transactional memory," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 315–324, March 2007. [Online]. Available: http://doi.acm.org/10.1145/1272998.1273029

[17] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, pp. 89–100, June 2007. [Online]. Available: http://doi.acm.org/10.1145/1273442.1250746

[18] R. Guerraoui, M. Herlihy, and B. Pochon, "Toward a theory of transactional contention managers," in *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, ser. PODC '05. New York, NY, USA: ACM, 2005, pp. 258–264. [Online]. Available: http://doi.acm.org/10.1145/1073814.1073863

[19] W. N. Scherer III and M. L. Scott, "Contention management in dynamic software transactional memory," in *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, Jul 2004.

[20] M. Couceiro, P. Romano, and L. Rodrigues, "Polycert: Polymorphic self-optimizing replication for in-memory transactional grids," in *The 12th International Middleware Conference, Middleware*. IST, 2011.