Efficient and Correct Transactional Memory Programs Combining Snapshot Isolation and Static Analysis

Ricardo J. Dias   João M. Lourenço   Nuno M. Preguiça

CITI and Departamento de Informática[*]

Universidade Nova de Lisboa, Portugal

{rjfd, joao.lourenco, nuno.preguica}@di.fct.unl.pt

Abstract

The use of the Snapshot Isolation (SI) level in Transactional Memory (TM) eliminates the need of tracking memory read accesses, reducing the run-time overhead and fastening the commit phase. By detecting only write-write conflicts, SI allows many memory transactions to succeed that would otherwise abort if serialized. This higher commit rate comes at the expense of introducing anomalous behaviors by allowing some real conflicting transactions to commit. We aim at improving the performance of TM systems by running programs under SI, while guaranteeing a serializable semantics. This is achieved by static analysis of TM programs using Separation Logic to detect possible anomalies when running under SI. To guarantee correct behavior, the program code can be automatically modified to avoid these anomalies. Our approach can have an important impact on the performance of single multi-core node TM systems, and also of distributed TM systems by considerable reducing the required network traffic.

1 Introduction

Multi-core architectures are becoming mainstream in computer industry. To explore the power of these new architectures programs must include multiple threads of activity. Transactional Memory (TM) [22, 15], implemented in software (STM) or hardware (HTM), was proposed as a mechanism to simplify the development of parallel programs, by providing software developers with transactions, a simple and well-known abstraction to manage concurrent accesses to shared data. The TM runtime guarantees a serializable semantics for concurrent transactions.

TM can also be used in a distributed setting to address the data sharing among processes running in different nodes. Distributed Software Transactional Memory (DSTM) inherits the same transactional model as defined for TM, but strives for different implementations techniques to better fit the distributed environment.

[*]This work was partially supported by Sun Microsystems under the *Sun Worldwide Marketing Loaner Agreement #11497*, by the Centro de Informática e Tecnologias da Informação (CITI), and by the Fundação para a Ciência e Tecnologia (FCT/MCTES) in the research projects PTDC/EIA-EIA/108963/2008 and PTDC/EIA-EIA/113613/2009 and research grant SFRH/BD/41765/2007.

TM runtime systems work by tracking the memory accesses made within transactions. This information is used to validate the consistency of the transaction at commit time. It is known that registering memory accesses is slower than the actual memory access by orders of magnitude, and is a main source of overhead in TM systems.

Database systems frequently rely on weaker isolation models to improve performance. In particular, Snapshot Isolation (SI) is widely used in industry. An interesting aspect of SI for TM systems is that conflicts among concurrent transactions are detected only by comparing write-sets. Thus, a TM that uses SI do not need to keep track of read accesses, thus considerably reducing the bookkeeping overhead.

Relaxing the isolation of a transactional program may lead previously correct programs to misbehave due to the anomalies resulting from malign data-races that are now allowed by the relaxed transactional runtime [2]. Fortunately, most programs do not exhibit such anomalies (e.g. it is known that most database benchmarks run without serializability anomalies under SI [11]). For those transactions that exhibit such anomalies, it is possible to modify the transaction code to avoid such problems [12].

In this paper, we propose a methodology to identify which transactions may lead to Snapshot Isolation anomalies in a Java based TM program using static analysis techniques. To guarantee correct behavior, the transaction code can be automatically modified to avoid these anomalies. Using this approach, we achieve improved performance by relying on the less expensive Snapshot Isolation-based TM runtime, while guaranteeing correctness of program execution by protecting all possible anomalous situations.

We build our approach to detect SI anomalies on top of Fekete et al. [12] work, which proposed to statically detect SQL anomalies in a database setting. We extend this approach to a general purpose programming language, Java, which imposes no limits to the program's data model and uses memory references (pointers) to indirectly access memory. These properties make the analysis process much more complex than standard SQL. Since the granularity of the operations inside memory transactions is at the level of read and write of memory words, we need to capture these accesses at the same granularity with the static analysis.

```
void Withdraw(boolean b, int value) {
  if (x + y > value)
    if (b) x = x - value;
    else y = y - value;
}
```

Figure 1: Withdraw program.

To this end, we use Separation Logic [20] to extract the read- and write-sets for each transaction in the program, and then use exactly the same technique as proposed by Fekete's to identify the existing anomalies.

The remainder of this paper is organized as follows. Section 2 introduces SI and the respective anomalies. In Section 3 we describe how the static analysis can be used to detect the SI anomalies and we discuss some challenges of this approach. Then, in Section 4, we present some preliminary results of using SI in concrete TM systems. In Section 5 we discuss the design of a system that integrates the use of SI and static analysis to provide anomaly-free execution of programs. Section 6 presents the related work and we finish with some concluding remarks in Section 7.

2 Snapshot Isolation

Snapshot Isolation (SI) [2] is a well known relaxed isolation level widely used in databases, where each transaction executes with relation to a private copy of the system state—a snapshot— taken at the beginning of the transaction and stored in a local buffer. All write operations are kept pending in the local buffer until they are committed in the global state. Reading modified items always refer to the pending values in the local buffer.

Tracking memory operations introduces some overhead, and TM systems running under serializable isolation level must track both memory read and write accesses, incurring in considerable performance penalties. Validating transactions in SI only requires to check if any two concurrent transaction wrote at a common data item. Hence the run-time system only needs to track the memory write accesses per transaction, ignoring the read accesses, possibly boosting the overall performance of the transactional run-time.

Although appealing for performance reasons, the use of SI may lead to non-serializable executions, resulting in two kinds of consistency anomalies: *write-skew* and *SI read-only anomaly* [12]. Consider the following example that suffers from the *write-skew* anomaly. A bank client can withdraw money from two possible accounts represented by two shared variables, x and y. The program listed in Figure 1 can be used in several transactions to perform bank operations customized by its input values. The behavior is based on a parameter b and on the sum of the two accounts. Let the initial value of x be 20 and the initial value of y be 80. If two transactions T_1 and T_2 execute concurrently, calling Withdraw(**true**, 30) and Withdraw(**false**, 90) respectively, then one possible execution history of these two transactions under SI is:

$$H = R_1(x,20)\ R_2(x,20)\ R_1(y,80)\ R_2(y,80)\ R_1(x,20)\ W_1(x,-10)\ C_1\ R_2(y,80)\ W_2(y,-10)\ C_2$$

After the execution of these two transactions the final sum of the two accounts will be -20, which is unacceptable. Such execution would never be possible under Serializable Isolation level, as the last transaction to commit would abort because it read a value that was written by the first (committed) transaction.

3 Static Analysis with Separation Logic

The foundations of static analysis to detect SI anomalies were introduced by Fekete et al. [12] and later improved by Jorwekar et al. [16]. Both works target the analysis of SQL code in the database setting. The analysis described in [12] may be divided in two phases. The first phase covers the extraction of the lookup and update accesses for each transaction, building their corresponding read and write sets. These sets are then used in the second phase to define dependency relationships between transactions that can possibly trigger an anomaly. To detect anomalies in the TM setting, only the first phase must be adapted to extract the memory read and write accesses, while the second phase can be reused with no further adaptations. In a third phase, the identifies anomalies can be corrected to make the application behave as if executing under Serializable Isolation level.

We propose to extract the read and write sets of each memory transaction present in the program by using Separation Logic [20], a shape analysis technique that models the heap using first order logic and the separation conjunction operator ($*$). To fit our goal, the symbolic execution using Separation Logic must be adapted to model the changes of the heap state rather than its validation. Using Separation Logic, the heap can be modeled as a symbolic heap [10] composed by a pure and a spatial part. The pure part captures the aliasing between variables, while the spatial part captures the structure of the heap. It is possible to capture the changes in the heap by analyzing the effects of a symbolic execution over the symbolic heap. From the evolution of the symbolic heap during a memory transaction, one can extract all the heap read and write accesses. In the end of the symbolic execution we have an abstract representation of the transaction read and write sets. This abstract representation is an over approximation of the real memory read and write accesses, which may include some false positives introduced by the analysis process.

With the information provided by the analysis we can define dependency relationships between memory transactions and identify those that can possibly trigger an anomaly. Knowing which variables are contributing to conflict, one can look for the statements where those variables are accessed and devise a correction for the anomaly, following the techniques described in [12]. Some techniques have higher performance penalties than others, so

we have to evaluate carefully which one better suites the needs of the program. The most common technique to correct SI is running the malign transaction under serializable isolation. This approach however imposes a strong performance penalty. Alternatively, one can introduce selective dummy write accesses on some of the variables, introducing potential write-write conflicts and forcing the transactions to follow the *First-Committer-Wins* rule. This later approach has potential for much better performance, but is also much harder to implement due to the difficulties in devising the correct set of dummy writes that will correct the anomaly.

3.1 Pitfalls and Challenges

The idea of correcting unsafe transactional code using static analysis is very appealing because we can provide a TM system with high performance and, simultaneously, maintain all the safety guarantees. But there are some issues that must be addressed towards this objective.

Application Nature The optimization based in the use of SI resorts on the assumption that in most applications transactions perform many more read than write memory accesses, and that the write-sets of concurrent transactions are probably disjoint. In the cases where theses assumptions do not apply, the use of SI may have reduced impact on the application's performance.

False Positives The results of the static analysis are always an over approximation of the real results. Therefore, for applications where the analysis generates a high set of false positives, the corrections addressing the false anomalies will necessarily incur in a strong the performance penalty.

The false positive rate introduced by the analysis is strongly influenced by the abstractions used to represent the heap state. Using a lower abstraction level—meaning that we manage more information—the false positive rate decreases but the analysis time and space complexity increases, possibly making it impracticable to use. By using a higher abstraction level, the analysis will run faster and will scale better for larger programs, but the false positive rate will also probably increase. A tradeoff must be found by carefully considering the kind of applications that use transactional memory. Our insight is that transactions used in implementing data structures are the most difficult to analyze, while the transactions that use those data structures will be easier.

Correction of Anomalies The correction of an anomaly is also subject of some difficulties. The main difficulty is that it may be hard or even impossible to precisely identify which code instructions trigger the anomaly, and in this case it may be also difficult to decide where to inject dummy writes to force transactions to synchronize. If a dummy write is executed in every possible control flow path of a pair of transactions, for instance at the beginning or end of each transaction, then the transactions are being serialized and the potential of parallelism inherent to the transactions is lost. In this case it would be preferable to run this transaction in serializable isolation instead of injecting dummy writes.

Validation of the System There is a major problem concerning the validation of the analysis as, to our best knowledge, there are no real world applications using transactional memory except the one described in [8]. Thus we have to resort to STM oriented data structures micro benchmarks, such as manipulating Red-Black Trees, Singly-Linked Lists and Skip Lists, or more complex ones, such as STAMP [7], Lee-TM [1], and STMBench7 [13]. The STAMP benchmark is composed by a set of applications that emulate a diversity of real world problems, being a good candidate for an overall validation of our proposals. Because of the intensive use of pointer manipulation, the data structures micro benchmarks may help in validating the precision of the static analysis and dummy write code injection.

4 Preliminary Results

As a preliminary testbed, we adapted JVSTM [5], a multiversion STM, to support SI. We ran some performance micro benchmarks comparing the throughput of a Linked List and a Skip List running in Serializable and in SI. Both implementations of the lists suffer from the write-skew anomaly when running under SI, triggered by the concurrent execution of the *insert* and *remove* transactions. The execution of the micro benchmarks under SI follows two variants. One executes the program with the anomaly present in the code, while the other executes the program with the anomaly corrected with dummy writes.

Figure 2 depicts the results of the execution of the Linked List and Skip List micro benchmarks, with a maximum of 10000 keys, for two workloads that differ in the number of update—insert and remove—transactions executed, with approximately 50% and 90% of updates. The tests were performed in a Sun Fire X4600 M2 x64 server, with eight dual-core AMD Opteron Model 8220 processors @ 2.8 GHz and 1024 KB of cache in each processor.

The serializable isolation variant corresponds to the original JVSTM algorithm. Since the JVSTM is a multiversion STM, the read-only transactions are already highly optimized and have the same performance as the read-only transactions running in the snapshot isolation variant. The observed performance improvement for SI depends only on the read-write (RW) transactions. The original JVSTM must keep track of the memory read accesses in RW transactions to latter validate the transaction at commit time, while the SI variant never tracks the memory read accesses. This performance improvement is higher when the frequency of updates increases. The SI variant per-

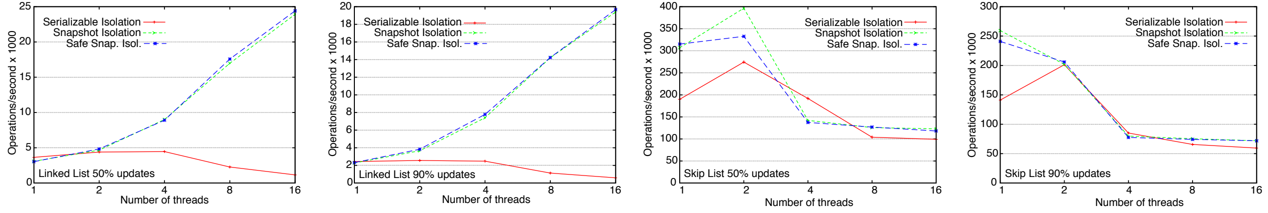

Figure 2: Linked List (left) and Skip List (right) performance throughput benchmarks with 50% and 90% of write operations.

	Read-Set		**Write-Set**		**Traffic**
	Avg	**Max**	**Avg**	**Max**	
Serializable	1992.9	4929	1.6	2	100%
Snap. Isol.	0.0	0	1.6	2	0.08%
Safe S. Isol.	0.0	0	2.0	2	0.1%

	Read-Set		**Write-Set**		**Traffic**
	Avg	**Max**	**Avg**	**Max**	
Serializable	36.3	103	2.0	20	100%
Snap. Isol.	0.0	0	2.0	22	5.2%
Safe S. Isol.	0.0	0	2.6	22	6.8%

Table 1: Read- and write-set statistic per transaction for a Linked List (left) and a Skip List (right).

forms better than the Serializable one for both lists, and scales much better for the Linked List than for the Skip List. This is due to the internal structure and organization of each data structure. The Skip List is a case where SI optimization opportunities are limited, as transactions have a much higher rate of write accesses.

Another important fact of these results is that the corrected version of the Snapshot Isolation, which is anomaly-free, has almost the same performance as the non-safe version. In this case the correction was a single dummy write introduced in the remove operation for both benchmarks.

These results on performance and scalability confirmed the potential performance benefits of using snapshot isolation in STM. To gain some insight about the potential performance gains of using SI in a Distributed Software Transactional Memory (DSTM) setting, we calculated the size of the read and write-sets for each variant. The size of the read- and write-sets is directly related to the network traffic generated by the DSTM runtime, hence we can extrapolate on the potential impact of using SI with DSTM.

Table 1 depicts the average and maximum size of the read- and write-sets for the execution of the three variants of the Linked List and Skip List micro benchmarks, with a maximum of 10000 keys and 90% of write operations. The SI variants always have an empty read-set. For the Linked List under JVSTM, the read-set has an average size of 1992.9. This result clearly depends on the nature of the benchmark application, and in the case of the Linked list, to insert a node in the middle of the list one has to traverse all nodes until the right position, implying large read sets. The average size of the write-set for all variants in both data structures is almost the same. The difference between the two SI versions is due to the dummy write introduced in the safe version of both data structures. This result allows to infer that the correction used to remove the anomaly was adequate, with a null or negligible impact on the transfer time of the write-set size over the network.

These preliminary results are encouraging and we plan to further investigate on the use of static analysis to allow the distributed transactional run-time to safely use snapshot isolation while providing serialization semantics.

5 System Design

Current STM systems are implemented either as an application library, as a language construct supported by a source-to-source compiler, or as annotations in the source code. DeuceSTM [17] is a framework for Java that supports transactions using method annotations in the source code. The programmer is required to annotate the methods that are to be executed as memory transactions and the framework transforms the Java bytecode of the application by injecting callbacks to the STM runtime. Although our preliminary results were produced using JVSTM, we intend to implement both the snapshot isolation algorithm and the static analysis in the DeuceSTM framework, exploring the already existing capabilities of this framework to perform the static analysis of Java Bytecode.

Before instrumenting the program code to inject the STM runtime callbacks, we will analyze the program using a technique based in Separation Logic to extract the read- and write-sets of each transaction in the program. Then, for each pair of transactions that may trigger an SI anomaly, we make a call to a module that is responsible for patching the Bytecode of one or both of the transactions, preventing the anomaly from happening. After the correction being made we can let DeuceSTM instrument the code with the runtime callbacks.

The DeuceSTM framework can be easily extended to support snapshot isolation (without multi-version) STM algorithms, but the real challenge is on how to support the implementation of snapshot isolation DSTM algorithms within the framework. For that purpose the instrumentation module of DeuceSTM must be modified to support distribution of object across different machines and support atomic communication primitives to allow the im-

plementation of distributed commits between distributed transactions. The framework must be able to uniquely identify each object independently from its location, and support an API to allow different implementations of the read- and write-accesses to distributed objects and different commit protocols. This will enable the implementation of a distributed snapshot isolation algorithm, and in conjunction with the initial verification of the code we can ensure the program will execute under SI as if under Serializable Isolation.

6 Related Work

Software Transactional Memory (STM) [22, 15] systems commonly implement the full serializability of memory transactions to ensure the correct execution of the programs. To the best of our knowledge, SI-STM [21] is the only current implementation of a STM using Snapshot Isolation. Their work focuses on improving the transactional processing throughput by using a snapshot isolation algorithm. They propose a SI safe variant of the algorithm where anomalies are dynamically avoided by enforcing validation of read-write conflicts. Our approach avoids this validation by using static analysis and correcting the anomalies before executing the program.

Recently, the STM community has started showing interest in transactional memory on a distributed environment. The Distributed Multiversioning (DMV) [19] is a serializable transactional memory system for clusters, based on a broadcast cache coherence scheme which distinguishes between update and read-only transactions. The Distributed Software Transactional Memory (DiSTM) [18] is built on top of the DSTM2 [14] engine with the purpose of serving as a testbed for transactional distributed algorithms. This work evaluates three broadcast-based transactional memory coherence protocols: TCC, serialization lease and multiple leases. All protocols are serializable. Bocchino et al. [4] created a highly scalable distributed transactional memory (cc-STM) that provides weak atomicity based on a PGAS model but the transactions run in serializable isolation. Couceiro et al. in [9] presents a study and implementation of four different implementations of cache coherence algorithms for distributed transactional memory systems. They reduce network traffic by using bloom-filters to encode part of the information to be transmitted. Performance evaluations show that the use of bloom filters greatly improve the performance of the overall system, even in the presence of false conflicts. Bieniusa et al. [3] present the implementation of a decentralized DSTM algorithm that executes under a variant of snapshot isolation. They avoid the anomaly problem by also tracking read accesses in read-write transactions and validate the read-set at commit time.

In our work, we aim at providing the serializability semantics under snapshot isolation for STM and DSTM systems. This is achieved by performing static analysis of the Bytecode and asserting that no SI anomalies will occur when executing the transactional application. This allow to avoid tracking read accesses in both read-only and read-write transactions, thus reducing the amount of network traffic and increasing system performance.

The use of Snapshot Isolation in databases is a common place, and there are some previous works on the detection of SI anomalies in this domain. Our work clearly builds on [12], which presents the theory of SI anomaly detection and a syntactic analysis to detect SI anomalies for the database setting. They assume applications are described in some form of pseudo-code, with no conditional (*if-then-else*) or cyclic (*for/while*) structures. The proposed analysis is informally described and applied to the database benchmark TPC-C [23]. A sequel of that work [16], describes a prototype which is able to automatically analyze database applications. Their syntactic analysis is based on the names of the columns accessed in the SQL statements that occur within the transaction. They also discuss some solutions to reduce the number of false positives produced by their analysis.

Although targeting similar results, our work deals with significantly different problems. The most significant one is related to the full power of general purpose languages and the use of dynamically allocated heap data structures. To tackle this problem, we use Separation Logic [20, 10] to model operations that manipulate heap pointers. Separation Logic has been subject of research in the last few years for its use in static analysis of dynamic allocation and manipulation of memory, allowing one to reason locally about a portion of the heap. It has been proven to scale for larger programs, such as the Linux kernel [6].

7 Conclusion

This position paper presents a framework to allow TM programs to execute under Snapshot Isolation, a relaxed isolation level, without any of the problems associated with SI anomalies. This approach allows for improved performance while guaranteeing correct program execution. The framework makes use of Separation Logic, a shape analysis technique, to detect transactions that may exhibit SI anomalies. These anomalies can be avoided by automatically modifying the transaction code.

We discuss the advantages and pitfalls of our approach. Our preliminary results show that running transactions under SI may have an important performance impact in both single-node STM and DSTM. Although the precision of the static analysis will have an important role on the performance achieved by the system, we believe that the applications that cannot be handled properly by our approach are rare. Although several works have previously proposed the use of SI in transactional memory, to our knowledge, ours is the first to address the problem of statically enforcing execution correctness.

References

[1] ANSARI, M., KOTSELIDIS, C., WATSON, I., KIRKHAM, C., LUJÁN, M., AND JARVIS, K. Lee-tm: A non-trivial benchmark suite for transactional memory. In *Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing* (Berlin, Heidelberg, 2008), ICA3PP '08, Springer-Verlag, pp. 196–207.

[2] BERENSON, H., BERNSTEIN, P., GRAY, J. N., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ANSI SQL isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1995), ACM, pp. 1–10.

[3] BIENIUSA, A., AND FUHRMANN, T. Consistency in hindsight: A fully decentralized stm algorithm. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (april 2010), pp. 1 –12.

[4] BOCCHINO, R. L., ADVE, V. S., AND CHAMBERLAIN, B. L. Software transactional memory for large scale clusters. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (New York, NY, USA, 2008), ACM, pp. 247–258.

[5] CACHOPO, J. A., AND RITO-SILVA, A. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program. 63* (December 2006), 172–185.

[6] CALCAGNO, C., DISTEFANO, D., O'HEARN, P., AND YANG, H. Compositional shape analysis by means of bi-abduction. In *Proc. of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2009), POPL '09, ACM, pp. 289–300.

[7] CAO MINH, C., CHUNG, J., KOZYRAKIS, C., AND OLUKOTUN, K. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization* (September 2008).

[8] CARVALHO, N., CACHOPO, J. A., RODRIGUES, L., AND SILVA, A. R. Versioned transactional shared memory for the fénixedu web application. In *Proceedings of the 2nd workshop on Dependable distributed data management* (New York, NY, USA, 2008), SDDDM '08, ACM, pp. 15–18.

[9] COUCEIRO, M., ROMANO, P., CARVALHO, N., AND RODRIGUES, L. D2stm: Dependable distributed software transactional memory. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing* (Washington, DC, USA, 2009), PRDC '09, IEEE Computer Society, pp. 307–313.

[10] DISTEFANO, D., O'HEARN, P. W., AND YANG, H. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of 12th International Conference (TACAS 2006)* (2006), Lecture Notes in Computer Science, Springer, pp. 287–302.

[11] ELNIKETY, S., DROPSHO, S., AND PEDONE, F. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (New York, NY, USA, 2006), EuroSys '06, ACM, pp. 117–130.

[12] FEKETE, A., LIAROKAPIS, D., O'NEIL, E., O'NEIL, P., AND SHASHA, D. Making snapshot isolation serializable. *ACM Trans. Database Syst. 30*, 2 (2005), 492–528.

[13] GUERRAOUI, R., KAPALKA, M., AND VITEK, J. Stmbench7: a benchmark for software transactional memory. In *Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 315–324.

[14] HERLIHY, M., LUCHANGCO, V., AND MOIR, M. 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* (New York, NY, USA, 2006), ACM, pp. 253–262.

[15] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND WILLIAM N. SCHERER, I. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing* (New York, NY, USA, 2003), ACM, pp. 92–101.

[16] JORWEKAR, S., FEKETE, A., RAMAMRITHAM, K., AND SUDARSHAN, S. Automating the detection of snapshot isolation anomalies. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases* (Vienna, Austria, 2007), VLDB Endowment, pp. 1263–1274.

[17] KORLAND, G., SHAVIT, N., AND FELBER, P. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG 10)* (January 2010).

[18] KOTSELIDIS, C., ANSARI, M., JARVIS, K., LUJAN, M., KIRKHAM, C., AND WATSON, I. Distm: A software transactional memory framework for clusters. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on* (Sept 2008), pp. 51–58.

[19] MANASSIEV, K., MIHAILESCU, M., AND AMZA, C. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2006), ACM, pp. 198–208.

[20] REYNOLDS, J. C. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science* (Washington, DC, USA, 2002), LICS '02, IEEE Computer Society, pp. 55–74.

[21] RIEGEL, T., FETZER, C., AND FELBER, P. Snapshot isolation for software transactional memory. In *TRANSACT'06: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing* (Ottawa, Canada, June 2006).

[22] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1995), ACM, pp. 204–213.

[23] TRANSACTION PROCESSING PERFORMANCE COUNCIL. *TPC-C Benchmark, Standard Specification, Revision 5.11*. Feb. 2010.