Snapshot Isolation Anomalies Detection in Software Transactional Memory

Ricardo J. Dias, João Costa Seco, and João M. Lourenço[⋆]

CITI — Departamento de Informática,
Universidade Nova de Lisboa, Portugal
{rjfd,joao.seco,joao.lourenco}@di.fct.unl.pt

Abstract. Some performance issues of transactional memory are caused by unnecessary abort situations where non serializable and yet non conflicting transactions are scheduled to execute concurrently.
Smartly relaxing the isolation properties of transactions may overcome these issues and attain considerable performance improvements. However, it is known that relaxing isolation restrictions may lead to runtime anomalies. In some situations, like database management systems, developers may choose that compromise, hence avoiding anomalies explicitly. Memory transactions protect the state of the program, therefore execution anomalies may have more severe consequences in the semantics of programs. So, the compromise between a relaxed isolation strategy and enforcing the necessary program correctness is harder to setup.
The solution we devise is to statically analyse programs to detect the kind of anomalies that emerge under snapshot isolation. Our approach allows a compiler to either warn the developer about the possible snapshot isolation anomalies in a given program, or possibly inform automatic correctness strategies to ensure Serializability.

Keywords: Snapshot Isolation, Serializable Anomalies, Software Transactional Memory, Static Analysis

1 Introduction

Concurrent programming is becoming mainstream due to the widespread use of multicore processors. Locks are an effective but low-level mechanism to control concurrent threads of execution in such systems, and there is a clear demand for more abstract programming mechanisms.

Concurrency control in Database Systems is achieved by using transactions, that usually comply with the standard properties of atomicity, consistency, isolation and durability (ACID). To achieve increased performance, transactional

⋆ This work was partially supported by Sun Microsystems and Sun Microsystems Portugal 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 Byzantium research project PTDC/EIA/74325/2006, research grant SFRH/BD/41765/2007, and the Streamline research project PTDC/EIA-CCO/104583/2008.

frameworks sometimes relax those properties and allow transactions to execute under more relaxed isolation levels [17,1]. In particular, databases frequently provide the developer the ability to choose among different isolation levels.

More relaxed isolation levels naturally lead to increased overall performance of the transactional system, but also to the triggering of transactional anomalies, such as dirty and unrepeatable reads. Serializability is the strongest isolation level. Snapshot Isolation (SI) is a relaxed isolation level that also avoids anomalies such as unrepeatable and dirt reads. However, SI still allows some other transactional anomalies such as *write skew* and *SI read-only* [7]. Many database applications are known to execute correctly under SI, thus making it a good compromise between correction—which concurrency anomalies are admitted and how do they affect the applications—and performance.

Transactional Memory (TM) was proposed as an alternative programming abstraction for concurrency control in multithreaded programs [16,11]. TM frameworks typically operate in full serializable mode and do not allow one to relax the isolation level. Thus, the potential of Snapshot Isolation for performance improvement, a *de facto* standard for the database world, has been neglected in TM programming until now. Figure 1 illustrates the potential of such improvement by means of a small experiment with a transactional memory benchmark executed in a Sun Fire x4600 with 16 cores. The benchmark operates over a linked list, executing insert, delete and lookup operations. In this example, one can observe that while the Serializable version does not scale under the increasing number of processors/threads, while the Snapshot Isolation version scales almost linearly with the number of processors/threads (note that the scale in the X-axis is logarithmic).

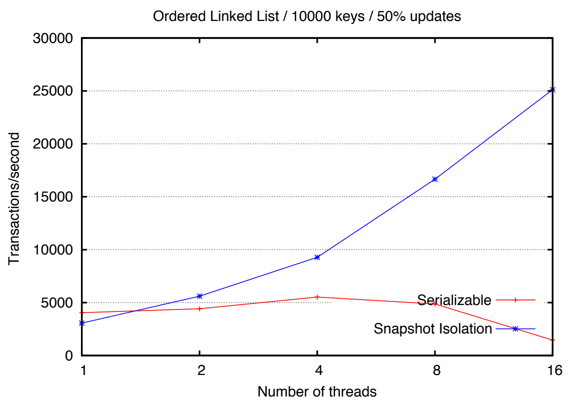

Fig. 1. Snapshot Isolation vs. Serializability in Transactional Memory

The above example is a strong motivation to further study how to use relaxed isolation levels, and in particular Snapshot Isolation, in the transactional memory setting. Unlike the database approaches where a domain specific language (SQL) is used to model database accesses, TM programs are usually defined in a general purpose programming language, and there is no evidence that there exists a large set of applications that will also execute correctly under weaker TM isolation levels without serious rewriting.

This work aims at asserting that a multithreaded transactional memory program will not trigger the well known SI anomalies when executing under Snapshot Isolation, thus leading to non-serializable executions. In this case, the application will execute as if under Serializable isolation level. Our work grounds in previous work in static detection of SI anomalies in databases [7], but our approach targets the very different domain of Transactional Memory.

As a testbed for our work, we defined a simple imperative language, with no support for pointers. Each transaction is an instance of a program written in this language. We perform a data-flow analysis over each program, extracting the information necessary to detect if the concurrent execution of a set of transactions will generate a serializable anomaly. If the analysis detects no serializable anomalies, than the application will execute correctly. In the opposite, if serialization anomalies are found, they should be considered as possible anomalies and confirmed by other means, as our analysis allow for false positives.

The main contributions of this work include:

– A new data-flow analysis to extract information from transactional programs. This analysis will extract compact read- and write-sets in order to define static dependencies between programs.
– The definition of static dependency between transactional programs using the information retrieved from the static analysis.
– A version of the algorithm proposed in [7] to detect SI anomalies, adapted and optimized for the TM setting. The algorithm will operate over a graph of static dependencies between programs, and will determine if the execution of such programs under SI will be serializable.

The rest of the paper is organized as follows: Section 2 describes the Snapshot Isolation model and the definition of serializable anomalies using static dependencies between programs. Section 3 describes the data-flow static method to gather information about read and write accesses in transactional programs, the procedure to generate the static dependencies using this information, and the algorithm to detect serializable anomalies in the static dependency graph. Section 4 discusses the relations among our work and the related ones. Finally, Section 5 presents some concluding remarks and discusses our plans of evolution of this work.

2 Snapshot Isolation

Snapshot Isolation [1] is a weaker isolation level than Serializable where each transaction performs its read operations in a private snapshot of the state, taken in the beginning of the transaction. All write operations performed by the transaction are stored in a local buffer. All read operations on data items previously written by the transaction are performed from its local buffer.

Considering that the lifetime of a successful transaction is the time span that goes from the moment it starts $start(T_i)$ until the moment it commits $commit(T_i)$. Two successful transactions T_1 and T_2 are said to be concurrent if:

$$[start(T_1), commit(T_1)] \cap [start(T_2), commit(T_2)] \neq \emptyset \tag{1}$$

The write operations of a transaction T_i are not visible to the remaining concurrent transactions. When a transaction T_i is ready to commit, it obeys the *First-Commiter-Wins* rule, which states that it can successfully commit only if there is not a concurrent transaction T_k $(i \neq k)$ which has committed write operations to some item that T_i is also changing. This means that if there are two concurrent transactions updating the same data item, only the first one to commit will succeed.

Snapshot Isolation has some advantages over the Serializable isolation level. By always reading from a snapshot, read-only transactions will never abort. A read-only transaction T_i only sees committed results before $start(T_i)$. Also, read-only transactions will never make read-write transactions to abort.

Snapshot Isolation sounds very appealing, however its application may lead to non serializable executions. These executions result in consistency anomalies that may happen when using Snapshot Isolation [7], namely the *write-skew* and *SI read-only* anomalies.

2.1 Static Isolation Anomalies

Other works have defined serializable anomalies under Snapshot Isolation in terms of database program dependencies [7]. In this work we use the same static dependency definition and adapt it for software transactional memory programs.

The SI anomalies can be described formally using static dependencies between transactional programs. Two transactional programs have a static dependency between them if both programs access the same data item and at least one of them performs a write access. Four types of static dependencies are defined in [7]:

- $P_i \xrightarrow{x-ww} P_j$: The transaction resulting from the execution of program P_i writes data item x and commits, and the transaction resulting from the execution of program P_j also writes data item x and commits.
- $P_i \xrightarrow{x-wr} P_j$: The transaction resulting from the execution of program P_i writes data item x and commits, and the transaction resulting from the execution of program P_j reads data item x, written by P_i, and commits.
- $P_i \xrightarrow{x-rw} P_j$: The transaction resulting from the execution of program P_i reads data item x and commits, and the transaction resulting from the execution of program P_j writes data item x, read by P_i, and commits, and programs P_i and P_j are not concurrent.
- $P_i \overset{x-rw}{\Rightarrow} P_j$: The transaction resulting from the execution of program P_i reads data item x and commits, and the transaction resulting from the execution of program P_j writes data item x, read by P_i, and commits, and programs P_i and P_j are concurrent.

The first three dependencies are said to be *non-vulnerable* dependencies and the last one is said to be a *vulnerable* dependency. Using these dependencies we can build a *Static Dependency Graph* [7] (SDG) where programs correspond to nodes and static dependencies correspond to edges.

The relation between the unsatisfiability of the Serializable property and static dependencies between programs can be signalled by the existence of certain kinds of dangerous structures in the SDG of an application.

Fekete et al. [7] defines the concept of *dangerous structure* in a static dependency graph. He shows that if some $SDG(\mathcal{A})$ has a dangerous structure then there are executions of application $\mathcal{A}$ which may be not serializable, and that if a $SDG(\mathcal{A})$ does not have any dangerous structure then all executions of application $\mathcal{A}$ are serializable.

Definition 1 (Dangerous structures) *We say that a $SDG(\mathcal{A})$ has a dangerous structure if it contains nodes P, Q and R (not necessarily distinct) such that:*

– *There is a vulnerable edge from R to P.*
– *There is a vulnerable edge from P to Q.*
– *Either $Q = R$ or there is a path in the graph from Q to R; that is, (Q, R) is in the reflexive transitive closure of the edge relationship.*

The detection of dangerous structures in a SDG can be performed algorithmically.

We next show how to build an SDG by analyzing the source code of an application and how to detect dangerous structures that point to possible anomalies.

3 Static Analysis

In this section we define a new static analysis procedure for a small imperative language and describe how to build a *Static Dependency Graph* [7] with the information given by the analysis. We next define how to detect execution anomalies.

The following grammar defines the abstract syntax of an imperative language:

$$\begin{array}{rcl}
\langle E \rangle & ::= & x \mid n \mid \langle E \rangle\ op\ \langle E \rangle \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{not}\ \langle E \rangle \\
\langle S \rangle & ::= & x := \langle E \rangle \mid \mathbf{skip} \mid \langle S \rangle\ ;\langle S \rangle \\
 & \mid & \mathbf{if}\ \langle E \rangle\ \mathbf{then}\ \langle S \rangle\ \mathbf{else}\ \langle S \rangle \mid \mathbf{while}\ \langle E \rangle\ \mathbf{do}\ \langle S \rangle \\
\langle P \rangle & ::= & \langle S \rangle
\end{array}$$

This language has integer (n), boolean literals (**true** and **false**), and variables (x). It contains the usual binary arithmetic, logic, and relational operations ($E_1\ \ op\ \ E_2$). The statements of the language include the conditional and loop statement as well as the variable assignment. We consider that an application is a set of programs defined over a set of shared variables, and each program corresponds to a single memory transaction. We apply the analysis separately to each program.

3.1 Read-Write Analysis

In order to define static dependencies between programs we need to know which data items are read or written by each program that comprises an application. Thus, we use a standard data-flow static analysis to obtain the set of variables read or written by a program. For that purpose we have defined a custom lattice and the appropriate transfer functions.

We establish a state for each shared variables for each node in the control-flow graph of a program. The state of a shared variable is a pair of values of the set $\Gamma = \{?, M, m, \top\}$. The first component of the pair indicates if a variable was read—its read state—and the second component of the pair indicates its write state. A "?" value in the read/write state for a variable x means that x is not read/written by the program. A "M" value in the read/write state for a variable x means that x is indeed read/written by the program. A "m" value in the read/write state for a variable x means that x may be read/written by the program (it is read/written in at least one possible execution path, but not in all). In Figure 2 is depicted the relation order of the lattice Γ.

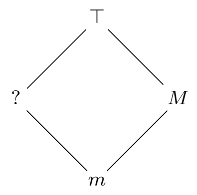

Fig. 2. Lattice Γ order relation diagram.

We now define the data-flow transfer functions over a lattice defined over the set $\Upsilon = \Gamma \times \Gamma \times Var$. The elements of the tuples denote the read state the write state and a shared variable (Var). An element of set Υ of the form $(M, m)_x$ means that variable x is read in every possible execution of the program and is written at least in one possible execution of the program (but not all).
The transfer functions Z_{en} and Z_{ex} map the labels of a control flow to Υ: $Z_{en}, Z_{ex} : \mathit{Label} \longrightarrow \Upsilon$.

Labels are identifiers of the program nodes in the control flow graph (CFG). Each node in the CFG as an entry and an exit point, and functions Z_{en} and Z_{ex} correspond to the entry and exit points respectively.

We define the analysis as a backward procedure by the following functions Z_{en} and Z_{ex}:

Definition 2 (Exit function)

$$Z_{ex}(l) = \begin{cases} \{(?, ?)_x \mid x \in FV(P)\} & \textit{if } l = final(P) \\ \sqcap\{Z_{en}(l') \mid (l', l) \in flow^R(P)\} & \textit{otherwise} \end{cases} \qquad (2)$$

where P represents the program we are analyzing, and $final(P)$ denotes the final label that program. $flow^R(P)$ denotes the set of reversed edges of the CFG. The operator $\sqcap$ denotes the greatest lower bound of the sets given by Z_{en}. In the beginning of the analysis, the subset of Υ is initialized with all variables belonging to the set of *Free Variables* and their state is set to $(?, ?)$ (not read nor written). The second entry of this function uses the greatest lower bound operator $\sqcap$ to join the information from more than one node, e.g., the `then` and `else` branches. Because we are using backward analysis, the beginning corresponds to the final label of the program.

Before we present the definition of the Z_{en} function we need to define a binary operator $\oplus$ which is used to modify the read/write state of a variable in a subset of Υ, $(\oplus : \Upsilon \times \Upsilon \longrightarrow \Upsilon)$: For all sets $U, V \subseteq \Upsilon$, $U \oplus V = U \setminus \big\{(t,s)_x : (t,s)_x \in U \wedge x \in Vars(V)\big\} \cup V$. Where $Vars(V)$ denotes the set of the variables present in V. The intuition of the operator $\oplus$ is: given $U \subseteq \Upsilon$, which corresponds to the original set, and $V \subseteq \Upsilon$ which corresponds to the set with the modified elements, the $U \oplus V$ operation will remove the elements of U which are also in V that have the same variable, and will return the set of the unmodified values of U with the values of V. This operator is used to override the read/write state of a variable by removing the previous information (in U) and adding the new information (in V). Now we will define the Z_{en} function:

Definition 3 (Entry Function)

$$Z_{en}(l) = \begin{cases} Z_{ex}(l) \oplus \big\{\big(M, \hat{\sigma}_y(Z_{ex}(l))\big)_y : \forall_y \in FV(E)\big\} \\ \qquad \oplus \big\{\big(\bar{\sigma}_x(Z_{ex}(l)), M\big)_x\big\} & \textit{if } B^l = [x := E]\textit{, where } B^l \in blocks(P) \\ Z_{ex}(l) \oplus \big\{\big(M, \hat{\sigma}_y(Z_{ex}(l))\big)_y : \forall_y \in FV(E)\big\} & \textit{if } B^l = [E]\textit{, where } B^l \textit{ is an elementary block with label } l \\ Z_{ex}(l) & \textit{otherwise} \end{cases} \tag{3}$$

Functions $\bar{\sigma}/\hat{\sigma}$ denote the value of the read/write state of a single variable $(\bar{\sigma}, \hat{\sigma} : Var \times \Upsilon \longrightarrow \Gamma)$. The $FV(P)$ function represents the set of free variables in program P.

The first case of Definition 3 introduces the modifications to the read/write state caused by an assignment block, where we change the write state of the assigned variable to M. We also change the read state of all free variables on the right side of the assignment to value M. The second case of the definition treats the evaluation of an expression where we change the read state of all the expression's free variables to value M. The last case in the definition corresponds to the unmodified propagation of the read/write state.

3.2 Generating Static Dependencies

If we consider an application as a set of programs that maybe launched in parallel, and given the read/write set analysis for all those programs, as described in Sect. 3.1, we can compare the set of read and write states of each program with all the other programs and create a Static Dependency Graph (*SDG*). Since we do not know *a priori* what programs will execute in parallel we pessimistically assume that all programs are concurrent even with several instances of themselves. Building a *SGD* graph requires $\binom{n}{2} + n$ comparisons. If we consider a large number of programs running in parallel, this may become unbearable. Other techniques can be used, such as the May-Happen-in-Parallel Analysis by Duesterwald and Soffa [5], to determine which programs will execute in parallel and hence help reducing the complexity of the graph construction procedure.

For all pair of programs (P_i, P_j) we compare the two corresponding read-/write states (subsets of Υ resulting from the R/W analysis) and produce a static dependency in the graph. The kind of dependency created depends on the read/write state and also varies if the two programs are concurrent or not. We say that two programs P_i and P_j are not concurrent if they have a write state M for the same variable. By the *First-Commiter-Wins* rule the execution of these two programs is always synchronized and if they run in parallel, one will necessarily abort.

There is a dependency relation between two programs if both access at least one shared data item that is modified by at least one of those programs. Static dependencies are defined from the analysis as follows:

Definition 4 *[Static Dependencies]*
For all programs P_i, P_j in an application, and with the read/write states $U_i, U_j \subseteq \Upsilon$ where U_i is the read/write state of P_i and U_j is the read/write state of P_j. If there is a variable x such that $(\_, w)_x \in U_i$ and $(r, \_)_x \in U_j$ where $w \neq\ ?$ and $r \neq\ ?$ then:

1. *if $(\_, \alpha)_x \in U_i$ and $(\_, \beta)_x \in U_j$ where $\alpha \neq\ ?$ and $\beta \neq\ ?$ then $P_i \xrightarrow{ww} P_j$*
2. *if $(\_, \alpha)_x \in U_i$ and $(\beta, \_)_x \in U_j$ where $\alpha \neq\ ?$ and $\beta \neq\ ?$ then $P_i \xrightarrow{wr} P_j$*
3. *if $(\alpha, \_)_x \in U_i$ and $(\_, \beta)_x \in U_j$ where $\alpha \neq\ ?$ and $\beta \neq\ ?$, and P_i and P_j are not concurrent then $P_i \xrightarrow{rw} P_j$*
4. *if $(\alpha, \_)_x \in U_i$ and $(\_, \beta)_x \in U_j$ where $\alpha \neq\ ?$ and $\beta \neq\ ?$, and P_i and P_j are concurrent then $P_i \overset{rw}{\Rightarrow} P_j$*

It is important to note that comparing two programs P_1 and P_2 implies comparing them in both directions. Most of the times this comparison generates dependencies in both ways. For instance, if we consider programs P_1 and P_2 such that $U_1 = \{(M, m)_x\}$ and $U_2 = \{(m, M)_x\}$, if we compare U_1 with U_2 there is a dependency $P_1 \xrightarrow{wr} P_2$ because P_1 may write variable x that is later read by P_2, and there is also a dependency $P_2 \xrightarrow{wr} P_1$ because P_2 may write variable x which may be read by P_1.

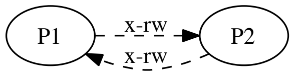

Fig. 3. An *SDG* with a cycle of read-write dependencies for the same variable.

Note that we generate non-vulnerable dependencies if we know that programs P_i and P_j are not concurrent. Otherwise we generate vulnerable dependencies between them.

As an example, consider the program P with state $U = \{(M, m)_x, (?, m)_y\}$ resulting from its data flow analysis. If we compare program P with itself the dependencies generated according to Definition 4 are:

- $P \xrightarrow{ww} P$: several instances of P write variable x or y.
- $P \xrightarrow{wr} P$: P may write variable x, which is read by another instance of P.
- $P \overset{rw}{\Rightarrow} P$: P reads variable x which may be written by another instance of P.

Filtering False Positives When traversing the edges in a *SDG*, consider the situation presented in Figure 3 where there are two vulnerable read-write dependencies for the same variable x forming a cycle between P_1 and P_2. In this case, the application of Definition 1 would identify this as a dangerous structure. Note that P_1 has an incoming vulnerable edge from P_2 and has an outgoing vulnerable edge to P_2, and there is (null length) path cyclic from P_2 to itself.

Now, consider an execution H with two transactions T_1 and T_2, result of the execution of P_1 and P_2 respectively. We argue that is not possible to define an execution H under Snapshot Isolation if there is a transactional dependency $T_1 \xrightarrow{x-rw} T_2$ and a transactional dependency $T_2 \xrightarrow{x-rw} T_1$. Consider that T_1 is executing concurrently with T_2 and there is a dependency $T_1 \xrightarrow{x-rw} T_2$ which states that T_1 reads variable x and T_2 writes variable x, and there is also a dependency $T_2 \xrightarrow{x-rw} T_1$ which states that T_2 reads variable x and T_1 writes variable x. This means that T_1 and T_2 are concurrent, that both write to variable x, and that both commit. However, by the *First-Committer-Wins* rule, one of the two transactions should have aborted, hence it is not possible to define such an execution. This incompatibility between edges also applies to the other kind of dependencies.

Given these observations, if we apply the definition of dangerous structures (Definition 1) to the *SDG* of Figure 3 and follow the edge $P_1 \overset{rw}{\Rightarrow} P_2$ then we can ignore the edge of the same kind for the same variable in the opposite direction, the edge $P_2 \overset{rw}{\Rightarrow} P_1$.

An extension to Definition 1 was made to enable the check for incompatible edges. Algorithm 1 presents the pseudo-code to detect dangerous structures. The *compatible* function will test if an edge e is compatible with the history of edges already visited.

Algorithm 1: Dangerous Structure detection algorithm with incompatibility check.

```
Data: nodes[], edges[], visited[]
Result: true or false
initialization;
foreach Node n : nodes do
    foreach Edge in : incoming(n, edges) do
        if vulnerable(in) then
            add(visited, in);
            foreach Edge out : outgoing(n, edges) do
                if vulnerable(out) and compatible(out, visited) then
                    add(visited, out);
                    if existsPath(target(out), source(in), visited) then
                        return true;
                    end
                end
            end
        end
    end
    clear(visited);
end
return false;
```

4 Related Work

Software Transactional Memory (STM) [16,11] (TM) is a new approach to concurrent programming, promising both, an efficient usage of parallelism and a powerful semantics for concurrency constraint. STM applies the concept of transactions, widely known from the Databases community, into the management of data in main memory. STM promises to ease the development of scalable parallel applications with performance close to finer grain locking but with the simplicity of coarse grain locking.

Memory transactions must only ensure two of the ACID properties: Atomicity and Isolation. The Consistency property is more relaxed as volatile memory does not have a fixed logical structure, like a database system does, over which one can make referential consistency assertions. And the Durability property may be dropped, as memory transactions operate in volatile memory (RAM), a non-persistent data storage.

In the past few years, several STM frameworks have been developed. Most of the STM frameworks take the form of software libraries, providing an API to export the transactional interface to the application [4,2,11,13]. This library-based approach allows the rapid prototyping of algorithms and their performance evaluation. Some other STM frameworks extend existing programming languages with transactional constructs supported directly by the compiler [10,8,14]. Most of these frameworks focus in managed languages such as Java, C#, and Haskell, while some other target unmanaged languages like C and C++.

All the above referenced works implement Serializable isolation to guarantee the correct execution of transactional memory programs. Only [15] implements a STM using Snapshot Isolation called SI-STM. In this work, the authors also proposed a SI safe version where SI anomalies were automatically avoided by the algorithm. Our approach is different since we do not require modifications to existing SI algorithms and we perform a static analysis to assert if a program will execute correctly under SI. To our best knowledge there is no other work in software transactional memory that follows this static approach to detect Serializable anomalies.

The use of Snapshot Isolation in databases is a common place, and there are some works related to the detection do SI anomalies. Our work is clearly inspired in [7]. This work proposes a static analysis methodology for database applications aiming at detecting SI anomalies. Their static analysis was described informally and was applied *ad-hoc* to the database benchmark TPC-C. The work presented in [12] describe a prototype which was already able to analyze database applications automatically, and also presented some solutions to reduce the number of false positives, but shared the theoretical base with [7]. There is another work described in [3] that detect and prevent SI anomalies dynamically with runtime information in database applications.

5 Concluding Remarks

Although static verification of Snapshot Isolation anomalies is not a new topic in database applications, in software transactional memory the use of Snapshot Isolation is much unexplored. The transactional anomalies triggered by the use of SI in transactional memory programs have strong impact in the execution correctness of such programs and is a major drawback to its widespread.

With this preliminary work we show that it may be possible to give stronger guarantees of correct execution under SI which allows to further explore SI algorithms in the context of STMs. We presented a simple data-flow analysis to extract the information of read and write accesses to variables in transactional programs. It gives good results as it does not allow any false negatives but allows some false positives which requires the programmer to verify by hand. Future work will target more complex language with pointers, and also towards techniques to correct programs that are detected to have Serializable anomalies, without changing its semantics and with low impact in their performance.

References

1. Hal Berenson, Phil Bernstein, Jim N. Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, New York, NY, USA, 1995. ACM.
2. João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

3. Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4):1–42, 2009.
4. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Distributed Computing*, volume 4167, pages 194–208. Springer Berlin / Heidelberg, October 2006.
5. Evelyn Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 36–48, New York, NY, USA, 1991. ACM.
6. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.
7. Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.
8. Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying applications using an open compiler framework. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, August 2007.
9. Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM.
10. Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.
11. Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.
12. Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1263–1274. VLDB Endowment, 2007.
13. Dalessandro Luke, Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Capabilities and limitations of library-based software transactional memory in c++. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, August 2007.
14. Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for c/c++. *SIGPLAN Not.*, 43(10):195–212, 2008.
15. Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *TRANSACT06*, Jun 2006.
16. Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.
17. Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, fifth edition, 2006.