Verification Tools for Transactional Programs

Adrian Cristal[1], Burcu Kulahcioglu Ozkan[2], Ernie Cohen, Gokcen Kestor[3], Ismail Kuru[2], Osman Unsal[1], Serdar Tasiran[2], Suha Orhun Mutluergil[2], and Tayfun Elmas[4]

[1] Barcelona Supercomputing Center, Barcelona, Spain
[2] Koc University, Istanbul, Turkey
[3] Pacific Northwest National Laboratory, Richland, WA
[4] Google, Mountain View, CA

Abstract. While transactional memory has been investigated intensively, its use as a programming primitive by application and system builders is only recently becoming widespread, especially with the availability of hardware support in mainstream commercial CPUs. One key benefit of using transactional memory while writing applications is the simplicity of not having to reason at a low level about synchronization. For this to be possible, verification tools that are aware of atomic blocks and their semantics are needed. While such tools are clearly needed for the adoption of transactional memory in real systems, research in this area is quite preliminary. In this chapter, we provide highlights of our previous work on verification tools for transactional programs.

1 Introduction

The verification of both sequential and concurrent programs using static and dynamic methods has been a field of intense study. Much research has also concentrated on specifying and verifying transactional memory (TM) algorithms and implementations. However, for transactional programs, whether the transactions employed are TM, database or distributed-system transactions, verification tool support is quite preliminary. This is not only the case for verifying data structure and program invariants, but also for simpler generic safety properties such as race freedom or the absence of null pointer dereferences. In this chapter, as representative of research in this space, we give an overview of three approaches we have worked on for verifying programs that mix transactional and non-transactional accesses.

In Section 2, we motivate the problem of verifying assertions, data structure and program invariants for transactional programs. Since the few concurrent program verification tools that can handle practical programming languages are not aware of even strong atomicity semantics for transactions, the work described in this section is the first of its kind that provides a workable tool to TM users. The static verification of properties for transactional programs becomes both more involved and more necessary when the TM platform being used provides more relaxed consistency guarantees such as snapshot isolation for performance reasons.

The technique presented in Section 2 builds on the VCC tool for verifying concurrent C programs and provides tool support for transactional programs running on relaxed platforms.

Section 3 focuses on dynamic techniques for detecting concurrency errors for programs that mix transactional and non-transactional accesses. A variety of programming disciplines and platform support for this setting have been investigated. In this section, we provide highlights of two dynamic race-detection techniques and tools that have been applied successfully. While the Goldilocks race and transaction-aware runtime described in Section 3.1 provides `DataRaceException` as a programming language construct for transactional program, the T-Rex tool in Section 3.2 is intended to be a debugging tool used to avoid undesirable interference between transactional and non-transactional accesses.

This chapter is not intended to exhaustively cover the literature on verifying properties of transactional programs. Rather, by presenting at some length three approaches we have worked on, we intend to highlight both the correctness and the tool-building concerns in this area. As transactional programs find wider use, especially because of commonly-available hardware TM support on general-purpose CPUs, we believe the need for such tools will intensify.

2 Static Verification for Transactional Programs

Transactions provide a convenient, composable mechanism for writing concurrent and distributed programs. A transactional execution platform can provide a strong or more relaxed programming semantics. The former simplifies program construction and verification, while the latter provides better performance and availability. This section is about a technique for verifying transactional programs that operate under relaxed semantics.

Static tools for code verification targeted at sequential programs [8,19,17], and the VCC verification tool [12] for verifying concurrent C programs have been quite successful. These tools are (when applicable) thread, function and object-modular, and scale well to large programs. For programmers interested in formally, exhaustively verifying formal specifications ranging from simple partial specifications such as the absence of null-pointer dereferences and out-of-bounds array accesses, to program invariants, assertions, procedure pre- and post-conditions these tools are indispensable. We present an overview of a prototype tool [29] for carrying out static, modular verification of assertions and invariants.

For transactional platforms, existing static verification tools cannot be used as is, since they are not aware of transactions or possible relaxed consistency semantics that may be offered by a transactional platform. The goal of the technique presented in this section is to provide a verification environment exactly like that of VCC but for programs running on transactional platforms. The verification approach provides scalability and modularity, as VCC does, but requires programmer annotations for procedure pre- and post-conditions and loops in the same way all existing modular static code verification tools do.

For performance reasons, many practical transactional platforms provide a weaker consistency guarantee than atomic, serializable transactions and non-transactional accesses. One very widely used such consistency model is snapshot isolation (SI), where the entire transaction is not guaranteed to be atomic, but all of the read accesses in the transaction are atomic and all the updates performed by the transaction are atomic. Many popular databases provide SI as the default consistency mode. Relaxed semantics and relaxed conflict detection schemes other than SI, such as programmer-defined conflict detection [39], and early release of read set entries [36] have been investigated in the database, software and hardware transactional memory communities. For distributed transactional programs, relaxed consistency semantics such as session SI [14] and parallel SI [37] have been investigated (See per-record time line consistency [11] and prefix consistency [38] for examples). In the rest of this section, for brevity, we focus on the SI relaxed consistency model.

When a transactional execution platform provides strong consistency and serializable transactions, the code of a transaction can be treated as sequential code. This significantly simplifies writing and verifying applications. For the increasingly common transactional execution platforms with relaxed semantics, one way to retrieve the simplicity of sequential reasoning is to enforce serializability via additional analyses or instrumentation, e.g. by preventing or avoiding write-skew anomalies. This approach can be useful some of the time, but, for many examples, may result in a loss of performance or availability and defeat the purpose of relaxed semantics. On platforms with relaxed semantics, much of the time, it is the application author's intent to implement a transactional program that is correct, e.g. satisfies assertions and invariants, without enforcing strong consistency or serializability. Typically, the way relaxed consistency exhibits itself in transactional code is in the form of "stale reads"' – data read by the transaction may not be the most recent version later during the transaction, or even at the time of the read access, in the case of geo-replicated databases. The verification technique presented in this section can handle such transactional programs.

We take a transactional program and the relaxed consistency semantics SI. Using the transactional program and the relaxed consistency model, we produce an augmented C program with VCC annotations. The program our approach outputs has the same structure as the input program, but includes an encoding of the relaxed transactional semantics and allows exactly the executions and interleavings specified by the relaxed semantics through the use of auxiliary variables in VCC. This program transformation can be viewed as augmenting the program with a high-level implementation of the transactional platform. The transformation is designed with special attention towards preserving the thread, function and object modularity of the verification of the sequential version of the program in VCC.

2.1 Motivating Examples

To motivate our approach, in this section, we use (Figure 1) the Labyrinth benchmark from the STAMP benchmark suite, one of the four benchmark programs we applied our method to. The Labyrinth program satisfies the desired invariants and procedure post-conditions despite its executions not being serializable. Enforcing serializability (as is typically accomplished by enforcing conflict serializability [31]) would be an unnecessary restriction that hurts performance.

Labyrinth is an example of a common parallel programming pattern. Transactions each read a large portion of the shared data, perform local computation and update only a small portion of the shared data.

```
// Program invariant:
// forall int i; 0<=i && i< pathlist->num_paths
//      ==> isValidPath(grid, pathsList->paths[i])

FindRoute(p1, p2) {
 transaction {
       localGridSnapshot = makeCopy(grid);
                        // Take snapshot of entire grid

       // Local, possibly long computation
       onePath = shortestPath(p1, p2, localGridSnapshot);

       // Desired post-conditions of shortestPath:
       assert(isValidPath(onePath, localGridSnapshot))
       assert(isConnectingPath(onePath, p1, p2);

       // Register points on onePath as "taken" on grid
       // Add onePath to pathsList
       gridAddPathIfOK(grid, pathsList, onePath);

       // FindRoute must ensure program invariants,
       // and the post-condition
       //   onePath in pathsList &&
       //        IsConnectingPath(onePath, p1, p2)
} }
```

Fig. 1. Outline for `FindRoute` code and specification

As shown in 1, each concurrent transaction runs an instance of the function `FindRoute` to route a wire in a three-dimensional grid (`grid`) from point `p1` to point `p2`. Wires are represented as paths: lists of points with integer x, y, and z coordinates, where consecutive entries in the list must be adjacent in the grid. The grid is represented as a three-dimensional array, where each entry `[i][j][k]` is the unique ID of the path (wire). A data structure `pathList` keeps pointers to all paths in an array.

Each execution of FindRoute(p1,p2) first takes a snapshot of the grid (line 1) by traversing it and then performs local computation using this local snapshot to compute a path (onePath, line 4) from p1 to p2. Observe that, during this local computation, other executions of FindRoute may complete and modify the grid. In other words, localGridSnapshot may be stale snapshot of grid. SI guarantees in this example that (i) the read of the entire grid in line 4 is atomic, (ii) that the updates to pathsList and grid in line 11 are atomic, but does *not* guarantee that the entire transaction is atomic.

Specification. Desired properties for this program are that (i) the grid is filled correctly by the information, and that (ii) no two paths overlap. The latter of these is implicitly ensured because each grid point contains a single wire ID number. The former is formally expressed below

```
isValidPath(int ***grid, path_t* p) =
  (forall int i; 0<= i < path->path_len ==>
     p->ID == grid[p->x[i]][p->y[i]][p->z[i]])
   forall int i; 0<= i < path->path_len-1 ==>
     isAdjacent(p->x[i],   p->y[i],   p->z[i],
                p->x[i+1], p->y[i+1], p->z[i+1])
```

FindRoute must preserve this invariant for all paths on pathList in addition to the post-conditions that onePath is a valid path that connects p1 to p2 and is in pathList.

Static Verification of Sequential FindRoute: When FindRoute is viewed as if it is running sequentially, with no interference from other transactions, it is straightforward to verify using VCC. The following are the key steps taken:

– We verify that the code for shortestPath (not shown) satisfies the post-conditions in lines 6 and 7.
– Using this fact, we verify that gridAddPathIfOK, if and when it terminates, satisfies the program invariant (no two paths overlap and pathsList and grid are consistent), and the desired post-conditions in 14.

To carry out the verification tasks above, static code verification tools, including VCC, require the programmer to write loop invariants as annotations. The rest of the verification of function post-conditions is carried out automatically.

Verifying FindRoute Under Relaxed Consistency: The verification of FindRoute under SI rests on the key observation that the conditions listed above for correctness of FindRoute under SI remain correct even when thread interference as described by SI occurs. The technique described in this section allows us to verify that this is the case mechanically using VCC.

In a given instance of FindRoute, if gridAddPathIfOK detects that onePath overlaps an existing wire, it explicitly aborts the transaction. Instances of FindRoute that complete do so because they have computed a path onePath that not only does not overlap any of the wires in the initial snapshot localGridSnapshot, but also does not overlap any of the paths added to the grid since.

The intuition behind FindRoute being correct while running under SI is as follows:

1. SI ensures that the traversal and copying of the grid in line 1 is carried out atomically.
2. SI ensures that the updates to `pathList` and `grid` performed by `gridAddPathIfOK` are carried out atomically.
3. To verify that an atomic, terminating execution of `gridAddPathIfOK` establishes the desired program invariant and post-condition, it is sufficient to know that the post-conditions established by `shortestPath` in lines 6 and 7 still hold at the time `gridAddPathIfOK` starts running. New paths that may have been added to `grid` since `grid` was copied into `localGridSnapshot` do not cause invariant violations, since the atomically-executed `gridAddPathIfOK` explicitly aborts the transaction if it detects that `shortestPaths` overlaps one of the paths in `grid`.

In our technique, we transform and augment the code for `FindRoute` to obtain another C program with VCC annotations. Verifying the resulting program in VCC amounts to checking that (3) continues to hold under thread interleavings constrained by (1) and (2).

Our technique accomplishes this as follows.

- The encoded program has exactly the set of thread interleavings allowed by SI. The auxiliary variables (e.g., version numbers for each grid element and wire, fictitious locks, etc.) and constraints ("assume" statements) on these variables built into the encoded program only allow executions where all read accesses in a transaction are carried out atomically and all write accesses are carried out atomically. There are no other restrictions on how the threads are interleaved.
- When VCC verifies the object and global invariants and procedure post-conditions (e.g., the `FindRoute` program invariant or post-condition of `shortestPath`) in the encoded concurrent program, it checks whether they are preserved under thread interference possible in the encoded program. Since the encoded program (an ordinary concurrent C program) allows exactly the interleavings specified by SI, this amounts to verifying that properties of the original program running under SI hold.

The encoded program preserves the structure of the original program, and does not inline code from other possibly interfering transactions.

2.2 Preliminiaries: Transactional Programs

The user provides the code for a transaction as a C function. The beginning and end of a transaction are indicated by calls to the `beginTrans()` and `endTrans()` functions. We make the committing of a transaction syntactically visible by a call to `commitTrans(`t, inv`)` in order to allow the programmer to specify an invariant that holds when the transaction is committed. Data shared by transactions is represented by aliasing among arguments of functions calls representing different transactions. Unless indicated otherwise, function arguments of the same type are treated as possibly aliasing to the same address. Shared data is represented by aliasing among arguments of functions calls representing different transactions. Transaction are not allowed to be nested.

We define states and the transition relation of a program under SI as follows: A *global state* is a tuple $GS = (GlVar, GlMem, TtoLcSts)$ such that

- $GlVar$ is the set of global variables, i.e., shared objects (structs) that multiple transactions hold references to in GS,
- $GlMem : GlVar \to Val$ maps global variables to their values in the memory, and
- $TtoLcSts : Tid \to L$ keeps local states of each transaction.

The local state of a transaction t contains $LcVar$, the set of objects local to t, $RSet \subseteq GlVar$ ($WSet \subseteq GlVar$) the set of global variables that have been read (written) by t since the beginning of the transaction.

An *action* is a unique execution of a statement by a transaction t in a state s. An *execution prefix* of a program P_{SI} is a tuple $E_N = (\boldsymbol{s}, \boldsymbol{\alpha})$ where $\boldsymbol{\alpha}$ is a finite sequence of actions $\alpha_0, \alpha_1, \ldots, \alpha_{N-1}$ and $\boldsymbol{s} = s_0, s_1, \ldots, s_N$ is a finite sequence of states such that $(s_i, \alpha_i) \to s_{i+1}$ for all $i < N$. An execution has the form:

$$s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_{N-1}} s_N$$

The transaction consistency semantics and conflict detection scheme, such as serial execution of transactions, conflict serializability, and SI specify which interleavings of actions from different transactions are allowed in an execution.

2.3 SI and Other Relaxed Conflict Detection

We write $Idx_E(\alpha_i)$ to refer to the index i of action α_i in the execution, and $Tr_E(\alpha_i)$ to refer to the transaction performing α_i. To make precise the sets of executions of a program allowed by different relaxed conflict-detection schemes, we define the *protected span* of a shared variable x within a transaction t for a given consistency model M. Intuitively, this span is a set of indices of actions with the property that, according to the consistency model, at none of these indices can an update to x in shared memory take place due to the commit action of a transactions other than t.

The following definition of snapshot isolation makes two simplifying assumptions. First, we assume that if a transaction both reads and writes to a variable, then the read comes before the write. Second, we assume that the effects of transactions that have not committed or do not commit are not visible to other transactions. We also take as implicit the usual requirement that of two concurrent transactions with write-write conflicts, at least one must abort.

Definition 1. *An execution E is said to obey snapshot isolation iff for all committed transactions t, (i) all read accesses performed by t are atomic, (ii) all write accesses performed by t are atomic, and (iii) if t both reads and writes to a variable x, the value of x in shared memory is not changed between the first access to x by t and the commit action of t.*

To specify snapshot isolation in terms of spans within an execution, we first define the snapshot read span of a variable x read by a transaction t. Let α_i be the first read action (of any variable) in a transaction t, and let α_j be the last read of a variable x by t. Then, the *snapshot read span* of x in t is the interval $[i, j]$. If x is never read in t, its snapshot read span is the empty interval. The protected span of a variable x in snapshot isolation is defined as follows:

- If x is only read by the transaction, the protected span of x is the snapshot read span of x.
- If x is both read and written to, then the protected span is the interval $[i, j]$ where i is the index of the first access of the transaction to x, and j is the index of the commit action of t.
- If x is only written to, the protected span is defined to be the write span of x, which is the interval $[i, j]$, where i is the index of the first write access to x by t, and j is the index of the commit action of t.
- Otherwise the protected span is empty.

Snapshot isolation requires that the protected span of each variable x does not contain any commit actions by other threads that write to x. Due to space restrictions, we omit a proof of the fact that this formulation of SI in terms of protected spans, which describes how certain implementations of SI operate, implies Definition 1.

Other related relaxed transactional semantics, such as !WAR can be defined using the concepts of read and write spans, version numbers, fictitious locks and assume statements in a similar way.

Relaxing Write-After-Read Conflict Detection. This semantics specifies the executions provided by a transactional memory with relaxed detection of conflicts using the `!WAR` annotation as described in [39]. In this semantics, the programmer annotates certain read actions to be *relaxed reads.* The protected span of a variable x in t is defined as the interval $[i, \mathit{Idx}(\mathit{commit}(t))]$, where α_i is the first regular (not relaxed) read action or write action accessing x as part of t. A relaxed read of x in t is simply required to return the result of the last write to x. Differently from serializable semantics, in read-relaxed semantics, after a relaxed read of x by t but before t commits other transactions are allowed to commit and update the value of x. However, conflicting writes are never allowed between a write access and the corresponding commit action.

2.4 Concurrency, VCC and Modular Verification

In this section, we informally, introduce the VCC mechanisms and conventions we make use of in our approach. VCC allows programmers to think C structs as objects and other base C types (int, char, double etc.,) as primitive types. VCC allows programmer to create `ghost` objects or declare `ghost` structs which can not modify the concrete program state but can be used for verification tasks. `ghost` structs can be C structs defined in the program or special types provided by VCC.

Each object has a unique owner at any given time. The concept of ownership is one mechanism using which access to objects shared between threads is co-ordinated, and invariants spanning multiple objects are stated and maintained. Objects can be annotated with any number of two-state transition invariants: first-order formulas in terms of any variables.

VCC allows the introduction of ghost variables of all types, including all C types, and more complex ones such as sets or maps. Ghost variables are (auxiliary) history variables, and they do not affect the execution of the program and values of program variables.

VCC performs modular verification in the following manner. Each function is annotated with pre- and post-conditions. Each loop is annotated with a loop invariant. Every struct may be annotated with two-state transition invariants. Code may also be annotated with assertions in VCC's first-order specification logic, in terms of the program and ghost variables in scope. VCC then verifies the code for one function at a time, using pre-post condition pairs to model function calls, loop invariants to model executions of loops, and "sequential" or "atomic" access, as described below, to model interference from concurrent threads. In "sequential" access, the thread accessing a variable obtains exclusive access to a variable `aVar` by obtaining ownership of `aVar`. Another way to coordinate access to shared variables in VCC is to mark them `volatile` and to require that any state transition of the program must adhere to the transition invariants of these objects.

2.5 Source-to-source Transformation for Simulating SI

In this section, we present our source-to-source transformation. We have chosen to implement our verification approach in this manner in order to expose to the users the constructs used in the encoding. Currently, this transformation is carried out manually following the procedure described in this section. In future work, we plan to provide tool support for this transformation.

The input to our transformation is C program P_{SI}. P_{SI} contains the program text and the correctness specifications. In VCC, these specifications are provided as

- an invariant for user-defined data types (structs),
- desired function pre- and post-conditions, given as boolean expressions in terms of variables in scope at function entry and exit,
- assertions, given as boolean expressions over transaction-local or shared variables

A global invariant that is to hold at the time a transaction commits can also be specified.

The output of the transformation is a program $\widetilde{P_{SI}} = \mathit{Encode}(P_{SI})$ that will be verified using VCC. It runs under ordinary C semantics and contains the kinds of VCC annotations described in Section 2.4. Verifying $\widetilde{P_{SI}}$ under ordinary VCC semantics is equivalent to verifying P_{SI} under transactional SI semantics.

The encoding is obtained via a high-level modelling of the operational semantics of SI. Since only the effects of succeding transactions are visible to other transactions, the high-level model does not include mechanisms such as rolling transactions back or aborted transactions. The transformation is described for SI. While a simpler transformation would have sufficed for SI, the construction we present here is necessary to generalize to other relaxed consistency models, such as early release of read entries, programmer-defined conflict detection, e.g. ignoring write-after-read conflicts.

$\widetilde{P_{SI}}$, the encoded version of a program P_{SI} is constructed as follows.

$\widetilde{P_{SI}}$ makes use of VCC statements of the form `assume(`ϕ`)`. A thread in a program can take a state transition by executing `assume(`ϕ`)` only at a state s that satisfies ϕ, in which case, program control moves on to the next statement. Interleavings disallowed by the consistency model M are expressed as a formula ψ in terms of objects' version numbers, and statements of the form `assume` $\neg\psi$ are used in the encoding.

Transforming Data Types: Each primitive C type used in the original program is replaced by a "wrapper" struct type. This is necessary so we can coordinate access to these variables using mechanisms provided by VCC.

For simplicity, we present the transformation for programs that only use `int` s as primitive types. In the transformation, each shared variable of type `int` is replaced with a variable of type `PInt` as shown below:

```
PInt{
   int inMem;          int inMemVNo;
   int inTM[Trans];    int inTMVNo[Trans];
   Lock lock;
   _(invariant \unchanged(inMemVNo) ==> \unchanged(inMem))
   _(invariant \forall int t;
           \unchanged(inTMVNo[t]) ==> \unchanged(inTM[t]))
};
```

In the definition above `PInt` stands for `struct Int*`. The "wrapper" type `PInt` holds the following information:

- a field `inMem` value that corresponds to the value of the variable in shared memory,
- a version number `inMemVNo` that gets incremented atomically each time the `inMem` field is written to,
- a (ghost) field `inTM[Trans]` which is a map from Tid to integers. `inTM[t]` holds the value of the transaction-local copy of the integer
- a (ghost) field `inTMVNo[Trans]` which is a map from Tid to integers. `inTMVNo[t]` is incremented atomically with each update of `inTM[t]`
- a (ghost) field `lock` that is used to convey to VCC when a transaction has exclusive access to the `int` variable

This wrapper type has an important invariant that indicates that a field's value remains unchanged if its version number remains unchanged. This invariant, along with `assume` statements involving version numbers allows us to represent constraints such as the value of a variable remaining unchanged between two accesses within a transaction.

For each global variable of type `int` in P_{SI}, the encoded program $\widetilde{P_{SI}}$ has a global variable of type `PInt`. For each global `int` variable (a) in P_{SI}, we denote the corresponding `PInt` variable in $\widetilde{P_{SI}}$ by $\tilde{a}$. When transforming the program syntactically, we use lowercase variables `a` to refer to variables of type `int` in the original program, and uppercase versions (`A`) to refer to the corresponding wrapper variable of type `PInt` in the encoded program.

To implement transactional semantics, we create an instance of the `Trans` struct per transaction.

```
Trans{
   bool holding[PInt];
   bool readSetInt[PInt];
   bool writeSetInt[PInt];
};
```

Fields of `Trans` are ghost maps. `readSetInt` and `writeSetInt` are maps that store `Int` objects read and written to by this transaction.

If there are struct declarations in the original program, `Trans` contains three maps for each field of these structs used following the same approach for `Int` s. The structs and their fields are flattened into maps.

Transforming a Transaction. The transformation is described assuming that the code has been decomposed so that each statement accesses a global variable at most once, as is typical in transactional applications. The code transformation makes use of a number of C functions whose pre- and post-conditions are presented later in this section. We only provide highlights of the transformation rules:

- Statements of the form `beginTrans(`t`)` remain unchanged in the transformed version. (see pre and post-conditions of this function below)
- Statements that only assign a value *val* to a local variable or a local variable to a local variable remain unchanged in the transformation.
- Statements that create a new shared variable `A` of type `Int` are transformed to `newPInt(A)`. This is similar for creating new shared variable of other types.
- Each statement `l = v` by transaction `t` that reads a global variable `v` into local variable l is transformed to an atomically-executed statement that performs the equivalent of the following VCC code atomically.

  ```
  assume( \forall PInt P;
            trans->readSet[P] ==>
                 trans->inTMVNo[P] == P->inMemVNo);
  l = transReadInt(trans, V);
  ```

  The specifics of `transReadInt` are described later in this section.
- Each statement `V = l` that writes the value of a local variable `l` to a shared variable `V` is transformed to atomically-executed statements that perform the equivalent of the following VCC code.

```
assume(V->\owner == t || V->\owner == NULL);
acquireLock(V,t);
assume(V->inTMVNo[t] == V->inMemVNo);
//V has not been written to since it was read by t.
transWrite(V, l, t);
```

This code enforces (as per SI semantics) if `V` is in the transaction's read set and write set, then `V` have not changed since a snapshot was taken.

- Each statement `commitTrans(t, inv)`, is transformed to the following atomically-executed sequence of statements:

```
assume( \forall PInt P;
        t->writeSetInt[P] ==>
           P->inTMVNo == P->inMemVNo + 1);
commitTrans(t);
assert(inv);
```

- For each statement `endTrans(t)`, in the encoded version, we replace the statement with `endAndCleanTrans(t)`.
- Each statement `assert(p)`, where `p` is a boolean expression in terms of local variables, is left as is in the encoded version. Each boolean expression `e` involved in a loop invariant, and function pre- and post-condition is transformed to a boolean expression `E`, where each appearance of a global variable `v` is replaced with a reference to the transaction-local copy `v->inTM[t]`.

The functions used in the encoded program are listed below together with their pre-conditions and post-conditions:

- `beginTrans(t)` creates a `Trans` structure for thread `t`. This function has no pre-condition and has the post-condition that the read and write sets of `t` and the set of variables `t` has exclusive ownership of are empty, i.e.,

```
\forall PInt P; !t->readSetInt[P] &&
           !t->writeSetInt[P] && !t->holding[P]
```

- `acquireLock(V, t)` is used to obtain exclusive access to `V` by transaction `t`. This is accomplished by using the fictitious (ghost) lock `V->lock`. Since we are verifying only succeeding executions of transactions (and assuming that aborted transactions have no visible effect), we call `acquireLock` in the encoded program only at a state where it will successfully complete. Thus, this function has the pre-condition that the global variable `V` has no owner or is owned by `t`, and the post-condition that the owner of `V` is the transaction `t`. The post-condition of `acquireLock(V,t)` also requires that `t->holding[V]` be true.
- `transRelaxedRead(V,t)` reads `V` in a transaction `t`. This function does not require `V` to be owned by `t` and has the post-condition that

```
t->readSetInt[V] == true &&
V->inTM[t] == V->inMEM &&
V->inTMVMo[t] == V->inMEMVNo
```

- `newPInt(V)` is used to create a new PInt variable. This function has the post-condition that `V->owner` is `t`. All version numbers associated with `V` are initialized to 0.
- `transWrite(V, l,t)` writes the value of the local variable `l` to the `inMem` field of `V` and atomically increments `v->inTMVNo[t]`. If `V` has been read previously by `t`, then this function requires that `V`'s version number has not changed since. These are expressed by the pre-condition

  ```
  V->\owner == t && V->inMemVNo == V->inTMVNo[t]
  ```

  and the post-condition

  ```
  t->writeSetInt[V] == true &&
  t->inTM[t] == l &&
  t->inTMVNo[t] == old(t->inTMVNo[t]) + 1
  ```

- `commitTrans(t)` commits a transaction by writing the updates performed by the transaction into the memory. Note that a valid execution can have only local statements (that only effect local state) after `commitTrans(`t`)` statement until it ends the transaction. This function is better explained by the following pseudocode

  ```
  _(atomic t {
        \foreach PInt P;
            if (ptrans->writeSetInt[P]) {
                 P->inMEM = P->inTM[t];
                 P->verNoInMEM = P->verNoInTM[t];
            }
  })
  ```

  Since VCC currently does not support loops inside `atomic` statements, the state update corresponding to the loop above is expressed as the function post-condition for `commitTrans` and the atomicity of the commit is accomplished using fictitious locks for objects for which `holding` is true.
- `endAndCleanTrans(t)` ends a transaction `t` by releasing the locks that the transaction holds, cleaning its read and write sets. It has the post-condition that `t` releases ownership of all objects it owns, and the `readSetInt`, `writesSetInt`, and `holding` are all reset to maps corresponding to empty sets.

The following theorem states the soundness of our verification approach.

Theorem 1 (Soundness). *Let P_{SI} be a transactional program and $\widetilde{P_{SI}}$ be the augmented program obtained from P_{SI} as described above. Then $\widetilde{P_{SI}}$ satisfies its specifications (assertions, invariants, function pre- and post-conditions) if and only if P_{SI} satisfies its specifications.*

It follows from this theorem that users can start with the program P, provide the desired specifications, and additional proof annotations. Then, to verify properties of P_{SI}, users can follow the source-to-source transformation approach described in this section and obtain $\widetilde{P_{SI}}$. Verifying the transformed specifications with the transformed annotations on $\widetilde{P_{SI}}$ is equivalent to verifying the specifications of P_{SI}, by the soundness theorem.

The source-to-source code transformation preserves the thread, function, and object structure of the original program. The newly-introduced objects representing transactions are local to each thread or transaction. All additional invariants introduced are per-object. There is no inlining of code from other, possibly interfering transactions, and the size of the transformed code is linear in the size of the original code.

2.6 Verifying Transformed Program with VCC

In this part, we explain how verification of the transformed program is performed on the grid example. For the grid, user provides the program invariant both as the pre-condition and post-condition of `findRoute` and specifications between lines 13-16 as post-condition for the original program.

Generally, program pre- and post-conditions are not enough for verification and the user may need extra ghost variables or annotations. Especially for the loops or other code blocks enclosed with curly parentheses, user should provide conditions about user defined shared or local objects that are satisfied throughout the code block and helps verification of the post-conditions. Since `findRoute` does not contain such code blocks. Hence, no extra annotation is needed.

Moreover, the user may need to provide extra annotations although the function does not contain any such code blocks. These annotations reflect the correctness intuition of the program. To our experience with SI, user should provide a condition that holds right after end of the read phase (after snapshot has been taken) such that this condition is preserved although other transactions interfere and modify data. In the grid example, assertions on lines 6,7 reflect the correctness intuition. `onePath` is a valid and connecting path for `localGrid` and `grid` when the snapshot was taken. It continues to hold during execution although other transactions interfere and modify `grid`. This information is enough for VCC to verify post-conditions of `findRoute`: Since `onePath` is a valid and connecting path on the `localGrid` and points on the `onePath` stays the same in `grid`, `onePath` becomes a valid and connecting path after call to `addGridPathIfOK`.

Note that the assertions added for verification on lines 6,7 do not include variables, fields or calls to functions introduced by the transformation. Therefore, user does not need any knowledge about transformation and these extra program parts. This is the case we encountered during the verification of examples. Correctness intuition based on local and shared user variables are enough for verification.

If the initial correctness intuition is not enough for verification for function post-conditions, user may come up with tighter and stricter annotations for verification of assertions or program post-conditions until the function is verified.

2.7 Experimental Demonstration

We applied our technique to the Genome, Labyrinth and Self-Organizing Map benchmarks as implemented in [39] and a StringBuffer pool example that we wrote ourselves. These examples have pre-annotated transactional code blocks which can be run under relaxed transactional semantics. We made precise and formally verified the correctness arguments for these implementations and for the `StringBuffer` example. Our work makes formal the correctness arguments in the work of Titos et al. [39] about the correctness of the transactions in the benchmarks and provides evidence that the intuitive reasoning about why programs can function correctly under TM relaxations can be expressed and verified systematically.

For each benchmark, we wrote partial specifications and statically verified that they hold for transactional code running with the regarding relaxed consistency semantics, starting from a VCC verification of the specifications on a sequential interpretation of the benchmark.

```
struct node_t { int key; node_t* next; ghost Set reach;}

bool list_insert(list_t *listPtr,
                 node_t *node) {
  node_t *prev, *curr = listPtr->head;

   do {
     prev = curr;
     curr = curr->next;
  } while (curr != NULL
          && key > curr->key);
  _(invariant loopInv(prev, curr, head, node))
  // loopInv(prev, curr, head, node) ==
  //          prevKey < key && prevKey < curKey
  //          && prev reachable from head
  //          && curr reachable from head

// assert(prev->next == curr);
  node->next = curr;
  prev->next = node;
  return true; // key was not present
}
```

Fig. 2. The insertion operation of a sorted linked list

- **Genome:** Figure 2 shows the pseudocode for a linked list implementation used in the `Genome` benchmark [10]. The code in the figure has been simplified for ease of presentation. In the part of this benchmark where relaxed consistency is used, concurrent transactions insert into a shared linked list. Transactions run under programmer-defined conflict detection, where write-after-read conflicts are ignored (`!WAR`), i.e., do not cause transactions to abort.

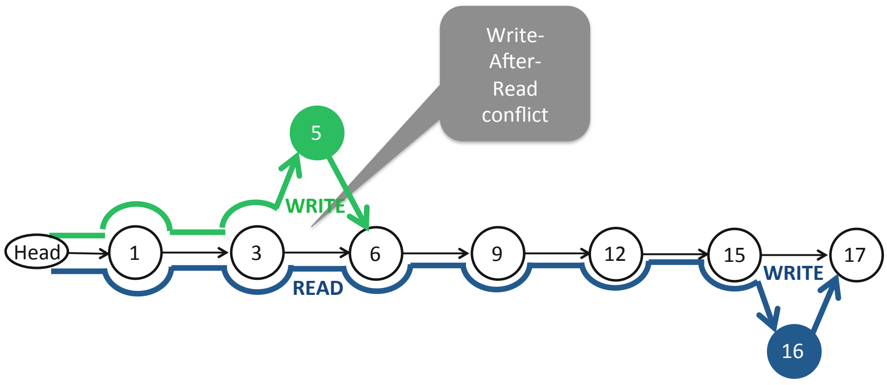

Fig. 3. Sorted linked list and a write-after-read conflict

Figure 3 illustrates how concurrent insertions experience write-after-read (`!WAR`) conflicts, and how, intuitively, it would be correct implementation to let an insertion commit even though it experiences a `WAR` conflict. Following [39], the body of `list_insert` is marked with the `!WAR` annotation to indicate that write-after-read conflicts should be ignored.
We verify that the linked list maintains two invariants under interference : (i) its nodes are in ascending order and (ii) linked list is not circular. We further verify that the `addNode(newNode)` Function satisfies the post-condition that the node it adds (`newNode`) is reachable from the head of the linked list. The read (traversal) phase of the `addNode` function finds a node `prev` in the list after which `newNode` is to be inserted. The assertion that `prev` is reachable from the head of the list and that the appropriate place for `newNode` to be inserted is right after `prev` is preserved despite interference caused by ignoring write-after-read conflicts.

- **SOM:** In this benchmark, concurrent transactions run the learning phase of the machine learning algorithm SOM. SOM contains a shared grid of which nodes are n-dimensional vectors. The learning function `solve` takes an n-dimensional vector v and the grid as input, calculates the Euclidean distance of v to each grid nodes, picks the closest one v' and moves nodes in a neighbourhood of v' closer to v.
- **StringBuffer** In this example, a pool of `StringBuffer` objects are implemented as a collection. Transactions to allocate or free a string buffer perform relaxed read on the shared collection. When a transaction finds a suitable object and wants to allocate it, it can commit ignoring other possible write operations (that allocate or free a string buffer object) on the collection. The example is written using programmer-defined conflict detection, in particular, using `!WAR` semantics. We verified that a data structure invariant and post-conditions of the `Allocate` and `Free` functions are satisfied.
- **Labyrinth:** This example and its verification process was described earlier in the section.

We have demonstrated the applicability of our verification approach on these examples that were written without assuming serializability and satisfied their specifications despite this. In each of these examples, our encoding facilitates thread- and procedure-modular correctness proofs that hold for an arbitrary number of threads. Programmer annotations on encoded program makes no reference to auxiliary encoding variables. Our experience with the SI and `!WAR` relaxed consistency models, which are very similar to other relaxed consistency models described earlier leads us to believe that our static verification technique is a useful tool for a programmer building applications in these settings.

2.8 Related Work

Relaxed Conflict Detection. Relaxed conflict detection has been devised to improve concurrent performance by reducing the number of aborted transactions. Titos et al. [39] introduce and investigate conflict-defined blocks and language construct to realize custom conflict definition. Our work builds on this work, and provides a formal reasoning and verification method for such programs. As we have shown with SI and `!WAR`, we believe that our method can easily be adapted to support other relaxed conflict detection schemes.

Enforcing (conflict) Serializability, Detecting Write-Skew Anomalies. There is a large body of research on verifying or ensuring conflict or view serializability of transactions even while the transactional platform is carrying out relaxed conflict detection [15,7,5,9,3,18]. In this work, we enable programmers to verify properties of transactional code on SI even when executions may not be serializable. This allows the user to prove the correctness of and use transactional code that allows more concurrency.

Linearizability. One way to allow low-level conflicts while preserving application-level guarantees is to use linearizability as the correctness criterion [26]. To prove linearizability of a transactional program P running under SI, one could use the encoded program we construct, $\widetilde{P}$ as the starting point in a linearizability or other abstraction/refinement proof. In this work, we have chosen not to do so for two reasons. First, abstract specifications with respect to which an entire program is linearizable may not exist or may be hard to write. Second, programmers would like to verify partial specifications such as assertions into their program in terms of the concrete program variables in scope. Verifying linearizability does not help the programmer with this task.

Encodings, Source-to-Source Transformations. As a mechanism for transforming a problem into one for which there exist efficient verification tools, source-to-source code transformations are widely-used in the programming languages and software verification communities. The work along these lines that is closest to ours in spirit involves verifying properties of programs running under weak memory models by transforming them into programs that run under sequential consistency semantics [6,4]. Our work also makes use of a source-to-source translation in order to transform the problem of verifying a transactional program running under SI to a generic C program that can be verified using VCC. Our transformation results in only a linear increase in code size. While we perform an encoding for representing different semantics from these studies, our encoding itself has some features that distinguish it from encodings devised for different verification purposes. During the transformation, the thread, object and procedure structure of the original program is preserved. No inlining of extra code modeling interference from other transactions is involved. We also have the practically important advantage that while verifying his code under SI, the user does not have to provide extra annotations in terms of the extra auxiliary variables in the encoded program.

3 Dynamic Verification for Transactional Programs

While there has been some preliminary work on temporal specifications for programs that use optimistic concurrency, including transactional programs [35,34], the vast majority of research on dynamic verification techniques for transactional programs has focused on detecting races. As with ordinary concurrent programs, race conditions are undesirable for two reasons. First, they result in non-deterministic outcomes for read accesses even for a given fixed execution and thread interleaving. Second, race conditions are symptomatic of higher-level programming errors, such as a certain concurrency discipline not being followed or the intended atomicity not being accomplished by the program.

As with most concurrent programming settings, there is much discussion on the definition of race conditions for transactional programs. In this section, we first provide an overview of various definitions of race conditions that are the most interesting from the point of view of an application programmer. We then highight two race detection tools for transactional programs from the literature: the Goldilocks tool for precisely detecting races in transactional Java programs, and the T-Rex tool for dynamic detection of potentially-harmful pairs of transactional and non-transactional conflicting accesses to a shared variable.

One category of race detection approaches are based on a precise definition of a happens-before relationship between actions in a transactional program. In such approaches, synchronization primitives in the transactional program are modeled in the same way as the programming language being used so that the definition of the happens-before relation is backwards compatible. There are several different ways of defining the happens-before relationship between software transactions in the literature [23,20,22,25,24]. Different definitions of happens before make different choices on whether two transactions are considered to synchronize with each other, usually based on what variables are accessed within the transactions.

Some definitions of the happens-before relationship in the literature are obtained by using an analogy to lock-protected programs. For instance, in the *Single Global Lock Atomicity* (SGLA) semantics [30], the happens-before relationship is defined as if all transactionally-executed code blocks are protected by the same, single global lock. This definition naturally integrates the happens-before relationship of the underlying programming language and the happens-before relationsips induced by the TM platform. On transactional platforms that implement this semantics, Dalesandro et al. [13] call a program *transactional data race free* (TDRF) if any two accesses to the same variable one of which is a write are ordered by the happens before relationship as defined in SGLA.

A race detection algorithm can either be explicitly based on a choice of a particular definition of a happens-before relationship in a transactional execution, or can formulate undesirable intereference between accesses without making explicit the underlying happens-before model. The former approach has the advantage of precision, even when the race detection algorithm makes the choice to allow false positives or negatives, since what exactly constitutes a false warning is known. The latter approach may be more applicable for use in conjunction with a wider variety of TM implementations. Goldilocks [16] is an example of the former category of approaches while T-Rex [28] is an example of the latter category.

In the transaction semantics used in Goldilocks, pairs of shared variable accesses where both accesses take place within the same transaction are considered to be race-free. A linearization of the projection of the happens-before ordering onto the commit actions corresponds to the atomic order of transactions as defined in [20]. Naturally, pairs of accesses executed transactionally are considered race free, reflecting the fact that these accesses are managed by the TM implementation and the programmer and the dynamic race checker should not be concerned with them.

3.1 Transaction-Aware, Precise Race Detection

The Goldilocks runtime is precise about when it throws `DataRaceException` in the presence of software transactions that manage a portion of the shared data. This issue brings about two challenges. First, for portions of the execution not contained in transactions, transactions become yet another synchronization primitive to be taken into account. Second, it is desirable to trust the correctness of a TM implementation and to avoid the cost of checking at runtime that it performs proper synchronization for its implementation variables and for accesses performed transactionally.

Goldilocks builds upon the Java Memory Model, which does not specify which happens-before edges (must) arise due to transactions and atomic code blocks. As observed in [21], there is not yet a consensus about the interaction between the semantics of transactions and the Java Memory Model. Such a specification serves as an interface between the implementers and users of TM. The TM implementer must guarantee, among other things, at least the existence of the required synchronization edges using Java language or lower-level primitives. The actual implementation (e.g. [27]) may actually provide more synchronization than required.

Goldilocks is based on the following interpretation of strongly-atomic transactions. All transactions, along with other synchronization operations are part of the global *synchronization order* of the Java memory model. Given two transactions `Tx1` and `Tx2`, Goldilocks requires that `Tx1` happens-before `Tx2` if and only if there exists a shared variable x that `Tx1` writes and `Tx2` reads. This does not include variables involved in the TM implementation but not visible to the programmer. The actual transaction manager may be performing stronger synchronization and may be using some of the same locks and/or volatile variables as the application program, but, proper synchronization of the application program should not rely on this. The Goldilocks approach is able to accomodate other, similar definitions of when a transaction happens-before another. For instance, it can handle the case where `Tx1` happens before `Tx2` if and only if there is a variable x that `Tx1` reads or writes and `Tx2` reads or writes.

To detect races at runtime, Goldilocks requires a transaction manager to provide or make possible for the runtime to collect the following information for each transaction:

- the shared variables read by the transaction
- the shared variables written by the transaction
- the place of commit point of the transaction in the global synchronization order

Goldilocks race-aware runtime makes use of the implementation of transactional (atomic) blocks via source-to-source translation of Hindman et. al. [27]. In this implementation, all shared variable reads and writes that are part of a transaction come after the first lock acquire associated with the transaction, and they come before the first lock release, which also constitutes the commit point of the transaction. Goldilocks extends the lockset update rules for the base precise race detection algorithm for Java in order to handle transactions. The nature of the lockset rules and the implementation made it possible to integrate this feature without significant restructuring. In [16], we demonstrate this way of handling transactions in our runtime on a hand-coded transactional data structure.

3.2 Detecting Potential Races in Transactional Programs

Dynamic race detection algorithms that precisely check a happens-before relationship handle many styles of synchronization. This often results in high computational overhead for both transactional and non-transactional programs. A happens-before-based algorithm needs to keep track of and synchronize access to a large amount of analysis metadata in order to precisely capture the execution and synchronization history. One other possible undesirable feature of such algorithms is that actual execution order of memory accesses and source program order may not co-incide because hardware and compiler instruction re-ordering may break this correspondence. Compiler optimizations such as the elimination of certain unnecessary accesses or, say, empty transactions may have significant consequences regarding the happens-before relationship. While the source program appears to have certain happens-before relationships, the actual execution might not. Since a programmer would ideally like to base correctness reasoning about the program on the program source, a discrepancy between source code and execution happens-before relationships can be dangerous.

While a runtime such as Goldilocks must precisely keep track of the happens-before relationships observed in an execution, for a debugging tool, this precision, even ignoring the computational cost, may not be desirable. A programmer might want to know about potential race conditions in executions that are similar to the one observed, even if the observed execution itself does not experience a race condition. Race detectors such as ones based on Eraser [32] for non-transactional programs, or the T-Rex [28] race detection tool for transactional programs instead track adherence to a certain concurrency discipline.

For transactional programs, race detection approaches based on a happens-before relationship necessarily make strong assumptions about TM implementations. For instance, since privatization and publication, based on program source code and a happens-before relationship, are expected to be safe patterns. However, in practice, especially for software TM implementations, for performance reasons, privatization and publication may not be safely supported [28]. For software TM implementations that do not support safe privatization and publication, speculative reads, buffered writes or the abort mechanism in the TM implementation may result in data races introduced by the TM implementation itself [28]. Kestor et al. [28] provide examples of programs in which updates can be lost or zombie transactions may have harmful effects on other transactions – effects that are not visible when only the source code of the transactional program and a precise happens-before relationship based on read and write-sets of succesful transactions are considered. As a result, race detection based on the happens-before relationship may not be the most appropriate practical approach for debugging transactional programs. Such programs may experience data corruption or even program crashes even if they appear to be free of races judging by a happens-before relationship and the program source code. Researchers have investigated concurrent programming disciplines to avoid undesirable interactions between non-transactional and transactional accesses [1,2]. These disciplines may be enforced by a compiler, a runtime, or by requiring the programmer's collaboration by notifying the compiler or runtime of transitions of variables between transactional and non-transactional access modes.

Instead, in T-Rex [28], Kestor et al. employ a correctness criterion that is less dependent on particular TM implementations, and, in particular, does not rely for correctness on a safe implementation of privatization and publication idioms. T-Rex defines a transactional data race so that in a correctly synchronized program, for a pair of accesses to the same shared variable at least one of which is a write, the following hold.

- Either both accesses take place within transactions, or
- the two accesses are separated by a global (i.e., involving all threads) synchronization operation such as a barrier or thread fork or join, and
- it is not the case that one of the accesses takes place in a fragment of the execution during which there is only one live thread

This definition of transactional data race is defined based on the set of accesses observed during an execution, and, in many ways, is independent of the particular interleaving of actions. Similar to data race detectors based on lockset algorithms, this definition does not need to witness a concurrent access to a shared memory location in particular program execution to report a potential race [33]. Consider a program that fails to protect a particular data access by enclosing it within a transaction. Race detection tools that track the precise happens-before relationship would signal an error only in some executions of this program, whereas our algorithm would signal a transactional data race in all executions of this program.

Using this pragmatic definition of correct synchronization, T-Rex is able to both be computationally efficient and detect noteworthy violations of the correctness criterion above in STAMP benchmarks.

4 Conclusion

The tools and techniques described in this chapter represent important but preliminary steps towards building software engineering and verification tools for programs that make use of transactional memory. Mature software engineering tools for transactional programs will need to be developed and integrated into the program authoring, compilation, testing and debugging toolchain as TM finds wider use. As language primitives implemented using TM become widespread, it is expected that semantics of TM-supported programming primitives will find their way into language and memory-model specifications.

References

1. Abadi, M., Birrell, A., Harris, T., Isard, M.: Semantics of transactional memory and automatic mutual exclusion. SIGPLAN Not. 43(1), 63–74 (2008)
2. Abadi, M., Harris, T., Moore, K.F.: A model of dynamic separation for transactional memory. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 6–20. Springer, Heidelberg (2008)
3. Adya, A.: Weak consistency: a generalized theory and optimistic implementations for distributed transactions. PhD thesis, AAI0800775 (1999)
4. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software Verification for Weak Memory via Program Transformation. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. LNCS, vol. 7792, pp. 512–532. Springer, Heidelberg (2013)
5. Alomari, M., Fekete, A., Röhm, U.: A robust technique to ensure serializable executions with snapshot isolation dbms. In: Proceedings of the 2009 IEEE International Conference on Data Engineering, ICDE 2009, pp. 341–352. IEEE Computer Society, Washington, DC (2009)
6. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 99–115. Springer, Heidelberg (2011)
7. Attiya, H., Ramalingam, G., Rinetzky, N.: Sequential verification of serializability. SIGPLAN Not 45, 31–42 (2010)

8. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
9. Cahill, M.J., Röhm, U., Fekete, A.D.: Serializable isolation for snapshot databases. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, pp. 729–738. ACM, New York (2008)
10. Minh, C.C., Chung, J.W., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: Proc. of the IEEE International Symposium on Workload Characterization, IISWC 2008 (September 2008)
11. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., Yerneni, R.: Pnuts: Yahoo!'s hosted data serving platform. Proceedings of the VLDB Endowment 1(2), 1277–1288 (2008)
12. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: Vcc: Contract-based modular verification of concurrent c. In: ICSE-Companion 2009, pp. 429–430 (May 2009)
13. Dalessandro, L., Scott, M.L., Spear, M.F.: Transactions as the Foundation of a Memory Consistency Model. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 20–34. Springer, Heidelberg (2010)
14. Daudjee, K., Salem, K.: Lazy database replication with snapshot isolation. In: Proceedings of the 32nd International Conference on Very Large Data Bases, pp. 715–726. VLDB Endowment (2006)
15. Dias, R.J., Distefano, D., Seco, J.C., Lourenço, J.M.: Verification of Snapshot Isolation in Transactional Memory Java Programs. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 640–664. Springer, Heidelberg (2012)
16. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: a race and transaction-aware java runtime. In: PLDI 2007: Proc. of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 245–255. ACM, New York (2007)
17. Fähndrich, M.: Static Verification for Code Contracts. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 2–5. Springer, Heidelberg (2010)
18. Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making snapshot isolation serializable. ACM Transactions on Database Systems (TODS) 30(2), 492–528 (2005)
19. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI 2002, pp. 234–245. ACM Press, New York (2002)
20. Grossman, D., Manson, J., Pugh, W.: What do high-level memory models mean for transactions? In: MSPC 2006: Proc. of the 2006 Workshop on Memory System Performance and Correctness, pp. 62–69. ACM Press, New York (2006)
21. Grossman, D., Manson, J., Pugh, W.: What do high-level memory models mean for transactions? In: Proceedings of the 2006 Workshop on Memory System Performance and Correctness, MSPC 2006, pp. 62–69. ACM, New York (2006)
22. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA 2003: Proc. of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, pp. 388–402. ACM Press, New York (2003)
23. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP 2005: Proc. of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 48–60. ACM Press, New York (2005)

24. Herlihy, M.: SXM1.1: Software transactional memory package for c#. Tech. rep., Brown University & Microsoft Research (May 2005)
25. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proc. of the Twentieth Annual International Symposium on Computer Architecture (1993)
26. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
27. Hindman, B., Grossman, D.: Atomicity via source-to-source translation. In: Proceedings of the 2006 Workshop on Memory System Performance and Correctness, MSPC 2006, pp. 82–91. ACM, New York (2006)
28. Kestor, G., Unsal, O.S., Cristal, A., Tasiran, S.: T-rex: A dynamic race detection tool for c/c++ transactional memory applications. In: Proceedings of the Ninth European Conference on Computer Systems, EuroSys 2014, pp. 20:1–20:12. ACM, New York (2014)
29. Kuru, I., Ozkan, B.K., Mutluergil, S.O., Tasiran, S., Elmas, T., Cohen, E.: Verifying programs under snapshot isolation and similar relaxed consistency models. In: Workshop on Transactional Computing, TRANSACT (2014)
30. Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.-R., Hudson, R.L., Saha, B., Welc, A.: Practical weak-atomicity semantics for java stm. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2008, pp. 314–325. ACM, New York (2008)
31. Papadimitriou, C.: The theory of database concurrency control. Computer Science Press (1986)
32. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)
33. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)
34. Sezgin, A., Tasiran, S., Muslu, K., Qadeer, S.: Run-time verification of optimistic concurrency. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 384–398. Springer, Heidelberg (2010)
35. Sezgin, A., Tasiran, S., Qadeer, S.: Tressa: Claiming the future. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 25–39. Springer, Heidelberg (2010)
36. Skare, T., Kozyrakis, C.: Early release: Friend or foe?. In: Workshop on Transactional Memory Workloads (June 2006)
37. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 385–400. ACM (2011)
38. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in bayou, a weakly connected replicated storage system. ACM SIGOPS Operating Systems Review 29(5), 172–182 (1995)
39. Titos, R., Acacio, M.E., García, J.M., Harris, T., Cristal, A., Unsal, O., Valero, M.: Hardware transactional memory with software-defined conflicts. In: High-Performance and Embedded Architectures and Compilation (HiPEAC 2012) (January 2012)