Verifying Programs under Snapshot Isolation and Similar Relaxed Consistency Models

Ismail Kuru
Koç University, Istanbul
ikuru@ku.edu.tr

Burcu Kulahcioglu Ozkan
Koç University, Istanbul
bkulahcioglu@ku.edu.tr

Suha Orhun Mutluergil
Koç University, Istanbul
smutluergil@ku.edu.tr

Serdar Tasiran
Koç University, Istanbul
stasiran@ku.edu.tr

Tayfun Elmas [1]
Google, Inc.
elmas@google.com

Ernie Cohen
Microsoft
ernie.cohen@acm.org

Abstract

We present a static verification approach for programs running under snapshot isolation (SI) and similar relaxed transactional semantics. Relaxed conflict detection schemes such as snapshot isolation (SI) are used widely. Under SI, transactions are no longer guaranteed to be serializable, and the simplicity of reasoning sequentially within a transaction is lost. In this paper, we present an approach for statically verifying properties of transactional programs operating under SI. Differently from earlier work, we handle transactional programs even when they are designed not to be serializable.

We present a source-to-source transformation which augments the program with an encoding of the SI semantics. Verifying the resulting program with transformed user annotations and specifications is equivalent to verifying the original transactional program running under SI – a fact we prove formally. Our encoding preserves the modularity and scalability of VCC's verification approach. We applied our method successfully to benchmark programs from the transactional memory literature.

1. Introduction

Transactions provide a convenient, composable mechanism for writing concurrent and distributed programs. They are used to write shared memory programs using transactional memory (TM), programs that access a single, central database, and, more recently, but increasingly widely, to write programs that access geo-replicated databases. In each of these settings, 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 paper is about a technique for verifying transactional programs that operate under relaxed semantics – a problem that has not yet been addressed.

For TM and database transactions, by strong semantics we mean atomicity and serializability of transactions. For programs operating on geo-replicated objects, we mean strong consistency of transactions. For TM and database transactions, the motivation for relaxed semantics is performance and avoidance of frequent transaction aborts, while, for geo-replicated data types, the motivation is providing availability to the objects or databases, even when the replicas, sometimes located on mobile devices, are disconnected (partitioned).

The best-known relaxed consistency semantics in the TM and database domains 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 (also known in TM literature as relaxed conflict detection schemes) other than SI, such as programmer-defined conflict detection [1], and early release of read set entries [2] have been investigated in the database, software and hardware transactional memory communities. For the distributed transactional programs, relaxed consistency semantics such as session SI [3] and parallel SI [4] have been investigated (See per-record time line consistency [5] and prefix consistency [6] for examples). Relaxations of strong semantics are widely used for geo-replicated databases are variants of eventual consistency, with additional guarantees relating different objects and the order in which different updates are propagated to different replicas.

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. Our verification technique is targeted at such transactional programs.

Although transactional programs operating under relaxed semantics are becoming increasingly commonplace, there are currently no tools for verifying properties (assertions and invariants). Static tools for code verification targeted at sequential programs [7–9], and the VCC verification tool [10] 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 transactional platforms providing more relaxed semantics such as SI or eventual consistency, these tools cannot be used as they are not aware of transactions or relaxed consistency

semantics. In this paper, we present a static code verification technique for transactional applications running under weak consistency semantics such as SI or eventual consistency. The goal of our technique is to provide a verification environment exactly like that of VCC but for programs running on relaxed 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.

In our approach, we take a transactional program and the semantics of the transactional platform that provides relaxed consistency. We produce an augmented C program with VCC annotations. The program our approach outputs is the same (has the same structure, etc.) 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. Our 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. The encoding avoids inlining code that other, interfering transactions that might be running concurrently.

To illustrate the applicability of our technique, we verified programs that were written to be correct even under relaxed transactional consistency. These programs were three benchmarks from the STAMP [11] transactional benchmarks suite and a `StringBuffer` pool example. We verified using our approach and tool that these examples satisfy application-level invariants and assertions despite relaxed transactional semantics. The user annotations required in each case reflected the correctness intuition in the relaxed transactional case, were relaxed versions of the annotations that would have been required had the program been running sequentially, and did not refer to the auxiliary variables involved in our encoding. In other words, the user was able to simply express the correctness intuition without referring to the mechanisms implementing the transactional semantics. Verification times for programs running under relaxed semantics were comparable with verification times for programs running sequentially.

Although we recognize the distinctions between different relaxed transaction semantics, in the rest of this paper, for brevity's sake, we use SI to refer to relaxed consistency models similar to SI and "serializability" to stand for the class of transactional platforms that provide atomic, strongly consistent, serializable transactions.

2. Motivating Example

In this section, we illustrate our approach (Figure 1) on the Labyrinth benchmark from the STAMP benchmark suite, one of the four benchmark programs we applied our method to. This example is typical of the design and correctness intuition for programs that satisfy desired assertions and invariants while operating under SI. The Labyrinth program is correct., i.e., satisfies the desired invariants and procedure post-conditions despite its executions not being serializable. Enforcing serializability (as is typically accomplished by enforcing conflict serializability [12]) would be an unnecessary restriction that hurts performance.

Labyrinth follows 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. Under conflict-serializability all concurrent transactions conflict and transactions can only run serially, one at a time.

As shown in 1, each concurrent transaction runs an instance of the function `FindRoute` to route a wire "Manhattan-style" 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

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

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

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

 5:    // Desired post-conditions of shortestPath:
 6:    assert(isValidPath(onePath, localGridSnapshot))
 7:    assert(isConnectingPath(onePath, p1, p2);
 8:
 9:    // Register points on onePath as "taken" on grid
10:    // Add onePath to pathsList
11:    gridAddPathIfOK(grid, pathsList, onePath);
12:
13:    // FindRoute must ensure program invariants,
14     // and the post-condition
15:    //   onePath in pathsList &&
16:    //        IsConnectingPath(onePath, p1, p2)
} }
```

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

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 `onePath` 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 invari-

ants as annotations. The rest of the verification of function post-conditions is carried out automatically.

Verifying `FindRoute` Under SI: 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. Our technique 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 7 and 8 still hold at the time `gridAddPathIfOK` starts running.

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.

While the exact form of the argument differs from benchmark to benchmark and can be somewhat more complicated than above, we have found that the argument has the following pattern:

1. The "read phase" and the "local computation phase" of the transaction establish some conditions in terms of program variables,
2. These post-conditions remain preserved even in the presence of interference allowed under SI.
3. These post-conditions suffice for the "write phase" to establish the desired invariants and transaction post-condition, and

In the example above, it was trivial to argue (3) since the post-conditions are in terms of transaction-local variables. For other examples, these post-conditions refer to shared state which may be mutated. The transaction remains correct as long as these mutations do not break the properties that the "write phase" depends on. Our tool allows the programmer to express these properties and verify that they remain preserved under interference allowed by SI.

3. Our Approach

3.1 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,
- $GlMem : GlVar \rightarrow Val$ maps global variables to their values in the memory, and
- $TtoLcSts : Tid \rightarrow L$ 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 $E_N = (\vec{s}, \vec{\alpha})$ where $\vec{\alpha}$ is a finite sequence of actions $\alpha_0, \alpha_1, \ldots, \alpha_{N-1}$ and $\vec{s} = s_0, s_1, \ldots, s_N$ is a finite sequence of states such that $(s_i, \alpha_i) \rightarrow s_{i+1}$ 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.

3.2 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.

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, 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 and Definition 1 coincide.

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 [1]. 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 $[i, Idx(commit(t))]$, 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.

3.3 Concurrency, VCC and Modular Verification

In this section, we briefly, and, due to space constraints, 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 coordinated, 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.

3.4 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

As explained in Section 3.1, the user can also specify a global invariant he would like to hold at the time a transaction commits.

The output of the transformation is a program $\widetilde{P_{SI}} = Encode(P_{SI})$ that will be verified using VCC. It runs under ordinary C semantics and contains the kinds of VCC annotations described in Section 3.3. We formally prove that 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 transaction. 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, and variants of eventual consistency in which transactions may see stale data but updates by transactions are required to be atomic.

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

$\widetilde{P_{SI}}$ 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]))
};
```

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.

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

```
Trans{
   bool holding[PInt];
   bool readsLockedSet[PInt];
   bool writesLockedSet[PInt];
};
```

In the definition above `PInt` stands for `struct Int*`. Fields of `Trans` are ghost maps. `readSetInt` and `writeSetInt` are maps that store `Int` objects read and written to by this transaction. `localIntCopy` keeps the transaction local values of `Int` s. If an `Int` object x is neither read nor written to by this transaction then `localIntCopy[x]` is null.

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.

Transformation 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. Due to space restrictions, we only provide highlights of the transformation rules:

- Statements of the form `beginTrans(`t`)` remain unchanged in the transformed version. (see pre- 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->lockedWritesInteger[P] ==>
             P->inTMVNo == P->inMemVNo + 1);
  commitTrans(t);
  assert(inv);
  ```

- For each statement `endTrans(t)`, we replace the statement with `endAndCleanTrans(t)` in the encoded version.
- 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->lockedReadsInteger[P] &&
             !t->writesLockSet[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 en-

coded 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.

- 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->readsLockSet[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->writesLockSet[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->writesLockSet[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.

- 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 readLockSet, writesLockSet, and holding are all reset to maps corresponding to empty sets.

The following theorem, the proof of which is available at
msrc.ku.edu.tr/projects/vcctm
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 (clearly automatable but not yet automated) 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.

3.5 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.

4. Experiments

We applied our technique to the Genome, Labyrinth and Self-Organizing Map benchmarks as implemented in [1] and a String-Buffer pool example that we wrote ourselves. Of the benchmark programs in [1], we picked these three because they made significant use of relaxed semantics to improve performance. These examples have pre-annotated transactional code blocks which can be run under relaxed transactional semantics, i.e., under SI, or, separately, using programmer-defined conflict detection as specified by !WAR annotations. All four of our examples are correct applications but their executions are not conflict serializable. 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. [1] 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.

Description and code for the benchmark examples and the results of the code transformation (verified with VCC) are available online at

`http://msrc.ku.edu.tr/projects/vcctm`.

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;}

1  bool list_insert(list_t *listPtr,
2                   node_t *node) {
3    node_t *prev, *curr = listPtr->head;
4
5    do {
6      prev = curr;
7      curr = curr->next;
8    } while (curr != NULL
9             && key > curr->key);
10   _(invariant loopInv(prev, curr, head, node))
11   // loopInv(prev, curr, head, node) ==
12   //           prevKey < key && prevKey < curKey
13   //           && prev reachable from head
14   //           && curr reachable from head
15
16   // assert(prev->next == curr);
17   node->next = curr;
18   prev->next = node;
19   return true; // key was not present
20 }
```

Figure 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 [11]. 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. 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 [1]. 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.

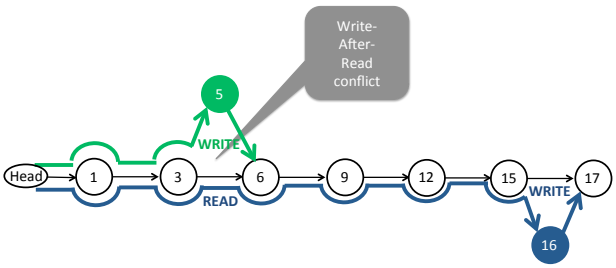

Figure 3. Sorted linked list and a write-after-read conflict.

- **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 paper.

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.

5. 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. [1] 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 [13–18]. 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 [19]. 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 [20, 21]. 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.

In this paper, we build on our earlier work [22] where we present a program abstraction that allows us to verify that the abstracted transactional program running under relaxed conflict detection is serializable. The verification approach we followed in [22] was different that it required a user-provided abstraction and was only partially mechanically checked. The approach we provide in this work is more general and the entire verification of the transactional program running under SI is carried out within the static verification tool VCC and the soundness of the verification approach is formally proven (Theorem 1).

6. Future Work

We plan to automate the transformations described in our paper. We also plan to and to apply our approach to verifying transactional programs running under other relaxed consistency models and eventual consistency.

References

[1] Titos, R., Acacio, M.E., Garca, 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)

[2] Skare, T., Kozyrakis, C.: Early release: Friend or foe? In: Workshop on Transactional Memory Workloads. (Jun 2006)

[3] Daudjee, K., Salem, K.: Lazy database replication with snapshot isolation. In: Proceedings of the 32nd international conference on Very large data bases, VLDB Endowment (2006) 715–726

[4] 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, ACM (2011) 385–400

[5] 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) (2008) 1277–1288

[6] 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. **29**(5) (1995) 172–182

[7] Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. CASSIS'04, Berlin, Heidelberg, Springer-Verlag (2005) 49–69

[8] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI '02, New York, NY, USA, ACM Press (2002) 234–245

[9] Fähndrich, M.: Static verification for code contracts. In: Proceedings of the 17th international conference on Static analysis. SAS'10, Berlin, Heidelberg, Springer-Verlag (2010) 2–5

[10] Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: Vcc: Contract-based modular verification of concurrent C. In: ICSE-Companion 2009. (may 2009) 429 –430

[11] Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC '08: Proc. of The IEEE International Symposium on Workload Characterization. (September 2008)

[12] Papadimitriou, C.: The theory of database concurrency control. Computer Science Press (1986)

[13] Dias, R.J., Distefano, D., Seco, J.C., Lourenço, J.: Verification of snapshot isolation in transactional memory Java programs. In Noble, J., ed.: ECOOP. Volume 7313 of Lecture Notes in Computer Science., Springer (2012) 640–664

[14] Attiya, H., Ramalingam, G., Rinetzky, N.: Sequential verification of serializability. SIGPLAN Not. **45** (January 2010) 31–42

[15] 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 '09, Washington, DC, USA, IEEE Computer Society (2009) 341–352

[16] 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 '08, New York, NY, USA, ACM (2008) 729–738

[17] Adya, A.: Weak consistency: a generalized theory and optimistic implementations for distributed transactions. PhD thesis (1999) AAI0800775.

[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) (2005) 492–528

[19] Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990) 463–492

[20] Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: Computer Aided Verification, Springer (2011) 99–115

[21] Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In Felleisen, M., Gardner, P., eds.: ESOP. Volume 7792 of LNCS., Springer (2013) 512–532

[22] Subasi, O., Elmas, T., Cristal, A., Harris, T., Tasiran, S., Tutos-Gil, R., Unsal, O.: On justifying and verifying relaxed detection of conflicts in concurrent programs. In: In WoDet'12