Abstract—Transactional memory (TM) provides an easy-to-use and high-performance parallel programming model for multicore systems. It simplifies parallel programming by supporting transactions that appear to execute atomically and in isolation. Despite the large amount of recent work on various TM implementations, very little has been devoted to precisely guarantee that these implementations have implemented the atomicity and isolation properties. In previous work we have proposed a framework on the correctness of STM programs by formally certifying the shared memory invariant at assembly level. Now the framework is extended and we certify the strong atomicity property of programs using STM in this paper. In particular, we formalize the strong atomicity as the shared memory consistence of states in our model and use a notion of “local guarantee” to check the shared memory consistence for verification. Our work provides a foundation for certifying realistic transactional programs and makes an important advance toward generating proof-carrying code.

Keywords—transactional memory; strong atomicity; proof-carrying code;

I. INTRODUCTION

The advent of multicore processors has brought concurrency into mainstream applications, however, it also brings great challenges to programmers for the concurrency management. They traditionally used locks to enforce synchronized concurrent memory accesses. However, locks are well-known software engineering issues that make parallel programming too complicated and may lead to problems such as deadlock, priority inversion, or convoying. Transactional memory provides an alternative concurrency management model that avoids these pitfalls associated with locks and significantly eases parallel programming.

TM simplifies concurrency management by supporting parallel tasks that appear to execute atomically and in isolation. In TM system, behavior of transactions must satisfy the following properties: a) atomicity: either the whole transaction executes or none of it; b) isolation: partial memory updates are not visible to other transactions [1].

Transactions should be atomic and isolated with respect to each other, but their relationship to non-transactional code is less clear. There are two models of atomicity semantics: strong atomicity semantics and weak atomicity semantics. Strong atomicity semantics is a semantics in which transactions are atomic and strongly isolated with respect both to other transactional and non-transactional codes. In essence, strong atomicity implicitly treats each instruction appearing outside transactions as its own singleton transaction. Weak atomicity semantics is a semantics in which transactions are atomic and weakly isolated only with respect to other transactions.

There have been several proposals for supporting transactional memory either by hardware [1], [2], [3] (HTM) or software [5], [6], [7], [8] (STM) with both approaches having their pros and cons. HTM provides a significant performance advantage, and enforces the strong atomicity semantics. However, HTM requires complicated hardware support and usually restricts the size of transactional code blocks. STM can easily support unbounded transactions, nested transactions with partial rollbacks, and conditional signaling [9]. However, STM lacks for performance so prior STM systems mostly implement weak atomicity, in which non-transactional codes are allowed to directly access shared memory and bypass the STM access protocols.

In STM programming, programmers mark the regions that should be executed atomically at high level; then the STM compiler packages operations in the region with libraries and certifies the shared memory invariant. It mainly focuses on the correct shared memory accesses in and out of transactions. However, the other significant properties such as the strong atomicity behaviors are entirely overlooked. In this paper, we attempt to extend our previous framework by formally certifying the strong atomicity semantics in concurrent programs using STM system. The atomicity and strong isolation properties are formalized as the consistence of visible shared memory between the beginning and the time after rolling back of a transaction in the extended framework. The consistence is certified by using a local guarantee that describes valid shared memory transitions. The local guarantees for the beginning and the time after rolling back of a transaction are constant while the others...
between transactional and non-transactional codes accessing the same shared data with at least one write access. In Figure 1 we present three examples of isolation violation for various STM implementations respectively.

- Single-lock STM: A transaction holds the global lock before execution to prevent interleaves from other transactions. Figure 1(a) illustrates an intermediate dirty read where thread 2 may read the intermediate value 1 from x. This intermediate dirty read may also exist in the eager-versioning STM, but can not occur in the lazy-versioning STM. In the single-lock STM system, there exists a global lock used for synchronization between transactions. However, non-transactional codes don’t hold the global lock before accessing shared data, so the anomaly happens.

- Eager-versioning STM [6], [7]: A transaction executes as though no other transaction is interleaved. It writes new values in place and logs old values, transaction can retry, undo changes when commit failed. Figure 1(b) illustrates a lost update where a non-transactional update is lost due to a transaction’s rollback. Assume thread 1 executes first and updates x to 2, then thread 2 executes and updates both x and y. Next transaction in thread 1 finds the happened conflict due to the modification of y by thread 2. It rolls back and restores x’s value back to 0 so thread 2’s update of x is lost.

- Lazy-versioning STM [8], [9]: Like eager-versioning STM, but it leaves old values in place and buffers new values. The new values will be updated on commit. The two anomalies above can not occur here, but it also has anomalies. Figure 1(c) illustrates overlapped writes where the commit operation isn’t atomic to non-transactional codes. Thread 1 initializes a field in the object el and then publishes the object by writing it to shared variable x. However, a lazy-versioning STM copies buffered values to the shared memory in no particular order. If thread 1 copies new value of x to memory first, then thread 2 executes and sees the published object in x but can not see the initialized value of its field. So r==0 holds in that case.

From these three examples we see that weak atomicity may lead to unexpected behaviors so it is important to implement the strong atomicity semantics in STMs. These anomalies of various STM systems we present above are all between transactional and non-transactional codes for the

### Figure 1. Anomalies on various STM implementations

<table>
<thead>
<tr>
<th></th>
<th>Initially x=0, y=0</th>
<th>Initially x=0</th>
<th>Initially x=null, e.v=0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 1</strong></td>
<td>atomic{</td>
<td>atomic{</td>
<td>atomic{</td>
</tr>
<tr>
<td></td>
<td>x = 1;</td>
<td>if (y==0)</td>
<td>r = -1;</td>
</tr>
<tr>
<td></td>
<td>r = x;</td>
<td>x = 1;</td>
<td>e.v = 1;</td>
</tr>
<tr>
<td></td>
<td>z = 2;</td>
<td>z = 2;</td>
<td>if (x != null)</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>y = 1;</td>
<td>x = e;</td>
</tr>
<tr>
<td></td>
<td>can r==1?</td>
<td>/<em>abort</em>/</td>
<td>r = x.v;</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>can x=0?</td>
<td></td>
<td>can r==0?</td>
</tr>
<tr>
<td><strong>Thread 2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thread 1</td>
<td>Thread 2</td>
<td></td>
</tr>
</tbody>
</table>

- It presents a train of thought that translates the strong atomicity in high level codes into the shared memory consistency at assembly level by a certifying compiler. Then we introduce local guarantees to certify the consistency in the extended framework.
- It presents a new design of the program logic after the added local guarantees in program specification. The soundness of the extended framework has also been proved.
- Our verification is fully mechanized in the Coq proof assistant [14]. We follow the FPCC style to give the soundness proof of the whole verification framework.

The rest of the paper is organized as follows. In section II we characterize weak atomicity behaviors and analyze the essence. In section III we summarize our previous framework on formal reasoning about transactional programs. Then the framework is extended with the atomicity proof in section IV, presenting the strong atomicity is indeed enforced in programs. We use an example to illustrate the verification in section V. Finally, we discuss related work and conclude in section VI.

### II. WEAK ATOMICITY BEHAVIORS

Strong atomicity provides a simple and intuitive view of transactional atomicity, which may be more difficult to implement efficiently. In contrast, weak atomicity provides a less intuitive model, but it may be easier to implement efficiently so prior STM systems mostly implement it. Under weak atomicity, there are many unexpected behaviors between transactional and non-transactional codes accessing...
lack of synchronization of shared data. To enforce the isola-
tion property between transactional and non-transactional
codes, there must be some mechanism which prevents non-
transactional accessing when a shared memory update has
arisen but has not been commit in transaction. These mecha-
nisms are bypassed in weak atomicity semantics due to
the performance while are required in the strong atomicity
semantics. Therefore, with strong atomicity semantics non-
transactional codes cannot observe the intermediate state of
transactions so these anomalies will not occur. Shpeisman et
al. [18] implement the strong atomicity by adding read and
write barriers in non-transactional codes. In our framework
described in next section, we modeled a software TM system
implementation based on storable locks. It uses storable
locks for synchronization of shared data both in and out
of transactions to implement the strong atomicity semantics.
And in section IV we use program logic for the verification
of atomicity property to ensure that programs have indeed
enforced the strong atomicity.

III. THE STM IMPLEMENTATION

In this section, we review our previous framework and
some key techniques used in the framework. It is at assembly
level for certifying safety properties in STM programs.
The whole framework contains two parts: abstract machine
model and program logic. Section III-A presents a simplified
STM system based on storable locks. Section III-B discusses
the program logic for proving the shared memory invariant.
The detailed presentation of the framework is in our latest
paper [13].

A. Abstract machine

(W) ::=(C.M,T1,...,Tn)

(T) ::= (R,pc,X)

(St) ::= (M.R,pc,X)

(C) ::= {f→1}∗

(M) ∈ Address → Word

(R) ∈ Register → Word

(r) ::= r0 | ... | r8

(f, l, pc) ::= i (nat num)

(w) ::= i (nat num)

(U) ::= act | cmt | abt

(B) ::= (R, pc)

(X) ::= e (H., R., B, U)

(i) ::= ... cas r1, r2, w(r3) | lw r1, w(r3)

| sw r1, w(r3) | lw r1, w(r3) | sw r1, w(r3)

| begin | validate r1 | commit

(l) ::= i[: j | f | rollback

(PS) ::= (n, {ψ1,...,ψm})

(CdHpSpec) ::= {1,...,a}∗

(StatePred) ::= ThreadState → Prop

(MemPred) ::= Memory → Prop

The syntax and verification constructs of the machine
is presented in Figure 2. A complete world consists of a
code heap C, a shared memory M and numbers of threads
made up of the register file R, the program counter pc and
transactional mechanism X. The transactional mechanism
is made up of a read set H, a write set Hw, a backup
file B and a transactional status U when in transaction.
We introduce three kinds of status act, cmt and abt for
transactions, denoting the various phases during the execu-
tion of a transaction. In status act, transaction reads data
from the shared memory, does some computation and then
buffers write attempts. Next if there are no conflicts, the
status of the transaction turns to cmt. Then the write buffers
commit and the transaction end. Otherwise the status of the
transaction turns to abt and the transaction rolls back. In this
model we use storable locks for synchronization of shared
memory among threads. The storable lock is implemented
by a non-reentrant spin-lock mutex which reserves a word
in memory to flag whether the mutex is locked or not. It
uses a cas instruction for locking while the normal write
instruction sw for unlocking. The locking behaviors are
limited in transactions. Locking instruction cas can only be
executed in status act while the unlocking instruction sw can
be anywhere. Once locking failed in transaction, the status
turns to abt and the transaction rolls back immediately.

The operational semantics of this model is presented in
Figure 3. It is a partial function Next_{pc,t} that computes
the next thread state of S performed by a given instruction
t when executed from location pc. The macro Npc_{pc,t} showed
in Figure 4 is a total function that computes the pc of
next instruction to be executed after the current instruction
t is completed.

\[
\text{if } t = \text{then } Npc_{pc,t}(M, R, pc, X) =
\]

\[
\text{bnz } r1, r2, f \quad \text{if } R(r1) = R(r2) \quad \text{pc} + 1 \quad \text{if } R(r1) \neq R(r2)
\]

\[
\text{bne } r1, r2, f \quad \text{pc} + 1 \quad \text{if } R(r1) = R(r2) \quad \text{f} \quad \text{if } R(r1) \neq R(r2)
\]

\[
\text{f} \quad \text{f}
\]

\[
\text{r} \quad R(r)
\]

\[
\text{rollback} \quad x \equiv pc
\]

\[
\text{Others} \quad pc + 1
\]

Figure 4. Auxiliary NextPC Macro

B. Program logic

We equip this machine with a construct \( \phi \) (Program
Specification) for expressing user-defined requirements. A
program specification is made up of a global invariant which
defines a criterion of the shared memory, and several code
heap specifications for threads. The program logic is a com-
bination of concurrent separation logic and permission
accounting in separation logic.

Concurrent separation logic (CSL) [10] is a well known
logic that provides a simple but powerful technique for
reasoning about shared memory concurrent programs. In
CSL, shared memory is partitioned and each part is protected by a unique lock. For each part of the partitions, an invariant is assigned to specify its well-formedness. When the lock is acquired, the thread takes advantage of mutual-exclusion provided by lock and treats the partial memory as private. Before releasing the lock, it must ensure that the part of memory is well-formed with regard to the corresponding invariant. With CSL, shared memory accesses must be put in conditional critical region to treat the part of memory as private, both in and out of transactions, then those anomalies in section II can not occur.

Permission Accounting in Separation Logic (PASL) [11] is a lightweight logical approach to race-free sharing of memory, based on the notion of permission to access. In PASL each cell in the shared memory associates with a permission set (zero and one in our model). A total permission (zero) can be split into two read-only permissions (one) as needed. On lock acquiring, one part with the read-only permission is moved to the thread’s private memory while the other one is left in shared memory for other threads’ speculative reads. When updating, it first combines the private memory with the shared memory to form a total permission, then updates both parts.

By the program logic that incorporates CSL and PASL, we can deal with speculative shared memory accesses in transaction. Our specification asserts the machine-level behavior of the program and is general enough for various safety requirements. Interested readers are referred to the previous work [13] for a complete modeling of the framework and a certified example.

IV. ATOMICITY PROOF

In this section, we extend our previous framework with the formal atomicity proof to enforce strong atomicity. In the previous work we mainly focus on the correct concurrent shared memory accesses and certify the shared memory invariant. However, the shared memory invariant is a well-formed characterization of the shared memory but lacks the ability for description of the relation between two different states. So it is impossible to certify the strong atomicity in our previous framework.

As we have mentioned, strong atomicity requires that transactions are atomic and strongly isolated. First, we formalize the atomicity property as the consistence of shared memory between thread states of the beginning and the time after rolling back of a transaction. Then aiming at the consistence, we introduce local guarantees in the program specification to describe valid shared memory transitions. The local guarantees for the beginning and the time after rolling back of each transaction are generated by the certifying compiler while the others can be generated automatically. In the atomicity proof, it implicitly requires to restore the modified data to previous state before the corresponding lock releasing in status act or abt, which is exactly the strong isolate property that we need.

By adding the local guarantee in the program specifications, the verification constructs and inference rules for program logic have to be extended too.

A. Inference rule

Verification constructs for program logic of our extended framework are introduced in Figure 5.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{WorldSpec} )</td>
<td>( \emptyset := (\emptyset, [p_1, \ldots, p_n]) )</td>
</tr>
<tr>
<td>( \text{CdlHypSpec} )</td>
<td>( \psi := { (n \sim \text{act} \ (a, g)) } )</td>
</tr>
<tr>
<td>( \text{StatePred} )</td>
<td>( n \in \text{ThreadState} \rightarrow \text{Prop} )</td>
</tr>
<tr>
<td>( \text{Guarantee} )</td>
<td>( g \in \text{ThreadState} \rightarrow \text{Memory} \rightarrow \text{Prop} )</td>
</tr>
<tr>
<td>( \text{MemPred} )</td>
<td>( n \in \text{Memory} \rightarrow \text{Prop} )</td>
</tr>
<tr>
<td>( \text{Interp}_a )</td>
<td>( { \text{Interp}_{\text{act}} } \in \text{StatePred} \times \text{Guarantee} \rightarrow \text{StatePred} )</td>
</tr>
<tr>
<td>( \text{Interp}_{\text{abt}} )</td>
<td>( { \text{Interp}_{\text{abt}} } \in \text{StatePred} \times \text{Guarantee} \rightarrow \text{Guarantee} )</td>
</tr>
</tbody>
</table>

Figure 3. Operational Semantics of the Machine

Figure 5. Verification Constructs for Program Logic
The whole world specification $\phi$ contains a global shared memory invariant $m$ and code heap specifications $\psi_1, \ldots, \psi_n$ for each thread. Compared with Figure 2, the code heap specification $\psi$ here adds a guarantee $g$ to each instruction sequence. The local guarantee $g$, just as in CCAP [16] and CMAP [17], describes valid shared memory transitions – it is safe for the current transaction to roll back only after making a memory transition allowed by $g$. Finally $\llbracket - \rrbracket_a$ and $\llbracket - \rrbracket_r$ take out the state predicate and guarantee from the pair $(a, g)$ respectively.

The memory predicate $m$ takes a style of PASL which is shown in Figure 6.

$$m ::= 1 \lor ^a v \mid m_1 \land m_2 \mid m_1 \lor m_2 \mid \exists x. m \mid \forall x. m$$

**Figure 6.** Permission Accounting in Separation Logic

$$a ::= \varepsilon \mid m \mid [m], [r] = v \mid [r] = v \mid \lambda. [x] = v \mid \lambda. [x] = v \mid \exists x. a \mid \forall x. a$$

$$\epsilon \equiv \lambda S. S = \epsilon \quad [r] = v \equiv \lambda S. S. R(r) = v \quad [m] \equiv \lambda S. S.M \quad \lambda. [x] = v \equiv \lambda S. S.X = v \quad \exists x. a \equiv \lambda S. \exists x. aS \quad \forall x. a \equiv \lambda S. \forall x. aS$$

$$U = \text{act/cmty/abt} \equiv \lambda S. S.X.U = \text{act/cmty/abt}$$

**Figure 7.** Assertion Language for Thread State

The assertion language for thread state is presented in Figure 7. To encode the specification and proofs, we take the use of Coq [14] and the underlying CiC [15] for mechanical verification. We encode the syntax using inductive definitions, and define the operational semantics and inference rules as a collection of relations.

We introduce some useful auxiliary definitions for program logic in Figure 8. And the macro $En \ (t)$ for required transactional status of each instruction $t$ is presented in Figure 9. They are the same as ones in our previous paper.

$$a \Rightarrow a' \equiv \forall S. a S \Rightarrow a' S$$

**Figure 8.** Auxiliary Definition for Program Logic

We use the following judgement forms to define the inference rules:

$$\phi_1, [(a_1, g_1), \ldots, (a_n, g_n)] \vdash W \quad \text{(well-formed world)}$$

$$\psi, m \vdash C, \psi' \quad \text{(well-formed code heap)}$$

$$\psi, m \vdash (a, g)pc \vdash I \quad \text{(well-formed instr. sequences)}$$

**Figure 9.** Required Transactional Status for Instructions

The inference rules for the extended framework are presented in Figure 10. A world is well-formed with regard to a world specification $\phi$ and pairs of thread state predicates and guarantees $(a_1, g_1), \ldots, (a_n, g_n)$ for each thread when the following conditions hold:

- There exists a code heap specification $\psi_i$ for each thread and a global invariant $m$ in the world specification $\phi$. For each thread, the code heap is well-formed regarding $\psi_i$ and $m$, moreover, the thread precondition pair $(a_i, g_i)$ is satisfied at the point of $pc_i$.
- There is a $n + 1$ parts partition of the shared memory $M$, where $M_i$ satisfies the global invariant $m$ and $M_1, \ldots, M_n$ satisfy each thread state predicate $a_i$ respectively.

A code heap is well-formed only if each instruction sequence in the code heap is well-formed.

Next, an instruction sequence is well-formed if it is composed of a single instruction $t$ and another instruction sequence $I$ and both of them are well-formed (rule-INSQ). A well-formed instruction (rule-INSN) requires:

- Thread state predicate restrict as previous: instruction $t$ can execute for all thread states specified by the current thread state predicate $a$ and the global invariant $m$. Furthermore, the new modified thread state must satisfy the thread state predicate for the target address of instruction $t$ given by $\psi$ and reestablish the global invariant $m$.
- Added restrict on local guarantee: if the current transaction is in status act or abt and the modified thread state satisfies the guarantee of the target address, then the original state satisfies $g$.

Here we left rules of instructions begin, commit and rollback alone. Rule-COMMIT checks the domain of private memory at current state with the beginning of transaction to enforce that all locks have been released before commit. Next rule-ROLLBACK records memory at the state of rolling back and goes back to check the consistency with memory of the beginning in rule-BEGIN.

Suppose the thread state transition sequence of the transaction (begin with begin and end with rollback) is $S_0, \ldots, S_n$. To show that the atomicity of the transaction is enforced through the guarantee $g$, we enforce the following chain of
In this section, we give an example to demonstrate the implications in the next section. The isolation property is well enforced. We will show an atomicity proof breakdown. In order to not be mentioned and guarantee states, so it cannot mention the partial memory which might have been privatized. In this way, lw, just delays memory privatization, not indeed pass it. So together with CSL and the atomicity proof, we can ensure that the strong atomicity is certified in our framework.

B. Soundness

The soundness of our extended framework inference rules with respect to the operational semantics for the machine is established following the syntactic approach of proving type soundness [20]. From the “progress” and “preservation” lemmas, we can guarantee that given a well-formed world under compatible assumptions, the current instruction sequence will be able to execute without getting “stuck”. Furthermore, any safety property derivable from the global invariant will hold throughout the execution and the strong atomicity is well enforced. We define \( W \rightarrow^n W' \) as the relation of \( n \)-step (\( n \geq 0 \)) world transitions. The soundness of the framework is formally stated as Theorem 4.3.

Lemma 4.1 (Progress)

For any \( W = (C, M, \{T_1, \ldots, T_n\}) \), if \( \phi, ([a_1, g_1], \ldots, [a_n, g_n]) \models W \), then for any thread \( T_i \), there exists \( M', T'_i \), such that \( (M, T_i) \rightarrow (M', T'_i) \).

Lemma 4.2 (Preservation)

If \( \phi, ([a_1, g_1], \ldots, [a_n, g_n]) \models W \), and \( W \rightarrow W' \), then \( (a_1', g_1'), \ldots, (a_n', g_n') \) specify different memories and the atomicity proof is breakdown. In order to not be mentioned in the local guarantee, the memory block must be restored to the previous state when the corresponding lock is released. So that its partial updates are invisible to other threads and the isolation property is well enforced. We will show an example to illuminate the details in the next section.

CSL well synchronized shared memory accesses both in and out of transactions except for the speculative read lw. It seems to break the isolation of concurrent threads due to the speculative read’s accessing shared memory without synchronization. However, the speculative read is only allowed in transactions and always followed by a conflict detection validate to check the validity of values. Instruction validate requires that locations that have been read speculatively must have been privatized. In this way, lw, just delays memory privatization, not indeed pass it. So together with CSL and the atomicity proof, we can ensure that the strong atomicity is certified in our framework.

The soundness of our extended framework inference rules with respect to the operational semantics for the machine is established following the syntactic approach of proving type soundness [20]. From the “progress” and “preservation” lemmas, we can guarantee that given a well-formed world under compatible assumptions, the current instruction sequence will be able to execute without getting “stuck”. Furthermore, any safety property derivable from the global invariant will hold throughout the execution and the strong atomicity is well enforced. We define \( W \rightarrow^n W' \) as the relation of \( n \)-step (\( n \geq 0 \)) world transitions. The soundness of the framework is formally stated as Theorem 4.3.

Lemma 4.1 (Progress)

For any \( W = (C, M, \{T_1, \ldots, T_n\}) \), if \( \phi, ([a_1, g_1], \ldots, [a_n, g_n]) \models W \), then for any thread \( T_i \), there exists \( M', T'_i \), such that \( (M, T_i) \rightarrow (M', T'_i) \).

Lemma 4.2 (Preservation)

If \( \phi, ([a_1, g_1], \ldots, [a_n, g_n]) \models W \), and \( W \rightarrow W' \), then \( (a_1', g_1'), \ldots, (a_n', g_n') \) specify different memories and the atomicity proof is breakdown. In order to not be mentioned in the local guarantee, the memory block must be restored to the previous state when the corresponding lock is released. So that its partial updates are invisible to other threads and the isolation property is well enforced. We will show an example to illuminate the details in the next section.

V. Example

Our framework is a realization of established verification techniques at the assembly level for concurrent programs. In this section, we give an example to demonstrate the mechanized verification of strong atomicity property for concurrent assembly code using our machine model.

A simple example of Fibonacci program is presented in Figure 11, which is the concurrent code that computes the next element of a Fibonacci sequence. The routine computes
the Fibonacci number by storing the last two numbers of the sequence into internal variables prev and curr. The variables prev and curr are shared between threads so it needs synchronization for access which is hidden in the CommitTransaction( ).

```c
int fib() {
    tx = StartTransaction();
    val_1 = StmtRead(tx, &prev);
    val_2 = StmtRead(tx, &curr);
    temp = val_1 + val_2;
    StmtWrite(tx, &prev, val_2);
    StmtWrite(tx, &curr, temp);
    }while(!CommitTransaction(tx));
}
```

**Figure 11.** Fibonacci Program

The assembly code for routine fib is presented in Figure 12. In the code it use a storable lock mutex for synchronization which relates to the inline synchronization in CommitTransaction( ). Together with the assembly code, we also present the set of precondition pairs and the shared memory invariant of the program Fibonacci for verification in Figure 12.

The shared memory invariant m and assertions a_i are the same as our previous paper [13]. Besides, we introduce guarantees g_0, g_1, g_2 for the atomicity proof. Just as mentioned in section IV, it records the shared memory before the time of rolling back in the local guarantee and then goes backwards to check the consistence at the beginning. g_0 is a constant truth predicate which is usually used in status cmt or outside transactions. g_1 states that consistence is satisfied, it is safe now to roll back the transaction. Next instruction cas acquires the storable lock mutex. If succeed, cell mutex of the shared memory is modified and the local guarantee turns to g_2 which states that cell mutex must restore to zero before rolling back. Then the lock is released at label ulk, the local guarantee turns back to g_1 again. Finally the control transfers to rollback, it initializes the parameter M in the local guarantee with the shared memory at current state and passes it back for checking. After the whole verification, we can confirm that atomicity is indeed enforced.

The property strong isolation of transactions is also implied in the atomicity proof. In the atomicity proof it requires that the memory block must be restored to the previous state before its corresponding lock is released in rolling back, so the partial updates is not visible to other threads. Here we modify the pre-example to show the breakdown of strong isolation property due to the violation in atomicity proof when the memory block have not been restored to initial state before unlocking.

In Figure 13 we add an instruction to modify cell curr but do not restore it to previous value before releasing mutex. After the modification, the local guarantee changes to g_3 which denotes that it is safe to roll back only after restore cell mutex to zero and cell curr to fib(n + 1). Next the storable lock mutex is released and the post condition is (a_7, g_4). g_4 denotes that cell curr must be restored to fib(n + 1) before rolling back. However, when the condition (a_7, g_4) is applied in the rule of the next instruction the proof breaks down. Cell curr mentioned in g_4 may not be in the memory described by the assertion a_7 ⊗ m because that
thread does not hold the storable lock anymore. Then \( g_4 \) is a false predicate due to the conflict between \( a_7 \) and \( g_4 \) and the proof can not go ahead. In order to finish the atomicity proof it must first restore \( curr \) and then \( g_4 \) will not mention cell \( curr \) anymore. In this way when a lock is released not in status \( cnt \), it implicitly acquires that the protected memory block looks like unchanged before. Then the strong isolation entirely holds since the intermediate memory updates are invisible to other threads. In summary, both the atomicity and strong isolation properties are guaranteed in the atomicity proof.

VI. RELATED WORK AND CONCLUSIONS

Transactional memory, as applied to programming languages, was first studied by Herlihy and Moss [1]. The primary goal is to make it easier to perform general atomic updates of multiple independent memory words, avoiding the problems of locks. It is a hardware implementation and rely on the assumption that transactions have short durations and small data sets. Shavit and Touitou [4] proposed the first software implementation handling transactions with statically known read and write sets. Next Blundell [12] distinguishes subtleties of transactional memory atomicity semantics. It defines strong atomicity and weak atomicity semantics to characterize the atomicity property between transactional and non-transactional codes. Presently prior software TM systems mostly implement weak atomicity and it allows violation of a transaction’s isolation if there is a data race between transactional and non-transactional codes. To enforce the strong atomicity semantics in STM, Shpeisman [18] implements non-transactional data accesses via read and write barriers. Recently, Guerraoui [22] presents opacity, an extension of the classical database serializability property with additional requirement that non-committed transactions are prevent from accessing inconsistent states. However, all these work above are lack of a formal reasoning of these correctness properties in STM implementations.

O’Hearn [10] proposed CSL for a high-level parallel language based on the separation logic [23]. It explicitly separates the private and shared memories and uses conditional critical regions (CCR) to permit the ownership transfer. CSL can specify well synchronized shared memory accesses and we adopt it in the program logic to formal reasoning strong atomicity. Recently, Brookes [24] provides a grainless semantics to CSL for parallel programs that share mutable states; Bornat et al. [11] proposed a refinement of CSL with fine-grained resource accounting.

In [25] we have presented a PCC framework for certifying concurrent programs using transactional memory. It treats the whole commit operation as a single primitive and there are no shared memory accesses outside transactions, so the strong atomicity semantics is enforced obviously. Next we refine the model by splitting the huge commit operation into several thin instructions and introduce storable locks for synchronization both in and out of transactions in [13]. At the same time the strong atomicity semantics is not enforced for sure and it depends on the actual implementation. In this paper we mainly focus on the formal reasoning the strong atomicity of STM programs. Actually we introduce a local guarantee for each instruction to specify valid memory transition before rolling back to enforce the strong atomicity. It is the first framework on formal certifying strong atomicity semantics in STM programs.

ACKNOWLEDGMENTS

We would like to thank Prof. Zhong Shao (Yale University) and anonymous for their inspiring discussions and suggestions on this paper. This research was supported by the National Natural Science Foundation of China under (Grant No.60673126, No.90718026) and Intel China Research Center. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

REFERENCES


