

Formal reasoning about concurrent programs using a lazy-STM system

Yong Li^{1,2}, Yu Zhang^{1,2}, Yi-Yun Chen^{1,2} and Ming Fu^{1,2}

¹*Department of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, P.R.China*

²*Software Security Laboratory, Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215123, P.R.China*

E-mail: {liyong, fuming}@mail.ustc.edu.cn; {yuzhang, yiyun}@ustc.edu.cn

Abstract Transactional memory (TM) is an easy-using parallel programming model that avoids common problems associated with conventional locking techniques. Several researchers have proposed alternative hardware and software TM implementations. However, few ones focus on formal reasoning about programs using TM system. In this paper, we propose a framework at assembly level for reasoning about concurrent programs using a lazy-STM system.

First, we give a software TM implementation based on storable locks. Then we define the semantics of the model operationally, and the synchronization constructs in transaction are light-weight and non-blocking, and it will not lead to deadlocks in transaction. Finally we devise a logic – a combination of permission accounting in separation logic and concurrent separation logic – to verify various properties of concurrent programs based on this machine model. The whole framework is formalized using a proof-carrying-code (PCC) framework.

Keywords program verification, transactional memory, proof-carrying-code, permission accounting in separation logic

1 Introduction

The advent of multi-core processors has brought concurrency into mainstream applications, how-

ever, it also brings great challenges to programmers for concurrency management. They traditionally used locks to enforce synchronized concurrent accesses. However, locks are well-known software engineering issues that make parallel programming exactly complicated and may lead to problems such as deadlock, priority inversion, or convoying. Transactional memory (TM) pro-

Regular paper.

Supported by the National Natural Science Foundation of China under (Grant No.60673126, No.90718026); Intel China Research Center

vides an alternative concurrency management model that avoids these pitfalls associated with locks and significantly eases parallel programming.

TM system simplifies concurrency management by supporting parallel tasks that appear to execute atomically and in isolation. There have been several proposals for hardware-based (HTM) [1, 2, 3, 4], software-based (STM) [5, 6, 7, 8, 9, 10], and hybrid [11, 12] implementations. But these various TM systems mainly focus on the performance issues of implementation and suggest different tradeoffs in the mechanisms used to track transactional state and buffer size, and the overheads associated with basic operations. There also exist a lot of work for the formalization of TM system [2, 13, 14, 15], but few focus on formal reasoning about programs based on these systems and making sure their soundness. Prior software TM systems mostly implement *weak atomicity* [14], which allows violation of a transaction's isolation if there is a data race between transactional and non-transactional codes. So programs based on these systems may lead to unexpected behaviors. Our previous work [16] has presented a framework for certifying concurrent programs using transactional memory. It is an abstract TM model that treats the huge committing operation of transactions as an instruction primitive.

In this paper, we extend our previous framework by splitting the huge committing opera-

tion of transaction into several small fundamental instructions. We have studied a wide range of existing software TM systems and designed an abstract machine model based on locks. It also supports irreversible actions in transaction by the technique of privatization. Besides the common shared memory invariant, programs using TM system also have some special properties related to TM. In this framework we focus on the correctness of programs using this TM system and formally prove these various properties in programs. Under the certification, we can declare that these programs are partial correct TM programs with respected to defined specifications. This paper makes the following novel contributions:

- First we model a software TM system based on storable locks. It is at assembly level with lazy version management and lazy conflict detection [9, 10]. We use storable locks to privatize the shared memory for conflict detection and data commit. Furthermore, we separate the whole progress of transactions into three phases, then irreversible action such as system calls and I/O can be delayed until data commit by compiler due to transaction's non-rolling back.
- We introduce a program logic that incorporates the *concurrent separation logic* [17] with *permission accounting in separation logic* [18] to deal with shared memory ac-

cesses in transaction. It splits each cell in the shared memory into two read-only parts, one of which is transferred to the locker’s private memory and the other one is left in the shared memory for read by other transactions. Our specification asserts the machine-level behavior of the program and is general enough for various safety requirements.

- The whole framework is implemented in Coq proof assistant [23] and we have proved several examples. The reasoning we describe at assembly level can be easily lifted up to higher levels.

The rest of the paper is organized as follows. Section 2 summarizes transactional memory and discusses the major design points in our framework. We propose our framework in section 3, it contains an abstract machine model and the program logic we use to reason. Then an example is shown to demonstrate how to reason about programs with the program logic in section 4. Finally, we discuss related work and conclude the paper in Section 5.

2 Transactional Memory

2.1 Transaction Memory Overview

Transactional memory system provides a simple concurrency control mechanism for parallel programming, it avoids many of the pitfalls associated with traditional locks. With TM, pro-

grammers only need to define an atomic code block, then the underlying system guarantees that the code block executes 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; and c) *consistency*: there is a single order of completion for transactions across the whole system [1]. Providing these properties requires *data version management* and *conflict detection*, whose implementations distinguish alternative TM proposals.

Data version management stores both new data and old data simultaneously, the new data is visible if transaction commits and the old data retains if transaction aborts. At most one of them can be stored in place (in the target memory) while the other one is buffered speculatively. On a store, a TM system can use *lazy version management* [6, 9] or *eager version management* [4, 7]. Lazy version management retains old data in place for faster abort and buffers new data in a redo-log speculatively. The redo-log defers all memory updates until transaction commits. Eager version management puts the new value in place for faster commit and logs the old value in an undo-log. Shared memory read is speculative (escape synchronization) in transaction, it needs to log the data in a read set for conflict detection before commit.

Conflict detection signals an overlap access of the same address by many transactions when at least one of them attempts to write a new value. It requires tracking addresses and values read by each transaction, and usually uses a read set to record them. *Lazy conflict detection* [6, 9] delays the conflict detection until transaction commits while *eager conflict detection* [7] checks for conflicts at reads and writes immediately. When conflict happens, the transaction needs to be rolled back. Prior STM systems [6, 9] provide a lazy conflict detection. For conflict detection, STM system can acquire locks for shared memory privatization to avoid interleaving from other transactions. The granularity of conflict detection may be at word-level, cache-line-level or object-level.

2.2 Design Points

In TM, transactional codes execute atomically and in isolation. Furthermore, the isolation is not only established between transactional codes but also between transactional and non-transactional codes. Existing software systems mostly implement the weak atomicity semantics, which allows violation of TM system's isolation when there exists a data race between transactional and non-transactional codes. Besides, the commit operation of transaction is assumed to be atomic, but in practice some STM implementations simplify this by allowing the non-atomic commit operation to improve performance.

Our previous work [16] avoids these problems by using a huge build-in primitive to express the whole committing operation of transaction, which detects conflicts and then commits new values in only a single instruction cycle. This seems to be at a high level and hides actual implementations.

In this paper we model a relaxed abstract transactional memory machine by splitting the huge commit operation in our previous work [16] into several low-level fundamental instructions, and storable locks are introduced for shared memory privatization to solve problems mentioned above. Each location in the shared memory is protected by a unique storable lock. The storable lock is implemented by a mutex reserving a word in memory to flag whether the mutex is locked or not. The acquirement of mutex is non-blocking and may abort transactions to avoid deadlocks. According to the implementations [6, 9, 10] of most STM systems, the model takes a lazy version management and lazy conflict detection at word-level granularity. In this framework we do not take nested transaction into account but leave it for future work.

3 The framework

In this section, we present an abstract machine that supports transactional memory and then give its operational semantics. Then a program logic that incorporates the *concurrent separation logic* (CSL) with *permission accounting in*

separation logic (PASL) is presented for the verification of assembly programs running on this model, it is similar in structure to other CAP systems [20, 21].

3.1 The Abstract Machine

(World)	\mathbb{W}	$::= (\mathbb{C}, \mathbb{M}, [\mathbb{T}_1, \dots, \mathbb{T}_n])$
(ThreadState)	\mathbb{S}	$::= (\mathbb{M}, \mathbb{T})$
(CodeHeap)	\mathbb{C}	$::= \{\mathbf{f} \rightsquigarrow \iota\}^*$
(Memory)	$\mathbb{M}, \mathbb{H}_r, \mathbb{H}_w$	$\in \text{Address} \rightarrow \text{Word}$
(Thread)	\mathbb{T}_i	$::= (\mathbb{R}, \text{pc}, \mathbb{X})$
(RegFile)	\mathbb{R}	$\in \text{Register} \rightarrow \text{Word}$
(Register)	\mathbf{r}	$::= \mathbf{r}_0 \mid \dots \mid \mathbf{r}_{31}$
(Address)	$\mathbf{f}, \mathbf{l}, \text{pc}$	$::= i(\text{nat nums})$
(Word)	\mathbf{w}	$::= i(\text{nat nums})$
(Xstate)	\mathbb{X}	$::= \varepsilon \mid (\mathbb{H}_r, \mathbb{H}_w, \mathbb{B}, \mathbb{A})$
(Bstate)	\mathbb{B}	$::= (\mathbb{R}, \text{pc})$
(Status)	\mathbb{A}	$::= \text{act} \mid \text{cmt} \mid \text{abt}$
(Instr)	ι	$::= \text{addu } \mathbf{r}_d, \mathbf{r}_s, \mathbf{r}_t \mid \text{addiu } \mathbf{r}_d, \mathbf{r}_s, \mathbf{w}$ $\mid \text{lw } \mathbf{r}_d, \mathbf{w}(\mathbf{r}_s) \mid \text{sw } \mathbf{r}_d, \mathbf{w}(\mathbf{r}_s)$ $\mid \text{casc } \mathbf{r}_d, \mathbf{r}_t, \mathbf{w}(\mathbf{r}_s) \mid \text{out } \mathbf{r}_d$ $\mid \text{beq } \mathbf{r}_s, \mathbf{r}_t, \mathbf{f} \mid \text{bne } \mathbf{r}_s, \mathbf{r}_t, \mathbf{f}$ $\mid \text{lw } \mathbf{r}_d, \mathbf{w}(\mathbf{r}_s) \mid \text{sw } \mathbf{r}_d, \mathbf{w}(\mathbf{r}_s)$ $\mid \text{begin} \mid \text{validate } \mathbf{r}_d \mid \text{commit}$
(InstrSeq)	\mathbb{I}	$::= \iota; \mathbb{I} \mid \mathbf{j} \mathbf{f} \mid \mathbf{j} \mathbf{r} \mathbf{r}_s$ $\mid \text{rollback}$

Figure 1: Syntax of Machine

The abstract machine is a straightforward extension of CAP by adding several transactional instructions (see Figure 1 for the syntax). The machine configuration contains a code heap \mathbb{C} , a global shared memory \mathbb{M} and a numbers of threads, each thread \mathbb{T} is made up of a register file \mathbb{R} , a program counter pc and \mathbb{X} . The \mathbb{X} is a special data structure for transaction, it is ε outside transaction while consists a read

set \mathbb{H}_r , a write set \mathbb{H}_w , a backup file \mathbb{B} and a transactional status \mathbb{A} inside transaction. The read set \mathbb{H}_r records values that are speculatively read from the shared memory for later conflict detection in transaction, the write set \mathbb{H}_w is a redo-log that buffers write attempts in transaction. At the beginning of a transaction the backup file \mathbb{B} records the register file \mathbb{R} and the program counter pc for rolling back once conflict happens. Thread state \mathbb{S} is a view of the machine configuration in a single thread's angle, containing part of the shared memory \mathbb{M} (allow to be accessed) and the corresponding thread \mathbb{T} . In this model we introduce three kinds of status **act**, **cmt** and **abt** for transactions, denoting various phases during the execution of the transaction. In status **act**, transaction reads and writes shared memory speculatively, then acquires locks and detects conflicts. If there are no conflicts, the status is turned to **cmt**. In this status transaction commits the data buffered in status **act**, then releases locks and completes. Irreversible actions such as I/O can be placed here by compiler or runtime system because transactions can not be rolled back in **cmt**. If conflict happens or locking failed, the status is turned to **abt**, then transaction only needs to release locks and rolls back.

The instruction set we present in the model is based on a subset of MIPS, and with several additional instructions for the implementation of transaction. Instruction **begin** marks the start of a transaction while **commit** marks the end,

and instruction `validate` is a representation of the conflict detection. Transactional instruction `lwt` is used for speculative read while `swt` denotes for speculative write. Next `out` is a representation of I/O. Finally we use a *compare-and-swap* (CAS) instruction to acquire the storable lock. Each location in the shared memory is protected by a unique storable lock.

The operational semantics is defined in Figure 2, which is a relation detailing the effect of instructions on thread state. We follow the preemptive thread model where execution of threads can be preempted at any program point, but execution of individual instructions is atomic.

Most of instructions are standard and straightforward, we only need to focus on those related to transaction. Instruction `begin` initializes the \mathbb{X} with an empty read set and an empty write set, and then records the current register file and program counter in the backup file, at last marks the status `act`. In this model we do not support nested transaction, so it requires that the \mathbb{X} must be ε when instruction `begin` executes. Then instruction `validate` checks the read set to detect conflicts. It compares each value recorded in the read set \mathbb{H}_r with the one at the same location in shared memory. If there exists one cell whose values in the read set and shared memory do not match, conflict happens. During the execution of instruction `validate`, the status is turned to `abt` if conflicts exist or `cmt` otherwise. Next instruc-

tion `commit` ends transactions by resuming the special mechanism \mathbb{X} to ε , it requires that the current transactional status must be `cmt`. Instruction `rollback` restarts the transaction by restoring the register file and program counter with the old one recorded in the backup file, and it must be executed at status `abt`. Instruction `cast` is a non-blocking synchronization here. It requires to be executed at status `act` and changes status to `abt` if locking fails. The corresponding operation for storable lock release is normal writing instruction `sw`, writing the lock cell a word that denotes it is free.

There are two modes of instructions for accessing the shared memory, the ordinary ones (`lw` and `sw`) require synchronization, while the special ones (`lwt` and `swt`) can be used in transaction without synchronization. In transaction, values must be consistent during various reads and writes. So instruction `lwt` first checks the write set \mathbb{H}_w and then the read set \mathbb{H}_r , if not presented in both sets (it is the first time accessing this location in transaction) it reads from the shared memory speculatively and logs the value in the read set \mathbb{H}_r for later conflicts detecting. With lazy version management, speculative write `swt` buffers new values in the write set \mathbb{H}_w and leaves the old values in place, these new values will be committed to the shared memory in status `cmt`. Instruction `sw` has two usages (data writing and lock release) in our model, and it isn't allowed to be executed in status `act` due to the lazy version management.

if $\iota =$	then $\text{Next}_{(\text{pc}, \iota)}(\mathbb{M}, \mathbb{R}, \text{pc}, \mathbb{X}) =$	
addu r_d, r_s, r_t	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{R}(r_s) + \mathbb{R}(r_t)\}, \text{pc} + 1, \mathbb{X})$	
addiu r_d, r_s, w	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{R}(r_s) + w\}, \text{pc} + 1, \mathbb{X})$	
lw $r_d, w(r_s)$	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{M}(\mathbb{R}(r_s) + w)\}, \text{pc} + 1, \mathbb{X})$	where $\mathbb{R}(r_s) + w \in \text{dom}(\mathbb{M})$
sw $r_d, w(r_s)$	$(\mathbb{M}\{\mathbb{R}(r_s) + w \rightsquigarrow \mathbb{R}(r_d)\}, \mathbb{R}, \text{pc} + 1, \mathbb{X})$	where $\mathbb{R}(r_s) + w \in \text{dom}(\mathbb{M}) \wedge \mathbb{X}.A \neq \text{act}$
beq r_s, r_t, f	$(\mathbb{M}, \mathbb{R}, f, \mathbb{X})$	if $\mathbb{R}(r_s) = \mathbb{R}(r_t)$
	$(\mathbb{M}, \mathbb{R}, \text{pc} + 1, \mathbb{X})$	if $\mathbb{R}(r_s) \neq \mathbb{R}(r_t)$
bne r_s, r_t, f	$(\mathbb{M}, \mathbb{R}, \text{pc} + 1, \mathbb{X})$	if $\mathbb{R}(r_s) = \mathbb{R}(r_t)$
	$(\mathbb{M}, \mathbb{R}, f, \mathbb{X})$	if $\mathbb{R}(r_s) \neq \mathbb{R}(r_t)$
j f	$(\mathbb{M}, \mathbb{R}, f, \mathbb{X})$	
jr r_s	$(\mathbb{M}, \mathbb{R}, \mathbb{R}(r_s), \mathbb{X})$	
out r_d	$(\mathbb{M}, \mathbb{R}, \text{pc} + 1, \mathbb{X})$	where $\mathbb{X}.A = \text{cmt}$
begin	$(\mathbb{M}, \mathbb{R}, \text{pc} + 1, (\emptyset, \emptyset, (\mathbb{R}, \text{pc}), \text{act}))$	where $\mathbb{X} = \varepsilon$
validate r_d	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow 1\}, \text{pc} + 1, (\mathbb{H}_r, \mathbb{H}_w, \mathbb{B}, \text{cmt}))$	if $\mathbb{X}.\mathbb{H}_r \subseteq \mathbb{M} \wedge \mathbb{X}.A = \text{act}$
	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow 0\}, \text{pc} + 1, (\mathbb{H}_r, \mathbb{H}_w, \mathbb{B}, \text{abt}))$	if $\mathbb{X}.\mathbb{H}_r \not\subseteq \mathbb{M} \wedge \mathbb{X}.A = \text{act}$
commit	$(\mathbb{M}, \mathbb{R}, \text{pc} + 1, \varepsilon)$	where $\mathbb{X}.A = \text{cmt}$
rollback	$(\mathbb{M}, \mathbb{R}', \text{pc}', \varepsilon)$	where $\mathbb{X}.A = \text{abt} \wedge \mathbb{X}.B = (\mathbb{R}', \text{pc}')$
lw_t $r_d, w(r_s)$	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{H}_w(\mathbb{R}(r_s) + w)\}, \text{pc} + 1, \mathbb{X})$	if $\mathbb{R}(r_s) + w \in \text{dom}(\mathbb{X}.\mathbb{H}_w)$;
	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{H}_r(\mathbb{R}(r_s) + w)\}, \text{pc} + 1, \mathbb{X})$	if $\mathbb{R}(r_s) + w \in \text{dom}(\mathbb{X}.\mathbb{H}_r)$;
	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{M}(\mathbb{R}(r_s) + w)\}, \text{pc} + 1, (\mathbb{H}_r\{\mathbb{R}(r_s) + w \rightsquigarrow \mathbb{M}(\mathbb{R}(r_s) + w)\}, \mathbb{H}_w, \mathbb{B}, A))$	if $\mathbb{R}(r_s) + w \in \text{dom}(\mathbb{M})$, where $\mathbb{X} \neq \varepsilon$
sw_t $r_d, w(r_s)$	$(\mathbb{M}, \mathbb{R}, \text{pc} + 1, (\mathbb{H}_r, \mathbb{H}_w\{\mathbb{R}(r_s) + w \rightsquigarrow \mathbb{R}(r_d)\}, \mathbb{B}, A))$	where $\mathbb{X} \neq \varepsilon$
cast $r_d, r_t, w(r_s)$	$(\mathbb{M}\{\mathbb{R}(r_s) + w \rightsquigarrow \mathbb{R}(r_t)\}, \mathbb{R}\{r_t \rightsquigarrow \mathbb{R}(r_d)\}, \text{pc} + 1, \mathbb{X})$	if $\mathbb{R}(r_d) = \mathbb{M}(\mathbb{R}(r_s) + w) \wedge \mathbb{X}.A = \text{act}$;
	$(\mathbb{M}, \mathbb{R}\{r_t \rightsquigarrow \mathbb{M}(\mathbb{R}(r_s) + w)\}, \text{pc} + 1, (\mathbb{H}_r, \mathbb{H}_w, \mathbb{B}, \text{abt}))$	if $\mathbb{R}(r_d) \neq \mathbb{M}(\mathbb{R}(r_s) + w) \wedge \mathbb{X}.A = \text{act}$; where $\mathbb{R}(r_d) + w \in \text{dom}(\mathbb{M})$

Figure 2: Operational Semantics of the Machine

Now it is clear that the big commit operation of our previous work [16] is a macro defined with these low-level instructions. It can be considered in this order: first acquires locks using **cast** to privatize shared memory for no interleaves with others, then validates these speculative reads recorded in the read set using **validate**. If all these passed, it specifies that the atomicity is maintained, then transaction sets the status to **cmt** and rewrite new values buffered in the write set. Finally all owned locks are released by **sw** and the transaction ends. Otherwise if the validation failed it can do nothing but releases the owned locks and rolls back.

We can not obtain the atomicity and isolation properties of transactional code just by the limitation of the syntax and semantics in this model. Transactions require that all the effect of instructions must be discarded before rolling back and the immediate values are invisible to others. All these need a formal verification, which is the emphasis of our another paper on TM. This paper focuses on the invariant proof of shared memory.

The macro of $\text{Npc}_{(\text{pc}, \iota)}$ showed in Figure 3 is a total function, it computes the program counter of next instruction to be executed after the current instruction is completed. It is defined as

if $\iota =$	then $\text{Npc}_{(\text{pc}, \iota)}(\mathbb{M}, \mathbb{R}, \text{pc}, \mathbb{X}) =$
beq r_s, r_t, f	$\begin{cases} f & \text{if } \mathbb{R}(r_s) = \mathbb{R}(r_t) \\ \text{pc} + 1 & \text{if } \mathbb{R}(r_s) \neq \mathbb{R}(r_t) \end{cases}$
bne r_s, r_t, f	$\begin{cases} \text{pc} + 1 & \text{if } \mathbb{R}(r_s) = \mathbb{R}(r_t) \\ f & \text{if } \mathbb{R}(r_s) \neq \mathbb{R}(r_t) \end{cases}$
j f	f
jr r_s	$\mathbb{R}(r_s)$
rollback	$\mathbb{X}.\mathbb{B}.\text{pc}$
Others	$\text{pc} + 1$

Figure 3: Auxiliary NextPC Macro

$\text{pc} + 1$ for arithmetic and data transfer instructions or addresses $f, \mathbb{R}(r_s)$ and $\mathbb{X}.\mathbb{B}.\text{pc}$ for control transfer instructions.

3.2 Concurrent Separation Logic (CSL)

CSL is an extension of the *separation logic*[25] for reasoning about shared memory concurrent programs. Separation logic is an extension of Hoare logic and used to describe memories. Figure 4 shows the assertion language for memory in separation logic, where \mathbf{m} denotes a predicate on memory.

\mathbf{m}	$::=$	$1 \mapsto v \mid \text{emp} \mid \mathbf{m}_1 * \mathbf{m}_2 \mid \mathbf{m}_1 \wedge \mathbf{m}_2 \mid \mathbf{m}_1 \vee \mathbf{m}_2$ $\exists x.\mathbf{m} \mid \forall x.\mathbf{m}$
\mathbf{m}	\in	$\text{Memory} \rightarrow \text{Prop}$
$1 \mapsto v$	$\stackrel{\text{def}}{=}$	$\lambda \mathbb{M}. \mathbb{M} = \{1 \rightsquigarrow v\}$
emp	$\stackrel{\text{def}}{=}$	$\lambda \mathbb{M}. \mathbb{M} = \emptyset$
$\mathbf{m}_1 \wedge \mathbf{m}_2$	$\stackrel{\text{def}}{=}$	$\lambda \mathbb{M}. \mathbf{m}_1 \mathbb{M} \wedge \mathbf{m}_2 \mathbb{M}$
$\mathbf{m}_1 \vee \mathbf{m}_2$	$\stackrel{\text{def}}{=}$	$\lambda \mathbb{M}. \mathbf{m}_1 \mathbb{M} \vee \mathbf{m}_2 \mathbb{M}$
$\exists x.\mathbf{m}$	$\stackrel{\text{def}}{=}$	$\lambda \mathbb{M}. \exists x. \mathbf{m} \mathbb{M}$
$\forall x.\mathbf{m}$	$\stackrel{\text{def}}{=}$	$\lambda \mathbb{M}. \forall x. \mathbf{m} \mathbb{M}$
$\mathbf{m}_1 * \mathbf{m}_2$	$\stackrel{\text{def}}{=}$	$\lambda \mathbb{M}. \exists \mathbb{M}_1, \mathbb{M}_2. \mathbb{M} = \mathbb{M}_1 \uplus \mathbb{M}_2$ $\wedge \mathbf{m}_1 \mathbb{M}_1 \wedge \mathbf{m}_2 \mathbb{M}_2$

Figure 4: Assertion Language for Memory in Separation Logic

Now we describe the semantics of these assertions. $1 \mapsto v$ holds when the memory has only a

single location 1 with the value v . **emp** holds on an empty memory. $\mathbf{m}_1 * \mathbf{m}_2$ holds if the memory can be split into two disjoint parts such that \mathbf{m}_1 holds on one part and \mathbf{m}_2 the other. $\mathbf{m}_1 \wedge \mathbf{m}_2$ holds if both \mathbf{m}_1 and \mathbf{m}_2 hold on the entire memory. $\mathbf{m}_1 \vee \mathbf{m}_2$ holds if either \mathbf{m}_1 or \mathbf{m}_2 holds on the entire memory. $\exists x.\mathbf{m}$ holds if there exists an x that $\mathbf{m}x$ holds on the memory. $\forall x.\mathbf{m}$ holds if for all x that $\mathbf{m}x$ holds on the memory.

In CSL, shared memory is partitioned and each part is protected by a unique lock. For each part of these partitions, an invariant is assigned to specify its well-formedness. The global invariant in our model is an union of invariants of all partitions. When the lock is acquired, the thread takes advantage of mutual-exclusion provided by locks and treats the part of memory as private. Before the lock releasing, it must ensure that the part of memory is well-formed with regard to the corresponding invariant again.

It is hard to reason about transactions just using CSL due to the speculative read. In CSL shared memory accesses must be put in conditional critical region to treat the part of memory as private. However, shared memory read is speculative without memory privatization in transaction. The speculative reads can even read data from memory that have been privatized by other threads. It violates modular certification since it needs to know other threads' private states. Next we borrow the idea from PASL to solve this problem.

3.3 Permission Accounting in Separation Logic (PASL)

In PASL the shared memory defined in Figure 1 is changed to the partial function :

$$\text{Memory} = \text{Address} \rightarrow (\text{Word} \times \text{Permission}).$$

Each cell in the shared memory associates with a permission bit (we simplify the original one and only allow “0” and “1” in our framework). A total permission (“0”) can be split into two read-only permissions (“1”) as needed. The operational semantics based on logic memory model is shown in Figure 6. Here v denotes for *Word* and u denotes *Permission*. In the semantics logic, it requires the total permission to update the shared memory.

Under the new memory model, the new assertion language for memory is presented in Figure 5. Comparing with Figure 4, we only replace $1 \mapsto v$ with $1 \xrightarrow{u} v$ and give its formal semantics. The same part is omit here for short. Next we give some notations: $1 \mapsto v$ is a short form for $1 \xrightarrow{0} v$, and can be split into two disjoint parts with read-only permission ($1 \xrightarrow{1} v$). Two single read-only memory parts with the same location and value can be composed to a single memory location with the total permission.

O’Hearn has shown [17] that separation logic can describe ownership transfer, where concurrent program threads move ownership of memory cells into and out of the shared memory. But in PASL it seems the permission rather than the cell that is transferred between threads

$$\begin{array}{lcl}
m & ::= & 1 \xrightarrow{u} v \mid \dots \\
1 \xrightarrow{u} v & \stackrel{\text{def}}{=} & \lambda M. M = \{1 \rightsquigarrow (v, u)\} \\
& & \vdots \\
1 \xrightarrow{0} v & \iff & 1 \mapsto v \\
1 \xrightarrow{0} v & \iff & 1 \xrightarrow{1} v * 1 \xrightarrow{1} v \\
1 \xrightarrow{u} v * 1 \xrightarrow{u'} v' & = & \begin{cases} 1 \xrightarrow{0} v & \text{if } v = v' \text{ and } u = u' = 1 \\ \text{undef} & \text{otherwise} \end{cases} \\
M_1 \perp M_2 & \stackrel{\text{def}}{=} & \forall l \in \text{dom}(M_1) \cap \text{dom}(M_2). \\
& & M_1(l).v = M_2(l).v \text{ and} \\
& & M_1(l).u = M_2(l).u = 1
\end{array}$$

Figure 5: Assertion Language for Memory with Permission Accounting

in an ownership transfer. In Figure 7 we give an example for explaining the process of the permission transfer.

shared memory	private memory
$\text{mutex} \mapsto 0 * 1 \xrightarrow{0} v$	emp
\Updownarrow	
$\text{mutex} \mapsto 0 * 1 \xrightarrow{1} v * 1 \xrightarrow{1} v$	emp
$\Downarrow \text{lock}(\text{cas}_t)$	
$\text{mutex} \mapsto 1 * 1 \xrightarrow{1} v$	$1 \xrightarrow{1} v$
$\Downarrow \text{unlock}(\text{sw})$	
$\text{mutex} \mapsto 0 * 1 \xrightarrow{1} v * 1 \xrightarrow{1} v$	emp
\Updownarrow	
$\text{mutex} \mapsto 0 * 1 \xrightarrow{0} v$	emp

Figure 7: Permission Transfer on Locking and Unlocking

The shared memory has two cells `mutex` and `1`, and the `mutex` is a storable lock cell which is used for mutually exclusive accessing of cell `1`. Initially the `mutex` is free and cell `1` has the total permission in shared memory while the thread’s private memory is empty. When the thread acquires the lock by `cast`, cell `1` with the total permission splits into two read-only parts and move one part to the thread’s private memory.

if $\iota =$	then $\text{Next}'_{(\text{pc}, \iota)}(\mathbb{M}, \mathbb{R}, \text{pc}, \mathbb{X}) =$	
lw $\mathbf{r}_d, \mathbf{w}(\mathbf{r}_s)$	$(\mathbb{M}, \mathbb{R}\{\mathbf{r}_d \rightsquigarrow \mathbb{M}(\mathbb{R}(\mathbf{r}_s) + \mathbf{w}).\mathbf{v}\}, \text{pc} + 1, \mathbb{X})$	where $\mathbb{R}(\mathbf{r}_s) + \mathbf{w} \in \text{dom}(\mathbb{M})$
sw $\mathbf{r}_d, \mathbf{w}(\mathbf{r}_s)$	$(\mathbb{M}\{\mathbb{R}(\mathbf{r}_s) + \mathbf{w} \rightsquigarrow (\mathbb{R}(\mathbf{r}_d), 0)\}, \mathbb{R}, \text{pc} + 1, \mathbb{X})$	where $\mathbb{M}(\mathbb{R}(\mathbf{r}_s) + \mathbf{w}).\mathbf{u} = 0 \wedge \mathbb{X}.A \neq \text{act}$
lw_t $\mathbf{r}_d, \mathbf{w}(\mathbf{r}_s)$	$(\mathbb{M}, \mathbb{R}\{\mathbf{r}_d \rightsquigarrow \mathbb{H}_w(\mathbb{R}(\mathbf{r}_s) + \mathbf{w}).\mathbf{v}\}, \text{pc} + 1, \mathbb{X})$ $(\mathbb{M}, \mathbb{R}\{\mathbf{r}_d \rightsquigarrow \mathbb{H}_r(\mathbb{R}(\mathbf{r}_s) + \mathbf{w}).\mathbf{v}\}, \text{pc} + 1, \mathbb{X})$ $(\mathbb{M}, \mathbb{R}\{\mathbf{r}_d \rightsquigarrow \mathbb{M}(\mathbb{R}(\mathbf{r}_s) + \mathbf{w}).\mathbf{v}\}, \text{pc} + 1,$ $(\mathbb{H}_r\{\mathbb{R}(\mathbf{r}_s) + \mathbf{w} \rightsquigarrow (\mathbb{M}(\mathbb{R}(\mathbf{r}_s) + \mathbf{w}), 0)\}, \mathbb{H}_w, \mathbb{B}, \mathbb{A}))$	if $\mathbb{R}(\mathbf{r}_s) + \mathbf{w} \in \text{dom}(\mathbb{X}. \mathbb{H}_w)$; if $\mathbb{R}(\mathbf{r}_s) + \mathbf{w} \in \text{dom}(\mathbb{X}. \mathbb{H}_r)$; if $\mathbb{R}(\mathbf{r}_s) + \mathbf{w} \in \text{dom}(\mathbb{M})$, where $\mathbb{X} \neq \varepsilon$
cas_t $\mathbf{r}_d, \mathbf{r}_t, \mathbf{w}(\mathbf{r}_s)$	$(\mathbb{M}\{\mathbb{R}(\mathbf{r}_s) + \mathbf{w} \rightsquigarrow (\mathbb{R}(\mathbf{r}_t), 0)\}, \mathbb{R}\{\mathbf{r}_t \rightsquigarrow \mathbb{R}(\mathbf{r}_d)\}, \text{pc} + 1, \mathbb{X})$ $(\mathbb{M}, \mathbb{R}\{\mathbf{r}_t \rightsquigarrow \mathbb{M}(\mathbb{R}(\mathbf{r}_s) + \mathbf{w}).\mathbf{v}\}, \text{pc} + 1, (\mathbb{H}_r, \mathbb{H}_w, \mathbb{B}, \text{abt}))$	if $\mathbb{R}(\mathbf{r}_d) = \mathbb{M}(\mathbb{R}(\mathbf{r}_s) + \mathbf{w}).\mathbf{v} \wedge \mathbb{X}.A = \text{act}$; if $\mathbb{R}(\mathbf{r}_d) \neq \mathbb{M}(\mathbb{R}(\mathbf{r}_s) + \mathbf{w}).\mathbf{v} \wedge \mathbb{X}.A = \text{act}$; where $\mathbb{M}(\mathbb{R}(\mathbf{r}_d) + \mathbf{w}).\mathbf{u} = 0$
others	$\text{Next}_{(\text{pc}, \iota)}(\mathbb{M}, \mathbb{R}, \text{pc}, \mathbb{X})$	

Figure 6: Operational Semantics with Logic Memory Model

The other part is left in the shared memory for speculative reads. When the lock is released by **sw**, the read-only part in private memory returns to the shared memory and the mutex is set to zero. If the thread attempts to update cell **l** after **cas_t**, it combines the private memory with the shared memory to generate a total permission by two read-only permissions and update both parts.

3.4 Program Specifications

We have defined the memory assertion language before, now the assertion language for thread state is presented in Figure 8. Here **m** is a predicate on memory that we defined in Figure 5.

Most of the definitions are simple and straightforward, here we explain some special ones. ε denotes the \mathbb{X} in the entire thread state is ε . $[\mathbf{m}]$, $[\mathbf{m}]_r$, $[\mathbf{m}]_w$ mean the memory, read set, write set in the entire thread state satisfies **m** respectively. $[\mathbf{r}] = \mathbf{v}$, $[\mathbf{r}]_b = \mathbf{v}$ describe the register **r** in the thread state's, backup state's register file respectively. $[\text{pc}] = \mathbf{v}$, $[\text{pc}]_b = \mathbf{v}$ describe

$$\begin{aligned} \mathbf{a} ::= & \varepsilon \mid [\mathbf{m}] \mid [\mathbf{m}]_r \mid [\mathbf{m}]_w \mid [\mathbf{r}] = \mathbf{v} \mid [\mathbf{r}]_b = \mathbf{v} \\ & \mid [\text{pc}] = \mathbf{v} \mid [\text{pc}]_b = \mathbf{v} \mid \mathbb{A} = \text{act/cmt/abt} \\ & \mid \mathbf{a}_1 \wedge \mathbf{a}_2 \mid \mathbf{a}_1 \vee \mathbf{a}_2 \mid \exists x. \mathbf{a} \mid \forall x. \mathbf{a} \end{aligned}$$

$$\begin{aligned} \mathbf{a} & \in \text{ThreadState} \rightarrow \text{Prop} \\ \varepsilon & \stackrel{\text{def}}{=} \lambda \mathbb{S}. \mathbb{S}. \mathbb{X} = \varepsilon \\ [\mathbf{m}] & \stackrel{\text{def}}{=} \lambda \mathbb{S}. \mathbf{m} \ \mathbb{S}. \mathbb{M} \\ [\mathbf{m}]_r & \stackrel{\text{def}}{=} \lambda \mathbb{S}. \mathbf{m} \ \mathbb{S}. \mathbb{X}. \mathbb{H}_r \\ [\mathbf{m}]_w & \stackrel{\text{def}}{=} \lambda \mathbb{S}. \mathbf{m} \ \mathbb{S}. \mathbb{X}. \mathbb{H}_w \\ [\mathbf{r}] = \mathbf{v} & \stackrel{\text{def}}{=} \lambda \mathbb{S}. \mathbb{S}. \mathbb{R}(\mathbf{r}) = \mathbf{v} \\ [\mathbf{r}]_b = \mathbf{v} & \stackrel{\text{def}}{=} \lambda \mathbb{S}. \mathbb{S}. \mathbb{X}. \mathbb{B}. \mathbb{R}(\mathbf{r}) = \mathbf{v} \\ [\text{pc}]_b = \mathbf{v} & \stackrel{\text{def}}{=} \lambda \mathbb{S}. \mathbb{S}. \mathbb{X}. \mathbb{B}. \text{pc} = \mathbf{v} \\ [\text{pc}] = \mathbf{v} & \stackrel{\text{def}}{=} \lambda \mathbb{S}. \mathbb{S}. \text{pc} = \mathbf{v} \\ \exists x. \mathbf{a} & \stackrel{\text{def}}{=} \lambda \mathbb{S}. \exists x. \mathbf{a} \ \mathbb{S} \\ \forall x. \mathbf{a} & \stackrel{\text{def}}{=} \lambda \mathbb{S}. \forall x. \mathbf{a} \ \mathbb{S} \\ \mathbf{a}_1 \wedge \mathbf{a}_2 & \stackrel{\text{def}}{=} \lambda \mathbb{S}. \mathbf{a}_1 \ \mathbb{S} \wedge \mathbf{a}_2 \ \mathbb{S} \\ \mathbf{a}_1 \vee \mathbf{a}_2 & \stackrel{\text{def}}{=} \lambda \mathbb{S}. \mathbf{a}_1 \ \mathbb{S} \vee \mathbf{a}_2 \ \mathbb{S} \\ \mathbb{A} = \text{act/cmt/abt} & \stackrel{\text{def}}{=} \lambda \mathbb{S}. \mathbb{S}. \mathbb{X}. \mathbb{A} = \text{act/cmt/abt} \end{aligned}$$

Figure 8: Assertion Language for State

the current program counter and the record program counter in the beginning respectively. The last one $\mathbb{A} = \text{act/cmt/abt}$ describe the current transaction status in the entire thread state.

The verification constructs of our logic is presented in Figure 9.

The world specification ϕ contains a global invariant **m** and code heap specifications

$$\begin{array}{ll}
(\text{WorldSpec}) & \phi ::= (\mathbf{m}, [\psi_1, \dots, \psi_n]) \\
(\text{CdHpSpec}) & \psi ::= \{1 \rightsquigarrow \mathbf{a}\}^* \\
\\
(\text{WF_World}) & \phi, [(\mathbf{a}_1, \dots, \mathbf{a}_n)] \vdash \mathbb{W} \\
(\text{WF_Codeheap}) & \psi, \mathbf{m} \vdash \mathbb{C} : \psi' \\
(\text{WF_Inseq}) & \psi, \mathbf{m} \vdash \{\mathbf{a}\} \text{pc} : \mathbb{I}
\end{array}$$

Figure 9: The Verification Constructs for Program Logic

ψ_1, \dots, ψ_n for each thread. The global invariant \mathbf{m} is a programmer specified predicate on shared memory, it must hold throughout the execution of the program by each thread. A code heap specification ψ assigns thread state assertion \mathbf{a} which expresses the precondition to execute to each instruction sequence. The last three are defined judgements for the well-formed world, well-formed code heap and well-formed instruction sequence respectively. Rules for these judgements will be presented in the following subsection.

Besides we provide a special notation for useful auxiliary definitions in Figure 10 for program logic. The first two are syntax sugars for propositions. The last two are syntax sugars for assertion language.

$$\begin{array}{ll}
\mathbf{a} \Rightarrow \mathbf{a}' & \stackrel{\text{def}}{=} \forall \mathbb{S}. \mathbf{a} \mathbb{S} \rightarrow \mathbf{a}' \mathbb{S} \\
\text{Domeq } \mathbb{M} \mathbb{M}' & \stackrel{\text{def}}{=} \forall 1. (1 \in \text{dom}(\mathbb{M}) \wedge 1 \in \text{dom}(\mathbb{M}')) \vee \\
& \quad (1 \notin \text{dom}(\mathbb{M}) \wedge 1 \notin \text{dom}(\mathbb{M}')) \\
\mathbf{a} \triangleleft (\mathbb{R}, \text{pc}, \mathbb{X}) & \stackrel{\text{def}}{=} \lambda \mathbb{M}. \mathbf{a} (\mathbb{M}, \mathbb{R}, \text{pc}, \mathbb{X}) \\
\mathbf{a} \oplus \mathbf{m} & \stackrel{\text{def}}{=} \lambda (\mathbb{M}, \mathbb{R}, \text{pc}, \mathbb{X}). (\mathbf{a} \triangleleft (\mathbb{R}, \text{pc}, \mathbb{X}) * \mathbf{m}) \mathbb{M}
\end{array}$$

Figure 10: Auxiliary Definition for Program Logic

3.5 Inference Rules

The inference rules of our program logic are presented in Figure 12. A world is well-formed with regard to a world specification ϕ and the thread state predicates $\mathbf{a}_1, \dots, \mathbf{a}_n$ for each thread when the following conditions hold:

- There exists a code heap specification ψ_i for each thread and a global invariant \mathbf{m} in the program specification ϕ . For each thread, the code heap is well-formed regarding ψ_i and \mathbf{m} , moreover, the thread state predicate \mathbf{a}_i is satisfied at the point of pc_i .
- There is a $n + 1$ parts partition of the shared memory \mathbb{M} , where \mathbb{M}_s satisfies the global invariant \mathbf{m} and $\mathbb{M}_1, \dots, \mathbb{M}_n$ satisfy each thread state predicate \mathbf{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 ι and another instruction sequence \mathbb{I} and both of them are well-formed (rule INSQ). A well-formed instruction requires the corresponding transaction status that defined in Figure 11 and fall into the following cases with the order of Figure 12:

- Weak memory instruction – Instruction ι can execute for all thread states specified by the current thread state predicate \mathbf{a} , and the new modified thread state must satisfy

if $\iota =$	then $\text{En}(\iota) =$
$\text{sw } r_d, r_s(w)$	$\lambda S. S.X.A \neq \text{act}$
$\text{lw}_t r_d, r_s(w)$	$\lambda S. S.X \neq \varepsilon$
$\text{sw}_t r_d, r_s(w)$	$\lambda S. S.X \neq \varepsilon$
$\text{cas}_t r_d, r_t, r_s(w)$	$\lambda S. S.X.A = \text{act}$
$\text{out } r_d$	$\lambda S. S.X.A = \text{cmt}$
begin	$\lambda S. S.X = \varepsilon$
validate r_d	$\lambda S. S.X.A = \text{act}$
commit	$\lambda S. S.X.A = \text{cmt}$
rollback	$\lambda S. S.X.A = \text{abt}$
Others	True

Figure 11: Enable for Instructions

the thread state predicate for the target address of instruction ι given by ψ .

- Strong memory instruction – Instruction ι can execute for all thread states specified by the current thread state predicate \mathbf{a} and the global invariant \mathbf{m} . Furthermore, the new modified thread state must satisfy the thread state predicate for the target address of instruction ι given by ψ and reestablish the global invariant \mathbf{m} . Note that the domain of the shared memory may be changed after the instruction executes.
- Instruction **commit** – In transactions, all the mutexes that acquired must be released before the ending, neither more nor less. So in rule COMMIT, we check the the domain of private memories at the current thread state with the beginning one to enforce it.

Our program logic in the framework is based on CSL, and it is more powerful than CSL for supporting speculative read. It also well suits for other STM models (such as eager version

$$\boxed{\phi, [\mathbf{a}_1, \dots, \mathbf{a}_n]} \vdash \mathbb{W} \quad (\text{Well-formed world})$$

$$\begin{aligned} \phi &= (\mathbf{m}, [\psi_1, \dots, \psi_n]) \\ \mathbb{M} &= \mathbb{M}_s \uplus \mathbb{M}_1 \uplus \dots \uplus \mathbb{M}_n \\ \mathbf{m} \mathbb{M}_s \quad \mathbf{a}_k(\mathbb{M}_k, \mathbb{R}_k, \mathbf{pc}_k, \mathbb{X}_k) \\ \psi_k, \mathbf{m} \vdash \mathbb{C} : \psi_k \quad \psi_k, \mathbf{m} \vdash \{\mathbf{a}_k\} \mathbf{pc}_k : \mathbb{C}[\mathbf{pc}_k] \quad \text{for all } k \\ \hline \phi, [\mathbf{a}_1, \dots, \mathbf{a}_n] \vdash (\mathbb{C}, \mathbb{M}, [(\mathbb{R}_1, \mathbf{pc}_1, \mathbb{X}_1), \dots, (\mathbb{R}_n, \mathbf{pc}_n, \mathbb{X}_n)]) \end{aligned} \quad (\text{WORLD})$$

$$\boxed{\psi, \mathbf{m} \vdash \mathbb{C} : \psi'} \quad (\text{Well-formed code heap})$$

$$\frac{\forall (\mathbf{pc}, \mathbf{a}) \in \psi' : \quad \psi, \mathbf{m} \vdash \{\mathbf{a}\} \mathbf{pc} : \mathbb{C}[\mathbf{pc}]}{\psi, \mathbf{m} \vdash \mathbb{C} : \psi'} \quad (\text{CDHP})$$

$$\boxed{\psi, \mathbf{m} \vdash \{\mathbf{a}\} \mathbf{pc} : \mathbb{I}} \quad (\text{Well-formed instr. sequences})$$

$$\frac{\psi, \mathbf{m} \vdash \{\mathbf{a}'\} \mathbf{pc} + 1 : \mathbb{I} \quad \psi \uplus \{\mathbf{pc} + 1 \rightsquigarrow \mathbf{a}'\}, \mathbf{m} \vdash \{\mathbf{a}\} \mathbf{pc} : \iota}{\psi, \mathbf{m} \vdash \{\mathbf{a}\} \mathbf{pc} : \iota; \mathbb{I}} \quad (\text{INSQ})$$

$$\frac{\begin{aligned} \iota &\notin \{\text{lw}_t, \text{sw}, \text{cas}_t, \text{commit}\} \\ \mathbf{a} &\Rightarrow \text{En}(\iota) \\ \mathbf{a} \Rightarrow (\lambda S. \psi(\text{Npc}_{(\mathbf{pc}, \iota)} \mathbb{S}) (\text{Next}'_{(\mathbf{pc}, \iota)} \mathbb{S})) \end{aligned}}{\psi, \mathbf{m} \vdash \{\mathbf{a}\} \mathbf{pc} : \iota} \quad (\text{W-INSN})$$

$$\frac{\begin{aligned} \iota &\in \{\text{lw}_t, \text{sw}, \text{cas}_t\} \\ \mathbf{a} \otimes \mathbf{m} &\Rightarrow \text{En}(\iota) \\ \mathbf{a} \otimes \mathbf{m} \Rightarrow (\lambda S. (\psi(\text{Npc}_{(\mathbf{pc}, \iota)} \mathbb{S}) \otimes \mathbf{m}) (\text{Next}'_{(\mathbf{pc}, \iota)} \mathbb{S})) \end{aligned}}{\psi, \mathbf{m} \vdash \{\mathbf{a}\} \mathbf{pc} : \iota} \quad (\text{S-INSN})$$

$$\frac{\begin{aligned} \mathbf{a} \Rightarrow (\lambda S. \psi(\mathbf{pc} + 1) (\text{Next}'_{(\mathbf{pc}, \iota)} \mathbb{S})) \wedge \text{En}(\text{commit}) \\ \forall S, S'. \mathbf{a} S \rightarrow \psi(S.X.B.\mathbf{pc}) S' \rightarrow \text{Domeq } S.M S'.M \end{aligned}}{\psi, \mathbf{m} \vdash \{\mathbf{a}\} \mathbf{pc} : \text{commit}} \quad (\text{COMMIT})$$

Figure 12: Inference Rules

management). Furthermore, it can be applied to TM implementations using read-write lock by extending the mode of permission.

The atomicity and isolation properties of transaction can also be formally verified in our framework. We introduce a local guarantee \mathbf{g} for each thread, as in SCAP [27], describing

valid memory updates – it is safe for the current transaction to roll back only after make a memory update allowed by \mathbf{g} . So when a transaction rolls back, it guarantees that the memory in the current thread state is consistent with the memory at the beginning of the transaction, just as nothing has been done. The isolation is enforced by CSL, in status \mathbf{act} the shared memory is required to be unchanged due to that the transaction may roll back, so only in status \mathbf{cmt} the memory updates can be visible to other threads. The whole progress is presented in detail in our another paper on TM.

3.6 Soundness

The soundness of our framework inference rules with respect to the operational semantics for the machine is established following the syntactic approach of proving type soundness [28]. 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. We define $\mathbb{W} \mapsto^n \mathbb{W}'$ as the relation of n -step ($n \geq 0$) world transitions. The soundness of the framework is formally stated as Theorem 3.3.

Lemma 3.1 (Progress) .

For any $\mathbb{W} = (\mathbb{C}, \mathbb{M}, [\mathbb{T}_1, \dots, \mathbb{T}_n])$, if $\phi, [\mathbf{a}_1, \dots, \mathbf{a}_n] \vdash \mathbb{W}$, then for any thread \mathbb{T}_i ,

there exists $\mathbb{M}', \mathbb{T}'_i$, such that $(\mathbb{M}, \mathbb{T}_i) \hookrightarrow (\mathbb{M}', \mathbb{T}'_i)$.

Lemma 3.2 (Preservation) .

If $\phi, [\mathbf{a}_1, \dots, \mathbf{a}_n] \vdash \mathbb{W}$, and $\mathbb{W} \mapsto \mathbb{W}'$, then exists $\mathbf{a}'_1, \dots, \mathbf{a}'_n$, such that $\phi, [\mathbf{a}'_1, \dots, \mathbf{a}'_n] \vdash \mathbb{W}'$

Theorem 3.3 (Soundness) .

If $\phi, [\mathbf{a}_1, \dots, \mathbf{a}_n] \vdash \mathbb{W}$, then for any $n \geq 0$, there exists a program \mathbb{W}' and $\mathbf{a}'_1, \dots, \mathbf{a}'_n$ such that $\mathbb{W} \mapsto^n \mathbb{W}'$ and $\phi, [\mathbf{a}'_1, \dots, \mathbf{a}'_n] \vdash \mathbb{W}'$.

We have implemented the complete framework [22] including the proofs for these two lemmas and the soundness theorem in the Coq proof assistant so we are confident that the framework is indeed sound.

4 Example

Our framework is a realization of established verification techniques [17, 18] at the assembly level for concurrent programs. In this section, we give an example to demonstrate the mechanized verification of safety properties (usually the shared memory invariant in parallel program) for concurrent assembly code.

A simple example of Fibonacci program is presented in Figure 13, 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 \mathbf{prev} and

`curr`. The variables `prev` and `curr` are shared between threads so it needs synchronization for access. In high-level programming these shared memory accesses are put in an atomic block, it hides the additional operations for transaction. In STM, these operations are packed in APIs as shown in Figure 13.

Atomic Block	STM implement
<pre>int fib() { atomic{ val = curr; curr += prev; prev = val; printf(curr); } }</pre>	<pre>int fib() { stmStart(); val_1 = stmRead(curr); val_2 = stmRead(prev); stmWrite(val_1+val_2, &curr); stmWrite(val_1, &prev); printf(curr); stmCommit(); }</pre>

Figure 13: Fibonacci Program

The assembly code for routine `fib` corresponds to a transaction of the Fibonacci program with STM presented in Figure 14. In the code it defers I/Os till data committing and assigns each location a unique lock for synchronization that relates to the inline synchronization routines in API `stmCommit`. Together with the assembly code, we also present the set of private assertions and the shared memory invariant of the the program Fibonacci for verification in Figure 14.

The shared memory invariant `m` denotes that values in locations `prev` and `curr` are con-joint fibonacci numbers when one of the locks is free (for lazy version management transaction, it requires all locks before updating share memory), but not related when the locks have been acquired (the part of memory has been

$$m \triangleq \exists a, b, c, d. (l_1 \mapsto a * l_2 \mapsto b * \text{prev} \xrightarrow{a} c * \text{curr} \xrightarrow{b} d) \wedge ((a = 0 \vee b = 0) \rightarrow \exists n. c = \text{fib}(n) \wedge d = \text{fib}(n+1)) \wedge a, b = 0/1$$

```
fib :  -{[emp] ∧ ε}
      begin
      -{[emp] ∧ [emp]r ∧ [emp]w ∧ [pc]b = fib ∧ Δ = act}
      lwt   t1, curr(r0)
      lwt   t2, prev(r0)
      addu   t2, t1, t2
      swt   t2, curr(r0)
      swt   t1, prev(r0)
      addiu  t1, r0, 1
      -{[emp] ∧ [t1] = 1 ∧ ∃v, v'. [curr ↦ v * prev ↦ v']r
        ∧ [curr ↦ v + v' * prev ↦ v]w ∧ [pc]b = fib
        ∧ Δ = act}
      // now acquire locks together
      cast   r0, t1, l1(r0)
      bne    r0, t1, rb
      cast   r0, t1, l2(r0)
      bne    r0, t1, ulk1
      -{[∃n. curr1 ↦ fib(n+1) * prev1 ↦ fib(n)]
        ∧ ∃v, v'. [curr ↦ v * prev ↦ v']r
        ∧ [curr ↦ v + v' * prev ↦ v]w
        ∧ [pc]b = fib ∧ Δ = act}
      validate t2
      beq     r0, t2, ulk2
      -{[∃n. [curr1 ↦ fib(n+1) * prev1 ↦ fib(n)] ∧ [emp]r
        ∧ [curr ↦ fib(n+2) * prev ↦ fib(n+1)]w
        ∧ [pc]b = fib ∧ Δ = cmt}
      lwt   t1, curr(r0)
      sw     t1, curr(r0)
      out    t1
      lwt   t2, prev(r0)
      sw     t2, prev(r0)
      -{[∃n. [curr1 ↦ fib(n+2) * prev1 ↦ fib(n+1)]
        ∧ [curr ↦ fib(n+2) * prev ↦ fib(n+1)]w
        ∧ [emp]r ∧ [pc]b = fib ∧ Δ = cmt}
      // now release locks together
      sw     r0, l2(r0)
      sw     r0, l1(r0)
      commit
      -{[emp] ∧ ε}
      j      fib
      -{[∃n. curr1 ↦ fib(n+1) * prev1 ↦ fib(n)] ∧ [emp]r
        ∧ ∃v, v'. curr ↦ v + v' * prev ↦ v]w ∧ [pc]b = fib
        ∧ Δ = abt}
      ulk2 : sw     r0, l2(r0)
            -{[∃n. curr1 ↦ fib(n+1)]
              ∧ ([emp]r ∧ ∃v, v'. curr ↦ v + v' * prev ↦ v]w
                ∨ ∃v, v'. curr ↦ v * prev ↦ v']r ∧ [emp]w)
              ∧ [pc]b = fib ∧ Δ = abt}
      ulk1 : sw     r0, l1(r0)
      rb :  -{[emp] ∧ [pc]b = fib ∧ Δ = abt
            ∧ ([emp]r ∧ [emp]w ∨ ∃v, v'. [curr ↦ v * prev ↦ v']r
              ∧ [curr ↦ v + v' * prev ↦ v]w)}
            rollback
```

Figure 14: Assembly Code with Assertions of Fibonacci

privatized by a thread and it may destroy the relation between `prev` and `curr`). From the rule S-INSN in Figure 12, strong memory in-

instructions add the invariant to the private assertion and reestablish the invariant after execution. At first sight we may conceive that the shared memory is unchanged, however, the invariant \mathbf{m} is unprecise (there exists at least two different memories that both satisfy \mathbf{m}) so the shared memory may be changed from one case of the \mathbf{m} to another one. Here we take the first execution of `cast` in Figure 14 as an example. If the lock `l` is one, the shared memory satisfies $\exists b, c, d. l_1 \mapsto 1 * l_2 \mapsto b * \text{prev}^1 \mapsto c * \text{curr}^b \mapsto d$ (which is precise), then locking failed, the shared memory is unchanged. Otherwise the shared memory satisfies $\exists b, c, d. l_1 \mapsto 0 * l_2 \mapsto b * \text{prev}^0 \mapsto c * \text{curr}^b \mapsto d$, after execution one part is transferred to thread's private memory and the modified shared memory satisfies $\exists b, c, d. l_1 \mapsto 1 * l_2 \mapsto b * \text{prev}^1 \mapsto c * \text{curr}^b \mapsto d$, which is another case of our invariant \mathbf{m} . So the shared memory can change from one case of the invariant to the other one.

All the work above is fully mechanized in the Coq proof assistant [23], including the machine model, the soundness proof and examples. Interested readers can check out the Coq implementation [22] for details.

5 Related Work and Conclusions

There has been a lot of work on the verification of concurrent programs, such as CCAP [20] and CMAP [21]. They extend the CAP framework proposed for assembly code and use the rely

–guarantee method [24, 20] for compositional concurrent program verification. It needs to check the non-interference properties between every two threads. O’Hearn [17] proposed CSL for a high-level parallel language based on the separation logic [25]. It explicitly separates the private and shared memories and uses conditional critical regions (CCR) to permit the ownership transfer. CSL uses invariants to preserve the well-formness of shared memory out of CCR. The CCR is usually implemented using the lock/unlock primitives and each lock corresponds to an invariant. Recently, Brookes [26] provides a grainless semantics to CSL for parallel programs that share mutable states; Bornat *et al.* [18] proposed a refinement of CSL with fine-grained resource accounting. We are inspired from these works and specify a global invariant on the shared memory for the correctness of interaction between threads.

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 [5] proposed the first software implementation handling transactions with statically known read and write sets. Next Herlihy *et al.* [8] built non-blocking STM that run on common hard-

ware and handle transactions with dynamically known read and write sets. It is designed with preemption safety as a major concern. All of these works above explore the various implementation strategies of TM systems to achieve better performance with less expense. Few of them formally reason about the correctness of these implementations and properties of programs.

Moore and Grossman [15] present a type system for spawning new threads in transaction programs and prove the widely-held belief that if each mutable memory location is used outside transactions or inside transactions (but not both) then strong and weak atomicity are indistinguishable. The semantics of transaction is high-level and small-step in a λ -calculus. But the TM system only allows at most one thread to execute a transaction at a time.

Our previous work [16] presents a framework for verifying concurrent programs using transactional memory. It focuses on the verification of invariant over the shared memory. However, the machine model is at a high level which concentrates the whole operations of transaction commit into an instruction, losing sight of various actual implementations and related properties.

In this paper we extend our previous framework by refining the big commit operation. As the previous one, it is a lazy software TM systems with lazy conflict detection and lazy version management. But it is based on the most fundamental instructions and uses

storable locks for memory privatization. In the framework we focus on the verification of shared memory properties. We specify a global invariant on shared memory for each program using this model and use a combination of CSL and PASL for verifying that the global invariant is held during the execution of the program. In future we hope to extend our TM system with nested transactions (open-nested transaction for especial) for concurrent program certifying, and even with the weak atomicity semantics.

Acknowledgments

We would like to thank Prof. Zhong Shao (Yale University) and anonymous for their inspiring discussions and suggestions on this paper. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

- [1] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *SIGARCH Comput. Archit. News*, pages 289–300, 1993.
- [2] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA'04: Proceeding of the 31st Annual International Symposium on*

- Computer architecture*, page 102, Washington, DC, 2004. IEEE Comp. Soc.
- [3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA'05: Proceeding of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, 2005. IEEE Comp. Soc.
- [4] K. E. Moore and D. Grossman. Log-based transactional memory. PhD thesis, Madison, WI, USA, 2007. Adviser-David A. Wood.
- [5] N. Shavit and D. Touitou. Software transactional memroy. In *PODC'95: Proceeding of the fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, New York, 1995. ACM Press.
- [6] T. Harris and K. Fraser. Language support for lightweight transactions In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 388–402, New York, 2003. ACM Press.
- [7] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP'06: Proceeding of the eleventh ACM SIGPLAN Symposium on Principles and Practice of parallel programming*, pages 187–197, New York, 2006. ACM Press.
- [8] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC'03: Proceedings of the twenty-second annual Symposium on Principles of Distributed Computing*, pages 92–101, New York, 2003. ACM Press.
- [9] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II In *Lecture Notes in Computer Science*, pages 194–208, 2006. Springer Berlin.
- [10] P. Felber, C. Fetzer, T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP'08: Proceedings of the 13rd ACM SIGPLAN Symposium on Principles and Practice of parallel programming*, New York, 2008. ACM Press.
- [11] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP'06: Proceedings of the eleventh ACM SIGPLAN Symposium on Principles and Practice of parallel programming*, pages 209–220, New York, 2006. ACM Press.
- [12] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *SIGOPS Oper.Syst.Rev.*, pages 336–346, 2006.
- [13] B. Liblit. An operational semantics for LogTM. Technical Report 1571, University of Wisconsin-Madison, August 2006
- [14] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. In *IEEE Comput. Archit. Lett.*, page 17, 2006
- [15] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *PPoPP'08: Proceedings of the 13rd*

- ACM SIGPLAN Symposium on Principles and Practics of parallel programming*, pages 51–62, New York, 2008. ACM Press.
- [16] L. Li, Y. Zhang, Y. Chen, and Y. Li. Certifying concurrent programs using transactional memory. In *JCST*, 2008.
- [17] P. W. O'Hearn. Resources, coucurency, and local reasoning. *Theor. Comput. Sci.*, pages 271–307, 2007.
- [18] R. Bornat, C. Calcagno, P.O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 259–270, New York, 2005. ACM Press.
- [19] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI'07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 78–88, New York, 2007. ACM Press.
- [20] D. Yu and Z. Shao. Verification of safety properties for concurrent assembly code. In *ICFP'04: Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming*, pages 175–188, Utah, September 2004. ACM Press.
- [21] X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *ICFP'05: Proceedings of the 2005 ACM SIGPLAN International Conference on Functional Programming*, pages 254–267, New York, 2005. ACM Press.
- [22] Y. Li. Coq implementation for formal reasoning about concurrent programs using a lazy-STM system. <http://sug.ustcsz.edu.cn/ccac/papers/frcptm>.
- [23] Coq Development Team. The Coq proof assistant reference manual. Coq release v8.1, October 2006.
- [24] C. B. Jones. Tentative steps toward a development method for interfering programs. In *ACM Tran. Program. Lang. Syst.*, pages 596–619, 1983. ACM Press.
- [25] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, 2002. IEEE Comp. Soc.
- [26] S. Brookes. A grainless semantics for parallel programs with shared mutable data. In *MFPS'06: Proceeding of the 21st Annual Conference on Mathematical Foundational of Programming Semantics*, pages 277–307, Washington, DC, 2006. IEEE Comp. Soc..
- [27] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular Verification of Assembly Code with Stack-Based Control Abstractions. In *PLDI'06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 401–414, June 2006. ACM Press.
- [28] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, pages 38–94, 1994.