Li L, Zhang Y, Chen YY *et al.* Certifying concurrent programs using transactional memory. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 24(1): 110–121 Jan. 2009

Certifying Concurrent Programs Using Transactional Memory

Long Li (李 隆), Yu Zhang (张 昱), Yi-Yun Chen (陈意云), and Yong Li (李 勇)

Department of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China

Software Security Laboratory, Suzhou Institute for Advanced Study, University of Science and Technology of China Suzhou 215123, China

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

Received December 19, 2007; revised June 23, 2008.

Abstract Transactional memory (TM) is a new promising concurrency-control mechanism that can avoid many of the pitfalls of the traditional lock-based techniques. TM systems handle data races between threads automatically so that programmers do not have to reason about the interaction of threads manually. TM provides a programming model that may make the development of multi-threaded programs easier. Much work has been done to explore the various implementation strategies of TM systems and to achieve better performance, but little has been done on how to formally reason about programs using TM and how to make sure that such reasoning is sound. In this paper, we focus on the semantics of transactional memory and present a proof-carrying code (PCC) system for reasoning about programs using TM . We formalize our reasoning with respect to the TM semantics, prove its soundness, and use examples to demonstrate its effectiveness.

Keywords program verification, transactional memory, proof-carrying code, concurrent program safety

1 Introduction

The multicore architectures, which provide infrastructures of better performance, have become popular. To benefit from this growth in performance, programmers are required and forced to develop multi-threaded programs. However, it is of great challenges for programmers to deal with subtle concurrent access to shared memory by multiple threads. Traditionally, programmers use locks to achieve mutual exclusion on concurrent access to shared memory. But lock-based programming is difficult to reason about and may lead to problems such as deadlock. Transactional memory (TM) is proposed as an alternate concurrency-control mechanism to avoid many of the pitfalls of the lock-based one. TM systems provide transactions as a programming structure for programmers and handle data races between threads automatically so that programmers do not have to reason about the interaction of threads manually. TM provides a programming model that may make the development of multi-threaded programs easier.

Much work has been done to explore the various implementation strategies of TM systems in hardware[1−4], in software[5−8], and in hybrid[9,10]. The performance issues of these implementations have also been discussed. And there has also been some work that explores the formalization of the semantics of TM[6,11−15]. But little has been done on how to formally reason about programs using TM and how to make sure that such reasoning is sound. Easy reasoning, one of the primary reasons to introduce transactional memory, is rarely referred to.

Proof-carrying code (PCC)[16] is a Hoare-style reasoning framework and has been well studied to reason about various properties of programs in recent years[17−20]. And there are also extensions for reasoning about multi-threaded programs using locks[21,22]. However, the appearance of programs using TM claims for to be a brand-new system.

In this paper, we present a system to certify concurrent programs using transactional memory. We define an abstract machine with built-in TM at assembly-level and present a program logic based on this machine to support the verification of different properties. We prove the soundness of our program logic with respect to the semantics of transactional memory and use examples to demonstrate its effectiveness. Our paper makes the following contributions.

Regular Paper

Supported by the National Natural Science Foundation of China under Grant Nos. 60673126 and 90718026, and Intel China Research Center.

• We present a system for certifying properties of programs using TM. Even though the programming using transactions is supposed to reason about easily, there are subtle properties that are expected by multi-threaded programs. To ensure such properties are correctly enforced in a concurrent program, a substantial way is to perform a formal verification. Our system is essentially a PCC system that performs such verification.

• We present a program logic for reasoning about properties of programs using TM in our system. And we prove it sound with respect to the TM semantics. TM systems free programmers from manually reasoning about the interaction of threads, but threads do cooperate on shared memory. To verify the properties of a program using TM, we not only verify the properties of each thread, but also emphasize the cooperation that is allowed on shared memory. In our system, we specify a global invariant on shared memory to justify the interleaving of transactions. This global invariant is required to hold all through the program. We establish the consistency for transactions in our system by requiring a transaction to transmit shared memory from one consistent state that respects the global invariant to another. We formalize such reasoning in a PCC system and prove it sound following the syntactic approach to proving type soundness. And we also use examples to demonstrate its effectiveness. Our reasoning reveals the easy reasoning aspect of programs using TM: programmers can reason about their threads in isolation and need only to make sure that their threads do not violate the global invariant when transactions update shared memory.

• Our system addresses the safety issues at assembly-level directly as PCC systems do. So we do not need to trust the complicated compilation and optimization, and can have a smaller trusted computing base to build executable PCC packages for programs using TM. Furthermore, our abstraction of TM systems is still general and similar to high-level ones. The reasoning we describe at assembly-level can be easily lifted up to higher levels.

All our work is mechanized in Coq proof assistant[23] to build machine checkable proofs. And the efforts needed to build proofs for PCC packages are significantly mitigated due to the easy reasoning property enforced by TM semantics.

The rest of this paper is organized as follows. In Section 2, we discuss the semantics and implementations of transactional memory and also the violations in practice. In Section 3, we describe the abstract machine we model and the program logic we use to reason. Then we present examples that are written and proven in our system in Section 4. In Section 5, we discuss the related work and then give the conclusion in Section 6.

2 Transactional Memory

In this section, we discuss the semantics of transactional memory and its implementation strategies. Then we study the violations in implementations and determine the compositions of the abstract machine in our system.

2.1 Semantics and Implementations

Transactional memory systems provide transactions as a programming structure for programmers. A transaction is a sequence of instructions that executes atomically and in isolation. The atomicity refers to the all-or-nothing effect on the memory. If a transaction commits, then all of its modifications to the memory appear to take effect all together and happen instantaneously. And if a transaction aborts (a rollback occurs subsequently), then none of its modifications takes effect, as if the transaction were never executed.

A transaction also runs in isolation. It executes as if it is the only active operation in the system, and no other operations make progress while it is running. This means that the effect of a transaction is not visible to other threads until the transaction commits.

The atomicity and isolation properties together give the illusion that there is no interaction between transactions, and a transaction executes as a single atomic step with respect to the other threads in the system. Thus, programmers can reason about the behavior of a thread without considering the interaction of the other threads, as if it is executed under a sequential circumstance.

The key mechanisms for a TM system to provide the atomicity and isolation properties are data versioning and conflict detection, whose implementations distinguish alternative TM proposals.

Data versioning manages multiple versions of data when transactions are being executed. A new version, produced by one of the not-yet-commit transactions, will become visible to all other threads after the transaction commits. The old version, produced by a previously committed transaction, will be preserved when the transaction aborts. With eager versioning, a write access within a transaction immediately writes to memory the new data and creates an undo log for the old one. Also the write access prevents other threads from accessing the same address until the transaction completes by using locks or equivalent mechanisms. When

the transaction aborts, the undo log is used to restore the old version of data.

Lazy versioning buffers all new data versions until the transaction completes. The new versions will be copied to the actual memory when the transaction commits. If the transaction aborts, the buffer is discarded and no further compensation is needed.

Conflict detection signals the multiple access to the same data with at least one transaction attempting to write a new version. Detection relies on tracking the read set (addresses read from) and write set (addresses written to) for each transaction. Under eager conflict detection, the system checks for conflicts whenever transactions read or write data. While under lazy conflict detection, all checks are delayed until the end of each transaction (i.e., when it tries to commit).

2.2 Violations

TM systems are supposed to enforce the properties of atomicity (all-or-nothing) and isolation (intermediate state not visible to other threads). But practical implementations may violate these properties more or less and may lead to unexpected behaviors of programs using TM as follows.

• *Atomicity Violation.* In practice, the rollback operation of eager versioning and the commit operation of lazy versioning are time-consuming. If the TM system allows other threads to access the data which rollback or commit may modify during their execution, threads may get invalid data (read data ahead of rollback or commit operations) or lose update (write data ahead). The solution is to claim exclusiveness for the data that rollback or commit is about to access before their execution.

• *Isolation Violation.* Threads may see the intermediate state of a transaction when implementations of eager versioning do not claim exclusiveness for the addresses it writes to. Blundell *et al.*[24] discussed this issue and called it weak atomicity.

• *Privatization Issue.* The atomicity and isolation properties of TM systems do not specify whether the access to shared data is allowed to bypass the TM system's mechanisms (outside transactions). If "yes", it introduces the privatization problem[25] we will not address for now. If "no", then TM systems should handle the access to shared data outside transactions too, because they may lead to unexpected behaviors of programs. Shpeisman *et al.*[26] discussed this problem and solved it by adding barriers with respect to TM system for every access to memory outside transactions. These barriers can be treated as featherweight transactions in our opinion.

To focus on the reasoning issue, we model the abstract machine with strict semantics of atomicity and isolation and require that all access to shared data must appear within transactions. The TM system in our system will be a lazy versioning, lazy conflict detection system. Being at the assembly level, the granularity of conflict detection is naturally word-level. And we do not support nested transaction for now since it is much more a performance consideration than the semantics of transaction basis. We leave this to our future work.

3 The System

In this section, we first present an abstract machine with transaction structures and its operational semantics. Then we present a program logic of this machine for reasoning.

3.1 Abstract Machine

Fig.1 defines the abstract machine and the syntax of the assembly language. A program $\mathbb{P}$ consists of a code heap $\mathbb{C}$, a global shared data heap $\mathbb{H}$ (referred to as a "global heap" in the remainder of this paper) and numbers of threads $\mathbb{TS}$ working on the global heap concurrently. The code heap $\mathbb{C}$ maps labels to instructions. The data heap $\mathbb{H}$ maps heap addresses to natural values. And the threads $\mathbb{TS}$ are modeled as a partial mapping from thread id t (natural numbers) to its owner thread $\mathbb{T}$. Each thread $\mathbb{T}$ consists of a local view of the global heap $\mathbb{H}_s$ (referred to as a "local heap" in the remainder of this paper), a write set $\mathbb{H}_w$ for recording the write attempts of its transaction, a register file $\mathbb{R}$, a program

(*Program*)	$\mathbb{P}$	::=	$(\mathbb{C}, \mathbb{H}, \mathbb{TS})$
(*Threads*)	$\mathbb{TS}$	::=	$\{t \rightsquigarrow \mathbb{T}\}^*$
(*Thread*)	$\mathbb{T}$	::=	$(\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}, \mathrm{pc}, \mathbb{L}, \mathbb{B})$
(*BackFile*)	$\mathbb{B}$	::=	$(\mathbb{R}, \mathrm{pc})$
(*CodeHeap*)	$\mathbb{C}$	::=	$\{f \rightsquigarrow \iota\}^*$
(*Heap*)	$\mathbb{H}, \mathbb{H}_s$	::=	$\{l \rightsquigarrow w\}^*$
(*WriteSet*)	$\mathbb{H}_w$	::=	$\{l \rightsquigarrow w\}^*$
(*LogFile*)	$\mathbb{L}$	::=	$\varnothing \mid (l, w) :: \mathbb{L}$
(*RegFile*)	$\mathbb{R}$	::=	$\{r \rightsquigarrow w\}^*$
(*Reg*)	r	::=	$\{\mathrm{r}k\}^{k \in \{0,\ldots,31\}}$
(*Labels*)	t, f, l, pc	::=	n (*nat nums*)
(*WordVal*)	w	::=	n (*nat nums*)
(*Instr*)	ι	::=	$\mathrm{addu}\, r_d, r_s, r_t \mid \mathrm{addiu}\, r_d, r_s, w$
		|	$\mathrm{subu}\, r_d, r_s, r_t \mid \mathrm{sltu}\, r_d, r_s, r_t$
		|	$\mathrm{andi}\, r_d, r_s, 1 \mid \mathrm{lw}\, r_d, w(r_s)$
		|	$\mathrm{sw}\, r_s, w(r_d) \mid \mathrm{beq}\, r_s, r_t, f$
		|	$\mathrm{bne}\, r_s, r_t, f \mid \mathrm{j}\, f \mid \mathrm{jr}\, r_s$
		|	starttrans | commit
(*InstrSeq*)	$\mathbb{I}$	::=	$\iota \mid \iota; \mathbb{I}$

Fig.1. Machine syntax.

$$(\mathbb{C}, \mathbb{H}, [(t_1 \rightsquigarrow \mathbb{T}_1), \ldots, (t_n \rightsquigarrow \mathbb{T}_n)]) \longmapsto (\mathbb{C}, \mathbb{H}', [(t_1 \rightsquigarrow \mathbb{T}_1), \ldots, (t_{k-1} \rightsquigarrow \mathbb{T}_{k-1}), (t_k \rightsquigarrow \mathbb{T}'_k), (t_{k+1} \rightsquigarrow \mathbb{T}_{k+1}), \ldots, (t_n \rightsquigarrow \mathbb{T}_n)])$$

if $(\mathbb{C}, \mathbb{H}, \mathbb{T}_k) \overset{k}{\longmapsto} (\mathbb{C}, \mathbb{H}', \mathbb{T}'_k)$ for any k

where

$$(\mathbb{C}, \mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}, \mathrm{pc}, \mathbb{L}, \mathbb{B})) \overset{k}{\longmapsto} (\mathbb{C}, \mathbb{H}', \mathbb{T}')$$

if $\mathbb{C}[\mathrm{pc}] =$	then $(\mathbb{H}', \mathbb{T}') =$	
addu r_d, r_s, r_t	$(\mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}\{r_d \rightsquigarrow \mathbb{R}(r_s) + \mathbb{R}(r_t)\}, \mathrm{pc}{+}1, \mathbb{L}, \mathbb{B}))$	
addiu r_d, r_s, w	$(\mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}\{r_d \rightsquigarrow \mathbb{R}(r_s) + w\}, \mathrm{pc}{+}1, \mathbb{L}, \mathbb{B}))$	
subu r_d, r_s, r_t	$(\mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}\{r_d \rightsquigarrow \mathbb{R}(r_s) - \mathbb{R}(r_t)\}, \mathrm{pc}{+}1, \mathbb{L}, \mathbb{B}))$	
sltu r_d, r_s, r_t	$(\mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}\{r_d \rightsquigarrow i\}, \mathrm{pc}{+}1, \mathbb{L}, \mathbb{B}))$	if $\mathbb{R}(r_s) < \mathbb{R}(r_t)$, $i = 1$, else $i = 0$
andi $r_d, r_s, 1$	$(\mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}\{r_d \rightsquigarrow i\}, \mathrm{pc}{+}1, \mathbb{L}, \mathbb{B}))$	if $\mathbb{R}(r_s)$ is odd, $i = 1$, else $i = 0$
lw $r_d, w(r_s)$	$(\mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}\{r_d \rightsquigarrow \mathbb{H}_s(l)\}, \mathrm{pc}{+}1, (\mathbb{L} :: (l, \mathbb{H}_s(l))), \mathbb{B}))$	if $l \notin dom(\mathbb{H}_w)$
	$(\mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}\{r_d \rightsquigarrow \mathbb{H}_s(l)\}, \mathrm{pc}{+}1, \mathbb{L}, \mathbb{B}))$	if $l \in dom(\mathbb{H}_w)$
		where $l = \mathbb{R}(r_s) + w \wedge l \in dom(\mathbb{H}_s)$
sw $r_s, w(r_d)$	$(\mathbb{H}, ((\mathbb{H}_s\{l \rightsquigarrow \mathbb{R}(r_s)\}), (\mathbb{H}_w \uplus \{l \rightsquigarrow \mathbb{R}(r_s)\}), \mathbb{R}, \mathrm{pc}{+}1, \mathbb{L}, \mathbb{B}))$	if $l = \mathbb{R}(r_d) + w \wedge l \in dom(\mathbb{H}_s)$
beq r_s, r_t, f	$(\mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}, f, \mathbb{L}, \mathbb{B}))$	if $\mathbb{R}(r_s) = \mathbb{R}(r_t)$
	$(\mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}, \mathrm{pc}{+}1, \mathbb{L}, \mathbb{B}))$	otherwise
bne r_s, r_t, f	$(\mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}, f, \mathbb{L}, \mathbb{B}))$	if $\mathbb{R}(r_s) \neq \mathbb{R}(r_t)$
	$(\mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}, \mathrm{pc}{+}1, \mathbb{L}, \mathbb{B}))$	otherwise
j f	$(\mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}, f, \mathbb{L}, \mathbb{B}))$	
jr r_s	$(\mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}, \mathbb{R}(r_s), \mathbb{L}, \mathbb{B}))$	
starttrans	$(\mathbb{H}, (\mathbb{H}, \varnothing, \mathbb{R}, \mathrm{pc}{+}1, \varnothing, (\mathbb{R}, \mathrm{pc})))$	
commit	$((\mathbb{H} \uplus \mathbb{H}_w), (\varnothing, \varnothing, \mathbb{R}, \mathrm{pc}{+}1, \varnothing, \mathbb{B}))$	if $\forall\ (l, w) \in \mathbb{L},\ \mathbb{H}(l) = w$
	$(\mathbb{H}, (\varnothing, \varnothing, \mathbb{B}.\mathbb{R}, \mathbb{B}.\mathrm{pc}, \varnothing, \mathbb{B}))$	otherwise

Fig.2. Operational semantics.

counter pc, a log file $\mathbb{L}$ for recording the read attempts of its transaction, and a back-up file $\mathbb{B}$ for restoring the state of a transaction when it rolls back. Here we ignore the fact that a thread may have a private data heap of its own, because the private data heap does not consist of shared data and it can be treated as an extension of the register file.

The register file $\mathbb{R}$ maps registers to natural values. The log file $\mathbb{L}$ records the heap address and the value it stores for every read when executing a transaction. And the back-up file $\mathbb{B}$ records the state of the register file $\mathbb{R}$ and the program counter pc when a transaction starts. The set of instructions we present here are the commonly used subset in RISC machines with additional instructions marking the start (`starttrans`) and the end (`commit`) of a transaction.

The step function ($\longmapsto$) of a program $\mathbb{P}$ is defined in Fig.2. The auxiliary function $(\mathbb{C}, \mathbb{H}, \mathbb{T}_k) \overset{k}{\longmapsto} (\mathbb{C}, \mathbb{H}', \mathbb{T}'_k)$ is used to define the effects of the execution of a thread $\mathbb{T}_k$. Here we follow the preemptive thread model where execution of threads can be preempted at any program point. This thread model is enforced by allowing any thread (id t_k) to execute at any program point in the definition of the step function. Thus, we define an interleaving model for multi-threaded machines where the instructions of individual thread execute with sequential consistency.

The operational semantics for most instructions are quite straightforward. We focus on the transaction related instructions here. The `starttrans` instruction signals the start of a transaction. It initializes the write set $\mathbb{H}_w$ and the log file $\mathbb{L}$ to empty and saves current register file $\mathbb{R}$ and program counter pc to the back-up file $\mathbb{B}$ to recover the state of the transaction when it rolls back. It also copies the global heap $\mathbb{H}$ to its local copy $\mathbb{H}_s$ to act as its view to the global heap when executing inside a transaction. The `commit` instruction checks whether the log file $\mathbb{L}$ is consistent with current global heap $\mathbb{H}$. If "yes", the transaction is allowed to commit its effects $\mathbb{H}_w$ to the global heap $\mathbb{H}$. If "no", register file and program counter are restored from the back-up file $\mathbb{B}$ to restart the transaction. For both cases, the local

heap $\mathbb{H}_s$, write set $\mathbb{H}_w$ and log file $\mathbb{L}$ are set empty.

The `lw` and `sw` instructions operate on local heap $\mathbb{H}_s$ only. Thus all the access attempts to the global heap $\mathbb{H}$ are required to operate on the local copy $\mathbb{H}_s$ first and take effect on the global heap $\mathbb{H}$ until the corresponding transaction commits. So the non-transactional access to shared memory $\mathbb{H}$ is denied. The `lw` instruction reads data and appends the log file $\mathbb{L}$ when the address it reads from is not in the domain of the write set $\mathbb{H}_w$. The `sw` instruction updates the local heap $\mathbb{H}_s$ directly and performs a unique merge on the write set $\mathbb{H}_w$ recording its write attempt to the global heap $\mathbb{H}$.

The function $\uplus$ defines a unique merge on heaps as follows:

$$\forall\, l.\ (\mathbb{H}_a \uplus \mathbb{H}_b)\ (l) = \begin{cases} \mathbb{H}_b\ (l), & l\ \in \mathbb{H}_b; \\ \mathbb{H}_a\ (l), & l\ \notin \mathbb{H}_b. \end{cases}$$

The abstract machine records the values but not the versions of the global heap in logs as what Herlihy *et al.*[7] have done and identifies the consistency of the log file $\mathbb{L}$ by checking whether all logged tuples (l, w) reflect the mapping relation in $\mathbb{H}$ ($\forall\ (l, w) \in \mathbb{L},\ \mathbb{H}(l) = w$). The implementation strategies of data versioning and conflict detection may vary in different TM systems. However, verification should check the consistency of both branches no matter what mechanisms cause the control flow to branch. So we implement a relatively simple mechanism in the abstract machine to remove the nondeterminacy of the system.

As shown in Fig.2, all instructions operate on local data structures of threads except `commit`. So the effects (on the global heap) of a transaction are not visible to other threads until the transaction commits. The isolation property is properly established. Also the update operation on the global heap and the rollback operation on local structures are defined to finish in a single step, we achieve atomicity by definitions.

The abstract machine we present here may have poor performance due to redundant rollbacks: successful commit of other threads will always cause a transaction to roll back, if the transaction has a read from the address that is just updated, no matter it happens before (conflict occurs) or after that commit (should be fine). But it defines a semantics of TM systems strictly and works fine with reasoning as we show later.

3.2 Program Specifications

In our system, transactions are identified as atomic operations that modify the global heap with respect to certain invariant. This invariant may be violated during the execution of a transaction, but will be reestablished when the transaction completes. Since the intermediate state of a transaction is not visible to the other threads, the violation of the invariant will not be observed by other threads, the global heap will always be in a consistent state that respects the invariant. To enforce such reasoning, we track the state of thread-local data structures during the execution of a thread to ensure that the desired properties are correctly enforced. And we also make sure that the update operation on the global heap does not violate the invariant on the global heap.

Fig.3 defines the specification constructs in our system for such reasoning. A program specification Φ consists of a global invariant *Inv* and code heap specifications Ψ for threads. The global invariant *Inv* specifies the restrictions on the global heap $\mathbb{H}$ when multiple threads are working on it concurrently. The code heap specification Ψ maps labels to preconditions. The global invariant *Inv* and the precondition p for an instruction together form the expectations at that program point.

$$\begin{array}{rrcl} (State) & \mathbb{S} & ::= & (\mathbb{H}, \mathbb{H}_s, \mathbb{H}_w, \mathbb{R}, \mathrm{pc}, \mathbb{L}, \mathbb{B}) \\ (Pstate) & \mathbb{S}_p & ::= & (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}, \mathrm{pc}, \mathbb{B}) \\ (Spred) & p & \in & Pstate \to Prop \\ (Invariant) & Inv & \in & Heap \to Prop \\ & & & \\ (ProgSpec) & \Phi & ::= & (Inv, [\Psi_1, \ldots, \Psi_n]) \\ (CdHpSpec) & \Psi & ::= & \{f \rightsquigarrow p\}^* \\ (Coerc) & [\![\mathbb{S}]\!] & ::= & (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}, \mathrm{pc}, \mathbb{B}) \end{array}$$

Fig.3. Specification syntax.

The validity of a precondition depends on the local data structures only. Note that the log file $\mathbb{L}$ is excluded, because programmers will not observe the transitions of the log file and the well-formedness of a `commit` instruction relies on the well-formedness of both branches, the state of the log file is not vital.

3.3 Inference Rules

The inference rules for a program are shown in Fig.4, and the rules for instructions are shown in Fig.5. We use the following judgement forms to define the inference rules:

$$\begin{array}{ll} \Phi, [p_1, \ldots, p_n] \vdash \mathbb{P} & \text{(well-formed program)}, \\ \Psi, Inv \vdash \{p\}\ (\mathbb{C}, \mathbb{H}, \mathbb{T}) & \text{(well-formed thread)}, \\ \Psi, Inv \vdash \mathbb{C} : \Psi' & \text{(well-formed code heap)}, \\ \Psi, Inv \vdash \{p\}\ f : \mathbb{I} & \text{(well-formed instr. sequence)}. \end{array}$$

A program is well-formed if the global heap respects the invariant on it and every thread of the program is

well-formed. Thus the verification of a multi-threaded program can be decomposed into the verification of its component threads. The reasoning is thread-modular in our system.

$$\boxed{\Phi, [p_1, \ldots, p_n] \vdash \mathbb{P}} \text{ (Well-formed program)}$$

$$\Phi = (Inv, [\Psi_1, \ldots, \Psi_n])$$

$$\frac{Inv\ \mathbb{H} \quad \forall\ k.\ \Psi_k, Inv \vdash \{p_k\}\ (\mathbb{C}, \mathbb{H}, \mathbb{TS}(t_k))}{\Phi, [p_1, \ldots, p_n] \vdash (\mathbb{C}, \mathbb{H}, \mathbb{TS})} \quad (\textsc{prog})$$

$$\boxed{\Psi, Inv \vdash \{p\}\ (\mathbb{C}, \mathbb{H}, \mathbb{T})} \text{ (Well-formed thread)}$$

$$Inv\ \mathbb{H} \quad p\ (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}, f, \mathbb{B})$$

$$\frac{\Psi, Inv \vdash \mathbb{C} : \Psi \quad \Psi, Inv \vdash \{p\}\ f : \mathbb{C}[f]}{\Psi, Inv \vdash \{p\}\ (\mathbb{C}, \mathbb{H}, (\mathbb{H}_s, \mathbb{H}_w, \mathbb{R}, f, \mathbb{L}, \mathbb{B}))} \quad (\textsc{thrd})$$

$$\boxed{\Psi, Inv \vdash \mathbb{C} : \Psi'} \text{ (Well-formed code heap)}$$

$$\frac{\forall\ f \in dom(\Psi') : \quad \Psi, Inv \vdash \{\Psi'(f)\}\ f : \mathbb{C}[f]}{\Psi, Inv \vdash \mathbb{C} : \Psi'} \quad (\textsc{cdhp})$$

Fig.4. Inference rules.

$$\boxed{\Psi, Inv \vdash \{p\}\ f : \mathbb{I}} \text{ (Well-formed instruction sequence)}$$

$$\frac{\begin{array}{c}\Psi(\mathbb{B}.\mathrm{pc}) = p'' \quad \Psi, Inv \vdash \{p'\}\ f{+}1 : \mathbb{I} \\ \forall\ \mathbb{S}.\ Inv\ \mathbb{S}.\mathbb{H} \wedge\ p\ [\![\mathbb{S}]\!] \rightarrow \forall\ (l, w) \in \mathbb{S}.\mathbb{L}, \mathbb{S}.\mathbb{H}(l) = w \\ \rightarrow Inv\ (\mathrm{Step}\ (\mathbb{S})).\mathbb{H} \wedge\ p'\ [\![\mathrm{Step}\ (\mathbb{S})]\!] \\ \forall\ \mathbb{S}.\ Inv\ \mathbb{S}.\mathbb{H} \wedge\ p\ [\![\mathbb{S}]\!] \rightarrow \neg\forall\ (l, w) \in \mathbb{S}.\mathbb{L}, \mathbb{S}.\mathbb{H}(l) = w \\ \rightarrow Inv\ (\mathrm{Step}\ (\mathbb{S})).\mathbb{H} \wedge\ p''\ [\![\mathrm{Step}\ (\mathbb{S})]\!]\end{array}}{\Psi, Inv \vdash \{p\}\ f : \mathrm{commit}; \mathbb{I}} \quad (\textsc{commit})$$

$$\frac{\begin{array}{c}\Psi(f') = p' \\ \forall\ \mathbb{S}.\ Inv\ \mathbb{S}.\mathbb{H} \wedge\ p\ [\![\mathbb{S}]\!] \rightarrow \\ Inv\ (\mathrm{Step}\ (\mathbb{S})).\mathbb{H} \wedge\ p'\ [\![\mathrm{Step}\ (\mathbb{S})]\!]\end{array}}{\Psi, Inv \vdash \{p\}\ f : \mathrm{j}\ f'} \quad (\textsc{j})$$

$$\frac{\begin{array}{c}\Psi(\mathbb{R}(r_s)) = p' \\ \forall\ \mathbb{S}.\ Inv\ \mathbb{S}.\mathbb{H} \wedge\ p\ [\![\mathbb{S}]\!] \rightarrow \\ Inv\ (\mathrm{Step}\ (\mathbb{S})).\mathbb{H} \wedge\ p'\ [\![\mathrm{Step}\ (\mathbb{S})]\!]\end{array}}{\Psi, Inv \vdash \{p\}\ f : \mathrm{jr}\ r_s} \quad (\textsc{jr})$$

$$\frac{\begin{array}{c}\Psi(f') = p'' \quad \Psi, Inv \vdash \{p'\}\ f{+}1 : \mathbb{I} \\ \forall\ \mathbb{S}.\ Inv\ \mathbb{S}.\mathbb{H} \wedge\ p\ [\![\mathbb{S}]\!] \rightarrow \mathbb{S}.\mathbb{R}(r_s) = \mathbb{S}.\mathbb{R}(r_t) \\ \rightarrow Inv\ (\mathrm{Step}\ (\mathbb{S})).\mathbb{H} \wedge\ p''\ [\![\mathrm{Step}\ (\mathbb{S})]\!] \\ \forall\ \mathbb{S}.\ Inv\ \mathbb{S}.\mathbb{H} \wedge\ p\ [\![\mathbb{S}]\!] \rightarrow \mathbb{S}.\mathbb{R}(r_s) \neq \mathbb{S}.\mathbb{R}(r_t) \\ \rightarrow Inv\ (\mathrm{Step}\ (\mathbb{S})).\mathbb{H} \wedge\ p'\ [\![\mathrm{Step}\ (\mathbb{S})]\!]\end{array}}{\Psi, Inv \vdash \{p\}\ f : \mathrm{beq}\ r_s, r_t, f'; \mathbb{I}} \quad (\textsc{beq})$$

$$\iota \in \{\mathrm{addu},\ \mathrm{addiu},\ \mathrm{subu},\ \mathrm{sltu},\ \mathrm{andi},\ \mathrm{lw},\ \mathrm{sw},\ \mathrm{starttrans}\}$$

$$\frac{\begin{array}{c}\Psi, Inv \vdash \{p'\}\ f{+}1 : \mathbb{I} \\ \forall\ \mathbb{S}.\ Inv\ \mathbb{S}.\mathbb{H} \wedge\ p\ [\![\mathbb{S}]\!] \rightarrow \\ Inv\ (\mathrm{Step}\ (\mathbb{S})).\mathbb{H} \wedge\ p'\ [\![\mathrm{Step}\ (\mathbb{S})]\!]\end{array}}{\Psi, Inv \vdash \{p\}\ f : \iota; \mathbb{I}} \quad (\textsc{seq})$$

Fig.5. Inference rules cont'.

A thread is well-formed if the precondition of the thread is satisfied and the invariant on the global heap holds. Also the code heap and the instruction sequence the thread is going to execute are required to be well-formed.

And a code heap is well-formed only if every instruction sequence of it is well-formed.

Most inference rules for instruction sequence are similar and grouped in the `SEQ` rule. It requires that the precondition together with the global invariant ensures the safe execution of the instruction; and that the resulting state respects the global invariant and satisfies the postcondition which also serves as the precondition of the remaining instruction sequence. Note that the `STARTTRANS` rule is also grouped in `SEQ` rule. Even though the instruction copies the global heap to its local heap, it does not need special treatment for its behavior.

The `BNE` and `BEQ` rules are similar and we present one here. Also they do not make much difference from the `SEQ` rule except that they check both branches where the control flow will switch depending on the equivalence of the values that corresponding registers store.

The `J` and `JR` rules require that it is safe to make a jump to the target address and the global invariant is preserved after jumping.

The `COMMIT` rule is similar to the `BEQ` rule except that it branches due to the consistence of the log file. It checks that both successful commit case and rollback case are well-formed. And the requirement that the resulting state respects the global invariant after successful commit actually claims for the checking of the update made by the successfully committed transaction on the global heap $\mathbb{H}$. Furthermore, it is easy to show that rollback is always allowed. A rollback is caused by an update on the global heap. Since we will restore local data structures of transactions, the difference of start and rollback state of a transaction is the global heap only. But all updates on the global heap are required to respect the global invariant, and the precondition relies on local data structures only. So it is always safe to roll back a transaction.

As shown in the inference rules, the invariant on the global heap is required to preserve for every instruction rule, so the invariant on the global heap is guaranteed all through the program. And the transitions of a thread are tracked in the preconditions of its instructions.

3.4 Soundness

The soundness of these inference rules with respect to the operational semantics of the abstract ma-

chine is established following the syntactic approach to proving type soundness. From the "progress" and "preservation" lemmas, we can guarantee that given a well-formed program under compatible preconditions, 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. The soundness of our system is formally stated as Theorem 1.

Lemma 1 (Progress). *$\Phi = (Inv, [\Psi_1, \ldots, \Psi_n])$. If there exist $p_1, \ldots, p_n$, such that $\Phi, [p_1, \ldots, p_n] \vdash \mathbb{P}$, then there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto \mathbb{P}'$.*

Lemma 2 (Preservation). *$\Phi = (Inv, [\Psi_1, \ldots, \Psi_n])$. If $\Phi, [p_1, \ldots, p_n] \vdash \mathbb{P}$ and $\mathbb{P} \longmapsto \mathbb{P}'$, then there exist $p'_1, \ldots, p'_n$ such that $\Phi, [p'_1, \ldots, p'_n] \vdash \mathbb{P}'$.*

Theorem 1 (Soundness). *$\Phi = (Inv, [\Psi_1, \ldots, \Psi_n])$. If there exist $p_1, \ldots, p_n$, such that $\Phi, [p_1, \ldots, p_n] \vdash \mathbb{P}$, then for any $n \geqslant 0$, there exist a program $\mathbb{P}'$ and $p'_1, \ldots, p'_n$ such that $\mathbb{P} \longmapsto^n \mathbb{P}'$ and $\Phi, [p'_1, \ldots, p'_n] \vdash \mathbb{P}'$.*

3.5 Separation Logic

We describe heaps using separation logic[27]. Except for implication, separation logic is similar to linear logic, where register and heap bindings are treated as intuitionistic and linear resources, respectively, extended with an axiom restricting each heap address to a single binding. We write A and B for heap predicates. We write $\mathbb{H} \Vdash A$ if the heap predicate A holds on heap $\mathbb{H}$. The syntax for the fragment of separation logic we use is given in Fig.6.

$$A, B \ ::= \ n \mapsto m \mid \texttt{emp} \mid A * B \mid A \wedge B \mid A \vee B \mid \exists\, x : P.B \mid \forall\, x \in S.A$$

Fig.6. Separation logic.

Now we describe these predicates informally. $n \mapsto m$ holds if the heap consists entirely of the binding of n to m. `emp` holds only on the empty heap. $A * B$ holds if the heap can be split into two disjoint parts such that A holds on one and B on the other. $A \wedge B$ holds if both A and B hold on the entire heap. $A \vee B$ holds if either A or B holds on the heap. $\exists\, x : P.B$ holds if there exists an x of type P such that Bx holds on the heap. P will be omitted when it is clear from the context. $\forall\, x \in S.A$ holds on a heap that satisfies Ax for all x in the finite set S.

4 Examples

In our system, not only the invariant on each thread of a program but also the invariant on the global heap which multiple threads cooperate on are checked. The expressiveness and the application to high-level programs of such invariance proof method have been well-known. It is the atomicity and isolation properties of transactional memory that make the requirement of invariant on shared memory reasonable.

In this section, we use examples to demonstrate the effectiveness of our reasoning system. And we present examples in high-level programs and their assembly counterparts, showing that reasoning about programs using TM in assembly level is essentially the same as in high-level under such reasoning.

4.1 Dining Philosopher Algorithm

A simplified example of dining philosophers in transactional program style is shown in Fig.7, where two philosophers (Thread_1 and Thread_2) share two forks (represented by memory locations $\texttt{fork}_1$ and $\texttt{fork}_2$). A philosopher picks up a fork by writing the thread id (1 or 2) into the memory location representing the fork, and puts down a fork by writing 0.

```
Variables:                          Invariant:
  nat fork1, fork2;                   fork1 = fork2

Thread1 :                           Thread2 :
while (TRUE) {                      while (TRUE) {
  starttrans;                         starttrans;
  if (! fork1){                       if (! fork2){
    fork1 = 1;                          fork2 = 2;
    fork2 = 1;                          fork1 = 2;
    commit;                             commit;
  }                                   }
  else {                              else {
    commit;                             commit;
    continue;                           continue;
  }                                   }
  // eat                              // eat
  starttrans;                         starttrans;
  fork1 = 0;                          fork1 = 0;
  fork2 = 0;                          fork2 = 0;
  commit;                             commit;
  // think                            // think
}                                   }
```

Fig.7. Simplified example of dining philosopher algorithm.

The invariant on the shared memory can be expressed as $(\texttt{fork}_1 = \texttt{fork}_2)$, indicating that both forks are free (when they are all equal to 0) or a philosopher is holding them both. In the more general case of three or more philosophers, the invariant on the shared memory can be expressed as $(\forall\, i.\texttt{fork}_i = 0 \vee \texttt{fork}_i = \texttt{fork}_{i+1} =$

$i \vee \texttt{fork}_i = \texttt{fork}_{i-1} = i-1$), indicating that the i-th philosopher is not holding forks or he is holding both forks he need or his fork is being held by others, where forks are free or picked in pairs and a philosopher is trying to pick his own fork and the fork of the philosopher next to him always. Note that we cannot ensure fairness in our system. Verification of such liveness properties is part of our future work.

$$Inv\ \ \mathbb{H} \triangleq \exists\ v.\ \ \mathbb{H} \Vdash \texttt{fork}_1 \mapsto v * \texttt{fork}_2 \mapsto v$$

L1: $-\{\text{pc} = \texttt{L1}\}$
  `starttrans`
  $-\{\exists\ v. \mathbb{H}_s \Vdash \texttt{fork}_1 \mapsto v * \texttt{fork}_2 \mapsto v$
    $\wedge \mathbb{H}_w \Vdash \texttt{emp} \wedge \mathbb{B}.\text{pc} = \texttt{L1}\}$
  `lw t1 fork1 zero`
  $-\{\exists\ v. \mathbb{H}_s \Vdash \texttt{fork}_1 \mapsto v * \texttt{fork}_2 \mapsto v$
    $\wedge \mathbb{H}_w \Vdash \texttt{emp} \wedge \mathbb{B}.\text{pc} = \texttt{L1} \wedge \mathbb{R}(\texttt{t1}) = \mathbb{H}_s(\texttt{fork}_1)\}$
  `bne t1 zero L3`
  $-\{\mathbb{H}_s \Vdash \texttt{fork}_1 \mapsto 0 * \texttt{fork}_2 \mapsto 0$
    $\wedge \mathbb{H}_w \Vdash \texttt{emp} \wedge \mathbb{B}.\text{pc} = \texttt{L1}\}$
  `addiu t2 zero 1`
  $-\{\mathbb{H}_s \Vdash \texttt{fork}_1 \mapsto 0 * \texttt{fork}_2 \mapsto 0$
    $\wedge \mathbb{H}_w \Vdash \texttt{emp} \wedge \mathbb{B}.\text{pc} = \texttt{L1} \wedge \mathbb{R}(\texttt{t2}) = 1\}$
  `sw t2 fork1 zero`
  $-\{\mathbb{H}_s \Vdash \texttt{fork}_1 \mapsto 1 * \texttt{fork}_2 \mapsto 0$
    $\wedge \mathbb{H}_w \Vdash \texttt{fork}_1 \mapsto 1 \wedge \mathbb{B}.\text{pc} = \texttt{L1} \wedge \mathbb{R}(\texttt{t2}) = 1\}$
  `sw t2 fork2 zero`
  $-\{\mathbb{H}_s \Vdash \texttt{fork}_1 \mapsto 1 * \texttt{fork}_2 \mapsto 1$
    $\wedge \mathbb{H}_w \Vdash \texttt{fork}_1 \mapsto 1 * \texttt{fork}_2 \mapsto 1 \wedge \mathbb{B}.\text{pc} = \texttt{L1}\}$
  `commit`
  $-\{\texttt{true}\}$
  `j L2`

L2: $-\{\text{pc} = \texttt{L2}\}$
  `starttrans`
  $-\{\exists\ v. \mathbb{H}_s \Vdash \texttt{fork}_1 \mapsto v * \texttt{fork}_2 \mapsto v$
    $\wedge \mathbb{H}_w \Vdash \texttt{emp} \wedge \mathbb{B}.\text{pc} = \texttt{L2}\}$
  `addiu t3 zero 0`
  $-\{\exists\ v. \mathbb{H}_s \Vdash \texttt{fork}_1 \mapsto v * \texttt{fork}_2 \mapsto v$
    $\wedge \mathbb{H}_w \Vdash \texttt{emp} \wedge \mathbb{B}.\text{pc} = \texttt{L2} \wedge \mathbb{R}(\texttt{t3}) = 0\}$
  `sw t3 fork1 zero`
  $-\{\exists\ v. \mathbb{H}_s \Vdash \texttt{fork}_1 \mapsto 0 * \texttt{fork}_2 \mapsto v$
    $\wedge \mathbb{H}_w \Vdash \texttt{fork}_1 \mapsto 0 \wedge \mathbb{B}.\text{pc} = \texttt{L2} \wedge \mathbb{R}(\texttt{t3}) = 0\}$
  `sw t3 fork2 zero`
  $-\{\mathbb{H}_s \Vdash \texttt{fork}_1 \mapsto 0 * \texttt{fork}_2 \mapsto 0$
    $\wedge \mathbb{H}_w \Vdash \texttt{fork}_1 \mapsto 0 * \texttt{fork}_2 \mapsto 0 \wedge \mathbb{B}.\text{pc} = \texttt{L2}\}$
  `commit`
  $-\{\texttt{true}\}$
  `j L1`

L3: $-\{\text{pc} = \texttt{L3} \wedge \mathbb{H}_s \Vdash \texttt{fork}_1 \mapsto 1 * \texttt{fork}_2 \mapsto 1$
    $\wedge \mathbb{H}_w \Vdash \texttt{emp} \wedge \mathbb{B}.\text{pc} = \texttt{L1}\}$
  `commit`
  $-\{\texttt{true}\}$
  `j L1`

Fig.8. Dining Philosophers in our system.

The corresponding specification and program for our system are shown in Fig.8. Only the code for Thread_1 is given because that of Thread_2 is similar. In this example, *Inv* defines the invariant on the global heap for this program ($\texttt{fork}_1 = \texttt{fork}_2$).

We explain the code block labeled `L1`, which corresponds to the operations of picking up both forks for philosopher 1. The precondition of this block indicates that all jumpings to this block are allowed because ($\text{pc} = \texttt{L1}$) is always valid when it is about to execute the instruction labeled `L1`. After the block starts a transaction, the write attempts $\mathbb{H}_w$ of this transaction are set empty and the global heap $\mathbb{H}$ is copied to local $\mathbb{H}_s$, so the local heap also respects the global invariant here (*Inv* $\mathbb{H}_s$). Also the target of rollback is set where the transaction starts ($\mathbb{B}.\text{pc} = \texttt{L1}$). Then it tries to pick up both forks (updates $\mathbb{H}_s$) after checking that the forks are free at its view. Note that the invariant is broken locally ($\mathbb{H}_s \Vdash \texttt{fork}_1 \mapsto 1 * \texttt{fork}_2 \mapsto 0$) when the first fork is picked (`sw t2 fork1 zero`), and is reestablished when the other fork is also picked before the transaction commits. Then it commits its picking up ($\mathbb{H}_w \Vdash \texttt{fork}_1 \mapsto 1 * \texttt{fork}_2 \mapsto 1$) to the global heap $\mathbb{H}$ if no forks have been picked and are being held by the other (Thread_2) during its try.

Note that the reasoning of Thread_1 needs no information of Thread_2. Due to the guarantee that all modifications to the global heap will respect the global invariant, we are able to perform such reasoning. And it is obvious that the composition of such reasoning is easy, since we need only to identify the compatibility of the invariants of different threads to increase concurrency.

When reasoning about the lock-based program for this example[21], programmers are required to ensure the non-interference between every two threads. And when composing threads, the non-interference between the newly-added one and all the existent is required. The concurrency-control mechanism, transactional memory, significantly eases the reasoning.

Interestingly, we can perform a similar reasoning for the corresponding high-level programs in Fig.7. We know that ($\texttt{fork}_1 = \texttt{fork}_2$) is valid when the transaction starts. And the forks are free to pick if ($\texttt{fork}_1 = 0 = \texttt{fork}_2$). Then the philosopher will commit his picking attempts successfully if shared memory has not been updated by the other thread. Finally, shared memory remains in a state that respects the invariant ($\texttt{fork}_1 = 1 = \texttt{fork}_2$).

Usually, high-level languages use a programming structure `atomic{...}` to identify a transaction. Such a transaction structure denies the flexibility of constructing a transaction, where branches may lead to different components of a same start of a transaction, as shown in Fig.7. For the same philosopher thread, an auxiliary variable is needed to record the picking attempts inside the transaction and used to decide whether to try for forks again before eating. Fig.9 presents the code fragment using an auxiliary variable for the atomic block. We can see that the assembly-level abstraction does provide certain convenience.

```
while (TRUE) {
  picked = FALSE;
  atomic{
    if (! fork_1){
      fork_1 = fork_2 = 1;
      picked = TRUE;
    }
  }
  if (! picked) continue;
  // eat
⋮
```

Fig.9. Auxiliary variable in atomic block.

4.2 Producer-Consumer Algorithm

Fig.10 presents the code for the producer-consumer algorithm where producers keep putting goods into the

```
Variables:                  Invariant:
bool Full, Empty;           !  (Full ∧ Empty)
int In, Out;                0 ⩽ In, Out < BFSIZE

Producer :                  Consumer :
while (TRUE) {              while (TRUE) {
  starttrans;                 starttrans;
  if (! Full)                 if (! Empty)
  {                           {
    In++;                       Out++;
    if (In == BFSIZE)           if (Out == BFSIZE)
      In -= BFSIZE;               Out -= BFSIZE;
    if (In == Out)              if (Out == In)
      Full = TRUE;                Empty = TRUE;
    Empty = FALSE;              Full = FALSE;
  }                           }
  commit;                     commit;
}                           }
```

Fig.10. Producer-consumer.

```
Inv ℍ ≜ ∃ v1, v2, v3, v4. ℍ ⊩ Full ↦ v1 * Empty ↦ v2*
          In ↦ v3 * Out ↦ v4
          ∧ v1 = TRUE → v2 = FALSE
          ∧ v2 = TRUE → v1 = FALSE
          ∧ v3 < BFSIZE ∧ v4 < BFSIZE

L_start:   −{pc = L_start}
        starttrans
        lw rfull Full  zero
        addiu rtrue zero TRUE
        bne rfull rtrue L_addin
        j L_commit

L_addin:   −{𝔹.pc = L_start ∧ ℍ_w ⊩ emp
           ∧ ∃ v1, v2, v3, v4. ℍ_s ⊩ Full ↦ v1*
             Empty ↦ v2 * In ↦ v3 * Out ↦ v4
             ∧ v3 < BFSIZE ∧ v4 < BFSIZE
             ∧ ℝ(rtrue) = TRUE ∧ v1 ≠ TRUE}
        lw rin In zero
        addiu rin rin 1
        addiu rtemp zero BFSIZE
        beq rin rtemp L_subR
        j L_fcheck

L_fcheck:
        sw rin In zero
        lw rout Out zero
        beq rin rout L_fset
        j L_setemp

L_setemp:  −{𝔹.pc = L_start
           ∧ ∃ v1, v2, v3, v4. ℍ_s ⊩ Full ↦ v1*
             Empty ↦ v2 * In ↦ v3 * Out ↦ v4
             ∧ v3 < BFSIZE ∧ v4 < BFSIZE
             ∧ ((v1 ≠ TRUE ∧ ℍ_w ⊩ In ↦ v3)
               ∨ (v1 = TRUE
                 ∧ ℍ_w ⊩ Full ↦ v1 * In ↦ v3))}
        addiu remp zero FALSE
        sw remp Empty zero
        j L_commit

L_commit:  −{𝔹.pc = L_start
           ∧ ∃ v1, v3, v4. ℍ_s ⊩ Full ↦ v1*
             Empty ↦ FALSE * In ↦ v3 * Out ↦ v4
             ∧ v3 < BFSIZE ∧ v4 < BFSIZE
             ∧ ((v1 ≠ TRUE
                 ∧ ℍ_w ⊩ Empty ↦ FALSE * In ↦ v3)
               ∨ (v1 = TRUE
                 ∧ (ℍ_w ⊩ Full ↦ v1 * Empty ↦ FALSE*
                   In ↦ v3 ∨ ℍ_w ⊩ emp)))}
        commit
        j L_start

L_subR:
        subu rin rin rtemp
        j L_fcheck

L_fset:
        sw rtrue Full  zero
        j L_setemp
```

Fig.11. Producer-consumer in our system.

storage while consumers keep consuming goods. The storage is modeled as a finite First In First Out queue in the program. Threads (producers and consumers) operate on the queue concurrently while keeping the queue consistent: the queue will not overflow and will not be full and empty simultaneously. This invariant is expressed as $!\,(\texttt{Full} \wedge \texttt{Empty}) \ \wedge \ (0 \leqslant \texttt{In}, \texttt{Out} < \texttt{BFSIZE})$, where `Full` and `Empty` indicate the queue is full or empty respectively, `In` points to the location where producers are about to put goods and `Out` points to the location where consumers are about to consume goods.

The assembly code for producers and the corresponding invariant for the program are presented in Fig.11. The specifications are also shown of where a producer thread is about to start a transaction (labeled `L_start`, to increase goods (labeled `L_addin`), to signal the non-emptiness of the storage (labeled `L_setemp`) and to commit the transaction (labeled `L_commit`). And the code and specifications for consumers are similar and not shown here.

Producers and consumers would start transactions and collect effects they attempt to make based on their local views to the global heap. Then they would commit their modifications to the global heap with respect to the global invariant when their attempts are validated.

Our work is fully mechanized in the Coq proof assistant, including the system, its soundness proof and the examples that are demonstrated above. Interested readers can check out our Coq implementation[28] for more details.

5 Related Work

Many methods have been proposed for reasoning about properties of both sequential and concurrent programs[29−33]. But most efforts on concurrent programs focus on the lock-based programs and do not aim at programs using TM. And the lock-based reasoning requires programmers to reason about the interference between threads. But as we present in this paper, TM-based reasoning does not. There is an easier way enforced by the TM semantics to perform TM-based reasoning.

Harris *et al.*[34] require an explicit definition of invariants from programmers to track the dependencies between transactions. But they do not specify the properties of a program. Concurrent separation logic[35] separates shared memory into pieces, each of which is guarded by a lock and respects certain invariant, to perform the verification of lock-based concurrent programs. We are enlightened by these work and specify invariants on shared memory to check the correctness of interaction. And we formalize our reasoning into a PCC[16] system.

Recent years, Shao *et al.* have developed CCAP[21], CMAP[22] and SAGL[36] as extensions to the PCC framework to verify properties of concurrent programs using locks. And we present an extension to enable verification of concurrent programs using TM.

Also, much work has started to explore the semantics of transactional memory. Moore and Grossman[13] have presented type systems to study the behaviors of spawning threads in transactional programs and showed the relation of strong and weak atomicity in TM system's implementations. They presented the semantics of transactions in a λ-calculus. In a later version of their work[14], they expanded to study the semantics of the rollback operation for eager versioning in details. They used different type systems to study different issues.

Abadi *et al.*[15] developed semantics and type systems for the Automatic Mutual Exclusion (AME) calculus, which encourage programmers to place as much of the program text inside transactions as possible. They also presented the semantics and correctness of the rollback operation in their systems.

Vitek *et al.*[11] used a variant of Featherweight Java[37] to define a framework to explore various implementation strategies of TM systems and to ensure the correctness of an implementation by establishing a serializability result. And they assumed all codes execute inside transactions.

However, most work on TM semantics has just explored definition of abstract machines and execution of transactional programs, and does not propose rules for certifying general properties of concurrent programs using TM.

6 Conclusion

In this paper we present a system for verifying concurrent programs using transactional memory. We modeled an assembly-level machine with built-in transactional memory. We incorporated the well-known invariant-based proof method into our system by requiring the preservation of invariant not only on properties of each thread of a program but also on the shared memory these threads cooperate on. We proved our reasoning sound with respect to the TM semantics. And we used a few examples to demonstrate the effectiveness of our techniques. For now, our system can reason about safety properties of programs using TM only. Enriching the expressiveness of the program logic of our system to enable the verification of liveness properties such as fairness will be part of our future work. And we also

plan to study the violations in implementations of TM systems in more depth and define a more flexible programming model for supporting transactional memory.

References

[1] C Scott Ananian, Krste Asanovic, Bradley C Kuszmaul, Charles E Leiserson, Sean Lie. Unbounded transactional memory. In *Proc. the Eleventh International Symposium on High-Performance Computer Architecture*, San Francisco, USA, Feb. 2005, pp.316–327.

[2] Hammond L, Wong V, Chen M, Carlstrom B D, Davis J D, Hertzberg B, Prabhu M K, Wijaya H, Kozyrakis C, Olukotun K. Transactional memory coherence and consistency. In *Proc. the 31st Annual International Symposium on Computer Architecture*, München, Germany, IEEE Computer Society, Jun. 2004, p.102.

[3] Herlihy M, Moss J E B. Transactional memory: Architectural support for lock-free data structures. In *Proc. the 20th Annual International Symposium on Computer Architecture*, San Diego, USA, May 1993, pp.289–300.

[4] Moore K E, Bobba J, Moravan M J, Hill M D, Wood D A. LogTM: Log-based transactional memory. In *Proc. the 12th International Symposium on High-Performance Computer Architecture*, Austin, USA, Feb. 2006, pp.254–265.

[5] Harris T, Fraser K. Language support for lightweight transactions. In *Proc. the 18th Annual ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, USA, Oct. 2003, pp.388–402.

[6] Harris T, Herlihy M, Marlow S, Peyton-Jones S. Composable memory transactions. In *Proc. the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, USA, Jun. 2005, pp.48–60.

[7] Herlihy M, Luchangco V, Moir M, Scherer III W N. Software transactional memory for dynamic-sized data structures. In *Proc. the Twenty-Second Annual Symposium on Principles of Distributed Computing* (*PODC '03*), Boston, Massachusetts, USA, ACM Press, 2003, pp.92–101.

[8] Shavit N, Touitou D. Software transactional memory. In *Proc. the 14th ACM Symposium on Principles of Distributed Computing*, Ottawa, Canada, Aug. 1995, pp.204–213.

[9] Kumar S, Chu M, Hughes C J, Kundu P, Nguyen A. Hybrid transactional memory. In *Proc. the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, USA, Mar. 2006, pp.209–220.

[10] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L Hudson, Chi Cao Minh, Benjamin Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proc. the 11th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (*PPoPP '06*), New York, USA, Mar. 2006, pp.187–197.

[11] Vitek J, Jagannathan S, Welc A, Hosking A L. A semantic framework for designer transactions. In *Proc. the 13th European Symposium on Programming*, Barcelona, Spain, *Lecture Notes in Computer Science* 2986, Springer-Verlag, Apr. 2004, pp.249–263.

[12] Ben Liblit. An operational semantics for LogTM. Version 1.0, Technical Report 1571, University of Wisconsin–Madison, Aug. 2006.

[13] Moore K F, Grossman D. High-level small-step operational semantics for transactions. In *Proc. the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, USA, Aug. 2007.

[14] Moore K F, Grossman D. High-level small-step operational semantics for transactions. In *Proc. the 35th ACM Symposium on Principles of Programming Languages*, San Francisco, USA, Jan. 2008, pp.51–62.

[15] Abadi M, Birrell A, Harris T, Isard M. Semantics of transactional memory and automatic mutual exclusion. In *Proc. the 35th ACM Symposium on Principles of Programming Languages*, San Francisco, USA, Jan. 2008, pp.63–74.

[16] Necula G C. Proof-carrying code. In *Proc. the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL '97*), Paris, France, ACM Press, 1997, pp.106–119.

[17] Appel A W. Foundational proof-carrying code. In *Proc. the 16th Annual IEEE Symposium on Logic in Computer Science* (*LICS'01*), Boston, USA, IEEE Computer Society, Jun. 2001, pp.247–258.

[18] Yu D, Hamid N A, Shao Z. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 2004, 50 (1-3): 101–127.

[19] Feng X, Shao Z, Vaynberg A, Xiang S, Ni Z. Modular verification of assembly code with stack-based control abstractions. In *Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'06*), Ottawa, Canada, Jun. 2006, pp.401–414.

[20] Ni Z, Shao Z. Certified assembly programming with embedded code pointers. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL'06*), Charleston, South Carolina, USA, ACM Press, 2006, pp.320–333.

[21] Yu D, Shao Z. Verification of safety properties for concurrent assembly code. In *Proc. the 9th ACM SIGPLAN International Conference on Functional Programming* (*ICFP'04*), Snowbird, Utah, Sept. 2004, pp.175–188.

[22] Feng X, Shao Z. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. the 10th ACM SIGPLAN International Conference on Functional Programming* (*ICFP'05*), Tallinn, Estonia, ACM Press, Sept. 2005, pp.254–267.

[23] The Coq proof assistant reference manual. The Coq release v8.0, Coq Development Team, Oct. 2005.

[24] Colin Blundell, E Christopher Lewis, Milo M K Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, Nov. 2006, 5(2): 17.

[25] Spear M F, Marathe V J, Dalessandro L, Scott M L. Privatization techniques for software transactional memory. In *Proc. the 26th ACM Symposium on Principles of Distributed Computing* (*PODC*), Portland, USA, Aug. 2007, pp.338–339.

[26] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L Hudson, Katherine F Moore, Bratin Saha. Enforcing isolation and ordering in STM. In *Proc. the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'07*), San Diego, California, USA, ACM Press, 2007, pp.78–88.

[27] Reynolds J C. Separation logic: A logic for shared mutable data structures. In *Proc. the 17th Annual IEEE Symposium on Logic in Computer Science* (*LICS'02*), Copenhagen, Denmark, IEEE Computer Society, 2002, pp.55–74.

[28] Li L, Zhang Y, Chen Y Y, Li Y. Certifying concurrent programs using transactional memory (documents and Coq implementation). Aug. 2007, http://ssg.ustcsz.edu.cn/ccac/.

[29] Floyd R W. Assigning meanings to programs. In *Proc. the Symposium on Applied Math.*, American Mathematical Society (ed.), Providence, R.I., 1967, Vol.19, pp.19–31.

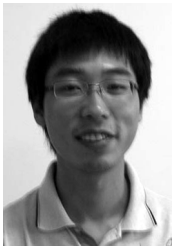
[30] Hoare C A R. An axiomatic basis for computer programming. *Communications of the ACM*, Oct. 1969, 12(10): 576–580.

[31] Owicki S, Gries D. An axiomatic proof technique for parallel programs. *Acta Informatica*, 1976, 6(4): 319–340.
[32] Lamport L, Schneider F B. The "Hoare logic" of CSP, and all that. *ACM Transactions on Programming Languages and Systems*, Apr. 1984, 6(2): 281–296.
[33] Lamport L. The temporal logic of actions. *ACM Transactions on Programming languages and Systems*, May 1994, 16(3): 872–923.
[34] Tim Harris, Simon Peyton Jones. Transactional memory with data invariants. In *Proc. The First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing* (*TRANSACT'06*), Ottawa, Canada, Jun. 2006.
[35] Peter W O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 2007, 375(1-3): 271–307.
[36] Feng X, Ferreira R, Shao Z. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proc. the 16th European Symposium on Programming* (*ESOP'07*), *LNCS* 4421, Braga, Portugal, Mar. 2007, pp.173–188.
[37] Atsushi Igarashi, Benjamin C Pierce, Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 2001, 23(3): 396–450.

Long Li is currently a Ph.D. candidate in Department of Computer Science & Technology, University of Science & Technology of China (USTC). He received his B.E. degree in computer science from USTC in 2003. His research interests involve language based software safety, program verification on assembly code level, garbage collection and concurrent program verification.

Yu Zhang received the M.E. degree in computer science from Hefei University of Technology in 1996 and the Ph.D. degree in computer science from University of Science & Technology of China (USTC) in 2004. She is currently an associate professor in Department of Computer Science & Technology at USTC. Her research interests include theory and implementation of programming language, especially on techniques for designing and implementing parallel programming languages, concurrent program analysis and verification, Just-in-time compiler assisted garbage collection. She is a member of China Computer Federation.

Yi-Yun Chen is a professor in Department of Computer Science & Technology, University of Science & Technology of China. He received his M.S. degree from East-China Institute of Computer Technology in 1982. His research interests include applications of logic (including formal semantics and type theory), techniques for designing and implementing programming languages and software safety and security. He is a member of China Computer Federation.



Yong Li is currently a Ph.D. candidate in Department of Computer Science & Technology at University of Science & Technology of China (USTC). He received his B.E. degree in computer science from USTC in 2005. His research interests involve language based software safety, program verification on assembly code level, and concurrent program verification.