An Operational Semantics for LogTM

Ben Liblit

Technical Report #1571

August 2006

An Operational Semantics for LogTM

Ben Liblit
Department of Computer Sciences, University of Wisconsin–Madison
liblit@cs.wisc.edu

Version 1.0, August 10, 2006

Abstract

We present a formal operational semantics for LogTM, a hardware-based nested transactional memory system. We define the proper execution of programs written in a small assembly language that includes memory accesses, nested closed and open transactions, partial rollback, commit and abort handlers, thread spawning, and escape actions. This is a working document, intended to reflect and codify the current best understanding of LogTM's operation in both common and corner cases. This formal semantics serves as a reference companion to other published discussions of LogTM, and specifically corresponds to the system described in "Supporting Nested Transactional Memory in LogTM" by Moravan et al.

1 Purpose and Status of This Document

This document presents a formal operational semantics for the LogTM hardware transactional memory system. We define a small assembly language of memory accesses and transaction operations, and state the required behavior of any LogTM implementation when executing programs in this language. Some implementation-specific behavior is identified but intentionally left unspecified. Additionally, this semantics models a forward log whereas current LogTM implementations use backward logs. This intentional deviation forces the semantics to be less dependent on the particulars of any given implementation.

This document is dense in mathematical notation with only a minimum of explanatory prose. It is not intended to be read alone, but rather should be treated as a reference companion to other published descriptions of the LogTM architecture and implementation [2, 3]. Specifically, we describe the behavior of the nested LogTM system of Moravan et al. [3]. We hope that this document will be particularly useful for documenting LogTM's behavior in complex corner cases that other treatments may hint at but not codify in detail.

A formal semantics of this nature is of similar complexity to program code, and furthermore has not been mechanically tested or verified. Revisions may be required to correct errors or to reflect future changes to LogTM. The reader is strongly urged to visit `<http://www.cs.wisc.edu/multifacet/papers/>`, where corrected or updated semantics will be posted as they become available. Additionally, any citations to this document should identify the version number of the semantics used (currently Version 1.0) to avoid confusion with respect to later changes.

2 Definitions

2.1 Fundamentals

Let $\vec{s} = \langle s_1, s_2, \ldots, s_n \rangle$ represent a possibly empty vector with ordered elements $s_1, s_2, \ldots, s_n$. The empty vector is written as $\langle\rangle$. Let "$\cdot$" be the right-associative single-element prepending operator such that $s_1 \cdot \langle s_2, \ldots, s_n \rangle = \langle s_1, s_2, \ldots, s_n \rangle$. Let "::" be the vector concatenation operator such that $\langle s_1, \ldots, s_j \rangle :: \langle s_{j+1}, \ldots, s_n \rangle = \langle s_1, \ldots, s_j, s_{j+1}, \ldots, s_n \rangle$.

By convention, numeric subscripts (s_i) are used to index vector elements while prime (s') and double prime (s'') marks identify successively updated versions of some initial value (s) as execution proceeds forward through time. This convention is informal and may be violated in certain awkward cases.

For any set S, let $\vec{S}$ represent the set of all vectors of elements of S.

For any function f, let $\mathit{domain}(f)$ be the domain of f.

For any function f and set D, let $\mathit{restrict}(f,D)$ be the domain restriction of f to D. That is, $\mathit{restrict}(f,D)(z) = f(z)$ when $z \in D$, but is undefined otherwise.

For any pair of functions f, g let "$f;g$" represent the cascading union of f and g defined as

$$(f;g)(x) = \begin{cases} f(x) & \text{if } x \in \mathit{domain}(f) \\ g(x) & \text{otherwise} \end{cases} \tag{1}$$

For any function f and values p, q, let $f[p \mapsto q]$ represent the function that maps p to q but is otherwise identical to f.

Let "_" represent a fresh, unnamed variable of appropriate type in context but otherwise unconstrained.

2.2 Instructions

Let *Addresses* be the set of addressable memory locations in a program's address space, with elements a, a_i, a', etc.

Let *Values* be the set of values that may be stored in addressable memory locations, with elements v, v_i, v', etc.

Let $\mathit{EscapeStates} = \{\mathit{escaped}, \mathit{unescaped}\}$ be the set of possible escape states, with elements e, e_i, e', etc.

Let *Instructions* be the set of possible machine instructions as seen by the LogTM system, with elements $x, x_i, x', y, y_i, y', z, z_i, z'$, etc. Informally we use x to represent regular execution, y to represent commit handlers, and z to represent abort handlers. *Instructions* is given by the following recursive grammar:

$$
\begin{array}{rcll}
\textit{Instructions} & ::= & \texttt{read}\ a & (2) \\
& | & \texttt{write}\ a\ v & (3) \\
& | & \texttt{begin} & (4) \\
& | & \texttt{commitClosed} & (5) \\
& | & \texttt{commitOpen}\ \vec{y}\,\vec{z} & (6) \\
& | & \texttt{escape} & (7) \\
& | & \texttt{unescape}\ \vec{y}\,\vec{z} & (8) \\
& | & \texttt{abort} & (9) \\
& | & \texttt{setState}\ e\,\vec{t} & (10) \\
& | & \texttt{spawn}\ \vec{x} & (11)
\end{array}
$$

for all $a \in \textit{Addresses}$, $e \in \textit{EscapeStates}$, $\vec{t} \in \overrightarrow{\textit{TransStates}}$, $v \in \textit{Values}$, and $\vec{x}, \vec{y}, \vec{z} \in \overrightarrow{\textit{Instructions}}$. See subsection 2.5 below for the definition of *TransStates*.

The `setState` "instruction" is a special operation used to manage the environments in which abort handlers run. It should never appear in any actual program.

2.3 Memory

Addresses and values should be defined at the granularity of logging for any given implementation. For example, if 32-bit values are logged, then *Values* represents 32-bit values and *Addresses* are 32-bit-aligned locations that can hold these values. Even if a program only wants a single byte, it must read a complete 32-bit value first.

Let $\textit{ReadSets} = 2^{\textit{Addresses}}$ be the set of possible read sets with elements r, r_i, r', etc.

Let $\textit{ValueMaps} = \textit{Addresses} \rightarrow \textit{Values}$ be the set of possible value maps with elements w, w_i, w', etc.

Let $\textit{MemoryMaps} = \textit{Addresses} \rightarrow \textit{Values}$ be the set of possible main memory maps with elements m, m_i, m', etc.

Although *ValueMaps* and *MemoryMaps* have the same types, by convention (and by construction) all elements of *MemoryMaps* are complete functions while *ValueMaps* includes partial functions.

2.4 Address Blocks

LogTM implementations may log memory operations at a larger granularity than that of individual addresses. For example, current implementations log 64-byte blocks while memory accesses may be 4- or even 1-byte aligned. Conditions O1, X1, X2, and X3, defined and used below, apply at the coarser granularity of the LogTM log. We must therefore model the distinction between addresses and address blocks.

Let *Blocks* be some fixed set of address blocks. Let $addressBlock : Addresses \rightarrow Blocks$ be a complete function that maps each address to its corresponding address block. This mapping is implementation-defined, but must remain fixed for the duration of any program.

Let $addressBlocks : 2^{Addresses} \rightarrow 2^{Blocks}$ be the element-wise extension of *addressBlock* across sets. That is, for any set of addresses $A \subseteq Addresses$, $addressBlocks(A) = \bigcup_{a \in A} addressBlock(a)$.

2.5 Transaction State

Let $TransStates = ReadSets \times ValueMaps \times \overrightarrow{Instructions} \times \overrightarrow{Instructions}$ be the set of possible transaction states with elements t, t_i, t', etc. Informally, one transaction state records

- the set of addresses read during this transaction
- the map from addresses to corresponding values written during this transaction
- the current sequence of commit handlers
- the current sequence of abort handlers

2.6 Thread States

Let *ThreadIds* be the set of possible thread IDs, with elements d, d_i, d', etc.

Let $ThreadStates = EscapeStates \times \overrightarrow{TransStates} \times \overrightarrow{Instructions}$ represent the set of possible thread states. A thread consists of an escape bit, a stack of transaction states, and an ordered vector of instructions to execute.

Let $\Pi = ThreadIds \rightarrow ThreadStates$ be the set of possible program states with elements π, π_i, π', etc. A program consists of a collection of named threads.

2.7 System State

Let $\Sigma = MemoryMaps \times \Pi$ be the set of possible system states with elements $\sigma, \sigma_i, \sigma'$, etc. A system consists of main memory and a running program.

Let *Initial* be the subset of initial system states, defined as the set of states $(m, \pi) \in \Sigma$ for which all of the following conditions hold:

$$domain(m) = Addresses \tag{12}$$

$$\pi = \emptyset[\_ \mapsto (unescaped, \langle\rangle, \_)] \tag{13}$$

Note that we require that m be complete without specifying its value for any given address. Every address must hold some initial but arbitrary value. The pool of active threads initially consists of a single thread in the base non-transactional state with some arbitrary sequence of pending instructions.

3 Semantics

3.1 Judgments

A judgment $\sigma \Downarrow m$ is read "starting in an initial system state σ, evaluation of all threads' instructions terminates with the main memory of the system in final state m."

A judgment $\sigma \Rightarrow^* \sigma'$ is read "starting in system state σ, running for zero or more steps leaves the system in sate σ'."

A judgment $\sigma \Rightarrow \sigma'$ is read "starting in system state σ, running for a single step leaves the system in state σ'."

A judgment $\sigma \vdash d \Rightarrow \sigma'$ is read "starting in system state σ, running thread d for a single step leaves the system in state σ'."

A judgment $\sigma \vdash d, x \Rightarrow \sigma'$ is read "starting in system state σ, evaluation of instruction x by thread d leaves the system in state σ'."

3.2 Sequential Execution, Thread Management, and Termination

The system begins in an initial state and runs until all threads have been reaped.

$$\text{HALT MEM}\quad \frac{\sigma \in \mathit{Initial} \qquad \sigma \Rightarrow^* (m, \emptyset)}{\sigma \Downarrow m}$$

Running multiple steps is the obvious transitive closure of running zero or more single steps.

$$\text{NO STEP}\quad \frac{}{\sigma \Rightarrow^* \sigma}$$

$$\text{MULTI STEP}\quad \frac{\sigma \Rightarrow \sigma' \qquad \sigma' \Rightarrow^* \sigma''}{\sigma \Rightarrow^* \sigma''}$$

A thread may be reaped when it has no instructions left to execute and is in the base non-transactional state. Note that if multiple threads can be reaped, the order in which they are reaped is arbitrary. Similarly, single steps involving thread execution may be interleaved arbitrarily with single steps of thread reaping.

$$\text{REAP}\quad \frac{\begin{array}{c}\sigma = (m, \pi) \qquad \pi(d) = (\mathit{unescaped}, \langle\rangle, \langle\rangle) \\ \sigma' = (m, \mathit{restrict}(\pi, \mathit{ThreadIds} - \{d\}))\end{array}}{\sigma \Rightarrow \sigma'}$$

A single step of the system consists of selecting one thread and allowing that thread to take a single step. Note that selection of the next thread to execute is non-deterministic. We do not model the thread scheduler in this semantics.

$$\text{SYSTEM STEP}\quad \frac{\sigma = (\_, \pi) \qquad d \in \mathit{domain}(\pi) \qquad \sigma \vdash d \Rightarrow \sigma'}{\sigma \Rightarrow \sigma'}$$

A single step of a thread consists of removing and evaluating the next pending instruction for that thread.

$$\text{FETCH}\quad \frac{\begin{array}{c}\sigma = (m, \pi) \qquad \pi(d) = (e, \vec{t}, x_0 \cdot \vec{x}) \\ \sigma' = (m, \pi[d \mapsto (e, \vec{t}, \vec{x})]) \\ \sigma' \vdash d, x_0 \Rightarrow \sigma''\end{array}}{\sigma \vdash d \Rightarrow \sigma''}$$

A new thread may be spawned during an escape action. The newly spawned thread begins in the base, non-transactional state.

$$\text{SPAWN}\quad \frac{\begin{array}{c}\sigma = (m, \pi) \qquad \pi(d) = (\mathit{escaped}, \_, \_) \\ d' \notin \mathit{domain}(\pi) \\ \sigma' = (m, \pi[d' \mapsto (\mathit{unescaped}, \langle\rangle, \vec{x})])\end{array}}{\sigma \vdash d, \texttt{spawn}\, \vec{x} \Rightarrow \sigma'}$$

3.3 Conflicting Memory Operations

3.3.1 Conflict Detection

Let $\mathit{allRead} : \overrightarrow{\mathit{TransStates}} \rightarrow 2^{\mathit{Addresses}}$ be the set of all addresses read by any transaction in the given transaction stack, defined as follows:

$$\mathit{allRead}(\langle\rangle) = \emptyset \tag{14}$$

$$\mathit{allRead}((r, \_, \_, \_) \cdot \vec{t}) = r \cup \mathit{allRead}(\vec{t}) \tag{15}$$

Let $\mathit{allWritten} : \overrightarrow{\mathit{TransStates}} \rightarrow 2^{\mathit{Addresses}}$ be the set of all addresses written by any transaction in the given transaction stack, defined as follows:

$$\mathit{allWritten}(\langle\rangle) = \emptyset \tag{16}$$

$$\mathit{allWritten}((\_, w, \_, \_) \cdot \vec{t}) = \mathit{domain}(w) \cup \mathit{allWritten}(\vec{t}) \tag{17}$$

Let *allAccessed* : $\overrightarrow{TransStates} \to 2^{Addresses}$ be the set of all addresses read or written by any transaction in the given transaction stack, defined as follows:

$$allAccessed(\vec{t}) = allRead(\vec{t}) \cup allWritten(\vec{t}) \tag{18}$$

Let $conflict \subseteq Addresses \times Addresses$ be the address conflict relation. This relation is intentionally left unspecified here. Address conflicts are implementation-defined and may even be nondeterministic. However, we require that *conflict* be reflexive: every address always conflicts with itself.

Let $writeConflict \subseteq \Pi \times ThreadIds \times Addresses$ be the write conflict relation, defined as follows:

$$\textsc{check write}\quad \frac{\begin{array}{cc} d' \neq d & \pi(d') = (\_, \vec{t}, \_) \\ a' \in allWritten(\vec{t}) & conflict(a, a') \end{array}}{writeConflict(\pi, d, a)}$$

Let $accessConflict \subseteq \Pi \times ThreadIds \times Addresses$ be the access conflict relation, defined as follows:

$$\textsc{check read}\quad \frac{\begin{array}{cc} d' \neq d & \pi(d') = (\_, \vec{t}, \_) \\ a' \in allAccessed(\vec{t}) & conflict(a, a') \end{array}}{accessConflict(\pi, d, a)}$$

3.3.2 Conflict Resolution

A read conflicts if some other thread has an uncommitted transactional write to a conflicting address. Conflicting reads abort.

$$\textsc{read conf}\quad \frac{\begin{array}{c} \sigma = (\_, \pi) \quad writeConflict(\pi, d, a) \\ \pi(d) = (unescaped, \_ \cdot \_, \_) \\ \sigma \vdash d, \texttt{abort} \Rightarrow \sigma' \end{array}}{\sigma \vdash d, \texttt{read}\ a \Rightarrow \sigma'}$$

A write conflicts if some other thread has an uncommitted transactional write to a conflicting address or an uncommitted transactional read from a conflicting address. Conflicting writes abort.

$$\textsc{write conf}\quad \frac{\begin{array}{c} \sigma = (\_, \pi) \quad accessConflict(\pi, d, a) \\ \pi(d) = (unescaped, \_ \cdot \_, \_) \\ \sigma \vdash d, \texttt{abort} \Rightarrow \sigma' \end{array}}{\sigma \vdash d, \texttt{write}\ a\ v \Rightarrow \sigma'}$$

Note that we provide no semantic rules for execution of conflicting reads or writes in the base, non-transactional state. The implied requirement is that these instructions stall until such time as they are non-conflicting. Thus, LogTM provides strong atomicity as defined by Blundell et al. [1].

3.4 Non-Conflicting Memory Operations

3.4.1 Escape

Let *obeyX3* $\subseteq \Pi \times \mathit{ThreadIds} \times \mathit{Addresses}$ be the Condition X3 compliance relation for a potential read, defined as follows:

$$\begin{aligned} \mathit{obeyX3}(\pi,d,a) \iff \forall d' \neq d.\pi(d') = (\_,\vec{t},\_) \\ \Rightarrow \mathit{addressBlock}(a) \notin \mathit{addressBlocks}(\mathit{allWritten}(\vec{t})) \end{aligned} \quad (19)$$

We intentionally do not state the semantics of escaped reads that violate Condition X3. However, any given implementation may choose to extend these semantics by defining the behavior of such reads.

An X3-compliant escaped read fetches a value from the current thread's nested transaction stack. Transactional state is not modified.

$$\text{ESCAPED READ}\ \frac{\begin{array}{c}\sigma = (m,\pi) \qquad \pi(d) = (\mathit{escaped},\vec{t},\_) \\ \mathit{obeyX3}(\pi,d,a) \\ v = \mathit{get}(a,m,\vec{t})\end{array}}{\sigma \vdash d, \texttt{read}\ a \Rightarrow \sigma}$$

Let $\mathit{get} : \mathit{Addresses} \times \mathit{MemoryMaps} \times \overrightarrow{\mathit{TransStates}} \to \mathit{Values}$ be the transactional value fetching function. Getting a value from a nested transaction stack requires finding the innermost value map with a mapping for the desired address.

$$\text{GET TOP}\ \frac{\vec{t} = (\_,w,\_,\_)\cdot\_ \qquad v = w(a)}{\mathit{get}(a,m,\vec{t}) = v}$$

$$\text{GET DEEP}\ \frac{\begin{array}{c}\vec{t} = (\_,w,\_,\_)\cdot\vec{t'} \qquad a \notin \mathit{domain}(w) \\ \mathit{get}(a,m,\vec{t'}) = v\end{array}}{\mathit{get}(a,m,\vec{t}) = v}$$

If no such value map is found, the value from main memory is provided as a last resort.

$$\text{GET MEM}\ \frac{v = m(a)}{get(a, m, \langle\rangle) = v}$$

Let $obeyX1 \subseteq \overrightarrow{TransStates} \times Addresses$ be the Condition X1 compliance relation for a potential write, defined as follows:

$$obeyX1(\langle\rangle, a) \tag{20}$$

$$obeyX1(\_ \cdot \vec{t}, a) \iff addressBlock(a) \notin addressBlocks(allWritten(\vec{t})) \tag{21}$$

Let $obeyX2 \subseteq \Pi \times ThreadIds \times Addresses$ be the Condition X2 compliance relation for a potential write, defined as follows:

$$obeyX2(\pi, d, a) \iff \forall d' \neq d . \pi(d') = (\_, \vec{t}, \_) \Rightarrow addressBlock(a) \notin addressBlocks(allAccessed(\vec{t})) \tag{22}$$

We intentionally do not state the semantics of escaped writes that violate Conditions X2 or X3. However, any given implementation may choose to extend these semantics by defining the behavior of such writes.

An X1- and X2-compliant escaped write directly updates main memory without changing any transactional state.

$$\text{ESCAPED WRITE MEM}\ \frac{\begin{array}{c}\sigma = (m, \pi) \qquad \pi(d) = (escaped, \vec{t}, \_) \\ obeyX1(\vec{t}, a) \qquad obeyX2(\pi, d, a) \\ \sigma' = (m[a \mapsto v], \pi)\end{array}}{\sigma \vdash d, \texttt{write}\ a\ v \Rightarrow \sigma'}$$

3.4.2 Non-Transactional

Even when no transaction is active, reads and writes may only proceed when they do not conflict with transactional activity in other threads. As noted earlier, this means that LogTM is strongly atomic as defined by Blundell et al. [1].

A non-conflicting read in the base, non-transactional state yields a value from main memory.

$$\text{READ BASE}\ \frac{\begin{array}{c}\sigma = (m, \pi) \qquad \neg writeConflict(\pi, d, a) \\ \pi(d) = (unescaped, \langle\rangle, \_) \\ v = m(a)\end{array}}{\sigma \vdash d, \texttt{read}\ a \Rightarrow \sigma}$$

A non-conflicting write in the base, non-transactional state updates a value in main memory.

$$\textsc{write base}\ \frac{\sigma = (m,\pi) \qquad \neg \mathit{accessConflict}(\pi,d,a) \\ \pi(d) = (\mathit{unescaped}, \langle\rangle, \_) \\ \sigma' = (m[a \mapsto v], \pi)}{\sigma \vdash d, \texttt{write}\ a\ v \Rightarrow \sigma'}$$

3.4.3 Transactional

A non-conflicting read within a transaction gets a value from the nested transaction stack for the current thread. The thread's read set is updated accordingly.

$$\textsc{read trans}\ \frac{\sigma = (m,\pi) \qquad \neg \mathit{writeConflict}(\pi,d,a) \\ \pi(d) = (\mathit{unescaped}, (r,w,\vec{y},\vec{z}) \cdot \vec{t}, \vec{x}) \\ \sigma' = (m, \pi[d \mapsto (\mathit{unescaped}, (r \cup \{a\}, w, \vec{y}, \vec{z}), \vec{x})]) \\ v = \mathit{get}(a,m,\vec{t})}{\sigma \vdash d, \texttt{read}\ a \Rightarrow \sigma'}$$

A non-conflicting write within a transaction stores or updates the value in the running transaction's value map.

$$\textsc{write trans}\ \frac{\sigma = (m,\pi) \qquad \neg \mathit{accessConflict}(\pi,d,a) \\ \pi(d) = (\mathit{unescaped}, (r,w,\vec{y},\vec{z}) \cdot \vec{t}, \vec{x}) \\ \sigma' = (m, \pi[d \mapsto (\mathit{unescaped}, (r, w[a \mapsto v], \vec{y}, \vec{z}) \cdot \vec{t}, \vec{x})])}{\sigma \vdash d, \texttt{write}\ a\ v \Rightarrow \sigma'}$$

3.5 Handler Registration

Unescapes and open commits may specify commit and/or abort handlers to run; these must be merged with those specified by sibling operations. Let *addHandler* : $\overrightarrow{\mathit{Instructions}} \times \mathit{Instructions} \times \overrightarrow{\mathit{Instructions}} \times \mathit{Instructions} \times \overrightarrow{\mathit{Instructions}} \rightarrow \overrightarrow{\mathit{Instructions}}$ be the handler combining function. Informally, the arguments to *addHandler* represent:

1. an existing instruction stream to run before the new handler
2. a handler prologue
3. a handler body
4. a handler epilogue

5. an existing instruction stream to run after the new handler

The result is a new instruction stream with all argument instructions sequenced in argument order. However, if the handler body is empty, then the prologue and epilogue are ignored as well and the first and last arguments are simply concatenated.

$$\text{ADD HANDLER EMPTY}\quad \frac{}{\mathit{addHandler}(\vec{x}_1, \_, \langle\rangle, \_, \vec{x}_5) = \vec{x}_1 :: \vec{x}_5}$$

$$\text{ADD HANDLER NONEMPTY}\quad \frac{\vec{x}_3 = \_ \cdot \_}{\mathit{addHandler}(\vec{x}_1, x_2, \vec{x}_3, x_4, \vec{x}_5) = \vec{x}_1 :: \langle x_2\rangle :: \vec{x}_3 :: \langle x_4\rangle :: \vec{x}_5}$$

LogTM commit handlers run in FIFO order; we therefore use *addHandler* with $\vec{x}_5 = \langle\rangle$ so that a new commit handler $\vec{x}_3$ is appended after any existing commit handlers $\vec{x}_1$. Conversely, LogTM abort handlers run in LIFO order; we therefore use *addHandler* with $\vec{x}_1 = \langle\rangle$ so that a new commit handler $\vec{x}_3$ is prepended before any existing abort handlers $\vec{x}_5$.

3.6 Transactions

The special `setState _ _` instruction replaces the current thread's escaped state and transaction stack. This instruction is used to restore earlier configurations before running abort handlers.

$$\text{SET STATE}\quad \frac{\begin{array}{c}\sigma = (m, \pi) \qquad \pi(d) = (\_, \_, \vec{x}) \\ \sigma' = (m, \pi[d \mapsto (e, \vec{t}, \vec{x})])\end{array}}{\sigma \vdash d, \texttt{setState}\; e\, \vec{t} \Rightarrow \sigma'}$$

Beginning a transaction materializes a new, empty read set and value map and pushes these onto the current thread's nested transaction stack. The commit handler is empty. The abort handler restores the thread state at the time of the begin and then restarts at that begin point. However, these are initial values only: both handlers can change during execution. Notice that we don't care whether the transaction is to be closed or open at this point. That only matters when we commit.

$$\text{BEGIN}\quad \frac{\begin{array}{c}\sigma = (m, \pi) \qquad \pi(d) = (\mathit{unescaped}, \vec{t}, \vec{x}) \\ t = (\emptyset, \emptyset, \langle\rangle, \texttt{setState}\; \mathit{unescaped}\, \vec{t} \cdot \texttt{begin} \cdot \vec{x}) \\ \sigma' = (m, \pi[d \mapsto (\mathit{unescaped}, t \cdot \vec{t}, \vec{x})])\end{array}}{\sigma \vdash d, \texttt{begin} \Rightarrow \sigma'}$$

Upon an abort, all instructions that would have followed in the normal execution sequence are discarded. Instead, the aborting transaction is popped and discarded. The normal instruction stream for the running thread is likewise discarded. Instead, execution continues with the aborting transaction's abort handler.

$$\text{ABORT}\ \frac{\sigma = (m,\pi) \qquad \pi(d) = (\mathit{unescaped}, (\_,\_,\_,\vec{z}) \cdot \vec{t}, \_) \\ \sigma' = (m, \pi[d \mapsto (\mathit{unescaped}, \vec{t}, \vec{z})])}{\sigma \vdash d, \texttt{abort} \Rightarrow \sigma'}$$

Observe that transaction rollback for aborts is atomic with respect to activity in all other threads. This is a substantial deviation from the LogTM implementation, in which rollback can be interleaved with execution by other threads. We claim, but have not proved, that no X3-compliant program can distinguish between atomic and non-atomic rollback. Informally, a program can only tell the difference if a different thread observes the aborting thread's uncommitted writes while they are being rolled back. However, Condition X3 forbids this sort of "peeking" by escaped reads, while normal read/write conflict detection forbids peeking by unescaped reads.

A closed commit with no enclosing transaction applies its stored writes to main memory. The committing transaction is then popped and discarded. Execution continues with the committing transaction's commit handler followed by the rest of the thread's normal instruction stream. Notice that this means that commit handlers execute *after* dropping isolation: while commit handlers are running, both they and other threads can observe stores performed by the committed transaction.

$$\text{COMMIT TOP CLOSED}\ \frac{\sigma = (m,\pi) \qquad \pi(d) = (\mathit{unescaped}, \langle(\_, w, \vec{y}, \_)\rangle, \vec{x}) \\ \sigma' = ((w;m), \pi[d \mapsto (\mathit{unescaped}, \langle\rangle, \vec{y} :: \vec{x})])}{\sigma \vdash d, \texttt{commitClosed} \Rightarrow \sigma'}$$

An open commit with no enclosing transaction is similar, except that the instruction-provided commit handler is run as an open transaction after the transaction's accumulated commit handlers but before continuing the rest of the thread's normal instruction stream.

$$\text{COMMIT TOP OPEN}\ \frac{\sigma = (m,\pi) \qquad \pi(d) = (\mathit{unescaped}, \langle(r, w, \vec{y}', \_)\rangle, \vec{x}) \\ \vec{y}'' = \mathit{addHandler}(\langle\rangle, \texttt{begin}, \vec{y}, \texttt{commitOpen}\ \langle\rangle\ \langle\rangle, \langle\rangle) \\ \sigma' = ((w;m), \pi[d \mapsto (\mathit{unescaped}, \langle\rangle, \vec{y}' :: \vec{y}'' :: \vec{x})])}{\sigma \vdash d, \texttt{commitOpen}\ \vec{y}\ \_ \Rightarrow \sigma'}$$

For a nested closed commit, the innermost read set, value map, commit handler, and abort handler are merged into those of the parent transaction. The committing transaction is then popped and discarded.

$$\text{COMMIT DEEP CLOSED}\quad \frac{\begin{array}{c}\sigma = (m,\pi) \qquad \pi(d) = (\mathit{unescaped}, \vec{t}, \vec{x}) \\ \vec{t} = t_1 \cdot t_2 \cdot \vec{t}' \quad t_1 = (r_1, w_1, \vec{y}_1, \vec{z}_1) \quad t_2 = (r_2, w_2, \vec{y}_2, \vec{z}_2) \\ t_2' = (r_1 \cup r_2, (w_1; w_2), \vec{y}_1 :: \vec{y}_2, \vec{z}_1 :: \vec{z}_2) \\ \sigma' = (m, \pi[d \mapsto (\mathit{unescaped}, t_2' \cdot \vec{t}', \vec{x})])\end{array}}{\sigma \vdash d, \texttt{commitClosed} \Rightarrow \sigma'}$$

Let *obeyO1* $\subseteq \overrightarrow{\mathit{TransStates}} \times \mathit{Addresses}$ be the Condition O1 compliance relation for a potential write, defined as follows:

$$\mathit{obeyO1}(\langle\rangle, a) \tag{23}$$

$$\mathit{obeyO1}(\_ \cdot \vec{t}, a) \iff \mathit{addressBlock}(a) \notin \mathit{addressBlocks}(\mathit{allWritten}(\vec{t})) \tag{24}$$

Note that *obeyO1* and *obeyX1* are identical, though they are used in different contexts.

For a nested open commit, writes held by the innermost value map are applied directly to main memory; all such writes must be O1-compliant. The committing transaction is then popped and discarded. Commit and abort handlers given in the commit instruction are registered with the parent transaction. If subsequently needed, these handlers will be run as open transactions. For the abort handler (but not the commit handler) we will restore the state of the thread to that it had just before this commit. Following the commit instruction we first execute any accumulated commit handlers and then proceed with the normal instruction stream.

$$\text{COMMIT DEEP OPEN}\quad \frac{\begin{array}{c}\sigma = (m,\pi) \qquad \pi(d) = (\mathit{unescaped}, \vec{t}, \vec{x}) \\ \vec{t} = t_1 \cdot t_2 \cdot \vec{t}' \qquad t_1 = (\_, w_1, \vec{y}_1, \_) \qquad t_2 = (r_2, w_2, \vec{y}_2, \vec{z}_2) \\ \vec{y}_2' = \mathit{addHandler}(\vec{y}_2, \texttt{begin}, \vec{y}, \texttt{commitOpen}\ \langle\rangle\ \langle\rangle, \langle\rangle) \\ \vec{z}_2' = \mathit{addHandler}(\langle\rangle, \texttt{setState}\ \mathit{unescaped}\ \vec{t}, \vec{z}, \texttt{commitOpen}\ \langle\rangle\ \langle\rangle, \vec{z}_2) \\ t_2' = (r_2, w_2, \vec{y}_2', \vec{z}_2') \\ \forall a \in \mathit{domain}(w_1). \mathit{obeyO1}(\vec{t}, a) \\ \sigma' = ((w_1; m), \pi[d \mapsto (\mathit{unescaped}, t_2' \cdot \vec{t}', \vec{y}_1 :: \vec{x})])\end{array}}{\sigma \vdash d, \texttt{commitOpen}\ \vec{y}\ \vec{z} \Rightarrow \sigma'}$$

If transactions were marked as open or closed when begun, then O1-compliance could be checked as each write instruction was performed. In the model used by this semantics, a transaction is only designated as open or closed at commit time. Therefore, O1-compliance checking must be postponed until open commit.

We intentionally do not state the semantics of open writes that violate Condition O1. However, any given implementation may choose to extend these semantics by defining the behavior of such writes.

3.7 Escape Actions

The hardware is not responsible for counting nested escape actions. To emphasize this, we leave the semantics of a double escape or double unescape undefined.

$$\textsc{Escape}\;\frac{\begin{array}{c}\sigma = (m,\pi) \qquad \pi(d) = (\mathit{unescaped},\vec{t},\vec{x}) \\ \sigma' = (m,\pi[d \mapsto (\mathit{escaped},\vec{t},\vec{x})])\end{array}}{\sigma \vdash d, \texttt{escape} \Rightarrow \sigma'}$$

Unescaping may require additional processing of commit and/or abort handlers. When unescaping inside a transaction, handlers are queued up to be used later depending on the outcome of that transaction.

$$\textsc{Unescape Trans}\;\frac{\begin{array}{c}\sigma = (m,\pi) \qquad\qquad \pi(d) = (\mathit{escaped},\vec{t},\vec{x}) \\ \vec{t} = t_1 \cdot \vec{t}' \qquad t_1 = (r_1, w_1, \vec{y}_1, \vec{z}_1) \qquad t_1' = (r_1, w_1, \vec{y}_1', \vec{z}_1') \\ \vec{y}_1' = \mathit{addHandler}(\vec{y}_1, \texttt{escape}, \vec{y}, \texttt{unescape}\ \langle\rangle\ \langle\rangle, \langle\rangle) \\ \vec{z}_1' = \mathit{addHandler}(\langle\rangle, \texttt{setState}\ \mathit{escaped}\ \vec{t}, \vec{z}, \texttt{unescape}\ \langle\rangle\ \langle\rangle, \vec{z}_1) \\ \sigma' = (m,\pi[d \mapsto (\mathit{escaped}, t_1' \cdot \vec{t}', \vec{x})])\end{array}}{\sigma \vdash d, \texttt{unescape}\ \vec{y}\ \vec{z}}$$

When unescaping in the base, non-transactional state, the abort handler is discarded and the commit handler is run immediately.

$$\textsc{Unescape Base}\;\frac{\begin{array}{c}\sigma = (m,\pi) \qquad\qquad \pi(d) = (\mathit{escaped},\langle\rangle,\vec{x}) \\ \vec{y}' = \mathit{addHandler}(\langle\rangle, \texttt{escape}, \vec{y}, \texttt{unescape}\ \langle\rangle\ \langle\rangle, \langle\rangle) \\ \sigma' = (m,\pi[d \mapsto (\mathit{escaped}, \langle\rangle, \vec{y}' :: \vec{x})])\end{array}}{\sigma \vdash d, \texttt{unescape}\ \vec{y}\ \_}$$

4 Acknowledgment

This document is the result of intensive discussion with Jayaram Bobba, Mark D. Hill, Kevin E. Moore, Michelle J. Moravan, Cindy Rubio Gonzalez, Michael M. Swift, David A. Wood, and Luke Yen of the University of Wisconsin–Madison.

References

[1] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Proceedings of the Fourth Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[2] Keven E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture*, Austin, Texas, February 11–15 2006. Institute of Electrical and Electronics Engineers.

[3] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 21–25 2006. Association for Computing Machinery.

Index