Communicating Memory Transactions

Mohsen Lesani Jens Palsberg

UCLA Computer Science Department
University of California, Los Angeles, USA
{lesani, palsberg}@cs.ucla.edu

Abstract

Many concurrent programming models enable both transactional memory and message passing. For such models, researchers have built increasingly efficient implementations and defined reasonable correctness criteria, while it remains an open problem to obtain the best of both worlds. We present a programming model that is the first to have opaque transactions, safe asynchronous message passing, and an efficient implementation. Our semantics uses tentative message passing and keeps track of dependencies to enable undo of message passing in case a transaction aborts. We can program communication idioms such as barrier and rendezvous that do not deadlock when used in an atomic block. Our experiments show that our model adds little overhead to pure transactions, and that it is significantly more efficient than Transactional Events. We use a novel definition of safe message passing that may be of independent interest.

Categories and Subject Descriptors D.1 [**Programming Techniques**]: D1.3 Concurrent Programming – Parallel programming

General Terms Languages, Design, Algorithms

Keywords Transactional Memory, Actor

1. Introduction

1.1. Background

Multi-cores are becoming the mainstream of computer architecture, and they require parallel software to maximize performance. Therefore, researchers sense the need for effective concurrent programming models more than ever before. We expect a concurrent programming model to provide means for both isolation and communication: concurrent operations on shared memory should be executed in isolation to preserve consistency of data, while threads also need to communicate to coordinate cooperative tasks. The classical means of programming isolation and communication is locks and condition variables [16]. Locks protect memory by enforcing that the memory accesses of blocks of code are isolated from each other by mutual exclusion. Condition variables allow threads to communicate: a thread can wait for a condition on shared memory locations and the thread that satisfies the condition can notify waiting threads. However, development and maintenance of concurrent data structures by fine-grained locks is notoriously hard and error-prone, and lock-based abstractions do not lend themselves well to composition. We need a higher level of abstraction.

A promising isolation mechanism to replace locks is memory transactions because they are easy to program, reason about, and compose [12]. The idea is to mark blocks of code as atomic and let the runtime system guarantee that these blocks are executed in isolation from each other. Researchers have developed several implementations [5][13], semantics [1][24][15], and correctness criteria [10][24] for memory transactions. In particular, we prefer to work with memory transactions that satisfy a widely recognized correctness criterion called opacity [10]. To complement memory transactions, which communication mechanism should replace condition variables? We want the addition of a communication mechanism to preserve opacity while adding little implementation overhead to pure transactions. Let us review the strengths and weaknesses of several known mechanisms.

1.2. Synchronizers, retry, and punctuation

Luchango and Marathe were the first to consider the interaction of memory transactions and they introduced synchronizers. A synchronizer is a shared data structure that can be accessed simultaneously by every transaction that requests access to it and hence additional concurrency control mechanisms are needed to protect the shared data against race conditions [20]. The transactions that synchronize on a synchronizer either all commit or all abort. The work is recently extended to transaction communicators [22].

To enable a transaction to wait for a condition, Harris and Fraser introduced guarded atomic blocks [11], and Haskell added the "retry" keyword [12]. On executing "retry", Haskell aborts and then retries the transaction. Later, Smaragdakis et al. [28] established the need for transactional communication. They showed that neither of the previous mechanisms supports programming of a composable barrier abstraction: if used in an atomic block, the barrier deadlocks. In contrast to Haskell's "retry", Smaragdakis et al. [28] and also Dudnik and Swift [7] advocated that the waiting transaction should be committed rather than aborted. They observed that if the transaction is aborted, all its writes are discarded, while if it is committed, its writes will be visible to other transactions, thereby enabling the transaction to leave information for other transactions before it starts waiting.

Dudnik and Swift used their observation as the basis for designing transactional condition variables [7]; their model allows no nesting of atomic blocks. Smaragdakis et al. [28] used their observation as the basis for designing TIC which enables programming of a barrier abstraction that won't deadlock even if it is used in an atomic block. TIC splits (“punctuates”) each transaction into two transactions; this may violate local invariants and therefore requires the programmer to provide code for reestablishing the local invariants. TIC executes that code at the point of the split, that is, after wait is called and before the first

PPoPP'11 February 12–16, 2011, San Antonio, Texas, USA.

half of the transaction is committed. As explained in [28], TIC breaks isolation and therefore doesn't satisfy opacity.

1.3. Message Passing

A dual approach to providing means for isolation and communication is to begin with a message passing model such as Actors [2] and Concurrent ML (CML) [26], and then add an isolation mechanism. Examples of such combinations include Stabilizers [30], Transactional Events (TE) [6], and Transactional Events for ML (TE for ML) [8].

In Stabilizers, threads can communicate by sending and receiving synchronous messages on channels. The programmer can mark locations of code as stable checkpoints. If a thread encounters a transient fault, it calls "stabilize", which causes the run-time system to revert back the current thread, and all threads with which it has transitively communicated, to their latest possible stable checkpoints. In summary, Stabilizers support program location recovery but not atomicity and isolation as explained in [30].

Inspired by CML and Haskell STM, TE provides the programmer with a sequencing combinator to combine two events such as synchronous sends and receives into one compound event. The combination is an all-or-nothing transaction in the sense that executing the resulting event performs either both or none of the two events. The sequencing combinator enables straightforward programming of: (1) a modular abstraction of guarded (conditional) receive (this is not possible in CML), (2) three-way-rendezvous (a generalization of barrier) (this is not possible with pure memory transactions [6]), and (3) memory transactions (by representing each location as a server thread). TE supports the completeness property, namely: if there exists an interleaving for a set of compound events such that their sends and receives are matched to each other, the interleaving is guaranteed to be found. While the completeness property can terminate some scheduler-dependent programs, scheduler-independence is the well known property expected from concurrent algorithms. More importantly, finding such an interleaving is NP-hard [6] and can be implemented with an exponential number of run-time search threads [6]. Our experiments show that the performance penalty can be excessive.

TE supports all-or-nothing compound events but it prevents any shared memory mutation inside compound events. In follow-up work on TE, the authors of TE in ML [8] explain that encoding memory as a ref server is inefficient. They extend TE to support mutation of shared memory in compound events. TE for ML logically divides a compound event into sections called chunks. Chunks are delimited by the sends and receives of the compound event. The semantics of TE for ML breaks the isolation of shared memory mutations of a compound event at the end of its chunks. At these points (i.e. before sends and receives), the shared memory mutations that are done in the chunk can be seen by chunks of other synchronizing events. Similar to the punctuation in TIC, chunking breaks isolation and thus doesn't satisfy opacity.

1.4. Our Approach

The above review shows that previous work has problems with either nesting of atomic blocks, opacity, or efficiency. Our goal is to do better. In this paper, we present Communicating Memory Transactions (CMT) that integrates memory transactions with a style of asynchronous communication known from the Actor model. CMT is the first model to have opaque transactions, safe asynchronous message passing, and an efficient implementation. We use a novel definition of safety for asynchronous message passing that generalizes previous work. Safe communication means that every committed transaction has received messages only from committed transactions. To satisfy communication safety, CMT keeps track of dependencies to enable undo of message passing in case a transaction aborts. We show how to program three fundamental communication abstractions in CMT, namely synchronous queue, barrier, and three-way rendezvous. In particular we show that our barrier and rendezvous abstractions do not deadlock when used in an atomic block. To enable an efficient implementation, CMT does not satisfy the completeness property [8] found in TE. Based on the transactional memory implementations TL2 [5] and DSTM2 [13], we present two efficient implementations of CMT. We will explain several subtle techniques that we use to implement the semantics. Our experiments show that our model adds little overhead to pure transactions, and that it is significantly more efficient than Transactional Events.

In Section 2 we discuss five CMT programs. In Section 3 we recall the optimistic semantics of memory transactions by Koskinen, Parkinson, and Herlihy [15], and in Section 4 we give a semantics of CMT as an extension of the semantics in Section 3. In Section 5 we explain our implementation of CMT, and in Section 6 we show our experimental results.

2. Examples

The goal of this section is to give examples of CMT programs and give an informal discussion of the semantics of CMT. In particular, we will illustrate the notions of communication safety, dependency and collective commit.

We use the following syntax: to delimit parallel sections of the program, || is used. $ch\ send\ e$ sends the result of expression e to channel ch. $x \coloneqq ch\ receive$ receives a message from channel ch and assigns it to the thread local variable x. To provide means of programming abstractions, macro definitions are allowed: $let\ macroName(params)\ t$. The body term t of the macro is inlined with $params$ at the call sites.

Let us start with a simple example: a server thread that executes a transaction in response to request messages from a client thread.

```
{ // Client                 | { // Server
  atomic                    |   atomic
    ch send unit            |     x := ch receive
} ||                        | }
```

The server transaction receives the tentative message from the client transaction and mutates memory according to the message. If the client transaction aborts, the message that it has sent is invalid. Therefore, the server transaction should commit only if the client transaction is committed. In other words, the communication is safe under the condition that a receiving transaction is committed only if the sender transaction is committed. We say that the receiving transaction depends on the sender transaction. If the sender aborts the receiver should abort as well. The abortion is propagated to depending transactions. If a receive is executed on a channel that is empty or contains an invalid message (a message sent by an aborted transaction), the receive suspends until a message becomes available.

Consider the two-way rendezvous abstraction that can swap values between two threads. (Rendezvous is a generalization of barrier that swaps values in addition to time synchronization.)

```
let rendezvous(sc1, rc1, sc2, rc2)  | let swap(x, sc, rc, v)
  atomic                            |   sc send v;
    x1 := sc1 receive;              |   x := rc receive
    x2 := sc2 receive;              |
    rc1 send x2;                    |
```

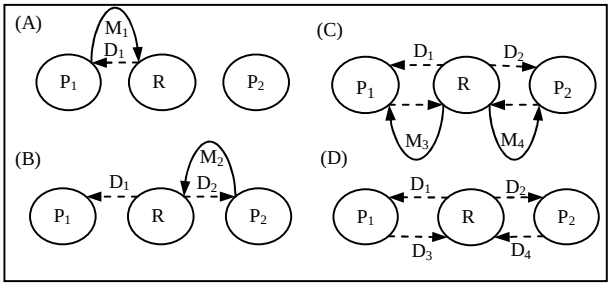


Figure 1. Interactions of 2-way Rendezvous

$rc_2\ send\ x_1$		

Consider the following program that employs the above abstractions. Each abstraction is inlined at its call sites and its parameters are substituted with passed arguments. Columns represent parallel parts of the program. (To discuss the interaction of transactions, the parties call *swap* inside atomic blocks.)

{ // $Party_1$	{ // *Rendezvous*	{ // $Party_2$
atomic	*rendezvous*(	*atomic*
// *code before*	$ch_1, ch_2,$	// *code before*
swap(	ch_3, ch_4)	*swap*(
$x, ch_1, ch_2, unit$);	} ||	$x, ch_3, ch_4, unit$);
// *code after*		// *code after*
} ||		}

Figure 1 shows the steps of execution of the above program. Solid arrows show messages and dashed arrows show dependencies. $Party_1$ sends a tentative $Message_1$ to Rendezvous. Rendezvous receives $Message_1$ and becomes dependent on $Party_1$ (Figure 1.A). The same happens for $Party_2$ (Figure 1.B). At this point, Rendezvous is dependent on both parties.

Assume that $Party_2$ aborts. The abortion is propagated to Rendezvous by $Dependency_2$. Rendezvous is also aborted and retried. On the retry, it receives $Message_1$ again. But as $Message_2$ is invalid, the second receive suspends. This means that Rendezvous repeats Figure 1.A again. It effectively ignores the aborted transaction of $Party_2$ and waits for another.

When $Party_2$ is retried, Figure 1.B is repeated. At this time, Rendezvous has received request messages from both parties. It tentatively sends swapped messages back to both $Party_1$ and $Party_2$. The parties are released from suspension and receive the messages. They get dependent on Rendezvous (Figure 1.C). At this time, parties and Rendezvous are interdependent (Figure 1.D).

Assume that $Party_2$ aborts in the code after *swap*. Dependencies propagate abortion to Rendezvous and then to $Party_1$. In other words, if one of the parties aborts, the Rendezvous and all the other parties are aborted and retried. This is the expected behavior: as $Party_2$ is aborted, the value that it has swapped with $Party_1$ is invalid. Therefore, $Party_1$ should be aborted as well. (This also matches the semantics expected from the barrier. As $Party_2$ aborts, it is retried. This means that it will reach the barrier again. By the semantics of the barrier, no party should pass the barrier when there is a party that has not reached the barrier. Thus, as $Party_2$ will reach the barrier, $Party_1$ should not have passed it. Therefore, $Party_1$ should be aborted as well.)

Finally, the transactions of Rendezvous, $Party_1$ and $Party_2$ reach the end of the atomic blocks. As they are interdependent, each of them can be committed only if the others are committed. If each of them obliviously waits until its dependencies are resolved, deadlock happens. As will be explained in the following sections, interdependent transactions are recognized as a cluster and transactions of a cluster are collectively committed.

In contrast to an implementation using Haskell retry, calling *swap* inside a nested atomic block does not lead to a deadlock. In addition, in contrast to TIC and TE for ML, opacity of transactions is satisfied.

Synchronous Queue	**Barrier**	**3-way Rendezvous**
The abstractions:	The abstractions:	The abstractions:
let syncSend(	*let barrier*(	*let rendezvous*(
sc, rc, v)	$bc_1, pc_1,$	$sc_1, rc_1,$
sc send v;	bc_2, pc_2)	$sc_2, rc_2,$
rc receive	*atomic*	sc_3, rc_3)
let syncReceive(	bc_1 *receive*;	*atomic*
x, sc, rc)	bc_2 *receive*;	$x_1 :=$ sc_1 *receive*;
$x :=$ *sc receive*;	pc_1 *send unit*;	$x_2 :=$ sc_2 *receive*;
rc send unit	pc_2 *send unit*	$x_3 :=$ sc_3 *receive*;
	let await(bc, pc)	rc_1 *send* $\langle x_2, x_3 \rangle$;
The program:	*bc send unit*;	rc_2 *send* $\langle x_1, x_3 \rangle$;
{ // *Sender*	*pc receive*	rc_3 *send* $\langle x_1, x_2 \rangle$;
syncSend(		*let swap*(x, sc, rc, v)
x, ch_0, ch_1)	The program:	*sc send v*;
} || { // *Receiver*	{ // *Barrier*	$x :=$ *rc receive*
syncReceive(	*barrier*(	
x, ch_0, ch_1)	$ch_1, ch_2,$	The program:
}	ch_3, ch_4)	{ // *Rendezvous*
	} || { // $Party_1$	*rendezvous*(
	await(ch_1, ch_2)	$ch_1, ch_2,$
	} || { // $Party_2$	$ch_3, ch_4,$
	await(ch_3, ch_4)	ch_5, ch_6)
	}	} || { // $Party_1$
		swap(
		$x, ch_1, ch_2, unit$)
		} || { // $Party_2$
		swap(
		$x, ch_3, ch_4, unit$
		} || { // $Party_3$
		swap(
		$x, ch_5, ch_6, unit$)
		}

Figure 2: CMT Programs

Similar to Two-way Rendezvous, the abstractions for Synchronous queue, Barrier and Three-way rendezvous can be programmed in CMT as shown in Figure 2. Please note that it is assumed that these basic abstractions are used only once. For example, the basic barrier abstraction is not a cyclic barrier. For the three-way rendezvous, we assume that e can be pairs of the form $\langle e, e \rangle$. Implementations of Barrier with Haskell retry, TIC and TE for ML and an implementation of CMT can be seen in the technical report [17] section 15.1. Implementations of Synchronous Queue and Rendezvous can be seen in the technical report [17] sections 15.2 and 15.3.

3. Memory Transactions

We now recall the optimistic semantics of memory transactions by Koskinen, Parkinson, and Herlihy [15]. Their semantics is the starting point for our semantics of CMT in Section 4. TL2 is an implementation that realizes this semantics.

Note: we have fixed a few typos in the syntax, semantics and definition of moverness after personal communication with the authors of [15].

3.1. Syntax

A configuration is a triple of the form $\langle \mathcal{T} \cdot \sigma_{sh} \cdot \ell_{sh} \rangle$. (To simplify reading of long configurations, "·" is used to separate elements of configurations.) $\mathcal{T}$ represents the set of threads. σ_{sh} denotes the shared store that contains objects. ℓ_{sh} is a log of pairs $\langle \tau^{cmt}, \ell_\tau \rangle$: each committed transaction τ^{cmt} and the operations it has performed ℓ_τ. $\mathcal{T}$ is a set of elements of the form $\langle \tau, s, \sigma_\tau, \overline{\sigma_\tau}, \ell_\tau \rangle$. τ is

$OCMD$

$$\langle\langle \tau, c; s, \sigma_\tau, \overleftarrow{\sigma_\tau}, \ell_\tau\rangle, \mathcal{T}\cdot\sigma_{sh}\cdot\ell_{sh}\rangle \xrightarrow{\perp}_o \langle\langle \tau, s, [\![c]\!]\sigma_\tau, \overleftarrow{\sigma_\tau}, \ell_\tau\rangle, \mathcal{T}\cdot\sigma_{sh}\cdot\ell_{sh}\rangle \qquad \tau \neq \perp$$

$OBEG$

$$\langle\langle \perp, \text{beg}; s, \sigma_\tau, \overleftarrow{\sigma_\tau}, []\rangle, \mathcal{T}\cdot\sigma_{sh}\cdot\ell_{sh}\rangle \xrightarrow{\langle\tau,\text{beg}\rangle}_o \langle\langle \text{fresh}(\tau), s, \text{snap}(\sigma_\tau, \sigma_{sh}), \sigma_\tau[\text{stmt} \mapsto \text{"beg; } s\text{"}], []\rangle, \mathcal{T}\cdot\sigma_{sh}\cdot\ell_{sh}\rangle$$

$OAPP$

$$\langle\langle \tau, x \coloneqq o.m; s, \sigma_\tau, \overleftarrow{\sigma_\tau}, \ell_\tau\rangle, \mathcal{T}\cdot\sigma_{sh}\cdot\ell_{sh}\rangle \xrightarrow{\langle\tau,o.m\rangle}_o \langle\langle \tau, s, \sigma_\tau[o \mapsto ([\![o]\!]\sigma_\tau).m, x \mapsto rv(([\![o]\!]\sigma_\tau).m)], \overleftarrow{\sigma_\tau}, \ell_\tau :: (\text{"}o.m\text{"})\rangle, \mathcal{T}\cdot\sigma_{sh}\cdot\ell_{sh}\rangle \qquad \tau \neq \perp$$

$OCMT$

$$\frac{\forall\langle\tau^{cmt}, \ell_{\tau'}\rangle \in \ell_{sh} : \tau^{cmt} > \tau \Rightarrow \ell_\tau \bar{\triangleright} \ell_{\tau'}}{\langle\langle \tau, end; s, \sigma_\tau, \overleftarrow{\sigma_\tau}, \ell_\tau\rangle, \mathcal{T}\cdot\sigma_{sh}\cdot\ell_{sh}\rangle \xrightarrow{\langle\tau,\text{cmt}\rangle}_o \langle\langle \perp, s, \text{zap}(\sigma_\tau), \text{zap}(\sigma_\tau), []\rangle, \mathcal{T}\cdot\text{merge}(\sigma_{sh}, \ell_\tau)\cdot\ell_{sh} :: \langle\text{fresh}(\tau^{cmt}), \ell_\tau\rangle\rangle} \quad \tau \neq \perp$$

$OABT$

$$\langle\langle \tau, s, \sigma_\tau, \overleftarrow{\sigma_\tau}, \ell_\tau\rangle, \mathcal{T}\cdot\sigma_{sh}\cdot\ell_{sh}\rangle \xrightarrow{\langle\tau,\text{abt}\rangle}_o \langle\langle \perp, [\![\text{stmt}]\!]\overleftarrow{\sigma_\tau}, \overleftarrow{\sigma_\tau}/\text{stmt}, \overleftarrow{\sigma_\tau}, []\rangle, \mathcal{T}\cdot\sigma_{sh}\cdot\ell_{sh}\rangle \qquad \tau \neq \perp$$

$$\text{snap}(\sigma_\tau, \sigma_{sh}) = \sigma_\tau[o \mapsto [\![o]\!]\sigma_{sh}] \quad \forall o \in \text{objects}(P) \qquad \text{merge}(\sigma_{sh}, [\ ]) = \sigma_{sh}$$
$$\text{zap}(\sigma_\tau) = \sigma_\tau[o \mapsto \perp] \quad \forall o \in \text{objects}(P) \qquad \text{merge}(\sigma_{sh}, (\text{"}o.m\text{"}) :: \ell_\tau) = \text{merge}(\sigma_{sh}[o \mapsto ([\![o]\!]\sigma_{sh}).m], \ell_\tau)$$

Figure 3: Optimistic Semantics of Memory Transactions

the transaction identifier (or $\perp$ that denotes that the code is executing outside transactions). Transaction identifiers are assumed to be ordered by the time that they are generated. s is the statement to be executed by the thread. Statements have the following syntax:

$s \to i; s \mid i$ *Statement*

$i \to beg \mid end \mid x \coloneqq o.m \mid \text{c} \mid skip$ *Instruction*

beg and end denote the start and end of transactions. We use $atomic\ s$ as a syntactic sugar for $beg; s; end$. $o.m$ denotes calling method m on shared object o. Commands (reading and writing) that are applied to thread-local state are represented by c. σ_τ is the transaction-local store of objects. $\overleftarrow{\sigma_\tau}$ is the backup store that stores states of (thread-local) objects before the transaction is started. It is used to recover state when the transaction aborts. The statement before the transaction is started is also backed up in $\overleftarrow{\sigma_\tau}$. The pattern $\overleftarrow{\sigma_\tau}; stmt \mapsto s$ denotes a back up store that maps the backed up statement to s. ℓ_τ is the ordered log of operations that has been performed by the transaction. The initial configuration is of the form $Conf_0^o = \langle\mathcal{T}_0 \cdot \sigma_{sh_0} \cdot []\rangle$. $\mathcal{T}_0$ is $\mathcal{T}_0 = \{T_1, \ldots, T_n\}$ where $T_i = \langle\perp, P_i, \sigma_{sh_0}, \sigma_{sh_0}; stmt \mapsto \perp, []\rangle$. $\{P_{i=1..n}\}$ are the parallel segments of the program. σ_{sh_0} is the store where every object is mapped to its initial state i.e. $\sigma_{sh_0} = \{\forall o \in objects(P).\, o \mapsto init(o)\}$. $M[k \mapsto v]$ denotes assigning value v to key k in map M. $[\![k]\!]M$ represents value of key k in map M.

3.2. Operational Semantics

The semantics [15] is shown in Figure 3. The semantics is a labeled transition system. The syntax supports nested atomic blocks. We can transform a program with nested atomics into an equivalent program with only top-level atomics by simply removing all inner atomics. Hence, it is sufficient that the semantics supports only top-level atomics.

We will now explain the five rules in Figure 3. The $OCMD$ rule applies the statement to the local store. $[\![c]\!]\sigma_\tau$ denotes application of the command c to the local store σ_τ. The $OBEG$ rule starts a new transaction. A $fresh$ transaction identifier is generated. $fresh(\tau)$ generates unique and increasing transaction identifiers τ. The current store and also the current statement are stored in the backup store. A snapshot of the current state of objects is taken from the shared store to the local store. The $OAPP$ rule executes a method. The method is applied to the local store and is logged in the local log. rv represents the returned value. As defined by [15], read and write operations on memory locations are special cases of method call. The $OCMT$ rule checks that the methods of the current transaction are right movers with respect to the methods of the transactions that have been committed since the current transaction has started. Right moverness ensures that tentative execution of a transaction can be committed even though other transactions have committed after it started. Please refer to the appendix for a detailed definition of right moverness. If the methods of the transaction satisfy the moverness condition, the transaction is committed. The methods of the local log are applied to the shared store. The local log is also saved with a $fresh$ id in the shared log. This is used to check moverness while later transactions are committing. The $OABT$ reduction aborts the transaction. The store and the statement that were saved in the backup store when the transaction was starting are restored.

3.3. Properties

The semantics satisfies opacity which is a correctness condition for memory transactions [10]. We say that a sequence of labels $l_1, \ldots l_n$ is given by $\to_o$ started from $Conf_1^o$ if there are configurations $Conf_{i=2..n}^o$ such that for each $i \in \{1..n-1\}$: $Conf_{i=1}^o \xrightarrow{l_i}_o Conf_{i=n+1}^o$.

THEOREM 1 (Opacity). Every sequence of labels $\langle\tau, \text{beg}\rangle$, $\langle\tau, o.m\rangle$, $\langle\tau, \text{abt}\rangle$ and $\langle\tau, \text{cmt}\rangle$ given by $\to_o$ started from $Conf_0^o$ is opaque (Proposition 6.2 of [15]).

4. Communicating Memory Transactions

We now present the syntax and semantics of CMT. The semantics adds a core message passing mechanism to the semantics presented in the previous section.

4.1. Syntax

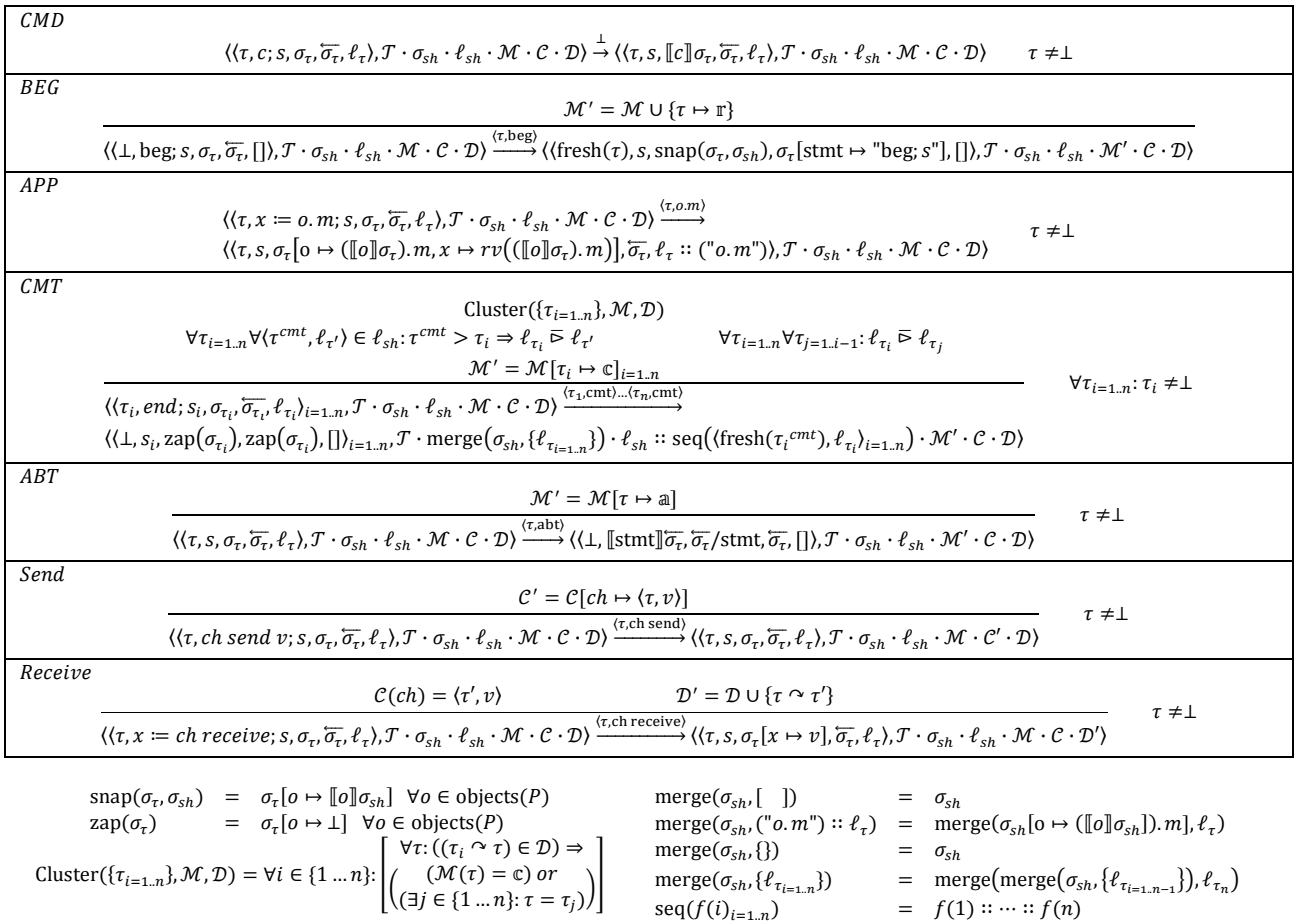
The syntax is extended as follows:

$s \to i; s \mid i$ *Statement*

$i \to beg \mid end \mid x \coloneqq o.m \mid c \mid skip$ *Instruction*
$\mid ch\ send\ e \mid x \coloneqq ch\ receive$

$ch\ send\ e$ sends the result of expression e to channel ch. $x \coloneqq ch\ receive$ receives a message from channel ch and assigns it to the thread local variable x. We assume that messages are primitive values.

The configuration of the semantics in section 3 is augmented with the following elements: $\mathcal{M}$, $\mathcal{C}$ and $\mathcal{D}$. Therefore a configuration is a tuple of the form $\langle\mathcal{T}\cdot\sigma_{sh}\cdot\ell_{sh}\cdot\mathcal{M}\cdot\mathcal{C}\cdot\mathcal{D}\rangle$. $\mathcal{M}$



$$\text{snap}(\sigma_\tau, \sigma_{sh}) = \sigma_\tau[o \mapsto [\![o]\!]\sigma_{sh}] \quad \forall o \in \text{objects}(P)$$
$$\text{zap}(\sigma_\tau) = \sigma_\tau[o \mapsto \perp] \quad \forall o \in \text{objects}(P)$$
$$\text{Cluster}(\{\tau_{i=1..n}\}, \mathcal{M}, \mathcal{D}) = \forall i \in \{1 \ldots n\}: \begin{bmatrix} \forall \tau: ((\tau_i \rightsquigarrow \tau) \in \mathcal{D}) \Rightarrow \\ \begin{pmatrix} (\mathcal{M}(\tau) = \mathbb{c})\ or \\ (\exists j \in \{1 \ldots n\}: \tau = \tau_j) \end{pmatrix} \end{bmatrix}$$

$$\begin{aligned} \text{merge}(\sigma_{sh}, [\ ]) &= \sigma_{sh} \\ \text{merge}(\sigma_{sh}, (\text{"}o.m\text{"}) :: \ell_\tau) &= \text{merge}(\sigma_{sh}[\text{o} \mapsto ([\![o]\!]\sigma_{sh}]).m], \ell_\tau) \\ \text{merge}(\sigma_{sh}, \{\}) &= \sigma_{sh} \\ \text{merge}(\sigma_{sh}, \{\ell_{\tau_{i=1..n}}\}) &= \text{merge}(\text{merge}(\sigma_{sh}, \{\ell_{\tau_{i=1..n-1}}\}), \ell_{\tau_n}) \\ \text{seq}(f(i)_{i=1..n}) &= f(1) :: \cdots :: f(n) \end{aligned}$$

Figure 4: CMT Semantics

maps each transaction id to the state of the transaction. The state of a transaction can be either $\mathbb{r}$ (running), $\mathbb{c}$ (committed), or $\mathbb{a}$ (aborted). A committed transaction has finished successfully, while an aborted transaction has stopped execution and had its tentative effects discarded. $\mathcal{C}$ is a partial function that maps channels ch to pairs of the form $\langle\tau, v\rangle$ where τ is the sender transaction and v is the current value of the channel. To guarantee communication safety, we track dependencies between transactions. $\mathcal{D}$ is the transaction dependency relation that is a set of elements of the form $\tau \rightsquigarrow \tau'$. Transaction τ is dependent on transaction τ', i.e. $\tau \rightsquigarrow \tau'$, if τ receives a message that is sent by τ'. The dependency to τ' is said to be resolved, if τ' is committed. The initial configuration is $Conf_0 = \langle\mathcal{T}_0 \cdot \sigma_{sh_0} \cdot [] \cdot \emptyset \cdot \emptyset \cdot \emptyset\rangle$. $\mathcal{T}_0$ and σ_{sh_0} are defined as the prior semantics.

4.2. Operational Semantics

The rules CMD, APP are not changed other than the addition of $\mathcal{M}\cdot\mathcal{C}\cdot\mathcal{D}$ to both sides of the rules. The two rules BEG and ABT have a small change. They set the state of the transaction in $\mathcal{M}$ to running $\mathbb{r}$ and aborted $\mathbb{a}$, respectively.

The $Send$ rule sends a message on a channel. The mapping $\mathcal{C}$ is updated to map the channel to the pair $\langle\tau, v\rangle$ where τ is the id of the current transaction and v is the sent value. The id of the sender transaction that is saved here is retrieved later when the message is received to record a dependency from the receiver to the sender. In CMT, each channel can hold a single value, while our implementation supports an arbitrary number of messages, as explained in section 5.

The $Receive$ rule receives a message from a channel. If there exists a value in the channel, the value is received and the dependency of the current transaction to the sender transaction is added to $\mathcal{D}$. The condition that the sender transaction is not aborted can be added as an optimization.

The semantics in Figure 4 supports transactions that can send and receive. It is straightforward to extend the semantics to allow code executing outside transactions to send and receive.

The CMT rule encodes the collective commitment of a cluster. A set of transactions are committed if they satisfy the following two conditions.

To respect dependencies, the first condition is that only transactions of clusters are committed where Cluster is defined as follows. A set of transactions that have reached the end of their atomic blocks (called terminated) is a cluster iff any unresolved dependencies of them are to each other. The transactions that are considered in the CMT rule have already reached the end of their atomic blocks. It is checked that their dependencies are either to other transactions of the set or to committed transactions.

It is notable why the following simple commitment condition is not used instead: a transaction that has reached the end of its atomic block is committed only if all its dependencies are already resolved. It is straightforward that this condition directly translates to communication safety. But it can lead to deadlock. For example, if two transactions receive messages from each other,

they are interdependent. As mentioned for the example of Section 2, if each transaction in a dependency cycle obliviously waits until its dependencies are resolved, it may wait forever. In classical distributed transactions [18][3], all receives happen at the beginning of sub-transactions. Therefore, the dependencies form a tree and hierarchical commit and two phase commit protocol (2PC) can be employed. In CMT receives can happen in the middle of transactions; thus, the dependencies can in general form a cyclic graph. A commitment condition is needed that guarantees communication safety and also allows commitment of transactions with cyclic dependencies. It is also notable that in contrast to edges in DB read-write dependence graphs [9] that represent serialization precedence of source to the sink transaction, edges in the message dependence graphs represent commit dependence of source to the sink transaction. The former cannot be cyclic but the latter can.

The second condition is the moverness of transactions of the cluster with respect to each other. In the basic commit rule, the moverness condition was that methods of the committing transaction are right movers with respect to methods of the recently committed transactions. In addition to that, as we commit a set of transactions, we need to check that there is an order of them where methods of each transaction in the order are right movers with respect to method of earlier transactions in the order. (Note that this order is not necessarily the causal order of sends and receives.) If the conditions are met, the local logs of the transactions are applied to the shared store, the local logs are stored in the shared log with $fresh$ ids, and the state of the transactions are set to committed $\mathbb{c}$ in $\mathcal{M}$.

4.3. Properties

4.3.1. Opacity

The semantics of Figure 4 extends the semantics of Figure 3 with communication semantics while preserving the opacity of transactions. This enables programmers to reason locally about the consistency of data in each atomic block.

THEOREM 2 (Opacity). Every sequence of labels $\langle \tau, \text{beg} \rangle$, $\langle \tau, o.m \rangle$, $\langle \tau, \text{abt} \rangle$ and $\langle \tau, \text{cmt} \rangle$ given by $\rightarrow$ started from $Conf_0$ is opaque.

High-level proof idea: Please refer to the technical report [17] section 10.1 for the formalization and the proof (29 pages). We reduce opacity for CMT to opacity for the semantics in Section 3. We show that for every sequence L of labels $\langle \tau, \text{beg} \rangle$, $\langle \tau, o.m \rangle$, $\langle \tau, \text{abt} \rangle$ and $\langle \tau, \text{cmt} \rangle$ that can be obtained from transitions of $\rightarrow$, there is a sequence of transitions of $\rightarrow_o$ that yield a sequence of labels L' that is the same as L other than addition of calls to a definite new object. By THEOREM 1, L' is opaque. We show that removing all calls to an object from a sequence of labels preserves opaqueness of the sequence. Therefore, as L' is opaque, L is opaque. ■

4.3.2. Communication Safety

Assume that a transaction τ_r receives a message m that is tentatively sent by another transaction τ_s. Receiving m and using its value is a part of the computation of τ_r. Therefore, validity of the computation of τ_r relies on validity of m. If τ_s finally aborts, m becomes invalid and τ_r should be prevented from committing. This means that the receiving transaction τ_r should not commit before the sending transaction τ_s is committed. The notion is formalized as the following correctness condition:

DEFINITION 1 (Communication). The communication relation for an execution is the set of receiver and sender transaction pairs in the execution.

Suppose $Exec = Conf_0 \xrightarrow{l_0} Conf_1 \xrightarrow{l_1} \dots \xrightarrow{l_{n-1}} Conf_n$. We define

$$Comm(Exec) = \{\, \tau_r \curvearrowright \tau_s \mid \exists i, j, ch\colon \; 0 \le i < j < n,\; l_i = (\tau_s, ch\ send), l_j = (\tau_r, ch\ receive),\; \forall k\colon (i < k < j) \Rightarrow \big(\forall \tau\colon l_k \neq (\tau, ch\ send)\big) \}$$

Intuitively, τ_s is the last sender on ch before τ_r receives.

DEFINITION 2 (Unsafe execution) A configuration $Conf_0$ can execute to an unsafe configuration iff there is an execution

$Exec = Conf_0 \xrightarrow{l_0} Conf_1 \xrightarrow{l_1} \dots \xrightarrow{l_{n-1}} Conf_n$, where

$Conf_n = \langle \mathcal{T}_n \cdot \sigma_{sh_n} \cdot \ell_{sh_n} \cdot \mathcal{M}_n \cdot \mathcal{C}_n \cdot \mathcal{D}_n \rangle$,

$\exists \tau_r, \tau_s\colon \; \mathcal{M}_n(\tau_r) = \mathbb{c}$

$\tau_r \curvearrowright \tau_s \in Comm(Exec)$

$\mathcal{M}_n(\tau_s) \neq \mathbb{c}$

THEOREM 3: Communication Safety: An initial configuration $Conf_0$ cannot execute to an unsafe configuration.

Please refer to the technical report [17] section 10.2 for the proof (16 pages).

High level proof idea: The first step is to prove $D_n = Comm(Exec)$ and thereby show that all members of $Comm(Exec)$ stem from the $Receive$ rule. Next we prove that when τ_r $receive$s a message from τ_s, τ_r is running, and we notice that the $Receive$ rule adds $\tau_r \curvearrowright \tau_s$ to $Comm(Exec)$. Later in the execution, τ_r may want to commit, and now the $\tau_r \curvearrowright \tau_s$ in $Comm(Exec)$ forces the CMT rule to ensure that τ_r only commits if either τ_s has already committed, or τ_r and τ_s commit together as members of the same cluster. ■

Our notion of communication safety generalizes a correctness criterion in [8]; let us explain why. Both TE and TE for ML support synchronous message passing. A high-level nondeterministic semantics "defines the set of correct transactions". In the high-level semantics, a set of starting transactions are stepped as follows: if there is a sequence of sub-steps that can match all the sends and receives of the transactions to each other, the transactions are committed together in single step. A low-level semantics is also defined that specifies stepping of the search threads that find the matching. It is proved that the low-level semantics complies with the high-level semantics. This essentially means that if a set of transactions are committed in the low-level semantics, each of them has communicated with transactions that are also committed at the same time. Our approach supports asynchronous messages. When a transaction sends a message, the message is enqueued in the recipient channel. Therefore, when a transaction is committing, there may not be matched receivers for the messages that it has sent but definite senders have sent the messages that it has received. Therefore, communication safety defines the condition that sender transactions are committed.

5. Implementation

We will now explain how we have implemented the calculus in Section 4 as the core functionality of a Scala [25] library called Transactors. Transactors integrate features of both memory transactions and actors. A transactor is an abstraction that consists of a thread and a channel that is called its mailbox. A mailbox is essentially a queue that can hold an arbitrary number of messages. Similar to the actor semantics [2], the messages in the mailbox are unordered. The thread of a transactor can perform the following operations both outside and inside transactions: reading from and

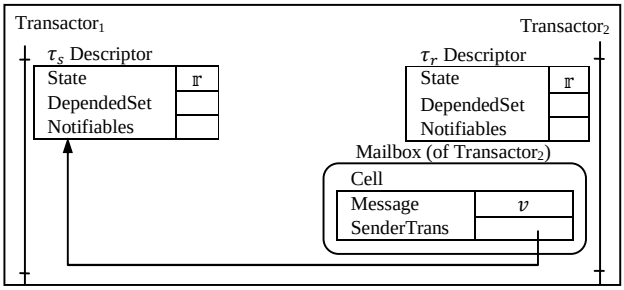


Figure 7. Sending

writing to shared memory and also sending messages to other transactors and receiving messages from its mailbox.

Recall that the starting point for Section 4 was Section 3 with its semantics of memory transactions. Similarly, the starting point for our implementation of the semantics in Section 4 is TL2, which implements Section 3's semantics of memory transactions. We explain how we have extended TL2 with an implementation of the new concepts in Section 4. In particular, we will explain about data structures that are built when messages are sent and received, the mechanism that notifies waiting transactions, cluster search and collective commit. (Our technique can work for other implementations of the semantics in Section 3 as well. In the technical report [17] section 13 we will explain how we have extended the implementation of DSTM2 in much the same way as we extended TL2. The pseudo codes of these two implementations can be found in the technical report [17] sections 12 and 14.)

In TL2, all memory locations are augmented with a lock that contains a version number. Transactions start by reading a global version-clock. Every read location is validated against this clock and added to the read-set. Written location-value pairs are added to the write-set. At commit, locks of locations in the write-set are acquired, the global version-clock is incremented and the read-set is validated. Then the memory locations are updated with the new global version-clock value and the locks are released.

In the implementation of transactors, the read and write procedures remain unchanged. As will be explained in subsection for the implementation of the CMT rule, we adapt the commit procedure to perform collective commitment of a cluster.

Each transaction has a descriptor that is a data structure that stores information regarding that transaction. This information includes the state of the transaction and a set that holds references to descriptors of depended transactions. Transactions change state as shown in Figure 5. Compared to the semantics in Section 4, the possible states of a transaction also include terminated. A transaction is terminated if it has reached the end of its atomic block and is not committed or aborted yet. The transaction descriptor also contains a set of notifiables and a message backup set that will be explained as we proceed.

In terms of $\mathcal{M}$ and $\mathcal{D}$ from the semantics, the descriptor of each transaction τ stores its state, $\mathcal{M}(\tau)$, and a set that holds references to descriptors of each τ' that $\tau \curvearrowright \tau' \in \mathcal{D}$. The mailboxes of transactors correspond to channels of $\mathcal{C}$. The semantics in Section 4 has seven rules. Two of those rules, CMD and APP, make no changes to the transaction map, channels, and dependencies. In the following five subsections we will explain how we implement

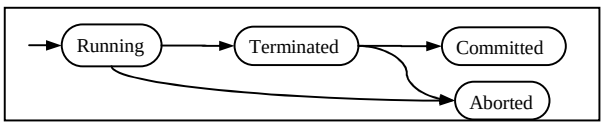


Figure 5. State transitions of a transaction

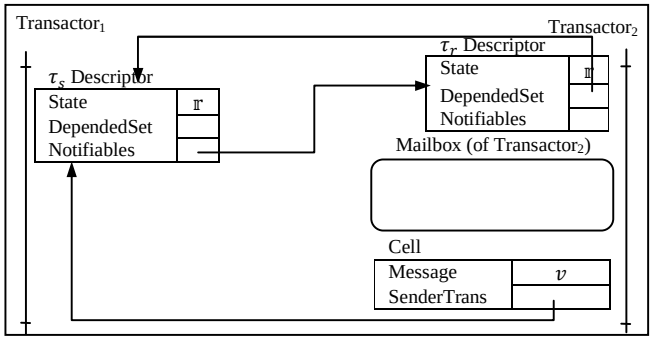


Figure 6. Receiving

the other five rules.

5.1. Starting a Transaction

We begin with the BEG rule. The rule changes M to $\mathcal{M}' = \mathcal{M} \cup \{\tau \mapsto \mathbb{r}\}$. When a transaction τ is started, a new transaction descriptor with the running state $\mathbb{r}$ is created and stored in a thread local variable. (Later, to get the descriptor of the current transaction, this thread local variable is checked. If the variable has no value, the execution is outside atomic blocks and otherwise, the value is the descriptor of the current transaction.) The global version-clock is read and the body of the atomic block is started.

5.2. Sending and Receiving a Message

Next we consider the $Send$ rule. The rule changes $\mathcal{C}$ to $\mathcal{C}' = \mathcal{C}[ch \mapsto (\tau, v)]$. When a message v is being sent, a new cell containing the message is enqueued to the mailbox. As the $Send$ rule defines, besides the message, the sender transaction τ saves a reference to the descriptor of itself in the new cell. If the recipient transactor has been suspended inside a transaction to receive a message, it is resumed. If the send is being executed outside transactions, a reference to a dummy transaction descriptor that is always committed is saved as the sender transaction in the cell and if the recipient transactor has been suspended to receive a message (inside or outside a transaction), it is desuspended. Figure 7 depicts relations of data structures while a message is being sent.

Next we consider the $Receive$ rule. The rule requires that $\mathcal{C}(ch) = (\tau_s, v)$ and changes $\mathcal{D}$ to $\mathcal{D}' = \mathcal{D} \cup \{\tau_r \curvearrowright \tau_s\}$. When a receive is being executed, cells of the mailbox are iterated. The reference to the descriptor of the sender transaction τ_s is obtained from each cell. The state of τ_s is read from its descriptor and the state of the message v of the cell is determined according to the state of τ_s. We use the terminology that (1) if the sender is committed, then the message is stable; and (2) if the sender is aborted, then the message is invalid. (3) if the sender is running or terminated, then the message is tentative. As any transaction that receives an invalid message should finally abort, invalid messages are dropped. This is the optimization that was mentioned for the $Receive$ rule. Thus, if the receive is being executed inside a transaction, a stable or tentative message is required to be taken from the mailbox. As executions that are outside transactions cannot be aborted, tentative messages can not be given to receives that are executed outside transactions. Therefore, a stable message is required for receives that are outside transactions. Cells are iterated and any invalid message is dropped until a required message is found. The thread suspends if a required message is not found until one becomes available. To track dependencies, if the found message is tentative, a reference to the descriptor of τ_s

is added to the depended set of the descriptor of the current transaction τ_r. The depended sets of descriptors constitute a dependency graph. We say that τ_1 is adjacent to τ_2 if the descriptor of τ_2 is in the depended set of the descriptor of τ_2.

Figure 6 depicts data structures and their relations while a message is being received. Assume that a transaction τ_s has sent a message v that is received by another transaction τ_r. Assume that τ_s is running and τ_r is being terminated. As τ_r has an unresolved dependency, it cannot be committed yet. Therefore, the thread running τ_r goes to the waiting state until τ_s aborts or commits. Hence, when τ_s is aborted or committed, it should notify τ_r. Notification is done by notifiables. When τ_r is receiving the tentative message v, the reference to the descriptor of τ_s is obtained from the cell that contains v and a reference to the descriptor of τ_r is subscribed to it as a notifiable. On abortion or commitment of a transaction (τ_s), all its registered notifiables are notified.

When a transaction aborts, its effects should be rolled back. The messages that it has received from its mailbox should be put back. Therefore, to track messages that are received inside a transaction, when a message is being received, the cell that the message is obtained from is added to a backup set in the transaction descriptor (not shown in the figures). The set is iterated when the transaction is being aborted and any cell that is not invalid is put back to the mailbox.

5.3. Abortion

Next, we consider the ABT rule. The rule changes $\mathcal{M}$ to $\mathcal{M}' = \mathcal{M}[\tau \mapsto \mathbb{a}]$. A transaction τ may deterministically abort as the result of resolution of a shared memory conflict. When τ is aborting, its state is set to aborted in its descriptor. Any cell of its backup set that is not invalid is put back to the mailbox. In addition, to wake up waiting transactions, τ propagates abortion to dependent transactions. Assume that $\{\tau_{r_{i=1..n}}\}$ is the set of transactions that are dependent on τ and $\{N_{i=1..n}\}$ is the set of notifiables that reference descriptors of $\{\tau_{r_{i=1..n}}\}$. τ notifies each N_i. The notification makes an abort event for τ_{r_i} if it is waiting. Finally, after notification, τ restarts its atomic block as a new transaction. On abortion of each τ_{r_i}, the same situation recurs, i.e. each of them notifies its own notifiables. Therefore, abortion of τ is propagated to transactions that are (transitively) dependent on τ. Note that by an implicit traversal of notifiable objects, abortion is propagated in the reverse direction of dependencies. The traversal avoids infinite loops by terminating at previously aborted transaction descriptors.

5.4. Termination and Commitment

Termination Every transaction that reaches the end of its atomic block sets the state of its descriptor to terminated. Then, the cluster search is started from the descriptor of the current transaction to check if it is possible to commit the transaction at this time. If the cluster search succeeds in finding a cluster, the transactions of the cluster are collectively committed and the atomic block returns successfully. Cluster search and collective commit are explained in the next subsection. If the cluster search cannot find a cluster at this time, the thread running the transaction goes to the waiting state. There are three different events that wake up a transaction from the waiting state:

- An Abortion event is raised when the transaction is notified of abortion of a depended transaction. On this event, the transaction starts abortion as explained above.

  A Dependency Resolution event: As will be explained in the collective commit procedure, a transaction that commits notifies all of the transactions that are dependent on it about the dependency resolution. On this event, as a dependency of the current transaction is known to be resolved, it may be able to commit; therefore, the cluster search is retried.
- A Commitment event is raised when the transaction is notified of that it is committed by the cluster search and collective commit that is started from another transaction. On this event, the notifiables that are registered to the descriptor of transaction are notified of the dependency resolution. The atomic block returns successfully.

Commitment Next, we consider the CMT rule. The rule has the condition that the set of transactions should be a cluster $\mathrm{Cluster}(\{\tau_{i=1..n}\}, \mathcal{M}, \mathcal{D})$ and also two moverness conditions $\forall \tau_{i=1..n} \forall \langle \tau^{cmt}, \ell_{\tau'} \rangle \in \ell_{sh} : \tau^{cmt} > \tau_i \Rightarrow \ell_{\tau_i} \trianglerighteq \ell_{\tau'}$ and $\forall \tau_{i=1..n} \forall \tau_{j=1..i-1} : \ell_{\tau_i} \trianglerighteq \ell_{\tau_j}$. If the rule is applied, it changes $\mathcal{M}$ to $\mathcal{M}' = \mathcal{M}[\tau_i \mapsto \mathbb{c}]_{i=1..n}$. According to the first condition, to commit a transaction, the dependency graph should be searched for a cluster containing the transaction. If the cluster search succeeds in finding a cluster, the collective commit algorithm is executed on the found cluster to check moverness conditions.

Cluster Search: A cluster is a set of terminated transactions whose dependencies are all to members of that same cluster or to committed transactions. A cluster search inputs a terminated transaction τ, and outputs either the smallest cluster that contains τ, or reports that no such cluster exists, or reports that τ must abort. We are looking for the smallest cluster because in a later phase we will have to order them, which is a time-consuming task. The smallest cluster is necessarily a strongly connected component (SCC) so we do cluster search with Tarjan's algorithm [29] for identifying SSCs. The idea is to gradually expand a candidate set of transactions containing τ until the candiate set is a cluster or the algorithm reports that no such clusters exists or that tau must abort. Specifically, if we have a candidate set and a dependency $\tau_r \rightsquigarrow \tau_s$, where τ_r is a member of the candidate set, then the cluster search does a case analysis of τ_s. If τ_s is:

- Terminated: we add τ_s to the candidate set.
- Committed: we do nothing, since the dependency is resolved.
- Running: we report that no such cluster exists.
- Aborted: we report that τ must abort.

If the Tarjan algorithm finds only one SCC, a cluster containing τ is found. On the other hand, if more than one SCC is found, the last SCC (that contains τ) is dependent on other SCCs. It is not a cluster before the other SCCs commit. Therefore, we report that no such cluster exists. (If more than one SCC is found, it is still possible to commit them. They can be committed in the order that they are found by Tarjan algorithm. This is because, the first SCC that is found is a cluster and also any SCC in the found sequence will be a cluster if the SCCs before it in the sequence are committed. But for simplicity, the current transaction waits for other SCCs to finalize.)

After the cluster search, we take one of three actions depending on the output. (1) if a cluster containing τ is found, then we commit all the transactions in the cluster; (2) if the result is that no such cluster exists, then we cache that information to avoid needlessly doing the search again before the graph changes: the thread running τ goes to the waiting state; and (3) if the result is that τ must abort, then we abort τ.

Although a transaction may wait after termination to be notified by other transactions, the implementation satisfies finalization, the progress property that we define as follows. We define that a transaction is finalized iff it is aborted or committed. We define that a transaction is settled iff it is terminated and it is

not transitively dependent on a running transaction. The finalization property is that every settled transaction is eventually finalized.

Collective Commit: To commit a set of transactions, it should be checked that there exists an order of commitment of the transactions where earlier transactions in the order do not invalidate later transactions in the order. This check corresponds to the condition $\forall\tau_{i=1..n}\forall\tau_{j=1..i-1}: \ell_{\tau_i} \overline{\triangleright} \ell_{\tau_j}$ that requires an order of transactions where operations of later transactions in the order are right movers in respect to operations of earlier transactions. In TL2, a write to a location invalidates a read from the same location. Therefore, an order is required where for each location, the reading transaction comes before the writing transaction. This condition is implemented as follows. A graph of transactions is made where a transaction τ_r has an edge to transaction τ_w if the read set of τ_r has an intersection with the write set of τ_w. If there is a cycle in the graph, a desired order does not exist. In this case, the current transaction starts abortion. Otherwise, it is possible to commit the transactions of the cluster together. Note that a pure write (writing to a location without reading it) does not conflict with another pure write and any order of commitment is valid for them. The lock for each location in the write sets of all the transactions is acquired. The global counter is incremented and is read as the write version. The read set of each transaction in the cluster is validated. This validation corresponds to the condition $\forall\tau_{i=1..n}\forall\langle\tau^{cmt}, \ell_{\tau'}\rangle \in \ell_{sh}: \tau^{cmt} > \tau_i \Rightarrow \ell_{\tau_i} \overline{\triangleright} \ell_{\tau'}$. If one of the locks cannot be acquired or a read set is not validated, the acquired locks are released and the current transaction is aborted. Otherwise, collective commit can be done. The write sets of the transactions are written to memory with the write version. The acquired locks are released. The state of the descriptor of each transaction is set to committed. Each transaction other than τ_s is notified of commitment. This notification makes a Commitment event. Each transaction that is committed sends dependency resolution notification to all notifiables $\{N_{i=1..n}\}$ that are registered to its descriptor. Each N_i references a receiving transaction τ_{r_i}. The notification makes a Dependency Resolution event for τ_{r_i}, if it is waiting. When a transaction is committed, the messages that it has sent become stable. Therefore, they can be received by receives that are executed outside transactions. Each transaction that is committed desuspends the transactors that it has sent a message to and are suspended on receives that are executed out of transactions.

6. Experimental Results

6.1. Benchmarks and Platform

We experiment with three benchmarks: A server benchmark and two benchmarks from STAMP [23]. We adopt the Server benchmark that is independently explained by [20] as the Vacation Reservation, by [14] as the Server Loop programming idiom and by [21] as the Job Handling system. A server thread handles requests from client threads. Each request should appear to be handled atomically i.e. the handling code of the server is a transaction. In addition, the request of the client thread may be sent inside a transaction. The transaction of a client may request the service multiple times. We experiment with two instances of this benchmark.

The service can simply be provision of unique ids [14]. A generic function (`serverLoop`) is offered in [14] to create servers. Employing Transactors, we provide a generic class (`Server`) that can be extended to create servers. The pseudo code of `Server` can be found in the technical report [17] section 15.4. We compare the message passing performance of our two implementations of Transactors with the implementation of TE for ML on a Server instance that generates unique ids. To the best of our knowledge, TE for ML is the closest semantics with similar goals. (We programmed and tried to conduct comparisons on other cases such as barrier, but the implementation of TE for ML took a very long time or deadlocked on these cases.)

As a tangible application of this programming idiom, consider a web application with two tiers: the application logic tier and the database tier. The system may be organized such that separate threads run the two tiers. The case study in [4] showed that to speed up handling future requests, the application logic tier may cache some of the data that it sends to the database tier. The application tier updates the cached data in the data structures and the database tier updates the data in the database. Although the updates are performed by different threads, they should be done atomically; either both or none should be seen by other threads.

We adopt the method suggested by [4] to unify memory and database transactions. The approach benefits form handlers that are registered to be run at different points of the transaction lifecycle. We extended our library to support registration of handlers for both of the implementations. We experiment with the authorship database scheme from [4]. We consider inserting a new paper info including its authors. Using our library, we define application logic transactor and database server transactor. The application logic transactor starts a transaction, sends an update request to the database server transactor, performs updates to the data structures and finishes the transaction after receiving an acknowledge message from the database server transactor. Upon receipt of a request, the database server transactor, executes a transaction comprised of queries to update data in the database and sends back an acknowledge message. The two transactions are interdependent and are collectively committed. (Note that if writing to the database is only to maintain a log for later accesses, the application logic transactor does not need to wait for the acknowledge message. In this case, only the database transaction is dependent on the application logic transaction and therefore, the application logic transaction can commit before the database transaction is done.) We study the overhead of cluster search and collective commit on this case.

To study the overhead of transactions supported by Transactors over pure transactions, we have adopted Kmeans clustering and Genome sequencing benchmarks from the Stanford transactional benchmark suite [23] and have programmed them in Scala using our Transaction and Transactors libraries.

The experiments are done on Intel(R) Core(TM)2 Duo CPU T7250 @2.00GHz and Linux 2.6.31-21-generic #59-Ubuntu. Scala version is 2.7.7.final (Java HotSpot(TM) Server VM, Java 1.6.0_17). TE for ML patch is on OCaml 3.08.1. MySQL version is 14.14 distribution 5.1.41. The database connector is MySQL Connector/J v.5.1.13. All the reported numbers are after warmup and are averages of results from repeated experiments. All the raw numbers are available in the technical report [17] section 16.

6.2. Measurements

Message Passing Performance The first experiment compares the performance of the unique id generator server in Transactors and in TE for ML over different number of repetitions of the client transaction. The same thread repeats the client transaction. (New threads are not launched for each repetition.) In this experiment, the number of requests of the client transaction is constant (equal to 2). Performance ratio represents the performance of Transactors divided by the performance of TE for ML. The two lines in Figure 8 show the performance ratio of the

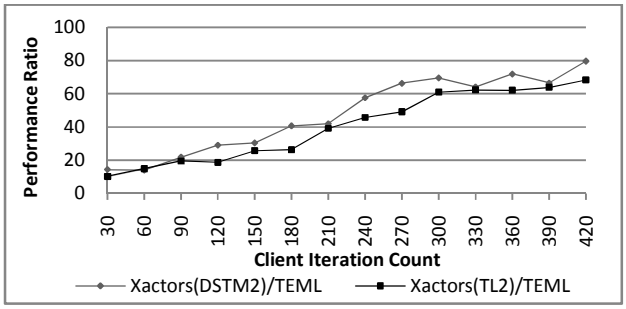


Figure 8. Server – Performance over Client Iteration Count

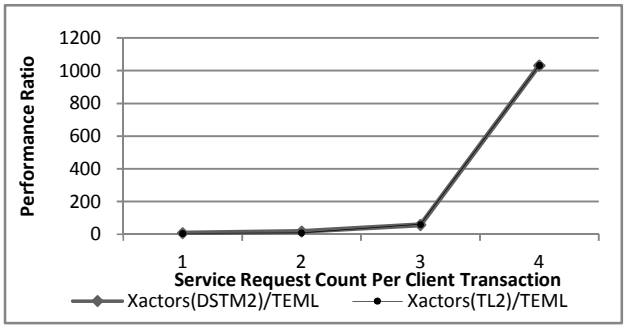


Figure 9. Server – Performance over Service Request Count

two implementations of Transactors over the implementation of TE for ML. The performance ratio increases with the number of client iterations.

The second experiment compares the performance of the server case over different number of requests of the client transaction. In this experiment, the number of repetitions of the client transaction is constant (equal to 40). Figure 9 shows the performance ratio of each of the implementations of Transactors over the implementation of TE for ML. As the number of requests increase, the performance ratio grows fast. (The two curves overlap at this scale.)

Overhead of Cluster Search and Collective Commit In this experiment, the application logic transactor maintains the set of papers and the map of each author to her set of papers. Upon addition of a new paper, the application logic transactor updates the papers set and the author-to-papers map. The database transactor inserts a row to the Paper table, gets the unique id assigned to the paper and for each author, inserts a row to the PaperAuthor table. We measure the time of the application logic transaction for insertion of a paper with four authors. Table 1 shows the percent of time that is spent in the cluster search and collective commit procedures.

Overhead over Pure Transactions Atomic blocks of Transactors provide opacity just like atomic blocks of basic memory transactions. Therefore, Transactors can be used wherever basic transactions are used. But as Transactors support communication, there is an overhead. We study this overhead on Kmeans clustering and Genome sequencing benchmarks. Each of

Table 1. Percent of Total Time Spent in Cluster Search and Collective Commit

	Cluster Search	Collective Commit
Xactors (DSTM2)	2.5	8.6
Xactors (TL2)	3.6	19.3

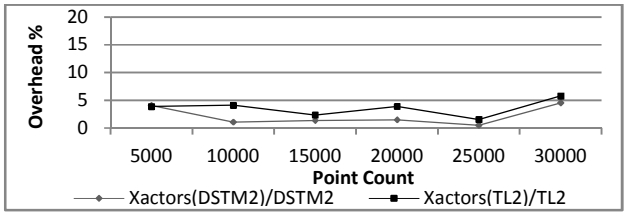


Figure 10. Kmeans Clustering – Performance Overhead

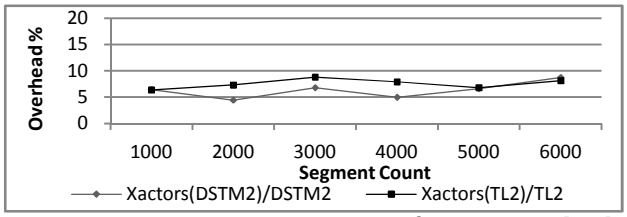


Figure 11. Genome Sequencing – Performance Overhead

our implementations of Transactors is based on an implementation of memory transactions. We compare the performance of each implementation of Transactors over the implementation of the memory transactions that it is based on. The performance overhead for the Kmeans and Genome cases over different input sizes is shown respectively in Figure 10 and Figure 11. The experiments show that the overhead is below ten percent.

6.3. Assessment

Message Passing Performance In the first experiment, the performance ratio increases with the number of client iterations. This is because Transactors use a constant number of threads. On the other hand in TE for ML, to support the completeness property, when a thread receives on a channel, every message that has been sent to the channel should be tried by a search thread. As the messages that are sent to a channel increase, the number of search threads for a receive statement increases and affects performance.

In the second experiment, in the executions with more requests in the client transaction, more messages are sent to the server channel. As mentioned for the first experiment, in TE for ML, increase in the number of messages that are sent to a channel affects performance of receive statements on the channel. Furthermore, for clients that send more requests, more `chooseEvt` statements are executed at the server thread. The number of search threads that reach a `chooseEvt` statement are doubled to try each branch. In effect, the exponential number of search threads aggravates the performance of TE for ML.

As mentioned before, TE for ML is inherently inefficient as its semantics requires finding the successful matching which is NP-hard. The measurements indicate that Transactors provide up to a thousand times faster communication than TE for ML.

Overhead of Cluster Search and Collective Commit The overhead in the implementation based on DSTM2 is relatively low. The overhead of the collective commit procedure in the implementation based on TL2 is relatively high due to the time consuming procedure of checking existence of an order of commitment that respects moverness. This procedure is the hot spot to be optimized.

Overhead over Pure Transactions In our implementations, special care is devoted to optimization of the paths that are passed by transactions that do not send or receive messages. The

measurements suggest that Transactors add less than ten percent overhead to non-communicating transactions.

7. Conclusion

This paper presents CMT that defines the semantics of transactional communication. The usefulness of CMT is shown by expressing three fundamental communication idioms. It is proved that the semantics satisfies opacity and communication safety. The semantics is implemented on top of two implementations of memory transactions. The experiments show that the implementations provide considerably efficient communication and add low overhead to non-communicating transactions.

8. References

[1] Abadi, M., Birrell, A., Harris, T., and Isard, M. 2008. Semantics of transactional memory and automatic mutual exclusion. SIGPLAN Not. 43, 1 (Jan. 2008), 63-74.

[2] Agha, Gul A. ACTORS: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, Massachusetts, 1986.

[3] Aguilera, M. K., Merchant, A., Shah, M., Veitch, A., and Karamanolis, C. 2007. Sinfonia: a new paradigm for building scalable distributed systems. In Proc. of SOSP '07. 159-174.

[4] Dias, R. J. and Lourenco, J. M.. 2009. Unifying Memory and Database Transactions. In Proc. of Euro-Par '09

[5] Dice, D., Shalev O., and Shavit N. Transactional locking II. In DISC'06, volume 4167 of Lecture Notes in Computer Science. Springer, 2006.

[6] Donnelly, K. and Fluet, M. 2008. Transactional events. J. Functional Programming. 18, 5-6 (Sep. 2008), 649-706.

[7] Dudnik P. and Swift, M. M. Condition Variables and Transactional Memory: Problem or Opportunity? In Proc. of TRANSACT'09.

[8] Effinger-Dean, L., Kehrt, M., and Grossman, D. 2008. Transactional events for ML. In Proc. of ICFP '08. 103-114.

[9] Gray J. Reuter A. 1992. Transaction Processing: Concepts and Techniques (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[10] Guerraoui, R. and Kapalka, M. 2008. On the correctness of transactional memory. In Proc. of PPoPP '08. 175-184.

[11] Harris, T. and Fraser, K. 2003. Language support for lightweight transactions. SIGPLAN Not. 38, 11 (Nov. 2003), 388-402.

[12] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. 2005. Composable memory transactions. In Proc. of PPoPP '05. 48-60.

[13] Herlihy, M., Luchangco, V., and Moir, M. 2006. A flexible framework for implementing software transactional memory. In Proc. of OOPSLA '06. 253-262.

[14] Kehrt, M., Effinger-Dean L., Schmitz M., Grossman D. Programming Idioms for Transactional Events. PLACES 2009.

[15] Koskinen, E., Parkinson, M., and Herlihy, M. 2010. Coarse-grained transactions. In Proc. of POPL '10. 19-30.

[16] Lampson, B. W. and Redell, D. D. 1980. Experience with processes and monitors in Mesa. Commun. ACM 23, 2 (Feb. 1980), 105-117.

[17] Lesani, M. and Palsberg J. Communicating Memory Transactions. Technical report, 2010. http://www.cs.ucla.edu/~lesani/papers/CommMemTrans.pdf

[18] Lipton, R. J. 1975. Reduction: a method of proving properties of parallel programs. Commun. ACM 18, 12 (Dec. 1975), 717-721.

[19] Liskov, B. 1988. Distributed programming in Argus. Commun. ACM 31, 3 (Mar. 1988), 300-312.

[20] Luchangco, V. and Marathe, V. J. Transaction Synchronizers. In Proc. of SCOOL'05.

[21] Luchangco, V. and Marathe, V. J. You are not alone: breaking transaction isolation. In Proc. of IWMSE '10. 50-53.

[22] Luchangco, V. and Marathe, V. J. Transaction Communicators: Enabling Cooperation Among Concurrent Transactions. In Proc. of PPoPP'11.

[23] Minh, C. C., Chung, J., Kozyrakis, C., Olukotun K. STAMP: Stanford Transactional Applications for Multi-Processing. In Proc. of IISWC '08.

[24] Moore, K. F. and Grossman, D. 2008. High-level small-step operational semantics for transactions. In Proc. of POPL '08. 51-62.

[25] Odersky, M. The Scala Language Specification. 2010. Programming Methods Laboratory. EPFL.

[26] Reppy, J. H. 1999 Concurrent Programming in ML. Cambridge University Press.

[27] Scott, M. L. Sequential specification of transactional memory semantics. In Proc. of TRANSACT'06.

[28] Smaragdakis, Y., Kay, A., Behrends, R., and Young, M. 2007. Transactions with isolation and cooperation. In Proc. of OOPSLA '07. 191-210.

[29] Tarjan, Robert, 1971. Depth-first search and linear graph algorithms. In Proc. of the 12th Annual Symposium on Switching and Automata Theory (13-15 Oct. 1971).,114-121.

[30] Ziarek, L., Schatz, P., and Jagannathan, S. 2006. Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In Proc. of ICFP '06. 136-147.

9. Appendix

The semantics uses a notion of right moverness [18] that we define here. Let Σ denote all the possible states of the store σ_{sh}. Let R denote the set of registers. For each $r \in R$, let $M_r = \{read, write\}$ denote the set of methods of r. For $r \in R$, $\sigma_1, \sigma_2 \in \Sigma$ and $m \in M_r$, let $\sigma_1 \xrightarrow{v \leftarrow r.m} \sigma_2$ denote the state transition from σ_1 to σ_2 by calling m on r that returns value v. Right moverness is defined as follows:

$\forall r_1, r_2 \in O, m_1 \in M_{r_1}, m_2 \in M_{r_2}$:

$r_1.m_1 \rhd r_2.m_2 \equiv \forall \sigma_1, \sigma_2, \sigma_3, \sigma_4 \in \Sigma$:

$$\left(\left(\sigma_1 \xrightarrow{v_1 \leftarrow r_1.m_1} \sigma_2 \; and \; \sigma_1 \xrightarrow{v_2 \leftarrow r_2.m_2} \sigma_3 \xrightarrow{v_1' \leftarrow r_1.m_1} \sigma_4\right) \Rightarrow (v_1 = v_1')\right)$$

According to the above definition, the right moverness relations are:

$\forall r_1, r_2 \in R, m_1, m_2 \in M_R: (r_1 \neq r_2) \Rightarrow (r_1.m_1 \rhd r_2.m_2)$

$r.read \rhd r.read$

$r.write \rhd r.read$

$r.write \rhd r.write$

Note that $r.read \rhd r.write$ is not correct.

Now we define right moverness for sequences of method calls. Let l denote a sequence of method calls on refisters R (that is $l = v_1 \leftarrow r_1.m_1, \ldots, v_n \leftarrow r_n.m_n$). Let $l_1 :: l_2$ denote the concatenation of the two sequences l_1 and l_2. Let $l[i]$ denote the ith method call in the sequence l. Let $l[1..i]$ denote the sequence of the first i method calls in the sequence l. Let $\sigma_1 \xrightarrow{l} \sigma_2$ denote multiple step transitions by l. That is if $l = v_1 \leftarrow r_1.m_1, \ldots, v_n \leftarrow r_n.m_n$ then $\sigma_1 \xrightarrow{l} \sigma_n \Leftrightarrow \sigma_1 \xrightarrow{v_1 \leftarrow r_1.m_1} \sigma_2 \ldots \sigma_{n-1} \xrightarrow{v_n \leftarrow r_n.m_n} \sigma_n$. Lifted right moverness is defined as follows: If $l_1 = v_1 \leftarrow r_1.m_1, \ldots, v_n \leftarrow r_n.m_n$ and l_2 are two sequences of methods then

$l_1 \rhd l_2 \equiv \forall i = 1..n: \forall \sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5 \in \Sigma$:

$$\left(\left(\sigma_1 \xrightarrow{l_1[1..i-1]} \sigma_2 \xrightarrow{v_i \leftarrow r_i.m_i} \sigma_3 \; and \; \sigma_1 \xrightarrow{l_2 :: l_1[1..i-1]} \sigma_4 \xrightarrow{v_i' \leftarrow r_i.m_i} \sigma_5\right) \Rightarrow (v_i = v_i')\right)$$

Note that although $r.read \rhd r.write$ is not correct, $r.write, r.read \rhd r.write$ is correct.