Technical Report

UCAM-CL-TR-759
ISSN 1476-2986

Number 759

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Coarse-grained transactions (extended version)

Eric Koskinen, Matthew Parkinson,
Maurice Herlihy

August 2011

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

ISSN 1476-2986

Coarse-Grained Transactions

(Extended Version)

Eric Koskinen[1] **Matthew Parkinson**[2] **Maurice Herlihy** [3]

August 2011

Abstract

Traditional transactional memory systems suffer from overly conservative conflict detection, yielding so-called false conflicts, because they are based on fine-grained, low-level read/write conflicts. In response, the recent trend has been toward integrating various abstract data-type libraries using ad-hoc methods of high-level conflict detection. These proposals have led to improved performance but a lack of a unified theory has led to confusion in the literature.

We clarify these recent proposals by defining a generalization of transactional memory in which a transaction consists of coarse-grained (abstract data-type) operations rather than simple memory read/write operations. We provide semantics for both pessimistic (e.g. transactional boosting) and optimistic (e.g. traditional TMs and recent alternatives) execution. We show that both are included in the standard atomic semantics, yet find that the choice imposes different requirements on the coarse-grained operations: pessimistic requires operations be left-movers, optimistic requires right-movers. Finally, we discuss how the semantics applies to numerous TM implementation details discussed widely in the literature.

[1]University of Cambridge
[2]Microsoft Research Cambridge
[3]Brown University

Chapter 1

Introduction

1.1 Motivation

Transactional Memory (TM) comes in many varieties: hardware [13], software [10, 27, 7], and mixtures of both [6, 23]. Common to all these approaches is the need to define and detect synchronization conflicts: circumstances under which two transactions cannot proceed in parallel. Existing TM systems detect conflicts at a *fine-grained* level: each transaction maintains a read set, encompassing the locations it read, and a write set, encompassing the locations it wrote. Two transactions are deemed to conflict if one's write set intersects the other's read or write set. Defining a synchronization conflict to be read/write conflict is appealing because it is usually easy to classify operations, whether in hardware or software, as reads or writes.

Nevertheless, there is an increasing realization that read/write conflicts support insufficient levels of concurrency for objects subject to contention. Conflict detection on the basis of read/write sets is conservative: two high-level operations may not logically conflict even though they share memory locations. For example, if two threads insert distinct items in a shared collection, then, logically, neither one should be affected by the other's actions. But if the collection type is implemented, say, as a linked list, then their read and write sets may inadvertently intersect in a harmless way, producing an unnecessary *false conflict*.

In response we have suggested replacing this low-level notion of read/write conflict with a high-level notion of conflicts between methods of abstract data-types, leveraging sequential method semantics [11]. Moreover, others have begun developing automated techniques to discover high-level semantics [26, 2, 8]. As we have shown, the resulting TMs often offer performance improvements of at least an order of magnitude.

At present, despite decades of previous work in the database community, there is no formal account for these high-level approaches in the context of programming languages. Consequently, the literature tends to be ad-hoc and numerous questions are open: What execution models are possible? How do these high-level approaches relate to traditional read/write-based transactional memory? What, precisely, are the requirements imposed on abstract data-types? Would a unified theory uncover other opportunities for improvement in the state-of-the-art?

In this paper we unify these ad-hoc techniques into a theory we call *coarse-grained transactions*, generalizing transactional memory with a semantics that answers these open questions. A coarse-grained transaction consists of operations on abstract data-types rather than simple memory read/write operations. We present both optimistic and pessimistic execution semantics of coarse-grained transactions, and show that both abide the standard atomic semantics. Rather than using a general notion of commutativity, we precisely characterize the requirements of operations in terms of left-movers and right-movers [21]. We find that the pessimistic execution requires that outstanding operations be left-movers with respect to each other, whereas the optimistic execution requires that operations be right-movers with respect to committed operations.

We then discuss how our semantics can be used to model a variety of implementation details in the literature. Our semantics encompasses both fine-grained operations (e.g., memory read/write) and coarse-grained operations (e.g., transactional boosting [11]). Our pessimistic execution semantics model, again, transactional boosting [11], whereas the optimistic execution semantics model STM implementations with lazy update strategies [7, 27] as well as recent work which resembles coarse-grained transactions [3]. Finally we discuss our treatment of a variety of other issues in the literature including (closed) nested transactions [24], checkpoints [16] and deadlock detection [17].

Example

```
T1: atomic {       ||    T2: atomic {
  x := 3;          ||      z++;
  z++;             ||      y := 4;
}                  ||    }
```

The transactions T1 and T2 in the example above execute a series of statements which read or write values from locations on the shared heap (x,y,z). If executed concurrently the transactions will conflict because both write to z, and one will be aborted.

However, if we assume incrementing z is a single instruction, notice any execution is serializable. This observation stems from the fact that increments to z *commute*: applying them in either order yields the same final state. Therefore, an interleaved execution:

T1:	(x:=3)			(z++)	(cmt)	
T2:		(z++)	(y:=4)			(cmt)

is equivalent to a serial execution which preserves the commit order:

T1:	(x:=3)	(z++)	(cmt)			
T2:				(z++)	(y:=4)	(cmt)

This serial order can be obtained by reasoning about moverness [21] of the operations performed by transactions. A read(a) operation can move to the left or right of a write(b:=e) operation (from distinct transactions) when a and b are distinct and a is not free in e. For example, to construct the serial history above, we moved the operation T2:(y:=4) to the right of T1:(z++). Moreover, incrementing z can be applied in either order and reach the same final state.

Coarse-Grained Transactions We saw above that two operations which each increment a variable commute, i.e. can move to the left or to the right of each other. This illustrated how the imprecise conflict detection in traditional STM implementations is unable to detect the serializability of some executions at the *fine-grained* level of memory options.

Now consider a higher level of abstraction, where transactions perform *coarse-grained* operations. In the following example two transactions perform operations on a data structure:

```
              global SkipList Set;
T1: atomic {           ||  T2: atomic {
  Set.add(5);          ||    if (Set.contains(6))
  Set.remove(3);       ||      Set.remove(7);
}                      ||  }
```

This illustrates the changing focus in the transactional memory community. The old read/write operations are replaced with arbitrary, coarse-grained operations. A transaction consists of operations (represented as *object methods*) on regions of the heap (represented as *object state*) which we will assume to be disjoint. The shared state above is a Set data structure, in this case implemented as a (linearizable) skip-list. Each transaction then proceeds by performing Set method calls: insert (x), remove(y), contains(z).

Now at this coarse-grained level, how can operations be synchronized? Because high-level operations may share memory locations, using a fine-grained synchronization technique (detecting read/write conflicts) impairs performance [11]. The solution is to instead base synchronization on how one method moves with respect to another. In the example above—assuming SkipList is linearizable—any of the methods can move to the left or right of any of the other methods provided their arguments are distinct. In other cases some operations may only move in one direction. For example, a semaphore increment operation can move to the left of a decrement operation but not vice-versa.

Recently, ad-hoc techniques have appeared in the literature which appear to be exploiting left-movers and right-movers to some degree, yet they lack a formal footing [11, 19, 3]. In this paper we show how these and future proposals can be characterized in a semantics based on left-movers and right-movers.

Semantics In the remainder of this paper we give a semantics for a generalized notion of transactions, which we call *coarse-grained transactions*, where transactions perform abstract data-type operations rather than simple memory read/write operations. Despite having a single atomic semantics, there are two execution semantics and each imposes different requirements on the moverness of the transactional operations:

1. *Pessimistic Execution Semantics* (Section 2.2) – Transactions will be delayed from taking steps in the semantics if the operations they intend to perform do not commute with the outstanding operations of other active transactions.
   This semantics requires that operations be *left-movers with respect to other uncommitted transactions.*
   This semantics is subject to *deadlock* [11] though recovery is possible [17] via *inverse* operations. This left-mover requirement also permits a transaction to perform an inverse operation that annihilates the transaction's most recent operation. Since all the pending operations to the right must be left movers, the most recent operation can be moved to the far right.

2. *Optimistic Execution Semantics* (Section 2.3) – The shared state is logically duplicated into transaction-local state and, thus, method call steps in the semantics

are always enabled. However, when transactions attempt to commit, they may be required to abort if they have performed method calls that do not commute with other concurrent transactions. Otherwise transaction-local changes are merged into the shared state.
This semantics requires that operations be *right-movers with respect to operations of other committed transactions.* This semantics includes traditional read/write software transactional memory with lazy update strategies [7, 27] as well as recent alternatives which involve user-defined conflict resolution [3]. This semantics is also subject to *livelock* (as is STM).

We show (Section **??**) that both of the above high-level semantics are equivalent to atomic semantics. We then discuss (Sections 4.1, 4.3) how the semantics applies to numerous TM implementation details discussed widely in the literature.

Limitations Coarse-grained transactions have two other requirements not addressed here. First, operations must be linearizable—a property guaranteed in traditional TM since read/write operations are typically atomic. Second, for simplicity we require that moverness take isolation into consideration. If two objects share memory one may not be able to reason about them independently. Type systems or alias analyses could potentially provide such guarantees.

1.2 Preliminaries

We will use the following language:

$$\begin{array}{rcl} s & ::= & c; s \mid \texttt{beg}\, t;\, s \mid \texttt{skip} \\ t & ::= & c; t \mid o.m; t \mid \texttt{end} \end{array}$$

The language consists of standard commands c such as assignment and branching, statements s which can be commands or initiate transactions, and transactions t which can be commands or coarse-grained operations (represented as method calls m on shared objects o). Expressions e are standard and involve variables, constants, algebraic expressions, λ-abstraction and application. Boolean expressions b are similar. Thread-local state can be modeled as shared objects that are only touched by one thread. For simplicity we assume a parallel composition rather than an explicit treatment of `fork`.

Transactions operate on a shared store of objects, manipulated with their corresponding methods. We denote the shared store σ_{sh}, and it contains shared objects, each denoted o. The shared store σ_{sh}^0 is defined as:

$$\sigma_{\text{sh}}^0 \triangleq \{\forall o \in \mathsf{objects}(P).o \mapsto \mathsf{init}(o)\}$$

where $\mathsf{objects}$ is the obvious function which returns the set of all objects used by the program P, and init returns the initial state of the given object.

1.3 Moverness

We now formally define the notions of left- and right-moverness which is due to Lipton [21]. We will use moverness to characterize the necessary restrictions placed on the object in the pessimistic and optimistic semantics.

Definition 1.3.1. *(Left Mover) o.m is a left-mover with p.n, denoted* $o.m \lhd p.n$*, if and only if for any state* σ

$$\begin{gathered}\{\sigma'' \mid \exists \sigma'.\sigma \xrightarrow{p.n} \sigma' \wedge \sigma' \xrightarrow{o.m} \sigma''\} \\ \subseteq \\ \{\sigma''' \mid \exists \sigma'.\sigma \xrightarrow{o.m} \sigma' \wedge \sigma' \xrightarrow{p.n} \sigma'''\}\end{gathered}$$

This definition gives the conditions under which one step in the semantics is a left mover with respect to another step. Specifically, in a given execution where two methods are executed, the condition is that the set of states reached is included in the set of possible states reached had the methods occurred in the opposite order. Of course when o is a distinct object from p, then for any two methods, the two are both movers with respect to each other:

It is important to note that in Definition 1.3.1, the relationship is an inclusion rather than an equivalence. The left-hand side of the inclusion is typically a concrete execution where specific states are known. By contrast, the right-hand side must account for non-determinism in the sequential specifications of the objects. For example, let o be a priority queue, and the method m be the removal of the highest-priority element. If there are multiple elements in the queue with the highest priority, then the semantics of the queue may state that any of them will be returned. Thus, σ' in the right-hand side is a *set* of states, as is the entire right-hand side.

We will later show that inclusion is sufficient for proving serializability of execution traces. Whereas previous proofs of serializability were based on finding *one* equivalent sequential execution for an interleaved execution, our proof technique instead shows that a given interleaved execution is contained within a set of executions, all of whom are sequential.

Definition 1.3.2. *(Right Mover* $\rhd$*, Both Mover* $\bowtie$*) Defined similarly.*

Definition 1.3.3. *(Lifted Movers) We define left mover lifted to sets, denoted* $\overline{\lhd}$*, as follows:*

$$M_1 \mathrel{\overline{\lhd}} M_2 \quad \triangleq \quad \forall (o.m, p.n) \in (M_1, M_2).\ \ o.m \lhd p.n$$

Lifted versions of $\rhd$*,* $\bowtie$ *are similar and denoted* $\overline{\rhd}$*,* $\overline{\bowtie}$*.*

Chapter 2

Semantics

2.1 Atomic Semantics

The language has the atomic semantics $\rightarrow_A$ and $\rightarrow_B$ given in Figure 2.1 The semantics is a labeled transition system. The domain of labels is $(\mathcal{O} \times \mathcal{M}) \cup \{\mathsf{beg}, \mathsf{cmt}\}$, representing transactional operations, transaction begin, and transaction commit. We also allow unlabeled transitions (as in the ACMD and ASKIP rules). A configuration in $\rightarrow_A$ is a pair T, σ representing the transactions T and the store of objects σ. We will assume that σ has per-transaction local components σ_τ which can only be modified by the corresponding transaction (in rule ACMD below). The initial configuration is a parallel composition of threads: $\mathcal{C}_0^{\mathsf{a}} = \{T_0, T_1, ..., T_n\}, \sigma^0$ where each $T_i \in s$ is the program code for the ith transaction. Reductions are taken when one element of T is selected and any enabled reduction is taken. The companion reduction $\rightarrow_B$ is similar, but configurations only involve a single transaction.

The reductions are as follows. (ACMD) The effect of a simple command c on σ is given by the semantics of c. (ASKIP) A `skip` reduces to the empty set. (ATXN) A transaction statement followed by a second statement reduces to the second statement provided that the transaction body can be reduced via $\rightarrow_B$. (AAPP) The only $\rightarrow_B$ reduction corresponds to a transactional operation $o.m$ which is reduced by applying the operation to the state.

The semantics is "atomic": all operations performed by a transaction are applied to the shared state in one step of $\rightarrow_B$, in which the transition is labeled with each of the transaction's operations. In the remainder of this chapter we present two execution semantics, one pessimistic and one optimistic. In Chapter 3 we show that each of these two execution semantics are serializable.

2.2 Pessimistic Execution Semantics

In this section we present the first of two semantics which correspond to implementation strategies. These semantics illustrate how different execution models place different requirements on the moverness of the coarse-grained operations. In Chapter 3 we will show that each is serializable.

Configurations in the pessimistic execution semantics are similar to the atomic semantics given in Section 1.2. However, the elements of T are of the form $\langle \tau, s, M, \sigma_\tau \rangle$,

$$\frac{}{\{c\,;s\} \cup T, \sigma \xrightarrow{\bot}_A \{s\} \cup T, [\![c]\!]\sigma}\ \text{ACMD} \qquad \frac{}{\{\texttt{skip}\} \cup T, \sigma \xrightarrow{\bot}_A T, \sigma}\ \text{ASKIP}$$

$$\frac{t, \sigma \xrightarrow{\mathcal{L}_1 \cdots \mathcal{L}_n}{}^{*}_{B} \texttt{end}, \sigma'}{\{\texttt{beg}\ t\,;s\} \cup T, \sigma \xrightarrow{\mathsf{beg}\mathcal{L}_1 \cdots \mathcal{L}_n\mathsf{cmt}}_A \{s\} \cup T, \sigma'}\ \text{ATXN}$$

$$\frac{}{c\,;t, \sigma \xrightarrow{\bot}_B t, [\![c]\!]\sigma}\ \text{BCMD} \qquad \frac{}{o.m\,;t, \sigma \xrightarrow{o.m}_B t, [\![o.m]\!]\sigma}\ \text{BAPP}$$

Figure 2.1: Atomic Semantics

where $\tau \in \mathbb{N}_\bot$ identifies the transaction if one is currently running, s denotes the program code, M denotes the sequence of object methods that τ has applied, and σ_τ is the transaction-local state. The initial configuration is a parallel composition of threads: $\mathcal{C}_0^{\mathsf{p}} = \{T_0, T_1, ..., T_n\}, \sigma_{\text{sh}}^0$, where each $T_i = \langle \bot, P_i, \emptyset, \sigma^0 \rangle$ and σ_{sh}^0 is as defined in the previous section.

The reductions for pessimistic execution are given in Figure 2.2. The PCMD step involves a thread executing a statement which impacts only thread-local state and control-flow. In the PBEG rule the system makes a transition in which a thread begins to execute a transaction. The effect is that the sequence of applied operations M is initialized to the empty sequence and the new transaction tuple is added to the set of active transactions. $\tau \in \mathbb{N}$ and the predicate $\mathsf{fresh}(\tau)$ indicates that τ is fresh and strictly increasing (the monotonicity requirement is unneeded here, but required for the optimistic execution semantics discussed next, which use τ as a timestamp).

The PAPP rule allows a transaction to apply a method m on a shared object o. The rule indicates the initiation of a method call, which is appended to the sequence of operations. The log also includes the current state and program counter, which may be used later in the PUNDO rule. The return value (given by rv) is stored in the thread-local variable x. (Note that we assume the shared object is linearizable.)

In these pessimistic execution semantics synchronization is considered throughout a transaction rather than simply at commit-time. Thus the PAPP step involves a check. The commit order of the active transactions is not determined *a priori*, so any active transaction may commit at any time. Therefore there is a restriction for the reduction: the new method call must be a left-mover with respect to all other active method calls. Left-moverness ensures that the current transaction could commit before all of the others.

Since synchronization is considered during the PAPP rules, the commit rule PCMT is trivial. We will see in the next section that these pessimistic semantics stand in contrast to the optimistic semantics in which the PAPP rule is always enabled, but a synchronization check is performed in the commit rule.

The commit rule PCMT merely involves discarding the set of active methods and terminating the transaction. All effects have already been applied to the shared state.

Finally the PUNDO rule allows a transaction to move backwards to the program location immediately preceding the most recently applied object method. Longer jumps are possible by multiple uses of the PUNDO rule. PUNDO is always enabled when M is non-empty. The PUNDO rule (as we discuss in Section 4.1) is useful to allow transactions

$$\frac{}{\langle\tau, s \in \{:=, \texttt{if}\};\ s', M, \sigma_{\text{lo}}\rangle, T, \sigma_{\text{sh}} \xrightarrow{\perp}_P \langle\tau, s', M, [\![s]\!]\sigma_{\text{lo}}\rangle, T, \sigma_{\text{sh}}}\ \text{PCMD}$$

$$\frac{}{\langle\perp, \mathsf{beg};\ s, M, \sigma_{\text{lo}}\rangle, T, \sigma_{\text{sh}} \xrightarrow{\mathsf{fresh}(\tau),\mathsf{beg}}_P \langle\mathsf{fresh}(\tau), s, [], \sigma_{\text{lo}}\rangle, T, \sigma_{\text{sh}}}\ \text{PBEG}$$

$$\frac{\{o.m\} \lhd \mathsf{meths}(T)}{\langle\tau, x := o.m;\ s, M, \sigma_{\text{lo}}\rangle, T, \sigma_{\text{sh}} \xrightarrow{\tau,o.m}_P \langle\tau, s, M :: (x := o.m;\ s, \sigma_{\text{lo}}, o.m), \sigma_{\text{lo}}[x \mapsto rv([\![o]\!]\sigma_{\text{sh}}.m)]\rangle, T, \sigma_{\text{sh}}[o \mapsto [\![o]\!]\sigma_{\text{sh}}.m]}\ \text{PA}$$

$$\frac{}{\langle\tau, s, M :: (s', \sigma'_{\text{lo}}, o.m), \sigma_{\text{lo}}\rangle, T, \sigma_{\text{sh}} \xrightarrow{\tau,o.m^{-1}}_P \langle\tau, s', M, \sigma'_{\text{lo}}\rangle, T, \sigma_{\text{sh}}[o \mapsto [\![o]\!]\sigma_{\text{sh}}.m^{-1}]}\ \text{PUNDO}$$

$$\frac{}{\langle\tau, \mathsf{end};\ s, M, \sigma_{\text{lo}}\rangle, T, \sigma_{\text{sh}} \xrightarrow{\tau,\mathsf{cmt}}_P \langle\perp, s, \emptyset, \sigma_{\text{lo}}\rangle, T, \sigma_{\text{sh}}}\ \text{PCMT}$$

$$\begin{array}{lcl}
\mathsf{meths}(\emptyset) & = & \emptyset \\
\mathsf{meths}(T' \cup \{\langle\tau, C, M, \sigma_\tau\rangle\}) & = & \mathsf{meths}_M(M) \cup \mathsf{meths}(T') \\
\mathsf{meths}_M([]) & = & \emptyset \\
\mathsf{meths}_M(M' :: (s, \sigma_{\text{lo}}, o.m)) & = & \mathsf{meths}_M(M') \cup \{(o.m)\}
\end{array}$$

Figure 2.2: Pessimistic Semantics $\rightarrow_P$

to escape from deadlock [17].

There is no moverness requirement for the PUNDO operation— there is a subtle reason why it is unneeded. As we show formally in Section 3.3, the "doomed" operation that the transaction is undoing is, in effect, a right-mover since all subsequent operations from other transactions are left-movers. Thus, the doomed operation can be moved to the far right and then annihilated with the inverse operation.

2.3 Optimistic Execution Semantics

We now present the optimistic execution semantics. We then, in Section 2.4, compare it to the pessimistic counterpart. The most significant difference between the two is that the optimistic semantics ensure synchronization at commit-time, whereas the pessimistic semantics ensure synchronization with each operation.

Before we continue, we emphasize that even though our model involves thread-local copies of the shared state, in many cases this is not a performance problem since (i) copies are unneeded when operations are read-only and (ii) copy-on-write can be used elsewhere. The existing TL2 [7] and Intel [27] implementations are evidence of this fact.

Configurations in the optimistic semantics are a triple $T, \sigma_{\text{sh}}, \ell_{\mathsf{sh}}$ where T is a set of per-transaction state, σ_{sh} is the same as the pessimistic shared state, and ℓ_{sh} is a log of pairs: each committed transaction and the operations it has performed. Elements of T are of the form $\langle\tau, s, \sigma_\tau, \overleftarrow{\sigma}_\tau, \ell_\tau\rangle$. τ is a unique transaction identifier, s is the next code

$$\frac{}{\langle \tau_\perp, s \in \{:=, \texttt{if}\};\ s, \sigma_{\mathrm{lo}}, \overleftarrow{\sigma}_{\mathrm{lo}}, \ell_\tau \rangle, T, \sigma_{\mathrm{sh}}, \ell_{\mathsf{sh}} \xrightarrow{\perp}_O \langle \tau_\perp, s, [\![s]\!]\sigma_{\mathrm{lo}}, \overleftarrow{\sigma}_{\mathrm{lo}}, \ell_\tau \rangle, T, \sigma_{\mathrm{sh}}, \ell_{\mathsf{sh}}}\ \text{OCMD}$$

$$\frac{}{\langle \perp, \mathsf{beg};\ s, \sigma_{\mathrm{lo}}, \overleftarrow{\sigma}_{\mathrm{lo}}, [] \rangle, T, \sigma_{\mathrm{sh}}, \ell_{\mathsf{sh}} \xrightarrow{(\tau,\mathsf{beg})}_O \langle \mathsf{fresh}(\tau), s, \mathsf{snap}(\sigma_{\mathrm{lo}}, \sigma_{\mathrm{sh}}), \sigma_{\mathrm{lo}}[\mathsf{stmt} \mapsto \text{“}\mathsf{beg};\ s\text{”}], [] \rangle, T, \sigma_{\mathrm{sh}}, \ell_{\mathsf{sh}}}\ \text{OBEG}$$

$$\frac{}{\begin{array}{l}\langle \tau, x := o.m;\ s, \sigma_{\mathrm{lo}}, \overleftarrow{\sigma}_{\mathrm{lo}}, \ell_\tau \rangle, T, \sigma_{\mathrm{sh}}, \ell_{\mathsf{sh}} \\ \xrightarrow{(\tau, o.m)}_O \langle \tau, s, \sigma_{\mathrm{lo}}[o \mapsto \sigma_{\mathrm{sh}}(o).m, x \mapsto rv(\sigma_{\mathrm{sh}}(o).m)], \overleftarrow{\sigma}_{\mathrm{lo}}, \ell_\tau @(\text{“}o.m\text{”}) \rangle, T, \sigma_{\mathrm{sh}}, \ell_{\mathsf{sh}}\end{array}}\ \text{OAPP}$$

$$\frac{\forall (\tau^{\mathsf{cmt}}, \ell_{\tau'}) \in \ell_{\mathsf{sh}}.\tau^{\mathsf{cmt}} > \tau \Rightarrow \ell_\tau \mathrel{\bar{\rhd}} \ell_{\tau'}}{\langle \tau, \mathsf{end};\ s, \sigma_{\mathrm{lo}}, \overleftarrow{\sigma}_{\mathrm{lo}}, \ell_\tau \rangle, T, \sigma_{\mathrm{sh}}, \ell_{\mathsf{sh}} \xrightarrow{(\tau,\mathsf{cmt})}_O \langle \tau, s, \sigma_{\mathrm{lo}}, \overleftarrow{\sigma}_{\mathrm{lo}}, [] \rangle, T, \mathsf{merge}(\sigma_{\mathrm{sh}}, \ell_\tau), \ell_{\mathsf{sh}} :: (\mathsf{fresh}(\tau^{\mathsf{cmt}}), \ell_\tau)}\ \text{OCMT}$$

$$\frac{}{\langle \tau, s, \sigma_{\mathrm{lo}}, \overleftarrow{\sigma}_{\mathrm{lo}}, \ell_\tau \rangle, T, \sigma_{\mathrm{sh}}, \ell_{\mathsf{sh}} \xrightarrow{(\tau,\mathsf{abt})}_O \langle \perp, \overleftarrow{\sigma}_{\mathrm{lo}}[\mathsf{stmt}], \overleftarrow{\sigma}_{\mathrm{lo}}, \overleftarrow{\sigma}_{\mathrm{lo}}, [] \rangle, T, \sigma_{\mathrm{sh}}, \ell_{\mathsf{sh}}}\ \text{OABT}$$

$$\begin{array}{lcl}
\mathsf{snap}(\sigma_{\mathrm{lo}}, \sigma_{\mathrm{sh}}) & \triangleq & \lambda o. \begin{cases} \sigma_{\mathrm{sh}}(o) & \text{if } o \in \mathrm{dom}(\sigma_{\mathrm{sh}}) \\ \sigma_{\mathrm{lo}}(o) & \text{otherwise} \end{cases} \\
\mathsf{merge}(\sigma_{\mathrm{sh}}, []) & = & \sigma_{\mathrm{sh}} \\
\mathsf{merge}(\sigma_{\mathrm{sh}}, (\text{“}o.m\text{”}) :: \ell_\tau) & = & \mathsf{merge}(\sigma_{\mathrm{sh}}[o \mapsto (\sigma_{\mathrm{sh}}(o)).m], \ell_\tau)
\end{array}$$

Figure 2.3: Optimistic Semantics $\rightarrow_O$

to be executed, σ_τ is the transaction-local copy of the shared objects, $\overleftarrow{\sigma}_\tau$ is a back-up copy of the state (used for aborting), and ℓ_τ is a log of the operations performed during a transaction (to be used to check moverness at commit-time). The notation $\tau_\perp$ means that either transaction is executing ($\tau_\perp = \tau$). or the thread is running outside of transaction mode, ($\tau_\perp = \perp$). The initial configuration is a parallel composition of threads: $\mathcal{C}_0^{\mathsf{opt}} = \{T_0, T_1, ..., T_n\}, \sigma^0_{\mathrm{sh}}, []$, where each $T_i = \langle \perp, P_i, \sigma^0, \sigma^0, [] \rangle$.

Labeled transitions occur when an element of T is selected and one of the reductions given in Figure 2.3 applies (for the complete semantics, see our Technical Report [18]). Like PCMD, the OCMD reduction merely applies changes to the local state, and leaves $\overleftarrow{\sigma}_\tau$ unchanged. The OBEG reduction represents the initiation of a transaction. A new identifier is generated, and the existing thread-local state is copied to $\overleftarrow{\sigma}_\tau$ to be used if the transaction is later aborted. A snapshot $\mathsf{snap}(\sigma_, \sigma_{\mathrm{sh}})$ is taken of the current state of all shared objects, duplicating them from the shared store to the local store. The current s is also stored in $\overleftarrow{\sigma}_\tau$.

In the reduction OAPP, a transaction applies a method. The operation is applied to the local store (containing copies of shared objects), and the operation is logged so it can be replayed on the shared state later at commit. $\overleftarrow{\sigma}_\tau$ is unchanged.

The OCMT reduction first identifies this set of recently committed transactions by comparing the timestamp of committed transactions with the begin timestamp of the current transaction. The reduction is only taken if all of the transaction's methods are right-movers with respect to methods of the recently committed transactions. If the reduction is taken, the thread-local operations from ℓ_τ are propagated to the shared state σ_{sh} via the merge function, and a new timestamp (denoted $\mathsf{fresh}(\tau^{\mathsf{cmt}})$) is appended to the commit log. The reduction then clears the thread local state, restoring it to $\sigma_\tau^0 \triangleq [\cdot \mapsto \bot]$, clearing the thread-local log ℓ_τ and discarding the unused $\overleftarrow{\sigma}_\tau$.

The last rule OABT corresponds to aborting a transaction. The saved thread-local store $\overleftarrow{\sigma}_\tau$ restored, code (i.e. program counter) is restored via $\overleftarrow{\sigma}_\tau$, and the log is discarded.

Remark: Partial Aborts Note that by contrast to the pessimistic version, this semantics does not allow partial aborts [16]. Adding such a facility can certainly be accomplished, but for ease of presentation, and since no existing implementations provide it, we have omitted it.

2.4 Comparison

As noted above, the most significant distinction between the two semantics is when synchronization occurs. The pessimistic semantics perform a check (left-moverness) with each coarse-grained operation, whereas the optimistic semantics perform a check (right-moverness) at commit-time. Intuitively, the pessimistic left-moverness preserves the invariant that any live transaction *can commit at any time.* By contrast, the optimistic right-moverness checks to see if a transaction's *thread-local store is logically consistent* given that other transactions may have committed since the copy was made. There are several considerations involved in the choice between execution semantics:

- *Aborts* – Optimistic transactions can always proceed with operations immediately modifying their thread-local store, however they may need to be aborted. Pessimistic transactions are always able to commit (modulo deadlock) but may have to be delayed before applying an operation.
- *State Duplication* – The optimistic semantics require, at least logically, thread-local duplication of the shared state. Sometimes duplication can be reduced (e.g. with copy-on-write strategies) or even eliminated (e.g. when a return value is unneeded or can be otherwise calculated). Alternatively, the pessimistic semantics does not require duplication.
- *Merge* – The optimistic semantics involve a commit-time merge of operations into the shared state. In the worst case this may involve replaying the entire operation, perhaps favoring the pessimistic semantics. However, there may be instances (such as in read/write STMs) where merge can be implemented efficiently.
- *Inverses* – When transactions are aborted, the pessimistic semantics performs inverse operations (PUNDO). If these operations are expensive or difficult to define, the optimistic semantics may be preferred, which simply discards the thread-local scratch pad.
- *Objects* – In some applications, left-movers are more common than right-movers or vice-versa. In such instances the corresponding semantics (pessimistic or optimistic, respectively) is more amenable because more concurrency is possible.

Some examples of the two semantics are as follows. An example of the optimistic semantics are the TL2 [7] STM implementation and Intel STM compiler [27], which we discuss in Section 4.1. In each, locks are used at commit-time, ensuring the left-moverness of pending updates. Another example of the optimistic semantics, discussed in Section 4.3.2, is recent work [3] in which operations are more coarse-grained and conflicts are resolved by programmer-provided rules. Finally, transactional boosting [11] is an example of pessimistic coarse-grained transactions. We discuss boosting in Section 4.3.1. A side-by-side empirical evaluation of pessimism versus optimism is left for future work.

Remark One might wonder whether the optimistic semantics are contained within the pessimistic semantics (or vice-versa). Unfortunately this is not the case because of the fundamental differences described earlier, and leads to a more-or-less disjoint union of the two semantics. Whereas the pessimistic semantics maintain an commutativity invariant (via the PAPP reduction), the optimistic semantics check at commit. The two semantics have different kind of logs: optimistic transactions have "intention logs" (ℓ_τ) where as pessimistic transactions have "undo logs" (M). One might try to think of the intention logs as being empty for pessimistic semantics and vice-versa, but then neither is contained within the other.

Chapter 3

Serializability

3.1 Execution Traces

In this section we define execution traces and prove several important properties of them. These properties will be used in Section **??** to show that pessimistic and optimistic execution models are both serializable.

Recall that labels are given by the reductions in the semantics (either $\rightarrow_P$ or $\rightarrow_O$). They are taken from the domain $\bot \cup (\mathbb{N} \times (o.m \cup \{\mathsf{beg}, \mathsf{cmt}, \mathsf{abt}\}))$, representing transaction commencement, completion and abortion, as well as applying operations on the shared objects.

Definition 3.1.1. *(Execution Trace) A trace ϵ is a series of labels:*

$$\epsilon = \mathcal{L}_0 \cdot \mathcal{L}_1 \cdot \mathcal{L}_2 \cdots$$

We denote an empty execution trace as $\circ$.

Definition 3.1.2. *(Finals) fin is an inductively defined predicate, mapping a set of initial configurations and an execution trace (sequence of labels) to a set of final configurations:*

$$\begin{aligned} [\mathit{fin}\ \mathbf{C}\ \circ] &\triangleq \mathbf{C} \\ [\mathit{fin}\ \mathbf{C}\ \mathcal{L}\cdot\epsilon] &\triangleq [\mathit{fin}\ \{C' \mid \forall C \in \mathbf{C}. C \xrightarrow{\mathcal{L}} C'\}\ \epsilon] \end{aligned}$$

For shorthand, we often use a single configuration as the first argument to fin, though we mean the usual singleton set. Observe that we can always expand subsequences of any execution:

Property 3.1.1. *(Containment Folding)*

$$[\mathit{fin}\ \mathbf{C}\ \epsilon\cdot\epsilon'] = [\mathit{fin}\ [\mathit{fin}\ \mathbf{C}\ \epsilon]\ \epsilon']$$

Proof: Proof by induction on ϵ. For details see our Technical Report [18].

Lemma 3.1.1. *(Lifting containment)*

$$\mathbf{C} \subseteq \mathbf{C}' \Rightarrow [\mathit{fin}\ \mathbf{C}\ \mathcal{L}] \subseteq [\mathit{fin}\ \mathbf{C}'\ \mathcal{L}]$$

Proof: Unrolling fin once and then proof by contradiction. For details see our Technical Report [18].

Definition 3.1.3. *(Movers lifted to labels)*

$$\mathcal{L}_1 \mathrel{\widehat{\lhd}} \mathcal{L}_2 \iff \forall C.[\mathit{fin}\ C\ \mathcal{L}_2 \cdot \mathcal{L}_1] \subseteq [\mathit{fin}\ C\ \mathcal{L}_1 \cdot \mathcal{L}_2]$$

Right- and both-movers $\widehat{\rhd}$ *and* $\widehat{\bowtie}$ *are similarly lifted.*

Property 3.1.2. $o.m \lhd p.n \iff (\tau, o.m) \mathrel{\widehat{\lhd}} (\tau', p.n)$ *and similar for* $\widehat{\rhd}$ *and* $\widehat{\bowtie}$*.*

Proof: Correspondence is definitional. For details see our Technical Report [18].

Definition 3.1.4. *(Execution containment)*

$$\epsilon \sqsubseteq \epsilon' \stackrel{\triangle}{=} \forall C.[\mathit{fin}\ C\ \epsilon] \subseteq [\mathit{fin}\ C\ \epsilon']$$

Lemma 3.1.2.

$$\epsilon_a \sqsubseteq \epsilon_b \Rightarrow \epsilon_a \cdot \epsilon' \sqsubseteq \epsilon_b \cdot \epsilon'$$

Proof: By induction on ϵ'. For details see our Technical Report [18].

3.2 Serial Traces

Recall that execution traces are given as series of labels in the transition systems of the appropriate operational semantics. For example, here is a trace of two transactions:

$$\epsilon = (\tau, \mathsf{beg}) \cdot (\tau, o.m) \cdot (\tau', \mathsf{beg}) \cdot (\tau', o.m) \cdot (\tau', \mathsf{cmt}) \cdot (\tau, \mathsf{cmt})$$

The following atomicity predicate indicates whether a transaction occurs within an execution trace without any interleaved labels from other transactions.

Definition 3.2.1. *(Serial Trace) A serial trace is given by the following predicate, inductively defined over an execution* ϵ*:*

$$\frac{\tau, \boldsymbol{\tau} \cup \{\tau\} \vdash \mathsf{serial}\ \epsilon}{\bot, \boldsymbol{\tau} \vdash \mathsf{serial}\ (\tau, \mathsf{beg}) \cdot \epsilon}\ \tau \notin \boldsymbol{\tau} \qquad \frac{\bot, \boldsymbol{\tau} \vdash \mathsf{serial}\ \epsilon}{\tau, \boldsymbol{\tau} \vdash \mathsf{serial}\ (\tau, \mathsf{cmt}) \cdot \epsilon}$$

$$\frac{\tau, \boldsymbol{\tau} \vdash \mathsf{serial}\ \epsilon}{\tau, \boldsymbol{\tau} \vdash \mathsf{serial}\ (\tau, o.m) \cdot \epsilon} \qquad \frac{}{\bot, \boldsymbol{\tau} \vdash \mathsf{serial}\ \circ}$$

This definition states that a trace is serial if each transaction appears in the trace without any interleaved events from other transactions. We now show that the atomic semantics given in Figure 2.1 are serializable:

Lemma 3.2.1. *(Atomic Serializability)*

$$\mathsf{serial}\ \epsilon \quad \forall \epsilon = \mathcal{C}_0^a \rightarrow_A^* \mathcal{C}^a$$

Proof: Trivial since all executions are of the form

$$\begin{gathered}(\tau, \mathsf{beg}) \cdot (\tau, o.m) \cdot (\tau, p.n) \cdots (\tau, \mathsf{cmt}) \cdot \\ (\tau', \mathsf{beg}) \cdot (\tau', o.m) \cdot (\tau', p.n) \cdots (\tau', \mathsf{cmt}) \\ \cdots\end{gathered}$$

For details see our Technical Report [18].

In the following lemma, let $\mathcal{L}_\tau^i$ denote an operation corresponding to transaction τ.

Lemma 3.2.2.

$$\mathsf{serial}\ \epsilon \Rightarrow \mathsf{serial}\ \epsilon \cdot (\tau, \mathsf{beg}) \cdot \mathcal{L}_\tau^0 \cdots \mathcal{L}_\tau^n \cdot (\tau, \mathsf{cmt})$$

when none of the labels in ϵ *correspond to* τ*.*

Proof: By derivation over serial. For details see our Technical Report [18].

3.3 Pessimistic Serializability

We now give our serializability result for the pessimistic semantics. The structure of our proof of serializability of executions is standard. We show that for any execution of the semantics, there exists an equivalent execution in which all transactions appear in a serial order (without interleavings of operations from other transactions).

Left-moverness Invariant
An important part of the correctness of the pessimistic semantics is the fact that an invariant is maintained about method calls of all active transactions. This is a result of the PAPP reduction. We define a predicate on a configuration which holds if all operations of uncommitted transactions are left-movers with respect to all operations of other uncommitted transactions. Note that we use the notation $\mathcal{L}_\tau$ to mean any label involving τ, i.e. $(\tau, \_)$.

Definition 3.3.1. *(Active Left-Movers)*

$$\mathsf{comm}_\triangleleft \triangleq \lambda\epsilon.\forall \mathcal{L}_\tau, \mathcal{L}_{\tau'} \in \epsilon.\mathcal{L}_\tau \mathrel{\widehat{\triangleleft}} \mathcal{L}_{\tau'} \text{ where } \tau \neq \tau'$$

Lemma 3.3.1.

$$\mathsf{comm}_\triangleleft\ \epsilon \Rightarrow \forall \tau.\epsilon \sqsubseteq \epsilon_\tau \cdot \epsilon_{U \setminus \tau}$$

Proof: Double induction (i) showing that for the first τ-label $\mathcal{L}_\tau = (\tau, \_)$ in ϵ, that $\mathsf{comm}_\triangleleft\ \epsilon \Rightarrow \epsilon_a \cdot \mathcal{L}_\tau \cdot \epsilon_b \sqsubseteq \mathcal{L}_\tau \cdot \epsilon_a \cdot \epsilon_b$ and (ii) showing that $\mathsf{comm}_\triangleleft\ \epsilon \Rightarrow \epsilon \sqsubseteq \epsilon_\tau \cdot \epsilon_{U \setminus \tau}$ by induction on the sequence of τ-labels occurring in ϵ. For details see our Technical Report [18].

Definition 3.3.2. *(Active transactions) The function* active *returns the set of transaction identifiers in ϵ which have not committed:*

$$\begin{aligned}
\mathsf{active}(\circ, A) &\triangleq A \\
\mathsf{active}((\tau, \mathsf{beg}) \cdot \epsilon, A) &\triangleq \mathsf{active}(\epsilon, A \cup \{\tau\}) \\
\mathsf{active}((\tau, \mathsf{cmt}) \cdot \epsilon, A) &\triangleq \mathsf{active}(\epsilon, A \setminus \{\tau\}) \\
\mathsf{active}((\tau, \mathsf{abt}) \cdot \epsilon, A) &\triangleq \mathsf{active}(\epsilon, A \setminus \{\tau\}) \\
\mathsf{active}((\tau, \_) \cdot \epsilon, A) &\triangleq \mathsf{active}(\epsilon, A)
\end{aligned}$$

Definition 3.3.3. *(Live methods) The function $\mathsf{live}(\epsilon, \boldsymbol{\tau})$ takes an execution ϵ and set of active transactions $\boldsymbol{\tau}$ and returns the set of operations performed by transactions which have not committed:*

$$\begin{aligned}
\mathsf{live}(\circ, \boldsymbol{\tau}) &\triangleq \emptyset \\
\mathsf{live}((\tau, o.m) \cdot \epsilon, \boldsymbol{\tau}) &\triangleq \begin{cases} \{o.m\} \cup \mathsf{live}(\epsilon, \boldsymbol{\tau}) & \text{if } \tau \in \boldsymbol{\tau} \\ \mathsf{live}(\epsilon, \boldsymbol{\tau}) & \text{otherwise} \end{cases} \\
\mathsf{live}((\_, \_) \cdot \epsilon, \boldsymbol{\tau}) &\triangleq \mathsf{live}(\epsilon, \boldsymbol{\tau})
\end{aligned}$$

Note that the (τ, abt) is not used in the pessimistic semantics, but will appear in the optimistic semantics. The following predicate holds if both an execution trace and a configuration agree on the set of all operations of uncommitted transactions.

Definition 3.3.4. *(Uncommitted consistency)*

$$\mathit{consist}(\epsilon, T) \quad \stackrel{\triangle}{=} \quad \mathit{meths}(T) = \mathit{live}(\epsilon, \mathit{active}(\epsilon, \emptyset))$$

Moverness in Pessimistic Semantics
We now show some moverness relationships which are specific to the pessimistic semantics and used throughout the proof.

Lemma 3.3.2.

$$(\tau, \mathit{beg}) \quad \widehat{\bowtie} \quad (\tau', \_) \tag{3.1}$$
$$(\tau, \mathit{cmt}) \quad \widehat{\lhd} \quad (\tau', \_) \tag{3.2}$$

Proof: By showing that reordering the execution leads to the same final configuration fin. For details see our Technical Report [18].

Note that (τ, cmt) is not a right-mover with $(\tau', \_)$. The proof would fail in the final case for Eqn. 3.2 where

$$[\mathsf{fin}\ C\ (\tau', o.m)\cdot(\tau, \mathsf{cmt})] = \emptyset$$

Moving an operation to the left of a commit may lead to an execution which is not valid since the antecedent of the PAPP rule may not hold. Informally, the transaction performing the operation may need to wait until the other commits.

In the following let $\epsilon_{\neg\tau}$ denote an execution which contains no labels corresponding to τ.

Corollary 3.3.3. *(To Lemma 3.3.1) In the pessimistic semantics,*

$$\mathit{comm}_{\lhd}\ \epsilon_{\neg\tau} \ \Rightarrow\ \epsilon_{\neg\tau}\cdot(\tau, \mathit{cmt}) \sqsubseteq (\tau, \mathit{cmt})\cdot\epsilon_{\neg\tau}$$

Proof: This is an immediate consequence of Lemma 3.3.1, noting the moverness of (τ, cmt) given by Equation 3.2.

Lemma 3.3.4. *(Implied right-moverness)*

$$\mathit{comm}_{\lhd}\ \epsilon \Rightarrow \epsilon\cdot(\tau, o.m)\cdot\epsilon_{\neg\tau} \sqsubseteq \epsilon\cdot\epsilon_{\neg\tau}\cdot(\tau, o.m)$$

Proof: Trivial induction. For details see our Technical Report [18].

Serializability
The next theorem shows that executions of the pessimistic semantics are strictly serializable. The theorem proceeds by showing inclusion in another execution of two parts: a prefix in which all committed transactions appear in a serial order (using serial) and a suffix of uncommitted operations in which $\mathsf{comm}_{\lhd}$ is invariant. We must also maintain as invariant the consistency ($\mathsf{consist}(\epsilon, \mathcal{C}^{\mathsf{p}}.T)$) between operations of uncommitted transactions in the execution trace and operations of uncommitted transactions as given by the configuration. Finally, note that when the PUNDO rule is taken, the corresponding doomed operation is moved to the right and then annihilated by the inverse operations.

Theorem 3.3.5. *(Pessimistic Serializability)*
For any $\epsilon = \mathcal{C}_0^{\mathsf{p}} \rightarrow_P^* \mathcal{C}^{\mathsf{p}}$,

$$\exists \epsilon_C \cdot \epsilon_U . \textit{serial}\ \epsilon_C \wedge \textit{comm}_{\triangleleft}\ \epsilon_U \wedge \textit{consist}(\epsilon_U, \mathcal{C}^{\mathsf{p}}.T) \wedge \epsilon \sqsubseteq \epsilon_C \cdot \epsilon_U$$

Proof: We prove the theorem by induction on the execution trace. constructing an appropriate $\epsilon_C \cdot \epsilon_U$ at each step. Consider the empty execution $\epsilon = \circ$, which starts in configuration $\mathcal{C}_0^{\mathsf{p}}$. We can define $\epsilon_C = \epsilon_U = \circ$. $\mathsf{serial}\ \epsilon_C$ holds by axiom. $\mathsf{comm}_{\triangleleft}\ \epsilon_U$ and $\mathsf{consist}(\epsilon_U, \mathcal{C}^{\mathsf{p}}.T)$ hold trivially. We now show that

$$\begin{gathered}\exists \epsilon_C \cdot \epsilon_U . \epsilon \sqsubseteq \epsilon_C \cdot \epsilon_U \wedge \mathsf{serial}\ \epsilon_C \wedge \mathsf{comm}_{\triangleleft}\ \epsilon_U \wedge \mathsf{consist}(\epsilon_U, \mathcal{C}^{\mathsf{p}}.T) \\ \wedge \epsilon \xrightarrow{\mathcal{L}}_P \epsilon' \\ \Rightarrow \\ \exists \epsilon_C{}' \cdot \epsilon_U{}' . \epsilon' \sqsubseteq \epsilon_C{}' \cdot \epsilon_U{}' \wedge \mathsf{serial}\ \epsilon_C{}' \wedge \mathsf{comm}_{\triangleleft}\ \epsilon_U{}' \wedge \mathsf{consist}(\epsilon_U{}', \mathcal{C}^{\mathsf{p}\prime}.T)\end{gathered}$$

We case split on $\mathcal{L}$.

Case $\mathcal{L} = \bot$: $\epsilon' = \epsilon$. Trivial.

Case $\mathcal{L} = (\tau, \mathbf{beg})$: Let $\epsilon_C{}' = \epsilon_C$ and $\epsilon_U{}' = \epsilon_U \cdot (\tau, \mathsf{beg})$. $\epsilon' \sqsubseteq \epsilon_C{}' \epsilon_U{}' \cdot (\tau, \mathsf{beg})$ by Lemma 3.1.2. $\mathsf{serial}\ \epsilon_C{}'$ holds because $\mathsf{serial}\ \epsilon_C$ holds. $\mathsf{consist}(\epsilon_U{}', \mathcal{C}^{\mathsf{p}\prime}.T)$ holds because no new operations have been performed. Equation 3.1 maintains the truth of $\mathsf{comm}_{\triangleleft}\ \epsilon_U{}'$.

Case $\mathcal{L} = (\tau, o.m)$: Let $\epsilon_C{}' = \epsilon_C$ and $\epsilon_U{}' = \epsilon_U \cdot (\tau, o.m)$. $\epsilon' \sqsubseteq \epsilon_C{}' \epsilon_U{}' \cdot (\tau, o.m)$ by Lemma 3.1.2. $\mathsf{serial}\ \epsilon_C{}'$ holds because $\mathsf{serial}\ \epsilon_C$ holds. By the PAPP rule, $\mathcal{C}^{\mathsf{p}\prime}.T = \mathcal{C}^{\mathsf{p}}.T \cup \{o.m\}$ and it is easy to see that $\mathsf{live}(\epsilon_U{}', \emptyset) = \mathsf{live}(\epsilon_U, \emptyset) \cup \{o.m\}$. Thus $\mathsf{consist}(\epsilon_U{}', \mathcal{C}^{\mathsf{p}\prime}.T)$ holds. $\mathsf{comm}_{\triangleleft}\ \epsilon_U{}'$ holds because $\mathsf{comm}_{\triangleleft}\ \epsilon_U$ and PAPP requires that $\{o.m\} \mathrel{\overline{\triangleleft}} \mathsf{meths}(\mathcal{C}^{\mathsf{p}}.T)$. Lifting the antecedent, we have that

$$\begin{aligned}\forall \mathcal{L}' \in \epsilon_U . \mathcal{L} \mathrel{\widehat{\triangleleft}} \mathcal{L}' \quad &\Longrightarrow \quad \forall \mathcal{L}_1, \mathcal{L}_2 \in \epsilon_U{}' . \mathcal{L}_1 \mathrel{\widehat{\triangleleft}} \mathcal{L}_2 \\ &\Longrightarrow \quad \mathsf{comm}_{\triangleleft}\ \epsilon_U{}'\end{aligned}$$

Case $\mathcal{L} = (\tau, o.m^{-1})$: Without loss of generality, let $\epsilon_U = \epsilon_a \cdot (\tau, o.m) \cdot \epsilon_{\neg\tau}$. By Lemma 3.3.4 we know that $\epsilon_U \sqsubseteq \epsilon_a \cdot \epsilon_{\neg\tau} \cdot (\tau, o.m)$. Now we append $(\tau, o.m^{-1})$ to both sides, and then chose $\epsilon_U{}' = \epsilon_a \cdot \epsilon_{\neg\tau}$.

Case $\mathcal{L} = (\tau, \mathbf{cmt})$: We first show the following inclusion:

$$\begin{array}{rcll} \epsilon' & = & \epsilon \cdot (\tau, \mathsf{cmt}) & \\ & \sqsubseteq & \epsilon_C \cdot \epsilon_U \cdot (\tau, \mathsf{cmt}) & \text{I.H. } \epsilon \sqsubseteq \epsilon_C \cdot \epsilon_U \\ & \sqsubseteq & \epsilon_C \cdot \epsilon_\tau \cdot \epsilon_{U \setminus \tau} \cdot (\tau, \mathsf{cmt}) & \text{Lemma 3.3.1} \\ & \sqsubseteq & \epsilon_C \cdot \epsilon_\tau \cdot (\tau, \mathsf{cmt}) \cdot \epsilon_{U \setminus \tau} & \text{Corollary 3.3.3} \end{array}$$

Now let $\epsilon_C{}' = \epsilon_C \cdot \epsilon_\tau \cdot (\tau, \mathsf{cmt})$. $\mathsf{serial}\ \epsilon_C{}'$ holds by Lemma 3.2.2. Letting $\epsilon_U{}' = \epsilon_{U \setminus \tau}$, observe that $\epsilon_U{}'$ is a subsequence of ϵ_U, so $\mathsf{comm}_{\triangleleft}\ \epsilon_U{}'$ holds.

3.4 Optimistic Serializability

We now prove serializability of the optimistic model. Unfortunately we cannot reuse the proof of the pessimistic model. The optimistic model is fundamentally different: shared memory is not modified in-place. Instead transactions modify local copies of memory and update the shared state at commit.

The optimistic execution model performs synchronization at *commit-time* rather than synchronizing with each operation. The effect is two-fold: (i) If a transaction aborts then recovery is achieved by simply discarding the transaction-local copy rather than invoking inverse operations. (ii) When a transaction τ commits it must check that its operations can be moved to the right of operations corresponding to any transaction τ' which has committed after τ began. This is why, for example, in TL2 [7] at commit-time transactions acquire a lock for each modified memory location.

Moverness in Optimistic Semantics

Lemma 3.4.1. *In the optimistic semantics,*

$$\begin{aligned}
(\tau, \mathsf{beg}) &\;\hat{\bowtie}\; (\tau', \mathsf{beg}) && (3.3)\\
(\tau, \mathsf{beg}) &\;\hat{\bowtie}\; (\tau', o.m) && (3.4)\\
(\tau, o.m) &\;\hat{\bowtie}\; (\tau', p.n) && (3.5)\\
(\tau, \mathsf{cmt}) &\;\hat{\bowtie}\; (\tau', o.m) && (3.6)\\
(\tau, \mathsf{cmt}) &\;\hat{\lhd}\; (\tau', \mathsf{beg}) && (3.7)\\
(\tau, \mathsf{cmt}) &\;\cdot\; (\tau', \mathsf{cmt}) && (3.8)\\
(\tau, \mathsf{beg}) &\;\hat{\rhd}\; (\tau', \mathsf{abt}) && (3.9)\\
(\tau, \mathsf{abt}) &\;\hat{\rhd}\; (\tau', \mathsf{abt}) && (3.10)
\end{aligned}$$

Proof: For details see our Technical Report [18].

The proof of the above Lemma involves showing that reordering the execution leads to the same final configuration fin. Eqn. 3.3 holds because swapping the order of which transaction begins first has no effect on the eventual commit order. Eqns. 3.4 and 3.5 hold because $o.m$ operations are applied to transaction-local copies of the shared state and does not impact concurrent transactions. Eqn. 3.7 holds because moving (τ, cmt) to the left of (τ', beg) reduces the burden on τ' when it ultimately tries to commit—τ' begins with a newer copy of the shared state and can ignore the effects of τ when scanning the commit log. Eqn. 3.8 holds because if a commit is moved to the left it may prevent the other transaction from committing since the commit log will be longer. Finally, Eqns. 3.10 and 3.9 hold because the OABT rule discards transaction-local changes.

Aborting a Transaction

Lemma 3.4.2. *In the optimistic semantics,*

$$(\tau, \mathsf{beg}) \cdot \epsilon_\tau \cdot (\tau, \mathsf{abt}) \sqsubseteq \circ$$

where ϵ_τ is an execution consisting of labels of the form $(\tau, o.m)$.

Proof: First we show that this holds when $\epsilon_\tau = \circ$, which requires us to show that $[\mathsf{fin}\ C\ (\tau, \mathsf{beg})\cdot(\tau, \mathsf{abt})] \subseteq [\mathsf{fin}\ C\ \circ]$. Or rather that $[\mathsf{fin}\ C\ (\tau, \mathsf{beg})\cdot(\tau, \mathsf{abt})] = C$. From the optimistic semantics it is easy to see that the OABT reduction restores the state which held before the OBEG reduction fired.

The lemma can now be proven by a rather trivial induction on ϵ_τ. In the base case when $\epsilon_\tau = (\tau, o.m)$, the claim above still holds because OAPP1 and OAPP2 only perform changes to transaction-local σ_τ and ℓ_τ, which are discarded by the OABT reduction. This of course holds for any larger ϵ_τ.

Lemma 3.4.3. *In the optimistic semantics,*

$$\epsilon\cdot(\tau, \mathit{beg})\cdot\epsilon_U\cdot(\tau, \mathit{abt}) \sqsubseteq \epsilon\cdot\epsilon_{U\setminus\tau}$$

Proof: Using induction and the moverness give in Eqns. 3.4 and 3.6, it is easy to show that

$$\epsilon\cdot(\tau, o.m)\cdot\epsilon_{\neg\tau} \sqsubseteq \epsilon\cdot\epsilon_{\neg\tau}\cdot(\tau, o.m)$$

where $\epsilon_{\neg\tau}$ consists of labels of the form $(\tau', o.m)$ or (τ', beg). We then show that the set of all n τ-labels in ϵ, that

$$\epsilon\cdot\mathcal{L}^n_\tau\cdot\epsilon^n_{\neg\tau}\cdots\mathcal{L}^1_\tau\cdot\epsilon^1_{\neg\tau} \sqsubseteq \epsilon\cdot\epsilon^n_{\neg\tau}\cdots\epsilon^1_{\neg\tau}\cdot\mathcal{L}^n_\tau\cdots\mathcal{L}^1_\tau$$

For details see our Technical Report [18].

Serializability

We prove the theorem again by showing that executions are contained within an alternate serial execution. However, there are some differences. Since transactions operate on thread-local state, the $(\tau, o.m)$ labels appear but do not correspond to modifications to the shared state. We are therefore free to discard these labels (so long as a commit label has not appeared). Thus when a transaction takes the OCMT step it can move all of its operations to the right of other committed transactions because of the antecedent and it need not consider other uncommitted transactions (because we temporarily abort them).

The proof depends on the following lemmata. We introduce the notation $\epsilon_{\{\tau,\tau',\ldots\}}$ to refer to an execution consisting of operations performed by τ,τ', or any other transactions. The notations $\epsilon_{\{\tau,\ldots\}}$, $\epsilon_{\{\tau,\tau'\}}$ and $\epsilon_{\{\tau',\ldots\}}$ are similar.

Lemma 3.4.4. *(Right of committed) If $\ell_\tau \mathrel{\bar{\rhd}} \ell_{\tau'}$ then*

$$\begin{gathered}\epsilon\cdot\epsilon_{\{\tau,\tau',\ldots\}}\cdot(\tau', \mathit{cmt})\cdot(\tau, \mathit{cmt})\cdot\epsilon' \\ \sqsubseteq \\ \epsilon\cdot\epsilon_{\{\tau',\ldots\}}\cdot(\tau', \mathit{cmt})\cdot\epsilon_{\{\tau\}}\cdot(\tau, \mathit{cmt})\cdot\epsilon'\end{gathered}$$

Proof: First, by induction we can move all τ operations to the right within $\epsilon_{\{\tau,\tau'\}}$ because of Eqn. 3.5. Then each $(\tau, o.m)$ can move to the right of (τ', cmt) because $\ell_\tau \mathrel{\bar{\rhd}} \ell_{\tau'}$. For details see our Technical Report [18].

Lemma 3.4.5. *(Move ops right)*

$$\begin{gathered}\epsilon\cdot(\tau, \mathit{beg})\cdot\epsilon_{\{\tau_1,\ldots,\tau_n\}}\cdot(\tau_1, \mathit{cmt})\cdot\epsilon_{\{\tau_2,\ldots,\tau_n\}}\cdot(\tau_2, \mathit{cmt})\cdots(\tau_n, \mathit{cmt})\cdot\epsilon_\tau \\ \sqsubseteq \\ \epsilon\cdot(\tau, \mathit{beg})\cdot\epsilon_{\{\tau,\tau_1,\ldots,\tau_n\}}\cdot(\tau_1, \mathit{cmt})\cdot\epsilon_{\{\tau,\tau_2,\ldots,\tau_n\}}\cdot(\tau_2, \mathit{cmt})\cdots(\tau_n, \mathit{cmt})\end{gathered}$$

Proof: Induction on n, using Lemma 3.4.4. For details see our Technical Report [18].

Lemma 3.4.6. *(Move* beg *right) Let $\epsilon^{oc}_{\overline{\tau}}$ be an execution consist of operations and commits of transactions other than τ. If $\ell_\tau \mathrel{\overline{\triangleright}} \ell_{\tau'}$ for all $\tau' \in \overline{\tau}$ then*

$$\epsilon{\cdot}(\tau, \mathit{beg}){\cdot}\epsilon^{oc}_{\overline{\tau}}{\cdot}\epsilon_{\{\tau\}}{\cdot}(\tau, \mathit{cmt}) \sqsubseteq \epsilon{\cdot}\epsilon^{oc}_{\overline{\tau}}{\cdot}(\tau, \mathit{beg}){\cdot}\epsilon_{\{\tau\}}{\cdot}(\tau, \mathit{cmt})$$

Proof: Induction and Eqn. 3.7. For details see our Technical Report [18].

Definition 3.4.1. *(Abort Factory)*

$$\epsilon_{\mathit{abt}}(\epsilon, A) \triangleq (\tau_1, \mathit{abt}){\cdot}\cdots(\tau_n, \mathit{abt}) \;\; \forall \tau_i \in \mathit{active}(\epsilon, \emptyset)$$

Theorem 3.4.7. *(Optimistic Serializability)*
For any $\epsilon = \mathcal{C}^{\mathit{opt}}_0 \rightarrow^*_O \mathcal{C}^{\mathit{opt}}$,

$$\exists \epsilon_S.\ \mathit{serial}\ \epsilon_S \epsilon_{\mathit{abt}}(\epsilon, \emptyset) \wedge \epsilon{\cdot}\epsilon_{\mathit{abt}}(\epsilon, \emptyset) \sqsubseteq \epsilon_S \epsilon_{\mathit{abt}}(\epsilon, \emptyset)$$

Proof: By induction on the execution trace, where all uncommitted transactions have been removed (i.e. aborted). Specifically, at each step of the proof we construct an appropriate serial ϵ_S for which we can show ϵ is included so long as both are appended by $\epsilon_{\mathsf{abt}}(\epsilon_U, \emptyset)$.

Base case: Consider the empty execution $\epsilon = \circ$, which starts in configuration $\mathcal{C}^{\mathsf{opt}}_0$. We can define $\epsilon_S = \circ$. serial ϵ_S holds by axiom, and $\sqsubseteq$ holds trivially. We now show that

$$\begin{gathered}\exists \epsilon_S.\ \mathsf{serial}\ \epsilon_S{\cdot}\epsilon_{\mathsf{abt}}(\epsilon, \emptyset) \wedge \epsilon{\cdot}\epsilon_{\mathsf{abt}}(\epsilon, \emptyset) \sqsubseteq \epsilon_S{\cdot}\epsilon_{\mathsf{abt}}(\epsilon, \emptyset) \wedge \epsilon \xrightarrow{\mathcal{L}}_O \epsilon' \\ \Rightarrow \\ \exists \epsilon_S'.\ \mathsf{serial}\ \epsilon_S'{\cdot}\epsilon_{\mathsf{abt}}(\epsilon', \emptyset) \wedge \epsilon'{\cdot}\epsilon_{\mathsf{abt}}(\epsilon', \emptyset) \sqsubseteq \epsilon_S'{\cdot}\epsilon_{\mathsf{abt}}(\epsilon_S', \emptyset)\end{gathered}$$

Now case split on $\mathcal{L}$.

Case $\mathcal{L} = \bot$: $\epsilon' = \epsilon$. Trivial.

Case $\mathcal{L} = (\tau, \mathbf{beg})$: Let $\epsilon_S' = \epsilon_S{\cdot}(\tau, \mathsf{beg})$.
The predicate serial $\epsilon_S'{\cdot}\epsilon_{\mathsf{abt}}(\epsilon', \emptyset)$ is equivalent (by Lemma 3.4.3) to serial $\epsilon_S{\cdot}\epsilon_{\mathsf{abt}}(\epsilon, \emptyset)$ which holds by induction. Again using Lemma 3.4.3 and the inductive hypothesis, the other obligation holds:

$$\epsilon'{\cdot}\epsilon_{\mathsf{abt}}(\epsilon', \emptyset) \sqsubseteq \epsilon_S'{\cdot}\epsilon_{\mathsf{abt}}(\epsilon_S', \emptyset)$$

Case $\mathcal{L} = (\tau, \mathsf{cmt})$: First we show

$$\begin{array}{lll} \epsilon{\cdot}(\tau, \mathsf{cmt}){\cdot}\epsilon_{\mathsf{abt}}(\epsilon^1, \emptyset) & & \\ \quad \sqsubseteq \quad \epsilon_S{\cdot}(\tau, \mathsf{cmt}){\cdot}\epsilon_{\mathsf{abt}}(\epsilon^1, \emptyset) & & \\ \quad \sqsubseteq \quad \epsilon_{\{\tau', U\}}{\cdot}(\tau, \mathsf{beg}){\cdot}\epsilon^1_{\{\tau, \tau', U\}}{\cdot}(\tau, \mathsf{cmt}){\cdot}\epsilon_{\mathsf{abt}}(\epsilon^1, \emptyset) & \text{(Rewrite)} \\ \quad \sqsubseteq \quad \epsilon_{\{\tau'\}}{\cdot}(\tau, \mathsf{beg}){\cdot}\epsilon^1_{\{\tau, \tau'\}}{\cdot}(\tau, \mathsf{cmt}) & \text{(Lm. 3.4.3)} \\ \quad \sqsubseteq \quad \epsilon_{\{\tau'\}}{\cdot}\epsilon^1_{\{\tau'\}}{\cdot}(\tau, \mathsf{beg}){\cdot}\epsilon^1_{\{\tau\}}{\cdot}(\tau, \mathsf{cmt}) & \end{array}$$

The last step is accomplished by induction with Lemmas 3.4.5 and 3.4.6. Now let $\epsilon' = \epsilon_{\{\tau'\}}\epsilon^1_{\{\tau'\}}{\cdot}(\tau, \mathsf{beg}){\cdot}\epsilon^1_{\{\tau\}}{\cdot}(\tau, \mathsf{cmt})$. By the inductive hypothesis, serial $\epsilon_{\{\tau'\}} \cdot \epsilon^1_{\{\tau'\}}$ holds and it is easy to show that serial $(\tau, \mathsf{beg}){\cdot}\epsilon^1_{\{\tau\}}{\cdot}(\tau, \mathsf{cmt})$ holds. Thus serial ϵ' holds.

Case $\mathcal{L} = (\tau, \textsf{abt})$: Let ${\epsilon_S}' = \epsilon_S \cdot (\tau, \textsf{abt})$. Predicate $\textsf{serial}\ {\epsilon_S}' \cdot \epsilon_{\textsf{abt}}(\epsilon', \emptyset)$ holds because $\textsf{serial}\ \epsilon_S {\cdot} \epsilon_{\textsf{abt}}(\epsilon, \emptyset)$ holds and appending an abort τ can be removed by Lemma 3.4.3 (and $\epsilon_{\textsf{abt}}$ now has a lesser obligation). So it is also trivial to show that

$$\epsilon' {\cdot} \epsilon_{\textsf{abt}}(\epsilon', \emptyset) \sqsubseteq {\epsilon_S}' {\cdot} \epsilon_{\textsf{abt}}(\epsilon', \emptyset)$$

Case $\mathcal{L} = (\tau, o.m)$: Let ${\epsilon_S}' = \epsilon_S \cdot (\tau, o.m)$. Predicate $\textsf{serial}\ {\epsilon_S}' \cdot \epsilon_{\textsf{abt}}(\epsilon', \emptyset)$ holds because $\textsf{serial}\ \epsilon_S {\cdot} \epsilon_{\textsf{abt}}(\epsilon, \emptyset)$ holds and appending a single operation adds an event that will be removed by $\epsilon_{\textsf{abt}}$. Using similar reasoning and we can show that

$$\epsilon' {\cdot} \epsilon_{\textsf{abt}}(\epsilon', \emptyset) \sqsubseteq {\epsilon_S}' {\cdot} \epsilon_{\textsf{abt}}(\epsilon', \emptyset)$$

Chapter 4

Applications

4.1 Traditional (Fine-grained) STM

Most transactional memory systems [7, 27] use the optimistic semantics at the fine-grained level of memory read/write operations. Transactions concurrently execute operations and when they try to commit they either succeed, in which case their effects atomically become visible, or abort, in which case their effects are discarded and the transactions restarts.

Formally, our optimistic semantics uses per-transaction copies of the shared state. For performance reasons, however, many transactional memory implementations do in-place modifications of the shared state. They rely on *undo*-logs (rather than our *intention*-logs) to reverse the effects of aborted transactions. Since the operations (`read` and `write` memory locations) are simple, it is easy to keep undo-logs which consist of previous memory values.

We now formalize traditional transactional memory using our semantics. Let $\mathcal{L} : \mathbb{N}$ be a set of locations and $\mathcal{V} : \mathbb{N}$ be a set of values. The shared store σ_{sh} consists of a single object η representing a memory, as well as per-transaction return values: $\sigma_{\text{sh}} = \langle \eta, r_{\tau_1}, r_{\tau_2}, ... \rangle$. We define the two method calls as follows:

$$\begin{aligned}
[\![\tau_i : \eta.\texttt{read}\ x]\!]\sigma_{\text{sh}} &= \sigma_{\text{sh}}[r_{\tau_i} \mapsto \eta[x]] \\
[\![\tau_i : \eta.\texttt{write}\ x\ v]\!]\sigma_{\text{sh}} &= \sigma_{\text{sh}}[\eta[x \mapsto v]]
\end{aligned}$$

When transaction τ_i reads x from the shared store, the r_{τ_i} component of the shared store is updated to contain the current value of x in η. The write operation updates η such that the value of x is v. Moverness is intuitive (let x, y and v, w each be distinct and $\cdot$ means neither $\lhd$ nor $\rhd$):

$$\eta.\texttt{read}\ x \quad \bowtie \quad \eta.\texttt{read}\ y \tag{4.1}$$
$$\eta.\texttt{write}\ x\ v \quad \bowtie \quad \eta.\texttt{write}\ y\ w \tag{4.2}$$
$$\eta.\texttt{read}\ x \quad \bowtie \quad \eta.\texttt{write}\ y\ v \tag{4.3}$$
$$\eta.\texttt{read}\ x \quad \bowtie \quad \eta.\texttt{read}\ x \tag{4.4}$$
$$\eta.\texttt{write}\ x\ v \quad \bowtie \quad \eta.\texttt{write}\ x\ v \tag{4.5}$$
$$\eta.\texttt{write}\ x\ v \quad \cdot \quad \eta.\texttt{write}\ x\ w \tag{4.6}$$
$$\eta.\texttt{read}\ x \quad \cdot \quad \eta.\texttt{write}\ x\ v \tag{4.7}$$

Existing transactional memory implementations mostly abide by the above rules. However they typically do not consider Eqn. 4.5 to be a both-mover.

Invisible reads Invisible reads [12, 32] and other forms of read-only access in transactional memory implementations exploit Eqns. 4.1 and 4.4. Since `read` operations are always both movers with respect to each other, it is always sound to allow read-only workloads to interleave in any order. Implementations have various ways of handling the advent of `write` operations in such workloads.

Eager Conflict Detection in TL2 The TL2 implementation involves an added optimization: eager conflict detection. Rather than keeping a thread-local view of memory and checking conflicts at commit, with each read/write operation, TL2 checks the location timestamp and compares it with the transaction-local begin timestamp. Transactions are aborted if a conflict is detected – conflicts correspond to any instance of read/write operations on the same location (Eqns. 4.4, 4.5, 4.6, 4.7). Eager conflict detection can be over-approximated in our semantics as simply a non-deterministic choice to take the OABT rule.

Checkpoints and Nested Transactions Our semantics is amenable to the "checkpoints" alternative [16] to nested transactions. We intentionally designed our semantics to be non-deterministic: the UNDO rule can be invoked at any point during a transaction, allowing the implementation to decide how many times to apply the UNDO rule. Checkpoints allow us to model (i) nested transactions [16] and (ii) mechanisms which escape deadlock [17] as described next.

Deadlock The pessimistic semantics may deadlock. Consider a configuration in which T includes $\langle \tau, o.m;\ C, \{p.n\}, \sigma\rangle$ and $\langle \tau', p.n';\ C', \{o.m'\}, \sigma'\rangle$. If neither $o.m \triangleleft o.m'$ nor $p.n \triangleleft p.n'$, then neither transaction will be able to take the PAPP step in the semantics.

There are implementation techniques for detecting such deadlock [17]. These techniques can recover from deadlock by performing either a full abort (reverting all of a transaction's operations through multiple instances of UNDO) or partial abort (reverting a subset of operations). Ideally, recovery would only revert the operations necessary to allow another transaction to make progress.

4.2 Opacity

In addition to serializability, recent work has proposed a new correctness condition called *opacity* [9] which holds for most modern implementations of transactional memory [7, 27]. Opacity involves serializability as well as two new requirements. We now show that both of our execution semantics are opaque.

Definition 4.2.1. *(Opacity) An execution ϵ is opaque, denoted* $\mathsf{opaque}\ \epsilon$ *if and only if*

1. *ϵ is serial ($\mathsf{serial}\ \epsilon$).*
2. *Operations performed by an aborted transaction are never visible to other transactions.*

3. Every transaction observes a consistent version of the state.

Proposition 4.2.1. *(Opacity of Pessimistic Execution) For any execution ϵ given by $\rightarrow_P$, opaque ϵ.*

Proof: Intuitively, the proposition holds because transactions only perform PAPP if all concurrent operations (including those of transactions which have or may abort) are left-movers, the transaction logically only observes the state of the system at commit-time. For details see our Technical Report [18].

Proposition 4.2.2. *(Opacity of Optimistic Execution) For any execution ϵ given by $\rightarrow_O$, opaque ϵ.*

Proof: Follows from the fact that the shared state is updated atomically in OCMT and that transactions duplicate the shared state in OBEG. For details see our Technical Report [18].

4.3 Coarse-grained Implementations

4.3.1 Pessimistic Coarse-Grained (Transactional Boosting)

In recent work we have given a methodology for enabling transactional access to concurrent data structures, say a ConcurrentSkipList, which already have built-in thread synchronization [11]. The methodology is based on pessimism, and allows us to use such off-the-self concurrent objects in a transactional setting without the pitfalls that would come with fine-grained synchronization.

The methodology also makes one particular choice about how coarse-grained synchronization is accomplished: abstract locks. The user defines an abstract locking discipline such that, for any two transactions executing concurrent method calls, the locks conflict if the method calls do not commute. Here is an example:

```
global SkipList Set;
global AbsLock keys[10];
```

```
T1: atomic {
  with(lock(keys[5])) {
    Set.add(5);
    with(lock(keys[3])) {
      Set.remove(3);
    }}}
```

```
T2: atomic {
  with(lock(keys[6])) {
    if(Set.contains(6)) {
      with(lock(keys[7])) {
        Set.remove(7);
    }}}}
```

The above example involves a SkipList implementation of a Set of integers. Set has the standard operations add(x), remove(y), and contains(z). add(x) commutes with add(y) if

x and y are distinct. An example locking discipline (as above) is to define one abstract lock per key (i.e. x, y, z). Thus, the abstract locks for $\mathsf{add}(x)$ and $\mathsf{add}(y)$ do not conflict when x and y are distinct, so these calls can proceed in parallel.

With our semantics in place, we can formalize the examples we described earlier [11]. Moreover, we can be more precise and consider $\lhd$ and $\rhd$ in addition to $\bowtie$ (commutativity):

- *Multi-set* – The moverness of a multi-set is similar to the commutativity of the Set given in Figure 1 of our previous work [11]. However, a multi-set has the following additional movers:

$$\mathsf{contains}(x)/\mathsf{true} \rhd \mathsf{insert}(x)/\mathsf{true}$$
$$\mathsf{remove}(x)/\mathsf{true} \rhd \mathsf{insert}(x)/\mathsf{true}$$

- *Priority Queue (Heap)* – We gave the commutativity of a priority queue in Figure 4 of our previous work [11]. The method $\mathsf{insert}(x)$ inserts an element into the heap and the method $\mathsf{removeMin}()/y$ returns the minimal element. Figure 4 of [11] can be refined:

$$\mathsf{insert}(x) \lhd \mathsf{removeMin}()/x$$

  In fact we noticed a mistake. Figure 4 of [11] claims that

$$\mathsf{insert}(x) \bowtie \mathsf{removeMin}()/y \quad \forall y \leq x$$

  but when $x = y$, only $\lhd$ holds.

- *Queue* – Similar to the multi-set, a queue has an operation which is a left-mover but not a right-mover:
  $\mathsf{enqueue}(x) \lhd \mathsf{dequeue}()$

- *Semaphore* – The standard semaphore is represented as an integer whose value is initially 0 with methods incr (increments the value) and decr (decrements the value). Applying decr when the value is 0 is either undefined or blocks. Therefore,
  $\mathsf{incr} \lhd \mathsf{decr}$.

4.3.2 Optimistic Coarse-Grained (Baldassin and Burckhardt)

Recently, Baldassin and Burckhardt proposed a new programming model which resembles coarse-grained transactions [3]. In their experience adapting a game application to transactions, they describe a technique based on user-provided specifications (i.e. moverness) to resolve conflicts.

We now discuss their implementation, using parenthetical comments to refer back to the corresponding rule in our optimistic execution semantics in Section 2.3. Baldassin and Burckhardt describe transactions which begin (OBEG) with a replica (σ_τ) of the shared state (σ_{sh}). Transactions then execute performing operations (OAPP) on their local copies, and when they commit their local updates are "propagated to all other replicas" (logically equivalent to $\mathsf{merge}(\sigma_{\mathrm{sh}}, \ell_\tau)$ in OCMT) and when transactional operations are read-only, a single shared state is used (direct equivalence to $\mathsf{merge}(\sigma_{\mathrm{sh}}, \ell_\tau)$).

We now give a formal treatment for the objects described [3]. In the following, the first two are base objects where any write is treated as a conflict (i.e. non-moverness) while concurrent reads are acceptable (i.e. both-moverness). The remaining three use higher-order moverness.

- *Values* (base)
  $\mathsf{get} \bowtie \mathsf{get} \quad \mathsf{set} \cdot \mathsf{get} \quad \mathsf{set} \cdot \mathsf{set}$
- *Collections* (base)
  $\mathsf{GetRO} \bowtie \mathsf{GetRO} \quad \mathsf{GetRW} \cdot \mathsf{GetRO} \quad \mathsf{GetRW} \cdot \mathsf{GetRW}$
- *SharedGrowingList* (wrapper) – This object appears to be a multiset where the order of element insertion does not matter:
  $\mathsf{insert}\ (x) \bowtie \mathsf{insert}\ (y)$ for all x, y.
- *SharedObjectWithPriority* (wrapper) – This object allows transactions to associate a priority with the operations performed on an object and at commit, conflicts are resolved according to priority. Note that this violates serializability, and may rely on application-specific invariants or behaviors for soundness.
- *SharedObjectWithMerge* (wrapper) – This general wrapper allows a user to define a commit-time merge procedure which computes new values in the shared store (σ_{sh}) given the values desired by committing transactions ($\sigma_\tau, \sigma_{\tau'}$, etc.). This can be thought of as a specific form of applying the operational log ℓ_τ to the shared state in $\mathsf{merge}(\sigma_{\text{sh}}, \ell_\tau)$.
  Since this merge procedure is provided, the assumption is that all transactional operations on these objects are both-movers or, in case of conflicts, operations can be inverted without aborting the transaction.

In the game application concurrent transactions are aligned on a per-frame basis, all beginning and committing (roughly) simultaneously. They must assume that any begin order was possible and thus all operations must be right movers with respect to each other – i.e. every operation must be a both-mover. By adding begin timestamps this requirement could be relaxed, and operations need only move to the right of committed operations. In a more general setting (not aligned to frames) right moverness may be useful.

Chapter 5

Related Work

Our work is the first formal treatment of *coarse-grained transactions*, which captures the trend in the recent literature away from low-level memory synchronization towards more efficient, higher-level synchronization. Our semantics, encompassing both pessimistic and optimistic execution models, unifies a wide range of ad-hoc techniques: traditional transactional memory [7, 27], transactional boosting [11], and recent optimistic techniques [3]. Moreover, our work provides a formal basis for an emerging new wave of research [26, 2] which uses automated techniques to discover moverness.

Transactional Boosting In prior work [11] we proved the correctness of Transactional Boosting using histories of events [31, 14]. Boosting is pessimistic and the proof in this paper of the pessimistic execution semantics roughly corresponds to the previous proof [11] when labels are aligned with events. Nonetheless, our treatment here has novelties:

1. We define a coarse-grained language of programs (rather than a library-style methodology [11]) and provide a formal proof of serializability for this language.
2. We have added support for non-determinism, showing equivalence by containment of an execution in a set of serial executions, rather than by finding a single serial execution.
3. We have added a second, optimistic semantics (boosting is purely pessimistic) which models modern read/write STMs such as TL2 [7] and Intel's STM compiler [27], as well as recent attempts at larger granularity [3] as described in Section 4.3.2.

Previous Semantics of STM Previous work on semantics of transactions are orthogonal and have focused instead on *isolation* of transactions (i.e. the privatization problem [29]). These semantics are purely fine-grained, and tackle the issue of ensuring shared memory is not accessed outside of a transaction. By contrast, in our semantics the shared/local distinction is assumed and we focus on the correctness criteria for coarser granularities.

For example, the AtomsFamily presented by Moore and Grossman [22] involves a taxonomy of semantics which vary based on degrees of isolation, availability of abort, and allowed parallel nesting. Another example is the semantics of Automatic Mutual Exclusion (AME) given by Abadi *et al.* [1]. The programming model of AME is substantially different from ours; its execution model involves spawning asynchronous method calls which are scheduled such that only non-conflicting code runs concurrently. This programming model can be used to model fine-grained optimistic implementations, with "eager" updates. We next discuss the distinction between eager and lazy.

Another important distinction is the choice of *which form of optimism* to model: eager or lazy updates to memory. The eager version may perform better but is subject to transactions observing inconsistent states (i.e. not opaque [9] as discussed in Section 4.2), which appears to have significant drawbacks. AME discusses distinction and why they chose to model lazy updates. The AtomsFamily explores both lazy (StrongBasic) and eager (Weak), using a type system to ensure serializability in the latter case.

In our work, lazy updates are fundamentally part of the model (in $\rightarrow_O$ the shared store is only updated in the OCMT rule). Our model matches that of TL2 [7] and Intel's transactional memory compiler [27].

Commutativity The notion of commutativity, more precisely moverness [21], is fundamental in databases and parallel programming. The literature includes Bernstein [5], Steele [30], Weihl [31], Schwartz and Spector [28], Beeri [4], and Korth [15]. More recently, the Jade programming language [25, 20] has been developed which includes support for programmer-provided specifications as to whether certain functions are commutative.

In the very recent literature there have been some techniques which attempt to detect commutativity. This work is inspired by Rinard and Diniz [26] whose *commutativity analysis* checked for concrete equivalence mostly via heap disjointness, and leveraged some limited arithmetic identities. Recently, Aleen and Clark presented a commutativity analysis [2]. Their work is a whole-program analysis for checking whether context-sensitive procedure invocations commute. To check equivalence of re-orderings they use random interpretation. Their work is orthogonal to ours, and could be used to detect or prove moverness of certain objects. Some effort may be needed since (1) their work is for parallelizing compilers, not transactions and (2) precision could be improved by considering left- and right-moverness rather than simply both-moverness.

Chapter 6

Conclusion

In the recent literature there has been a new trend of broadening transactional operations from fine-grained memory operations to more coarse-grained operations, often drastically improving performance. We have presented a semantics which unifies a broad range of such ad-hoc techniques, encompassing both pessimistic (e.g. transactional boosting) and optimistic (e.g. traditional STM and recent coarse-grained work [3]) execution semantics. We have proven serializability and found that the choice of execution semantics imposes different requirements, expressed in terms of left- and right-movers, on the coarse-grained operations themselves. Our work serves also as a formal foundation for recent automated techniques for discovering moverness [26, 2]. We expect that this will be a promising research direction for the future.

Acknowledgments We thank Byron Cook, Tim Harris, Michael Hicks and Peter Sewell for their comments on an earlier draft. We acknowledge funding from a Gates scholarship (Koskinen), a Royal Academy of Engineering Research Fellowship (Parkinson), and NSF grant 0811289 (Herlihy).

Bibliography

[1] Abadi, M., Birrell, A., Harris, T., and Isard, M. Semantics of transactional memory and automatic mutual exclusion. In *The 35th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages (POPL'08)* (2008), G. C. Necula and P. Wadler, Eds., ACM, pp. 63–74.

[2] Aleen, F., and Clark, N. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)* (2009), M. L. Soffa and M. J. Irwin, Eds., ACM, pp. 241–252.

[3] Baldassin, A., and Burckhardt, S. Lightweight software transactions for games. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism (HotPar'09)* (2009).

[4] Beeri, C., Bernstein, P., Goodman, N., Lai, M.-Y., and Shasha, D. A concurrency control theory for nested transactions (preliminary report). In *Proceedings of the 2nd annual ACM symposium on Principles of distributed computing (PODC'83)* (New York, NY, USA, 1983), ACM Press, pp. 45–62.

[5] Bernstein, A. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers 15*, 5 (1966), 757–763.

[6] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)* (New York, NY, USA, 2006), ACM Press, pp. 336–346.

[7] Dice, D., Shalev, O., and Shavit, N. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)* (September 2006).

[8] Flanagan, C., Freund, S. N., and Yi, J. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI'08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2008), ACM, pp. 293–303.

[9] Guerraoui, R., and Kapalka, M. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP'08)* (New York, NY, USA, 2008), ACM, pp. 175–184.

[10] Harris, T., Marlow, S., Jones, S. L. P., and Herlihy, M. Composable memory transactions. *Commun. ACM 51*, 8 (2008), 91–100.

[11] Herlihy, M., and Koskinen, E. Transactional boosting: A methodology for highly concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'08)* (2008).

[12] Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual symposium on Principles of distributed computing (PODC'03)* (2003), ACM Press, pp. 92–101.

[13] Herlihy, M., and Moss, J. E. B. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)* (1993), ACM Press, pp. 289–300.

[14] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[15] Korth, H. F. Locking primitives in a database system. *J. ACM 30*, 1 (1983), 55–79.

[16] Koskinen, E., and Herlihy, M. Checkpoints and continuations instead of nested transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures (SPAA'08)* (New York, NY, USA, 2008), ACM, pp. 160–168.

[17] Koskinen, E., and Herlihy, M. Dreadlocks: efficient deadlock detection. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures (SPAA'08)* (New York, NY, USA, 2008), ACM, pp. 297–303.

[18] Koskinen, E., Parkinson, M., and Herlihy, M. Coarse-grained transactions. Tech. Rep. 759, Computer Laboratory, University of Cambridge, 2009.

[19] Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., and Chew, L. P. Optimistic parallelism requires abstractions. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)* (2007), J. Ferrante and K. S. McKinley, Eds., ACM, pp. 211–222.

[20] Lam, M., and Rinard, M. Coarse-grain parallel programming in Jade. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'91)* (1991), ACM New York, NY, USA, pp. 94–105.

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

[22] Moore, K. F., and Grossman, D. High-level small-step operational semantics for transactions. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'08)* (New York, NY, USA, 2008), ACM, pp. 51–62.

[23] Moravan, M. J., Bobba, J., Moore, K. E., Yen, L., Hill, M. D., Liblit, B., Swift, M. M., and Wood, D. A. Supporting nested transactional memory in logtm. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)* (New York, NY, USA, 2006), ACM Press, pp. 359–370.

[24] Moravan, M. J., Bobba, J., Moore, K. E., Yen, L., Hill, M. D., Liblit, B., Swift, M. M., and Wood, D. A. Supporting nested transactional memory in logtm. *SIGOPS Oper. Syst. Rev. 40*, 5 (2006), 359–370.

[25] Rinard, M., and Lam, M. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems (TOPLAS) 20*, 3 (1998), 483–545.

[26] Rinard, M. C., and Diniz, P. C. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS) 19*, 6 (November 1997), 942–991.

[27] Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., and Hertzberg, B. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'06)* (New York, NY, USA, 2006), ACM, pp. 187–197.

[28] Schwarz, P. M., and Spector, A. Z. Synchronizing shared abstract types. *ACM Transactions on Computer Systems 2*, 3 (1984), 223–250.

[29] Spear, M. F., Marathe, V. J., Dalessandro, L., and Scott, M. L. Privatization techniques for software transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing (PODC'07)* (New York, NY, USA, 2007), ACM, pp. 338–339.

[30] Steele, Jr, G. L. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'90)* (New York, NY, USA, 1990), ACM Press, pp. 218–231.

[31] Weihl, W. E. Data-dependent concurrency control and recovery (extended abstract). In *Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC'83)* (New York, NY, USA, 1983), ACM Press, pp. 63–75.

[32] William N. Scherer, I., and Scott, M. L. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th annual ACM symposium on Principles of distributed computing (PODC'05)* (New York, NY, USA, 2005), ACM, pp. 240–248.