The Push/Pull Model of Transactions

Eric Koskinen *
IBM TJ Watson Research Center, USA

Matthew Parkinson
Microsoft Research Cambridge, UK

Abstract

We present a general theory of serializability, unifying a wide range of transactional algorithms, including some that are yet to come. To this end, we provide a compact semantics in which concurrent transactions PUSH their effects into the shared view (or UNPUSH to recall effects) and PULL the effects of potentially uncommitted concurrent transactions into their local view (or UNPULL to detangle). Each operation comes with simple criteria given in terms of commutativity (Lipton's left-movers and right-movers).

The benefit of this model is that most of the elaborate reasoning (coinduction, simulation, subtle invariants, etc.) necessary for proving the serializability of a transactional algorithm is already proved within the semantic model. Thus, proving serializability (or opacity) amounts simply to mapping the algorithm on to our rules, and showing that it satisfies the rules' criteria.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics

General Terms Languages, Theory

Keywords Push/Pull transactions, abstract data-types, transactional memory, transactional boosting, commutativity, movers

1. Introduction

Recent years have seen an explosion of research on methods of providing `atomic` sections in modern programming languages, typically implemented via transactional memory (TM). The `atomic` keyword provides programmers with a powerful concurrent programming building block: the ability to specify when a thread's operations on shared memory should appear to take place instantly when viewed by another thread.

To support such a construct, we must be able to reason about atomicity. Implementations typically achieve this by dynamically detecting conflicts between concurrent threads. This can be done by tracking memory operations in hardware [14–16] or software [4, 6, 8, 13, 27]. Meanwhile, an alternate approach exploits abstract-level notions of conflict over linearizable data-structure operations such as commutativity [11, 20, 21, 30]. Both levels of abstraction also chose between optimistic execution, pessimistic execution, or mixtures of the two. Finally, there are multiple notions of correctness, and circumstances under which one may be preferable to another.

Unfortunately, we lack a unified way of formally describing this myriad of models, implementations and correctness criteria. This leads to confusion when trying to understand comparative advantages/disadvantages and how/when models can be combined or are interoperable. For example, we may need to understand how one might want to combine *memory-level* hardware transactions [16, 37] for unstructured memory operations with *abstract-level* data-structure operations (*e.g.* transactional boosting [11] and open nesting [30]). Today, at best, we have two custom semantics for reasoning about the models individually, but no unified view.

We present a simple calculus that illuminates the core of transactional memory systems. In our model concurrent transactions PUSH their effects into the shared log (or UNPUSH to roll-back) and PULL in the effects of potentially uncommitted concurrent transactions (or UNPULL to detangle). Moreover, transactions can PUSH or PULL operations in non-chronological orders, provided certain commutativity (left/right-movers [25]) conditions hold. We have proved that this model is serializable, discussed in Section 5. To cope with the non-monotonic nature of the model (arising from UNPUSH, UNPULL, etc.), we devised a novel preservation invariant that is closed under rewinding both the local and global logs. The benefit of this semantic model is that most of the elaborate reasoning (coinduction, simulation relations, invariants, etc.) necessary for proving the correctness of a transactional algorithm is contained within the semantic model, and need only be proved once.

Our work formulates an expressive class of transactions and we have applied it to a wide range of TM systems including: optimistic read/write software TMs [6, 8], hardware transactional memories ([16], [15]), pessimistic TMs [4, 11, 27], hybrid opt./pess. TMs such as irrevocability [39], open nested transactions [30], and abstract-level techniques such as boosting [11].

Our choice of expressiveness includes transactions that are not opaque [10]: transactions *may* share their uncommitted effects. This choice carves out a design space for implementations to take advantage of the full spectrum of possibilities (*e.g.* dependent transactions [32], open nested transactions [30], liveness [3]) and is relatively unrestrictive in terms of TM correctness criteria. However, despite expressive power, the model also gives the appropriate criteria to ensure serializability [31]. Meanwhile, we can also identify restrictions on the model for which opacity is recovered.

* Koskinen was previously supported in part by NSF CCF Award #1421126.

PLDI'15 , June 13–17, 2015, Portland, OR, USA
Copyright © 2015 ACM 978-1-4503-3468-6/15/06. . . $15.00
DOI: http://dx.doi.org/10.1145/(to come)

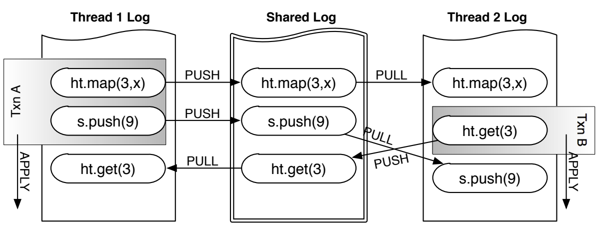
In our experience we have found that our model provides a mathematically rigorous foundation for intuitive concepts (*e.g.* PUSH and PULL) used in colloquial conversations contrasting TM systems.

Limitations. We have proved serializability by hand but we hope to verify our work with a proof assistant. Also, our work models safety properties of transactions (*i.e.* serializability, opacity) and a direction for future work is to consider liveness/progress issues.

2. Overview

In this paper we distill the essence of reasoning about transactional implementations into a semantic model we call Push/Pull transactions. The model consists of a few simple rules—named PUSH, PULL, etc.—that correspond to natural stages in a transactional memory algorithm. For example, after a transaction applies an effect locally it then may PUSH this effect out into the shared view, where other transactions may PULL the effect into their local view.

The Push/Pull model has no concrete state, only a shared log of the object operations that have been applied, as well as per-thread local logs. Here is an informal illustration:



Once a transaction applies an operation *op* to its local log via the APPLY rule, it may PUSH this *op* to the shared log. At this stage, the transaction may not have committed. Meanwhile, other threads may PULL the operation into their local log. The PULL case enables transactions to update their local view with operations that are permanent (that is, that correspond to committed transactions) or even to view the effects of another uncommitted transaction (*e.g.* for early conflict detection [13] or to establish a dependency [32]). Push/Pull also includes an UNPULL rule which discards a transaction's knowledge of an effect due to another thread, and an UNPUSH rule which removes a thread's operation from the shared view, perhaps implemented as an inverse. The UNAPPLY rule is useful for rewinding a transaction's local state. Finally, there is a simple commit rule CMT that, roughly, stipulates that all operations must have been PUSHed and all PULLed operations must have been committed.

Different algorithms will use different combinations of these rules (*cf.* Section 6). Push/Pull is expressive enough to describe a wide range of transactional implementations, all with only a few simple, tangible rules. Pessimistic algorithms [4, 11, 27] PUSH immediately after a local APPLY, optimistic algorithms [6, 8] PUSH their operations on commit, and hybrid [39] algorithms do a mixture of the two. Opaque [10] transactions do not PULL uncommitted effects. Non-opaque algorithms, such as dependent transactions [32], permit a transaction to PULL in uncommitted effects. From different patterns of Push/Pull rule usage one can derive correctness proofs for many transactional memory algorithms.

Example. Consider the transactional boosting [11, 12] hashtable implementation given in Figure 1. Recall that a boosted transaction uses a linearizable base object (here a `ConcurrentSkipListMap`), along with abstract locking to ensure that only commutative operations occur concurrently. In this example a thread executing the `atomic` block in `put` acquires a lock corresponding to the `key` of

```
1  class BoostedSkipListMap[Key,Val] {
2    val abstractLock = new AbstractLock()
3    val map = new ConcurrentSkipListMap[Key,Val]()
4    def put(key: Key, value: Val, t=Tx.current) {
5      atomic {
6        abstractLock lock key
7        if (map contains key) {
8          var oldValue = map(key)
9          Tx.onAbort( () ⇒
10           map.put(key, oldValue)
11           abstractLock unlock key
12         )
13       } else {
14         Tx.onAbort( () ⇒
15           map.remove(key)
16           abstractLock unlock key
17         )
18       }
19       map.put(key,value)
20       abstractLock unlock key
21     }
22   }
23 }
```

Figure 1. An implementation of *transactional boosting* [11] which uses abstract locking and commutativity to safely perform `put` operations on a shared map, implemented as a `ConcurrentSkipList`.

interest (Line 6). In this way, no two transactions will conflict because if they try to access the same `key` one will block. Within the `put` method there are two scenarios depending on whether `key` is already defined in the `map` (Line 7) and, consequently, there are two cases for how to handle an abort. Finally, `put` ends by updating `map` (Line 19) and unlocking the `abstractLock` (Line 20).

We can describe this algorithm intuitively with the Push/Pull model. A diagram depicting different reachable Push/Pull configurations is given in Figure 2. For simplicity, we focus on the shared log G and only the log L_i of transaction i. Each grey box represents a single operation; the unimportant ones have been left blank. The transitions between configurations are labeled with the Push/Pull transition rule and the corresponding line number of Figure 1 where that transition happens.

When the transaction begins it implements a PULL (Line 5) implicitly because, in transactional boosting, modifications are made directly to the shared state so the local view is the same as the shared view. It may, for example, PULL a `put(5)` operation from G that has already been committed (denoted as such with a $\checkmark$) and append the operation to its log L_i. Next, thread i may APPLY the `put(3)` operation, appending it to its local log, and then PUSH `put(3)` by appending it to the global log (both Line 19). Due to the pessimistic nature of boosting, APPLY and PUSH always happen together while in other, more optimistic transactional algorithms, an operation may be PUSHed at a later stage. If thread i commits (Line 21), it takes the CMT rule and marks the `put(3)` operation as committed with a $\checkmark$.

Threads may also move in a backward direction, undoing their effects (as in the `onAbort` handlers in Figure 1). If an abort is signaled, the transaction performs UNAPPLY and UNPUSH, both implemented by an inverse `map` operation (Lines 10 and 15, depending on whether the `key` was already in the `map`). In this simple diagram, the backward rules (UNPUSH and UNAPPLY) revisit earlier configurations but, with other thread interactions, backward rules may lead to new configurations. Transactions may use these rules in subtle ways, including PULLing uncommitted operations and PUSHing op-

Example Push/Pull Configurations and Transitions

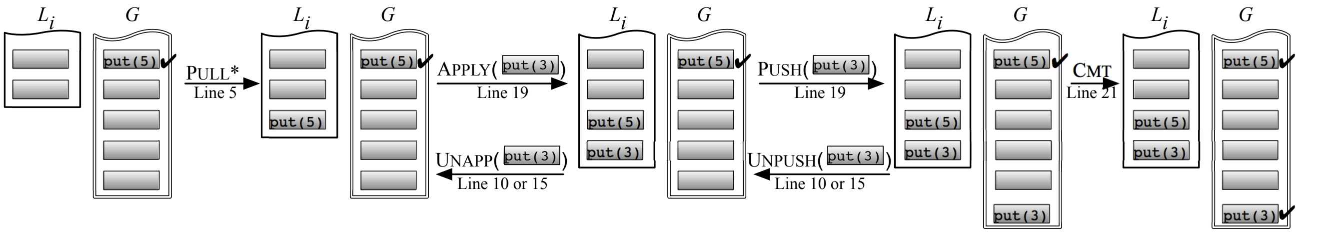


Figure 2. Using Push/Pull to model transactional boosting. The shared state is a log G and the state of a given thread i is represented by its local log L_i. Arrows indicate transitions in the Push/Pull model and they are annotated with the name of the Push/Pull rule.

erations in an order different from the local APPLY order. Another rule called UNPULL allows transactions to discard operations that have been PULLed from the shared log. This rule is useful to detangle a transaction that has viewed the effects of another transaction that may abort.

Correctness criteria. Despite the expressiveness of the rules, threads are not permitted to perform them whenever they please. Each rule is accompanied by certain correctness criteria, formalized in Section 4. For example, a criterion on the PUSH rule is that the operation being PUSHed *must be able to commute with (more precisely: move to the right of) all other threads' operations in the global log that have not yet been committed* (PUSH criterion (*ii*) in Figure 4). This particular criterion is the essence of boosting: that the commutative operations of multiple transactions can be executed concurrently, and boosting ensures this with abstract locking. As another example, a criterion for UNPUSH(op) is that everything PUSHed chronologically after op could still have been PUSHed if op hadn't been pushed (UNPUSH criterion (*ii*) in Figure 4). This holds trivially in this example because there are no operations other than the `put`.

Proofs of serializability. In Section 5 we prove that if an implementation satisfies all of the rules' correctness criteria, then it is serializable. In this sense we have done the hard work of reasoning about transactional memory algorithms. The full formal serializability argument involves showing a simulation relation between an interleaved machine and a sequential history. The Push/Pull model encapsulates the difficult components of this argument (*e.g.* simulation proofs, coinduction, etc.) while, on the outside, offering rules that are simple and intuitive.

Consequently, we believe that our work will clean up transactional correctness proofs. For a user to prove the correctness of their algorithm they must simply: (1) demarcate the algorithm into fragments: PUSH, PULL, etc. (2) prove the implementation satisfies the respective correctness criteria. Moreover, proofs of correctness criteria do not typically involve elaborate simulation relations or coinductive reasoning, but rather algebraic (*i.e.* commutative) properties of sequential code. In the above example, abstract locking ensures that no two `put` operations on the same key may happen concurrently. Therefore, all that must be done is to prove that the `put(x)` commutes with `put(y)` when `x` ≠ `y`. Proofs involving commutativity can be aided by recent works in the literature [7, 17].

3. Language and Atomic Semantics

In this section we describe a generic language of transactions and define an idealized semantics for concurrent transactions called the atomic semantics, in which there are no interleaved effects on the shared state. We later introduce the Push/Pull semantics and show that it simulates the atomic semantics. Due to lack of space, this paper provides mainly the intuition behind the Push/Pull model. The full details can be found in our technical report [19].

Language. We assume a set M of method calls and individual methods are written, for example, as `ht.put("a",5)`. Threads execute code c from some programming language that includes thread forking, local stack updates $\mathtt{local}_R$, transactions $\mathtt{tx}\ c$, method names such as m, and a `skip` statement. The local stack (sometimes referred to as local state) is over space Σ, is separate from the logs, and is used in order to model arguments and return values. Local stack updates $\mathtt{local}_R$ involve a relation $R \subseteq \Sigma \times \Sigma$. Our first trick is to abstract away the programming language with a few functions:

- $c \lightning_{\mathtt{tx}}(m, c')$: Within a transaction, code c can be reduced to the pair (m, c') where m is a next reachable method call in the reduction of c, with remaining code c'.
- $c \lightning (t, c')$: Outside of a transaction, code c can be reduced to the pair (t, c'). Here c' is the remaining code, and t is either a local stack update $\mathtt{local}_R$, a transaction $\mathtt{tx}\ c$, or a fork.
- $\mathsf{fin}(c)$: This predicate is true provided that there is a reduction of c to `skip` that does not encounter more work, e.g. a method call, a transaction, a fork, or a local operation.

These definitions allow us to obtain a simple semantics, despite an expressive input language, with functions to resolve nondeterminism between method operation names and at the end of a transaction. We assume that code is well-formed in that a single operation name m is always contained within a transaction (this issue of isolation [28] is orthogonal).

Example 1. *One could use the generic language:*

$$c ::= c_1 + c_2 \mid c_1 \,;\, c_2 \mid (c)^* \mid \mathtt{skip} \mid \mathtt{tx}\ c \mid m \mid \mathtt{local}_R$$

This grammar additionally consists of nondeterministic choice, sequential composition, and nondeterministic looping. We elide the definition of $c \lightning_{tx}(n, c')$ for lack of space, but it is straightforward. For example, if $c = \mathtt{tx}\,(\mathtt{skip};(c_1 + (m + n));c_2)$, then one path through c reaches method n with remaining code c_2. That is: $c \lightning_{tx}(n, c_2)$.

To make things more concrete, examples in this paper will typically use the above language. We do not permit syntactically nested transactions[1], however our model permits threads to roll backwards to any execution point [18] thus modeling the partial abort nature of nested transactions.

Operations and logs. State is represented in terms of logs of operation records. An operation record (or, simply, *operation*) $op = \langle m, \sigma_1, \sigma_2, id \rangle$ is a tuple consisting of the operation name m, a

[1] For a discussion, see [18].

thread-local pre-stack σ_1 (method arguments), a thread-local post-stack σ_2 (method return values), and a unique identifier id. An op is from space Ops. We assume a predicate $\mathsf{fresh}(id)$ that holds provided that id is globally unique (details omitted for lack of space). In the atomic semantics defined below, the shared state ℓ : $\mathsf{list}\ op$ is an ordered list of operations (more information is needed in the Push/Pull semantics, discussed later). We use notations such as $\ell_1 \cdot \ell_2$ and $\ell \cdot op$ to mean list append and appending a singleton, resp.

Parameter 3.1 (From logs to states: allowed)**.** *We require a prefix-closed predicate on operation lists* $\mathsf{allowed}\ \ell$ *that indicates whether an operation log* ℓ *corresponds to a state.*

For convenience we will also write ℓ allows $\langle m, \sigma_1, \sigma_2, id\rangle$ which simply means $\mathsf{allowed}\ \ell \cdot \langle m, \sigma_1, \sigma_2, id\rangle$. For example, if we have a simple TM based on memory read/write operations we expect $\mathsf{allowed}\ \ell \cdot \langle \mathtt{a\ :=\ x}, [x \mapsto 5], [x \mapsto 5, a \mapsto 5], id\rangle$, but $\neg\mathsf{allowed}\ \ell \cdot \langle \mathtt{a\ :=\ x}, [x \mapsto 5], [x \mapsto 5, a \mapsto 3], id\rangle$ or more elaborate specifications that involve multiple tasks. Ultimately, we expect the allowed predicate to be induced by the implementation's operations on the state and the initial state.

We define a precongruence over operation logs $\ell_1 \preccurlyeq \ell_2$ coinductively, by requiring that all allowed extensions of the log ℓ_1, are also allowed extension to the log ℓ_2. This definition will ultimately be used in the simulation between Push/Pull and an atomic machine. We use a coinductive definition so that the precongruence can be defined up to all infinite suffixes.

Definition 3.1 (Shared log precongruence $\preccurlyeq$)**.** *For all* ℓ_1, ℓ_2,

$$\frac{\mathsf{allowed}\ \ell_1 \Rightarrow \mathsf{allowed}\ \ell_2 \quad \forall op.\ (\ell_1 \cdot op) \preccurlyeq (\ell_2 \cdot op)}{\ell_1 \preccurlyeq \ell_2}\ gfp$$

Informally, the above greatest fixpoint says that there is no sequence of observations we can make of ℓ_2, that we can't also make of ℓ_1. This is more general than simply requiring that the set of states reached from executing the first log be included in the second. Unobservable state differences are also permitted.

Atomic semantics. We define a simple semantics, given in Figure 3, in which transactions are executed instantly, without interruption from concurrent threads. The semantics is a relation $\xrightarrow{a}$ over pairs consisting of a list of concurrent threads $\mathbf{A}$ and a shared log ℓ. A single thread $(c, \sigma) \in \mathbf{A}$ is a code c and local stack σ.

We begin with Figure 3(b). The atomic machine can take an AFIN step when there is a thread (c, σ) that can complete, *i.e.* $\mathsf{fin}(c)$. The AFORK rule allows a new thread executing code c_1 (also under local state σ) to be forked from thread (c, σ). The ALOCAL rule involves manipulating the thread-local state σ to σ' via relation R. Finally, the ATXN rule says that if thread executing code c_1 can reduce to a transaction $\mathtt{tx}\ c$ with remaining code c_2, then the transaction c is executed atomically by the big step rules $\Downarrow$ described next.

Figure 3(a) illustrates the big step semantics $\Downarrow$ which completely reduce a transaction. The rule BSFIN can be used if $\mathsf{fin}(c)$ holds: that c can be reduced to $\mathtt{skip}$, thus denoting the end of the transaction with a resulting log ℓ. Alternately, the rule BSFIN uses $\lightning_{\mathtt{tx}}$ to find a next operation m. This rule is taken provided that the operation $\langle m, \sigma, \sigma'\rangle$ is permitted and that (c_2, σ') can further be entirely reduced to σ'', ℓ_2. In this way, $\Downarrow$ appends the entire transaction's operations (unique IDs are unneeded in the atomic semantics) to the shared log.

4. The Push/Pull Model

In this section we describe the Push/Pull model. Concurrent threads execute the language described in the previous section but now

(a) **Atomic Machine Big Step Transaction Rules** $\Downarrow$

$$\frac{\mathsf{fin}(c)}{(c,\sigma), \ell \Downarrow c, \ell}\ \text{BSFIN} \qquad \frac{\begin{array}{c} c \lightning_{\mathtt{tx}} (m, c_2) \\ \ell_1\ \mathsf{allows}\ \langle m, \sigma, \sigma'\rangle \\ (c_2, \sigma'), \ell \cdot \langle m, \sigma, \sigma'\rangle \Downarrow \sigma'', \ell_2 \end{array}}{(c, \sigma), \ell_1 \Downarrow \sigma'', \ell_2}\ \text{BSSTEP}$$

(b) **Atomic Machine Rules** $\xrightarrow{a}$

$$\frac{\mathsf{fin}(c)}{\mathbf{A}_1 \cdot (c,\sigma) \cdot \mathbf{A}_2, G \xrightarrow{a} \mathbf{A}_1 \cdot \mathbf{A}_2, G}\ \text{AFIN}$$

$$\frac{c \lightning (\mathtt{fork}\ c_1, c_2)}{\mathbf{A}_1 \cdot (c,\sigma) \cdot \mathbf{A}_2, G \xrightarrow{a} \mathbf{A}_1 \cdot (c_1,\sigma) \cdot (c_2, \sigma) \cdot \mathbf{A}_2, G}\ \text{AFORK}$$

$$\frac{c \lightning (\mathtt{local}_R, c') \quad R\ \sigma\ \sigma'}{\mathbf{A}_1 \cdot (c,\sigma) \cdot \mathbf{A}_2, G \xrightarrow{a} \mathbf{A}_1 \cdot (c',\sigma') \cdot \mathbf{A}_2, G}\ \text{ALOCAL}$$

$$\frac{c_1 \lightning (\mathtt{tx}\ c, c_2) \quad (c,\sigma), G \Downarrow \sigma', G'}{\mathbf{A}_1 \cdot (c_1,\sigma) \cdot \mathbf{A}_2, G \xrightarrow{a} \mathbf{A}_1 \cdot (c_2,\sigma') \cdot \mathbf{A}_2, G'}\ \text{ATXN}$$

Figure 3. Atomic semantics of concurrent threads.

transaction interleavings are possible. Moreover, we describe rules APPLY, UNAPPLY, PUSH, UNPUSH, PULL, UNPULL, CMT which can be made by a given transaction to control how its effects are shared with the environment or view the effects made by the environment.

As in the atomic semantics, the Push/Pull semantics has a reflexive, transitive reduction $\mathbf{T}, G \rightsquigarrow \mathbf{T}', G'$ that reduces a list of threads $\mathbf{T}$: $\mathsf{list}\ (c \times \sigma \times L)$ and a global log G to $\mathbf{T}', G'$. Here, however, there is a per-thread local log L and the structure of L and the global log G is more complicated, as described below.

The reductions of the form $\mathbf{T}, G \rightsquigarrow \mathbf{T}', G'$ are given in Figure 4(a). Again, $\rightsquigarrow$ has rules for finishing a thread (FIN), forking (FORK) and local state manipulations (LOCAL), none of which alter the local log L.

The BEGIN rule can be used when it is possible for a thread to begin a transaction (c_1 can be reduced to $\mathtt{tx}\ c$). The STEP rule permits a single thread to take a step of one of the six Push/Pull rules (the $\underrightarrow{dir}$ relation described next). Finally, the CMT rule can be taken when a transaction can reduce its code to $\mathtt{skip}$. We will return to this rule at the end of this section.

The single-thread reduction relation $\underrightarrow{dir}$ has two directional types: $\underrightarrow{\mathsf{fwd}}$ and $\underrightarrow{\mathsf{bwd}}$. The three $\underrightarrow{\mathsf{fwd}}$ rules APPLY, PUSH, and PULL pertain to transactions making forward progress and the $\underrightarrow{\mathsf{bwd}}$ rules UNAPPLY, UNPUSH, and UNPULL pertain to transactions rewinding. Later we will use this directional distinction to set up invariants that are closed under rewind.

Figure 4(b) lists the six proof rules that form the core of Push/Pull. These rules pertain to a thread that is in the process of executing a transaction $\mathtt{tx}\ c$ and they manipulate the local stack, local log, and shared log in various ways. The local log L : $\mathsf{list}\ (Ops \times l)$ is a list of operations, each with an additional flag l, as to the status of the operation:

l	::=	$\mathsf{unpushed}\ c$	(local operation)
	|	$\mathsf{pushed}\ c$	(local operation shared to global view)
	|	pulled	(some other transaction's operation)

These flags keep track of the status of a given operation and from where it came. Additionally, the unpushed and pushed flags save the code c that was active when the log entry was created. There is also a global log G : $\mathsf{list}\ (Ops \times g)$ with flag g that distinguishes between operations that have or have not been committed: g ::=

(a) Push/Pull **Machine Rules** $\rightsquigarrow$

$$\frac{\mathsf{fin}(c)}{\mathbf{T}_1 \cdot \{c, \sigma, L\} \cdot \mathbf{T}_2, G \rightsquigarrow \mathbf{T}_1 \cdot \mathbf{T}_2, G}\ \textsc{fin} \qquad \frac{c \lightning (\mathtt{fork}\ c_1, c_2)}{\mathbf{T}_1 \cdot \{c, \sigma, L\} \cdot \mathbf{T}_2, G \rightsquigarrow \mathbf{T}_1 \cdot \{c_1, \sigma, L\} \cdot \{c_2, \sigma, L\} \cdot \mathbf{T}_2, G}\ \textsc{fork}$$

$$\frac{c_1 \lightning (\mathtt{local}_R, c_2) \quad R\,\sigma_1\,\sigma_2}{\mathbf{T}_1 \cdot \{c_1, \sigma_1, L\} \cdot \mathbf{T}_2, G \rightsquigarrow \mathbf{T}_1 \cdot \{c_2, \sigma_2, L\} \cdot \mathbf{T}_2, G}\ \textsc{local}$$

$$\frac{c_1 \lightning (\mathtt{tx}\, c, c_2)}{\mathbf{T}_1 \cdot \{c_1, \sigma, L\} \cdot \mathbf{T}_2, G \rightsquigarrow \mathbf{T}_1 \cdot \{(\mathtt{tx}\, c, c_2), \sigma, L\} \cdot \mathbf{T}_2, G}\ \textsc{begin} \qquad \frac{\mathsf{fin}(c) \quad L \subseteq G \quad \mathsf{cmt}(G, L, G')}{\mathbf{T}_1 \cdot \{(\mathtt{tx}\, c, c_1), \sigma, L\} \cdot \mathbf{T}_2, G \rightsquigarrow \mathbf{T}_1 \cdot \{c_1, \sigma, L\} \cdot \mathbf{T}_2, G'}\ \textsc{cmt}$$

$$\frac{\{c, \sigma, L\}, G\ \underrightarrow{dir}\ \{c', \sigma', L'\}, G'}{\mathbf{T}_1 \cdot \{(\mathtt{tx}\, c, c_1), \sigma, L\} \cdot \mathbf{T}_2, G \rightsquigarrow \mathbf{T}_1 \cdot \{(\mathtt{tx}\, c', c_1), \sigma', L'\} \cdot \mathbf{T}_2, G'}\ \textsc{step}$$

where $\mathsf{cmt}(G_1, L_1, G_2) \equiv$

$$G_2 \ = \ \mathsf{map}\ \left(\lambda\,(\mathit{op}, g).\ \begin{cases} (\mathit{op}, \mathsf{gCmt}) & \text{if } \mathit{op} \in \lfloor L_1 \rfloor_{\mathsf{pushed}} \\ (\mathit{op}, g) & \text{otherwise} \end{cases}\right) G_1$$

(b) Push/Pull **Step Rules** $\underrightarrow{dir}$

$$\frac{\begin{array}{ll} (i)- & c_1 \lightning_{\mathtt{tx}} (m, c_2) \\ (ii)- & L_1\ \mathsf{allows}\ \langle m, \sigma_1, \sigma_2 \rangle \\ (iii)- & \mathsf{fresh}(id) \end{array}}{\{\mathtt{tx}\ c_1, \sigma_1, L_1\}, G_1\ \underrightarrow{\mathsf{fwd}}\ \{\mathtt{tx}\ c_2, \sigma_2, L_1 \cdot [\langle m, \sigma_1, \sigma_2, id \rangle, \mathsf{unpushed}\ c_1]\}, G_1}\ \textsc{apply}$$

$$\frac{}{\{\mathtt{tx}\ c_1, \sigma_1, L_1 \cdot [\langle m, \sigma_2, \sigma_3, id \rangle, \mathsf{unpushed}\ c_2]\}, G_1\ \underrightarrow{\mathsf{bwd}}\ \{\mathtt{tx}\ c_2, \sigma_2, L_1\}, G_1}\ \textsc{unapply}$$

$$\frac{\begin{array}{ll} (i)- & \mathit{op} \blacktriangleleft \lfloor L_1 \rfloor_{\mathsf{unpushed}} \\ (ii)- & \lfloor G_1 \rfloor_{\mathsf{gUCmt}} \setminus \lfloor L_1 \cdot L_2 \rfloor_{\mathsf{pushed}} \blacktriangleleft \mathit{op} \\ (iii)- & G_1\ \mathsf{allows}\ \mathit{op} \end{array}}{\{\mathtt{tx}\ c_1, \sigma_1, L_1 \cdot [\mathit{op}, \mathsf{unpushed}\ c_2] \cdot L_2\}, G_1\ \underrightarrow{\mathsf{fwd}}\ \{\mathtt{tx}\ c_1, \sigma_1, L_1 \cdot [\mathit{op}, \mathsf{pushed}\ c_2] \cdot L_2\}, G_1 \cdot [\mathit{op}, \mathsf{gUCmt}]}\ \textsc{push}$$

$$\frac{\begin{array}{ll} (i)- & \mathsf{allowed}\ G_1 \cdot G_2 \\ (ii)- & \lfloor L_2 \rfloor_{\mathsf{pushed}} \blacktriangleleft \mathit{op} \end{array}}{\{\mathtt{tx}\ c_1, \sigma_1, L_1 \cdot [\mathit{op}, \mathsf{pushed}\ c_2] \cdot L_2\}, G_1 \cdot [\mathit{op}, g] \cdot G_2\ \underrightarrow{\mathsf{bwd}}\ \{\mathtt{tx}\ c_1, \sigma_1, L_1 \cdot [\mathit{op}, \mathsf{unpushed}\ c_2] \cdot L_2\}, G_1 \cdot G_2}\ \textsc{unpush}$$

$$\frac{\begin{array}{ll} (i)- & \mathit{op} \notin L \\ (ii)- & L\ \mathsf{allows}\ \mathit{op} \\ (iii)- & \mathit{op} \blacktriangleleft \lfloor L \rfloor_{\mathsf{pushed}} \cup \lfloor L \rfloor_{\mathsf{unpushed}} \end{array}}{\{\mathtt{tx}\ c_1, \sigma_1, L\}, G_1 \cdot [\mathit{op}, g] \cdot G_2\ \underrightarrow{\mathsf{fwd}}\ \{\mathtt{tx}\ c_1, \sigma_1, L \cdot [\mathit{op}, \mathsf{pulled}]\}, G_1 \cdot [\mathit{op}, g] \cdot G_2}\ \textsc{pull}$$

$$\frac{(i)- \quad \mathsf{allowed}\ L_1 \cdot L_2}{\{\mathtt{tx}\ c_1, \sigma_1, L_1 \cdot [\mathit{op}, \mathsf{pulled}] \cdot L_2\}, G\ \underrightarrow{\mathsf{bwd}}\ \{\mathtt{tx}\ c_1, \sigma_1, L_1 \cdot L_2\}, G}\ \textsc{unpull}$$

Figure 4. (a) The machine reductions of Push/Pull. (b) The Push/Pull rules. Notations $\setminus, \notin, \cdot, \subseteq$ are all lifted to lists where equality is given by ids. We will refer to the premise criteria of each rule as, for example, "PUSH criterion (*ii*)." Criteria that are written in gray font are not strictly necessary. See inline discussion.

gUCmt | gCmt. Each proof rule comes with criteria, labeled as APPLY criterion (*i*), APPLY criterion (*ii*), etc.

We will use the following liftings of set operations to lists:

$$\begin{array}{rcl} \langle m_1, \sigma_1, \sigma_1', id_1 \rangle \in L & \equiv & \exists i.\ L[i].\mathsf{id} = id_1 \\ G \setminus L & \equiv & \mathsf{filter}\ (\lambda\,(\mathit{op}, g).\ \mathit{op} \notin L)\ G \\ L \subseteq G & \equiv & \forall i.\ L[i].\mathsf{op} \in G \end{array}$$

where we use $L[i]$ to refer to the ith list element of L. We also use $L[i].\mathsf{op}$ to access ith operation tuple, ignoring the paired flag. The notation $L[i].\mathsf{id}$ further accesses the identifier of this ith operation. Notice that inclusion is based on equality over operation IDs. In $\setminus$ the order is determined by the first operand, G.

The* APPLY *rule. APPLY is similar to the BSSTEP rule in the atomic semantics: it can be used if there is a nondeterministic path in code c_1 that reaches a method m (with continuation code c_2). APPLY criterion (*ii*) specifies that method m must be allowed by the sequential specification with post-stack σ_2. If so, the new operation is appended to the local log L_1 with fresh operation id_1 (formalization of fresh in APPLY criterion (*iii*) is omitted). Intuitively, this rule applies some next method m to the local log but does not yet share it by sending it to the global log; it is marked as such with flag unpushed. The APPLY rule also records the pre-code c_1 in the local log so that the transaction can later be reversed (*i.e.* aborted or undone). Indeed, the rule UNAPPLY moves backwards by taking the last item in the local log and, provided that it is still unpushed, recalls the previous local stack and code.

The* PUSH *rule. A transaction may choose to share its effects with the global view via the PUSH rule. This reduction changes an operation's flag from unpushed to pushed in the local log and appends the operation to the global log, provided three conditions hold. These conditions use the notion of *left-mover* [25] which is an algebraic property of operations. We provide a novel coinductive definition of left-mover that builds upon log precongruence:

Definition 4.1 (Left-mover over logs)**.** *For all* $\mathit{op}_1, \mathit{op}_2$

$$\mathit{op}_2 \blacktriangleleft \mathit{op}_1 \quad \equiv \quad \forall \ell.\ \ell \cdot \{\mathit{op}_1, \mathit{op}_2\} \preccurlyeq \ell \cdot \{\mathit{op}_2, \mathit{op}_1\}.$$

Intuitively, operation op_2 can move to the left of operation op_1 provided that whenever we are allowed to do $op_1 \cdot op_2$, we are also allowed to do $op_2 \cdot op_1$ and the resulting log is the same (precongruent). The proof of serializability involves several fairly straightforward lemmas pertaining to allowed and left/right moverness, omitted for lack of space.

PUSH criterion (*i*) specifies that the pushed operation op *is able to move to the left of all unpushed operations in the local log.* This, intuitively, means that we can publish op as if it was the next thing to happen after all the operations published thus far by the current transaction. Here we have lifted $\blacktriangleleft$ to lists:

$$L_1 \blacktriangleleft L_2 \;\equiv\; \forall\, op_1 \in L_1,\, op_2 \in L_2.\, op_1 \blacktriangleleft op_2$$

and defined projections such as $\lfloor L_1 \rfloor_{\mathsf{unpushed}}$ using:

$$\lfloor L_1 \rfloor_l \;\equiv\; \mathsf{map\ fst\ (filter\ }(\lambda(op, l').\, l = l')\ L_1)$$

We have similarly defined $\lfloor G \rfloor_{\mathsf{gUCmt}}$.

> *Application:* Most existing implementations satisfy this trivially because operations are PUSHed in the same order that they are APPLYed.
> *Application:* In STMs that use redo-logs [9, 35] out-of-order PUSHing may occur. These implementations collect the write-set (*e.g.* {`x = 1; y = 2; z = 3; x = 4;`}) in a hashtable and, just before committing, may PUSH these writes to the shared log in the order they appear (*e.g.* {`(x,4),(y,2),(z,3)`}) in the hashtable. This can be viewed as out-of-order PUSHing where, furthermore, the PUSH of `(x,4)` is viewed as a PUSH of `(x,1)` immediately followed by a PUSH of `(x,4)`[2].

PUSH criterion (*ii*) is that *all uncommitted operations in the shared log $\lfloor G \rfloor_{gUCmt}$—except those due to the current transaction–can move to the right of the current operation op.* This condition ensures that if the transaction commits at any point, it can serialize before all concurrent uncommitted transactions. (Recall that we have lifted $\searrow$ to lists where equality is given by the operation IDs and the order is determined by the first operand, in this case G_1.)

> *Applications:* A boosted transaction immediately performs a PUSH at the linearization point because it modifies the shared state in place. Optimistic STMs don't perform PUSH until commit-time (unless there is some early conflict detection [13] which involves a form of PUSH. In boosted [11] and open nested [30] transactions, a commutativity requirement is sufficient to ensure this condition.

PUSH criterion (*iii*) is that op is allowed by the sequential specification of the global log. (Here we have lifted allowed to global logs.)

Note these APPLY, PUSH, PULL rules are about operations, which include *reads* as well as *writes*. For example, when a transaction PUSHes a read it is effectively announcing to the shared log the fact that it is accessing the particular memory location. This fact is crucial to how we can show that snapshot isolation violates serializability.

The* UNPUSH *rule. An operation op that has been PUSHed to the shared log can be UNPUSHed. This amounts to swapping the local flag from pushed to unpushed and removing the corresponding global log entry for op. PUSH criterion (*i*) ensures that G_2 does not depend on op and PUSH criterion (*ii*) is that everything pushed chronologically after op could still have been pushed if op hadn't been pushed. Note that PUSH criterion (*i*) is not strictly necessary because we can prove that it must hold whenever an UNPUSH occurs.

> *Application:* When a boosted transaction aborts (*e.g.* due to deadlock) it must undo its effects on the shared state. This is modeled via the UNPUSH rule and typically implemented via inverse operations (such as `remove` on an element that had been added). The UNPUSH rule is also needed in open nested transactions [30] and for ensuring liveness [3].

The* PULL *rule. Transactions can learn about the published effects of other transactions by PULLing operations from the global log into their local logs. An operation op can be pulled from the global log provided that is wasn't pulled before (PULL criterion (*i*)) and that the local log allows it (PULL criterion (*ii*)) according to the sequential specification. A transaction can only learn about the shared state through PULLing. In most applications, a transaction will PULL operations in chronological order. However, there are many examples for which this is not true. In a transaction that operates over two shared data-structures a and b, it may PULL in the effects on a even if they occurred after the effects on b because the transaction is only interested in modifying a. When the PULL rule occurs, the operation is appended to the local log L and marked as pulled.

Finally, PULL criterion (*iii*) is that *everything that the current transaction has currently done locally must be able to move to the right of op.* This ensures that the transaction can behave as if the pulled effect preceded the transaction. We have marked this criterion in gray, indicating that it is not strictly necessary. One could imagine allowing transactions to PULL uncommitted, conflicting effects. However, we don't believe such behaviors to be particularly interesting or realistic.

> *Application*: Many traditional STMs are opaque [10] (transactions cannot view the effects of other uncommitted transactions). Such systems never execute PULL operations marked as gUCmt and can only view operations that have been marked gCmt.
> *Application*: Some (non-opaque) transaction A may become *dependent* [32] on another transaction B if the effects of B are released to A before B commits. This is captured by B performing a PUSH of some effects that are then PULLed by A even though B has not committed.
> *Application*: So-called *consistency* in validation and timestamp extension [33].

The* UNPULL *rule. A PULLed operation may be removed from the local log. UNPULL criterion (*i*) is that the local log is allowed without operation op. Informally, this means that the transaction must not have done anything that depended on op. Without this criterion the local log might become invalid with respect to the sequential specification.

> *Applications*: Breaking dependencies [32], open nested transactions [30], liveness [3].

The* CMT *rule. If there is a path through `tx` c that reaches `skip` (CMT criterion (*i*)), then the transaction can commit. There are three additional conditions: CMT criterion (*ii*) is that the local log L_1 must be contained within the global log G_1, indicating that *all of the transaction's operations have been pushed*. CMT criterion (*iii*) says that *all pulled operations correspond to transactions that have been committed*. Finally, CMT criterion (*iv*) is that the global log is updated to G_2 in which all of the transaction's operations are marked as committed. This is achieved with the $\mathsf{cmt}(G_1, L_1, G_2)$ predicate, defined at the bottom of Figure 4. The CMT rule serves as the instantaneous moment when all of a transaction's effects become permanent. Note that a transaction does not have to PULL all committed operations. Instead, transactions check whether they conflict with other transactions' operations each time they PUSH an operation.

[2] We thank an anonymous reviewer for this example.

We strived to design the Push/Pull model such that it encompasses all serializable systems that we know of. To date, we have not found any (serializable) implementations that cannot be described by this model. Systems such as distributed transactions, P2P communications and concurrent revisions [2] are non-serializable and certainly fall outside of the Push/Pull model.

5. Serializability

We have proved serializability of the Push/Pull machine, via a simulation between a Push/Pull machine and the atomic semantics. For lack of space, we merely describe the structure of the proof. The full proof can be found in the companion technical report [19].

Preservation invariant. The heart of the simulation requires that we prove an invariant of the system that the shared log is equivalent to what it would be if concurrently executing transactions removed their PUSHed effects and instead PUSHed them atomically. More precisely, imagine that at a given moment there is a shared log G, and a given thread $T = \{c, \sigma, L\}$ atomically marks all of its pushed operations as committed, reaching a shared log of G_{post}. Note that T may still have unpushed operations $\lfloor L \rfloor_{\mathsf{unpushed}}$. The invariant states that there is a precongruence between the shared log reached by completing T from $G_{\mathsf{post}} \cdot \lfloor L \rfloor_{\mathsf{unpushed}}$ and the shared log that would have been reached if T rewound itself and atomically ran the entire transaction from G (that is, $G \setminus L$, *i.e.* the previous shared log, with all operations belonging to T filtered out). As described so far, the commit preservation invariant (which holds for each $T = \{c, \sigma, L\}$ and G) would look like the following:

$$
\begin{array}{l}
\forall G_{\mathsf{post}}.\ \mathsf{cmt}(G, L, G_{\mathsf{post}}) \ \Rightarrow \\
\forall \sigma', \ell_a.\ (c, \sigma), G_{\mathsf{post}} \cdot \lfloor L \rfloor_{\mathsf{unpushed}} \Downarrow \sigma', \ell_a \ \Rightarrow \\
\ \exists \ell_b.\ \ \mathsf{otx}(\{c, \sigma, L\}), G \setminus L \Downarrow \sigma', \ell_b \ \ \wedge \ \ \ell_a \preccurlyeq \ell_b
\end{array}
$$

where otx rewinds the transaction to its original state/code, recorded in L.

Partial rewind. This is not enough to give us the simulation result as the property is not an invariant. As the system makes steps that undo operations from the logs, the property must be closed with respect to these *backwards* steps. Thus we need the above to hold after any partial rewinding of the local log and/or partial removal of other transactions' uncommitted operations in the shared log. We define a *self-rewind* relation denoted

$$\{c, \sigma, L\}, G \ \circlearrowleft_{\mathsf{self}} \ \{\text{‘}c, \text{‘}\sigma, \text{‘}L\}, \text{‘}G$$

which allows us to cope with the fact that a transaction may have PULLed operations from another uncommitted transaction. In particular, we preserve the fact that the transaction *may* be able to detangle from the uncommitted transaction, and atomically commit. We also define a *shared log partial rewind* denoted $G \ \circlearrowleft_L \ \text{‘}G$ which permits uncommitted operations of other transactions (*i.e.* excluding operations in L) to be dropped from the shared log.

The preservation invariant as follows:

Definition 5.1 (Commit preservation invariant)**.** *For all G,*

$$
\begin{array}{lr}
\forall \text{“}G.\ G \ \circlearrowleft_L \ \text{“}G.\ \Rightarrow & (0) \\
\forall \{\text{‘}c, \text{‘}\sigma, \text{‘}L\}.\ \{c, \sigma, L\}, \text{“}G \ \circlearrowleft_{\mathit{self}} \ \{\text{‘}c, \text{‘}\sigma, \text{‘}L\}, \text{‘}G \ \Rightarrow & (1) \\
\forall G_{\mathit{post}}.\ \mathit{cmt}(\text{‘}G, \text{‘}L, G_{\mathit{post}}) \ \Rightarrow & (2) \\
\forall \sigma', \ell_a.\ (\text{‘}c, \text{‘}\sigma), G_{\mathit{post}} \cdot \lfloor \text{‘}L \rfloor_{\mathit{unpushed}} \Downarrow \sigma', \ell_a \ \Rightarrow & (3) \\
\ \exists \ell_b.\ \ \mathit{otx}(\{\text{‘}c, \text{‘}\sigma, \text{‘}L\}), G \setminus L \Downarrow \sigma', \ell_b \ \ \wedge \ \ \ell_a \preccurlyeq \ell_b & (4)
\end{array}
$$

Intuitively, this invariant means that under any dropping of others' uncommitted operations (Line 0) and after partially rewinding $\circlearrowleft_{\mathsf{self}}$ the local transaction to some local log $\text{‘}L$ (Line 1), if the transaction is now able to atomically "commit" by swapping commit flags (Line 2) and running the rest of the transaction (Line 3), then the shared log reached ℓ_a, is contained within a shared log ℓ_b that would have been reached if the thread appended its entire transaction to G atomically.

Theorem 5.1 (Serializability)**.** *Push/Pull is serializable.*

Proof. The full proof can be found in our technical report [19]. Here we instead sketch the proof, which is via a simulation relation between $\rightsquigarrow$ and $\xrightarrow{a}$. The simulation relation is defined as follows:

$$\mathbf{T}, G \sim \mathbf{A}, \ell \quad \equiv \quad (\mathsf{map\ rewind}\ \mathbf{T}) = \mathbf{A} \ \wedge \ \lfloor G \rfloor_{\mathsf{gCmt}} \preccurlyeq \ell$$

where $\mathsf{rewind}\ T$ rolls a transaction back to its original code. We define $\mathbf{T} \sim \mathbf{A}$ and $G \sim \ell$ with the appropriate conjunct from above.

We must consider each $\rightsquigarrow$ step from Figure 4 and show that an appropriate $\mathbf{A}', \ell'$ can be found. In each case, the inductive hypothesis gives us that the simulation relation rewinds all uncommitted transactions in $\mathbf{T}$ to obtain $\mathbf{A}$ and drops all uncommitted operations from G to obtain ℓ. Moreover, we rely on several invariants holding for $\mathbf{T}, G$ as well as $\mathbf{T}', G'$ (most significantly, the commit preservation invariant).

Most cases are trivial, because we map $\mathbf{T}', G'$ to the previous $\mathbf{A}, \ell$. The exception is the CMT case, where we use the preservation invariant to show that after a CMT, we can find a new atomic machine configuration that maintains the simulation relation. □

6. Implementations

We now discuss how our model can be used to reason about a wide variety of implementations in the literature. In each case, we recast the implementation strategy in terms of the Push/Pull model and discuss how the implementations satisfy the conditions of each rule in Push/Pull.

Opacity. For general Push/Pull transactions, opacity [10] does not necessarily hold: transactions may view the uncommitted effects of other concurrent transactions. However, there are several ways that we can characterize opacity as a fragment of Push/Pull transactions. For example, if transactions do not perform PULL operations during execution then they are *opaque*.

We can take things a step further. An active transaction T may PULL an operation op' that is due to an uncommitted transaction T' provided that T will never execute an op that does not commute with op. This suggests an interesting way of ensuring opacity while PULLing uncommitted effects by examining (statically or dynamically) the set of all reachable operations a transaction may perform.

Optimistic Models. STMs such as TL2 [6], TinySTM [8], McRT STM [34] are optimistic (or mostly-optimistic) and do not share their effects until they commit. Transactions begin by PULLing all operations (there are never uncommitted operations) by simply viewing the shared state. As they continue to execute, they APPLY locally and do not PUSH until an uninterleaved moment when they check the second PUSH condition on all of their effects (which is approximated via read/write sets) and, if it holds, PUSH everything and CMT. Effects are pushed in order so the first PUSH condition is trivial. If a transaction discovers a conflict, it can simply perform UNAPPLY repeatedly and needn't UNPUSH.

Transactions that use checkpoints [18] and (closed) nested transactions [29] do not share their effects until commit time. They are similar to the above optimistic models, except that place-markers are set so that, if an abort is detected, UNAPPLY only needs to be performed for some operations.

Pessimistic Models. Matveev and Shavit [27] describe how pessimistic transactions can be implemented by delaying write operations until the commit phase. In this way, write transactions appear to occur instantaneously at the commit point: all write operations are PUSHed just before CMT, with no interleaved transactions.

Consequently, read operations perform PULL only on committed effects. Boosting [11] is also a pessimistic model, as discussed in Section 2.

Mixed Models. For the irrevocable transactions of [39], there is at most one pessimistic ("irrevocable") transaction and many optimistic transactions. The pessimistic transaction PUSHes its effects instantaneously after APPLY.

Reading Uncommitted Effects. As discussed in Section 4, the early release mechanism [13] and dependent transactions [32] can be modeled with Push/Pull. In early release, an executing transaction T communicates with T' to determine whether the transactions conflict. This is modeled as T' performing a PUSH(op) and T checking whether it is able to PULL(op). A *dependent transaction* T will PULL the effects of another transaction T'. This comes with the stipulation that T does not commit until T' has committed. If T' aborts, then T must abort. However, note that T must only move backwards (via $\underrightarrow{\text{bwd}}$) insofar as to detangle from T'.

7. Implementations That Are Yet to Come

The Push/Pull model is expressive, permitting transactions to announce their effects in orders different from the way they are done locally (see the PUSH rule). Moreover, transactions can undo their effects in different orders from the order they were announced in (see the UNPUSH rule).

The utility of this expressiveness can be demonstrated by a more elaborate yet-to-be-implemented setting: combining hardware transactions with boosting [11]. Transactions in hardware are optimistic in nature, while boosting is pessimistic. Consequently, the order of APPLY, PUSH and PULL and the undo operations UNAPPLY, UNPUSH and UNPULL must be very flexible. For boosting, one needs to PULL all operations on the data-structure, APPLY the current operation, and then PUSH the operation immediately. For a hardware transaction, we initially PULL operations for the current snapshot of the memory location in question, and merely APPLY it locally, saving the PUSHing until the final commit. Thus we need to leverage the semantics' support for PUSHing in arbitrary orders. Adding the possibility of transactions failing at various points, leads to the need for flexibility in the order of undo operations as well.

Consider the following example transaction that accesses a boosted version of a `ConcurrentSkipList` and a boosted version of a `ConcurrentHashTable`, as well as integer variables `size`, `x`, and `y` that are controlled via a hardware transactional memory [16]:

```
1  BoostedConcurrentSkipList skiplist;
2  BoostedConcurrentHashTable hashT;
3  HTM int size;
4  HTM int x, y;
5
6  atomic {
7    skiplist.insert(foo);
8    size++;
9
10   hashT.map(foo => bar);
11   if (*)
12     x ++;
13   else
14     y ++;
15 }
```

Let us say that execution proceeds, modifying the `skiplist`, incrementing `size`, updating the `hashT`, and the following the `if` branch. At this underlined point when `x` is about to be incremented, let us say that the hardware transactional memory detects a conflict with a concurrent access to `x`.

The Push/Pull model shows that the implementation can rewind (UNPUSH) the effects of the HTM, but leave the effects of the boosted objects (which are expensive to replay) in the shared view. So the HTM can discard the effects to `x` and `size` with UNPUSHP, perform a partial rewind via UNAPPLY, then execute Lines 11–15.

In terms of the Push/Pull model, the transaction has performed the rules given in Figure 5. This figure decomposes the elaborate behavior into the simple Push/Pull rules. We can then construct a correctness argument for the example from the criteria of each rule, and the hard work of the simulation proof is done for us.

Transaction begins.	PULL(snapshot of HT vars `x`,`y`,`size`)
	PULL(all `skiplist` operations)
	APPLY(`skiplist.insert(foo)`),
	PUSH(`skiplist.insert(foo)`),
	APPLY(`size++`),
	PULL(all `hashT` operations)
	APPLY(`hashT.map(foo=>bar)`),
	PUSH(`hashT.map(foo=>bar)`),
	APPLY(`x++`),
Push HTM ops:	PUSH(`size++`),
	PUSH(`x++`),
HTM signals abort.	UNPUSH(`x++`),
	UNPUSH(`size++`),
Rewind some code:	UNAPPLY(`x++`),
March forward again:	APPLY(`y++`),
Uninterleaved commit:	PUSH(`size++`),
	PUSH(`y++`),
	CMT

Figure 5. Decomposing behavior in terms of Push/Pull rules.

8. Related Work

In our prior work we provided a formal semantics for abstract-level data-structure transactions [20]. This prior semantics separated pessimistic models from optimistic ones. The model presented in this paper is more expressive because it permits mixtures of these two flavors. This is useful when combining hardware [15, 16] with abstract-level data-structure [11] transactions. Moreover, Push/Pull transactions may observe the effects of uncommitted, non-commutative transactions as seen in dependent transactions [32] and open nesting [30].

Others [22] describe a method of specifying and verifying TM algorithms. They specify some transactional algorithms in terms of I/O automata [26] and this choice of language enables them to fully verify those specifications in PVS. In our work, we have aimed at a more abstract goal: to uncover the fundamental nature of transactions in the form of a general-purpose model. We leave the goal of full algorithm verification (and automated tools) to future work.

There are other works in the literature that are focused on a variety of orthogonal semantic issues, including the privatization problem [1, 28, 36], correctness criteria such as dynamic/static/hybrid atomicity [38], and message passing within transactions [23]. These works are concerned with models that are restricted to read/write STMs and limited in expressive power (e.g. restricted to opacity [10]). Others have looked at ways to decompose proofs of opacity [24]. Semantics also exist for other programming models that are similar to transactions [2] but are not serializable. Finally,

[5] described some small hand proofs for particular transactional memory algorithms.

9. Conclusions and Future Work

We have described an expressive model of transactions and shown that it is capable of serving as proof of serializability for a wide variety of transactional memory algorithms. We work with pure logs and develop a model in which transactions pass around their effects by PUSHing to or PULLing from a shared log. The model gives rise to simple proof rules that allow us to more easily construct proofs for a wide range of transactional behaviors—optimism, pessimism, opacity, dependency, etc.—all within a unified treatment.

As a next step we plan to formalize our work in a proof assistant. Another important avenue of future work is to develop models of existing implementations and show that they are serializable using the Push/Pull model.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback and the NSF for supporting Koskinen (CCF Award #1421126).

References

[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), pp. 63–74.

[2] BURCKHARDT, S., AND LEIJEN, D. Semantics of concurrent revisions. In *Proceedings of the 20th European Symposium on Programming (ESOP'11)* (2011), G. Barthe, Ed., vol. 6602, pp. 116–135.

[3] BUSHKOV, V., GUERRAOUI, R., AND KAPALKA, M. On the liveness of transactional memory. In *Proceedings of the Symposium on Principles of Distributed Computing (DISC'12)* (2012), pp. 9–18.

[4] CHEREM, S., CHILIMBI, T. M., AND GULWANI, S. Inferring locks for atomic sections. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI'08)* (2008), pp. 304–315.

[5] COHEN, A., PNUELI, A., AND ZUCK, L. D. Mechanical verification of transactional memories with non-transactional memory accesses. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV'08)* (2008), vol. 5123, pp. 121–134.

[6] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)* (September 2006).

[7] DIMITROV, D., RAYCHEV, V., VECHEV, M., AND KOSKINEN, E. Commutativity race detection. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14), Edinburgh, UK* (2014).

[8] FELBER, P., FETZER, C., AND RIEGEL, T. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)* (2008), pp. 237–246.

[9] FELBER, P., KORLAND, G., AND SHAVIT, N. Deuce: Noninvasive concurrency with a Java STM. In *Electronic Proceedings of the workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)* (2010).

[10] 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)* (2008), ACM, pp. 175–184.

[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., AND KOSKINEN, E. Composable transactional objects: A position paper. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems (ESOP'14)* (2014), vol. 8410, pp. 1–7.

[13] 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), pp. 92–101.

[14] 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), pp. 289–300.

[15] IBM. Alphaworks Software Transactional Memory Compiler. `http://ibm.biz/alphaworks-stm-compiler`.

[16] INTEL. Transactional synchronization in haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`.

[17] KIM, D., AND RINARD, M. C. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)* (2011), pp. 528–541.

[18] KOSKINEN, E., AND HERLIHY, M. Checkpoints and continuations instead of nested transactions. In *Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'08)* (2008), pp. 160–168.

[19] KOSKINEN, E., AND PARKINSON, M. The push/pull model of transactions [extended version]. Tech. Rep. RC25529, IBM Research, 2015.

[20] KOSKINEN, E., PARKINSON, M. J., AND HERLIHY, M. Coarse-grained transactions. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)* (2010), ACM, pp. 19–30.

[21] 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), pp. 211–222.

[22] LESANI, M., LUCHANGCO, V., AND MOIR, M. A framework for formally verifying software transactional memory algorithms. In *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR'12)* (2012), vol. 7454, pp. 516–530.

[23] LESANI, M., AND PALSBERG, J. Communicating memory transactions. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'11)* (2011), pp. 157–168.

[24] LESANI, M., AND PALSBERG, J. Decomposing opacity. In *Proceedings of the 28th International Symposium on Distributed Computing (DISC'14)* (2014), pp. 391–405.

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

[26] LYNCH, N. A., AND TUTTLE, M. R. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)* (1987), pp. 137–151.

[27] MATVEEV, A., AND SHAVIT, N. Towards a fully pessimistic STM model. In *Proceedings of the 2012 Workshop on Transactional Memory (TRANSACT12)* (2012).

[28] 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)* (2008), pp. 51–62.

[29] 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 Operating Systems Review 40*, 5 (2006), 359–370.

[30] NI, Y., MENON, V. S., ADL-TABATABAI, A.-R., HOSKING, A. L., HUDSON, R. L., MOSS, J. E. B., SAHA, B., AND SHPEISMAN, T. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'07)* (2007), pp. 68–78.

[31] PAPADIMITRIOU, C. The serializability of concurrent database updates. *Journal of the ACM (JACM) 26*, 4 (1979), 631–653.

[32] RAMADAN, H. E., ROY, I., HERLIHY, M., AND WITCHEL, E. Committing conflicting transactions in an stm. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)* (2009), pp. 163–172.

[33] RIEGEL, T., FETZER, C., AND FELBER, P. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'07)* (2007), pp. 221–228.

[34] 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 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)* (2006), pp. 187–197.

[35] SPEAR, M., DALESSANDRO, L., MARATHE, V., AND SCOTT, M. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP'09)* (2009).

[36] SPEAR, M. F., MARATHE, V. J., DALESSANDRO, L., AND SCOTT, M. L. Privatization techniques for software transactional memory. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC'07)* (2007), pp. 338–339.

[37] WANG, A., GAUDET, M., WU, P., AMARAL, J. N., OHMACHT, M., BARTON, C., SILVERA, R., AND MICHAEL, M. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (2012), pp. 127–136.

[38] WEIHL, W. E. Data-dependent concurrency control and recovery (extended abstract). In *Proceedings of the 2nd annual ACM symposium on Principles of Distributed Computing (PODC'83)* (1983), ACM Press, pp. 63–75.

[39] WELC, A., SAHA, B., AND ADL-TABATABAI, A.-R. Irrevocable transactions and their applications. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA'08)* (2008), ACM, pp. 285–296.