Transactions in Relaxed Memory Architectures

BRIJESH DONGOL, Brunel University London, UK
RADHA JAGADEESAN, DePaul University, USA
JAMES RIELY, DePaul University, USA

The integration of transactions into hardware relaxed memory architectures is a topic of current research both in industry and academia. In this paper, we provide a general architectural framework for the introduction of transactions into models of relaxed memory in hardware, including the SC, TSO, ARMV8 and PPC models. Our framework incorporates flexible and expressive forms of transaction aborts and execution that have hitherto been in the realm of software transactional memory. In contrast to software transactional memory, we account for the characteristics of relaxed memory as a restricted form of distributed system, without a notion of global time. We prove abstraction theorems to demonstrate that the programmer API matches the intuitions and expectations about transactions.

CCS Concepts: • **Theory of computation → Parallel computing models**; **Abstraction**;

Additional Key Words and Phrases: Relaxed Memory Models, Hardware Transactional Memory

ACM Reference Format:
Brijesh Dongol, Radha Jagadeesan, and James Riely. 2018. Transactions in Relaxed Memory Architectures. *Proc. ACM Program. Lang.* 2, POPL, Article 18 (January 2018), 29 pages. https://doi.org/10.1145/3158106

1 INTRODUCTION

Locks are ubiquitous in programming because they provide mutual exclusion and ordering properties to concurrent threads. However, the use of locks is subtle as exemplified by the common problems of *priority-inversion* (a lower-priority process holding a lock needed by a higher-priority process cannot release the lock if its execution is preempted) and *deadlocks* (cyclic dependencies on locks caused by processes contending dynamically for a collection of locks).

Such issues have motivated a transactional approach to thread synchronization, including hardware transactional memory (HTM) [Herlihy and Moss 1993] and software transactional memory (STM) [Shavit and Touitou 1995]. Larus and Kozyrakis [2008], Harris et al. [2010], Guerraoui and Kapalka [2010] and Grossman et al. [2007] provide a broad introduction to the subject. HTM exploits the power of extant cache-coherence protocols, which are already tuned for performance in a concurrent context. This permits their performance to match hand-crafted software that uses fine-grained locks and CAS instructions without losing the simplicity of coarse-grained locks [Yoo et al. 2013]. On the other hand, the "bounded" and "best-effort" hardware transactional model is limited by the capacity constraints of caches in terms of the number of locations that can be included in a hardware transaction (e.g., see Nguyen [2015] for a recent experimental evaluation), so a transaction can fail when its working set exceeds the capacity of the hardware. STM achieves

Authors' addresses: Brijesh Dongol, Department of Computer Science, Brunel University London, UK, brijesh.dongol@brunel.ac.uk; Radha Jagadeesan, School of Computing, DePaul University, USA, rjagadeesan@cs.depaul.edu; James Riely, School of Computing, DePaul University, USA, jriely@cs.depaul.edu.

2475-1421/2018/1-ART18
https://doi.org/10.1145/3158106

transactional guarantees in software without any such limitations. Some STM systems also support flexible and expressive ways of composing transactions [Haines et al. 1994; Harris et al. 2005]. Despite the impressive advances in the implementation of STMs, researchers have also considered variations of hybrid transactional memory systems [Damron et al. 2006; Harris et al. 2010] that seek to preserve the progress properties of STMs while building on the performance of HTM implementations.

Inspired by database theory [Eswaran et al. 1976], the two key elements of correctness of committed transactions enunciated by Herlihy and Moss [1993] are:

- Atomicity: transactions appear to execute sequentially, i.e., without interleaving.
- (Strict) serializability of committed transactions: There is a sequential order among committed transactions that is consistent with their real-time order.

These conditions are intimately related to the notion of *linearizability* [Herlihy and Wing 1990] because they ensure that every concurrent history (an interleaved sequence of invocation and response events) can be mapped to a legal sequential history (where each invocation is immediately followed by its matching response) such that the real-time order induced by the concurrent history over transactions is preserved (see [Armstrong et al. 2017]).

The focus of this paper is the study of transactions under relaxed memory models.

Sequential consistency [Lamport 1979] enforces a total order on memory operations — reads and writes to the memory — respecting the program order of each individual thread in the program. Modern multicore architectures are relaxed and permit executions that are not sequentially consistent. Adve and Gharachorloo [1996] and Adve and Boehm [2010] provide a tutorial introduction with detailed bibliography on architectures and their impact on language design. This motivates models of *relaxed memory* in hardware, such as TSO [Sewell et al. 2010], Power [Sarkar et al. 2011], and runtime systems, such as Java [Manson et al. 2005; Sevcík 2008] and C++ [Batty et al. 2011; Boehm and Adve 2008]. Alglave et al. [2014] provide a systematic and general framework that illustrates the key ideas.

Our aim is to provide a general framework to integrate transactions into a hardware memory model. The signature feature of relaxed memory models is the discarding of the global real-time clock — these are distributed systems after all, albeit of a somewhat restricted kind. Thus, the key issue that we have to address is the "semantics of transactions with respect to a memory model weaker than sequential consistency" [Grossman et al. 2006]. In the hardware memory models of interest, the global real-time order is replaced by ordering relations that determine when actions in a thread can be seen by other threads. Following the vocabulary of Alglave et al. [2014], consider two of the observable orders of interest:

- *coherence order*, which relates the writes to the same location into a total order, and
- *causal order*, which relates an action to those that causally precede it, as determined by program order and synchronization.

The causal order always includes *dependencies* from a write of a variable to a read by another thread that sees that write; this is referred to as a *(external) reads-from order*. The coherence order induces *anti-dependencies* from a read to a write that overwrites the value seen by the read; this is often referred to as *from-read order*. There are other architecture-specific components that can influence the observable orders; we return to these notions in Section 2.

For the rest of this section, we focus on developing an observation-based notion of serializability. We first discuss committed transactions in isolation, then in combination with non-transactional memory operations and aborted transactions.

1.1 Observation-Based Serialization for Committed Transactions

Our version of the correctness of committed transactions replaces strict serializability with *observable serializability*. Whereas strict serializability requires that there be a committed transaction order that preserve real-time order, observable serializability requires only that the committed transaction order preserve orders that can be observed, such as coherence and causal ordering.

Observable serializability permits more executions than strict serializability. For example, consider the following execution, where we assume time flows from left to right. We use Rxv to denote read events and Wxv to denote and write events, where x is a location and v is a value. We use different kinds of braces/parenthesis to indicate the begin and end of transactions. In implementations, the begin and end actions have durations, but we ignore this duration for simplicity since they do not matter to the examples in this introduction. Also for simplicity, we use examples in which each thread executes at most one transaction — we refer to the transactions as T1, T2, T3, etc., where the numbers correspond to the thread identifier. As is common in the relaxed memory literature, in all our examples we assume that all variables are initialized to 0.

```
Thread 1: (Rx2                   )
Thread 2:       [ Wx2  Ry1                     ]
Thread 3:                               <      Wy1 >
```

In the above execution, due to the causal reads-from dependencies, T3 must precede T2, and T2 must precede T1 in any serialization of the transactions. If the real-time order had to be respected, the execution above would be forbidden; this is the case for criteria such as final-state opacity [Guerraoui and Kapalka 2010] and TMS [Doherty et al. 2013]. However, since the order between T1 and T3 is not observable in a relaxed memory architecture, the serialization T3 T2 T1 is acceptable, and therefore observable serializability permits this execution.

This view has broader implications for our current understanding of the efficiency of TM implementations. Consider the following example from Hans et al. [2016], which is key to their proofs of lower bounds and impossibility results for some STM implementations.

```
Thread 1: (Rx0                          Wy1         )
Thread 2:                  < Rx0   Wx1  >
```

Observable serializability requires that T1 be ordered before T2, due to the from-read anti-dependency from Rx0 in T1 to Wx1 in T2. This example is termed "reverse-commit anti-dependency" because T2 commits before T1, yet the serialization must order T1 before T2. Since the commit order between the two transactions is not directly observable, this execution is permitted by observable serializability.

1.2 Non-transactional Events and Isolation

We now explore the interaction between transactional and non-transactional memory events, often characterized by a notion known as *isolation* [Blundell et al. 2005], i.e., when and how can a non-transactional event interact with transactional events in other threads. Inspired by Grossman et al. [2006] and Dalessandro et al. [2010], we use an *order-based approach* and define:

- *isolated* transactions, which do not permit a non-transactional event to be both causally preceded and followed by events that are part of the same transaction, and
- *relaxed* transactions, which do permit interleaved causal orders.

Consider the following example [Blundell et al. 2005], where the memory action in thread 2 is non-transactional (indicated by the absence of surrounding transaction brackets).

```
Thread 1: ( Wx1                             Wx2         )
Thread 2:                  Rx1
```

This execution is not permitted by an isolated transaction since Rx1 is reads-from ordered after Wx1 and from-read ordered before Wx2. The following variants are however acceptable for both isolated and relaxed transactions:

```
Thread 1: ( Wx1                                Wx2            )
Thread 2:                  Rx0
```

and

```
Thread 1: ( Wx1                                Wx2            )
Thread 2:                  Rx2
```

Thus, we permit one-way order, where the non-transactional reads are ordered either before or after the memory events of T1.

1.3 Aborted Transactions and Observability

We now discuss the implications of aborted transactions in the context of observable serializability. The literature on TMs reflects an effort to reduce unnecessary aborts with the aim of improving performance [Guerraoui et al. 2008; Hans et al. 2016; Keidar and Perelman 2009] or to eliminate aborts altogether [Afek et al. 2012]. Observable serializability may be preferable to strict serializability in this regard, since it requires fewer aborts, as discussed above.

The absence of a global real-time order and explicit notions of causality in memory models provides a different perspective on the status of aborted transactions (see Dziuma et al. [2015] for a survey). In particular, the use of observable serializability leads to a concomitant increase in flexibility over conditions such as *opacity* [Guerraoui and Kapalka 2008, 2010] and *TMS2* [Doherty et al. 2013]. Consider the following example:

```
Thread 1: ( Rx2            abort)
Thread 2:       [ Wx2    Ry1                      ]
Thread 3:                                < Wy1   >
```

Here we use notation "abort)" to denote a transaction that ends with an abort. Note that due to reads-from dependencies, T3 is causally ordered before T2, which is in turn causally ordered before T1, and hence, their serialization must respect this ordering even though T1 is an aborted transaction. Final-state opacity [Guerraoui and Kapalka 2008] does not allow this execution since T1 occurs before T3 in real-time order, creating a cycle with respect to the existing causal order. However, observable serializability does permit this execution since it does not impose a real-time ordering constraint. We reemphasize that such a liberalization of real-time order *does not* alter a programmer's view of opaque transactions since all observable orders are preserved.

Discarding real-time order also impacts other models of aborted transactions such as TMS1 [Doherty et al. 2013]. From a programmer's perspective of the opacity model, modifications to thread-local variables made by an aborted transaction are not rolled back (e.g., because local caches may not be cleared) [Attiya et al. 2013]; thus, the client can detect the presence of an aborted opaque transaction. This is contrasted with the programmer perspective of TMS1 [Attiya et al. 2014], which rolls back the local state of a transaction if it aborts. A client cannot be affected by an aborted undoable transaction; so, the presence of such an aborted transaction is undetectable by the user. We refer to TMS1-style transactions as *undoable* transactions.

The justification for the actions in an aborted undoable transaction in Doherty et al. [2013] is based on the construction of a local history of the aborted transaction. The real-time ordering makes the construction of such a local history subtle. In contrast, the causality order provides a ready-made view of the local history of a transaction. Consider the following example from Doherty et al. [2013, Figure 5].

```
  Thread 1: (Rx0                      Wy1      )
  Thread 2:      [ Wx2         ]
  Thread 3:           < Rx2       Ry0                     abort>
```

First note that there is no possible serialization order among the three transactions, as detected by an *observable cycle* (cf. [Alglave et al. 2014]):

- T2 must be reads-from ordered before T3.
- T3 must be from-read ordered before T1 (since T1 overwrites the value of y read by T3).
- T1 must be from-read ordered before T2 (since T2 overwrites the value of x read by T1).

So, this execution does not satisfy the (final-state) opacity criterion, even in the relaxed world (where real-time order is ignored). On the other hand, if we assume transactions are undoable, using a TMS1-style notion of correctness, we are able to perform two separate serializations:

(1) serialize the committed transactions T2 and T3, excluding the aborted T1, and
(2) serialize the aborted transaction T1 with its unique causal predecessor, T2.

Since both of these serializations are valid, the full execution is validated. This model is consistent with programmer-centric view that aborted transactions can be removed and a system replayed with only committed transactions.

The restriction to a causal history in this analysis is reminiscent of the VWC criterion [Imbs and Raynal 2012] for aborted transactions and implementations thereof (e.g. [Diegues and Romano 2015]). The observable order of a relaxed memory model provides a robust operational foundation for the partial histories that motivates VWC and TMS1. In Section 4, we show an example that is permitted by our treatment but not by VWC.

1.4 Our Results

The main contribution of this paper is a framework for the integration of transactions into hardware relaxed memory models.

Our work is placed in the context of the framework described by Alglave et al. [2014], described in Section 2. We establish the following results.

- In Section 3, we describe a semantics of transactions as an addition to any architecture captured by the framework of Alglave et al. [2014]. The architectures that we can address include SC, TSO, PPC and the recently developed ARMV8. The semantics is local, i.e., it only incorporates ordering constraints in the style of the standard specifications in [Alglave et al. 2014]. In particular, it does not assert the existence of global orders on transactions.
- Our framework for transactions is very general, providing variations along two dimensions: isolated vs relaxed, abortion models of opaque vs undoable. In addition, transactions can be nested, and placed flexibly with respect to program order and threads. We present several examples in Section 4.
- Sections 5 and 6 present specialisations to ghb-architectures [Alglave 2010], i.e., those that ensure a *global happens-before* relation. In Section 5, we describe a novel automata-based characterization of execution in ghb-architectures.
- For ghb-architectures, we show that any program execution is equivalent to one with a global total order on transactions that explains the execution's reads, and in which the operations of any given transaction are contiguous. This shows that the local constraints of the semantics suffice to ensure that transactions behave as expected from a programmer's perspective. We discuss this and other results in Section 6.
- We describe a small programming language in Section 7. The executions of this language illustrate the observational power afforded by our model of transactions.

- We have used MemAlloy [Wickerson et al. 2017] and the results of Chong et al. [2017] to relate our approach to models of extant transactional hardware. We refer to Section 8 for detailed comments.

2 A MODEL FOR WEAK MEMORY AND TRANSACTIONS

Alglave et al. [2014] provide an exhaustive study of relaxed memory models. This is a technical tour-de-force that is expressive enough to account for a variety of architectures. We use their approach as a tool to introduce the key ideas behind relaxed memory and as the setting for our addition of transactional features.

In this section, we provide a brief overview, referring the interested reader to the original paper [Alglave et al. 2014] and thesis [Alglave 2010] for further details. In this section and those following, we consider examples from several memory models: SC (Sequential consistency), TSO (Total store ordering), ARMv7 (ARM version 7) and ARMv8 (ARM version 8).

An *event* e is a tuple consisting of a unique identifier, an action label, and other data such as thread identifiers. *Actions* include reads and writes, as well as architecture-dependent actions such as fences. Reads and writes operate on a *memory*. We use $x \in \mathbb{N}$ for memory addresses and $v \in \mathbb{N}$ for memory values.

An *execution* is a tuple consisting of a set of events and some relations over those events (described below). For any execution, one is able to derive a happens-before relation hb over events (which formalizes the notion of *causality*). We write relations using both set and arrow notation. For relation o, "$d \xrightarrow{\text{o}} e$" is synonymous with "$\langle d, e\rangle \in \text{o}$" and "$d \text{ o } e$".

A *pre-execution* is a tuple comprising a set of events and a strict subset of the relations used to define an execution. Let $\mathbb{E}$ range over pre-executions and $\mathcal{E}$ over executions.

An *architecture* $\mathcal{A}$ is a function from pre-executions to executions. To simplify notation, we assume that $\mathcal{A}(\mathbb{E})$ extends $\mathbb{E}$. A *pre-execution* has the form $\mathbb{E} = \langle E, \text{po}, \text{rf}, \text{co}, \text{data}, \text{addr}, \text{ctrl}\rangle$ and an *execution* has the form $\mathcal{A}(\mathbb{E}) = \langle \mathbb{E}, \text{ppo}, \text{fences}, \text{prop}\rangle$. The relations in a pre-execution have the following intended interpretations, where $\mathsf{M}xv$ is a memory event, i.e., either $\mathsf{R}xv$ or $\mathsf{W}xv$.

- po (program order), which defines a total order on the actions of each thread. Actions of different threads are unrelated.
- co (coherence order), which defines a total order on the writes to each location. Writes of different locations are unrelated.
- rf (reads from), which maps each read, with label $\mathsf{R}xv$ (for any x, v), to with a unique matching write, with label $\mathsf{W}xv$ (for the same x, v).
- addr (address dependency), where a source read is causally linked to the value x that is subsequently accessed by the target event $\mathsf{M}xv$.
- data (data dependency), where a source read causally linked to the value of v that is subsequently accessed by the target event $\mathsf{M}xv$.
- ctrl (control dependency), where a source read is causally linked to a branch between itself and the target event.

The architecture of an execution determines three derived relations:

- ppo (preserved program order), which is a suborder derived from po by removing order between actions that commute according to the architecture.
- fences, which relates events in po that are separated by a fence.
- prop (propagation order), which relates writes that must propagate to memory in a particular order. Propagation order is distinct from co, which only relates writes on the same location.

Example 2.1. Consider the following execution.

```
Thread 1: Wx1          Ry0
Thread 2:    Wy1    Rx0
```

The only possible rf relation is $\{\langle Rx0, Wx0\rangle, \langle Ry0, Wy0\rangle\}$, both reading initial values. (Since all events have distinct labels, we let the label stand for the corresponding event.) The intended co order is $Wx0$ (the initializing write to x) before $Wx1$, and $Wy0$ (the initializing write to y) before $Wy1$. The execution above can be observed in architectures, such as TSO, that do not preserve program order between a write and a read, but is not observable under SC. □

To formalize the notion of an allowable execution, Alglave et al. [2014] define the following additional relations. First, for any relation o, we can define an "external" restriction, denoted by appending an e, as follows: $\mathrm{oe} = \{\langle e, d\rangle \mid \langle e, d\rangle \in \mathrm{o} \wedge \mathrm{thread}(e) \neq \mathrm{thread}(d)\}$. Thus, rfe relates read events from a thread to a matching write from another thread. The additional relations are defined as follows.

- $\mathrm{fr} = \{\langle r, w_1\rangle \mid \exists w_0. w_0 \xrightarrow{\mathrm{rf}} r \wedge w_0 \xrightarrow{\mathrm{co}} w_1\}$ (from-read order), where r is a read, and w_1 is a write which must come after r, since r has seen a write that preceded w_1 on the same location. In Example 2.1, the from-read relation for x is $\{\langle Rx0, Wx1\rangle\}$ and similarly for y.
- $\mathrm{hb} = \mathrm{ppo} \cup \mathrm{fences} \cup \mathrm{rfe}$ (happens-before order), which orders causally related events.
- $\mathrm{com} = \mathrm{co} \cup \mathrm{rf} \cup \mathrm{fr}$ (communication order), which combines the relations that constrain the order of reads and writes to the memory subsystem.
- $\mathrm{poloc} = \{\langle e, d\rangle \in \mathrm{po} \mid \mathrm{location}(e) = \mathrm{location}(d)\}$ (program order per location), which is program order, restricted to the same location.

Example 2.2 ([Alglave et al. 2014]). In SC, all program order is maintained in ppo. Thus, there is no need for fences and we have:

$$\mathrm{ppo} = \mathrm{po} \qquad \mathrm{fences} = \emptyset \qquad \mathrm{prop} = \mathrm{po} \cup \mathrm{rf} \cup \mathrm{fr}$$

In contrast to SC, under TSO, ppo is derived from po by removing all orders from writes to reads (denoted WR). The fences relation coincides with the mfence instruction (see [Alglave et al. 2014] for details).

$$\mathrm{ppo} = \mathrm{po} \setminus \mathrm{WR} \qquad \mathrm{fences} = \mathrm{mfence} \qquad \mathrm{prop} = \mathrm{ppo} \cup \mathrm{rfe} \cup \mathrm{fr}$$

The early read available to local writes is reflected in the fact that only rfe contributes to global propagation. We will soon see the impact of this distinction on the validity of execution. □

An execution $\mathcal{E}$ is considered to be *valid*, denoted correct($\mathcal{E}$), if it satisfies the following axioms:

$$\begin{array}{ll} \mathrm{acyclic}(\mathrm{hb}) & (\textit{NoThinAir}) \\ \mathrm{acyclic}(\mathrm{poloc} \cup \mathrm{com}) & (\textit{SCPerLocation}) \\ \mathrm{irreflexive}(\mathrm{fre}; \mathrm{prop}; \mathrm{hb}^*) & (\textit{Observation}) \\ \mathrm{acyclic}(\mathrm{co} \cup \mathrm{prop}) & (\textit{Propagation}) \end{array}$$

By *NoThinAir*, causality cannot be cyclic; by *SCPerLocation*, each location taken separately is sequentially consistent; by *Observation*, actions hidden in the causal past cannot be observed; and by *Propagation*, writes must be propagated in an order consistent with coherence.

Example 2.3. Returning to the execution in Example 2.1, for SC, the execution has a cycle:

$$Ry0 \xrightarrow{\mathrm{fr}} Wy1 \xrightarrow{\mathrm{ppo}} Rx0 \xrightarrow{\mathrm{fr}} Wx1 \xrightarrow{\mathrm{ppo}} Ry0$$

In TSO, this cycle is not present since the two ppo edges are not present. □

3 MODELING TRANSACTIONS

This section develops an observation-based notion of correctness for executions extended with transactional events. We develop a flexible axiomatic framework that describes the behaviors of many different types of transactions, including nested transaction, and their interaction with transactional and non-transactional events.

A key component of our framework is a mechanism for lifting standard relations to the level of transactions (see Definition 3.1). Using this, we develop a "per transaction" notion of correctness (see Definition 3.8), which enables different types of transactions to exist within a single system.

3.1 Characterizing Transaction Types

We first describe the different types of transactions permitted within our framework and informally describe some of their inter-relationships and characteristics.

Let TransactionId be a set of transaction identifiers. A transaction $t \in \mathsf{TransactionId}$ can be nested inside another $s \in \mathsf{TransactionId}$, and hence, we assume TransactionIds are arranged as a tree $\geq_{\mathsf{nest}}$, where $s \geq_{\mathsf{nest}} t$ iff t is nested inside of s. The root of the tree is a distinguished top element, $\top$, such that for any $s \in \mathsf{TransactionId}$, we have $\top \geq_{\mathsf{nest}} s$. TransactionIds can be partitioned in different ways, as follows.

- Transactions may be Committed or Aborted. As a system executes, there may be *live* transactions that are not yet Committed or Aborted. In this case, we say that the execution is valid as long as each live transaction can be assigned either Committed or Aborted, and the resulting execution is valid.
- Transactions can be declared as Relaxed or Isolated. This attribute influences the transaction's interaction with non-transactional code. In both cases, events within the same transaction cannot be causally interleaved with other transactions. However, while non-transactional events may be causally interleaved with the events of a relaxed transaction, this is forbidden in an isolated transaction.
- Transactions can be declared as Opaque and Undoable. This attribute determines the transaction behavior in the case that it is aborted. The programming language intuition for this division is as follows.
  - The intuition behind Opaque transactions is that every Opaque transaction must fit within a total order of committed transactions. Such a history is perforce global; so, aborted Opaque transactions have to be validated as being consistent with this global perspective. The existence of an aborted Opaque transaction is visible to the execution, and therefore these cannot simply be removed.
  - On the other hand, an aborted Undoable transaction can be replaced by a `skip`, leaving behind an execution that consists solely of Committed and aborted Opaque transactions. That is, aborted Undoable transactions can leave no residue of their execution. Consequently, we expect the future actions that depend on an aborted Undoable transaction to also be part of an aborted Undoable transaction. The Undoable transactions are validated only with respect to their own local causal history, permitting great flexibility in executing them, e.g., with speculation.

We assume that the top transaction $\top$ is Committed, Relaxed and Opaque. Since there must be a relationship of containment between parent and child transactions, we require that if $s \in \mathsf{Aborted}$ and $s \geq_{\mathsf{nest}} t$ then $t \in \mathsf{Aborted}$. Moreover, if a parent transaction is Isolated, then all of its children will behave as though they are Isolated, regardless of whether they are declared to be Isolated or Relaxed. It is reasonable to require that every Undoable transaction is also Isolated: if a thread cannot use a value from an aborted transaction, we should also not allow values from the aborted transaction to be seen by non-transactional events. Thus, we have three overall categories of transactions: $\mathsf{RelaxedOpaque} = \mathsf{Relaxed} \cap \mathsf{Opaque}$, $\mathsf{IsolatedOpaque} = \mathsf{Isolated} \cap \mathsf{Opaque}$ and $\mathsf{Undoable} = \mathsf{Isolated} \cap \mathsf{Undoable}$.

Since we are developing a general framework, we place very few restrictions on transactions. Below we list some surprising flexibility in our model.

- We permit transactions that are not contiguous in po. Such transactions are permitted in the transaction architecture for Power [Cain et al. 2013]. Consider:

  $<\ \mathtt{W}x1 \quad \mathtt{yield} \quad \mathtt{W}y2 \quad \mathtt{resume} \quad \mathtt{W}z3\ >$

  The atomic code of the transaction is $<\ \mathtt{W}x1\ \ \mathtt{W}z3\ >$ whereas the code between the yield and resume ($\mathtt{W}y2$ in this case) is not part of the transaction. The motivation for this feature in the Power architecture is as an aid in debugging, e.g., to observe the intermediate states of the transaction, or to provide intermediate values for the rest of the transaction. Our model supports these use cases for relaxed transactions. Of course, there could be more than one such debugging point in a transaction.
- We do not confine a transaction to be inside a single thread. This interesting feature of parallelism inside a transaction has already been investigated in the STM context by Diegues and Cachopo [2013], where the aim is to describe coarse-grained transactions that permit efficient implementations on multicore systems.
- Following the classical STM view of composable transactions [Haines et al. 1994; Harris et al. 2005], we permit unrestricted combinations of all the above features and nesting.

Our intent is not that a given architecture permit the full flexibility of our transaction model. Rather, our aim is to establish general principles and properties that hold even when restricted to subsystems of our full model.

3.2 Transactions and Pre-executions

We now describe a method for introducing transactions into a pre-execution and describe a technique for enhancing the existing orders in the pre-execution (via lifting) to take transactional behavior into account.

For any pre-execution $\mathbb{E}$, let $\mathsf{trans} : E \longrightarrow \mathsf{TransactionId}$ be a (total) function that maps each event of $\mathbb{E}$ to a $\mathsf{TransactionId}$. Non-transactional events are mapped to $\top$ and we let

$$\mathsf{TransEv} = \{e \in E \mid \mathsf{trans}(e) \neq \top\}$$

be the set of *transactional events*. We use a set $\mathsf{descend}(t)$, which includes events assigned to the same or nested transactions, i.e.,

$$\mathsf{descend}(e) = \{e' \mid \mathsf{trans}(e) \geq_{\mathsf{nest}} \mathsf{trans}(e')\}$$

Note that if $\mathsf{trans}(d) \geq_{\mathsf{nest}} \mathsf{trans}(e)$ then $\mathsf{descend}(d) \supseteq \mathsf{descend}(e)$. For any set $T \subseteq \mathsf{TransactionId}$, we write $e \in T$ as shorthand for $\mathsf{trans}(e) \in T$.

It is common in the literature on transactions to include events that denote the beginning and end of transactions. While not strictly necessary, such markers are convenient in some definitions and examples. To denote these markers, we augment the set of actions with transaction begin/commit/abort actions, which are lifted to events in the standard manner. As discussed above, we use pairs of brackets, e.g., `( )`, to denote matching begin and commit events, and prefix the closing bracket with `abort` to denote a transaction that aborts. As is common in the literature, we assume that the begin marker causally precedes every event in the transaction and the end marker causally follows every event in the transaction.

We now define our mechanism for lifting relations over events to the level of transactions, which is possible if the source or target of the relation is a transactional event.

Definition 3.1. Suppose $\mathsf{o} \subseteq E \times E$. Define $\mathsf{lift}(\mathsf{o})$ so that $e \xrightarrow{\mathsf{lift}(\mathsf{o})} d$ whenever either

(1) $e \xrightarrow{\mathsf{o}} d$, or
(2) $e' \xrightarrow{\mathsf{o}} d$ for some $e' \in \mathsf{descend}(e)$, $d \notin \mathsf{descend}(e)$ and $e \notin \mathsf{Undoable} \cap \mathsf{Aborted}$ and either $e \in \mathsf{Isolated}$ or $d \in \mathsf{TransEv}$, or
(3) $e \xrightarrow{\mathsf{o}} d'$ for some $d' \in \mathsf{descend}(d)$, $e \notin \mathsf{descend}(d)$ and $d \notin \mathsf{Undoable} \cap \mathsf{Aborted}$ and either $d \in \mathsf{Isolated}$ or $e \in \mathsf{TransEv}$. □

The purpose of lift is to make the events in a transaction appear atomic. We do this by extending every order o to include all of the events of the transaction. By Definition 3.1(1), we have $\mathsf{o} \subseteq \mathsf{lift}(\mathsf{o})$. The next two clauses deal with o-successors and o-predecessors of transactional events, respectively. We discuss the successor clause in detail; the predecessor clause is similar.

Roughly, we require that $e \xrightarrow{\mathsf{lift}(\mathsf{o})} d$ when $e' \xrightarrow{\mathsf{o}} d$ for some e' such that $\mathsf{trans}(e') = \mathsf{trans}(e) \neq \mathsf{trans}(d)$. The three conjuncts of Definition 3.1(2) refine this as follows.

- To handle nested transactions, we require $e' \in \mathsf{descend}(e)$ and $d \notin \mathsf{descend}(e)$ rather than just $\mathsf{trans}(e') = \mathsf{trans}(e)$ and $\mathsf{trans}(d) \neq \mathsf{trans}(e)$.
- Actions that are both Aborted and Undoable may behave non-atomically, so we don't want to lift in this case. Therefore we require $e \notin \mathsf{Aborted} \cap \mathsf{Undoable}$.
- Whereas Isolated transactions must be atomic to all other events, Relaxed transactions need only be atomic to all other transactional events. Therefore we require either $e \in \mathsf{Isolated}$ or $d \in \mathsf{TransEv}$.

We now consider three examples that provide intuition on our lifting construction.

Example 3.2 (Empty and singleton transactions). An *empty transaction* is characterized by a pair of begin and end markers (possibly aborting) with no transactional memory event. Empty transactions do not affect any lifted relation, or in other words, an execution is unaltered by the addition or removal of empty transactions.

A *singleton transaction*, instead, contains a single memory event. Singletons also do not contribute to any lifted relation. Hence, an execution that contains only singleton committed transactions is valid if, and only if, the execution is valid when none of the events are transactional. In the presence of non-singleton transactions, an event e in a singleton transaction is still different from the event e being non-transactional. □

Further examples for empty and singleton transactions are given in Section 4.2.

Example 3.3 (Isolated vs relaxed transactions). Consider the execution with an isolated transaction T1 and a relaxed transaction T3. Recall that we assume all variables are initialized to 0.

```
Thread 1: [ Wx1 Wu1]
Thread 2: Ry1  Rx1
Thread 3: <Wy1 Wz1 Ru1 >
```

Consider above execution with $\mathsf{W}x1 \xrightarrow{\mathsf{rf}} \mathsf{R}x1$, $\mathsf{W}y1 \xrightarrow{\mathsf{rf}} \mathsf{R}y1$ and $\mathsf{W}u1 \xrightarrow{\mathsf{rf}} \mathsf{R}u1$. In the lifted relation, we get $\mathsf{W}u1 \xrightarrow{\mathsf{lift}(\mathsf{rfe})} \mathsf{R}x1$, but no edge from $\mathsf{W}z1$ to $\mathsf{R}y1$, because T3 is relaxed. We also get lifted edges from both $\mathsf{W}x1$ and $\mathsf{W}u1$ to all of $\mathsf{W}y1$, $\mathsf{W}z1$ and $\mathsf{R}u1$.

This example also illustrates the earlier comment that lifting does not preserve transitive closure, since we get no edge from $\mathsf{W}x1$ to $\mathsf{R}y1$, even though they are transitively linked via T3. □

Example 3.4 (Parent and child transactions). Consider the following execution where T1 is a nested transaction comprising an outer transaction [T1] and an inner transaction <T1>. We use labels to distinguish the identical actions in the execution.

```
Thread 1: [ Wx1  <Wy1 Rz1>  Ru1]
Thread 2: a:Ry1  b:Rx1  Wz1 Wu1
Thread 3: ( c:Ry1)
Thread 4: {d:Rx1}
```

There is a unique write fulfilling each read, thus determining rf. Let us consider all possible cases.

- [T1] and <T1> are both isolated. In this case, the lifted relation introduces lifted rfe edges
  - from all events in T1 to a:Ry1, b:Rx1, c:Ry1, and d:Rx1 (by Definition 3.1(2)), and
  - from Wz1 and Wu1 to all the events in T1 (by Definition 3.1(3)).

  This highlights the fact that nested isolated transactions can be flattened into a single isolated transaction.
- [T1] is isolated and <T1> is relaxed. This case has the same behavior as when both transactions are isolated. This shows that the stronger isolation guarantees of the outer transaction overwrites the isolation guarantees of any inner transaction.
- [T1] is relaxed and <T1> is isolated. In this case, the lifted relation adds edges:

  $$\mathsf{R}z1 \xrightarrow{\text{lift(rfe)}} \mathsf{a{:}R}y1 \qquad\qquad \mathsf{W}z1 \xrightarrow{\text{lift(fre)}} \mathsf{W}y1$$

  in addition to edges to the events in threads 3 and 4 from all events of the first thread.
- [T1] and <T1> are both relaxed. In this case, the lifting introduces edges from all events of T1 to the events in T3 and T4. □

3.3 Correctness

We now formalize the notion of correctness for pre-executions enhanced with transactions. Our definition copes with each of the different types of transactions discussed in Section 3.1.

The notion of a correct transaction is defined with respect to a general notion of *causality*, formalized as a relation "causal" over events. The causal history of an event is the minimum information needed to justify the event.

We describe the general recipe for the definition of the causality relation. The causal relation always includes the following relations:

- rf ∪ data ∪ addr, thus the immediate use of a write or read is part of the causal history.
- $(\mathsf{chb} \cup \mathsf{poloc} \cup \mathsf{ctrl}) \setminus \{(d, e) \mid d \in \mathsf{Aborted}, e \notin \mathsf{Aborted}\}$, where chb is a causal happens-before relation (see below).

To justify the ignored edges in the second relation, consider an event e that is not an event of an aborted transaction, that is chb preceded by an event of an aborted transaction. In this case, we ignore the chb edges from events of the aborted transaction to e, since the events of the aborted transaction are not needed to validate e. This models the case of pure aborts, where the aborted transaction does not leave behind any residue.

As alluded to earlier, some hardware transaction models permit aborted transactions to leave a residue via a failure bit that is visible even after the abort. The above constraint does not record these dependencies in the causal history. One can imagine variants where this dependency is permitted in the causal history.

Example 3.5 (Instantiating a causal happens-before for specific architectures).

TSO: In TSO, we have:

$$\mathsf{chb} = \mathsf{hb} = \mathsf{fences} \cup \mathsf{ppo} \cup \mathsf{rfe}$$

i.e., hb is as described by Alglave et al. [2014]. In some formulations of TSO, chb includes coe and fr; we do not include them here because they do not capture causality in our sense. In particular, co (and consequently fr) is a total order on writes to the same location that is selected post-hoc — an execution is correct if there exists an appropriate co. On the other hand, there is less freedom to select an appropriate instantiation of the orders included in chb above. Thus, since the relations data, addr and ctrl are empty for TSO, we have

$$\mathsf{causal} = \mathsf{rf} \cup ((\mathsf{hb} \cup \mathsf{poloc}) \setminus \{(d, e) \mid d \in \mathsf{Aborted}, e \notin \mathsf{Aborted}\}).$$

ARMV8: In ARMV8, we have:

$$\mathsf{chb} = \mathsf{dob} \cup \mathsf{aob} \cup \mathsf{bob}$$

This definition is derived from the notion of "observed before" in the ARMV8 specification [Deacon 2017; Flur et al. 2016; Pulte et al. 2018]. Namely, dob (dependency ordered before), aob (atomic ordered before) and bob (barrier ordered before) are subcomponents of the ordered before relation.

PPC: In PPC, chb is again the happens-before relation for PPC as given by Alglave et al. [2014].

We require that the architectures under consideration preserve correctness for causal history, i.e., any correct pre-execution $\mathbb{E}$ remains correct when restricted to a causally closed subset of events of E. We let $\mathbb{E} \mid D$ denote the restriction of $\mathbb{E}$ to the events of D.

Definition 3.6. D is *causally closed* for $\mathcal{A}(\mathbb{E}) = \langle E, \ldots \rangle$ if $D \supseteq \{e \in E \mid \exists d \in D.\ e \xrightarrow{\mathsf{causal}^*} d\}$. □

Definition 3.7. Let $\mathbb{E}$ be a pre-execution. We say an architecture $\mathcal{A}$ *preserves causal history* iff for any causally-closed D, if $\mathsf{correct}(\mathcal{A}(\mathbb{E}))$ then $\mathsf{correct}(\mathcal{A}(\mathbb{E} \mid D))$. □

For the rest of this paper, we assume that architectures under consideration preserve causal history; this requirement holds for each memory model under consideration.

We now formalize our notion of correctness. For any structure $\mathbb{D} = \langle D, \mathsf{o}_1, \mathsf{o}_2, \ldots, \mathsf{o}_n \rangle$, we let $\mathsf{lift}(\mathbb{D}) = \langle D, \mathsf{lift}(\mathsf{o}_1), \mathsf{lift}(\mathsf{o}_2), \ldots, \mathsf{lift}(\mathsf{o}_n) \rangle$ be the structure obtained by applying the lifting in Definition 3.1 to each o_i pointwise.

Definition 3.8. We say a pre-execution $\mathbb{E} = \langle E, \ldots \rangle$ is *legal* iff each of the following hold, where $T = \mathsf{Undoable} \cap \mathsf{Aborted}$ and $\mathsf{rwdep} = \mathsf{rf} \cup \mathsf{data} \cup \mathsf{addr} \cup \mathsf{ctrl}$:

(1) $\mathsf{correct}(\mathsf{lift}(\mathcal{A}(\mathbb{E} \mid D)))$, where $D = E \setminus \mathsf{events}(T)$,
(2) $\mathsf{correct}(\mathsf{lift}(\mathcal{A}(\mathbb{E} \mid D)))$, for every $t \in T$, where $D \supseteq \mathsf{events}(t)$ is causally closed for $\mathcal{A}(\mathbb{E})$,
(3) $\forall d \in \mathsf{Aborted}. \forall e \in E.\ d \xrightarrow{\mathsf{rwdep}} e$ implies $e \in \mathsf{Aborted}$,
(4) $\forall d \in \mathsf{Aborted} \cap \mathsf{Undoable}. \forall e \in E.\ d \xrightarrow{\mathsf{rwdep}} e$ implies $e \in \mathsf{Aborted} \cap \mathsf{Undoable}$. □

In the next section, we provide a suite of examples to illustrate the definition. Here, we identify some of its basic features.

Definition 3.8(1) ensures that all committed transactions and all Opaque transactions (including aborted) form a valid execution. Note that in this case of the definition, we ignore the aborted Undoable transactions.

Definition 3.8(2) addresses the aborted Undoable transactions. For each of these, we are permitted to choose a *local* causal history to validate against. This means that different subhistories may be used to validate different aborted Undoable transactions. This permits Undoable transactions to be executed with a great deal of speculation and flexibility, as we illustrate by the examples in the next section. Moreover, the flexibility matches architectural transactional specifications, where global consistency is enforced only at the point of commitment.

Definition 3.8(3) ensures that the only dependencies caused by events in an aborted transaction are seen in other aborted transactions. For example, the readers of values written by an aborted transaction are also aborted transactions. This restriction is slightly looser than in STMs, where the restriction is usually that the rf-chain of aborted transactions is of length 1. Our definition is motivated by the desire to be very flexible about executing transactions as outlined above. The STM view has great relevance for maintaining general software invariants; however, it is unclear whether it is warranted in a HTM context[1].

Definition 3.8(4) ensures that the events of undoable aborted transactions do not affect any other kinds of events. This requirement allows one to ignore aborted Undoable transactions when analyzing validity of an execution, because it permits us to remove an aborted Undoable transaction and all its dependent successors.

4 IMPACT OF OBSERVATION-BASED SERIALIZATION

In this section, we explore the implications of different sorts of transactions through the lens of observation-based serializability via a series of examples, many of which are drawn from the literature. We organize the examples into four groups, focusing on speculative execution of transactions (subsection 4.1), transaction ordering (subsection 4.2), the interaction between transactional and non-transactional code (subsection 4.3), and the observability of aborted transactions (subsection 4.4). In the examples, we assume that all variables are initialized to `0`.

4.1 Speculative Execution of Transactions

We first explore a couple of examples to illustrate the flexibility available in executing Undoable transactions. Consider two undoable transactions as follows. This is a permitted execution.

```
Thread 1:  <Wx1     Wx2    >
Thread 2:  [ Rx1           abort]
```

We permit this execution because we assume transactions may execute speculatively. In particular, the second transaction speculatively reads Wx1, but it cannot commit because the read is invalidated by the second write in the committed transaction T1. So, T2 aborts. Formally, the causal history of the Rx1 only includes the first write and not the second write from T1.

Consider the following execution in TSO. This execution is permitted in the case where the sole transaction is Undoable.

```
Thread 1:  Wx1  fence    < Ry0  abort>
Thread 2:  Wy1  fence      Rx0
```

We first note that the above execution is not a valid TSO execution if Ry0 is not part of a transaction. Our model permits this execution because it is flexible with respect to aborted Undoable transactions. The causal history of the aborted transaction is the initialization of x and Wx1. The execution without the events of the aborted transaction is also clearly valid.

The motivation to permit such executions in our model is to not restrain the transactional execution even by the constraints of the underlying memory model. Of course, the above execution would not be valid if the transaction were committed; so, committed transactions do indeed have to obey the constraints of the underlying memory model.

[1]The formal properties of our model however do not depend on the presence of this restriction, so the reader might view the rest of this paper with that in mind.

4.2 Observation-Based Serialization

4.2.1 Empty Transactions. As discussed in Example 3.2, empty transactions do not contribute to any lifted relation, and hence, they do not contribute to a happens-before. Therefore they can be removed and one can pretend that they do not exist.

This treatment conflicts with the idea that transactions enforce a happens-before relation. For example, the draft C++ standard [Luchangco et al. 2013; Ni et al. 2008] says "The transaction order contributes to the synchronizes with relationship defined in the C++11 standard. In particular, each EndTransaction operation synchronizes with the next StartTransaction operation in the transaction order executed by a different thread".

Explicit coordination release is needed to *emulate* the behavior specified by the C++ standards for transactions. For example, in TSO, one could associate a variable `tord` with transactions, associate every EndTransaction with the increment of `tord` and every StartTransaction with a read of `tord`.

4.2.2 Singleton Transactions. Singleton transactions also do not contribute to any lifted relation, and hence, an execution containing only committed singleton transactions is valid exactly when the execution is valid when all of the events are non-transactional.

Singleton aborted transactions, however, do make a difference. Consider the following execution:

```
Thread 1:  < Wx1   abort>
Thread 2: Rx1
```

The execution is disallowed due to Definition 3.8(3), but would be allowed if Wx1 was non-transactional.

4.2.3 Control Dependencies from Aborted Transactions. We do not allow aborted Undoable transactions to leak information via control-dependencies. To see where this makes a difference, consider the code block:

```
atomic{S1; b=committed?1:0}; if (b) S2
```

where `b` is a local boolean-valued register that is set to 1 if the transaction in the `atomic` block commits, and to 0 otherwise. We assume `b=committed?1:0` is executed as the last statement in the transaction even if `S1` aborts halfway through execution. Thus, the program branches based on the value of `b`, setting up a control dependency from the actions in `S1` to those in `S2` even if the transaction aborts.

Our model does not permit such behavior for undoables, however, we can *emulate* the detection of the abortion by associating a boolean variable `b` with each transaction that is set to 0 before the transaction and to 1 as the last action in the transaction. For example, the code block above can be emulated by:

```
b=0; atomic{S1; b=1}; if (b) S2
```

Hence, the fact that a transaction has aborted can be detected even when control dependencies are disallowed.

4.3 Mixed-Mode Transactions and Isolation

In this section, we consider the interactions between transactional and non-transactional code, highlighting the differences between isolated and relaxed transactions.

4.3.1 Non-interference. In a mixed-mode context, a non-transactional write could interfere with a reads within a transaction. Following Blundell et al. [2005], we take the view that isolated transactions satisfy non-interference, where relaxed transactions may not. Consider the following execution, from [Grossman et al. 2006, Figure 2]. This execution is allowed if T1 is relaxed and disallowed if T1 is isolated.

```
Thread 1:  <Rx1      Rx2    >
Thread 2: Wx1
Thread 3: Wx2
```

The memory model induces two external reads-from edges: Wx1 $\xrightarrow{\text{rfe}}$ Rx1 and Wx2 $\xrightarrow{\text{rfe}}$ Rx2.

If T1 is relaxed, no new edges are added by lifting. So, the execution is considered to be valid. However, if T1 is isolated, transactional lifting introduces new edges as follows.

- The rfe edges respectively induce lifted edges Wx1 $\xrightarrow{\text{lift(rfe)}}$ Rx2 and Wx2 $\xrightarrow{\text{lift(rfe)}}$ Rx1.
- If Wx1 $\xrightarrow{\text{coe}}$ Wx2 then we have Rx1 $\xrightarrow{\text{fre}}$ Wx2, which creates a lifted edge Rx1 $\xrightarrow{\text{lift(fre)}}$ Wx2. Thus, we have a cycle between Rx1 and Wx2.
- The case where Wx2 $\xrightarrow{\text{coe}}$ Wx1 is symmetric.

4.3.2 *Containment.* The *containment* property [Blundell et al. 2005] states that whenever a write in a committed transaction is overwritten by another write to the same location in the same transaction, the first write should not be visible, even to a non-transactional read. Isolated transactions satisfy containment whereas relaxed transactions do not.

Consider the following execution, from [Grossman et al. 2006, Figure 3]. This execution is allowed if T1 is relaxed and disallowed if T1 is isolated.

```
Thread 1:  <  Wx1  Wx2>
Thread 2: Rx1
```

The memory model induces edges Wx1 $\xrightarrow{\text{rfe}}$ Rx1 and Rx1 $\xrightarrow{\text{fre}}$ Wx2. If T1 is relaxed, no new edges are added via lifting, so the execution is valid. However, if T1 is isolated, Wx1 $\xrightarrow{\text{rfe}}$ Rx1 induces Wx2 $\xrightarrow{\text{lift(rfe)}}$ Rx1, which in combination with Rx1 $\xrightarrow{\text{fre}}$ Wx2 creates a cycle between Rx1 and Wx2.

4.3.3 *Non-transactional Code Affects Containment.* The following example shows how non-transactional code has the effect of weakening containment, even between transactions. This phenomenon is not observable for isolated transactions.

Consider the following execution, from [Grossman et al. 2006, Figure 5]. This execution is allowed if T1 is relaxed and disallowed if T1 is isolated. In this example, the idea is that the events in thread 2 result from a client executing r=x; y=r, which creates a data dependency from Rx1 to Wy1. Grossman et al. [2006] say that some find this example surprising, presumably because T3 is able to observe an intermediate state of T1.

```
Thread 1:  <Wx1     Wx2>
Thread 2:   Rx1 data Wy1
Thread 3:  [Ry1]
```

If T1 is isolated, the execution is disallowed by containment. However, if T1 is relaxed the execution is permitted. Here, the memory model creates reads-from edges Wx1 $\xrightarrow{\text{rfe}}$ Rx1 and Wy1 $\xrightarrow{\text{rfe}}$ Ry1 as well as a from-read edge Rx1 $\xrightarrow{\text{fre}}$ Wx2, but lifting adds no new edges. So, the acyclicity conditions are satisfied and the execution is valid.

4.3.4 *Interleaving Unprotected Code between Protected Code.* The following example shows that relaxed transactions may be forced to interleave with non-transactional code. Consider the following execution. This execution is allowed if T1 is relaxed and disallowed if T1 is isolated.

```
Thread 1: <Wy1    Rx1>
Thread 2: Ry1 addr Wx1
```

If the transaction is isolated, the execution is not permitted because there is a cycle. From the memory model, we have $\mathsf{W}y1 \xrightarrow{\mathsf{rfe}} \mathsf{R}y1$ and $\mathsf{W}x1 \xrightarrow{\mathsf{rfe}} \mathsf{R}x1$ as well as address dependency $\mathsf{R}y1 \xrightarrow{\mathsf{addr}} \mathsf{W}x1$. The first of these induces $\mathsf{R}x1 \xrightarrow{\mathsf{lift(rfe)}} \mathsf{R}y1$. The execution is permitted if the transaction is relaxed. In that case, any path in the execution automaton has to perforce interleave as follows:

$$\mathsf{W}y1\ \mathsf{R}y1\ \mathsf{W}x1\ \mathsf{R}x1$$

4.3.5 Thread-Safe Lazy Initialization. Consider a scenario where we wish to use a transactional write to x as a flag to indicate the publication of a second variable y that is part of the same transaction. In the terminology of Grossman et al. [2006], we refer to such a publication idiom as *thread-safe lazy initialization.* Consider the following execution, from [Grossman et al. 2006, Figure 6]. This execution is allowed if T1 is relaxed and disallowed if T1 is isolated.

```
Thread 1:  <Wx1     Wy1>
Thread 2:    Rx1 addr  Ry0
```

Note that we have $\mathsf{R}y0 \xrightarrow{\mathsf{fre}} \mathsf{W}y1$. If T1 is isolated, lifting the memory-model edge $\mathsf{W}x1 \xrightarrow{\mathsf{rfe}} \mathsf{R}x1$ induces $\mathsf{W}y1 \xrightarrow{\mathsf{lift(rfe)}} \mathsf{R}x1$, which when composed with $\mathsf{R}x1 \xrightarrow{\mathsf{addr}} \mathsf{R}y0$ yields order from $\mathsf{W}y1$ to $\mathsf{R}y0$. So, this execution is not permitted, as the address dependency forces the read of y to be 1.

However, if T1 is relaxed, lifting does not induce any new edges. So, even though there is an address dependency between the two reads in thread 2, $\mathsf{R}y0$ is permitted and the above execution is valid. In this case, the use of x as a flag to indicate the publication of y does not work as intended.

4.3.6 Aborted Writer Transactions. Our final mixed-mode example highlights the interaction between a writing transaction that aborts and a non-transactional read, and shows that writes in an aborted transaction can never be seen by a client. Consider the following execution, from [Grossman et al. 2006, Figure 4]. This execution is disallowed regardless of whether T1 is relaxed or isolated. Consider the following execution.

```
Thread 1:  <Wx1 abort>
Thread 2: Rx1
```

We do not permit rf edges from an aborted transaction to a non-transactional operation so the above execution is not allowed regardless of whether T1 is isolated or relaxed.

4.4 Aborted Transactions and Observability

Our final set of examples pertains to executions with aborted transactions and considers their interaction with observability. For each example in this section, all code is protected by a transaction, and hence, we do not need to consider the difference between isolated and relaxed transactions. Instead the focus is on the differences between opaque and undoable transactions. Throughout this section, we assume all transactions are isolated.

4.4.1 Aborted Opaque Transactions and Happens-Before. We have seen in Example 4.3.6 that the value written within an aborted transaction cannot be read. However, an aborted transaction can affect happens-before orders, which potentially affects the validity of an execution.

Consider the following execution. This execution is allowed if T1 is undoable and disallowed if T1 is opaque.

```
Thread 1: [ Wx1 ]
Thread 2:           < Wy1 >
Thread 3:                      ( Rx1 Ry0 )
Thread 4:                      {Rx0 Ry1 abort}
```

If T4 is an opaque transaction, the execution is not permitted because we have a cycle:

$$\mathtt{R}x\mathtt{0} \xrightarrow{\text{fre}} \mathtt{W}x\mathtt{1} \xrightarrow{\text{rfe}} \mathtt{R}x\mathtt{1} \xrightarrow{\text{lift(fre)}} \mathtt{W}y\mathtt{1} \xrightarrow{\text{lift(rfe)}} \mathtt{R}x\mathtt{0}$$

where $\mathtt{R}x1 \xrightarrow{\text{lift(fre)}} \mathtt{W}y1$ is induced by lifting $\mathtt{R}y0 \xrightarrow{\text{fre}} \mathtt{W}y1$ and $\mathtt{W}y1 \xrightarrow{\text{lift(rfe)}} \mathtt{R}x0$ is induced by lifting $\mathtt{W}y1 \xrightarrow{\text{lift(fre)}} \mathtt{R}y1$. Thus, even though there are no rf edges from an aborted transaction, the hb from an aborted transaction can affect the validity of an execution.

However, the execution is permitted if T4 is an undoable transaction. In this case, we have to validate the execution without T4:

```
Thread 1: [ Wx1 ]
Thread 2:           < Wy1 >
Thread 3:                        ( Rx1 Ry0 )
```

and the execution which includes the causal past of T4:

```
Thread 2:           < Wy1 >
Thread 4:                        {Rx0 Ry1 abort}
```

both of which are clearly valid.

4.4.2 TMS1 and Read Switching. In TMS1, a transactional read can be validated by "temporarily" justifying the read using an aborted transaction, then "switching" to a valid committed transaction at a later point in time. Due to this, TMS1 has the property that removal of an aborted transactions can make an execution invalid. Consider the following execution, from [Doherty et al. 2013, Figure 3]. Since the execution contains two events with the same action, we use labels a and b to distinguish them.

```
Thread 1: <a:Wx1             abort>
Thread 2:       [      Rx1      ]
Thread 3:                        ( b:Wx1 )
```

Consider a real-time order execution of the above, i.e., where Rx1 begins after a:Wx1 finishes, but before b:Wx1 begins. This execution is permitted by TMS1 by allowing Rx1 to initially read from a:Wx1, then "switching" the justifying reads to b:Wx1 when T1 aborts. Note that without T1, the execution would not satisfy TMS1 since T2 and T3 are real-time ordered and T2 does not have a validating write in its prefix.

We do not require "read-switching". Since we ignore real-time order, our definition forces Rx1 to be satisfied by b:Wx1, since there are cannot be any reads-from edges from aborted transactions. The execution is valid regardless of whether the aborted transaction is opaque or undoable.

In general, for many of the examples, the seeming extra flexibility provided by TMS1 in terms of matching is intuitively compensated for by a relaxation from real-time to observable order. On the other hand, consider the execution above without T1. As already discussed, TMS1 forbids this new execution, but when using our definitions, the execution remains valid.

4.4.3 Omitting Undoable Aborted Transactions. The next example highlights the difference between an undoable and an opaque transaction. Consider the following example from [Doherty et al. 2013, Figure 5]. This execution is allowed if T3 is undoable and disallowed if T3 is opaque.

Assuming real-time order is respected, this execution is not permitted by opacity, but is permitted by TMS1 and VWC.

```
Thread 1: (Rx0                      Wy1       )
Thread 2:     [ Wx2         ]
Thread 3:          < Rx2       Ry0                     abort>
```

The example is not permitted by our definitions if T3 is opaque because of a cycle. Note that we have memory model edges $\mathsf{W}x2 \xrightarrow{\mathsf{rfe}} \mathsf{R}x2$ and $\mathsf{R}y0 \xrightarrow{\mathsf{fre}} \mathsf{W}y1$, thus lifting gives us $\mathsf{W}x2 \xrightarrow{\mathsf{lift(rfe)}} \mathsf{R}y0$ and $\mathsf{R}y0 \xrightarrow{\mathsf{lift(fre)}} \mathsf{R}x0$, respectively. Since we also have $\mathsf{R}x0 \xrightarrow{\mathsf{fre}} \mathsf{W}x2$, the example contains a cycle. However, the example is permitted if T3 is an undoable transaction. To verify this we must consider

(1) the example without T3 (i.e., restricted to T1 and T2 only), which is clearly valid, and
(2) the causal-history of T3:

```
Thread 2:        [ Wx2        ]
Thread 3:             < Rx2       Ry0                     abort>
```

which is also clearly valid.

4.4.4 Differentiating TMS1 from VWC. The following execution, from [Doherty et al. 2013, Figure 6], is allowed if T4 is undoable and disallowed if T4 is opaque. Assuming real-time order, this example is not permitted by traditional opacity or VWC, but is permitted by TMS1. We use labels to distinguish the identical actions Rx0 in T1 and T3.

```
Thread 1: < a:Rx0           Wy1         >
Thread 2:   [Wx2      ]
Thread 3:       (        b:Rx0       Wz3      abort)
Thread 4:        {Rx2      Ry0        Rz3       abort}
```

Now consider the example using our definitions.

If both T3 and T4 are opaque, this execution is not permitted as seen by the cycle

$$\mathsf{W}x2 \xrightarrow{\mathsf{rfe}} \mathsf{R}x2 \xrightarrow{\mathsf{lift(fre)}} \mathsf{W}y1 \xrightarrow{\mathsf{lift(fre)}} \mathsf{W}x2$$

where the lifted edges are obtained from the memory-model edges $\mathsf{R}y0 \xrightarrow{\mathsf{fre}} \mathsf{W}y1$ and $\mathsf{R}x0 \xrightarrow{\mathsf{fre}} \mathsf{W}x2$, respectively.

If T3 is undoable and T4 is opaque, then the execution is illegal because an undoable transaction (T3 in this case) cannot causally influence a non-undoable transaction (T4 in this case).

If T3 is opaque and T4 is undoable, we must verify

(1) the execution restricted to T1, T2 and T3:

```
Thread 1: < a:Rx0           Wy1         >
Thread 2:   [Wx2      ]
Thread 3:       (        b:Rx0       Wz3      abort)
```

(2) the causal history of T4:

```
Thread 2:   [Wx2      ]
Thread 3:       (        b:Rx0       Wz3      abort)
Thread 4:        {Rx2      Ry0        Rz3       abort}
```

Both of these are valid.

Finally, if both T3 and T4 are undoable, we must verify

(1) the execution restricted to T1 and T2
(2) the causal history of T3:

```
Thread 1: < a:Rx0           Wx1         >
Thread 3:       (        b:Rx0       Wz3      abort)
```

(3) the causal-history of T4 shown above.

All three of these are valid.

5 GLOBAL HAPPENS BEFORE AND AN OPERATIONAL CHARACTERIZATION

A focus of this paper is architectures whose pre-executions can be enhanced with an acyclic *global happens before* relation ghb. We follow Chapter 3.3 of Alglave [2010] for the basic definitions. This class of models includes SC, TSO, and ARMV8, but does not include ARMV7 or PPC (see Example 5.3). Our main contribution in this section is a novel automata-based characterization of the executions of such models.

5.1 Global Happens Before

A global event is one which is published to every processor, and so has to be accounted for by every processor. Thus, an execution can be enhanced with a ghb relation iff *all* the memory events can be embedded into a single global timeline. While the existence of such a global timeline is self-evident for SC, perhaps surprisingly, it also applies to TSO and ARMV8. The idea is to carefully separate out local and global events. For example, in TSO, the write to a local buffer is not part of the global time line; it only gets published to the global time line when the write is flushed to main memory.[2] In the words of Alglave [2010], ghb focuses on "the history of the system from the main memory's point of view."

A ghb-*architecture* is an architecture that extends a pre-execution with a ghb relation. That is, if $\mathcal{A}$ is a ghb-architecture and $\mathbb{E}$ a pre-execution, $\mathcal{A}(\mathbb{E})$ is a tuple of the form $\langle \mathbb{E}, \text{ghb}, \dots \rangle$. We refer to any pre-execution that is extended with a ghb relation as a ghb-*execution*.

One of the contributions of this paper is to describe conditions for ghb-executions under which no explicit transaction order is necessary (Theorem 6.2). In order to state this result, we parameterize the axioms by a relation $R \subseteq E \times E$. The standard axioms are recovered by setting $R = \emptyset$.

Definition 5.1. Given a ghb-execution $\mathcal{E}$ and relation $R \subseteq\ E \times E$, we write $\text{correct}_{\text{ghb}}(\mathcal{E}, R)$ if ghb is a global happens before relation such that both of the axioms below hold:

$$\text{acyclic}(\text{ghb} \cup R) \qquad (\textit{GlobalHappensBefore})$$

$$\text{acyclic}(\text{poloc} \cup \text{com} \cup R) \qquad (\textit{SCPerLocation})$$

Example 5.2. The ghb relation for SC and TSO architectures are from [Alglave 2010] and ghb for ARMV8 is the ordered-before relation ob [Deacon 2017; Flur et al. 2016; Pulte et al. 2018].

SC: $\text{ghb} = \text{po} \cup \text{co} \cup \text{rf} \cup \text{fr}$
TSO: $\text{ghb} = \text{ppo} \cup \text{co} \cup \text{rfe} \cup \text{fr} \cup \text{fences}$
ARMV8: $\text{ghb} = \text{ob}$ □

Example 5.3. Any ghb-architecture supports multi-copy atomicity [Maranget et al. 2012]. Consider the following example (known as IRIW). As usual, all locations are initialized to `0`.

```
Thread 1: Wx1
Thread 2: Wy1
Thread 3: Rx1 addr Ry0
Thread 4: Ry1 addr Rx0
```

Here, coherence must order the initializing writes before the two write events in threads 1 and 2. Furthermore, the reads in both threads must follow program order due to an address dependency. Now, since there exists a from-read (anti-dependency) from Rx`0` to Wx`1` and from Ry`0` to Wy`1` we have a cycle in any ghb-architecture. However, this example is allowed by ARMV7, demonstrating that ARMV7 is not a ghb-architecture. □

The following lemma identifies a sufficient set of restrictions on an execution to guarantee that it has a ghb relation.

[2]A similar notion of delayed writes in global time is a central idea behind TSO-linearizability [Derrick et al. 2014].

Lemma 5.4. *Suppose* $\mathcal{E} = \langle \mathbb{E}, \ldots \rangle$ *is an execution such that* $\mathsf{correct}(\mathcal{E})$. *If* $\mathcal{D} = \langle \mathbb{E}, \mathsf{ghb} \rangle$, *then* $\mathsf{correct}_{\mathsf{ghb}}(\mathcal{D}, R)$ *provided* $\mathsf{ghb} = \mathsf{prop}$ *and* prop *is a transitive relation that satisfies:*

$$\mathsf{hb} \cup (\mathsf{fr}; \mathsf{prop}) \subseteq \mathsf{prop} \qquad \mathsf{prop}; \mathsf{poloc} = \mathsf{poloc}; \mathsf{prop} = \mathsf{prop} \qquad \mathsf{coe} \cup \mathsf{fre} \cup \mathsf{rfe} \subseteq \mathsf{prop}\ .$$

5.2 Transactions in ghb Executions

To define legality of transactions in a ghb-architecture, we apply the lifting constructions from Definition 3.1 to ghb and adapt Definition 3.8. In particular, we say a pre-execution $\mathbb{E}$ is *legal* for a ghb-architecture $\mathcal{A}$ if the conditions in Definition 3.8 hold with $\mathsf{correct}(\mathsf{lift}(\mathcal{A}(\mathbb{E} \mid D)))$ in clauses (1) and (2) replaced by $\mathsf{correct}_{\mathsf{ghb}}(\mathsf{lift}(\mathcal{A}(\mathbb{E} \mid D)), \emptyset)$.

A technical point to note is that $\mathsf{lift}(\mathsf{ghb})$ is not necessarily transitively closed. However, the transitive closure of a lift-closed relation is lift-closed. For example, $\mathsf{lift}((\mathsf{lift}(\mathsf{ghb}))^*) = (\mathsf{lift}(\mathsf{ghb}))^*$.

The lifted versions of ghb and $\mathsf{poloc} \cup \mathsf{com}$ are acyclic. In the light of our earlier discussion, acyclicity of $\mathsf{lift}(\mathsf{ghb})$ is the same as the irreflexivity of $\mathsf{lift}(\mathsf{ghb})^+$. Acyclicity of $\mathsf{poloc} \cup \mathsf{com}$ is the assertion that the transaction ordering does not contradict the total order of reads and writes for each location.

Our definitions integrate transactions with the ghb relation induced by a memory model to avoid unintuitive results.

Example 5.5 ([Dalessandro and Scott 2009]). Consider the following execution.

```
Thread 1: [Wx1 Ry0]
Thread 2: <Wy1 Rx0>
```

This example should not be permitted because both serializations of the transactions are counterintuitive. Indeed, this behavior is not permitted in our model even for relaxed transactions. From the memory model, we obtain $\mathsf{R}x0 \xrightarrow{\mathsf{fre}} \mathsf{W}x1$ and $\mathsf{R}y0 \xrightarrow{\mathsf{fre}} \mathsf{W}y1$, which become lifted edges $\mathsf{R}x0 \xrightarrow{\mathsf{lift(fre)}} \mathsf{R}y0$ and $\mathsf{R}y0 \xrightarrow{\mathsf{lift(fre)}} \mathsf{R}x0$, leading to a cycle. □

This leads naturally to an analogue of *transactional sequential consistency* [Dalessandro and Scott 2009][3], which holds if there is a global total order on events that (a) explains the execution's reads, (b) is consistent with program order of each thread, and (c) keeps the events of each transaction contiguous. In the setting of relaxed memory models, transactional sequential consistency is too strong. However, one can replace program order above by ghb, then show that any program execution is equivalent to one with a global total order on operations consistent with ghb, in which the operations of any given transaction are contiguous. We prove such an abstraction theorem later in the paper (see Corollary 6.3).

5.3 Execution Automaton

We operationalize the intuition behind ghb by developing an automaton that accepts such executions. This automata characterization is a key tool in the proof of the abstraction theorem for transactions.

The automaton generalizes the well-known store buffering operational model of TSO. In comparison to the automaton of Alglave et al. [2014]; intuitively, the ghb ordering constrains the executions considerably by only permitting paths that are linearizations of ghb.

Consider an execution $\mathcal{E}$ with event set E. We build a ghb-*automaton* $\mathcal{M}_{\mathcal{E}}$ from $\mathcal{E}$ as follows. Let $\langle \mathsf{ghb}, \mathsf{polocs} \rangle_{\mathsf{lex}}$ be the lexicographic ordering of events where we order by ghb first and by polocs second. The states of $\mathcal{M}_{\mathcal{E}}$ are triples of the form $\langle \mathsf{U}, \mathsf{L}, \mathsf{G} \rangle$, where U is the set of events that are yet to be processed, L is the set of locally visible events (i.e., those that are visible to some set of threads),

[3]Transaction sequential consistency is similar to the *StrongBasic semantics* by Moore and Grossman [2008], the *strong semantics* by Abadi et al. [2011], *strong isolation* by Harris et al. [2010] and *transactional memory with store atomicity* by Maessen and A. [2007].

and G is the set of events are globally visible. We require the following, where $\uplus$ denotes disjoint union.

$$E = \mathsf{U} \uplus \mathsf{L} \tag{1}$$
$$\mathsf{G} \subseteq \mathsf{L} \tag{2}$$
$$\forall d \in E, e \in \mathsf{G}.\ (d, e) \in (\mathsf{ghb} \cup \mathsf{com}) \implies d \in \mathsf{G} \tag{3}$$
$$\forall d \in E, e \in \mathsf{L}.\ (d, e) \in \mathsf{ghb} \implies d \in \mathsf{L} \tag{4}$$

By (1), E is partitioned into U and L. In this context, L represent events whose processing has begun. By (2), every globally visible event is also locally visible. Invariant (3) ensures that the publication order into G is consistent with the ghb order and with the com order, while invariant (4) ensures that events are processed in an order consistent with ghb. Note that a consequence of (3) is that global publication preserves the SC order per location.

The initial state of the automaton $\mathcal{M}_{\mathcal{E}}$ is $\langle E, \emptyset, \emptyset \rangle$, so all events are unprocessed. The final state of the automaton is $\langle \emptyset, E, E \rangle$, i.e, all events are processed and published globally.

There are four kinds of transitions.

Local write: If w is $\langle \mathsf{ghb}, \mathsf{polocs} \rangle_{\mathsf{lex}}$-minimal in U then:

$$\langle \mathsf{U} \uplus \{w\}, \mathsf{L}, \mathsf{G} \rangle \longrightarrow \langle \mathsf{U}, \mathsf{L} \uplus \{w\}, \mathsf{G} \rangle$$

Publication: If d is $\langle \mathsf{ghb}, \mathsf{polocs} \rangle_{\mathsf{lex}}$-minimal in $\mathsf{L} \setminus \mathsf{G}$, and $\forall e \in E.\ (e, d) \in \mathsf{com} \implies e \in \mathsf{G}$, then:

$$\langle \mathsf{U}, \mathsf{L} \uplus \{e\}, \mathsf{G} \rangle \longrightarrow \langle \mathsf{U}, \mathsf{L}, \mathsf{G} \uplus \{e\} \rangle$$

Local read: If r is $\langle \mathsf{ghb}, \mathsf{polocs} \rangle_{\mathsf{lex}}$-minimal in U, and $(w, r) \in \mathsf{rf} \setminus \mathsf{rfe}$, then:

$$\langle \mathsf{U} \uplus \{r\}, \mathsf{L}, \mathsf{G} \rangle \longrightarrow \langle \mathsf{U}, \mathsf{L} \uplus \{r\}, \mathsf{G} \rangle$$

Global read: If r is $\langle \mathsf{ghb}, \mathsf{polocs} \rangle_{\mathsf{lex}}$-minimal in U, and $(w, r) \in \mathsf{rfe}$ and $w \in \mathsf{G}$, then:

$$\langle \mathsf{U} \uplus \{r\}, \mathsf{L}, \mathsf{G} \rangle \longrightarrow \langle \mathsf{U}, \mathsf{L} \uplus \{r\}, \mathsf{G} \uplus \{r\} \rangle$$

For any transition that removes events from U, we ensure that we follow the $\langle \mathsf{ghb}, \mathsf{polocs} \rangle_{\mathsf{lex}}$ order over U; a Local write transition only checks this minimality condition. Publication makes a locally visible event, d, globally visible. Therefore, in addition to following the $\langle \mathsf{ghb}, \mathsf{polocs} \rangle_{\mathsf{lex}}$ order on the unpublished local events, we also ensure that the all writes that precede d (in com order) are already globally visible. By the assumptions on ghb, this also ensures that the entire SC prefix of this variable is already in G. In a local-read, since r reads from w in same thread in $\mathcal{E}$, we know that w precedes r in $\langle \mathsf{ghb}, \mathsf{polocs} \rangle_{\mathsf{lex}}$ order, thus allowing us to deduce that $w \in \mathsf{L}$. This read is not yet globally seen. In a global-read, r reads from w in a different thread in $\mathcal{E}$, so we know that w is ahead of r in ghb, thus allowing us to deduce that $w \in \mathsf{L}$. However, we do have to ensure that w is globally visible, hence the hypothesis $w \in \mathsf{G}$. Since the writes are published only when all the other memory actions of this location that are ahead in the com order are public, we deduce that the write w is in fact the last write on this location in G. In this case, the read is already globally visible, so we also add it to the set of globals.

The transition relation of the automaton is monotone; i.e. in any state of the automaton, the execution of a transition in a state does not disable the other enabled transitions in the state. Formally, if $\langle \mathsf{U}, \mathsf{L}, \mathsf{G} \rangle \longrightarrow \langle \mathsf{U}_1, \mathsf{L}_1, \mathsf{G}_1 \rangle$, and $\langle \mathsf{U}, \mathsf{L}, \mathsf{G} \rangle \longrightarrow \langle \mathsf{U}_2, \mathsf{L}_2, \mathsf{G}_2 \rangle$, then:

$$\langle \mathsf{U}_1, \mathsf{L}_1, \mathsf{G}_1 \rangle \longrightarrow \langle \mathsf{U}_1 \cap \mathsf{U}_2, \mathsf{L}_1 \cup \mathsf{L}_2, \mathsf{G}_1 \cup \mathsf{G}_2 \rangle$$

and

$$\langle \mathsf{U}_2, \mathsf{L}_2, \mathsf{G}_2 \rangle \longrightarrow \langle \mathsf{U}_1 \cap \mathsf{U}_2, \mathsf{L}_1 \cup \mathsf{L}_2, \mathsf{G}_1 \cup \mathsf{G}_2 \rangle$$

Thus, the branching in the automaton is best viewed as expressing deterministic concurrency, as opposed to non-determinism.

We now prove that states in the automaton are never stuck.

Theorem 5.6. *If* $\mathsf{correct}_{\mathsf{ghb}}(\mathcal{E}, \emptyset)$*, all states of* $\mathcal{M}_{\mathcal{E}}$ *have a path to final state* $\langle \emptyset, E, E\rangle$*.*

Proof. We proceed by contradiction. Consider a state $\langle \mathsf{U}, \mathsf{L}, \mathsf{G}\rangle$ where we don't have an enabled transition. Let $S \subseteq \mathsf{U}$ be the set of $\langle \mathsf{ghb}, \mathsf{polocs}\rangle_{\mathsf{lex}}$-minimal elements.

- Since the LOCAL-WRITE rule is not enabled, S does not have any writes.
- We now show that $\mathsf{L} \subseteq \mathsf{G}$. If not, consider any e that is $\langle \mathsf{ghb}, \mathsf{polocs}\rangle_{\mathsf{lex}}$-minimal in L. Since the transition is not enabled, there exists $d \in E, (d, e) \in \mathsf{com} \wedge e \notin \mathsf{G}$. Since d is not in S (otherwise e would not be $\langle \mathsf{ghb}, \mathsf{polocs}\rangle_{\mathsf{lex}}$-minimal), we deduce that there is a $d' \in S, (d', d) \in \mathsf{ghb}$. Thus, we deduce that $(d', e) \in \mathsf{ghb}$, contradicting $\langle \mathsf{ghb}, \mathsf{polocs}\rangle_{\mathsf{lex}}$-minimality of e.
- So, we deduce that S is a set of reads, say $r_1 \ldots, r_n$. Since the Local read rule is not enabled, we further deduce that all the writes matching these reads are external. Since none of the reads are enabled for the Global read rule, we deduce that none of the writes are in G. Since G agrees with L on writes, this means that for each of the satisfying writes, $w_1 \ldots w_m$, there is a corresponding read in S that precedes it in $\langle \mathsf{ghb}, \mathsf{polocs}\rangle_{\mathsf{lex}}$. Since, ghb is closed under pre and post composition with $\langle \mathsf{ghb}, \mathsf{polocs}\rangle_{\mathsf{lex}}$, we deduce that there is a cycle in ghb, which is a contradiction. □

6 ABSTRACTION FOR TRANSACTIONAL EXECUTIONS

This section establishes the main properties of ghb-executions extended with transactions. Namely, that for any ghb-architecture (which includes SC, TSO, and ARMV8, but not PPC or ARMV7), any legal pre-execution ensures client abstraction.

We first define a relation txo that relates any two transaction ids that are not in a nesting relationship. From subsection 3.1, recall that $\geq_{\mathsf{nest}}$ defines a tree over TransactionId.

Definition 6.1. A relation txo over transaction identifiers is a *transaction ordering* if for every $t, s \in \mathsf{TransactionId}$ if $t \ngeq_{\mathsf{nest}} s \wedge s \ngeq_{\mathsf{nest}} t$ then either $t \xrightarrow{\mathsf{txo}} s$ or $s \xrightarrow{\mathsf{txo}} t$.

We lift txo from transactions to events as follows: $d \xrightarrow{\mathsf{txo}} e$ whenever $\mathsf{trans}(d) \xrightarrow{\mathsf{txo}} \mathsf{trans}(e)$.

The following theorem states the essence of observation-based serializability. It ensures that given a correct ghb-execution, we can find an equivalent execution with a global total order on operations, consistent with ghb and poloc, such that the operations of any given transaction appear contiguously. The idea is that any correct execution with transactions has an intrinsic partial order on transactions; the relation R defined below represents the *observable partial order* of the execution.

Theorem 6.2. *Suppose* $\mathcal{E}$ *is a* ghb*-execution such that* $\mathsf{correct}_{\mathsf{ghb}}(\mathcal{E}, \emptyset)$*. Let* R *be a relation over* TransactionId *defined by* $R = \{(t, s) \mid t \ngeq_{\mathsf{nest}} s \wedge s \ngeq_{\mathsf{nest}} t \wedge \exists e \in t, d \in s.\ (e, d) \in (\mathsf{lift}(\mathsf{ghb}))^*\}$ *and let* txo *be a transaction ordering that linearizes* R*. Then* $\mathsf{correct}_{\mathsf{ghb}}(\mathcal{E}, \mathsf{txo})$*.*

Proof. Since $\mathsf{correct}_{\mathsf{ghb}}(\mathcal{E}, \emptyset)$ and $(\mathsf{lift}(\mathsf{ghb}))^*$ is transitive, R is a partial order. We show that *any* linearization of R that is a transaction ordering of $\geq_{\mathsf{nest}}$ satisfies the conclusion of the theorem. This stronger statement conforms with our expectation that R captures all the observable order.

In this proof, we use the linearization of the execution provided by the automaton as per Theorem 5.6. Let $\leq$ be any total order that extends txo and nesting, i.e., $t \xrightarrow{\mathsf{txo}} s \Rightarrow t \leq s$ and $t \geq_{\mathsf{nest}} s \Rightarrow s \leq t$.

Consider the path in the automaton from initial state to final state. In every state, we use the transition that is enabled by the event e as follows:

> If there is any enabled transition using a transactional event, choose the transaction that is $\leq$-minimum.

Since $(\mathsf{lift}(\mathsf{ghb}))^*$ is $\mathsf{lift}(\cdot)$ closed, all events in a transaction have identical order properties with respect to events in other transactions, i.e., for non-nested transactions t, s, if there exists an event e of t related by $(\mathsf{lift}(\mathsf{ghb}))^*$ to an event d of s, then any event e of t is related by $(\mathsf{lift}(\mathsf{ghb}))^*$ to any event d of s.

Consequently, the path construction above ensures that in the global order generated by this path of the automaton, any two non-nested transactions never overlap. □

Our main abstraction result for transactions follows as an immediate corollary. Given a legal execution, the first item considers the subexecution induced by deleting the aborted undoable transactions. For this subexecution, the corollary provides a programmer friendly perspective, via an abstract and correct execution that imposes a global total order (consistent with nesting) on transactions such that the operations of any given transaction are contiguous. The second item considers each of the aborted undoables that were removed in the first item; for each of them, it provides a similar programmer friendly explanation, but only on the transactions and memory events that are part of the causal history of the aborted undoable transactions under consideration.

To define legality of transactions in a ghb-architecture, we adapt Definition 3.8. In particular, we say a pre-execution $\mathbb{E}$ is *legal* for a ghb-architecture $\mathcal{A}$ if the conditions in Definition 3.8 hold with $\mathsf{correct}(\mathsf{lift}(\mathcal{A}(\mathbb{E} \mid D)))$ in (1) and (2) replaced by $\mathsf{correct}_{\mathsf{ghb}}(\mathsf{lift}(\mathcal{A}(\mathbb{E} \mid D)), \emptyset)$.

Corollary 6.3 (Abstraction for transactions). *Suppose $\mathbb{E}$ is a pre-execution that legal for a* ghb-*architecture $\mathcal{A}$. Let $T = \mathsf{Undoable} \cap \mathsf{Aborted}$. Then both of the following hold.*

(1) *There exists an order* txo *satisfying the conditions of Theorem 6.2 for $\mathcal{A}(\mathbb{E} \mid D)$, where $D = E \setminus \mathsf{events}(T)$.*
(2) *For every $t \in T$, there exists a txo_t satisfying the conditions of Theorem 6.2 for $\mathcal{A}(\mathbb{E} \mid D)$, where $D \supseteq \mathcal{A}(\mathsf{events}(t))$ is causally closed for $\mathbb{E}$.*

Proof. For (1), by $\mathsf{correct}_{\mathsf{ghb}}(\mathsf{lift}(\mathcal{A}(\mathbb{E} \mid D)), \emptyset)$ and Theorem 6.2, we deduce the existence of txo satisfying the conditions of Theorem 6.2 for this execution.

For (2), for any $t \in T$, we have $\mathsf{correct}_{\mathsf{ghb}}(\mathsf{lift}(\mathcal{A}(\mathbb{E} \mid D)), \emptyset)$. So, by Theorem 6.2, we deduce the existence of txo_t satisfying the conditions of Theorem 6.2 for this execution. Note that txo_t is an independent order for each t, unrelated to txo. □

The next corollary deduces that a correct execution maintains Consistency (the "C" in ACID), i.e. if every transaction is a correct transformation of the state[4], then so is a correct execution. In the following corollary, we are considering predicates $\phi(x, y)$ of the kind $x + y = 10$.

First, a couple of preliminary definitions. A predicate $\phi(x_1, \ldots, x_n)$ holds for an execution $\mathcal{E}$ if $\phi(v_1, \ldots, v_n)$ holds, where v_i is the value of the last write from a non-aborted transaction in the co order of $\mathcal{E}$. A predicate $\phi(x_1, \ldots, x_n)$ is said to be an invariant for a transaction t in an execution $\mathcal{E}$ if ϕ holds for any execution of t in isolation whenever an initial memory satisfies ϕ. We show that ϕ is invariant for any legal execution with no non-transactional writes to any of the x's.

Corollary 6.4. *Suppose $\mathbb{E}$ is a pre-execution that is legal for some* ghb-*architecture such that*

- *ϕ holds for the initial memory state, and*
- $\mathsf{trans}(e) \neq \top$, *for every e such that $\exists x.\ \exists v.\ \mathsf{label}(e) = \mathsf{W}xv$, i.e., every write is transactional.*

If ϕ is an invariant for each committed transaction in $\mathbb{E}$, then ϕ holds at the end of the execution.

[4]The actions in a transaction taken in isolation without any interference do not violate any of the integrity constraints associated with the state

Proof. We use the abstract execution order txo $= t_1, \ldots, t_n$ yielded by the first item of the Corollary 6.3 and proceed by induction on the length of txo.

Base case follows from assumption since, ϕ holds for the initial memory.

For the inductive case, consider $t_1, \ldots, t_n, t_{n+1}$. There are two cases based on the commitment status of t_{n+1}.

- t_{n+1} is aborted. Since the validity of ϕ in $\mathcal{E}$ depends only on the final writes from committed transactions in the co order, t_{n+1} does not affect it. Result follows from induction hypothesis.
- t_{n+1} is committed. By induction hypothesis, ϕ holds in the subexecution induced by $t_1, \ldots, t_n$. Since all events in $\mathcal{E}$ belong to one of t's, by Definition 3.8(3) t_{n+1} only reads from the last write (in the co order) of committed transactions in $t_1, \ldots, t_n$. Result follows from inductive hypothesis on $t_1, \ldots, t_n$ and ϕ-invariance of t_{n+1}. □

7 PROGRAMS TO GENERATE EXECUTIONS

We sketch a language of clients which allows rollback of aborted undoable transactions. By virtue of corollary 6.3, such a client cannot detect whether a legal execution includes an explicit order on transactions. Thus, these clients capture the observational power afforded by our model. The language departs from the traditional presentation of transactions in several respects.

Starting from an initial map assigning a command to each threads, the semantics generates a set of candidate pre-executions, as described in Section 2. Memory locations are represented concretely, as integers. The candidate pre-executions can then be filtered using a notion of correctness, as described in Section 3. The semantics must be instrumented to generate any dependency information required by the architecture.

Thread-local state is carried in registers. The language is abstract with respect to registers, but concrete with respect to memory addresses and TransactionIds, which are represented as integers. Thus, values, memory locations and TransactionIds are all derived from a single syntactic category of *expressions*.

Let r range over the set of Registers. Let v, t, x range over a set of Values. Let op range over a set of operators. Then the syntax of (side-effect free) *expressions* is as follows.

$$\boldsymbol{E}, \boldsymbol{T}, \boldsymbol{X} \ ::= \ r \ \mid \ v \ \mid \ \mathsf{op}(\boldsymbol{E}_1, \ldots, \boldsymbol{E}_n)$$

We use the metavariable $\boldsymbol{T}$ to represent an expression that is being used as a TransactionId, $\boldsymbol{X}$ to represent an expression that is being used as a memory location, and $\boldsymbol{E}$ to represent an expression that is being used as a value. Note that a single register may be used in all three contexts. This means that a thread can invent memory addresses and TransactionIds.

Let τ range over the set {RelaxedOpaque, IsolatedOpaque, Undoable}. Then the syntax of *commands* is as follows.

$$\begin{aligned} \boldsymbol{C} \ ::= \ & \mathsf{new}\ r{=}\boldsymbol{E}\ \mathsf{in}\ \boldsymbol{C} \mid r{=}\boldsymbol{E} \mid \boldsymbol{C}_1 \,;\, \boldsymbol{C}_2 \mid \mathsf{while}\ \boldsymbol{E}\ \mathsf{do}\ \boldsymbol{C} \mid \mathsf{if}\ \boldsymbol{E}\ \mathsf{then}\ \boldsymbol{C}_1\ \mathsf{else}\ \boldsymbol{C}_2 \\ \mid\ & \mathsf{new}\ r{=}{\star}\boldsymbol{X}\ \mathsf{in}\ \boldsymbol{C} \mid {\star}\boldsymbol{X}{=}\boldsymbol{E} \\ \mid\ & \mathsf{new}\ r{=}\mathsf{begin}\ \tau\ \mathsf{in}\ \boldsymbol{C} \mid \boldsymbol{T}[\boldsymbol{C}] \mid \mathsf{abort} \mid \mathsf{ifcommit}\ \boldsymbol{T}\ \mathsf{then}\ \boldsymbol{C}_1\ \mathsf{else}\ \boldsymbol{C}_2 \end{aligned}$$

The first line gives standard constructs to allocate and update registers and execute sequences of commands. The second line gives the constructs for reading and writing memory. The last line gives the transaction-oriented constructs.

The execution of a memory read or write causes the creation of a new event with the appropriate label. For writes, the value is determined. For reads, any value is possible. Rather than determine the rf relation as we generate the thread-local semantics, we instead allow a read to return any value, and filter the out the impossible reads later, when generating a rf relation.

Syntax for transactions is typically given using block structure. Instead, we adopt explicit transaction markers. Combined with the use of integers for TransactionIds, the use of explicit markers make it is possible to express other idioms such as suspend/resume and multi-threaded transactions. To mimic the block-structured approach in our language, one would write something such as the following.

$$\mathsf{new}\ r\mathsf{=begin\ IsolatedOpaque\ in}\ (r[\boldsymbol{C}_0]\ ;\ \mathsf{ifcommit}\ r\ \mathsf{then}\ \boldsymbol{C}_1\ \mathsf{else}\ \boldsymbol{C}_2)$$

In evaluation, this first allocates a new TransactionId and binds it to register r. Then it executes $\boldsymbol{C}_0$ as part of the new transaction and attempts to commit. Then either $\boldsymbol{C}_1$ or $\boldsymbol{C}_2$ executes, depending upon whether the transaction commits or aborts.

Whereas all memory addresses are valid, the same is not true for TransactionIds. The semantics maintains a partial map $\rho : \mathsf{TransactionId} \rightharpoonup \{\mathsf{Live}, \mathsf{Committed}, \mathsf{Aborted}\}$ and a relation R giving the nesting relations of transactions. These are defined only for TransactionIds that have been returned from a prior call to begin. To handle transaction types, we simply partition TransactionIds into three sets based on the bottom two bits, representing RelaxedOpaque, IsolatedOpaque and Undoable. At top-level, execution begins with TransactionId $\top$, which is RelaxedOpaque and always Committed. A begin adds a fresh Live transaction to ρ and places it as a child of the current transaction in R.

If a thread attempts to use a TransactionId that is not defined in ρ, it becomes *stuck* and will no longer evaluate. The same is true with registers, which belong to the transaction that created them. We design a semantics to enforce the last clause of Definition 3.8: We require that an unrelated transactions cannot read from a register belonging to an Undoable, and an Undoable cannot write to a register belonging to an unrelated transaction. Aborts cascade from parent to child: a committed child will change its state to Aborted when its parent aborts. Therefore, it is safe for a child to read a register of its parent, but not vice versa. Publicly readable registers are associated with $\top$.

Commands attributed to an aborted transaction are skipped. An abort forces the current transaction (and it's descendants) to abort. Note that the state of a proper descendant may change from Committed to Aborted when the parent aborts. However, if the current transaction is already committed, then an attempt to abort is an error, causing the thread to become stuck. In addition to explicit aborts, a live transaction may nondeterministically abort at any time.

An ifcommit causes the transaction to attempt to commit. In the case where the commit succeeds, the local registers of the committing transactions are made publicly available by reassigning them to $\top$. If the commit fails and the transaction is aborted, then no code from that transaction or any of its descendants will ever execute again; thus, its registers are effectively discarded.

8 OTHER RELATED WORK

We have already cited the related work in context throughout the paper. In this section, we provide links to some missing context.

Hardware transactional models have been explored for the Pentium and Power [Cain et al. 2013] microprocessors. Pentium either uses the older Hardware Lock Elision model or the newer Restricted Transactional Memory (RTM) instruction set. The Power model permits nested transactions and also supports a suspend and resume instruction, so the transactional instructions are not necessarily contiguous in the program order. Both models implement isolated undoable transactions. The design of a transactional architecture for the ARM microprocessor is an ongoing research project[5]. Our approach to the relaxed memory of these architectures is through the intellectual pathway provided by [Alglave 2010; Alglave et al. 2014; Deacon 2017; Flur et al. 2016; Pulte et al. 2018].

[5]See http://materials.dagstuhl.de/files/16/16471/16471.StephanDiestelhorst.Slides.pdf.

The relationship between our framework and these existing hardware transactional models is established via the pioneering work of Chong et al. [2017]. From our perspective, their paper makes three important contributions. First, building on their earlier work [Wickerson et al. 2017], they describe a tool for modeling hardware transactions. Second, they provide detailed models of several extant transactional hardware architectures and their experimentally validate their models with respect to hardware execution. Third, they describe a theoretical extension of ARMV8 with transactions. The authors graciously provided us an early version of their tool, MemAlloy[6], along with their formalizations of the hardware models.

We have used MemAlloy extensively to validate that the Isolated and Undoable subset of our framework does indeed conform to Chong et al.'s empirical observations. In particular, we are able to establish that, apart from the issue of the control dependencies on aborted transactions (see Example 4.2.3), our models permit all the executions that are permitted by the hardware transaction models. In particular, our TSO model includes all behaviors allowed by the hardware, and our ARMV8 model includes all behaviors allowed by the theoretical ARMV8 transactional memory model described in [Chong et al. 2017]. Thus, our abstraction theorem applies to both TSO and ARMV8. For PPC [Cain et al. 2013], Chong et al.'s model attempts to capture the essential features of the PPC without imposing a global order on transactions. This is related to the fact that since the PPC relaxed memory model does not satisfy multi-copy atomicity, it is not a ghb-architecture, and hence our abstraction theorem does not apply. Chong et al.'s model is more expressive than the PPC hardware specification, and our formalization of PPC permits all behaviors allowed by their model. These claims for TSO, ARMV8 and PPC have been verified up to models of size 5; beyond this MemAlloy times out. Future versions of MemAlloy will permit more extensive exploration of the state space and hence models of bigger size. We will revisit our experiments to revalidate when that happens.

While this paper focuses on transactions in hardware memory models, we have drawn extensive inspiration from transaction in software transactional memory. This thriving area of literature encompasses correctness criteria, algorithms, implementations and experiments, and has found its way into several programming languages, notably Haskell in GHC 6.4, C++ [Luchangco et al. 2013] and experimental designs and systems for Java [Jagannathan et al. 2005], C# [Abadi et al. 2009]. We refer the reader to the introductory books and surveys [Guerraoui and Kapalka 2010; Harris et al. 2010; Larus and Kozyrakis 2008; Scott 2015] and tutorials on the subject such as [Grossman et al. 2007].

We draw inspiration from two ideas found in the STM literature. First, the variety of semantic correctness conditions [Dziuma et al. 2015] that we alluded to the introduction, albeit suitably modified to the relaxed memory setting, provide a rich design space — in contrast, TSO and ARMV8 only support undoable aborted transactions. Second, the rich and varied idea of composable transactions: [Diegues and Cachopo 2013; Haines et al. 1994; Harris et al. 2005] develop methods to compose transactions with all the standard programming combinators, including sequencing, hierarchy, higher-order functions and parallel composition, providing a striking contrast to the comparatively sparse methods of composition in the hardware models above.

9 SUMMARY

We have explored an expressive model of transactions in a general framework for relaxed memory. Our key contributions are twofold: first, the recognition that observable serializability provides an opportunity to reexamine traditional formulations of correctness with the aim of validating more transactions, even while maintaining the programmer model; second, providing an analysis

[6]MemAlloy is based on (bounded) model-checking, allowing users to automatically compare hardware memory models. It is available from https://github.com/johnwickerson/memalloy.

of expressive and composable transactions inspired by STMs in the hardware context. The latter is applicable to SC, TSO and the newly designed ARMv8 architectures. Our intent is not that a given design permit all the features of our model; rather, we aim to establish general properties that hold even for restrictions of our full model.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1617175. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. This work has benefitted greatly from discussions with Nathan Chong and John Wickerson. We also thank our anonymous POPL reviewers and shepherd, Viktor Vafeiadis, for their thoughtful suggestions that have helped improve the paper.

REFERENCES

M. Abadi, A. Birrell, T. Harris, and M. Isard. 2011. Semantics of Transactional Memory and Automatic Mutual Exclusion. *ACM Trans. Program. Lang. Syst.* 33, 1, Article 2 (Jan. 2011), 50 pages.

M. Abadi, T. Harris, and M. Mehrara. 2009. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP*, D. A. Reed and V. Sarkar (Eds.). ACM, 185–196.

S. V. Adve and H.-J. Boehm. 2010. Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM* 53, 8 (2010), 90–101.

S. V. Adve and K. Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *Computer* 29, 12 (1996), 66–76.

Y. Afek, A. Matveev, and N. Shavit. 2012. Pessimistic Software Lock-Elision. In *DISC (Lecture Notes in Computer Science)*, M. K. Aguilera (Ed.), Vol. 7611. Springer, 297–311.

J. Alglave. 2010. *A shared memory poetics.* PhD thesis. Université Paris 7 and INRIA.

J. Alglave, L. Maranget, and M. Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74.

A. Armstrong, B. Dongol, and S. Doherty. 2017. Proving Opacity via Linearizability: A Sound and Complete Method. In *FORTE (Lecture Notes in Computer Science)*, Vol. 10321. Springer, 50–66.

H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. 2013. A Programming Language Perspective on Transactional Memory Consistency. In *PODC*. ACM, New York, NY, USA, 309–318.

H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. 2014. Safety of Live Transactions in Transactional Memory: TMS is Necessary and Sufficient. In *DISC (Lecture Notes in Computer Science)*, F. Kuhn (Ed.), Vol. 8784. Springer, 376–390.

M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. 2011. Mathematizing C++ concurrency. In *POPL*. ACM, 55–66.

C. Blundell, E. C. Lewis, and M. M. K. Martin. 2005. Deconstructing Transactions: The Subtleties of Atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*.

H.-J. Boehm and S. V. Adve. 2008. Foundations of the C++ concurrency memory model. In *PLDI*, R. Gupta and S. P. Amarasinghe (Eds.). ACM, 68–78.

H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Q. Le. 2013. Robust architectural support for transactional memory in the Power architecture. In *ISCA*, A. Mendelson (Ed.). ACM, 225–236.

N. Chong, T. Sorensen, and J. Wickerson. 2017. The Semantics of Transactions and Weak Memory in x86, Power, ARMv8, and C++. *ArXiv e-prints* (Oct. 2017). arXiv:cs.PL/1710.04839

L. Dalessandro and M. L. Scott. 2009. Strong Isolation is a Weak Idea. In *TRANSACT '09: 4th Workshop on Transactional Computing*.

L. Dalessandro, M. L. Scott, and M. F. Spear. 2010. Transactions As the Foundation of a Memory Consistency Model. In *DISC (Lecture Notes in Computer Science)*. Springer-Verlag, Berlin, Heidelberg, 20–34.

P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. 2006. Hybrid Transactional Memory. *SIGOPS Oper. Syst. Rev.* 40, 5 (Oct. 2006), 336–346.

W. Deacon. 2017. ARM64 cat file. https://github.com/herd/herdtools7/commit/daa126680b6ecba97ba47b3e05bbaa51a89f27b7.

J. Derrick, G. Smith, and B. Dongol. 2014. Verifying Linearizability on TSO Architectures. In *IFM (Lecture Notes in Computer Science)*, Vol. 8739. Springer, 341–356.

N. Diegues and J. Cachopo. 2013. Practical Parallel Nesting for Software Transactional Memory. In *DISC (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., New York, NY, USA, 149–163.

N. Diegues and P. Romano. 2015. Time-Warp: Efficient Abort Reduction in Transactional Memory. *ACM Trans. Parallel Comput.* 2, 2, Article 12 (June 2015), 44 pages.

S. Doherty, L. Groves, V. Luchangco, and M. Moir. 2013. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.* 25, 5 (2013), 769–799.

D. Dziuma, P. Fatourou, and E. Kanellou. 2015. *Consistency for Transactional Memory Computing.* Springer International Publishing, Cham, 3–31.

K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (Nov. 1976), 624–633.

S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *POPL*. ACM, 608–621.

D. Grossman, J. Manson, and W. Pugh. 2006. What do high-level memory models mean for transactions?. In *Memory System Performance and Correctness*. ACM, New York, NY, USA, 62–69.

D. Grossman, V. Menon, S. Srinivas, and C. Zilles. 2007. Transactional Memory in Managed Runtimes - Hardware/Software View. https://www.microarch.org/micro40

R. Guerraoui, T. A. Henzinger, and V. Singh. 2008. Permissiveness in Transactional Memories. In *DISC (Lecture Notes in Computer Science)*, G. Taubenfeld (Ed.), Vol. 5218. Springer, 305–319.

R. Guerraoui and M. Kapalka. 2008. On the Correctness of Transactional Memory. In *PPoPP*. ACM, New York, NY, USA, 175–184.

R. Guerraoui and M. Kapalka. 2010. *Principles of Transactional Memory.* Morgan & Claypool Publishers.

N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. 1994. Composing First-class Transactions. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1719–1736.

S. Hans, A. Hassan, R. Palmieri, S. Peluso, and B. Ravindran. 2016. Opacity vs TMS2: Expectations and Reality. In *DISC (Lecture Notes in Computer Science)*, C. Gavoille and D. Ilcinkas (Eds.), Vol. 9888. Springer, 269–283.

T. Harris, J. Larus, and R. Rajwar. 2010. *Transactional Memory, 2nd edition.* Morgan & Claypool Publishers.

T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. 2005. Composable Memory Transactions. In *PPoPP*. ACM, New York, NY, USA, 48–60.

M. Herlihy and J. E. B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, A. J. Smith (Ed.). ACM, 289–300.

M. P. Herlihy and J. M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

D. Imbs and M. Raynal. 2012. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theor. Comput. Sci.* 444 (2012), 113–127.

S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. 2005. A transactional object calculus. *Science of Computer Programming* 57, 2 (2005), 164 – 186.

I. Keidar and D. Perelman. 2009. On avoiding spare aborts in transactional memory. In *SPAA*, F. M. auf der Heide and M. A. Bender (Eds.). ACM, New York, NY, USA, 59–68.

L. Lamport. 1979. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Computers* 46, 7 (1979), 779–782.

J. Larus and C. Kozyrakis. 2008. Transactional Memory. *Commun. ACM* 51, 7 (July 2008), 80–88.

V. Luchangco, J. Maurer, M. Moir, H. Boehm, J. Gottschlich, M. Michael, T. Riegel, M. Scott, T. Shpeisman, M. Spear, and M. Wong. 2013. Transactional Memory Support for C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3718.pdf

J.-W. Maessen and A. 2007. Store Atomicity for Transactional Memory. *Electronic Notes in Theoretical Computer Science* 174, 9 (2007), 117 – 137. Proceedings of the Thread Verification Workshop (TV 2006).

J. Manson, W. Pugh, and S. V. Adve. 2005. The Java memory model. In *POPL*, J. Palsberg and M. Abadi (Eds.). ACM, 378–391.

L. Maranget, S. Sarkar, and P. Sewell. 2012. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf.

K. F. Moore and D. Grossman. 2008. High-level small-step operational semantics for transactions. In *POPL*, G. C. Necula and P. Wadler (Eds.). ACM, 51–62.

A. T. Nguyen. 2015. *Investigation of Hardware Transactional Memory.* Master's thesis. MIT.

Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. 2008. Design and Implementation of Transactional Constructs for C/C++. In *OOPSLA*. ACM, New York, NY, USA, 195–212.

C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell. 2018. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In *POPL*. To appear.

S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. 2011. Understanding POWER multiprocessors. In *PLDI*. ACM, 175–186.

M. Scott. 2015. Transactional Memory Today. *SIGACT News* 46, 2 (June 2015), 96–104.

J. Sevcík. 2008. *Program Transformations in Weak Memory Models.* PhD thesis. Laboratory for Foundations of Computer Science, University of Edinburgh.

P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97.

N. Shavit and D. Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95).* ACM, New York, NY, USA, 204–213.

J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides. 2017. Automatically comparing memory consistency models. In *POPL*, G. Castagna and A. D. Gordon (Eds.). ACM, 190–204.

R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. 2013. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *SC*, W. Gropp and S. Matsuoka (Eds.). ACM, 19:1–19:11.