The Semantics of Transactions and Weak Memory in x86, Power, ARMv8, and C++

NATHAN CHONG, TYLER SORENSEN, and JOHN WICKERSON

Weak memory models trade programmability for performance, while transactional memory (TM) aims for ease of programming, sometimes at a performance cost. The semantics of both have been studied in detail, but their *combined* semantics is not well understood. This is problematic because TM is currently available, either natively or via extensions, in widely-used architectures and languages such as x86, Power, and C++, all of which have weak memory models. Our work aims to provide an formal understanding of the interplay between weak memory and TM, without which, programmers cannot robustly reason about programs that use both of these important concurrent programming constructs.

Our first research contribution is the definition of axiomatic models that capture the combination of TM and weak memory as it exists in x86, Power, and C++, and one way that the ARMv8 memory model could be extended to support TM. Our second contribution is the systematic methodology we use to validate our models. This methodology is based around a new tool that is able to synthesise – automatically and exhaustively – exactly the minimal test programs that distinguish any two axiomatic models. This enables us to validate our models by comparing them against both an upper-bound model (transactional sequential consistency) and a lower-bound model (the non-transactional version of each model), and then checking that the outcomes of all the generated programs on existing hardware (where available) are consistent with our models. We further validate our models by checking compilation schemes from C++ transactions to hardware transactions.

ACM Reference format:
Nathan Chong, Tyler Sorensen, and John Wickerson. 2017. The Semantics of Transactions and Weak Memory in x86, Power, ARMv8, and C++. 1, 1, Article 1 (July 2017), 29 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Reasoning about the behaviour of a multi-threaded program requires a model of how shared memory behaves. The simplest memory model is *sequential consistency* (SC) [Lamport 1979], which guarantees that all memory operations occur in an order that respects the instruction order in each thread, and that all writes to shared memory are immediately visible to all other threads. Such strong guarantees impede performance, so modern parallel architectures (such as x86, Power, and ARMv8) and concurrent languages (such as C++) sacrifice them to varying extents. The precise guarantees provided by these *weakly consistent* (or 'weak') memory models have been widely studied in recent years (Alglave et al. [2014] provide a survey).

Where weak memory increases the burden on concurrent programmers, *transactional memory* (TM) promises to ease it. TM [Herlihy and Moss 1993] allows instructions to be gathered into *transactions*, and guarantees that each appears (to other threads) either to execute entirely and instantaneously, or not at all. TM is becoming increasingly widespread, and several popular architectures and languages now provide some degree of support for it. For instance, Power has offered TM since version 2.07 [IBM 2013], Transactional Synchronization Extensions (TSX) is available for x86 [Intel Developer Zone 2012], and work is ongoing to add transactions to the C++ standard [ISO/IEC 2015].

2017. XXXX-XXXX/2017/7-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

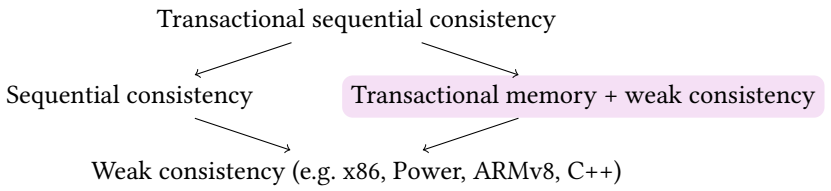


Fig. 1. The relationship between consistency and transactions. $M \rightarrow N$ means M allows fewer program behaviours than N.

Though conceptually simple, TM has proved challenging to implement correctly in practice. Cain et al. [2013] point out that it took twenty years for Herlihy and Moss's proposal to 'become a commercial reality', and still TM implementations are dogged by subtle processor defects [Hachman 2014; Intel 2017] and by security flaws due to unintended interactions between TM and other architectural features [Jang et al. 2016]. There is a significant opportunity here to improve our understanding through formal specification and verification, which has previously been successfully applied to debug feature interactions in such complex artifacts as the RISC-V architecture [Trippel et al. 2017], the C++ standard [Vafeiadis et al. 2015], and graphics processors [Alglave et al. 2015].

And indeed, the formal semantics of TM has been widely studied. However, no existing work has formalised how it interacts with the weak memory models that modern architectures and languages actually provide (see §9 for more discussion of related work). The interaction between TM and weak memory is particularly subtle because where weak memory allows *more* behaviours than SC, TM allows *fewer* (Fig. 1). Indeed, Wong [2014] argues that this interaction is among the most challenging aspects of adding TM to the C++ standard.

In this paper, we use formalisation to get a handle on the interaction between TM and weak memory. We first develop axioms that capture various forms of TM (§3), and then combine them with existing axioms for classifying memory models. In this way, we show how several implementations of TM upon weak memory models can be characterised. Specifically, we propose models for how transactions behave in x86 (§5), Power (§6), ARMv8 (§7), and C++ (§8). This range of targets demonstrates the effectiveness of our approach at both the language and the architecture level, and also at various development stages: x86 and Power have relatively mature TM implementations that we can test, C++ TM is in active development, and ARMv8 TM is currently hypothetical.

Our axiomatic models promise substantial benefits to several computing communities: to programmers, who will become able to write transactional code for these architectures with increased confidence; to compiler writers, who will be able to verify that their transformations on transactional code are correct; and to architects, who will obtain formal specifications against which their implementations can be tested.

To validate our models, we follow a novel methodology centred on a new tool that we have developed (§4). Our tool, which is based on a SAT-solving backend, is the first that is able to synthesise – automatically and exhaustively – exactly the minimal test programs (up to a bounded size) that distinguish any two axiomatic memory models. This enables us to validate our transactional models by comparing them against both an *upper-bound* model (transactional SC [Dalessandro and Scott 2009]) and a *lower-bound* model (the non-transactional version of each model), and then using the Litmus tool [Alglave et al. 2011] to check that all the test programs generated from these comparisons only exhibit behaviours on existing hardware (where available) that are consistent with our models. Our tool, which can be used to support the development of any axiomatic memory model (not just transactional ones), is being prepared for an open-source release.

Companion material. Our companion material includes all the models we propose (in the `.cat` format of Alglave et al. [2014]), the automatically-generated litmus tests that we used to validate our models, litmus tests corresponding to all the executions discussed individually in the paper, and a proposed refinement to the text of the C++ TM draft specification.

2 BACKGROUND: AXIOMATIC MEMORY MODELS

A memory model defines how threads interact with shared memory. In this section, we quickly review standard background material on axiomatic memory models, which is covered more fully by Alglave et al. [2014], Wickerson et al. [2017], and Lustig et al. [2017].

The input to an axiomatic memory model is a set of *candidate executions*. These are obtained by considering only thread-local behaviour and assuming a non-deterministic memory system. Its output is the set of executions deemed to be allowed in the presence of the actual memory system.

2.1 Executions

Let $\mathbb{X}$ be the set of all executions. Each execution takes the form of a graph, whose vertices represent runtime memory-related events, and whose labelled edges represent various types of dependency between these events. Every event in an execution belongs to zero or more of the following sets:

- E, the set of all events,
- R, the set of read events, and
- W, the set of write events.

The *loc* function maps each event in $R \cup W$ to the location it accesses. The *rval* function maps each event in R to the value it reads, and the *wval* function maps each event in W to the value it writes.

Events in an execution are connected by the following relations:

- *po*, program order (a.k.a. sequenced-before),
- *addr*/*ctrl*/*data*, an address/control/data dependency,
- *sloc*, the 'same location' equivalence (derived from the *loc* function),
- *rf*, the 'reads-from' relation, and
- *co*, the coherence order (a.k.a. modification order).

Program order (*po*) forms, for each thread, a strict total order over all the events in that thread. Address, data, and control dependencies are within program order, and always originate from a read. Address dependencies always go to a read or write, and data dependencies always go to a write. The reads-from relation (*rf*) connects writes to reads, with no read having more than one incoming *rf* edge. If $(e_1, e_2) \in rf$ then $(e_1, e_2) \in sloc$ and $wval(e_1) = rval(e_2)$. The coherence order (*co*) relates all and only writes to the same location. It forms, for each location, a strict total order over the writes to that location.

Notation. Given a relation r, r^{-1} is its inverse, $r^?$ is its reflexive closure, r^+ is its transitive closure, and r^* is its reflexive transitive closure. We use $\neg$ for the complement of a set or relation (implicitly with respect to the set of all events or event pairs in the execution). We write ';' for relational composition ($r_1 ; r_2 = \{(x,z) \mid \exists y.\, (x,y) \in r_1 \wedge (y,z) \in r_2\}$). Note that $\cap$ binds tighter than ';', which in turn binds tighter than $\cup$. We write $[s]$ to lift a set s to a subset of the identity relation ($[s] = \{(x,x) \mid x \in s\}$). For the non-transitive edges of a relation r, we write $\mathrm{imm}(r) = r \setminus (r; r^+)$. We write r_e and r_i to restrict a relation r to being inter-thread and intra-thread, respectively. That is, $r_\mathrm{e} = r \setminus (po \cup po^{-1})^*$ and $r_\mathrm{i} = r \cap (po \cup po^{-1})^*$. Finally, we write r_loc as an abbreviation for $r \cap sloc$.

Derived relations. The *from-read* relation

$$fr \quad = \quad (rf^{-1}; co) \cup ([R \setminus \mathrm{range}(rf)]; sloc; [W]) \tag{1}$$

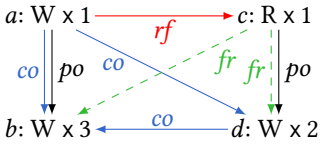


Fig. 2. A sample execution

Initially: `x = 0`	
`store(x, 1)`	`r0 = load(x)`
`store(x, 3)`	`store(x, 2)`
Test: `r0 == 1 && x == 3`	

Fig. 3. A litmus test for the execution in Fig. 2

relates each read event to all the writes that are *co*-later than the write the read observed. The first disjunct covers the case where the read observes a write in the execution; the second covers the case where the read observes zero, the value with which all locations are implicitly initialised.

Example. We depict executions using diagrams like the one in Fig. 2. Here, the *po* edges divide the execution's events into two threads. Each event is labelled either as a read (R) or as a write (W).

2.2 Consistent executions

A memory model determines which of a program's candidate executions are allowed by judging each execution against a consistency predicate. Following Alglave et al. [2014], each consistency predicate is phrased as a conjunction of acyclicity, irreflexivity, and emptiness axioms on event relations. One axiom that is common to all the models we consider is

$$\mathbf{acyclic}(po_{\mathrm{loc}} \cup com) \qquad \text{(Coherence)}$$

where

$$com \quad = \quad rf \cup co \cup fr \cup co;rf\,. \qquad (2)$$

The four disjuncts of the *com* relation capture the four ways communication can flow from an event e_1 to an event e_2: either e_2 reads from e_1, e_2 overwrites e_1, e_2 overwrites the write observed by e_1, or e_2 reads from a write that overwrites e_1. The Coherence axiom prohibits cycles formed by *com* and *po* edges on the same location, which entails that when an execution is restricted to events on any single location, it is sequentially consistent.

2.3 Generating litmus tests from executions

In order to test whether an execution of interest is observable in practice, it is necessary to convert it into a *litmus test* (i.e., a program with a postcondition) [Collier 1992]. This litmus test is constructed so that the postcondition only passes when the particular execution of interest has been taken. In a non-transactional setting, it is well known how to convert executions into litmus tests [Alglave et al. 2010; Wickerson et al. 2017]. We illustrate the process here via a worked example. In §3.1 we extend this process to the transactional setting.

The execution given in Fig. 2 corresponds to the pseudocode litmus test given in Fig. 3. Read events have become loads, writes have become stores, and the *po* edges have induced the order of instructions and their partitioning into threads.

Forcing rf. To ensure that the litmus test passes only when the intended *rf* edges are present, the postcondition checks that each local register holds the value written by the store it was intended to observe. In this example, the *rf* edge from *a* to *c* is checked by the `r0 == 1` in the postcondition.

Forcing co. Ensuring that the intended *co* edges are present is a little more subtle. We can employ the litmus test's postcondition to check the final value of each memory location, and thereby fix all the *co*-maximal writes. But when there are more than two writes to a location, this constraint

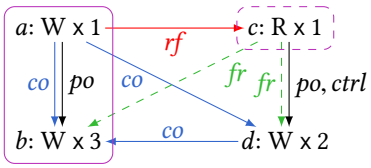


Fig. 4. A sample execution with transactions

Initially: x = 0, ok = 1	
txbegin	txbegin
store(x, 1)	r1 = load(x)
store(x, 3)	if (r1 == 1) txabort(42)
txend	txabort(1)
goto Succ	Fail: r2 = txstatus
Fail: ok = 0	store(x, 2)
Succ:	
Test: ok == 1 && r2 == 42 && x == 3	

Fig. 5. A litmus test for the execution in Fig. 4

is not enough to fix its entire *co*-chain. Therefore, we impose that every *co* edge in an execution must be 'forced'. (Wickerson et al. use the term 'dead'.) In our example, the *co* edge from *a* to *d* is forced because *a* is observed by *c*, which precedes *d* in program order. An execution that had the edge the other way round is guaranteed to violate Coherence, and thus be inconsistent in all the memory models we study in this paper. Wickerson et al. provide more general constraints that capture exactly when a *co* edge is forced.

3 AXIOMATISING TRANSACTIONS

Transactional memory is a concurrent programming model that allows the programmer to group sequences of instructions into a *transaction*, whose execution should appear instantaneous to an outside observer [Herlihy and Moss 1993]. Some important attributes of transactions are that they are *atomic*, which means that each transaction either executes in its entirety or not at all, and that they are *isolated* from the interference from other transactions (and, optionally, other non-transactional code). TM can be provided either at the architecture level or in software. In this paper, since we aim only to specify TM and not to concern ourselves with its implementation, we are able to formalise both forms of TM within a unifying framework.

In order to extend axiomatic memory modelling techniques to support transactions, we extend the form of executions with two additional relations,

- *stxn*, which relates events in the same successful (committed) transaction, and
- *ftxn*, which relates events in the same failed (aborted) transaction.

Both of these are partial equivalence relations (i.e., symmetric and transitive). We assume that each *stxn*-class and *ftxn*-class coincides with a contiguous subset of *po*. Diagrammatically, we represent *stxn*-classes using solid boxes and *ftxn*-classes using dashed boxes. For instance, the execution in Fig. 4 comprises one successful transaction (events *a* and *b*) and one failed transaction (event *c*).

Limitations. Because we assume events in the same transaction to be contiguous in *po*, our formalisation does not currently handle pausing and resuming transactions (as provided by, for instance, Power's `tsuspend` and `tresume` instructions). Moreover, because we represent each transaction simply by an equivalence class, we cannot model nested transactions in their full generality [Harris et al. 2010]. However, we can handle flat nesting (where nested transactions are treated simply as part of their outermost parent transaction), as used by both TSX [Intel 2017, p. 386] and Power [IBM 2015, p. 882]. In this case, there is no need to distinguish the child from the parent since both will either commit successfully or fail together. Nesting in C++ is more subtle because there are two kinds of transactions that can be alternated [Adl-Tabatabai et al. 2012].

3.1 Generating litmus tests from transactional executions

Following on from §2.3, we now explain how to convert an execution of interest into a litmus test in the presence of transactions, using the execution in Fig. 4 as an example.

The litmus test corresponding to this execution is shown in Fig. 5. Successful transactions (*stxn*-classes) are straightforward: the instructions in the transaction simply need enclosing in instructions that begin and end a transaction. We write these as `txbegin` and `txend` in our pseudocode; our tool specialises these appropriately for each target architecture. The postcondition needs to check that the transaction succeeded, which we arrange by setting a register in the transaction's fail-handler. Specifically, in our pseudocode, we assume that an aborting transaction jumps to the `Fail` label; therefore checking that `ok` is still 1 in the postcondition implies that the transaction succeeded.

Failing transactions require more infrastructure. When a programmer aborts their transaction, it is usually possible to provide a reason for this. We support this in our pseudocode through the `txabort` instruction's parameter. In Fig. 5, the transaction aborts with code 42 only if all the reads observed their intended values; otherwise, it aborts with code 1. In the fail-handler, we assume that this abort code is made available in a `txstatus` register, which we copy into a fresh register (so that `txstatus` can be reused for any subsequent transactions), and then check in the postcondition.

There are a few subtleties for handling dependency edges in the presence of failed transactions. First, although we are able to construct a litmus test that only passes when the failed transaction reads specific values, we are not able to construct address or data dependencies from these reads. This is because the registers written inside the failed transaction are reset by the `txabort` instruction, thus breaking any dependency chain. For this reason, we require that in all the executions we generate, no *addr* or *data* edges leave failed transactions. Control dependencies that leave failed transactions, however, can be constructed in the litmus tests; indeed, because the choice of `txabort` instruction is conditional on all the values read by the transaction, we automatically have control dependencies from every read in a failed transaction to every event after the transaction. For this reason, we require that all executions we generate satisfy

$$[R]\,;\mathrm{outof}(\mathit{ftxn}) \quad \subseteq \quad \mathit{ctrl}$$

where

$$\begin{aligned} \mathrm{outof}(r) &= (r \cap \mathit{id})\,;\neg r \\ \mathrm{into}(r) &= \neg r\,;(r \cap \mathit{id}) \end{aligned}$$

Given a partial equivalence relation r, $\mathrm{into}(r)$ relates e_1 to e_2 if e_2 is in the domain of r but e_1 and e_2 are not in the same r-class, and $\mathrm{outof}(r)$ relates e_1 to e_2 if e_1 is in the domain of r but e_1 and e_2 are not in the same r-class.

3.2 Weak and strong isolation

In this subsection, we explain how the *isolation* property of transactions can be captured as a property of an execution graph.

Blundell et al. [2006] identify two forms of isolation: weak and strong. (They use the term 'atomicity', but we follow Harris et al. [2010] and use 'isolation'.) In essence, a TM system provides *weak* isolation if transactions are isolated from other transactions; that is, their intermediate state cannot affect or be affected by other transactions. It provides *strong* isolation if, in addition, transactions are isolated from non-transactional code.

The four 3-event SC executions in Fig. 6 illustrate the difference between strong and weak isolation. The first and last of these correspond to what Blundell et al. call *non-interference* and *containment*, respectively.

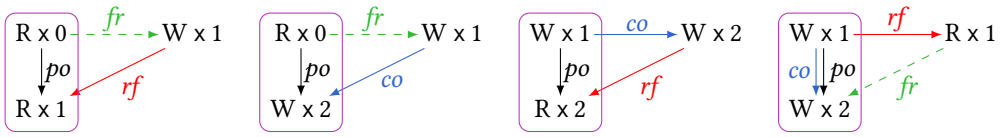


Fig. 6. SC executions that are allowed by weak isolation but forbidden by strong isolation

Failures of isolation can be characterised as cycles in communication edges (rf, co, and fr) between transactions. To define these cycles, the following construction is useful for lifting a relation between events to a relation between transactions:

$$\text{weaklift}(r,t) \quad = \quad t;(r \setminus t);t.$$

Specifically, if events e_1 and e_2 are in different (successful) transactions and are related by r, then every event in e_1's transaction will be related to every event in e_2's transaction by the relation $\text{weaklift}(r, stxn)$. We can extend this construction to include non-transactional events:

$$\text{stronglift}(r,t) \quad = \quad \text{weaklift}(r, t \cup [\neg\text{domain}(t)]).$$

Here, if events e_1 and e_2 are related by r and not in the same (successful) transaction, then $\text{stronglift}(r, stxn)$ will also relate e_1 (and any other events in the same transaction) to e_2 (and any other events in the same transaction). The stronglift construction is analogous to placing each non-transactional event in a singleton transaction and then invoking weaklift.

Weak and strong isolation can then be captured by the following axioms.

$$\textbf{acyclic}(\text{weaklift}(com, stxn)) \qquad \text{(WeakIsolation)}$$

$$\textbf{acyclic}(\text{stronglift}(com, stxn)) \qquad \text{(StrongIsolation)}$$

A cycle in com between two transactions (or, in the case of strong isolation, between transactional and non-transactional events) indicates that the intermediate state of a transaction is either affecting or being affected by events outside of that transaction: a failure of isolation.

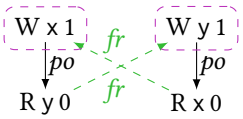
3.3 Restrictions on failed transactions

Reading from failed transactions. We allow write events inside failed transactions, but we do not allow these writes to be observed outside of their transaction; they are assumed to be deleted when the transaction aborts. Therefore, all of our memory models include the axiom

$$\textbf{empty}(rf \cap \text{outof}(ftxn)) \qquad \text{(AbortRead)}$$

which deems an execution inconsistent if a reads-from edge leaves a failed transaction. Nonetheless, we remark that some 'eager versioning' implementations of software TM allow violations of the AbortRead axiom [Shpeisman et al. 2007].

Reading before failed transactions. We have already forbidden reads-from edges to *leave* failed transactions; from-read (fr) edges that *enter* failed transactions are problematic too. To see this, consider the following store-buffering execution where both writes are inside failed transactions.



(3)

Notwithstanding the apparent cycle in $po \cup fr$, any reasonable memory model must allow this execution, because any reasonable machine can perform the writes to x and y, then abort the transactions (which cancels the writes), and then read the original zeroes from both locations. To

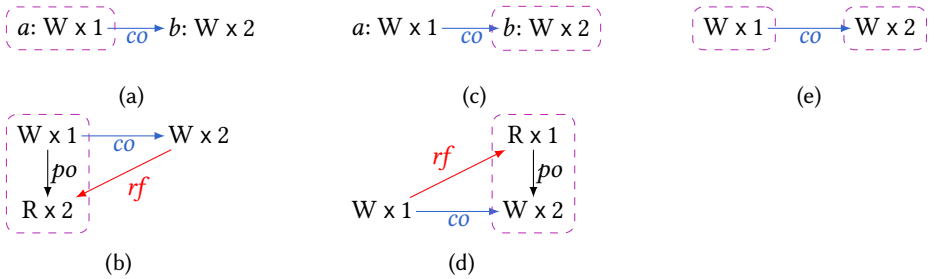
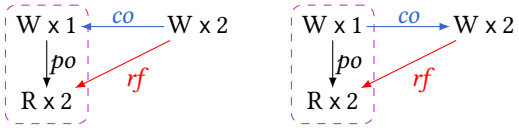


Fig. 7. Coherence order and failed transactions

ensure that executions like this are always allowed, we redefine the *fr* relation in each transactional memory model so that it does not enter failed transactions. We do this by subtracting into(*ftxn*).

Failed transactions and coherence order. Conventionally, the coherence order (*co*) connects all of the writes to the same location in an execution. The situation is complicated when some of these writes are in failed transactions, because these writes are not durable outside of their transaction.

One could remove failed transactional writes from executions altogether, but this is unsatisfactory because it is reasonable for a transaction to write to a location, read it, and then abort; without the write event there can be no justification for the read. One could remove *co* edges from failed transactional writes, but this is also unsatisfactory, because when there are multiple writes inside a failed transaction, we rely on the *co* edges between them to resolve their order. One could just remove *co* edges that enter or exit a failed transaction, but even this is unsatisfactory, as shown by the following pair of executions, which differ only by the direction of their *co* edge.



In the left execution, the read observes a stale value. This behaviour, which is a violation of Coherence, is forbidden by almost all memory models. In the right execution, the read observes a concurrent write that occurs after the beginning of the transaction. This behaviour violates strong isolation, but a memory model that only offers weak isolation may wish to allow it. Therefore we must allow *co* edges to enter and exit failed transactions.

In summary: we leave the definition of *co* unchanged in the presence of failed transactions.

Forcing coherence order. Although we allow *co* edges to travel into and out of failed transactions, some of these edges are not forceable (as defined in §2.3). First, consider a *co* edge that leaves a failed transaction (Fig. 7a). We cannot simply check `x == 2` in the postcondition of the generated litmus test – this condition will hold even if event b occurred first, because event a is invisible by the time the postcondition is evaluated. Nonetheless, if a is followed by a read that observes x at 2 (Fig. 7b) then we *can* deduce that *co* contains (a, b), because (b, a) would violate Coherence.

Second, consider a *co* edge that enters a failed transaction (Fig. 7c). As before, we cannot force this edge via the postcondition `x == 2`. However, if b is preceded by a read that observes x at 1 (Fig. 7d), then we can deduce that $(a, b) \in co$, because, once again, (b, a) would violate Coherence.

Third and finally, consider a *co* edge that connects two failed transactions (Fig. 7e). These *co* edges cannot be forced: since both writes fail, each is invisible to the other's transaction.

In summary: to ensure that all the executions we consider are forceable, we restrict our attention to executions where:

Execution $(E, R, W, po, addr, ctrl, data, stxn, ftxn, sloc, rf, co)$ is consistent iff:	
SeqCst:	$\mathbf{acyclic}(po \cup com)$
	where $fr = (rf^{-1}; co \cup [R \setminus \mathrm{range}(rf)]; sloc; [W]) \setminus \mathrm{into}(ftxn)$
	$com = rf \cup co \cup fr \cup co; rf$
TxnOrder:	$\mathbf{acyclic}(\mathrm{stronglift}(po \cup com, stxn \cup ftxn))$
AbortRead:	$\mathbf{empty}(rf \cap \mathrm{outof}(ftxn))$

Fig. 8. Consistency axioms for SC [Shasha and Snir 1988], with extensions for TSC highlighted

Table 1. Using an execution X to validate a model M against an implementation

	X empirically observable	X not empirically observable
X allowed by M	ok	M may be incomplete (consider refining)
X not allowed by M	M is unsound (refine)	ok

$$(ftxn \cap id); \mathrm{imm}(co_e); \neg(ftxn \cap id) \subseteq ftxn; rf^{-1} \quad (4)$$

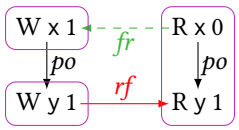
$$\neg(ftxn \cap id); \mathrm{imm}(co_e); (ftxn \cap id) \subseteq rf; ftxn \quad (5)$$

$$(ftxn \cap id); \mathrm{imm}(co_e); (ftxn \cap id) \subseteq \emptyset \quad (6)$$

3.4 Transactional sequential consistency

Transactional sequential consistency (TSC) is due to Dalessandro and Scott [2009]. It is a stronger form of SC in which events that are consecutive within a transaction must appear consecutively in the overall execution order. Operationally, TSC can be understood as a guarantee that while one thread is inside a transaction, all other threads are blocked from making any progress. Axiomatically, where SC is characterised by forbidding $(po \cup com)$-cycles, TSC can be characterised by additionally forbidding $(po \cup com)$-cycles between transactions and non-transactional events (see Fig. 8).

TSC is strictly stronger than the combination of strong isolation and SC. For instance, it forbids the following behaviour that is allowed by both strong isolation and SC.



(7)

TSC can be seen as the strongest (reasonable) model of TM. As such, it provides for us a useful 'upper bound' when comparing transactional memory models.

Note that we use $stxn \cup ftxn$ in the TxnOrder axiom (rather than just $stxn$) in order to constrain failing transactions as well as successful ones. This is because we intend TSC as an upper bound model. It also matches the operational model of TSC: a failing transaction would lock the machine in the same way as a successful transaction would.

4 METHODOLOGY

We use the following methodology to develop the memory models presented in this paper.

(1) We make a first attempt at a model, informed by specifications, programming manuals, research papers, and discussions with designers.
(2) For each proposed strengthening or weakening of the model, we use the Memalloy tool [Wickerson et al. 2017] to generate tests that become disallowed or allowed as a result of the change, and use the results of running these tests on existing hardware (where available) using the Litmus tool, and discussions with designers, to decide whether to accept the refinement.
(3) To validate that the model is not too strong, we use the extended version of Memalloy (described in §4.1) to generate all the tests (up to a bound) that the model disallows but are allowed by the non-transactional baseline model. That is, we search for executions in the set

$$\mathit{consistent}(\mathit{baseline}) \setminus \mathit{consistent}(M)$$

and convert each to a litmus test. We run each test on existing hardware (where available). If any of these tests are empirically observable, the model is too strong (i.e. unsound), and we must refine it (see Table 1).
(4) To validate that the model is not too weak, we use the same methodology to generate all the tests (up to a bound) that the model allows but are disallowed by TSC. That is, we search for executions in the set

$$\mathit{consistent}(M) \setminus \mathit{consistent}(\mathit{TSC})$$

and convert each to a litmus test, which we run on existing hardware (where available). If any of these tests cannot be empirically observed, the model may be unnecessarily weak, and we should consider refining it (see Table 1).

4.1 Extending the Memalloy tool

The Memalloy tool as presented by Wickerson et al. takes *two* memory models (say, M and N) and generates a *single* execution that distinguishes them (i.e., is inconsistent under M but consistent under N). That is, it calculates

$$\mathit{distinguishing}(M,N) \quad = \quad \text{choose}\{X \in \mathbb{X} \mid X \in \mathit{consistent}(N) \setminus \mathit{consistent}(M)\}$$

where choose(S) picks an arbitrary element from a set S. On the other hand, the litmus test generator of Lustig et al. [2017] takes a *single* memory model and generates *all* (minimal) executions that are inconsistent under M. That is, it calculates

$$\mathit{conformance\text{-}suite}(M) \quad = \quad \min\{X \in \mathbb{X} \mid X \notin \mathit{consistent}(M)\}.$$

where $\min(S)$ is the $\sqsubset$-minimal subset of a set S of executions, and $X \sqsubset Y$ holds when execution X can be obtained from execution Y by:

- removing an event (plus any incident edges),
- removing a dependency edge (i.e., *addr*, *ctrl*, *data*) or a fence edge (e.g. *mfence* in the x86 model), or
- downgrading a fence (e.g. replacing a *sync* fence with an *lwsync* fence in the Power model) or an event (e.g. replacing an *SC* write with a *Rel* write in the C++ model).

In this work, we have developed a new tool that combines Lustig et al.'s technique for finding minimal executions and Memalloy's ability to compare models. This combination is necessary because although Lustig et al.'s tool is useful for testing the *soundness* of a model – i.e., if any of the tests that it generates can be observed on hardware, then we can deem the model unsound (or the hardware defective) – it is not useful for testing the *completeness* of a model; i.e., checking that the behaviours a model allows are indeed empirically observable. One might expect that calculating

$$\mathit{conformance\text{-}suite}(\mathit{SC}) \setminus \mathit{conformance\text{-}suite}(M)$$

would suffice for generating all the non-SC executions that a model M allows, but this set is not actually useful, because *conformance-suite*(SC) does not contain enough executions. For instance, *conformance-suite*(SC) will not contain any executions with dependency edges; such dependencies can always be removed because the SC model is dependency-agnostic. The problem is one of premature minimisation. It can be solved by first generating distinguishing executions and *then* minimising; that is, by calculating the following set:

$$\textit{distinguishing-suite}(M,N) \quad = \quad \min\{X \in \mathbb{X} \mid X \in \textit{consistent}(M) \setminus \textit{consistent}(N)\}.$$

Our new tool is an extension of Memalloy that calculates this set.

Extending minimality to handle transactions. We have also extended the definition of minimality to be sensitive to transactions. To do this, we have extended the $\sqsubset$ relation so that $X \sqsubset Y$ also holds when X can be obtained from Y by making a transactional event non-transactional (i.e. removing all of its incident *stxn* or *ftxn* edges). This is a slightly coarse notion of minimality – a more refined version would also allow a large transaction to be chopped into two smaller ones – but it is cheap to implement in the constraint solver because we need only to quantify over a single transactional event. As a result of this coarseness, Memalloy may generate some executions that do not appear minimal. This slight over-generation is not problematic.

4.2 Extending the Litmus tool

We have extended the Litmus tool [Alglave et al. 2011] to handle transactional tests by adding support for instructions for TM and querying the `txstatus` register. This was straightforward as there are only minor differences between the TM interfaces of x86 and Power.

5 TRANSACTIONS IN X86

The Intel TSX specification gives two interfaces for using TM on the x86 architecture: Restricted Transactional Memory (RTM)[1] and Hardware Lock Elision (HLE). In our formalisation of TSX, we focus on RTM. This provides new instructions (`XBEGIN`, `XEND` and `XABORT`) that enable software to explicitly start, commit, and abort transactions.

5.1 Background: the x86 memory model

The x86 memory model follows the SPARC Total Store Ordering (TSO) model [Owens et al. 2009]. Informally, all threads share a total order over all stores, except for a relaxation allowing store-buffering (store-to-load reordering). More formally, executions for x86 are extended to include the relations:

- *rmw*, which relates successful atomic read-modify-write events (e.g., `CMPXCHG` or instructions with a `LOCK` prefix), and
- *mfence*, which relates events separated by a memory fence (`MFENCE` instruction).

An x86 execution is consistent if it satisfies the Coherence, AtomicRMW and Order axioms of Fig. 9. The Coherence and AtomicRMW axioms are standard for enforcing coherence and the atomicity of read-modify-writes, respectively. The happens-before relation (*hb*) represents a partial order over events on which all threads must agree; the relation must be acyclic as required by Order. The *hb* relation is induced by fences: both explicit (*mfence*) due to fence instructions and implicit (*implied*) due to `LOCK`'d instructions; preserved program order (*ppo*), which allows the relaxation for store-buffering; inter-thread observation (rf_e) and communication (*fr* and *co*).

[1] 'Restricted' since not all instructions can be used within a transaction

x86 execution $(E, R, W, po, addr, ctrl, data, rmw, stxn, ftxn, sloc, mfence, rf, co)$ is consistent iff:

COHERENCE:	$\mathbf{acyclic}(po_{\text{loc}} \cup com)$
	where $fr = (rf^{-1}; co \cup [R \setminus \text{range}(rf)]; sloc; [W]) \setminus \text{into}(ftxn)$
	$com = rf \cup co \cup fr \cup co; rf$
ORDER:	$\mathbf{acyclic}(hb)$
	where $ppo = ((W \times W) \cup (R \times W) \cup (R \times R)) \cap po$
	$txbegin = po \cap \text{into}(stxn)$
	$txend = po \cap \text{outof}(stxn)$
	$L = \text{domain}(rmw) \cup \text{range}(rmw)$
	$implied = [L]; po \cup po; [L] \cup txbegin \cup txend$
	$hb = mfence \cup ppo \cup implied \cup rf_{\text{e}} \cup fr \cup co$
ATOMICRMW:	$\mathbf{empty}(rmw \cap (fr_{\text{e}}; co_{\text{e}}))$
STRONGISOLATION:	$\mathbf{acyclic}(\text{stronglift}(com, stxn))$
TXNORDER:	$\mathbf{acyclic}(\text{stronglift}(hb, stxn))$
ABORTREAD:	$\mathbf{empty}(rf \cap \text{outof}(ftxn))$
ATOMICFTXN:	$\mathbf{empty}(ftxn \cap ((fr_{\text{e}} \cup co_{\text{e}}); rf_{\text{e}}))$

Fig. 9. x86 consistency axioms [Alglave et al. 2014, adapted], with our transactional additions highlighted

5.2 Adding transactions

Strong isolation. The Intel manual specifying TSX does not explicitly state that transactions are strongly isolating. However, we interpret that this is the case due to the definition of transaction conflicts between a transaction and a 'conflicting access', which is not restricted to being transactional itself [Intel 2017, p. 384]. In other words, non-transactional accesses can cause transaction conflicts. Hence, our first addition is to add our axiom for strong isolation, STRONGISOLATION.

Implicit transaction fences. Additionally, the Intel manual gives `LOCK`'d semantics to successful transactions: 'a successfully committed [transaction] … has the same ordering semantics as a `LOCK` prefixed instruction' [Intel 2017, p. 387]. This justifies the additional *implied* clauses, *txbegin* and *txend*, for entry into and out of successful transactions. Note that we do not add similar constraints for failing transactions since the specification does not give fence semantics to the `XBEGIN` instruction itself, but rather to a successful transaction delimited by an `XBEGIN`/`XEND` pair.

Lifting hb. The additions given so far are insufficient to forbid executions involving cycles in *hb* between transactions. For example, consider the following transactional variant of the message-passing (**MP**) litmus test on the left below. The chain $fr; ppo; rf$ is not in com^+, but is in hb^+. Since happens-before gives ordering requirements that all threads must agree on, a cycle like this implies a problem with transaction serialisation. Memalloy finds 26 distinguishing 4-event executions (including this one), none of which were observed in our empirical testing.

(8)

Based on this intuition and the empirical unobservability of these executions, we strengthen our model to include the axiom TxnOrder to disallow happens-before cycles between successful transactions. Note that we do not disallow happens-before cycles between the union of successful and failing transactions ($stxn \cup ftxn$) since this would disallow a failing variant of the execution where we replace the successful transaction with a failing one (shown on the right above). We allow this execution because the TSX specification gives no explicit ordering requirements for failing transactions. This said, our empirical testing did not find any witnesses of this weak behaviour.

Failing transactions. For failing transactions we include the AbortRead axiom defined in §3.3. This forbids a read observing a write of a failing transaction (except for reads *within* the failing transaction itself). This is justified by the atomicity (all-or-nothing) guarantee of TSX: 'On a transactional abort, the processor will discard all updates performed' [Intel 2017, p. 383].

Our final addition is to give some atomicity guarantees to failing transactions via the AtomicFtxn axiom, which is similar in spirit to the standard AtomicRMW axiom for read-modify-writes. The principle we observe is that strongly isolating transactions must also give atomicity guarantees to failing transactions since a machine implementing TM cannot know during execution whether a transaction will ultimately succeed or not. For example, if a transaction reads the same location twice without performing an intermediate write, strong isolation ensures that the reads observe the same write. Similarly, strong isolation guarantees that if a transaction writes to a location, this write will be observed by subsequent reads. The AtomicFtxn axiom formalises this intuition by forbidding the following two shapes:

(9)

5.3 Empirical testing

Table 2 gives the results obtained using our testing strategy (§4). We first generate distinguishing litmus tests up to a bounded number of events ($|E|$). We report solve times running Memalloy on a 4-core Haswell i7-4771 machine with 32GB RAM using a timeout of 3600 seconds (1 hour). Comparison of our transactional x86 model against the baseline (non-transactional) x86 model and the TSC model give the 'Forbid' and 'Allow' sets of tests, respectively. For each set we give the number of tests (T) found by Memalloy; this number is non-exhaustive when solving reached timeout. Secondly, we run each test 1 000 000 times on two TSX implementations: a Haswell (i7-4771)[2] and a Broadwell-Mobile (i7-5650U).[3] Since our table merges observations from two implementations we say a test is seen (S) if it is observed on either implementation and not seen (¬S), otherwise.

The results give confidence that our model is sound up to 5-event executions (exhaustively) and 10-event executions (partially). There are no tests forbidden by our model that are empirically observable on the TSX implementations that we tested. The 'Allow' set of tests shows that our model permits several liberties (282 tests), but very few are observable empirically (30 tests). We examined the set of 30 observed tests and note that the majority (24 tests) have positive witnesses rarely (fewer than 10 out of 1 000 000 runs). Comparing results between implementations we found that the Broadwell-Mobile machine was responsible for the majority (23 tests) of observations. This result is unexpected since Broadwell has the same microarchitecture as Haswell (the main difference is process node generation). We plan to extend our testing campaign to include further TSX implementations to explore these issues further. For both 'Forbid' and 'Allow' sets we see that

[2]https://ark.intel.com/products/77656/Intel-Core-i7-4771-Processor-8M-Cache-up-to-3_90-GHz
[3]https://ark.intel.com/products/84995/Intel-Core-i7-5650U-Processor-4M-Cache-up-to-3_20-GHz

Table 2. Empirical testing of our transactional x86 model on Intel Haswell and Broadwell machines

$\|E\|$	**Forbid**				**Allow**			
	Solve (Sec)	T	S	¬S	Solve (Sec)	T	S	¬S
2	1	2	0	2	1	0	0	0
3	2	6	0	6	1	0	0	0
4	6	26	0	26	3	6	2	4
5	191	45	0	45	47	10	4	6
6	3600∗	167	0	167	3600∗	38	16	22
7	3600∗	372	0	372	3600∗	79	4	75
8	3600∗	514	0	514	3600∗	124	4	120
9	3600∗	37	0	37	3600∗	21	0	21
10	3600∗	9	0	9	3600∗	4	0	4
Sum		1178	0	1178		282	30	252

the solve time switches to timeout from 5 to 6 events. There are exponentially more solutions as Memalloy searches for larger tests, but beyond 8 events we see that the time taken to find each individual solution starts to dominate.

6 TRANSACTIONS IN POWER

The Power architecture has provided TM support since version 2.07 [Cain et al. 2013; IBM 2013]. It provides `tbegin`, `tend`, and `tabort` instructions for starting, committing, and aborting transactions. Further instructions for pausing and resuming transactions are not considered in this work. Power also provides 'rollback-only' transactions that are not atomic and can only access thread-local data; we do not consider these either.

6.1 Background: the Power memory model

The Power memory model is distinguished from those of the x86 and ARMv8 architectures in that it is not *multicopy-atomic* [Collier 1992]: writes can propagate to *some* threads before they propagate to *all* threads. For this reason, the memory model must include explicit axioms to handle the propagation of writes between threads.

There exist several formalisations of the Power memory model, surveyed by Alglave et al. [2014]. Power executions have the following additional relations:

- *rmw*, which relates successful atomic read-modify-write events (generated by *exclusive* instructions, e.g. `lwarx` and `stwcx`),
- *isync*, which relates events separated by an instruction-synchronisation fence,
- *lwsync*, which relates events separated by (at least) a lightweight-synchronisation fence, and
- *sync*, which relates events separated by a full synchronisation fence.

Since full fences are stronger than their lightweight counterparts, we have $sync \subseteq lwsync$.

A Power execution is consistent if it satisfies all of the axioms in Fig. 10. The happens-before relation (*hb*) is formed from inter-thread observations (rf_e), the preserved fragment of the program order (*ppo*), and lightweight/full fences (*fence*). We do not give the full definition of *ppo* as it is rather complex [Alglave et al. 2014], but it includes at least: address and data dependencies, control dependencies to writes, and reads ordered before writes to the same location. The happens-before relation must have no cycles (Order). The *prop* relation represents the order in which writes are

Power execution $(E, R, W, po, addr, ctrl, data, rmw, stxn, ftxn, sloc, isync, lwsync, sync, rf, co)$ is consistent iff:

Coherence:	$\mathbf{acyclic}(po_{\text{loc}} \cup com)$
	where $fr = (rf^{-1}; co \cup [R \setminus \text{range}(rf)]; sloc; [W]) \setminus \text{into}(ftxn)$
	$com = rf \cup co \cup fr \cup co; rf$
Order:	$\mathbf{acyclic}(hb)$
	where $ppo = \ldots$
	$txbegin = po \cap \text{into}(stxn)$
	$txend = po \cap \text{outof}(stxn)$
	$fence = sync \cup (lwsync \setminus (W \times R)) \cup txbegin \cup txend$
	$hb = rf_{\text{e}}^{?}; (ppo \cup fence); rf_{\text{e}}^{?}$
Propagation:	$\mathbf{acyclic}(co \cup prop)$
	where $prop_1 = [W]; rf_{\text{e}}^{?}; fence; rf_{\text{e}}^{?}; hb^{*}; [W]$
	$prop_2 = com_{\text{e}}^{*}; (rf_{\text{e}}^{?}; fence; rf_{\text{e}}^{?})^{*}; hb^{*}; sync; hb^{*}$
	$txprop = com_{\text{e}}; stxn; hb^{*}$
	$prop = prop_1 \cup prop_2 \cup txprop$
Observation:	$\mathbf{irreflexive}(fr_{\text{e}}; prop; hb^{*})$
ExclusiveWrites:	$\mathbf{acyclic}(([\text{range}(rmw)]; po; [\text{range}(rmw)]) \cup co)$
AtomicRMW:	$\mathbf{empty}(rmw \cap (fr_{\text{e}}; co_{\text{e}}))$
StrongIsolation:	$\mathbf{acyclic}(\text{stronglift}(com, stxn))$
TxnPropagation:	$\mathbf{acyclic}(\text{stronglift}(co \cup prop, stxn))$
AbortRead:	$\mathbf{empty}(rf \cap \text{outof}(ftxn))$
AtomicFtxn:	$\mathbf{empty}(ftxn \cap ((fr_{\text{e}} \cup co_{\text{e}}); rf_{\text{e}}))$

Fig. 10. Power consistency axioms [Alglave et al. 2014, adapted], with our transactional additions highlighted

propagated to other threads; this relation must not contradict the coherence order (Propagation). The Observation axiom governs which writes a read can observe. An exclusive write must not overwrite another exclusive write that is sequenced later in the same thread (ExclusiveWrites). Finally, read-modify-write pairs must be atomic with respect to other threads (AtomicRMW).

6.2 Adding transactions

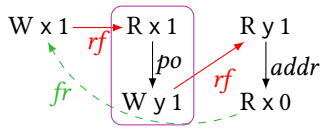
Read-modify-writes. The Power manual states that when a read-modify-write operation (i.e., a pair of `lwarx`/`stwcx` instructions) is split across a transaction boundary, the operation will always fail [IBM 2015, p. 823]. Therefore, we add the assumption that every *rmw* edge is either wholly within a transaction or wholly outside.

Isolation. The Power manual states that transactions 'appear atomic with respect to both transactional and non-transactional accesses performed by other threads' [IBM 2015, p. 882], so we add our axiom for strong isolation, StrongIsolation.

Barriers around transactions. The manual also states that 'a `tbegin` instruction that begins a successful transaction creates a memory barrier', as does 'a `tend` instruction that ends a successful

transaction' [IBM 2015, p. 824–5]. Accordingly, we add to the *fence* relation all the program-order edges that enter or exit a successful transaction.

Barriers within transactions. Transactions also contain an 'integrated memory barrier', which ensures that writes observed by a successful transaction are propagated to other threads before writes performed by the transaction itself [IBM 2015, p. 825]. This behaviour is epitomised by the **WRC**-style litmus test given by Cain et al. [2013, Fig. 6], whose execution of interest is as follows.



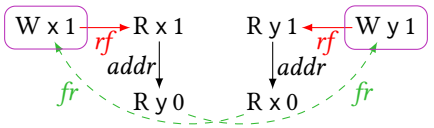
(10)

To rule out this execution, it suffices to add the following chains of edges

$$rf_e ; stxn ; [W] \tag{11}$$

into the *prop* relation. The expression comprises events pairs (e_1, e_2) where e_1 and e_2 are writes in different threads, and e_1 is observed by a read in e_2's transaction.

Transaction ordering. The Power manual further states that 'successful transactions are serialised in some order', and that it is impossible for a thread to observe contradictions to this order [IBM 2015, p. 824]. This behaviour is epitomised by a variant of the **IRIW** litmus test given by Cain et al. [2013, Fig. 5], whose execution of interest is reproduced below.



(12)

It would be possible to interpret this part of the manual literally and extend our model with an explicit total order over all successful transactions, but this would make the model computationally expensive to simulate and reason about (see §8.2.2 for more discussion of this). Instead, we seek to disallow behaviours in which different threads observe incompatible transaction orders.

The existing Power memory model disallows (non-transactional) **IRIW** executions when each pair of reads is separated by a *sync*; it does this by inferring a *prop* cycle between the two reads that observe the initial values, and then relying on the Propagation axiom to forbid such cycles. Since Cain et al.'s execution is also based on **IRIW**, it seems appropriate to rely on a similar mechanism to forbid it. Therefore, we add the following chains of edges

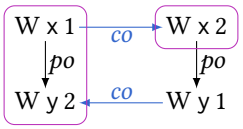
$$fr_e ; stxn ; rf_e ; ppo \tag{13}$$

into the *prop* relation. This is sufficient to rule out execution (12). We remark that an execution similar to (12) in which only one of the writes is transactional *is* observable on Power, and our model duly allows it.

Failing transactions. To handle failing transactions, we include the standard AbortRead and AtomicFtxn axioms.

Refining the axioms. A comparison of TSC against the model we have assembled so far reveals several executions that are allowed by our model but could not be observed empirically. Among

these is the following execution of the **2+2W** litmus test:



(14)

Since the tend instruction in the second thread gives rise to a *prop* edge between that thread's writes, the execution exhibits a cycle in $co \cup prop$ among transactions and non-transactional events. We therefore add a lifted variant of the existing Propagation axiom, which we call TxnPropagation, to rule out this execution.

Further empirically-unobservable executions that our model needs to forbid are variants of the classic **SB** (left) and **R** (right) executions:

(15)

These executions can be ruled out in the same way that we ruled out (12): in each case we can induce a *prop* cycle between the two non-transactional accesses. To do this, we add to *prop* the following chains of edges

$$com_e ; stxn ; hb^*$$

which generalises both (11) and (13).

6.3 Empirical testing

Table 3 gives our testing results. We generate tests that distinguish our transactional Power model from the baseline (non-transactional) Power model and from the TSC model. Each test is run 10 000 000 times on an 80 core POWER8 (TN71-BP012) machine. We use the 'affinity' parameter provided by Litmus in order to place threads incrementally across the logical processors, which is necessary for stimulating **IRIW**-style behaviours.

The results give confidence that our model is sound up to 5-event executions (exhaustively) and 9-event executions (partially). There are no tests forbidden by our model that are empirically observable on the Power machine that we tested. In contrast to our x86 testing campaign (§5.3), most of the behaviours that our transactional Power model allows *could* be observed empirically. Many of the unobserved-but-allowed tests are based on the load-buffering (**LB**) shape, which has never been observed on any existing Power machine, even without transactions.[4]

7 TRANSACTIONS IN ARMV8

The ARMv8 architecture does not currently include TM support. Therefore, the extensions proposed in this section are unofficial and purely academic. Nonetheless, we have followed similar principles for adding TM that we have used for x86 and Power, and the decisions we made are distilled from conversations with ARM architects.

7.1 Background: the ARMv8 memory model

The ARMv8 memory model is a middle ground between the strength of x86 and the weakness of Power. The model was simplified and formalised in the transition from v7 to v8 [ARM 2017], the principal change being to strengthen the memory model to be multicopy-atomic.

ARMv8 executions have the following additional fields:

[4] http://moscova.inria.fr/~maranget/cats/model-power/all.html#sec4

Table 3. Empirical testing of our transactional Power model on a POWER8 (TN71-BP012) machine

$\|E\|$	**Forbid**				**Allow**			
	Solve (Sec)	T	S	¬S	Solve (Sec)	T	S	¬S
2	4	2	0	2	3	0	0	0
3	7	6	0	6	4	0	0	0
4	92	84	0	84	19	17	13	4
5	3600∗	399	0	399	666	24	20	4
6	3600∗	395	0	395	3600∗	86	71	15
7	3600∗	32	0	32	3600∗	32	31	1
8	3600∗	16	0	16	3600∗	28	28	0
9	3600∗	0	0	0	3600∗	1	1	0
Sum		934	0	934		188	164	24

ARMv8 execution $(E, R, W, Acq, Rel, po, addr, ctrl, data, rmw, stxn, ftxn, sloc, dmb, dmbld, dmbst, isb, rf, co)$ is consistent iff:

$$
\begin{aligned}
\textsc{Coherence}:\quad & \mathbf{acyclic}(po_{\mathrm{loc}} \cup com)\\
& \quad \text{where } fr = (rf^{-1}; co \cup [R \setminus \mathrm{range}(rf)]; sloc; [W]) \setminus \mathrm{into}(ftxn)\\
& \qquad com = rf \cup co \cup fr \cup co; rf\\
\textsc{Order}:\quad & \mathbf{acyclic}(ob)\\
& \quad \text{where } dob = addr \cup data \cup ctrl; [W] \cup (ctrl \cap isb); [R] \cup addr; isb; [R] \cup\\
& \qquad addr; po; [W] \cup (ctrl \cup data); co_{\mathrm{i}} \cup (addr \cup data); rf_{\mathrm{i}}\\
& \qquad aob = rmw; rf_{\mathrm{i}}^{?}\\
& \qquad txbegin = po \cap \mathrm{into}(stxn)\\
& \qquad txend = po \cap \mathrm{outof}(stxn)\\
& \qquad bob = dmb \cup [Rel]; po; [Acq] \cup [R]; dmbld \cup [Acq]; po \cup\\
& \qquad\quad [W]; dmbst; [W] \cup po; [Rel]; co_{\mathrm{i}}^{?} \cup txbegin \cup txend\\
& \qquad ob = com_{\mathrm{e}} \cup dob \cup aob \cup bob\\
\textsc{AtomicRMW}:\quad & \mathbf{empty}(rmw \cap (fr_{\mathrm{e}}; co_{\mathrm{e}}))\\
\textsc{StrongIsolation}:\quad & \mathbf{acyclic}(\mathrm{stronglift}(com, stxn))\\
\textsc{TxnOrder}:\quad & \mathbf{acyclic}(\mathrm{stronglift}(ob, stxn))\\
\textsc{AbortRead}:\quad & \mathbf{empty}(rf \cap \mathrm{outof}(ftxn))\\
\textsc{AtomicFtxn}:\quad & \mathbf{empty}(ftxn \cap ((fr_{\mathrm{e}} \cup co_{\mathrm{e}}); rf_{\mathrm{e}}))
\end{aligned}
$$

Fig. 11. ARMv8 consistency axioms [ARM 2017; Deacon 2017, adapted], with our transactional extensions highlighted

- *Acq* and *Rel*, the sets of acquire-read and release-write events (generated by `LDAR` and `STLR` instructions, respectively),
- *rmw*, which relates successful atomic read-modify-write events (generated by *exclusive* instructions, e.g. `LDXR` and `STXR`),

- *dmb*, *dmbld*, and *dmbst*, which relate events separated by data-memory barriers (DMB, DMBLD, and DMBST instructions), and
- *isb*, which relates events separated by instruction-synchronising barriers (ISB instructions).

Since full DMB barriers are stronger than DMBLD and DMBST barriers, we have $dmb \subseteq dmbld$ and $dmb \subseteq dmbst$.

An ARMv8 execution is consistent if it satisfies the Coherence, AtomicRMW and Order axioms of Fig. 11. The Coherence and AtomicRMW axioms are standard. The ordered-before relation *ob* is a partial order over events on which all threads must agree (Order). The *ob* relation is induced by communication (com_e), dependencies (*dob*), atomic read-modify writes (*aob*), and barriers (*bob*).

7.2 Adding transactions

We propose a transactional ARMv8 model based on our experience defining the transactional models of x86 and Power.

Read-modify-writes. ARMv8 has exclusive instructions, which we treat similarly to those of Power: we do not consider executions that split a read-modify-write operation across a transaction boundary.

Isolation. The StrongIsolation axiom is a natural choice for hardware TM.

Barriers around transactions. Similarly to x86 and Power, we define an implicit fence for successful transactions (*txbegin* and *txend*), but not for failing transactions.

Lifting ob. Since *ob* in ARMv8 plays a similar role to *hb* in x86, we forbid executions with cycles in *ob* between successful transactions (TxnOrder) since this would imply a problem with transaction serialisation.

Failing transactions. Finally, we add the AbortRead and AtomicFtxn for failing transactions since these are also common to x86 and Power.

7.3 Distinguishing executions

Although we cannot perform empirical testing, we can still search for distinguishing executions (Table 4). Comparing the number of tests generated for each bound against our x86 and Power results shows a higher number of 'Forbid' tests. We attribute this to the larger number of (equivalent) mechanisms in the ARMv8 model for strengthening an execution. For example, an address dependency *addr* between a read and write can equally be replaced with a data dependence *data*, a barrier *dmbld*, or by strengthening the read (resp., write) into an acquire (resp., release). Therefore, for each execution involving an address dependency Memalloy will also find the same execution with each of these variants.

8 TRANSACTIONS IN C++

We now turn our attention from hardware TM to software TM. Although these two types of TM are implemented in very different ways, we are able to handle both within our axiomatic framework because we concern ourselves only with the *specification* of TM, not its implementation.

TM is supported in C++ via a draft technical specification that has been under development by the C++ TM Study Group since 2012 [ISO/IEC 2015] based on an early proposal by Shpeisman et al. [2009]. In this section, we investigate and formalise how the proposed TM extensions interact with the existing C++ memory model, and detail a possible simplification to the specification. The C++ TM Study Group has listed 'conflict with the C++ memory model and atomics' as one of the hardest challenges of adding TM support to C++ [Wong 2014]; our work addresses precisely this challenge.

Table 4. Distinguishing execution solve times for our transactional ARMv8 model

$\|E\|$	**Forbid**		**Allow**	
	Solve (Sec)	T	Solve (Sec)	T
3	3	6	2	0
4	99	129	18	17
5	3264	542	775	24
6	3600∗	1457	3600∗	89
7	3600∗	816	3600∗	107
8	3600∗	68	3600∗	34
9	3600∗	1	3600∗	0
10	3600∗	4	3600∗	0
Sum		3023		271

The transactional C++ extension offers two types of transactions. *Relaxed* transactions are written using a `synchronized{...}` block. They can contain arbitrary code, but are only guaranteed to isolated from other transactions and cannot be aborted. *Atomic* transactions are written using an `atomic{...}` block. They are guaranteed to be isolated from any other code and can be aborted (by raising a `tx_exception`), but they cannot contain side-effecting code or atomic operations.

8.1 Background: the C++ memory model

In C++, every memory location is deemed either atomic or non-atomic [Batty et al. 2016, 2011]. We shall write `a`, `b`, `c` etc. for non-atomic locations, and `x`, `y`, `z`, etc. for atomic locations. Furthermore, every memory operation is deemed either atomic or non-atomic. Atomic reads and writes never access non-atomic locations, and non-atomic reads never access atomic locations. In C++, a restricted coherence order is used (called co'), which only connects writes that access *atomic* locations.

Atomic operations are further divided into acquire atomics, release atomics, and SC atomics, so events in a C++ execution belong to zero or more of the following sets:

- A, the atomic events,
- Acq, the atomic events with 'acquire' semantics (i.e., with `memory_order_acquire` or stronger),
- Rel, the atomic events with 'release' semantics (i.e., with `memory_order_release` or stronger),
- SC, the atomic events with SC semantics (i.e., with `memory_order_seq_cst`), and
- $NAL = \{e \mid loc(e) \text{ is a non-atomic location}\}$, the events that access a non-atomic location.

Unlike the architecture-level memory models we have considered so far in this paper, the C++ memory model defines *two* predicates on executions (Fig. 12). The first characterises the *consistent* candidate executions. If any consistent execution violates a second *race-freedom* predicate, then any execution is allowed – the program's semantics is completely undefined. Otherwise, the allowed executions are the consistent executions.

A C++ execution is consistent if it satisfies all of the consistency axioms given at the top of Fig. 12. The first of these, HbCom, governs the happens-before relation, which in C++ is constructed from the program order and release/acquire synchronisation between threads (sw_e). The definition of sw is complex as it includes cases to handle fences and the 'release sequence'; it is explained more fully by Batty et al. [2016]. The NARead axiom states that any read from a non-atomic location must observe the most recent (in happens-before) write. Finally, SeqCst forbids certain cycles among SC events. This axiom incorporates a refinement due to Lahav et al. [2017] that weakens the model

C++ execution $(E, R, W, F, NAL, A, Acq, Rel, SC, po, addr, ctrl, data, stxn, ftxn, sloc, rf, co)$ is consistent iff:

HbCom: $\mathbf{acyclic}(hb_{\text{loc}} \cup com)$

where $co' = co \setminus NAL^2$

$fr = (rf^{-1}; co' \cup [R \setminus \text{range}(rf)]; sloc; [W]) \setminus id \setminus \text{into}(ftxn)$

$com = rf \cup co' \cup fr \cup co'; rf$

$sw = [Rel]; ([F]; po)^?; [W \cap A]; po_{\text{loc}}{}^*; rf^+; [R \cap A]; (po; [F])^?; [Acq]$

$tsw = \text{weaklift}(com, stxn \cup ftxn)$

$hb = (sw_{\text{e}} \cup tsw \cup po)^+$

NARead: $rf; [NAL] \subseteq \text{imm}([W]; hb_{\text{loc}})$

where $hb = rf_{\text{e}}{}^?; (ppo \cup lwsync); rf_{\text{e}}{}^?$

SeqCst: $\mathbf{acyclic}((([F]; po)^?; (po \cup (po; hb; po) \cup hb_{\text{loc}} \cup co' \cup fr); (po; [F])^?) \cap SC^2)$

AbortRead: $\mathbf{empty}(rf \cap \text{outof}(ftxn))$

C++ execution $(E, R, W, F, NAL, A, Acq, Rel, SC, po, addr, ctrl, data, stxn, ftxn, sloc, rf, co)$ is race-free iff:

NoDataRace: $\mathbf{empty}(conflict_{\text{e}} \setminus A^2 \setminus (hb \cup hb^{-1}))$

where $conflict = ((W \times W) \cup (R \times W) \cup (W \times R)) \cap sloc \setminus id$

Fig. 12. C++ consistency and race-freedom axioms [Batty et al. 2016; Lahav et al. 2017, adapted], with our transactional additions highlighted and deletions ~~struck through~~

slightly so that it can compile correctly to the Power architecture. Although this refinement is unofficial, we include it here so that we can usefully check compilation to Power in the transactional case (see §8.3).

A consistent C++ execution is race-free if it satisfies the NoDataRace axiom at the bottom of Fig. 12, which states that whenever two events in different threads are in conflict and are not both atomic, they must be ordered by happens-before. Two events are said to be in conflict if they both access the same location and at least one is a write.

8.2 Adding transactions to the C++ memory model

The draft specification for C++ TM involves two amendments to the C++ memory model.

First, it clarifies that although events in an aborted transaction cannot be observed, they can still participate in data races. For this reason, it is important that we have included failing transactions (*ftxn*) in our formalisation. Without these, our model would be unsound. It is not necessary to change the existing definition of a data race.

The second amendment is to define when transactions synchronise with one another. To this end, the draft specification states that an execution is consistent only if there exists a total order over all of its transactions such that:

(Requirement 1) this order does not contradict the happens-before relation, and

(Requirement 2) if a transaction T_1 is ordered before a conflicting transaction T_2, then the end of T_1 synchronises with the start of T_2 [ISO/IEC 2015, §1.10].

8.2.1 Initial formalisation. We can straightforwardly incorporate these requirements into the formal model by taking the following three steps.

(1) We can extend the form of executions with a transaction-ordering relation, *to*. This relation forms a total order among all of the *stxn*-equivalence classes and *ftxn*-equivalence classes in an execution.
(2) To capture **Requirement 2**, we can add a relation for capturing when a transaction synchronises with (*tsw*) another transaction, and update the definition of happens-before to include this new relation:

$$tsw = \text{weaklift}(conflict, stxn \cup ftxn) \cap to \tag{16}$$
$$hb = (sw_e \cup po \cup tsw)^+. \tag{17}$$

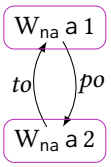
The *tsw* relation contains the pair (e_1, e_2) if e_1 is in one transaction, e_2 is in another transaction that is ordered after the first, and the transactions conflict.[5]
(3) To capture **Requirement 1**, we can add a new axiom to the consistency predicate that prevents *to* from contradicting happens-before:

$$\textbf{irreflexive}(to; hb) \tag{TxnHb}$$

8.2.2 Criticism of initial formalisation. This formulation presented above is unsatisfying for two reasons. First, it is awkward that the transaction order is used to *define* happens-before but is also forbidden to *contradict* happens-before. It is important that the transaction order contributes to the *definition* of happens-before because it is the only mechanism through which executions such as the following one can be deemed race-free.

$$\boxed{\mathrm{W_{na}\ a\ 2}} \qquad \boxed{\mathrm{W_{na}\ a\ 1}} \tag{18}$$

Although the execution has two non-atomic writes in different threads to the same non-atomic location, it is race-free because once the order between the transactions is resolved by the *to* relation, the writes will be ordered by happens-before. It is also important that the transaction order is forbidden to *contradict* happens-before because it is the mechanism that rejects nonsensical executions such as the following one, in which the transaction order violates the program order.



The second reason why this formulation is unsatisfying is the presence of the total order over transactions. For an execution to be deemed consistent, it is necessary to show the existence of a suitable *to*. It therefore becomes difficult to show that an execution is *inconsistent*, because one must iterate over all possible instantiations of *to* (which corresponds to finding all permutations of an execution's transactions). Simulation tools such as Herd have been shown to experience significant slowdown in similar circumstances [Batty et al. 2016]. Moreover, having quantification over relations within the consistency predicate makes it harder for Memalloy to compare the model against other models, since it must switch from plain Alloy to the (slower) AlloyStar solver that can handle higher-order quantification [Milicevic et al. 2015].

[5]The specification technically demands that the *end* of the first transaction synchronises with the *start* of the second, but we instead have *all* events in the first transaction synchronising with *all* events in the second. This is simpler, and is equivalent because happens-before is closed under program order.

8.2.3 Revised formalisation. Fortunately, it is possible to address both of these concerns by formulating the transactional C++ memory model without relying on a total order over transactions. The basic idea is to observe that if two transactions are in conflict, then (in almost all cases) the transaction order can be deduced from the existing *rf*, *co*, and *fr* edges, and if they are not in conflict, then there is no need to impose a total order between them at all.

In more detail: first observe that whenever two events in an execution are in conflict, they must be linked one way or the other by *com*, where we define $com = rf \cup co \cup fr \cup co;rf$, as we did in (2). That is, $conflict \subseteq com \cup com^{-1}$. Second, recall from §8.1 that C++ uses a restricted coherence order (co') that only relates writes on *atomic* locations. For instance, in execution (18) above, there is no co' edge between the writes because a is a non-atomic location. As such, we have no way to resolve the order between the two transactions.

We can resolve this problem by modifying the C++ memory model so that the coherence order relates writes to *all* locations, rather than just those to atomic locations. We used Memalloy to compare our modified model against the original, and were able to confirm that the models are equivalent on all executions with fewer than eight events.[6]

It is now the case that whenever two transactions are in conflict, they will be connected one way or the other by *com*. We can use the direction of this *com* edge to resolve the order of those transactions. Specifically, let us say that a transaction synchronises with (*tsw*) another when they are connected by *com*, and then extend happens-before to include *tsw*, as follows.

$$tsw = \text{weaklift}(com, stxn \cup ftxn) \tag{19}$$

$$hb = (sw_{\text{e}} \cup tsw \cup po)^{+} \tag{20}$$

By simply extending the definition of *hb* like this, we avoid the need for the *to* relation altogether, and we avoid adding any axioms (like TxnHb) to the consistency predicate.

To make our proposal more concrete, our companion material includes some text that the draft specification could incorporate, which is currently being reviewed by the C++ TM Study Group.

8.2.4 Isolation for atomic transactions. Thus far, we have only given C++ transactions a weakly-isolating semantics. This ensures the desired behaviour for relaxed transactions, but what about atomic transactions, which are required to be strongly-isolating? The draft specification addresses this problem simply by requiring that atomic transactions do not contain atomic operations. The idea is that in order for a non-transactional event to observe or affect the intermediate state of a transaction, it must be in conflict with an event in that transaction. Since that event is required to be non-atomic, we must have a data race. Thus, for race-free programs, atomic transactions are guaranteed – in a vacuous sense – to be strongly-isolating.

To obtain confidence in this claimed property of the memory model, we used Memalloy to search for C++ executions that are consistent and race-free under the transactional C++ memory model but are forbidden under strong isolation. It confirmed that there are no such executions with fewer than eight events.

8.2.5 Transactional data-race freedom. A central property of the C++ memory model is its SC-DRF guarantee [Adve and Hill 1990; Boehm and Adve 2008]: if a C++ program is free from data races and avoids non-SC atomic operations, then it only has SC semantics. This guarantee can be

[6]Note that dropping the restriction on *co* in this manner is not a new idea. Both Vafeiadis et al. [2015] and Batty et al. [2015] have investigated dropping the distinction between non-atomic and atomic locations altogether, which entails an all-locations coherence order. However, unifying all locations like this also entails observable changes to the model – Vafeiadis et al.'s proposal violates an existing compiler optimisation and Batty et al. show that it introduces out-of-thin-air problems. In this paper, we propose *only* to extend the set of events over which the coherence order is defined, leaving the non-atomic/atomic distinction intact. This does not result in an observable change to the model.

lifted to a transactional setting to give what might be called a TSC-TDRF guarantee [Dalessandro and Scott 2009; Shpeisman et al. 2009]: if a C++ program is free from data races, avoids non-SC atomic operations, and does not have atomic operations inside transactions, then it only has TSC semantics.

We used Memalloy to search for executions that violate this guarantee. It confirmed that there are no such executions with fewer than six events.

8.3 Mapping C++ transactions to x86, Power, and ARMv8

One way to validate the models we have presented in this paper is to check the soundness of compilation processes that map C++ transactions into x86, Power, and ARMv8 transactions. A more realistic compiler would be considerably more complex – perhaps relying on an implementation of software TM as a fallback option should a hardware transaction fail – but our simple and direct mapping is nonetheless useful to compare the guarantees given to transactions at the software and the hardware levels. We only consider the compilation of C++'s *atomic* transactions, because its *relaxed* transactions cannot be aborted (and hardware transactions may abort).

Specifically, we use Memalloy to search for a pair of executions, X and Y, such that X is a C++ execution that is *forbidden* by the C++ consistency axioms, Y is an x86, Power, or ARMv8 execution that is *allowed* by the respective architecture-level consistency axioms, and X is related to Y via the appropriate compilation mapping. Such a pair would demonstrate that the compilation mapping is invalid. Wickerson et al. [2017] have encoded non-transactional compilation mappings, so it remains for us to extend these to handle transactions. We do this by requiring the mapping relation (which we call π) to preserve all *stxn* and *ftxn* edges (in both directions); that is:

$$\begin{aligned}(Y.stxn) &= \pi^{-1};(X.stxn);\pi\\ (Y.ftxn) &= \pi^{-1};(X.ftxn);\pi\end{aligned}$$

where $X.stxn$ indicates the *stxn* relation in execution X, and so on.

Memalloy confirmed that no C++ execution with fewer than six events can miscompile to x86, Power, or ARMv8.

9 RELATED WORK

Several authors have investigated the interplay between weak memory and TM, both in the context of formalising their semantics (§9.1) and testing or verifying TM implementations (§9.2).

9.1 The semantics of TM in the presence of weak memory

Grossman et al. [2006] identify several tricky corner cases that arise when attempting to extend Java's weak memory model to handle transactions. Shpeisman et al. [2007] identify further corner cases associated with weak isolation, including some that involve failed transactions. Our methodology can be seen as a way to automate the generation of corner cases like these.

Maessen and Arvind [2007] propose a framework for axiomatising weak memory models in the presence of transactions, based on a must-not-reorder function between pairs of instructions. Dalessandro et al. [2010] propose a memory modelling framework in which transactions are the primary unit and other constructions (such as locks and atomic operations) are defined in terms of transactions. Unlike our work, neither of these modelling frameworks are experimentally validated.

Other TM correctness criteria include 'opacity', which requires that successful transactions are serialisable and that failed transactions observe a consistent state [Doherty et al. 2012; Guerraoui and Kapalka 2008]. The original definition of opacity is unsuitable for our setting as it does not allow for non-transactional code, but it has been extended so that it can be parameterised by a memory model for the non-transactional fragment [Guerraoui et al. 2010].

Cerone et al. [2015] have studied the weak consistency guarantees provided by transactions in the context of database systems. A key difference compared to our work is that the weak behaviours we observe can be attributed not to weakly-consistent transactions, but to the weakly-consistent non-transactional events that surround strongly-consistent transactions. Moreover, Cerone et al. assume that all events are in transactions. Nonetheless, similar axiomisations can be used in both settings, and similar weak behaviours can manifest. For instance, Cerone et al. [2017, Theorem 11] define serialisability by forbidding all cycles in weaklift(com, $stxn$). We used Memalloy to confirm that all ARM, x86, Power, and C++ executions with fewer than seven events, with all events in successful transactions, are (unsurprisingly) serialisable. More interesting results could be obtained by extending Cerone et al.'s axiomatisations to include non-transactional events and then checking whether our models satisfy weaker properties like snapshot isolation or parallel snapshot isolation.

9.2 Testing and verifying TM in the presence of weak memory

The closest work to ours is by Manovit et al. [2006], who present an automatic tool for testing implementations of software TM above a weak memory model. Like us, they use automatically-generated litmus tests to probe the implementations, but where our test suite is constructed to be exhaustive and to contain only 'interesting' tests, their tests are randomly generated. Manovit et al. also assume that no memory location is accessed both transactionally and non-transactionally (which makes their framework inadequate for testing hardware TM), and they ignore failed transactions.

Going beyond testing, Guerraoui et al. [2009] have verified implementations of software TM on several weak memory models. Regarding the verification of programs that use transactions, Kuru et al. [2014] show how to verify a C program that uses weakly consistent transactions by transforming it into an equivalent program that includes an explicit encoding of the consistency model, and then using a standard verification tool.

9.3 Tool support for memory model development

As noted in §4, our methodology for developing our memory models builds on tools due to Wickerson et al. [2017] and Lustig et al. [2017], both of which use Alloy [Jackson 2012] as a backend. An alternative workflow has been recently proposed by Bornholt and Torlak [2017], who introduce the MemSynth tool for synthesising memory models from a corpus of litmus tests and their expected outcomes. A comparison of these two methodologies would be interesting, once MemSynth is extended to support software-level memory models (and control dependencies, which it currently does not handle). MemSynth is based on a SAT-solving backend that is more programmable than Alloy; this brings both the advantage of faster solving times and the disadvantage of higher complexity for the user.

The Diy tool of Alglave et al. [2010] has been widely used to generate litmus tests by enumerating relaxations of SC. Our tool has the key advantage of being easily extensible to custom constructs such as transactions. Moreover, Diy has been known to miss some tests that distinguish models (because it relies on the user to supply relaxations [Wickerson et al. 2017]), and also to generate more tests than may be needed because, unlike our tool and those listed above, it is not sensitive to particular memory model.

10 CONCLUSION

In this paper we have extended existing axiomatic memory models for x86, Power, ARMv8 and C++ to account for transactional memory. Using our extensions to Memalloy, we synthesised meaningful sets of litmus tests that precisely capture the subtle interactions between weak memory and transactions. These tests allowed us to validate our new models by (1) running them on hardware where available, (2) discussions with engineers, and (3) checking against technical specifications.

Finally, we have shown (up to a bound) that our model for C++ transactions compiles naturally to our architectural models.

10.1 Future directions

There exist several opportunities for future work.

First, when constructing litmus tests for failing transactions, we only generate litmus tests in which transactions deliberately self-abort, rather than failing because of conflicts. Is it possible to construct litmus tests in which transactions fail naturally (e.g. because of a conflict) and a different mechanism leaks the values they observed (e.g. via their updates to the page table)? And is it possible to probe more behaviours of an implementation by extending the form of litmus tests to include special instructions that trigger microarchitectural events [Reid 2016], such as transaction conflicts?

Second, we have only considered axiomatic memory models in this paper, but there exist operational accounts of the memory models for x86 [Owens et al. 2009], Power [Sarkar et al. 2011], ARMv8 [Flur et al. 2016], and C++ [Nienhuis et al. 2016]. Can these operational models be extended to handle transactions, and how do they compare against our axiomatic models?

Third, Wickerson et al. [2017] have expressed several compiler optimisations as Alloy constraints and validated them against weak memory models. Can common transaction-related transformations, such as hardware lock elision and transaction chopping [Shasha et al. 1995], be cast into this form, and then validated against the memory models presented in this paper?

Fourth, the transactional C++ memory model formalised in §8 remains unsatisfactory because of the rather artificial restriction that atomic transactions must not contain atomic operations. How should the axioms of the model be amended to allow this restriction to be lifted?

REFERENCES

Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, and Justin Gottschlich. 2012. Original Draft Specification of Transactional Language Constructs for C++ Version 1.1. (February 2012). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3725.pdf

Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering - A New Definition. In *Int. Symp. on Computer Architecture (ISCA)*. https://doi.org/10.1145/325096.325100

Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU concurrency: weak behaviours and programming assumptions. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/2694344.2694391

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-642-14295-6_25

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests Against Hardware. In *Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. https://doi.org/10.1007/978-3-642-19835-9_5

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: modelling, simulation, testing, and data-mining for weak memory. *ACM Trans. on Programming Languages and Systems (TOPLAS)* 36, 2 (2014). https://doi.org/10.1145/2627752

ARM. 2017. *ARMv8 Architecture Reference Manual*. https://static.docs.arm.com/ddi0487/b/DDI0487B_a_armv8_arm.pdf

Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. In *ACM Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2914770.2837637

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Europ. Symp. on Programming (ESOP)*. https://doi.org/10.1007/978-3-662-46669-8_12

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *ACM Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1926385.1926394

Colin Blundell, E. C. Lewis, and Milo M. K. Martin. 2006. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters* 5, 2 (2006). https://doi.org/10.1109/L-CA.2006.18

Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1379022.1375591

James Bornholt and Emina Torlak. 2017. Synthesizing Memory Models from Framework Sketches and Litmus Tests. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3062341.3062353

Harold W. Cain, Brad Frey, Derek Williams, Maged M. Michael, Cathy May, and Hung Le. 2013. Robust Architectural Support for Transactional Memory in the Power Architecture. In *Int. Symp. on Computer Architecture (ISCA)*. https://doi.org/10.1145/2485922.2485942

Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *Int. Conf. on Concurrency Theory (CONCUR)*. https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2017. Algebraic Laws for Weak Consistency. In *Int. Conf. on Concurrency Theory (CONCUR)*. https://arxiv.org/abs/1702.06028

William W. Collier. 1992. *Reasoning about Parallel Architectures*. Prentice Hall.

Luke Dalessandro and Michael L. Scott. 2009. Strong Isolation is a Weak Idea. In *ACM Workshop on Transactional Computing (TRANSACT)*. http://transact09.cs.washington.edu/33_paper.pdf

Luke Dalessandro, Michael L. Scott, and Michael F. Spear. 2010. Transactions as the Foundation of a Memory Consistency Model. In *Int. Conf. on Distributed Computing (DISC)*. https://doi.org/10.1007/978-3-642-15763-9_4

Will Deacon. 2017. Update aarch64.cat to align with the ARMv8 memory model. https://github.com/herd/herdtools7/commit/daa126680b6ecba97ba47b3e05bbaa51a89f27b7. (2017).

Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2012. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing* 25, 5 (2012). https://doi.org/10.1007/s00165-012-0225-8

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *ACM Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2837614.2837615

Dan Grossman, Jeremy Manson, and William Pugh. 2006. What Do High-Level Memory Models Mean for Transactions?. In *ACM Workshop on Memory Systems Performance & Correctness (MSPC)*. https://doi.org/10.1145/1178597.1178609

Rachid Guerraoui, Thomas A. Henzinger, Michal Kapalka, and Vasu Singh. 2010. Transactions in the Jungle. In *Symp. on Parallelism in Algorithms and Architectures (SPAA)*. https://doi.org/10.1145/1810479.1810529

Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. 2009. Software Transactional Memory on Relaxed Memory Models. In *Int. Conf. on Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-642-02658-4_26

Rachid Guerraoui and Michal Kapalka. 2008. On the Correctness of Transactional Memory. In *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/1345206.1345233

Mark Hachman. 2014. Intel finds specialized TSX enterprise bug on Haswell, Broadwell CPUs. *PCWorld* (August 2014). http://www.pcworld.com/article/2464880

Tim Harris, James Larus, and Ravi Rajwar. 2010. *Transactional Memory* (2nd ed.). Morgan & Claypool. https://doi.org/10.2200/S00272ED1V01Y201006CAC011

Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Int. Symp. on Computer Architecture (ISCA)*. https://doi.org/10.1145/173682.165164

IBM. 2013. *Power ISA (Version 2.07)*.

IBM. 2015. *Power ISA (Version 3.0)*.

Intel. 2017. 6th Generation Intel Processor Family: Specification Update. (June 2017). https://www3.intel.com/content/dam/www/public/us/en/documents/specification-updates/desktop-6th-gen-core-family-spec-update.pdf

Intel. 2017. *Intel 64 and IA-32 Architectures: Software Developer's Manual*. https://software.intel.com/en-us/articles/intel-sdm

Intel Developer Zone. 2012. Transactional Synchronization in Haswell. (February 2012). https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell

ISO/IEC. 2015. *Technical Specification for C++ Extensions for Transactional Memory*. Draft technical specification. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf

Daniel Jackson. 2012. *Software Abstractions – Logic, Language, and Analysis* (revised ed.). MIT Press.

Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/2976749.2978321

Ismail Kuru, Burcu Kulahcioglu Ozkan, Suha Orhun Mutluergil, Serdar Tasiran, Tayfun Elmas, and Ernie Cohen. 2014. Verifying Programs under Snapshot Isolation and Similar Relaxed Consistency Models. In *ACM Workshop on Transactional Computing (TRANSACT)*. http://transact2014.cse.lehigh.edu/kuru.pdf

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3062341.3062352

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979). https://doi.org/10.1109/TC.1979.1675439

Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In *Int. Conf. on Architectural Support for Programming Languages and*

Operating Systems (ASPLOS). https://doi.org/10.1145/3037697.3037723

Jan-Willem Maessen and Arvind. 2007. Store Atomicity for Transactional Memory. *Electronic Notes in Theoretical Computer Science* 174, 9 (2007). https://doi.org/10.1016/j.entcs.2007.04.009

Chaiyasit Manovit, Sudheendra Hangal, Hassan Chafi, Austen McDonald, Christos Kozyrakis, and Kunle Olukotun. 2006. Testing Implementations of Transactional Memory. In *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*. https://doi.org/10.1145/1152154.1152177

Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. 2015. Alloy*: A General-Purpose Higher-Order Relational Constraint Solver. In *Int. Conf. on Software Engineering (ICSE)*. https://doi.org/10.1007/s10703-016-0267-2

Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An Operational Semantics for C/C++11 Concurrency. In *ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/3022671.2983997

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics (TPHOLs)*. https://doi.org/10.1007/978-3-642-03359-9_27

Alastair Reid. 2016. Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture. In *Formal Methods in Computer-Aided Design (FMCAD)*. https://doi.org/10.1109/FMCAD.2016.7886675

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1993498.1993520

Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. on Database Systems* 20, 3 (1995). https://doi.org/10.1145/211414.211427

Dennis Shasha and Marc Snir. 1988. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. on Programming Languages and Systems (TOPLAS)* 10, 2 (1988). https://doi.org/10.1145/42190.42277

Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang Ni, and Adam Welc. 2009. Towards Transactional Memory Semantics for C++. In *Symp. on Parallelism in Algorithms and Architectures (SPAA)*. https://doi.org/10.1145/1583991.1584012

Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. 2007. Enforcing Isolation and Ordering in STM. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1273442.1250744

Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2017. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3037697.3037719

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *ACM Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2775051.2676995

John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *ACM Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3009837.3009838

Michael Wong. 2014. Transactional Language Constructs for C++. In *C++ Conference (CppCon)*. http://bit.ly/2tWk4uz

A PROPOSED AMENDMENT TO THE TRANSACTIONAL C++ SPECIFICATION

A.1 Original text

The original text is as follows [ISO/IEC 2015, §1.10]:

(1) There is a global total order of execution for all outer blocks. If, in that total order, T_1 is ordered before T_2,
- no evaluation in T_2 happens before any evaluation in T_1 and
- if T_1 and T_2 perform conflicting expression evaluations, then the end of T_1 synchronizes with the start of T_2.

A.2 Proposed text

To accommodate the proposal from §8 of our paper, we propose the following replacement text:

(1) An operation A *communicates to* a memory operation B on the same object M if:
- A and B are both side effects and A precedes B in the modification order of M;
- A is a side effect, B is a value computation, and the value computed by B is the value stored either by A or by another side effect C that follows A in the modification order of M; or
- A is a value computation, B is a side effect, and B follows in the modification order of M the side effect that stored the value computed by A.

(2) The end of outer transaction T_1 synchronises with the start of outer transaction T_2 if an operation in T_1 communicates to an operation in T_2.