On Atomicity in Presence of Non-atomic Writes*

Constantin Enea[1] and Azadeh Farzan[2]

[1] Univ. Paris Diderot, cenea@liafa.univ-paris-diderot.fr
[2] University of Toronto, azadeh@cs.toronto.edu

Abstract. The inherently nondeterministic semantics of concurrent programs is the root of many programming errors. Atomicity (more precisely conflict serializability) has been used to reduce the magnitude of this nondeterminism and therefore make it easier to understand the behaviour of the concurrent program. Serializability, however, has not been studied well for programs executed under memory models weaker than sequential consistency (SC), where writes are not atomic, i.e., they may be committed to the main memory later than issued. In this paper, we define the notion of conflict serializability for the Total Store Ordering (TSO) memory model, and study the relation between TSO-serializability and the well-known notions of SC-serializability and robustness. We investigate the algorithmic problem of monitoring program executions for violations of serializability, and provide lower bound complexity results for the problem, and new algorithms to perform the monitoring efficiently.

1 Introduction

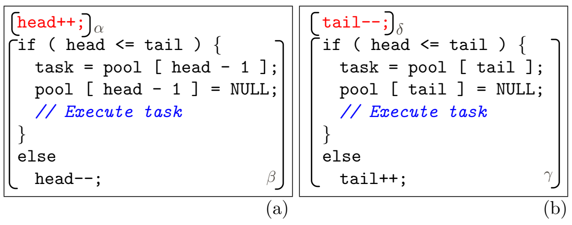
While writing a concurrent program, a programmer often prefers to have non-interfered access to shared data that is manipulated by a thread, since this permits the reasoning about the correctness of the code to be done locally and therefore simplifies the process. *Atomicity* is a *generic* correctness criterion that is inspired by this view. Informally, an *atomic* code block has the same behaviour under interfering actions of other threads as it does when executed without interference (serially). Establishing atomicity of code blocks eases the task of reasoning about the program by substantially reducing the number of interleavings that need to be considered. Moreover, non-atomicity hints at the existence of potential bugs; a study of concurrency errors [20] shows that a majority of reported errors in concurrent programs (around 69%) are atomicity violations.

Several notions of atomicity have been introduced in the literature. A widely recognized notion is *conflict serializability* [21], introduced as a correctness criterion with a tractable monitoring algorithm that guarantees *atomicity*. It is assumed that a program's code is divided into code blocks (such as procedures, loop bodies, or even single statements) that are called *transactions*. An execution is conflict serializable if it is *equivalent* to a serial execution, i.e. an execution in which all transactions are executed in a sequential non-interleaved fashion. The key element of this definition is the notion of *equivalence* which allows permutation of non-conflicting statements to establish an equivalent serial execution.

* An extended version of this paper including the missing proofs can be found at [1].

There has been a huge body of research in the recent years that studies the problems of static and dynamic checking of atomicity, which is almost entirely based on the assumption that the programs are executed under a sequentially consistent (SC) memory model. Weak memory models have been duly getting a lot of attention in the programming languages and systems research communities, and yet the question of atomicity under a weak memory model has not been studied well. Let us start by an example to motivate why weak memory models require a carefully tailored notion of atomicity.

Consider the program with two methods in Fig. 1. Array pool implements a pool of tasks with two pointers head and `tail` pointing to its beginning and end. The invariant is head $\leq$ tail and the pool is empty if head = tail. The procedures (a) and (b) take elements from the pool's head and tail, respectively. Imagine a program that is running these two procedures in two threads (transactions are marked by brackets in the figure). Once a thread atomically modifies `head/tail`, interference from the other thread is tolerated. But, when it is about to modify the `pool`, it requires mutual exclusion. It is easy to verify that every execution of this program is conflict serializable (under sequential consistency). Even though both (a) and (b) potentially write to an element of the array pool, the conditional ensures that it is never the same element. Now consider the same program executed under the Total Store Order (TSO) memory model where writes are first stored in a thread-local buffer and non-deterministically flushed into the shared memory at a later time. When head + 1 = tail, the `if` condition may succeed in both (a) and (b). A write to head performed by (a) may be propagated to (b) after the condition is tested in (b), conversely a write to tail performed by (b) may be propagated to (a) after the condition is tested in (a). This is the behaviour that is strictly disallowed under SC. In that case, both threads access the same element of the array pool by first reading it and then writing to it. This is a classic violation of atomicity. Moreover, assuming that the threads are grabbing tasks from this task pool to execute, this non-atomic behaviour can lead to a real program error if a non-idempotent task ends up being executed twice by two different threads. We need a notion of atomicity that is aware of such erroneous TSO-executions, and declares them as non-atomic.



```
[head++;]α
[if ( head <= tail ) {
   task = pool [ head - 1 ];
   pool [ head - 1 ] = NULL;
   // Execute task
 }
 else
   head--;                 ]β
```
(a)

```
[tail--;]δ
[if ( head <= tail ) {
   task = pool [ tail ];
   pool [ tail ] = NULL;
   // Execute task
 }
 else
   tail++;                 ]γ
```
(b)

Fig. 1. Task pool (transactions marked by brackets).

In this paper, we propose a new notion of atomicity, called *TSO-serializability*, which is inspired by the standard notion of conflict serializability under SC, in the sense that is syntactic, efficient to monitor, and helpful for the programmer to facilitate local reasoning. Yet, it makes special considerations for (i) non-atomicity of writes to the shared memory under TSO, and (ii) possible reorderings of shared memory accesses made by the same thread, allowed under TSO but not under SC. The idea is that TSO-serializability lifts the *relaxedness* of the orderings of individual statements under TSO to the level of atomic blocks (viewed as composite statements). For example, since TSO allows for two statements $\mathit{write}(x)\mathit{read}(y)$ to be reordered to $\mathit{read}(y)\ \mathit{write}(x)$ (indicating that the write is committed later), therefore we expect the two-transaction sequence $[write(x_1)write(x_2)]\ [read(y_1)read(y_2)]$ to be allowed to be reordered to $[read(y_1)read(y_2)]\ [write(x_1)write(x_2)]$ in an *equivalent* execution.

We provide a formal justification for the notion of TSO-serializability presented in this paper by stating its precise relation to SC-serializability and *robustness*. Robustness [6] is a property of a program stating that the program does not exhibit non-SC behaviour if executed on a weaker memory model such as TSO. If a program is robust, and it is SC-serializable, then for any reasonable notion of TSO-serializability, one should expect it to be serializable under TSO. That is exactly what we prove for our proposed notion of TSO-serializability. The converse, however, does not always hold. If a program exhibits strictly more behaviours under TSO (compared to SC), it is expected that some of these behaviours may not serializable, while all SC behaviours are.

Since TSO-serializability is formulated based on the concept of a syntactic conflict relation (similar to standard SC-serializability), a monitoring algorithm for TSO-serializability can be adapted from the classic algorithm for conflict serializability effortlessly; a program execution can be monitored for TSO-serializability violations using a similar algorithm as SC-serializability [21] and in the same polynomial time complexity. There is, however, a practical impediment in the way of monitoring programs for TSO-serializability violations, and that is how to obtain an execution to monitor in the first place. To obtain a detailed TSO execution (including the information about when writes were committed to memory), the monitor needs access to inner workings of the cache coherence protocol. This implies a very complicated monitor design which will likely have huge performance setbacks. Conceptually, there is a lightly distributed system that needs to be monitored, and observing global snapshots of which are costly.

We propose the notion of *traces*, as an abstraction of executions (in the form of a set of executions) which forgets information about the exact time of write commits. In a trace, once a write is issued by a thread, it can be committed at any point in the future, consistently with all the other accesses in the trace. We pose and solve the problem of monitoring a trace for TSO-serializability violations. Since a trace represents a set of executions, it is expected that this problem should be more complex than the monitoring problem of a single execution. We prove that the problem is in general NP-complete, but fixed-parameter tractable. We propose an algorithm to solve it in polynomial time if the number of threads in the program is considered to be a constant.

2 Multithreaded Programs and Their Executions

Events. A program consists of a number of threads running concurrently and communicating through shared variables. Each thread runs a sequence of transactions, which are themselves sequences of events. We fix arbitrary sets $\mathbb{T}$, $\mathbb{T}r$, $\mathbb{V}$, and $\mathbb{D}$ of thread identifiers, transaction identifiers, variable names, and values.

For a given thread identifier t, we fix the sets $\mathbb{R}_t = \{rd_t(x,v)_i : i \in \mathbb{T}r, x \in \mathbb{V}, v \in \mathbb{D}\}$ and $\mathbb{W}_t = \{wr_t(x,v)_i : i \in \mathbb{T}r, x \in \mathbb{V}, v \in \mathbb{D}\}$ of *read* and *write events*. Events are indexed by thread and transaction identifiers. *Fence events* (which concern the internal workings of TSO and which are explained later in this section) are denoted by fn_t^i. We omit the transaction identifier i when it is understood from the context, or it is irrelevant. Let $\mathbb{E}_t = \mathbb{R}_t \cup \mathbb{W}_t \cup \{fn_t^i : i \in \mathbb{T}r\}$ and $\mathbb{E} = \bigcup_t \mathbb{E}_t$.

Programs. A sequence of events σ is called *serial* when every two events of the same transaction are not separated by an event of another transaction, and *well-formed* when each transaction identifier is used at most once and for each thread t, the projection of σ on events of thread t is serial. A *program* P is abstractly represented as a prefix-closed set of well-formed sequences of events (representing all possible interleavings of events of different threads). The semantics of a program P for a specific memory model consists only of those sequences that are feasible under that memory model.

Memory models. An *SC-execution* is a sequence of events $\eta \in \mathbb{E}^*$ where roughly, each read event reads the value written by the last preceding write. An *SC-execution of a program* P is an SC-execution η such that $\eta \in P$.

Under TSO, a write $wr_t(x,v)_i$ (called also a *write-issue*) is first stored in a thread-local FIFO buffer, called the *store buffer*, before being non-deterministically flushed into the shared memory. The written value may become visible to other threads at a later time. Flushing the store buffers introduces additional events $wr\text{-}com_t(x,v)_i$, called *write-commit* events, for removing a write $wr_t(x,v)_i$ from the store buffer of t and execute it on the shared memory. We say that the write-commit $wr\text{-}com_t(x,v)_i$ *corresponds* to that write, and denote it by $wr_t(x,v)_i \sim wr\text{-}com_t(x,v)_i$. Write-commits *inherit the transaction identifier* of the corresponding write-issue (regardless of when they occur). A read $rd_t(x,v)$ prefetches the value v written by the last write to x in the buffer of t, and if no such write exists, the value v is retrieved from the shared memory. A fence event fn_t is enabled only when the buffer of t is empty. Let $\mathbb{W}c_t = \{wr\text{-}com_t(x,v)_i : i \in \mathbb{T}r, x \in \mathbb{V}, v \in \mathbb{D}\}$, $\mathbb{E}^{tso}{}_t = \mathbb{E}_t \cup \mathbb{W}c_t$, and $\mathbb{E}^{tso} = \bigcup_t \mathbb{E}^{tso}{}_t$. For any $e \in \{rd_t(x,v)_i, wr_t(x,v)_i, wr\text{-}com_t(x,v)_i\}$, $th(e) = t$ and $var(e) = x$. A sequence of events $\eta \in (\mathbb{E}^{tso})^*$ satisfying this semantics is called a *TSO-execution*. A *TSO-execution of a program* P is a TSO-execution η such that the projection of η on $\mathbb{E}$ belongs to P. Fig. 2(a) pictures a TSO-execution of the program in Fig. 1.

3 Conflict Serializability

Conflict serializability was introduced in [21] as a syntactic (and tractable to monitor) notion that ensures *atomicity*. Instead of considering the data manipulated by transactions, a conservative "conflict relation", relating the individual actions of transactions, is defined which guarantees atomicity regardless of the data values read and written by individual actions. A conflict relation relates events with their values projected away, that we also call events (and inherit all the notations from Sec. 2 for sets of events). Conflict serializability is a property of a sequence of events (without values), which are also called SC/TSO-executions. Note that such a sequence represents a *set* of executions, where different values can be assigned to individual events (consistently).

Formally, a *conflict relation* is an irreflexive binary relation $\circledcirc \subseteq \mathbb{E} \times \mathbb{E}$. For a pair of events $e, e' \in \mathbb{E}$, we write $e \circledcirc e'$ to stand for $(e,e') \in \circledcirc$ and $e \not\circledcirc e'$ to stand for $(e,e') \notin \circledcirc$. Intuitively, whenever $e \circledcirc e'$, the effect of executing e after e' *may* differ from that of executing e before e'. The conflict relation depends on the underlying memory model. For instance, the conflict relation $\circledcirc_{SC}$ from [10] assumes sequential consistency: $e \circledcirc_{SC} e'$ whenever e and e' are events of the same thread (i.e., $th(e) =$

$th(e')$) or they access the same variable, and one of them is a write (i.e., $(e, e') \in (\mathbb{R} \cup \mathbb{W})^2 \setminus \mathbb{R}^2$ and $var(e) = var(e')$).

Given an execution $\eta = \eta_1 e e' \eta_2$ (where e and e' are events and η_1 and η_2 are executions), we say an execution $\eta' = \eta_1 e' e \eta_2$ is derived from η by a $\circledcirc$-*valid swap* if and only if $e \not\circledcirc e'$. A permutation η' of an execution η is $\circledcirc$-*preserving* if and only if η' can be derived from η through a sequence of $\circledcirc$-*valid* swaps.

An execution η is *conflict serializable* w.r.t. the conflict relation $\circledcirc$ if and only if there exists an execution η' that is a $\circledcirc$-preserving serial permutation of η. We call the notion of conflict serializability based on $\circledcirc_{SC}$ *SC-serializabiliy* for short. A program P is *SC-serializable* iff every SC-execution of P is SC-serializable.

An equivalent characterization of conflict serializability can be established through *conflict graphs* [21], where the graph was constructed for a specific conflict relation. The same definition can be easily adapted for any conflict relation.

Definition 1 (Event-Graph). *The* event-graph *of an execution η is the directed graph $EG_\eta = \langle V, E\rangle$ where there is a node in V for each event in η, and E contains an edge from u to v iff $e(u) \circledcirc e(v)$ and $e(u)$ occurs before $e(v)$ in η (where $e(v)$ is the event of execution η corresponding to the graph node v).*

Intuitively, one can think of the event-graph of an execution η as a structure that represents the order between all conflicting events in η.

The conflict-graph of an execution η is defined based on the *event-graph* of η by grouping all events indexed by the same transaction identifier as a new node, and considering the directed graph that is induced on these new transaction nodes. Let $tr(v)$ be the set of events that belong to a transaction node v.

Definition 2 (Conflict-Graph). *The* conflict-graph *of an execution η is the directed graph $CG_\eta = \langle V', E'\rangle$ where V' includes one node for each transaction identifier in η, and we have $(v, v') \in E'$ iff there exists events $e \in tr(v)$ and $e' \in tr(v')$ such that $(e, e') \in E$ where $EG_\eta = (V, E)$ is the event-graph of η.*

Theorem 1. *(from [21]) For a conflict relation $\circledcirc$, an execution η is conflict-serializable if and only if CG_η is acyclic.*

In [21], a polynomial time algorithm is presented that uses the conflict graph and Th. 1 to monitor an execution under SC for violations of serializability.

Event-graphs and conflict-graphs of SC-executions are defined as in Def. 1 and Def. 2, respectively, using $\circledcirc_{SC}$ instead of $\circledcirc$.

4 Serializability Under TSO

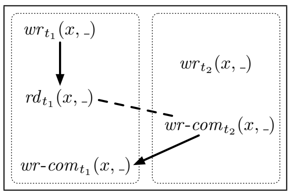
In this section, we propose a conflict relation for TSO and justify the suitability of the obtained notion of conflict serializability by relating it to the classic SC serializability.

4.1 TSO Conflict Relation

The TSO conflict relation $\circledcirc_{TSO}$ is formally defined as follows:

$$
\begin{array}{rll}
e \circledcirc_{TSO} e' \iff & \star\ th(e) \neq th(e') \wedge (e,e') \in (\mathbb{R} \cup \mathbb{W}c)^2 \setminus \mathbb{R}^2 \wedge var(e) = var(e') & \\
& \text{except the following cases:} & (1) \\
(i) & e = rd_{t_1}(x) \ \wedge\ e' = wr\text{-}com_{t_2}(x) \ \wedge\ e \,\|\, e' & \\
& e = wr\text{-}com_{t_1}(x) \ \wedge\ e' = rd_{t_2}(x) \ \wedge\ e' \,\|\, e & \\
(2) & \star\ th(e) = th(e') \text{ except the following cases:} & \\
(ii) & e \in \mathbb{R}_t \cup \mathbb{W}_t \ \wedge\ e' \in \mathbb{W}c_t \ \wedge\ \neg e \sim e' & \\
& e \in \mathbb{W}c_t \ \wedge\ e' \in \mathbb{R}_t \cup \mathbb{W}_t \ \wedge\ \neg e \sim e' & \\
(iii) & e = wr_t(x) \ \wedge\ e' = rd_t(y) \wedge x \neq y & \\
& e = rd_t(x) \ \wedge\ e' = rd_t(y) \wedge x \neq y \wedge rd_t(x) \text{ is buffered} &
\end{array}
$$

Similar to the SC conflict relation, $\circledcirc_{TSO}$ declares events accessing the same shared memory location, where at least one of them is a *write-commit* conflicting (see (1) above). However, since under TSO, some *read* events may access values by reading from a local buffer (instead of the shared memory), there are exceptions to this general rule involving such reads.



A read $rd_{t_1}(x)$ event that occurs between a $wr_{t_1}(x)$ event and the corresponding $wr\text{-}com_{t_1}(x)$ event, and where $wr_{t_1}(x)$ is the most recent write-issue event before $rd_{t_1}(x)$, fetches its value from the store buffer that holds the value written by $wr_{t_1}(x)$. In this case, according to the TSO semantics, event $rd_{t_1}(x)$ should not be in conflict with a write-commit $wr\text{-}com_{t_2}(x)$ of another thread that happens in parallel with it; that is, when $wr\text{-}com_{t_2}(x)$ occurs between the pair of events $wr_{t_1}(x)$ and $wr\text{-}com_{t_1}(x)$ (as illustrated in the figure on the right). Such a pair of parallel read and write-commit events, which we denote by $rd_{t_1}(x) \,\|\, wr\text{-}com_{t_2}(x)$, should not be conflicting since one is a read from a local store buffer and the other a write to the shared memory (accesses to two different resources).

Similar to the SC conflict relation, $\circledcirc_{TSO}$ declares events within the *same thread* to be in conflict (see (2) above). Again, there are exceptions to this rule. A write-commit is *not* in conflict with other read and write events in the same thread (see (ii) above), except for its corresponding write issue (which must always precede it). Other exceptions (see (iii) above) are related to the relaxations of the program order allowed by the TSO semantics. There is no conflict between a write and a read event of the same thread on different variables. This exception is natural since it extrapolates the behaviour of the memory model at the level of events to the level of transactions, i.e., write-only transactions can be reordered with respect to later read-only transactions. Finally, TSO semantics relaxes the program order between a $rd_t(x)$ event that fetches its value from the store buffer and a future $rd_t(y)$ event of a different variable $y \neq x$ (see also [1]).

Buffered Reads. The relative ordering of $wr_t(x)/wr\text{-}com_t(x)$ events corresponding to the read event $rd_t(x)$ (of the same thread) determines whether the read fetches its value from the buffer (or the shared memory). Therefore, every read event $rd_t(x)$, that is preceded by a write $wr_t(x)$ of the same thread and no fence event fn_t in between, may or may not be fetching its value from the local buffer, depending on when the write gets committed to the memory. This runtime information is unavailable when a programmer is reasoning at the level of the source code. We choose to call any such read, that *may*

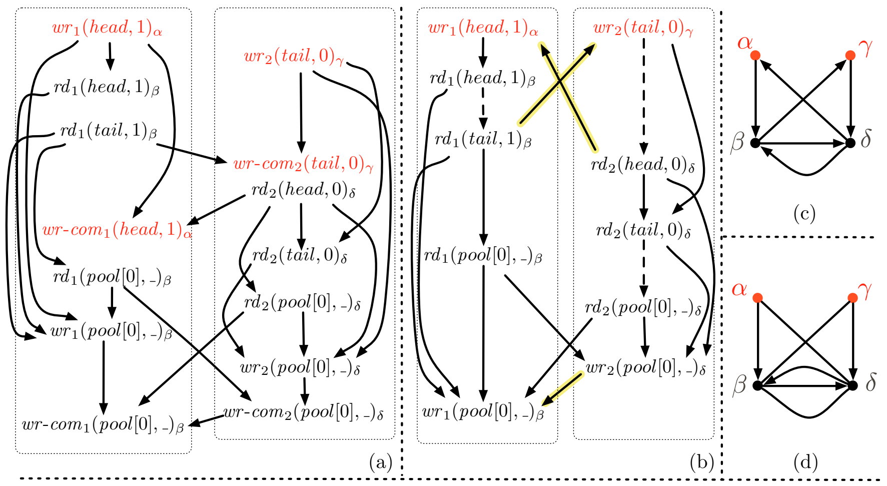


Fig. 2. (a) A TSO-execution η (events are ordered from top to bottom) and its $\circledcirc_{TSO}$ event-graph EG_η. **(b)** Ignoring dashed edges, the write-contraction of EG_η. Dashed edges represent conflicts added by $\circledcirc_{TSO-po}$. Ignoring dashed edges and redefining the highlighted edges to be undirected, the trace event-graph EG_τ of $\tau = trace(\eta)$. **(c)** The conflict-graph induced by EG_η. **(d)** The conflict-graph induced by EG_τ.

fetch its value from the buffer, a *buffered* read and exclude the mutual conflicts between these reads and later reads to other variables from $\circledcirc_{TSO}$ (see (iii) above). This way, we feel that the definition of conflict relation stays true to its main purpose, i.e. defining a notion atomicity that is helpful to programmers reasoning about their code.

The following proposition formally states the fact that all order relaxations introduced in the definition of $\circledcirc_{TSO}$ are consistent with the TSO semantics:

Proposition 1. *Any $\circledcirc_{TSO}$-preserving permutation of a TSO-execution η is also a TSO-execution.*

The notion of conflict serializability based on $\circledcirc_{TSO}$ is called *TSO-serializability*. A program P is *TSO-serializable* iff every TSO-execution of P is TSO-serializable. Event/conflict-graphs of TSO-executions are defined as in Def. 1 and Def. 2, respectively, by replacing $\circledcirc$ with $\circledcirc_{TSO}$. An equivalent of Th. 1 then provides an efficient (poly-time) procedure to monitor an execution for TSO-serializability violations. Fig. 2(a) illustrates the event-graph of a non TSO-serializable execution of the program in Fig. 1. The conflict-graph in Fig. 2(c) contains a cycle.

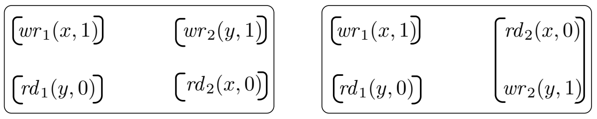
4.2 Connection to SC-serializability

Beyond Prop. 1, we substantiate our definition of TSO-serializability by formally relating it to the widely accepted notion of SC-serializability. We show that SC-serializability implies TSO-serializability for *robust* programs. Intuitively, a program is robust if it does not exhibit non-SC behaviour; in other words, each of its TSO-executions is equivalent to another execution of the same program under SC. Under

SC, every write-issue is immediately followed by the corresponding write-commit (i.e. no delay in propagating the write).

Let $\circledcirc_{TSO-po}$ be a strengthening of $\circledcirc_{TSO}$ in which the program order is maintained for all pairs of events in $\mathbb{E}$ in the same thread. Formally, a TSO-execution η is *SC-equivalent* when there exists an execution η' that is a $\circledcirc_{TSO-po}$-preserving permutation of η and every write-issue of η' is immediately followed by the corresponding write-commit. A program P is *robust* when every TSO-execution of P is SC-equivalent. One can check SC-equivalence by letting every pair of write-issue and corresponding write-commit events to form a transaction, and checking conflict serializability of the execution consisting of these transactions and all other events as single transactions (more details in [1]). The conflict graph defined this way is called a *write-contraction*. For instance, the TSO-execution in Fig. 2(a) is not SC-equivalent (since there is a cycle in Fig. 2(b)) which implies that the program in Fig. 1 is not robust.

Theorem 2. *A program P is TSO-serializable if it is robust and SC-serializable.*



The reverse of Theorem 2 doesn't hold. For instance, both programs above are TSO-serializable although the program in the left is not robust and the program in the right is not SC-serializable. The program in the left is TSO-serializable since every event is a transaction and events in the same thread are not in conflict, and it is not robust since intuitively, both reads don't see the value written by the other thread. The program in the right is TSO-serializable because the events in thread 1 are not in conflict while it is not SC-serializable since it admits only one execution where the events of thread 1 take place in between the two events of thread 2.

A program P is called *transaction-fenced* when for every $\sigma \in P$, every transaction in σ, i.e., every maximal sub-sequence of events indexed by the same transaction identifier, ends with a fence [3]. For transaction-fenced programs, the converse of Th. 2 is true:

Theorem 3. *A transaction-fenced program P is TSO-serializable iff it is robust and SC-serializable.*

5 Trace TSO-Serializability

There are practical obstacles in the way of implementing a monitor that can observe a TSO-execution of a program. The monitor is subject to the same distributed nature of the memory as individual program threads, and tracking write-commits of threads requires a manipulation of the cache-coherence protocols running in the multi-core chip with potentially high performance overheads. We introduce a notion of serializability for TSO that does not require to be aware of the exact timing of write-commits. This notion applies to abstractions of TSO-executions called *traces* that forget write-commits,

[3] Transaction-fenced programs are not necessarily robust since statements inside a transaction may not be followed by a fence.

assuming that a write-commit can happen at any point in time after its corresponding write-issue (consistent with the TSO semantics). This effectively means that the serializability of a *set* of executions (namely those where the forgotten write-commits reappear at any of the consistent points) is monitored instead of a single execution.

The trace of an execution η, denoted by $trace(\eta)$, is the projection of η on $\mathbb{E}$ (basically leaving out all write-commits). The set of executions $Execs(\tau)$ represented by a trace τ is the set of all TSO-executions η such that $trace(\eta) = \tau$.

Definition 3 (Trace TSO-Serializability). *A trace τ is* TSO-serializable *iff every execution in $Execs(\tau)$ is TSO-serializable.*

The most important property of $\tau = trace(\eta)$ for some execution η is that τ can soundly be used to check if η is not TSO-serializable.

Proposition 2. *If execution η is not TSO-serializable then the trace $trace(\eta)$ is not TSO-serializable.*

We introduce a conflict relation $\overrightarrow{\circledcirc}_{TSO}$ for traces and a characterization of serializability based on that conflict relation. Intuitively, $\overrightarrow{\circledcirc}_{TSO}$ stands for the union of the conflict relations for all the individual executions of that trace, where a write event represents both the write-issue and the corresponding write-commit. The relation $\overrightarrow{\circledcirc}_{TSO}$ over traces is the union of two disjoint relations $\overrightarrow{\circledcirc}_{TSO}$ and $\overline{\circledcirc}_{TSO}$. Given $e, e' \in \mathbb{E}$,

$e \overrightarrow{\circledcirc}_{TSO} e'$ iff $th(e) = th(e')$ except the following cases:

$e = wr_t(x) \;\wedge\; e' = rd_t(y) \wedge x \neq y$

$e = rd_t(x) \;\wedge\; e' = rd_t(y) \wedge x \neq y \wedge rd_t(x)$ is *a buffer read*

$th(e) \neq th(e')$ and $(e \circledcirc_{SC} e' \wedge fence(e, e')$

or $e = rd_t(x) \wedge e' = wr_{t'}(x) \wedge e$ is not buffered)

$e \overline{\circledcirc}_{TSO} e'$ iff $th(e) \neq th(e') \wedge e \circledcirc_{SC} e' \wedge \neg e \overrightarrow{\circledcirc}_{TSO} e'$

The conflicts between events of the same thread are included in $\overrightarrow{\circledcirc}_{TSO}$ since the order between such events is fixed in all the executions of the trace. Two events of different threads are in conflict if they are so under the classic SC conflict relation, and they are related by $\overrightarrow{\circledcirc}_{TSO}$ iff they are *separated by a fence* (since the fence ensures they are ordered in the same way in all executions) or if they are a non-buffered read (reading from the shared memory) together with a write (since a read cannot see the value of a write that hasn't been issued yet). Formally, e and e' are *fence-separated*, denoted by $fence(e, e')$, when e occurs before e', e is an action of thread t, and τ contains a fence fn_t between e and e'. In contrast, $\overline{\circledcirc}_{TSO}$ relates events that are conflicting under $\circledcirc_{SC}$ but may appear in different orders in different executions of a trace, for example two write events (of the same variable) performed by two different threads. Recall that a write represents both the write-issue and the corresponding write-commit.

Similar to the case of executions, having a graph theoretic characterization of serializability for traces is useful for algorithm design. We define the event-graph of a trace τ that contains a directed edge from event e to event e' iff $e \overrightarrow{\circledcirc}_{TSO} e'$ and an undirected edge between e and e' iff $e \overline{\circledcirc}_{TSO} e'$.

Definition 4 (Trace Event-Graph). *The* event-graph *of a trace τ is the graph $EG_\tau = \langle V, E, U\rangle$ where there is a node in V for each event in τ, E is a set of directed edges (u, v) such that $e(u)$ occurs before $e(v)$ in τ and $e(u)\overrightarrow{\circledcirc}_{TSO}e(v)$, and U is a set of undirected edges $\{u, v\}$ such that $e(u)$ occurs before $e(v)$ in τ and $e(u)\overline{\circledcirc}_{TSO}e(v)$ (where $e(v)$ is the event of τ corresponding to the node v).*

Formally, an *orientation* of a graph $G = \langle V, E, U\rangle$ with a set E of directed edges and a set U of undirected edges is a directed graph $\langle V, E \cup E'\rangle$ such that for every undirected edge $\{u, v\} \in U$, E' contains (u, v) or (v, u). An orientation of EG_τ is *valid* when the resulting directed graph is acyclic.

The next result relates valid orientations of the trace event-graph and write-contractions of the trace's executions event-graphs. Recall that the *write-contraction* of an event-graph EG_η is the graph EG^c_η where every node representing a write event $wr_t(x)$ is merged with the node representing the corresponding write-commit event $wr\text{-}com_t(x)$ (note that a contracted edge disappears and does not turn into a self-loop).

Theorem 4. *For an execution $\eta \in Execs(\tau)$, the write-contraction of EG_η is a valid* orientation *of EG_τ. Conversely, every valid orientation of EG_τ is the write-contracted event-graph EG_η for some $\eta \in Execs(\tau)$.*

This leads to an interesting observation: EG_τ of a trace τ can be viewed as the union of the write-contractions EG^c_η of all $\eta \in Execs(\tau)$, so that when all EG^c_ηs agree on the direction of an edge between two nodes, that edge appears as a directed edge in EG_τ and when at least two EG^c_ηs disagree on the direction of an edge between two nodes, that edge appears as an undirected edge in EG_τ.

Also, Th. 4 leads us to the following characterization of trace TSO-serializability based on orientations of trace event-graphs.

Theorem 5. *A trace τ is TSO-serializable iff every acyclic orientation of EG_τ induces an acyclic conflict-graph.*

Alternatively, one can directly define the notion of a conflict graph for traces. The event graph of a trace EG_τ induces a graph over the transactions in the same sense as the conflict graph of an execution.

Definition 5 (Trace Conflict-Graph). *The* conflict-graph *of a trace τ is the graph $CG_\tau = \langle V', E', U'\rangle$ where V' includes one node for each transaction in τ, and we have $(v, v') \in E'$ iff there exists actions $a \in tr(v)$ and $a \in tr(v')$ such that $(a, a') \in E$ and we have $\{v, v'\} \in U'$ iff there exists actions $b \in tr(v)$ and $b' \in tr(v')$ such that $\{b, b'\} \in U$ where $EG_\tau = (V, E, U)$ is the event-graph of τ.*

For instance, the conflict-graph of the trace of the execution in Fig. 2(a) is given in Fig. 2(d). Serializability of a trace τ can be stated as a combined property of its conflict-graph CG_τ and its event-graph EG_τ.

Corollary 1. *Trace τ is not TSO serializable iff there exists a cycle c in $CG_\tau = \langle V', E', U'\rangle$ such that if $\{u_1, \dots u_m\} \subseteq U'$ participate in c and $\{e_1, \dots, e_m\}$ are the same set of edges oriented in the direction of the cycle, then there exists a valid orientation $\langle V, E''\rangle$ of the event-graph $EG_\tau = \langle V, E, U\rangle$ with $\{e_1, \dots, e_m\} \subseteq E''$.*

6 Monitoring TSO-Serializability of Traces

In this section, we discuss the algorithmic aspect of monitoring *traces* for violations of TSO-serializability. Remember that (Section 3) monitoring one execution for violation of TSO-serializability is poly-time checkable.

Given a trace τ, we want to check whether τ is TSO-serializable. We start by demonstrating that the general problem is NP-complete, and then propose polynomial time algorithms for approximations of this check. Specifically, we show that (i) under the assumption that the number of threads is a constant, there exists a sound and complete polynomial time algorithm that reports violations of TSO-serializability in a trace τ, and (ii) if the program is *transaction-fenced*, then TSO-serializability can be checked in polynomial time.

6.1 NP-Completeness of Trace TSO-Serializability Checking

Th. 5 provides an equivalent characterization of trace TSO-serializability, namely that every acyclic orientation of the trace event-graph induces an acyclic conflict-graph. It turns out that this check is NP-complete. We demonstrate this by reducing the known NP-complete problem of checking for the existence of a hamiltonian path in a given graph G to this problem.

Theorem 6. *For a trace τ, the problem of checking whether τ is TSO-serializable is* NP-*complete.*

6.2 Fixed-Parameter Tractability

The good news is that there exists an algorithm for monitoring a trace for TSO-serializability violations which is polynomial time if one assumes the number of threads to be a constant. Given a trace of length n with k participating threads, it is easy to devise an exponential algorithm that finds a TSO-serializability violation if one exists and operates in $O(n^k)$ time. However, considering that usually n (the number of events) is very large, it is desirable to have an algorithm with a running time where the exponent k does not appear over n, but over some constant instead.

In this section, we propose an algorithm of complexity $O(n + c^k)$, where c is a constant that depends on the number of shared variables in the program, k is the number of threads, and n is the length of the trace. The main observation that gives rise to such an algorithm is that there is a *concise witness* to violation of TSO-serializability, and it suffices to search for the existence of such a witness algorithmically. We start by defining this concise witness, which always exists if an arbitrary witness exists.

Given the event graph EG_τ of a trace τ, checking serializability of τ reduces to deciding if there is a valid orientation of EG_τ that induces a cycle over the conflict graph CG_τ. We will observe that if a valid orientation of EG_τ induces a cycle, then this orientation induces a *simple* cycle (to be defined) over CG_τ.

Naturally, if the directed edges of the conflict graph CG_τ already form a cycle (which can be checked in polynomial time on the size of the graph), then there is nothing left to be done; we have found our TSO-serializability violation witness. Therefore, we assume that CG_τ is acyclic if it is restricted to its directed edges; let us call this graph $\overrightarrow{CG_\tau}$. Similarly, $\overrightarrow{EG_\tau}$ refers to EG_τ restricted to its directed edges. We use the notation $a \lessdot_\tau b$ to denote that $\overrightarrow{EG_\tau}$ contains a path from event a to event b. Similarly, for transactions tr_1 and tr_2, we use the notation $tr_1 \lessdot_\tau tr_2$ iff $\overrightarrow{CG_\tau}$ contains a path from tr_1 to tr_2. The relation $\lessdot_\tau$ captures the ordering constraints between events/transactions that are imposed by the directed conflict edges. For an event a, we use $tr(a)$ to refer to the transaction that encloses a.

Let us assume that we have a cycle $c = tr_1 tr_2 \dots tr_m$ over the conflict graph CG_τ. For each pair of consecutive transactions tr_i and tr_{i+1}, let event b_i be the source and event a_{i+1} be the destination of the conflict edge between tr_i and tr_{i+1} that participates in the cycle (rotating back from b_m to a_1).

We say that cycle c can be *simplified* if there exist two transactions tr and tr' on it where $tr \lessdot_\tau tr'$ and the segment of the cycle between tr and tr' contains at least one *undirected* conflict edge. By taking this segment of the cycle between tr and tr' and replacing it with the directed path (i.e. a path formed entirely of directed conflict edges) in the conflict graph from tr to tr', we *simplify* the cycle; we know that such a path exists by the definition of $tr \lessdot_\tau tr'$. Intuitively, during simplification we get rid of *undirected edges* and replace them by directed paths; note that undirected edges are soft constraints in a trace which reflect that the order between two events is undetermined.

Definition 6. *A* simple *cycle is a cycle that cannot be further simplified.*

Below, we state two properties of simple cycles that are very useful for reducing the search space of our algorithm.

Proposition 3. *In every* simple *cycle $c = tr_1 tr_2 \dots tr_m tr_1$ over the conflict graph CG_τ of a trace τ (equivalently $c = a_1 b_1 a_2 b_2 \dots a_m b_m a_1$ if the cycle is referenced by its conflict edges instead of its nodes) satisfies the following properties: (i) There exists at least one index k such that $a_k \not\lessdot_\tau b_k$. (ii) Every two transactions tr and tr' that appear on c with an undirected edge somewhere in the middle of them (i.e. on the segment between tr to tr') cannot belong to any chain (i.e. directed path) of the graph. In other words, we have $tr \not\lessdot_\tau tr'$.*

Property (ii) from the proposition above is straightforward yet significant because it implies that any simple cycle over the conflict graph can be viewed as a cycle where undirected edges connect segments of chains (i.e. directed paths) in the graph together, never visiting the same chain twice. We make use of the notion of *profiles* introduced in [11] for this algorithm. The idea is to summarize all possible entry/exits into each chain of $\overrightarrow{CG_\tau}$ (that may participate in a simple cycle) as a set of pairs (of events), and look for cycles involving those pairs only.
Consider an event a of the event graph EG_τ. Let

$$pair(a) = \{b | \, b \in tr(a) \ \lor \ tr(a) \lessdot_\tau tr(b))\}$$

The idea is that once a witness cycle enters $tr(a)$ through a conflict edge with destination a, some $b \in pair(a)$ is the event from which the cycle can leave the chain (i.e. directed path) that contains $tr(a)$ and $tr(b)$. In other words, $\{a\} \times pair(a)$ is the set of all possible path segments that start with a and can be part of a simple cycle witnessing a violation of TSO-serializability.

```
Input: $\Pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$ smallest partition of $\overrightarrow{CG_\tau}$ into chains.
Output: a witness to violation of TSO-serializability, if one exists.

For all $\pi_i \in \Pi$ and each $tr \in \pi_i$
  For all events $a, b \in tr$ where $a \nless_\tau b$, and each choice of events
    For all events $a', b'$ where $tr(a'), tr(b') \in \pi_i$ and $b' \in pair(b)$ and $a \in pair(a')$
      For all $p_1 \in profile(\pi_1), \ldots, p_m \in profile(\pi_m)$
          If $(a', b')$ together with $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_m$ includes
          a TSO-serializability violation, then report a violation.
```

Fig. 3. Algorithm for searching for all simple cycle witnesses. The choices of events for a, b, a', b' is over read and write events only. The chains $\pi_1, \ldots \pi_m$ are by definition disjoint.

```
$P_\pi = \varnothing$
For variables $x$, if $\exists$ events $a \in \pi$ where $a = rd(x)/wr(x)$ then $P_\pi = P_\pi \cup \{(a)\}$.
For each pair of variables $x$ and $y$ (can be equal)
  If $\exists$ events $a, b \in \pi$ where $a = rd(x)/wr(x)$ and $b = rd(y)/wr(y)$ and $b \in pair(a)$
      then $P_\pi = P_\pi \cup \{(a, b)\}$.
```

Fig. 4. Algorithm for computing the set of all profiles of a transaction chain π.

Moreover, Prop. 3(i) states that at least for one transaction in the cycle we have a pair of events (a, b) of the same transaction where $a <_\tau b$ but where a and b participate in the witness cycle, which is directed from b back to a. The algorithm presented in Fig. 3 starts by enumerating all such pairs of events that belong to a single transaction (outermost loop). It then proceeds to find the matching entry/exit events (i.e. a' and b') for the witness cycle in the chain containing a and b (the next nested loop). Finally, the innermost loop enumerates all possible choices of profiles for the remaining chains (other than the one containing a, b, a' and b'), and then the innermost statement checks if these choices form a valid witness cycle together.

The algorithm in Fig. 3 uses a function *profile* that returns the set of all profiles for a given chain. A profile of a chain is a set of elements of the following three forms: (i) a single event (a), when the witness conflict cycle enters and exits a chain at the same single event a, (ii) a pair of events (a, b) of some transaction tr, where the witness cycle enters/exits a chain at two events of the same transaction tr, and (iii) a pair of events (a, b), where a witness cycle enters a transaction in event a, then follows a chain of transactions on a directed path in the conflict graph and exits the chain through an event b (i.e. $tr(a) <_\tau tr(b)$). The set of profiles of a chain can be computed using the algorithm in Fig. 4.

Soundness and Completeness Here, we formally argue that it suffices for the algorithm to search for *simple* cycle witness to violation of TSO-serializability. The important observation is that:

Proposition 4. *If a trace τ is not TSO-serializable, then there exists a simple cycle witnessing the violation of TSO-serializability.*

It remains to argue that the algorithm, through the use of profiles, will definitely find a simple cycle violation of TSO-serializability if one exists.

Proposition 5. *For every partitioning Π of $\overrightarrow{CG_\tau}$ into a set of chains, and every simple cycle violation of TSO-serializability c, we have that c visits every chain in Π at most once.*

It is important to note that the above statement is independent of the choice of partitioning of $\overrightarrow{CG_\tau}$ into chains. It is straightforward to see that a cycle's footprint in every chain can be captured through one of the three possibilities that we introduced for profiles. Finally, we conclude the soundness and completeness of the algorithm in Fig. 3:

Theorem 7. *Algorithm in Fig. fig:alg1 discovers a violation of TSO serializability in trace τ iff one exists.*

Complexity Analysis A key observation about $\overrightarrow{EG_\tau}$ is that for any trace τ, if $\overrightarrow{CG_\tau}$ is restricted to a single thread and global read and write events, then the size of the largest anti-chain of it is at most 2. In other words, in every thread, there are at most two events a and b such that $a \nprec_\tau b$ and $b \nprec_\tau a$. This is a direct implication of the definition of $\circledcirc_{TSO}$; the only events that are not ordered in each thread are $wr(y)$ and $rd(x)$ when $x \neq y$, and the events appear in that order in the trace. Any other event that can be independent of $wr(y)$ will have to be a read event of some other variable, say $rd(z)$ which is in conflict with $rd(x)$ and therefore ordered with respect to it (similar argument for events independent of $rd(x)$). We will make use of the following well-known theorem about the width of a partial order:

Theorem 8 (Dilworth's Theorem). *For every partial order, there exists an anti-chain A, and a partition of the order into a family P of chains, such that $|P| = |A|$ (which is referred to as the width of the partial order). Moreover, such an A is the largest anti-chain in the order.*

Since $\overrightarrow{EG_\tau}$ is acyclic, by Dilworth's Theorem, we know that it can be partitioned into at most p (maximal) chains (i.e. directed paths) where p is the size of the largest anti-chain of $\overrightarrow{EG_\tau}$. The size of the largest anti-chain of $\overrightarrow{EG_\tau}$ restricted to each thread (and ignoring the buffered reads) is at most 2. If we assume that there are k threads in the program, this implies that $\overrightarrow{EG_\tau}$ (ignoring the buffered reads) can be partitioned into $2k$ chains. If we have m shared variables in the program, then each such chain can be summarized as at most $(2m)^2$ possible profiles (i.e. all possible combinations of $2m$ reads and $2m$ writes).

Now, let us add consideration for the *buffered* reads. In each thread, all buffered reads of the same variable are conflicting and form a chain. Therefore, in the worst case, we can account for all buffered reads of a single thread, by adding m extra chains, where each consists of all buffered reads of some variable x (there are at most m different variables). There are in total km of such chains for all k threads. However, every such chain (of buffered reads of x) can be represented by a single trivial profile $(rd(x))$.

Our algorithm ends up enumerating all possible profiles for such partitioning of $\overrightarrow{EG_\tau}$ into a family of chains. There are at most $((2m)^2)^{2k}$ different selection of profiles to consider. It is easy to see that it takes $O(n)$ time (n is the length of the trace) to compute the set of all profiles.

We need to argue that given the combination of the fixed km (trivial) profiles and a choice of $2k$ profiles (from $((2m)^2)^{2k}$ many choices), a violation can be found in polynomial time, if one exists. This is equivalent to having a system of (at most) $(m + 2)k$ components, where each component is a single event, a pair of events connected by an undirected edge, or a pair of components linked by a directed edge. The goal is to find a cycle in this system that obeys the direction of the directed edges. A slightly modified depth-first search algorithm can find the cycle in time polynomial in mk.

To summarize, the complexity of the algorithm is $O(n + c^k)$ where n is the length of the trace, c depends only on the number of shared variables in the program, and k is the number of program threads.

Theorem 9. *For a program P with a fixed number of threads, the algorithm in Fig. 3 discovers a witness to violation of TSO-serializability of any trace of P in time polynomial on the length of the trace.*

6.3 Poly-time Monitor for Transaction-Fenced Programs

An alternative way of avoiding the high complexity of monitoring traces for TSO-serializability violations, for instance when there is a large number of threads in the program, is to simplify this check by ensuring that every transaction ends with a fence event (and hence making all its updates visible to other threads when it ends). As stated in Theorem 3, TSO-serializability is equivalent to the conjunction of robustness and SC-serializability for such programs.

A witness to non-robustness of a program can be discovered through a targeted search (for a specific pattern of violations) in the space of *SC-executions* of the program [6] using an algorithm that works in polynomial time for a given execution. The combination of these two monitors, a poly-time monitor for SC-serializability and a poly-time monitor for robustness, gives rise to an efficient monitor for TSO-serializability that observes only SC-executions of a program and looks for robustness or SC-serializability violations. Every violation to TSO-serializability will manifest as an SC-serializability violation or as a robustness violation for a transaction-fenced program.

The advantages of this result are twofold: (i) when transactions are naturally fenced (e.g. a lot of Java library methods are like this), it provides a poly-time algorithm for monitoring TSO-serializability, and (ii) when transactions are not naturally fenced, and the program has a large number of threads (which limits the applicability of the algorithm in Sec.6.2), it provides the programmer with a solution: namely, to insert a fence at the end of each transaction that is not already fenced, and gain an efficient sound and complete monitor for TSO-serializability. Having a transaction-fenced program has the additional advantage that it allows to reason about the more familiar notions of SC-serializability and robustness instead of directly reasoning about TSO-serializability.

7 Related Work

To the best of our knowledge, this paper provides the first definition of conflict serializability under TSO. Conflict serializability was introduced in Papadimitriou [21] for database transactions. Decision procedures for conflict serializability of finite-state concurrent models executed under an SC semantics were proposed in [10, 11] and [5]. Both static [13, 17, 24, 26] and dynamic tools [12, 25, 14, 23] have been developed to check SC serializability, as well as transactional memory techniques that enforce serializability at run time [18, 9, 22, 16]. The non-atomicity of writes under TSO poses new algorithmic challenges for monitoring serializability. Since observing the detailed sequence of write issues and commits is not efficiently possible (without access to the cache coherence mechanism), any dynamic analysis needs to monitor executions with missing information, that effectively stand for sets of executions. We propose a new monitoring algorithm for traces (i.e. sets of executions) that searches for certain type of cycles in graphs with both directed and undirected edges, which is more challenging than the classic serializability monitor that searches for a cycle in a directed graph [21].

Linearizability has been studied for concurrent *objects* running under TSO [7, 15, 19]. This provides a means of establishing a relation between a concrete and an abstract object, which must hold in the context of *every possible client* of the object. The abstract object methods need not be atomic. In contrast, serializability is a property that is applicable to *programs* and the atomicity of a transaction is considered in the context of one specific program (in contrast to all possible clients).

Notions of robustness for TSO programs have been investigated in [2, 4, 3, 6, 8]. However, we are not aware of any work that establishes a relationship between robustness and atomicity under different memory models as done in this paper.

Bibliography

[1] On atomicity in presence of non-atomic writes (extended version). `www.cs.toronto.edu/~azadeh/extended/tacas16-extended.pdf`.

[2] J. Alglave and L. Maranget. Stability in weak memory models. In *CAV 2011*, pages 50–66, 2011.

[3] A. Bouajjani, R. Meyer, and E. Möhlmann. Deciding robustness against total store ordering. In *ICALP 2011*, pages 428–440, 2011.

[4] A. Bouajjani, E. Derevenetc, and R. Meyer. Checking and enforcing robustness against TSO. In *ESOP 2013*, pages 533–553, 2013.

[5] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Verifying concurrent programs against sequential specifications. In *ESOP 2013*, pages 290–309, 2013.

[6] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV 2008*, pages 107–120, 2008.

[7] S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In *ESOP 2012*, pages 87–107, 2012.

[8] J. Burnim, K. Sen, and C. Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. In *TACAS 2011*, pages 11–25, 2011.

[9] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS 2009*, pages 157–168, 2009.

[10] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *CAV 2008*, pages 52–65, 2008.

[11] A. Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *TACAS 2009*, pages 155–169, 2009.

[12] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Sci. Comput. Program.*, 71(2):89–109, 2008.

[13] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4), 2008.

[14] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI 2008*, pages 293–303, 2008.

[15] A. Gotsman, M. Musuvathi, and H. Yang. Show no weakness: Sequentially consistent specifications of TSO libraries. In *DISC 2012*, pages 31–45, 2012.

[16] T. L. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA 2003*, pages 388–402, 2003.

[17] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *VMCAI 2004*, pages 175–190, 2004.

[18] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA 1993*, pages 289–300, 1993.

[19] R. Jagadeesan, G. Petri, C. Pitcher, and J. Riely. Quarantining weakness - compositional reasoning under relaxed memory models (extended abstract). In *ESOP 2013*, pages 492–511, 2013.

[20] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, 2008.

[21] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[22] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[23] A. Sinha, S. Malik, C. Wang, and A. Gupta. Predicting serializability violations: SMT-based search vs. DPOR-based search. In *HVC 2011*, pages 95–114, 2011.

[24] C. von Praun and T. R. Gross. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology*, 3(6):103–122, 2004.

[25] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Software Eng.*, 32(2):93–110, 2006.

[26] J. Yi, T. Disney, S. N. Freund, and C. Flanagan. Cooperative types for controlling thread interference in java. In *ISSTA 2012*, pages 232–242, 2012.

A Program Semantics

A.1 Sequential Consistency Semantics

In a sequentially consistent (SC) environment, the threads write and read directly from the shared memory. The state of the shared memory is modeled as a map $\theta : \mathbb{V} \to \mathbb{D}$ from variables to values. The relation $\to^{e}_{SC}$ models the effect of an event e on the shared memory ($\theta[x \mapsto v]$ is identical to θ except that the variable x is mapped to the value v):

$$\textsc{Read}\;\frac{\theta(x) = v}{\theta \xrightarrow[SC]{rd_t(x,v)} \theta} \qquad \textsc{Write}\;\frac{}{\theta \xrightarrow[SC]{wr_t(x,v)} \theta[x \mapsto v]} \qquad \textsc{Other}\;\frac{a \in \{fn_t\}}{\theta \xrightarrow[SC]{a} \theta}$$

An *SC-execution* is a well-formed sequence of events $\eta = e_0 \ldots e_n \in \mathbb{E}^*$ such that there exist $\theta_0, \ldots, \theta_{n+1}$ with $\theta_i \to^{e_i}_{SC} \theta_{i+1}$, for each $0 \le i \le n$. For a program P, an SC-execution $\eta = e_0 \ldots e_n \in P$ is called an SC-execution of P.

A.2 Total Store Ordering Semantics

Under TSO, a write event $wr_t(x,v)_i$ is first stored in a thread-local FIFO buffer, called *store buffer*, before being non-deterministically flushed into the shared memory. Therefore, the written value may be visible to other threads only at a later time. To help intuition, we sometimes use the term *write-issue* event instead of simply write event. Flushing store buffers introduces additional events $wr\text{-}com_t(x,v)_i$, called *write-commit* events, for removing a write $wr_t(x,v)_i$ from the store buffer of t and executing it on the shared memory. We say that the write-commit $wr\text{-}com_t(x,v)_i$ *corresponds* to that write, and write $wr_t(x,v)_i \sim wr\text{-}com_t(x,v)_i$. *Note that write-commits inherit the transaction identifier of the corresponding write-issue.* A read event $rd_t(x,v)$ prefetches the value v written by the last write to x in the buffer of t or if no such write is found, it receives the value v stored in the shared memory. A fence event fn_t is enabled only when the buffer of t is empty. Let $\mathbb{W}c_t = \{wr\text{-}com_t(x,v)_i : i \in \mathbb{T}r, x \in \mathbb{V}, v \in \mathbb{D}\}$, ${\mathbb{E}^{tso}}_t = \mathbb{E}_t \cup \mathbb{W}c_t$, and $\mathbb{E}^{tso} = \bigcup_t {\mathbb{E}^{tso}}_t$.

The state of a program P with threads $\{1, \ldots, n\}$ is modeled as a tuple $\Theta = \langle \theta, buf_1, \ldots, buf_n \rangle$, where θ is the state of the shared memory and buf_i is the local buffer of thread i. The relation $\to^{e}_{TSO}$, defined in Figure 5, models the effect of an event e on the program state.

A *TSO-execution* is a sequence of events $\eta = e_0 \ldots e_n \in (\mathbb{E}^{tso})^*$ such that there exist $\Theta_0, \ldots, \Theta_{n+1}$ with $\Theta_i \to^{e_i}_{TSO} \Theta_{i+1}$, for each $0 \le i \le n$, and the projection over events in $\mathbb{E}$ is well-formed (i.e., write-commits are allowed to escape transaction boundaries).

For a program P, a TSO-execution $\eta = e_0 \ldots e_n$ such that the projection of η on $\mathbb{E}$ belongs to P is called a TSO-execution of P.

Note that due to the non-determinism in flushing the store buffers, different TSO-executions may have the same projection on to the set $\mathbb{E}$. For instance, the projection of

$$\textsc{Prefetch}\ \frac{buf_t = \alpha \cdot wr_t(x,v) \cdot \beta,\ x \notin \beta}{\Theta \xrightarrow[TSO]{rd_t(x,v)} \Theta} \qquad \textsc{Read-shared}\ \frac{x \notin buf_t,\ \theta(x) = v}{\Theta \xrightarrow[TSO]{rd_t(x,v)} \Theta}$$

$$\textsc{Write-issue}\ \frac{}{\Theta \xrightarrow[TSO]{wr_t(x,v)} \Theta[buf_t \mapsto wr_t(x,v) \cdot buf_t]}$$

$$\textsc{Write-commit}\ \frac{buf_t = \alpha \cdot wr_t(x,v)}{\Theta \xrightarrow[TSO]{wr\text{-}com_t(x,v)} \Theta[\theta(x) \mapsto v]} \qquad \textsc{Fence}\ \frac{buf_t = \epsilon}{\Theta \xrightarrow[TSO]{fn_t} \Theta}$$

Fig. 5. TSO Semantics

both TSO-executions

$$wr_t(x,v)\ wr_{t'}(y,v')\ wr\text{-}com_t(x,v)\ wr\text{-}com_{t'}(y,v')$$
$$wr_t(x,v)\ wr_{t'}(y,v')\ wr\text{-}com_{t'}(y,v')\ wr\text{-}com_t(x,v)$$

on to the set $\mathbb{E}$ is $wr_t(x,v)\ wr_{t'}(y,v')$.

B Proofs of Section 4

B.1 Reordering reads under TSO

We show that TSO can reorder reads on different variables using the following slightly modified version of the Dekker mutual exclusion algorithm:

```
Thread 1    Thread 2

x := 1;     y := 1;
rx := x;    ry := y;
r1 := y;    r2 := x;
```

The following is a possible TSO-execution of this program (as usually, we ignore events writing or reading thread-local variables, e.g., `rx` and `ry`):

$$\begin{array}{rl}
wr_1(x,1) & \\
 & wr_2(y,1) \\
rd_1(x,1) & \\
 & rd_2(y,1) \\
rd_1(y,0) & \\
 & rd_2(x,0) \\
wr\text{-}com_1(x,1) & \\
 & wr\text{-}com_2(y,1)
\end{array}$$

The only way to interpret this execution as a classical SC-interleaving is to permute the last reads in each thread to the beginning of the thread, past both the write and the read on x and respectively, y. More precisely, the SC-interleaving that corresponds to this execution is:

$$\begin{array}{ll}
rd_1(y,0) & \\
 & rd_2(x,0) \\
wr_1(x,1) & \\
 & wr_2(y,1) \\
rd_1(x,1) & \\
 & rd_2(y,1)
\end{array}$$

B.2 Proof of Proposition 1

Let $\eta = \eta_1 \cdot e \cdot e' \cdot \eta_2$ be a TSO-execution where values are explicit within events such that $e \not\circledcirc_{TSO} e'$. Let Θ_i with $i \in [0,3]$ be program states such that

$$\Theta_0 \xrightarrow[TSO]{\eta_1} \Theta_1 \xrightarrow[TSO]{e \cdot e'} \Theta_2 \xrightarrow[TSO]{\eta_2} \Theta_3$$

(we use the standard extension of a transition relation to sequences of events).

We show that permuting the events e and e' results in the same state Θ_2, i.e.,

$$\Theta_1 \xrightarrow[TSO]{e' \cdot e} \Theta_2$$

which is enough to conclude that any $\circledcirc_{TSO}$-preserving permutation of η is also a TSO-execution. When both e and e' are read events, or when e and e' access different variables and they belong to different threads the claim holds trivially. In the following, we consider the remaining cases:

1. $e = rd_t(x,v)$ **and** $e' = wr_{t'}(y,v')$ **where** $t \neq t'$**:** Follows from the fact that e reads from the shared memory or the store buffer of t while e' adds an element to the store buffer of t'.

2. $e = rd_t(x,v)$ **and** $e' = wr\text{-}com_{t'}(x,v')$ **where** $t \neq t'$**:** By the definition of $\circledcirc_{TSO}$, η_1 must contain a write issue $wr_t(x,v)$ and η_2 the corresponding write-commit for that write-issue $wr\text{-}com_t(x,v)$. Since e reads from the store buffer of t and doesn't access the shared memory, it remains enabled even if executed after e'. The fact that permuting these two events leads to the same program state is straightforward.

3. $e \in \mathbb{E}_t$ **and** $e' = fn_{t'}$ **where** $t \neq t'$**:** Since e doesn't access the store buffer of t', e' remains enabled even if executed before e. For the reverse, e' doesn't modify the value of any variable, therefore e remains enabled even if executed after e'. Since e' does not modify the program state, permuting e and e' leads to the same state.

4. $e = wr_t(x,v)$ **and** $e' = rd_{t'}(x,v')$ **where** $t \neq t'$**:** similar to the case (1).

5. $e = wr_t(x,v)$ **and** $e' = wr\text{-}com_t(x,v')$ **and** $\neg e \sim e'$**:** Since these two events do not correspond to each other, the store buffer contains at least one element in Θ_2. Therefore, adding and removing one element from it are commutative.

6. $e = wr_t(x,v)$ **and** $e' = wr\text{-}com_{t'}(x,v')$ **where** $t \neq t'$**:** Since e and e' add and respectively, remove an element from two different store buffers, they are commutative.

7. $e = wr_t(x,v)$ **and** $e' = wr_{t'}(x,v')$ **where** $t \neq t'$**:** Since e and e' add an element to two different store buffers, they are commutative.

8. $e = wr\text{-}com_t(x,v)$ **and** $e' = rd_t(x,v')$**:** If e' accesses the store buffer of t (i.e., η_1 contains a write issue $wr_t(x,v'')$ and η_2 the corresponding write-commit for that write-issue $wr\text{-}com_t(x,v'')$) then the two events commute because one accesses the store buffer and one the shared memory. Otherwise, e removes the last element from the store buffer and if e' executes before e it will access exactly that element and be equally enabled.

9. $e = wr\text{-}com_t(x,v)$ **and** $e' = rd_{t'}(x,v')$ **where** $t \neq t'$**:** similar to the case (2).

10. $e = wr\text{-}com_t(x,v)$ **and** $e' = wr_t(x,v')$ **and** $\neg e \sim e'$**:** similar to the case (5).

11. $e = wr\text{-}com_t(x,v)$ **and** $e' = wr_{t'}(x,v')$ **where** $t \neq t'$**:** similar to the case (6).

12. $e = fn_{t'}$ **and** $e' \in \mathbb{E}_t$ **where** $t \neq t'$**:** similar to the case (3).

B.3 Deciding SC-equivalence Using Event Graphs

We introduce a characterization of SC-equivalence based on event graphs. The *write-contraction* of an event-graph EG_η of a TSO-execution η is the graph EG^c_η where every node representing a write event $wr_t(x,v)$ is merged with the node representing the corresponding write-commit event $wr\text{-}com_t(x,v)$ (note that a contracted edge disappears and does not turn into a self-loop). Figure 2(b) pictures the write-contraction of the event-graph in Figure 2(a) (the dashed edges should be ignored).

Lemma 1. *The write-contraction of the event-graph EG_η of a TSO-execution η, built using $\circledcirc_{TSO}$, is acyclic.*

Proof. Let $e_1 = wr_{t_1}(x)$ and $e_2 = wr_{t_2}(y)$ be two write-issue events of η such that e_1 occurs before e_2 in η. We prove that merging these two write-issues with the corresponding write-commits doesn't introduce a cycle.

Since write-issue events are not conflicting with events of other threads, EG_η contains a path from e_1 to e_2 only if $t_1 = t_2$. Note that it is impossible to have a path from e_2 to e_1.

Let $e'_1 = wr\text{-}com_{t_1}(x)$ and $e'_2 = wr\text{-}com_{t_2}(y)$ be the write-commits corresponding to e_1 and e_2, respectively. Then, EG_η contains an edge from e'_1 to e'_2 and one from e_i to e'_i, for all $i \in \{1,2\}$. Since the event-graph EG_η is itself acyclic, it can not contain a path from e_2 to e_1, from e'_2 to e'_1, or from e'_i to e_i, where $i \in \{1,2\}$. Therefore, merging e_1 with e'_1 and e_2 with e'_2 could result in a cycle only if EG_η contains a path from e_2 to e'_1 or from e'_2 to e_1. The latter is clearly not possible because EG_η contains already a path from e_1 to e'_2 and EG_η is acyclic.

Now, assume by contradiction that EG_η contains a path from e_2 to e'_1. Then, EG_η must contain a path from e_2 to a read event $e_3 = rd_{t_3}(z)$ and a path from e_3 to e'_1.

By the definition of $\circledcirc_{TSO}$, we must have $t_1 = t_3$. This is because a write-issue can be connected to a read occurring before its corresponding write-commit only if the read is an event of the same thread. We must also have $z = y$. If e_2 and e_3 were not accessing the same variable, then they could be connected only if they were fence-separated which is not the case since e_3 occurs before e_2' or if e_2 was connected to another read $e = rd_{t_2}(y)$ and the latter connected to e_3. The latter scenario is not possible since by definition, e is a buffer read and thus, not it conflict with e_3.

Now, reads can be connected to write-commits of the same thread only if there exists another write-commit of a different thread that conflicts with both. Let $e_4 = wr_{t_1}(x)$ be the last write-issue on x of thread t_1 before e_3. Then, $e_4 = e_2$ or it is a write-issue that occurs after e_2 in η. Therefore, the write-commit corresponding to e_4 is either e_2' or it occurs after e_2' in η. By the definition of the inter-thread conflicts in $\circledcirc_{TSO}$, e_3 is not in conflict with any other write-commit of a different thread that occurs before the write-commit corresponding to e_4, and in particular, before e_2'. Therefore, EG_η can not contain a path from e_3 to e_2', which finishes the proof. □

The following result is a straightforward consequence of definitions.

Theorem 10. *A TSO-execution η is SC-equivalent iff the write-contraction of the event-graph EG^c_η, built using $\circledcirc_{TSO-po}$, is acyclic.*

Proof. The result follows from Theorem 1, by defining every pair of write and resp., write-commit events to be a transaction. □

For instance, the graph in Figure 2(b), including the dashed edges, is the write-contraction of the event-graph EG^c_η, built using $\circledcirc_{TSO-po}$, where η is the execution in Figure 2(a). This graph contains a cycle which shows that the execution is not SC-equivalent.

B.4 SC Programs

A program P is called an *SC-program* when for every $\sigma \in P$ every occurrence of a write event $wr_t(x, v)$ in σ is immediately followed by a fence event fn_t. The following result is a direct consequence of the definitions.

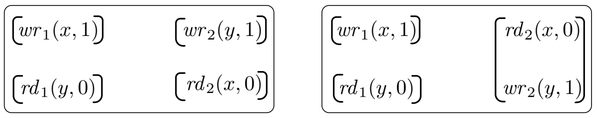
Theorem 11. *Let P be an SC-program. Then, P is TSO-serializable iff it is SC-serializable.*

B.5 Proofs of Section 4.2

Proof of Theorem 2 Let η be a TSO-execution of P. Since P is robust, there exists an execution η' that is a $\circledcirc_{TSO-po}$-preserving permutation of η and every write event of η' is immediately followed by the corresponding write-commit event. Note that η' has the same event-graph as η (when $\circledcirc_{TSO-po}$ or $\circledcirc_{TSO}$ is used as a conflict relation).

Let η'' be an SC-execution of P such that the projection of η' on $\mathbb{E}$ is exactly η''. (The existence of η'' is a direct consequence of the SC and TSO-semantics.)

By the definition of $\circledcirc_{SC}$ and $\circledcirc_{TSO}$, the write-contraction $EG^c_{\eta'}$ of the event-graph of η', built using $\circledcirc_{TSO}$, is a spanning sub-graph of $EG_{\eta''}$ (i.e., it has the same set of nodes but possibly fewer edges). The write-contraction $EG^c_{\eta'}$ may have fewer edges between events of the same thread, for instance writes and reads on different variables. Therefore, if the conflict graph induced by $EG_{\eta''}$ is acyclic then so is the conflict-graph induced by $EG^c_{\eta'}$. Since the latter is the same as the conflict graph of η, we conclude that η is TSO-serializable. □



The reverse of Theorem 2 doesn't hold. For instance, both programs above are TSO-serializable although the program in the left is not robust and the program in the right is not SC-serializable. The program in the left is TSO-serializable since every event is a transaction and events in the same thread are not in conflict, and it is not robust since intuitively, both reads don't see the value written by the other thread. The program in the right is TSO-serializable because the events in thread 1 are not in conflict while it is not SC-serializable since it admits only one execution where the events of thread 1 take place in between the two events of thread 2.

Proof of Theorem 3 ($\Leftarrow$) Direct consequence of Theorem 2.

($\Rightarrow$) Let η be a TSO execution of P. Assume by contradiction that η is not robust. By Theorem 10, the write-contraction G of the event-graph of η, built using $\circledcirc_{TSO\text{-}po}$, is cyclic, i.e., G contains a sequence of nodes $n_1, \ldots, n_k$ with $n_1 = n_k$ and $(n_i, n_{(i+1) \bmod k})$ an edge in G, for each $1 \le i \le k$. We show that for each $1 \le i \le k$, either n_i and $n_{(i+1) \bmod k}$ represent events of the same transaction or there exists a path from n'_i to $n'_{(i+1) \bmod k}$ in the event-graph EG_η of η, built using $\circledcirc_{TSO}$, where n_i and n'_i, resp., $n_{(i+1) \bmod k}$ and $n'_{(i+1) \bmod k}$, represent events of the same transaction. This implies that the conflict graph induced by EG_η is cyclic, which contradicts the hypothesis that P is TSO-serializable.

Suppose that n_i and $n_{(i+1) \bmod k}$ represent the events e and e' of different transactions. If e and e' are events of the same thread, by the definition of G, e occurs before e' in η. Since P is fenced, e and e' are fence-separated and EG_η contains a path from the node representing e to the node representing e'. If e and e' are events of different threads, then EG_η contains an edge from the node representing $\gamma(e)$ to the node representing $\gamma(e')$, where γ is the identity on read events and it maps write events to the corresponding write-commit events.

Now, assume by contradiction that P is not SC-serializable. Let η be an SC-execution of P which is not SC-serializable. By the definition of the TSO-semantics, there exists a TSO-execution η' of P where every write is immediately followed by the corresponding write-commit event such that the write-contraction of $EG^c_{\eta'}$, built using $\circledcirc_{TSO}$, is a spanning sub-graph of EG_η. We show however that the conflict graph $CG_{\eta'}$ equals CG_η, which implies that $CG_{\eta'}$ is also cyclic and that P is not TSO-serializable.

Let (n, n') be an edge of $CG_{\eta'}$ defined by two events e and e' belonging to two different transactions. By definition, e occurs before e' and $e \circledcirc_{TSO} e'$. If e and e' are events of the same thread, then $e \circledcirc_{SC} e'$ and the edge (n, n') occurs in CG_η as well. If e and e' are events of different threads, then $\delta(e) \circledcirc_{SC} \delta(e')$ where δ is the identity on read and write events and it maps write-commit events to the corresponding write events. Also, by the definition of η', $\delta(e)$ occurs before $\delta(e')$ in η, which implies that (n, n') is an edge of CG_η.

For the reverse, let (m, m') be an edge of CG_η defined by two events f and f' belonging to two different transactions. By definition, f occurs before f' and $f \circledcirc_{SC} f'$. If f and f' are events of the same thread, then they are separated by a fence event (since they belong to two different transactions and each transaction ends with a fence), and the edge (m, m') occurs in $CG_{\eta'}$ as well. If f and f' are events of different threads, then $\gamma(e) \circledcirc_{TSO} \gamma(e')$, and by the definition of η', $\gamma(e)$ occurs before $\gamma(e')$ in η'. Therefore, the edge (m, m') occurs in $CG_{\eta'}$ as well. □

C Proofs of Section 5

C.1 Proofs of Theorems 4 and 5

Theorems 4 and 5 are a direct consequence of the following two lemmas.

Lemma 2. *For every $\eta \in Execs(\tau)$, EG^c_η is a valid orientation of EG_τ.*

Proof. For every $\eta \in Execs(\tau)$, EG_τ contains exactly the same set of edges as EG^c_η but some of them are undirected. Then, by Lemma 1, EG^c_η is acyclic which finishes the proof. □

Lemma 3. *Let τ be a trace. For every acyclic orientation G of EG_τ, there exists an execution $\eta \in Execs(\tau)$ such that $EG^c_\eta = G$.*

Proof. We prove the result by induction on the size of τ. The base case is trivial.

Let $\tau = \tau' \cdot e$ be a trace with $Execs(\tau) \neq \emptyset$ and assume the result holds for the trace τ'. Note that the trace event-graph EG_τ contains all the edges of $EG_{\tau'}$ together with some undirected or directed edges towards e (by definition, we can not have directed edges starting in e).

Let G be an acyclic orientation of EG_τ. By definition, G contains an acyclic orientation G' of $EG_{\tau'}$. By the induction hypothesis, there exists an execution $\eta' \in Execs(\tau')$ such that $EG^c_{\eta'} = G'$. W.l.o.g. we assume that for every write-issue in η', the corresponding write-commit occurs also in η'. In the following, we assume that η' is an execution where values are present. We show that η' can be extended to an execution η such that $trace(\eta) = \tau$ and $EG^c_\eta = G$. Several cases are to be discussed.

Case e = $rd_t(x)$: Let $e_1 = wr\text{-}com_{t'}(x, v)$ with $t' \neq t$ be the first write-commit event in η' such that G contains an edge from e to the write-issue $wr_{t'}(x, v)$ corresponding to e_1. Note that the edges in G starting in e can end only in a write-commit of a different thread. Essentially, we want to define η by inserting the event $rd_t(x, v')$, for some v' to be determined in the following, in η' right before e_1. However, it may happen that η' contains write-issues and reads after e_1, and this would make the trace of η being different than τ. But, since G is acyclic we can assume w.l.o.g. that there is no write-issue or read after e_1. Then, in order to ensure that e is in conflict with e_1 it must not happen that e_1 is followed by a write-commit of thread t on x whose corresponding write-issue is before e. Here, we use again the fact that G is acyclic with implies that there is a path from the latter write-issue to e_1 but not the reverse. Assume by contradiction that there exists a path from e_1 to this write-issue. This would create a cycle with the edge from this write-issue to e (which exists because of the intra-thread conflicts in $\circledcirc_{TSO}$) and the edge from e to e_1. Therefore, e_1 is not followed by a write-commit of thread t. Let η be the execution obtained from η' by inserting the event $rd_t(x, v')$ right before e_1, where v' is the value written by the last write-commit before e_1 in η'. From the previous arguments, it follows that η is a valid TSO-execution and that $EG^c_\eta = G$.

Case e = $wr_t(x)$: Let $e_1 \in \mathbb{E}_{t'}$ with $t' \neq t$ be the event in η' such that G contains an edge from e to e_1, if e_1 is a read, or to the write-issue corresponding to e_1, if e_1 is a write-issue. Note that the edges in G starting in e can end only in an event of a different thread. Also, let $e_2 \in \mathbb{R}_{t''}$ with $t'' \neq t$ be the first read event in η' such that G contains an edge from e to e_2. Using the same reasoning as in the previous case, we can assume w.l.o.g. that there is no write-issue or read after e_1. Let η be the execution obtained from η' by inserting the two events $wr_t(x, v') \cdot wr\text{-}com_t(x, v')$ right before e_1, where v' is the value read by e_2 (if e_2 doesn't exist then we can use any value for v'). We have that η is a valid TSO-execution and $EG^c_\eta = G$.

Case e = fn_t: This case is trivial, and η can be defined by appending e to η'. □

D Proofs for Section 6

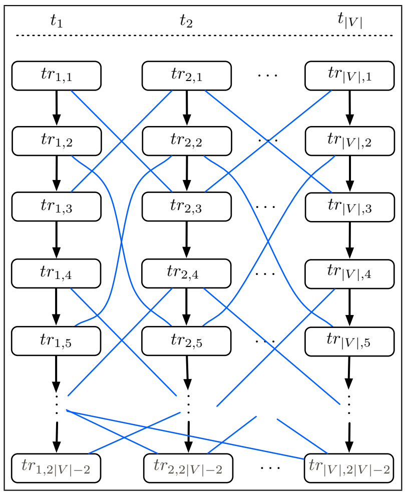
D.1 Proof of Theorem 6

Proof. First, clearly the problem is in NP since one can guess an orientation, check whether it is valid in polynomial time (equivalent to checking acyclicity of a directed graph), and if yes, check if the conflict graph that is induced by that orientation is acyclic in polynomial time.

Second, we sketch the reduction from the hamiltonian path problem. Given an arbitrary undirected graph G, the goal is to check whether $G = (V, E)$ has a hamiltonian path between two given nodes. We construct a trace τ_G such that G has a hamiltonian cycle iff τ_G is not TSO-serializable.

Let $V = \{v_1, \dots, v_n\}$ be the set of nodes in G. Without loss of generality, we assume is to the goal is to check whether there exists a hamiltonian path from v_1 to v_n. Trace τ_G consists of n threads, each executing a sequence of transactions. Each thread t_i corresponds to the node v_i of G, and executes transactions $tr_{i,1} \dots tr_{i,n_i}$ in that order. The number of transactions executed by each thread t_i corresponds to the edge degree of v_i, more specifically we have $n_i = 2(n-1) \times m_i$ where m_i is the edge degree of v_i in G (i.e. nodes that are connected to v_i by an edge). Each transaction contains exactly one event which is a write.

We organize the sequence of transactions of each thread into $2n-2$ groups. Each group contain m_i transactions. The figure on the right demonstrates the transaction groups in a (partial) conflict graph for τ_G, where $tr_{i,j}$ represents the jth transaction group of thread t_i. Intuitively, each transaction group of thread t_i captures, through conflict edges adjacent to its transactions, the edges adjacent to node v_i. Each transaction has a conflict with the next transaction of the same thread (black directed edges). Each transaction in group g where $g = 2, 4, \ldots, 2k$ ($k < n-2$) of thread t_i has a conflict with a transaction of thread t_j in group $g+3$ (blue undirected edges) iff edge (v_i, v_j) belongs to G. As an exception, each transaction in groups 1 and $2n-2$ of t_i has a conflict with a transaction of thread t_j in groups 3 and $2n-4$ respectively iff edge (v_i, v_j) belongs to G.



In order to achieve the arrangement of the conflict edges among transactions as described above, we use write events to a set of variables, with two transactions that are supposed to have a conflict to include a write to a common variable. Consider the fact that each edge $(v_i, v_j) \in E$ appears $n-1$ times in CG_τ between $n-1$ different pairs of transaction. For each such instance of such an edge, we need a new fresh variable to create that conflict and that conflict only. To this end, we make use of a set of variables $\{x_{g,\{i,j\}}\}$ to capture the instance of edge (v_i, v_j) for the gth transaction group. Based on this idea, each thread t_i executes a sequence events as seen on the right (with the transaction groups marked), under the assumption that node $v_i \in V$ of G is adjacent to nodes in $\{v_{j_1}, \ldots v_{j_m}\}$. For example, the $wr_{x_{1,\{i,j_1\}}}(t_i)$ event of the first transaction group of thread t_i is in conflict with the $wr_{x_{1,\{j_1,i\}}}(t_i)$ event of the second transaction group of thread t_{j_1}.

$wr_{t_i}(x_{1,\{i,j_1\}})$
$\ldots$
$wr_{t_i}(x_{1,\{i,j_m\}})$
$wr_{t_i}(x_{2,\{i,j_1\}})$
$\ldots$
$wr_{t_i}(x_{2,\{i,j_m\}})$
$wr_{t_i}(x_{1,\{i,j_1\}})$
$\ldots$
$wr_{t_i}(x_{1,\{i,j_m\}})$
$\ldots$
$wr_{t_i}(x_{2|V|-2,\{i,j_1\}})$
$\ldots$
$wr_{t_i}(x_{2|V|-2,\{i,j_m\}})$

Finally, we introduce three extra transactions. First, a transaction of a separate thread t_0 consisting of the two statements $wr_x(t_0) rd_y(t_0)$. Then, we add a new transaction consisting of a single event $wr_x(t_n)$ to the end of the first transaction group in thread t_n. Lastly, we add a new transaction with a single event $wr_y(t_1)$ to the beginning of the last transaction group of thread t_1. The idea is to have a directed conflict edge from event $rd_y(t_0)$ to event $wr_y(t_1)$ and an undirected conflict edge between $wr_x(t_0)$ and $wr_x(t_n)$, which connect this extra transaction the existing grid that we illustrated above.

To complete this proof, we have to make two arguments: (i) there is a trace τ_G that gives rise to the conflict graph with these transactions, and (ii) the trace is not TSO-serializable iff G has a hamiltonian cycle.

Define trace τ_G as serial execution of all threads from t_0 to t_n. It is easy to verify that, by definition of $\circledcirc_{TSO}$, all writes to common variables will end up with undirected conflict edges between them, and we will have a directed edge from event $rd_y(t_0)$ to event $wr_y(t_1)$, since t_0 is executed before t_1.

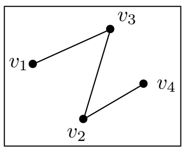
Let us assume that graph G has n nodes and a hamiltonian path $v_1 v_{i_2} \dots v_{i_{n-1}} v_n$. Based on the definition of transactions, there exist undirected conflict edges in the conflict graph of τ_G connecting transaction groups $tr_{1,2n-2}$ to $tr_{i_2,2n-4}$, $tr_{i_2,2n-4}$ to $tr_{i_3,2n-6}$, and so on, with finally a conflict edge connecting $tr_{i_{n-1},2}$ to $tr_{n,1}$. Moreover, there are two directed conflict edges connecting the singe transaction of t_0 to $tr_{1,2n-2}$, and $tr_{n,1}$ to t_0. Therefore, there exist a cycle in the conflict graph. It is left to argue that the directing of the undirected conflict edges in the orientation of this cycle does not create a cycle in the event graph of τ_G; in other words, these directed edges are a subset of a valid orientation of the event graph. This is straight forward, once one considers that the only (preexisting) directed edges are the ones between transaction of the same thread (in the same column), and the fact that the aforementioned cycle visits every column exactly once; therefore, the existing directed edges cannot participate in a cycle in the event graph together with the oriented edges of the cycle. We can conclude that trace τ_G is not TSO-serializable.

Now let us assume that τ_G is not TSO-serializable. This implies that there is an orientation of the undirected edges in the event graph of τ_G which induces a cycle in the conflict graph of τ_G. Let us focus on shortest such cycle, where by shortest we specifically mean one that visits as few threads as possible. We argue that this cycle in the conflict graph corresponds to a hamiltonian path from v_1 to v_n in G. First, consider that the only transaction with more than a single event in it (since we have to have at least one of those as part of non-serializability scenario) is the transaction of thread t_0 , which will have to be part of any cycle that witnesses a violation of TSO-serializability.

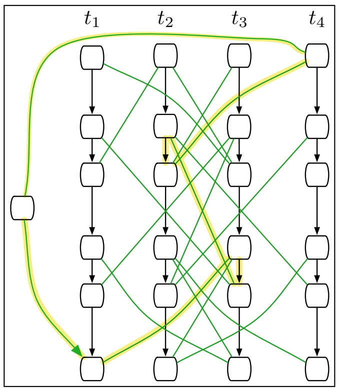
Second, consider that any cycle witness of TSO non-serializability may not visit any column of the conflict graph of τ_G more than once; otherwise, such a cycle either (i) creates a cycle over the event graph combined with the vertical inter-thread transaction edges of the column/thread that it visits twice (i.e. if the second time it visits a column is at an earlier transaction that the first time), or (ii) can be shortened (i.e. if the second time it visits a column is at a later transaction, then that part of the cycle can be replaced by a path through the thread/column).

Third, consider that any cycle witness of TSO non-serializability that starts from transaction of thread t_0 (and therefore ends in it) has to go from this transaction to thread t_1 (since the only outgoing edge out of t_0 goes to this transaction) and and make its way to the the end of transaction transaction group in t_n (which has the only other conflict edge that is connected to thread t_0). A witness cycle that goes from t_0 to t_1 and then ends in t_n (and back to t_0) only following (cross-thread) conflict edges that correspond to edges in graph G represents a path in graph G from v_1 to v_n.

Lastly, any cycle that visits no column more than once and has to go from the last transaction group of t_1 to the first transaction group of t_n has to go through every single thread. Remember that each conflict edge goes from a transaction in transaction group k of some thread to a transaction group $k - 2$ of another thread. In other words, with each cross-thread conflict edge, a cycle can make proceed up the trisection grid that we constructed by exactly two rows. Since there are $2n - 2$ rows in total, every thread $t_2, \dots, t_{n-1}$ has to be visited on the way from transaction group $2n - 2$ of thread t_1 to transaction group 1 of t_n. Note that Since the first and last transaction groups are exceptional, the path goes from the $2n - 2$ group of t_1 to the $2n - 3$ group of the next thread on the path, and then it skips two groups at a time until it makes its way to group 1 of t_n. To sum up, we have argued that this cycle visits every column of the event/transaction graph exactly once, which means the corresponding path visits each node G exactly once and hence is a hamiltonian path by definition.



To clarify the construction, consider an example for graph G as illustrated on the left. This graph has a hamiltonian path $v_1v_3v_2v_4$. Below, the conflict graph that is constructed for this graph is illustrated with threads $t_1 \dots t_4$ respectively corresponding to nodes $v_1 \dots v_4$.



The conflict graph is illustrated on the right where (for brevity) each box represents a transaction group. The edges to individual transactions are all illustrated as edges that go into their transaction group. For example, the a transaction in the last transaction group of thread t_1 is connected to a transaction of the 5th transaction group of threads t_3 since we have the edge (v_1, v_3) in the graph. And, so it goes. The extra transaction (of thread t_0, not marked) is only conflicted with the last transaction group of thread t_1 and the first transaction group of thread t_4. The witness for non-serializability in this conflict graph is the cycle that is highlighted in yellow, which corresponds to the hamiltonian path $v_1v_3v_2v_4$ in the graph.

D.2 Proofs of statements from Section 6.2

Proof. (of Proposition 3) It is straightforward to argue for (i), because if it is not true, then the conflict graph cycle becomes a cycle in the event graph EG_τ as well, and hence an invalid witness.

Let us assume that (ii) does not hold, i.e. there exists two transactions tr and tr' that both belong to some chain π, and yet there is an undirected edge between them in c. This means that, by definition, the cycle can be *simplified* further, and therefore is not simple.

Before we prove Proposition 4, the simple observation is that:

Proposition 6. *If c is a cycle over the the conflict graph induced by a valid orientation of EG_τ for some trace τ, and c' is the (one step) simplified version of τ, then c' is also a cycle in the conflict graph induced by the same valid orientation of EG_τ.*

Proof. (of proposition 6) Since c is a cycle in the conflict graph induced by a valid orientation of EG_τ, it means that all the (originally) undirected edges that are used in c are directed in the direction of the cycle in that valid orientation. A one step simplification, removes at least one of these (originally) undirected edges, and replaces it with a sequence of (originally directed edges). These directed edges are a constant in every valid orientation of EG_τ and therefore part of this one as well. c' is using fewer (originally) undirected edges, and more directed edges than c, if c is a cycle in this orientation of EG_τ, then so is c'.

which leads us to the proof of Proposition 4:

Proof. (of proposition 4) It is straightforward: keep simplifying the witness cycle until it cannot be simplified.

Proof. (of proposition 5) A direct implication of Proposition 3.

D.3 Extra material for Section 6.3

Proof. (of Theorem 12) ($\Leftarrow$) Clearly, the TSO-execution η' is not SC-equivalent. The PO-completion of $EG^c_{\eta'}$ contains a path from $wr_{t'}(x, v')$ to the last event of thread t in η, an edge from this last event of t to $rd_t(x, v)$, and an edge from $rd_t(x, v)$ to $wr_{t'}(x, v')$ (because the write-commit corresponding to $wr_{t'}(x, v')$ is executed after the write is issued and thus later than $rd_t(x, v)$). Therefore, P is not robust.

($\Rightarrow$) Let η' be a non SC-equivalent TSO-execution of P of minimal trace, i.e., for any prefix τ of $trace(\eta')$, all the TSO-executions of P of trace τ are SC-equivalent. W.l.o.g. we assume that η' contains a write-commit event for each write event. Let G be the PO-completion of $EG^c_{\eta'}$.

We show that there must exist a thread in η' which ends by executing a read event. Assume by contradiction that all threads end by executing a write event; let $wr_{t_1}(y, v_1)$ be the write event among the last writes of each thread that commits the last. Note that G contains no edge starting in $wr_{t_1}(y, v_1)$. Therefore, the execution η'' that doesn't execute $wr_{t_1}(y, v_1)$ is also a valid non SC-equivalent TSO-execution, which contradicts the minimality assumption.

Let $rd_t(x, v)$ be a read among the last reads of each thread. Then, $rd_t(x, v)$ must belong to a cycle of G. Otherwise, removing $rd_t(x, v)$ we get a valid TSO-execution of a smaller trace that is non SC-equivalent, which again contradicts the minimality assumption. Since η' doesn't contain other events of thread t that follow $rd_t(x, v)$ in the program order, the successor of $rd_t(x, v)$ in the cycle must be a write event $wr_{t'}(x, v')$ of a different thread. Also, the predecessor of $rd_t(x, v)$ in the cycle must be an event of thread t. Otherwise, if the predecessor of $rd_t(x, v)$ is a write event $wr_{t''}(x, v'')$, then by the definition of G, there is an edge from $wr_{t''}(x, v'')$ to $wr_{t'}(x, v')$, which shows that G has a cycle that doesn't include $rd_t(x, v)$.

Let η be the execution obtained from η' by deleting $rd_t(x, v)$. By the minimality assumption, η is SC-equivalent, therefore η can also be seen as an SC-execution. Since the edge from $rd_t(x, v)$ to $wr_{t'}(x, v')$ is included in $EG^c_{\eta'}$ as well, and $EG^c_{\eta'}$ is acyclic, $rd_t(x, v)$ can be inserted in η somewhere before $wr_{t'}(x, v')$ in order to obtain a TSO-execution that has the same write-contraction as η'.

D.4 Monitoring Robustness

Although robustness is a property of TSO-executions, we show that it can be checked efficiently (in polynomial time) on SC-executions. Essentially, we prove that minimal robustness violations [4] are formed of an SC-execution η and a read event which may not be necessarily enabled under the SC semantics but it is consistent with the program order in η.

For a sequence of events $\eta \in P$, a read event $rd_t(x,v)$ is *enabled* in η if η contains a write event $wr_{t''}(x,v)$ writing the value read by $rd_t(x,v)$, and $\eta \cdot rd_t(x,v) \in P$.

Theorem 12. *A program P is not robust iff there exists an SC-execution η of P, and a read event $rd_t(x,v)$ enabled in η such that*

1. *EG_η contains a path from a write event $wr_{t'}(x,v')$ with $t \neq t'$ to the last event of thread t in η, and*
2. *there exists a TSO-execution η' obtained from η [5] by inserting $rd_t(x,v)$ before $wr_{t'}(x,v')$.*

Proof. ($\Leftarrow$) Clearly, the TSO-execution η' is not SC-equivalent. The PO-completion of $EG^c_{\eta'}$ contains a path from $wr_{t'}(x,v')$ to the last event of thread t in η, an edge from this last event of t to $rd_t(x,v)$, and an edge from $rd_t(x,v)$ to $wr_{t'}(x,v')$ (because the write-commit corresponding to $wr_{t'}(x,v')$ is executed after the write is issued and thus later than $rd_t(x,v)$). Therefore, P is not robust.

($\Rightarrow$) Let η' be a non SC-equivalent TSO-execution of P of minimal trace, i.e., for any prefix τ of $trace(\eta')$, all the TSO-executions of P of trace τ are SC-equivalent. W.l.o.g. we assume that η' contains a write-commit event for each write event. Let G be the PO-completion of $EG^c_{\eta'}$.

We show that there must exist a thread in η' which ends by executing a read event. Assume by contradiction that all threads end by executing a write event; let $wr_{t_1}(y,v_1)$ be the write event among the last writes of each thread that commits the last. Note that G contains no edge starting in $wr_{t_1}(y,v_1)$. Therefore, the execution η'' that doesn't execute $wr_{t_1}(y,v_1)$ is also a valid non SC-equivalent TSO-execution, which contradicts the minimality assumption.

Let $rd_t(x,v)$ be a read among the last reads of each thread. Then, $rd_t(x,v)$ must belong to a cycle of G. Otherwise, removing $rd_t(x,v)$ we get a valid TSO-execution of a smaller trace that is non SC-equivalent, which again contradicts the minimality assumption. Since η' doesn't contain other events of thread t that follow $rd_t(x,v)$ in the program order, the successor of $rd_t(x,v)$ in the cycle must be a write event $wr_{t'}(x,v')$ of a different thread. Also, the predecessor of $rd_t(x,v)$ in the cycle must be an event of thread t. Otherwise, if the predecessor of $rd_t(x,v)$ is a write event $wr_{t''}(x,v'')$, then by the definition of G, there is an edge from $wr_{t''}(x,v'')$ to $wr_{t'}(x,v')$, which shows that G has a cycle that doesn't include $rd_t(x,v)$.

Let η be the execution obtained from η' by deleting $rd_t(x,v)$. By the minimality assumption, η is SC-equivalent, therefore η can also be seen as an SC-execution. Since

[4] Non SC-equivalent executions such that all of their prefixes are SC-equivalent

[5] An SC-execution is viewed as a TSO-execution where every write event is immediately followed by the corresponding write-commit event.

the edge from $rd_t(x,v)$ to $wr_{t'}(x,v')$ is included in $EG^c_{\eta'}$ as well, and $EG^c_{\eta'}$ is acyclic, $rd_t(x,v)$ can be inserted in η somewhere before $wr_{t'}(x,v')$ in order to obtain a TSO-execution that has the same write-contraction as η'. □

Given an SC-execution η and a read event $rd_t(x,v)$ enabled in η, checking the existence of a write event $wr_{t'}(x,v')$ satisfying the conditions of Theorem 12 can be done in polynomial time: one can enumerate all write events of η and all sequences η' obtained by adding $rd_t(x,v)$ to η. Whether a sequence η' is a valid TSO-execution is equivalent to the acyclicity of its event graph.

E Fenced Transactions

Proof. (of Theorem 12) (⇐) Clearly, the TSO-execution η' is not SC-equivalent. The PO-completion of $EG^c_{\eta'}$ contains a path from $wr_{t'}(x,v')$ to the last event of thread t in η, an edge from this last event of t to $rd_t(x,v)$, and an edge from $rd_t(x,v)$ to $wr_{t'}(x,v')$ (because the write-commit corresponding to $wr_{t'}(x,v')$ is executed after the write is issued and thus later than $rd_t(x,v)$). Therefore, P is not robust.

(⇒) Let η' be a non SC-equivalent TSO-execution of P of minimal trace, i.e., for any prefix τ of $trace(\eta')$, all the TSO-executions of P of trace τ are SC-equivalent. W.l.o.g. we assume that η' contains a write-commit event for each write event. Let G be the PO-completion of $EG^c_{\eta'}$.

We show that there must exist a thread in η' which ends by executing a read event. Assume by contradiction that all threads end by executing a write event; let $wr_{t_1}(y,v_1)$ be the write event among the last writes of each thread that commits the last. Note that G contains no edge starting in $wr_{t_1}(y,v_1)$. Therefore, the execution η'' that doesn't execute $wr_{t_1}(y,v_1)$ is also a valid non SC-equivalent TSO-execution, which contradicts the minimality assumption.

Let $rd_t(x,v)$ be a read among the last reads of each thread. Then, $rd_t(x,v)$ must belong to a cycle of G. Otherwise, removing $rd_t(x,v)$ we get a valid TSO-execution of a smaller trace that is non SC-equivalent, which again contradicts the minimality assumption. Since η' doesn't contain other events of thread t that follow $rd_t(x,v)$ in the program order, the successor of $rd_t(x,v)$ in the cycle must be a write event $wr_{t'}(x,v')$ of a different thread. Also, the predecessor of $rd_t(x,v)$ in the cycle must be an event of thread t. Otherwise, if the predecessor of $rd_t(x,v)$ is a write event $wr_{t''}(x,v'')$, then by the definition of G, there is an edge from $wr_{t''}(x,v'')$ to $wr_{t'}(x,v')$, which shows that G has a cycle that doesn't include $rd_t(x,v)$.

Let η be the execution obtained from η' by deleting $rd_t(x,v)$. By the minimality assumption, η is SC-equivalent, therefore η can also be seen as an SC-execution. Since the edge from $rd_t(x,v)$ to $wr_{t'}(x,v')$ is included in $EG^c_{\eta'}$ as well, and $EG^c_{\eta'}$ is acyclic, $rd_t(x,v)$ can be inserted in η somewhere before $wr_{t'}(x,v')$ in order to obtain a TSO-execution that has the same write-contraction as η'.