Explaining Relaxed Memory Models with Program Transformations

Ori Lahav and Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

Abstract. Weak memory models determine the behavior of concurrent programs. While they are often understood in terms of reorderings that the hardware or the compiler may perform, their formal definitions are typically given in a very different style—either axiomatic or operational. In this paper, we investigate to what extent weak behaviors of existing memory models can be fully explained in terms of reorderings and other program transformations. We prove that TSO is equivalent to a set of two local transformations over sequential consistency, but that non-multi-copy-atomic models (such as C11, Power and ARM) cannot be explained in terms of local transformations over sequential consistency. We then show that transformations over a basic non-multi-copy-atomic model account for the relaxed behaviors of (a large fragment of) Power, but that ARM's relaxed behaviors cannot be explained in a similar way. Our positive results may be used to simplify correctness of compilation proofs from a high-level language to TSO or Power.

1 Introduction

In a uniprocessor machine with a non-optimizing compiler, the semantics of a concurrent program is given by the set of interleavings of the memory accesses of its constituent threads (also known as *sequential consistency*). In multiprocessor machines and/or with optimizing compilers, however, more behaviors are possible; they are formally described by what is known as a *weak memory model*. Typical examples of such "weak" behaviors are in the SB (store buffering) and LB (load buffering) programs below:

$$\begin{array}{l} x := 1; \\ a := y; \ \textit{// 0} \end{array} \;\Big\|\; \begin{array}{l} y := 1; \\ b := x; \ \textit{// 0} \end{array} \qquad\qquad \begin{array}{l} a := x; \ \textit{// 1} \\ y := 1; \end{array} \;\Big\|\; \begin{array}{l} b := y; \ \textit{// 1} \\ x := 1; \end{array}$$

Assuming all variables are 0 initially, the weak behaviors in question are the ones in which a and b have the values mentioned in the program comments. In the SB program on the left this behavior is allowed by all existing weak memory models, and can be easily explained in terms of reordering: the hardware may execute the independent store to x and load from y in reverse order. Similarly, the behavior in the LB program on the right, which is allowed by some models, can be explained by reordering the load from x and the subsequent store to y. This explanation remains the same whether the hardware itself performs out-of-order execution, or the compiler, as a part of its optimization passes, performs these transformations, and the hardware actually runs a reordered program.

Formal memory models, however, choose a somewhat more complex explanation. Specifically, axiomatic memory model definitions construct a graph of memory access events for each program execution and impose various constraints on which store each load can read from. Similarly, operational definitions introduce concepts such as buffers, where the stores reside for some time before being propagated to other processors.

In this paper, we try to reconcile the formal model definitions with the more intuitive explanations in terms of program transformations. We consider the mainstream implemented memory models of TSO [16], C11's Release/Acquire fragment [7], Power [4], and ARM [12], and investigate whether their weak behaviors can be fully accounted for in terms of program transformations that are allowed in these models. In this endeavor, we have both positive and negative results to report on.

First, in §3, we show that the TSO memory model of the x86 and SPARC architectures can be precisely characterized in terms of two transformations over sequential consistency: write-read reordering and read-after-write elimination.

Second, in §4, we present examples showing that C11's Release/Acquire memory model cannot be defined in terms of a set of transformations over sequential consistency. This, in fact, holds for any memory model that allows non-multi-copy-atomic behaviors (where two different threads may observe a store of a third thread at different times), such as the full C11, Power, ARM, and Itanium models. Here, besides local instruction reorderings and eliminations we also consider the sequentialization transformation, that explains some non-multi-copy-atomic behaviors, but fails to account for all of them.

Next, in §5, we consider the Power memory model of Alglave et al. [4]. We show that the weak behaviors of this model, restricted to its fragment without "control fences" (Power's `isync` instructions), can be fully explained in terms of local reorderings over a stronger model that does not allow cycles in the entire program order together with the reads-from relation. In §6, we show that this is not possible for the ARM model: it allows some weak behaviors that cannot be explained in terms of local transformations over such stronger model.

Finally, in §7, we outline a possible application of the positive results of this paper, namely to simplify correctness of compilation proofs from a high-level language to either TSO or Power.

The proofs of this paper have also been formulated in Coq and are available at: `http://plv.mpi-sws.org/trns/`.

1.1 Related Work

Previous papers studied soundness of program transformations under different memory models (see, e.g., [15,18]), while we are interested in the "completeness" direction, namely whether program transformations completely characterize a memory model.

Concerning TSO, it has been assumed that it can be defined in terms of the two transformations mentioned above (e.g., in [2,9]), but to our knowledge a formal equivalence to the specification in [16] has not been established before.

In the broader context of proposing a fixed memory model for Java, Demange et al. [10] prove a very close result, relating a TSO-like machine and local transformations of executions. Nevertheless, one of the transformations of [10] does not correspond to a local program transformation (as it depends on the write that was read by each read). We also note that the proofs in [10] are based on an operational model, while we utilize an equivalent axiomatic presentation of TSO, that allows us to have simpler arguments.

Alglave et al. [3] provide a method for reducing verification under a weak memory model to a verification problem under sequential consistency. This approach follows a global program transformation of a completely different nature than ours, that uses additional data structures to simulate the threads' buffers.

Finally, assuming a sequentially consistent hardware, Ševčík [19] proves that a large class of compiler transformations respect the DRF guarantee (no weak behaviors for programs with no data races) and a basic non-thin-air guarantee (all read values are mentioned in some statement of the program). The results of the current paper allow the application of Ševčík's theorems for TSO, as it is fully explained by transformations that are already covered as compiler optimizations. For the other models, however, our negative results show that the DRF and non-thin-air guarantees do not follow immediately from Ševčík's theorems.

2 Preliminaries: Axiomatic Memory Model Definitions

In this section, we present the basic axiomatic way of defining memory models.

Basic notations. Given a binary relation R, $R^?$, R^+, and R^* respectively denote its reflexive, transitive, and reflexive-transitive closures. The inverse relation is denoted by R^{-1}. We denote by $R_1; R_2$ the left composition of two relations R_1, R_2. A relation R is called *acyclic* if R^+ is irreflexive. When R is a strict partial order, $R|_{\text{imm}}$ denotes the relation consisting of all *immediate* R*-edges*, i.e., pairs $\langle a, b\rangle \in R$ such that for every c, $\langle c, b\rangle \in R$ implies $\langle c, a\rangle \in R^?$, and $\langle a, c\rangle \in R$ implies $\langle b, c\rangle \in R^?$. Finally, we denote by $[A]$ the identity relation on a set A. In particular, $[A]; R; [B] = R \cap (A \times B)$.

We assume finite sets Tid, Loc, and Val of thread identifiers, locations, and values. We use i as a metavariable for thread identifiers, x, y, z for locations, and v for values. Axiomatic memory models associate a set of graphs (called *executions*) to every program. The nodes of these graphs are called *events*, and they are related by different kinds of edges.

Events. An *event* consists of an identifier (natural number), a thread identifier (or 0 for initialization events), and a *type*, that can be `R` ("read"), `W` ("write"), `U` ("atomic update"), or `F` ("fence"). For memory accesses (`R`, `W`, `U`) the event also contains the accessed location, as well as the read and/or written value. Events in each specific memory model may contain additional information (e.g., fence type or C11-style access ordering). We use $a, b, ...$ as metavariables for events.

The functions tid, typ, loc, val_r and val_w respectively return (when applicable) the thread identifier, type, location, read value, and written value of an event.

Notation 1. Given a relation R on events, $R|_x$ denotes the restriction of R to events accessing location x, and $R|_{loc}$ denotes the restriction of R to events accessing the same location (i.e., $R|_x = \{\langle a, b\rangle \in R \mid loc(a) = loc(b) = x\}$ and $R|_{loc} = \bigcup_{x \in loc} R|_x$).

Executions. An *execution* G consists of:[1]

1. a finite set $G.\mathtt{E}$ of events with distinct identifiers. This set always contains a set $G.\mathtt{E}_0$ of initialization events, consisting of one write event assigning the initial value for every location. We assume that all initial values are 0.
2. a binary relation $G.\mathtt{po}$, called *program order*, which is a disjoint union of relations $\{G.\mathtt{po}_i\}_{i \in \{0\} \cup \mathsf{Tid}}$, such that $G.\mathtt{po}_0 = G.\mathtt{E}_0 \times (G.\mathtt{E} \setminus G.\mathtt{E}_0)$, and for every $i \in \mathsf{Tid}$, the relation $G.\mathtt{po}_i$ is a strict total order on $\{a \in G.\mathtt{E} \mid tid(a) = i\}$.
3. a binary relation $G.\mathtt{rf}$, called *reads-from*, which is a set of reads-from edges. These are pairs $\langle a, b\rangle \in G.\mathtt{E} \times G.\mathtt{E}$ satisfying $a \neq b$, $typ(a) \in \{\mathtt{W}, \mathtt{U}\}$, $typ(b) \in \{\mathtt{R}, \mathtt{U}\}$, $loc(a) = loc(b)$, and $val_w(a) = val_r(b)$. It is required that an event cannot read from two different events (i.e., if $\langle a_1, b\rangle, \langle a_2, b\rangle \in G.\mathtt{rf}$ then $a_1 = a_2$).
4. a binary relation $G.\mathtt{mo}$, called *modification order*, whose properties vary from one model to another.

We identify an execution G with a set of tagged elements with the tags $\mathtt{E}$, $\mathtt{po}$, $\mathtt{rf}$, and $\mathtt{mo}$. For example, $\{\mathtt{E} : a, \mathtt{E} : b, \mathtt{po} : \langle a, b\rangle\}$ (where a and b are events) denotes an execution with $G.\mathtt{E} = \{a, b\}$, $G.\mathtt{po} = \{\langle a, b\rangle\}$, and $G.\mathtt{rf} = G.\mathtt{mo} = \emptyset$. Further, for a set E of events, $\{\mathtt{E} : E\}$ denotes the set $\{\mathtt{E} : e \mid e \in E\}$. A similar notation is used for the other tags, and it is particularly useful when writing expressions like $G \cup \{\mathtt{rf} : rf\}$ (that stand for the extension of an execution G with a set rf of reads-from edges). In addition, we denote by $G.\mathtt{T}$ ($\mathtt{T} \in \{\mathtt{R}, \mathtt{W}, \mathtt{U}, \mathtt{F}\}$) the set $\{e \in G.\mathtt{E} \mid typ(e) = \mathtt{T}\}$. We may also concatenate the event sets notations, and use a subscript to denote the accessed location (e.g., $G.\mathtt{RW} = G.\mathtt{R} \cup G.\mathtt{W}$ and $G.\mathtt{W}_x$ denotes all events $a \in G.\mathtt{W}$ with $loc(a) = x$). We omit the prefix "$G.$" when it is clear from the context.

The exact definition of the set of executions associated with a given program depends on the particular programming language and the memory model. Figure 1 provides an example. Note that in this initial stage the read values are not restricted whatsoever, and the reads-from relation $\mathtt{rf}$ and the modification order $\mathtt{mo}$ are still empty. We refer to such executions as *plain executions*.

Now, the main part of a memory model is the specification of which of the executions of a program P are allowed. The first requirement, agreed by all memory models, is that every read should be justified by some write. Such executions will be called *complete* (formally, G is complete if for every $b \in \mathtt{RU}$, we

[1] Different models may include some additional relations (e.g., a dependency relation between events is used for Power, see §5).

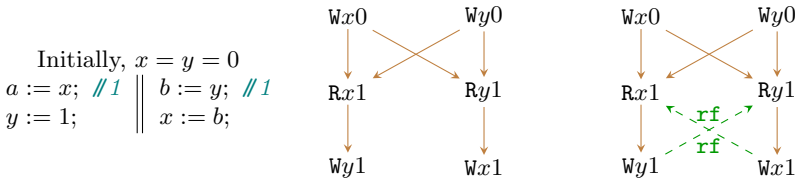

Fig. 1. A program together with one of its plain executions, and a complete execution extending the plain one. Solid arrows denote the transitive reduction of po (i.e., omitting edges implied by transitivity). The variables a, b are local registers, and these are not mentioned in executions.

have $\langle a, b\rangle \in \mathtt{rf}$ for some event a). To filter out disallowed executions among the complete ones, each memory model M defines a notion of when an execution G is M-*coherent*, which is typically defined with the help of a few *derived relations*, and places several restrictions on the rf and mo relations. Then, we say that a plain execution G is M-*consistent* if there exist relations rf and mo such that $G \cup \{\mathtt{rf} : rf\} \cup \{\mathtt{mo} : mo\}$ is a complete and M-coherent execution. The semantics of a program under M is taken to be the set of its M-consistent executions.

2.1 Sequential Consistency

As a simple instance of this framework, we define sequential consistency (SC). There are multiple equivalent axiomatic presentations of SC. Here, we choose one that is specifically tailored for studying the relation to TSO in §3.

Definition 1. An execution G is SC-*coherent* if the following hold:

1. mo is a strict total order on WUF.
2. hb is irreflexive.
3. mo;hb is irreflexive.
4. rb;hb is irreflexive.
5. rb;mo is irreflexive.
6. rb;mo;hb is irreflexive.

where:

- $\mathtt{hb} = (\mathtt{po} \cup \mathtt{rf})^+$ (*happens-before*)
- $\mathtt{rb} = (\mathtt{rf}^{-1}; \mathtt{mo}|_{loc}) \setminus [\mathtt{E}]$ (*reads-before*)

Intuitively speaking, mo denotes the order in which stores happen in the memory, hb represents a causality order between events, and rb says that a read is before a write to the same location if it reads from a prior write in modification order. Figure 2 depicts the conditions for SC-coherence. It can be easily seen that the weak behavior of the SB program in the introduction is disallowed under SC due to condition 6 (together with conditions 1 and 3), while the one of the LB program is disallowed under SC due to condition 2.

Proposition 1. *Our notion of* SC*-consistency defines sequential consistency [14].*

Proof (Outline). The SC-coherence definition above guarantees that $(\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb})^+$ is a partial order. Following [17], any total order extending this partial order defines an interleaving of the memory accesses,

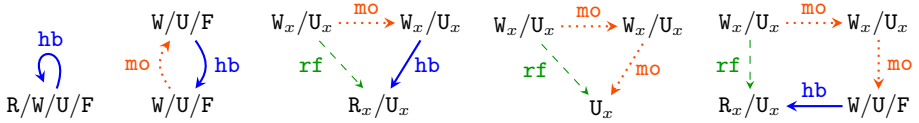

Fig. 2. Illustration of SC's irreflexivity conditions.

which agrees with po and ensures that every read/update obtains its value from the last previous write/update to the same location. For the converse, one can take mo to be the restriction of the interleaving order to WUF. □

3 TSO

In this section, we study the TSO (*total store ordering*) memory model provided by the x86 and SPARC architectures. Its common presentation is operational: on top of usual interleaving semantics, each hardware thread has a queue of pending memory writes (called *store buffer*), that non-deterministically propagate (in order) to a main memory [16]. When a thread reads from a location x, it obtains the value of the last write to x that appears in its buffer, or the value of x in the memory if no such write exists. Fence instructions flush the whole buffer into the main memory, and atomic updates perform flush, read, write, and flush again in one atomic step.

To simplify our formal development, we use an *axiomatic* definition of TSO from [13]. By [16, Theorem 3] and [13, Theorem 5], this definition is equivalent to the operational one.[2]

Definition 2. An execution G is TSO-*coherent* if the following hold:

1. mo is a strict total order on WUF.
2. hb is irreflexive.
3. mo;hb is irreflexive.
4. rb;hb is irreflexive.
5. rb;mo is irreflexive.
6. rb;mo;rfe;po is irreflexive.
7. rb;mo;[UF];po is irreflexive.

where hb and rb are defined as in Definition 1, and:

– $\mathtt{rfe} = \mathtt{rf} \setminus \mathtt{po}$ (*external reads-from*)

The first five conditions of the TSO-coherence definition are the same as those of SC-coherence. Conditions 6 and 7 are relaxations of condition 6 in the SC-coherence definition (depicted in Fig. 3). Intuitively speaking, mo is the order in which the writes propagate to the main memory of the TSO-machine, and the two conditions ensure that a read from the main memory can only read from the last write (to the same location) that was propagated.

Next, we present the key lemma that identifies more precisely the difference between TSO and SC.

[2] Lahav et al. [13] treat fence instructions as syntactic sugar for atomic updates of a distinguished location. Here, we have fences as primitive instructions that induce fence events in the program executions.

Fig. 3. Illustration of the alternative irreflexivity conditions of TSO. Requiring an external reads-from edge or an update/fence (that flush the store buffer) immediately after the second mo-edge ensures that events a, b and c are in main memory at the point d is executed and therefore the rf-edge $\langle a, d\rangle$ corresponds to reading from the main memory, rather than from the local buffer.

Lemma 1. *Irreflexivity of the following relation suffices to guarantee that a* TSO*-coherent complete execution G is also* SC*-coherent:*

$$\mathtt{rb}; \mathtt{mo}; [\mathtt{W}]; (po' \cup rfi); [\mathtt{R}]; \mathtt{po}^?$$

where $po' = \mathtt{po}|_{imm} \setminus (\mathtt{po}|_{loc} \cup (\mathtt{mo}; \mathtt{rf}))$, *and* $rfi = \mathtt{po}|_{imm} \cap \mathtt{rf}$.

Now, we turn to our first main positive result, showing that TSO is precisely characterized by write-read reordering and read-after-write elimination over sequential consistency. First, we define write-read reordering.

Definition 3 (Write-read reordering). For an execution G and events a and b, ReorderWR(G, a, b) is the execution G' obtained from G by inverting the program order from a to b, i.e., it is given by: $G'.\mathtt{po} = (G.\mathtt{po} \setminus \{\langle a, b\rangle\}) \cup \{\langle b, a\rangle\}$, and $G'.\mathtt{C} = G.\mathtt{C}$ for every other component C. ReorderWR(G, a, b) is defined only when $\langle a, b\rangle \in [\mathtt{W}]; \mathtt{po}|_{\mathrm{imm}}; [\mathtt{R}]$ and $loc(a) \neq loc(b)$.

The condition $\langle a, b\rangle \in \mathtt{po}|_{\mathrm{imm}}$ guarantees that only adjacent accesses are reordered. This transformation does not inspect the rf and mo components of G, and thus also applies to plain executions. This fact ensures that it corresponds to a program transformation. Note that additional rewriting are sometimes needed in order to make two adjacent accesses in the program's execution to be adjacent instructions in the program. For example, to reorder the store $x := 1$ and load $a := y$ in the following program, one can first rewrite the program as follows:

$$\begin{array}{l} x := 1; \\ \mathtt{if}\ b\ \mathtt{then}\ a := y; \\ \mathtt{else}\ y := 2; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} x := 1; \\ \mathtt{if}\ b\ \mathtt{then}\ a := y; \\ \mathtt{if}\ \neg b\ \mathtt{then}\ y := 2; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} \mathtt{if}\ b\ \mathtt{then}\ a := y; \\ x := 1; \\ \mathtt{if}\ \neg b\ \mathtt{then}\ y := 2; \end{array}$$

Similarly, reordering of local register assignments and unfolding of loops may be necessary. To relate reorderings on plain executions to reorderings on (non-straightline) programs, one should assume that these transformations may be freely applied.

Remark 1. Demange et al. [10, Definition 5.3] introduce a related *write-read-read* reordering, which allows to reorder a read before a write and a sequence of subsequent reads *reading from that write*. This reordering does not correspond to a local program transformation, as it inspects the reads-from relation, that is not available in plain executions, and cannot be inferred from the program code.

The second transformation we use, called *WR-elimination*, replaces a read from some location directly after a write to that location by the value written by the write (e.g., $x := 1; a := x \rightsquigarrow x := 1; a := 1$). Again, we place conditions to ensure that the execution transformation corresponds to a program one.

Definition 4 (Read-after-write elimination). For an execution G and events a and b, $\mathsf{RemoveWR}(G, a, b)$ is the execution G' obtained by removing b from G, i.e., G' is given by: $G'.\mathtt{E} = G.\mathtt{E} \setminus \{b\}$, and $G'.\mathtt{C} = G.\mathtt{C} \cap (G'.\mathtt{E} \times G'.\mathtt{E})$ for every other component $\mathtt{C}$. $\mathsf{RemoveWR}(G, a, b)$ is defined only when $\langle a, b\rangle \in [\mathtt{W}]; \mathtt{po}|_{\mathrm{imm}}; [\mathtt{R}]$, $loc(a) = loc(b)$, and $val_w(a) = val_r(b)$.

Note that WR-reordering is unsound under SC (the reordered program may exhibit behaviors that are not possible in the original program). WR-elimination, however, is sound under SC. Nevertheless, WR-elimination is needed below, since, by removing a read access, it may create new opportunities for WR-reordering.

We can now state the main theorem of this section. We write $G \rightsquigarrow_{\mathsf{TSO}} G'$ if $G' = \mathsf{ReorderWR}(G, a, b)$ or $G' = \mathsf{RemoveWR}(G, a, b)$ for some a, b.

Theorem 1. *A plain execution G is TSO-consistent iff $G \rightsquigarrow^*_{\mathsf{TSO}} G'$ for some SC-consistent execution G'.*

The rest of this section is devoted to the proof of Theorem 1. First, the soundness of the two transformations under TSO is well-known.

Proposition 2. *If $G \rightsquigarrow_{\mathsf{TSO}} G'$ and G' is TSO-consistent, then so is G.*

The converse is not generally true. It does (trivially) hold for eliminations:

Proposition 3. *Let G be a complete and TSO-coherent execution. Then, $\mathsf{RemoveWR}(G, a, b)$, if defined, is complete and TSO-coherent.*

Proof. Removing a read event from an execution reduces all relations mentioned in Definition 2, and hence preserves their irreflexivity. □

Proposition 4. *Let G be a complete and TSO-coherent execution. Let a, b such that $\mathsf{ReorderWR}(G, a, b)$ is defined. If $\langle a, b\rangle \notin \mathtt{mo}; \mathtt{rf}$, then $\mathsf{ReorderWR}(G, a, b)$ is complete and TSO-coherent.*

Proposition 5. *Suppose that G is complete and TSO-coherent but not SC-coherent. Then, $G \rightsquigarrow_{\mathsf{TSO}} G'$ for some TSO-coherent complete execution G'.*

Proof. By Lemma 1, there must exist events $a \in \mathtt{W}$ and $b \in \mathtt{R}$, such that $\langle a, b\rangle \in po' \cup rfi$, (where po' and rfi are the relations defined in Lemma 1). Now, if $\langle a, b\rangle \in po'$, we can apply WR-reordering, and take $G' = \mathsf{ReorderWR}(G, a, b)$. By Proposition 4, G' is complete and TSO-coherent. Otherwise, $\langle a, b\rangle \in rfi$. In this case, we can apply WR-elimination, and take $G' = \mathsf{RemoveWR}(G, a, b)$. By Proposition 3, G' is complete and TSO-coherent. □

We can now prove the main theorem.

Proof (of Theorem 1). The right-to-left direction is easily proven using Proposition 2, by induction on the number of transformations in the sequence deriving G' from G (note that the base case trivially holds as SC-consistency implies TSO-consistency). We prove the converse. Given two plain executions G and G', we write $G' < G$ if either $G'.\mathtt{E} \subset G.\mathtt{E}$ or ($G'.\mathtt{E} = G.\mathtt{E}$ and $[\mathtt{W}]; G'.\mathtt{po}; [\mathtt{R}] \subset G.\mathtt{po}$). Clearly, $<$ is a well-founded partial order. We prove the claim by induction on G (using $<$ on the set of all executions). Let G be an execution, and assume that the claim holds for all $G' < G$. Suppose that G is TSO-consistent. If G is SC-consistent, then we are done. Otherwise, by Proposition 5, $G \rightsquigarrow_{\mathsf{TSO}} G'$ for some TSO-consistent execution G'. It is easy to see that we have $G' < G$. By the induction hypothesis, $G' \rightsquigarrow^*_{\mathsf{TSO}} G''$ for some SC-consistent execution G''. Then, we also have $G \rightsquigarrow^*_{\mathsf{TSO}} G''$. □

4 Release-Acquire

Next, we turn to the *non-multi-copy-atomic* memory model (i.e., two different threads may detect a store by a third thread at different times) of C11's Release/Acquire. By RA we refer to the memory model of C11, as defined in [7], restricted only to programs in which all reads are acquire reads, writes are release writes, and atomic updates are acquire-release read-modify-writes (RMWs). We further assume that this model has no fence events. Fence instructions under RA, as proposed in [13], can be implemented using atomic updates to an otherwise unused distinguished location.

Definition 5. An execution G is RA-*coherent* if the following hold:

1. $\mathtt{mo}$ is a disjoint union of relations $\{\mathtt{mo}_x\}_{x \in \mathsf{Loc}}$, such that each relation $\mathtt{mo}_x$ is a strict total order on $\mathtt{W}_x\mathtt{U}_x$.
2. $\mathtt{hb}$ is irreflexive.
3. $\mathtt{mo}; \mathtt{hb}$ is irreflexive.
4. $\mathtt{rb}; \mathtt{hb}$ is irreflexive.
5. $\mathtt{rb}; \mathtt{mo}$ is irreflexive.

where $\mathtt{hb}$ and $\mathtt{rb}$ are defined as in Definition 1.

Note that unlike SC and TSO, the relation $\mathtt{mo}$ in the RA-coherence definition relates only events accessing the same location. The following IRIW (independent reads, independent writes) program shows that RA is more than local program transformations over SC.

$$\begin{array}{l} a := x; \ \mathit{//1} \\ b := y; \ \mathit{//0} \end{array} \Bigg\| \ x := 1; \ \Bigg\| \ y := 1; \ \Bigg\| \ \begin{array}{l} c := y; \ \mathit{//1} \\ d := x; \ \mathit{//0} \end{array}$$

The behavior in question is allowed under RA, although RA forbids any reorderings and eliminations in this program. In particular, reordering of reads is unsound under RA (because RA supports message passing). One may observe that this behavior can be explained if we add *sequentialization* to the set of program transformations, to allow transformations of the form $C_1 \parallel C_2 \rightsquigarrow C_1; C_2$

and $C_1; C_1' \parallel C_2 \rightsquigarrow C_1; C_2; C_1'$. By sequentializing the $x := 1$ store instruction to be before its corresponding load we obtain the program on the left:

$$\begin{array}{l} x := 1; \\ a := x; \ /\!/ 1 \\ b := y; \ /\!/ 0 \end{array} \Bigg\| \; y := 1; \Bigg\| \begin{array}{l} c := y; \ /\!/ 1 \\ d := x; \ /\!/ 0 \end{array} \quad \rightsquigarrow \quad \begin{array}{l} b := y; \ /\!/ 0 \\ x := 1; \\ a := 1; \ /\!/ 1 \end{array} \Bigg\| \; y := 1; \Bigg\| \begin{array}{l} c := y; \ /\!/ 1 \\ d := x; \ /\!/ 0 \end{array}$$

Now, this behavior is allowed under SC after applying a WR-elimination followed by a WR-reordering in the first thread (obtaining the program on the right). At the execution level, sequentialization increases its po component, and it is sound under RA, simply because it may only increase all the relations mentioned in Definition 5. Note that, unlike RA and SC, sequentialization is unsound under TSO: while the weak behavior of the IRIW program is forbidden under TSO, it is allowed after applying sequentialization. Other examples show that sequentialization is unsound under Power and ARM as well [1].

Even with sequentialization, however, we cannot reduce RA to SC, as the following program demonstrates.

$$\begin{array}{l} y := 1; \\ x := 1; \\ a := x; \ /\!/ 3 \\ b := z; \ /\!/ 0 \end{array} \Bigg\| \; x := 3; \Bigg\| \begin{array}{l} z := 1; \\ x := 2; \\ c := x; \ /\!/ 3 \\ d := y; \ /\!/ 0 \end{array}$$

The behavior in question is allowed by RA (by putting the write of 3 to x after the two other writes to x in mo). In this program, no sound reorderings or eliminations can explain the weak behavior, and, moreover, any possible sequentialization will forbid this behavior.

In fact, the above example applies also to SRA, the stronger version of RA studied in [13], obtained by requiring that mo is a total order on WU (as in TSO), instead of condition 1 in Definition 5 (but still excluding irreflexivity of rb; mo; hb that is required for SC-coherence). As RA, SRA forbids thread-local transformations in this program, but allows its weak behavior.

5 Power

In this section, we study the model provided by the Power architecture, using the recent axiomatic model by Alglave et al. [4]. Here, our positive result is somewhat limited:

1. Like RA, the Power model is non-multi-copy-atomic, and thus, it cannot be explained using transformations over SC. Instead, we explain Power's weak behaviors starting from a stronger non-multi-copy-atomic model, that, we believe, is easier to understand and reason about, than the Power model.
2. Power's control fence (`isync`) is used to enforce a stronger ordering on memory reads. Its special effect cannot be accounted for by program transformations (see example in [1]). Hence, we only consider here a restricted fragment

of the Power model, that has two types of fence events: `sync` ("strong fence") and `lwsync` ("lightweight fence"). $G.\mathtt{F}_{\mathtt{sync}}$ and $G.\mathtt{F}_{\mathtt{lwsync}}$ respectively denote the set of events $a \in G.\mathtt{E}$ with $typ(a)$ being `sync` and `lwsync`.

The Power architecture performs out-of-order and speculative execution, but respects dependencies between instructions. Accordingly, Power's axiomatic executions keep track of additional relations for data, address and control dependency between events, that are derived directly from the program syntax. For example, in all executions of $a := x;\ y := a$, we will have a data dependency edge from the read event to the write event, since the load and store use the same register a. Here, we include all sort of dependencies in one relation between events, denoted by deps. Note that we always have $\mathtt{deps} \subseteq \mathtt{po}$, and that only read and update events may have outgoing dependency edges.

Based on deps, the Power model employs a relation called *preserved program order*, denoted ppo, which is a subset of po that is guaranteed to be preserved. The exact definition of ppo is somewhat intricate (we refer the reader to [4] for details). For our purposes, it suffices to use the following properties of ppo:

$$[\mathtt{RU}]; (\mathtt{deps} \cup \mathtt{po}|_{loc})^+; [\mathtt{WU}] \subseteq \mathtt{ppo} \qquad \text{(ppo-lower-bound)}$$

$$\mathtt{ppo} \cap \mathtt{po}|_{\mathrm{imm}} \subseteq (\mathtt{deps} \cup \mathtt{po}|_{loc})^+ \qquad \text{(ppo-upper-bound)}$$

Remark 2. Atomic updates are not considered in the text of [4]. In the accompanying herd simulator, they are modeled using pairs of a read and a write events related by an atomicity relation. Here we follow a different approach, model atomic updates using a single update event, and adapt herd's model accordingly. Thus we are only considering Power programs in which `lwarx` and `stwcx` appear in separate adjacent pairs. These instructions are used to implement locks and compare-and-swap commands, and they indeed appear only in such pairs when following the intended mapping of programs to Power [6].

Using the preserved program order, Power-coherence is defined as follows (the reader is referred to [4] for further explanations and details).

Definition 6. An execution G is Power-*coherent* if the following hold:

1. mo is a disjoint union of relations $\{\mathtt{mo}_x\}_{x \in \mathsf{Loc}}$, such that each relation $\mathtt{mo}_x$ is a strict total order on $\mathtt{W}_x\mathtt{U}_x$.
2. hb is acyclic. (*no-thin-air*)
3. $\mathtt{po}|_x \cup \mathtt{rf} \cup \mathtt{rb} \cup \mathtt{mo}$ is acyclic for every $x \in \mathsf{Loc}$. (*SC-per-loc*)
4. $\mathtt{rbe}; \mathtt{prop}; \mathtt{hb}^*$ is irreflexive. (*observation*)
5. $\mathtt{mo} \cup \mathtt{prop}$ is acyclic. (*propagation*)
6. $\mathtt{rb}; \mathtt{mo}$ is irreflexive. (*atomicity*)
7. $\mathtt{mo}; [\mathtt{U}]; \mathtt{po}; [\mathtt{U}]$ is acyclic.

where rb is defined as in Definition 1, and:

- $\mathtt{sync} = \mathtt{po}; [\mathtt{F}_{\mathtt{sync}}]; \mathtt{po}$ and $\mathtt{lwsync} = \mathtt{po}; [\mathtt{F}_{\mathtt{lwsync}}]; \mathtt{po}$
- $\mathtt{fence} = \mathtt{sync} \cup ([\mathtt{RU}]; \mathtt{lwsync}; [\mathtt{RWU}] \cup ([\mathtt{W}]; \mathtt{lwsync}; [\mathtt{WU}]))$ (*fence order*)

- $\mathtt{rfe} = \mathtt{rf} \setminus \mathtt{po}$ and $\mathtt{rbe} = \mathtt{rb} \setminus \mathtt{po}$ (*external reads-from and reads-before*)
- $\mathtt{hb} = \mathtt{ppo} \cup \mathtt{fence} \cup \mathtt{rfe}$ (*happens-before*)
- $\mathtt{prop}_1 = [\mathtt{WU}]; \mathtt{rfe}^?; \mathtt{fence}; \mathtt{hb}^*; [\mathtt{WU}]$
- $\mathtt{prop}_2 = ((\mathtt{mo} \cup \mathtt{rb}) \setminus \mathtt{po})^?; \mathtt{rfe}^?; (\mathtt{fence}; \mathtt{hb}^*)^?; \mathtt{sync}; \mathtt{hb}^*$
- $\mathtt{prop} = \mathtt{prop}_1 \cup \mathtt{prop}_2$ (*propagation relation*)

In particular, Power allows the weak behavior in the LB program presented in the introduction. Indeed, unlike the other models discussed above, the Power model does not generally forbid $(\mathtt{po} \cup \mathtt{rf})$-cycles. Thus, Power-consistent executions are not "prefix-closed"— it may happen that G is Power-consistent, but some $\mathtt{po}$-prefix of G is not. This makes reasoning about the Power model extremely difficult, because it precludes the understanding a program in terms of its partial executions, and forbids proofs by induction on $\mathtt{po}$-prefixes of an execution. In the following we show that all weak behaviors of Power can be explained by starting from a stronger prefix-closed model, and applying various reorderings of independent adjacent memory accesses to different locations. First, we define the stronger model.

Definition 7. An execution G is SPower-*coherent* if it is Power-coherent and $\mathtt{po} \cup \mathtt{rf}$ is acyclic.

Note that this additional acyclicity condition is a strengthening of the *"no-thin-air"* condition in Definition 6. A similar strengthening for the C11 memory model was suggested in [8], as a straghitforward solution to the "out-of-thin-air" problem (see also [5]). In addition, the same acyclicity condition was assumed for proving soundness of FSL [11] (a program logic for C11's relaxed accesses).

Next, we turn to relate Power and SPower using general reorderings of adjacent memory accesses.

Definition 8 (Reordering). For an execution G and events a and b, $\mathsf{Reorder}(G, a, b)$ is the execution G' obtained from G by inverting the program order from a to b, i.e., it is given by: $G'.\mathtt{po} = (G.\mathtt{po} \setminus \{\langle a, b\rangle\}) \cup \{\langle b, a\rangle\}$, and $G'.\mathtt{C} = G.\mathtt{C}$ for every other component $\mathtt{C}$. $\mathsf{Reorder}(G, a, b)$ is defined only when $a, b \notin \mathtt{F}$ and $\langle a, b\rangle \in \mathtt{po}|_{\mathrm{imm}} \setminus \mathtt{deps}$, and $loc(a) \neq loc(b)$.

We write $G \rightsquigarrow_{\mathsf{Power}} G'$ if $G' = \mathsf{Reorder}(G, a, b)$ for some a, b.

Proposition 6. *Suppose that $G \rightsquigarrow_{\mathsf{Power}} G'$. Then, G is* Power*-coherent iff G' is* Power*-coherent.*

The following observation is useful in the proof below.

Proposition 7. *The following relation is acyclic in* Power*-coherent executions:*

$$\mathtt{deps} \cup \mathtt{po}|_{loc} \cup (\mathtt{po}; [\mathtt{F}]) \cup ([\mathtt{F}]; \mathtt{po}) \cup \mathtt{rfe}$$

Theorem 2. *A plain execution G is* Power*-consistent iff $G \rightsquigarrow^*_{\mathsf{Power}} G'$ for some* SPower*-consistent execution G'.*

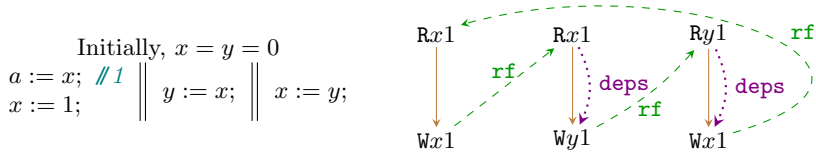

Fig. 4. A weak behavior of ARM, that is not explained by program transformations.

Proof. The right-to-left direction is proven by induction using Proposition 6. We prove the converse. Let G be a Power-consistent plain execution, and let rf and mo be relations such that $G_0 = G \cup \{\mathtt{rf} : rf\} \cup \{\mathtt{mo} : mo\}$ is complete and Power-coherent. Let S be a total strict order on $\mathtt{E}$ extending the relation R given in Proposition 7. Let G' be the execution given by $G'.\mathtt{po} = \bigcup_{i \in \mathsf{Tid}} \{\langle a, b\rangle \in S \mid tid(a) = tid(b) = i\} \cup (\mathtt{E}_0 \times \mathtt{E})$ (where $\mathtt{E}_0$ is the set of initialization events in G), while all other components of G' are as in G. It is easy to see that $G \rightsquigarrow^*_{\mathsf{Power}} G'$. Indeed, recall that a list L of elements totally ordered by $<$ can be sorted by repeatedly swapping adjacent unordered elements $l_i > l_{i+1}$ (as done in "bubble sort"). Since $R \subseteq S$, no reordering step from G to G' will reorder dependent events, events accessing the same location, or fence events. Now, Proposition 6 ensures that $G'_0 = G' \cup \{\mathtt{rf} : rf\} \cup \{\mathtt{mo} : mo\}$ is complete and Power-coherent. To see that it is also SPower-coherent, note that $(\mathtt{E} \setminus \mathtt{E}_0); (G'_0.\mathtt{po} \cup G'_0.\mathtt{rf}) \subseteq S$. □

Remark 3. Note that the reordering operation does not affect the dependency relation. To allow this, and still understand reordering on the program level, we actually consider a slightly weaker model of Power than the one in [4], that do not carry control dependencies across branches. For instance, in a program like $a := y; (\mathtt{if}\ a\ \mathtt{then}\ z := 1); x := 1$, which can be a result of reordering of the stores to x and z in $a := y; x := 1; (\mathtt{if}\ a\ \mathtt{then}\ z := 1)$, we will not have a control dependency between the load of y and the store to x.

6 ARM

We now turn to the ARM architecture and show that it cannot be modeled by any sequence of sound reorderings and eliminations over a basic model satisfying $(\mathtt{po} \cup \mathtt{rf})$-acyclicity.

Consider the program in Fig. 4. Note that no reorderings or eliminations can be applied to this program. In the second and the third threads, reordering is forbidden because of the dependency between the load and the subsequent store. On the first thread, there is no dependency, but since the load and the store access the same location, their reordering is generally unsound, as it allows the load to read from the (originally subsequent) store. Moreover, this program cannot return $a = 1$ under a $(\mathtt{po} \cup \mathtt{rf})$-acyclic model, because the only instance of the constant 1 in the program occurs after the load of x in the first thread. Nevertheless, this behavior is allowed under both the axiomatic ARMv7 model of Alglave et al. [4] and the ARMv8 Flowing and POP models of Flur et al. [12].

The axiomatic ARMv7 model [4] is the same as the Power model presented in Section 5, with the only difference being the definition of ppo (preserved program order). In particular, this model does not satisfy (ppo-lower-bound) because $[\mathtt{RU}];\mathtt{po}|_{loc};[\mathtt{WU}] \not\subseteq \mathtt{ppo}$. Hence, the first thread's program order in the example above is not included in ppo, and there is no happens-before cycle. For the same reason, our proof for Power does not carry over to ARM.

In the ARMv8 Flowing model [12], consider the topology where the first two threads share a queue and the third thread is separate. The following execution is possible: (1) the first thread issues a load request from x and immediately commits the $x := 1$ store; (2) the second thread then issues a load request from x, which gets satisfied by the $x := 1$ store, and then (3) issues a store to $y := 1$; (4) the store to y gets reordered with the x-accesses, and flows to the third thread; (5) the third thread then loads $y = 1$, and also issues a store $x := 1$, which flows to the memory; (6) the load of x flows to the next level and gets satisfied by the $x := 1$ store of the third thread; and (7) finally the $x := 1$ store of the first thread also flows to the next level. The POP model is strictly weaker than the Flowing model, and thus also allows this outcome.

7 Application: Correctness of Compilation

Our theorems can be useful to prove correctness of compilation of a programming language with some memory model (such as C11) for the TSO and Power architectures. We outline this idea in a more abstract setting.

Let $[\![P]\!]_{\mathsf{M}}$ denote the possible behaviors of a program P under memory model M. A formal definition of a behavior can be given using a distinguished *world* location, whose values are inspected by an external observer. Assume some compilation scheme from a source language C to a target language A (i.e., a mapping of C instructions to sequences of A ones), and let $\mathtt{compile}(P_C)$ denote the program P_A obtained by applying this scheme on a program P_C. Further, assume memory models M_C and M_A (we do not assume that M_C has an axiomatic presentation; an operational one would work out the same). Correct compilation is expressed by:

$$\forall P_C.\ [\![\mathtt{compile}(P_C)]\!]_{\mathsf{M}_A} \subseteq [\![P_C]\!]_{\mathsf{M}_C}.$$

Applied on the program level, the $(2 \Rightarrow 1)$ directions of Theorems 1 and 2 provide us with the following:

$$\forall P_C.\ [\![\mathtt{compile}(P_C)]\!]_{\mathsf{M}_A} \subseteq \bigcup\{[\![P'_A]\!]_{\mathsf{SM}_A} \mid P'_A \ s.t.\ \mathtt{compile}(P_C) \rightsquigarrow^*_{\mathsf{M}_A} P'_A\},$$

where SM_A is a stronger model than M_A (SC for TSO and SPower for Power). Then, correctness of compilation easily follows from the following two conditions. First, compilation should be correct for the strong model SM_A:

$$\forall P_C.\ [\![\mathtt{compile}(P_C)]\!]_{\mathsf{SM}_A} \subseteq [\![P_C]\!]_{\mathsf{M}_C}.$$

Second, there should exist a set of source program transformations, described by $\rightsquigarrow_{\mathsf{M}_C}$, that (i) is sound for M_C, i.e.,

$$\forall P_C, P'_C.\ P_C \rightsquigarrow_{\mathsf{M}_C} P'_C \implies [\![P'_C]\!]_{\mathsf{M}_C} \subseteq [\![P_C]\!]_{\mathsf{M}_C};$$

and (*ii*) captures all target transformations from a compiled program:

$$\forall P_C, P'_A.\ \texttt{compile}(P_C) \rightsquigarrow_{\mathsf{M}_A} P'_A \implies \exists P'_C.\ \texttt{compile}(P'_C) = P'_A \wedge P_C \rightsquigarrow^*_{\mathsf{M}_C} P'_C.$$

For TSO meeting the first condition is trivial, because sequential consistency is the strongest model. In the case of Power, proving this property for SPower is easier than for Power. Roughly speaking, to show that behaviors of SPower are allowed by a model M_C would require less "features" of M_C, and can be done by induction on $(\texttt{po} \cup \texttt{rf})^+$ in SPower-coherent executions.

Fulfilling the second requirement is typically easy, because the source language, its memory model, and the mapping of its statements to processors are often explicitly designed to enable such transformations. In fact, when one aims to validate an *optimizing* compiler, the first part of the second requirement should be anyway established. For example, consider the compilation of C11 to TSO. Here, we need to show that WR-reordering and WR-elimination on compiled code could be done by C11-sound transformations on corresponding instructions of the source. Indeed, the mapping of C11 accesses to TSO instructions (see [7]) ensures that any adjacent WR-pair results from adjacent C11 accesses with access ordering strictly weaker than `sc` (sequential consistent accesses). Reordering and eliminations in this case is known to be sound under the C11 memory model [18].

8 Conclusion

In this paper, we have shown that the TSO memory model and (a substantial fragment of) the Power memory model can be defined by a set of reorderings and eliminations starting from a stronger and simpler memory model. Nevertheless, the counterexamples in Sections 4 and 6 suggest that there is more to weak memory consistency than just instruction reorderings and eliminations.

We further sketched a possible application of the alternative characterizations of TSO and Power: proofs of compilation correctness can be simplified by using the soundness of local transformations in the source language. To follow this approach in a formal proof of correctness of a compiler, however, further work is required to formulate precisely the syntactic transformations in the target programming language. In the future, we also plan to investigate the application of these characterizations for proving soundness of program logics with respect to TSO and Power.

Acknowledgments. We would like to thank the FM'16 reviewers for their feedback. This research was supported by an ERC Consolidator Grant for the project "RustBelt", funded under Horizon 2020 grant agreement no. 683289.

References

1. Coq development for this paper and further supplementary material, available at the following URL: `http://plv.mpi-sws.org/trns/`
2. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. Computer 29(12), 66–76 (Dec 1996)
3. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Proceedings of the 22Nd European Conference on Programming Languages and Systems. pp. 512–532. ESOP'13, Springer-Verlag, Berlin, Heidelberg (2013)
4. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. 36(2), 7:1–7:74 (Jul 2014)
5. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: Proceedings of the 24th European Symposium on Programming. pp. 283–307. ESOP 2015 (2015)
6. Batty, M., Memarian, K., Owens, S., Sarkar, S., Sewell, P.: Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 509–520. POPL '12, ACM, New York, NY, USA (2012)
7. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 55–66. POPL '11, ACM, New York, NY, USA (2011)
8. Boehm, H.J., Demsky, B.: Outlawing ghosts: Avoiding out-of-thin-air results. In: Proceedings of the Workshop on Memory Systems Performance and Correctness. pp. 7:1–7:6. MSPC '14, ACM, New York, NY, USA (2014)
9. Burckhardt, S., Musuvathi, M., Singh, V.: Verifying local transformations on relaxed memory models. In: Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction. pp. 104–123. CC'10/ETAPS'10, Springer-Verlag, Berlin, Heidelberg (2010)
10. Demange, D., Laporte, V., Zhao, L., Jagannathan, S., Pichardie, D., Vitek, J.: Plan B: A buffered memory model for Java. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 329–342. POPL '13, ACM, New York, NY, USA (2013)
11. Doko, M., Vafeiadis, V.: A program logic for C11 memory fences. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. Lecture Notes in Computer Science, vol. 9583, pp. 413–430. Springer (2016)
12. Flur, S., Gray, K.E., Pulte, C., Sarkar, S., Sezgin, A., Maranget, L., Deacon, W., Sewell, P.: Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 608–621. POPL 2016, ACM, New York, NY, USA (2016)
13. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 649–662. POPL 2016, ACM, New York, NY, USA (2016)
14. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers 28(9), 690–691 (1979)

15. Morisset, R., Pawan, P., Zappa Nardelli, F.: Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 187–196. PLDI '13, ACM, New York, NY, USA (2013)
16. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics. pp. 391–407. TPHOLs '09, Springer-Verlag, Berlin, Heidelberg (2009)
17. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. 10(2), 282–312 (Apr 1988)
18. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Zappa Nardelli, F.: Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 209–220. POPL '15, ACM, New York, NY, USA (2015)
19. Ševčík, J.: Safe optimisations for shared-memory concurrent programs. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 306–316. PLDI '11, ACM, New York, NY, USA (2011)

A Appendix: Additional Proofs

We provide proofs for the claims in the paper. The proofs have also been formulated in Coq and are available at: `http://plv.mpi-sws.org/trns/`.

Proof (of Proposition 1). Suppose that the conditions of Definition 1 hold. First, we claim that $R = \mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic. Indeed, consider a cycle in R of minimal length. Since $\mathtt{mo}$ is total on $\mathtt{WUF}$, such a cycle cannot contain more than two events in $\mathtt{WUF}$ (otherwise, the cycle can be shortened). Irreflexivity of $\mathtt{hb}$ and $\mathtt{rb};\mathtt{hb}$ implies that it must contain an $\mathtt{mo}$-edge $\langle a, b\rangle$ (note that $[\mathtt{U};\mathtt{rb} \subseteq \mathtt{mo}$ since $\mathtt{rb};\mathtt{mo}$ is irreflexive). Since it cannot contain additional events from $\mathtt{WUF}$ besides a and b, we have $\langle b, a\rangle \in \mathtt{hb};\mathtt{rb}^?$. However, this contradicts the fact that $\mathtt{rb};\mathtt{mo};\mathtt{hb}$ and $\mathtt{mo};\mathtt{hb}$ are irreflexive.

Now, let S be some total order extending R. Then, $\mathtt{po} \subseteq S$, and to prove the claim, it suffices to show that in S every read/update reads from the last write/update to the same location. Suppose otherwise, and let $\langle a, c\rangle \in \mathtt{rf}$ such that $\langle a, b\rangle, \langle b, c\rangle \in S$ for some write/update event b with $loc(b) = loc(a)$. Since $\mathtt{mo} \subseteq S$ and $\mathtt{mo}$ is total on $\mathtt{WU}$, we must have $\langle a, b\rangle \in \mathtt{mo}$. We obtain that $\langle c, b\rangle \in \mathtt{rb}$, which contradicts the fact that $\langle b, c\rangle \in S$.

The converse is easier: one can take $\mathtt{mo}$ to be the restriction of the interleaving order of the memory accesses to $\mathtt{WUF}$. We leave the details to the reader. □

The following simplification of Definition 2 is useful in the sequel.

Proposition 8. *An equivalent definition of* TSO*-coherence is obtained by replacing conditions 2,3,4 by irreflexivity of the following: (i)* $\mathtt{mo};\mathtt{rfe}$*, (ii)* $\mathtt{mo}^?;\mathtt{rfe}^?;\mathtt{po}$*, and (iii)* $\mathtt{rb};\mathtt{rfe}^?;\mathtt{po}^?$.

Proof. One direction is obvious (since $\mathtt{rfe}, \mathtt{rfe}^?;\mathtt{po} \subseteq \mathtt{hb}$). For the converse, suppose that G satisfies the alternative conditions. We show that $\mathtt{hb}$, $\mathtt{mo};\mathtt{hb}$, and $\mathtt{rb};\mathtt{hb}$ are irreflexive.

First, we claim that $R = \mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo}$ is acyclic. Indeed, consider a cycle in R of minimal length. Since $\mathtt{mo}$ is total on $\mathtt{WUF}$, such a cycle cannot contain more than two events in $\mathtt{WUF}$. Since $\mathtt{rf} \subseteq \mathtt{WU} \times \mathtt{E}$, $\mathtt{mo} \subseteq \mathtt{WUF} \times \mathtt{WUF}$, $\mathtt{po}$ is transitive, and the three relations are irreflexive, it follows that either $\mathtt{mo};\mathtt{rfe}$ or $\mathtt{mo}^?;\mathtt{rfe}^?;\mathtt{po}$ is not irreflexive.

It follows that $\mathtt{hb}$ and $\mathtt{mo};\mathtt{hb}$ are irreflexive. It remains to show that $\mathtt{rb};\mathtt{hb}$ is irreflexive. Suppose for contradiction that $\langle a, b\rangle \in \mathtt{rb}$ and $\langle b, a\rangle \in \mathtt{hb}$. Then, since $\mathtt{rb};\mathtt{rfe}^?;\mathtt{po}^?$ is irreflexive, there exists some c such that $\langle b, c\rangle \in \mathtt{hb}$, and $\langle c, a\rangle \in \mathtt{rfe};\mathtt{po}^?$. Since R is acyclic, it follows that $\langle b, c\rangle \in \mathtt{mo}$. Since $\langle a, b\rangle \in \mathtt{rb}$, we cannot have $\langle c, a\rangle \in \mathtt{rfe}$. Hence, $\langle c, a\rangle \in \mathtt{rfe};\mathtt{po}$, and this contradicts the irreflexivity of $\mathtt{rb};\mathtt{mo};\mathtt{rfe};\mathtt{po}$. □

Proof (of Lemma 1). Suppose that G is complete and TSO-coherent but not SC-coherent. Then, condition 6 of Definition 1 must not hold, that is: $\langle a, b\rangle \in \mathtt{rb}$,

$\langle b, c\rangle \in \texttt{mo}$, and $\langle c, a\rangle \in \texttt{hb}$ for some a, b, c. As shown in the proof of Proposition 8, TSO-coherence ensures that $R = \texttt{po} \cup \texttt{rf} \cup \texttt{mo}$ is acyclic. Note that it follows that a cannot be an update event. Indeed, otherwise, we would have $\langle b, a\rangle \in \texttt{mo}$, which contradicts the fact that $\texttt{rb};\texttt{mo}$ is irreflexive.

Now, consider an R-path of maximal length from c to a, and let d be the last write/update/fence event on this path before a, and e be its immediate successor on this path. Then, we have $\langle e, a\rangle \in ([\texttt{R}]; R)^*$. Hence, $e \in \texttt{R}$ and $\langle e, a\rangle \in \texttt{po}^?$. In addition, since $\texttt{mo}$ is total on $\texttt{WUF}$ and $\langle b, d\rangle \in R^+$, we have $\langle b, d\rangle \in \texttt{mo}$.

We claim that we cannot have $\langle d, e\rangle \in \texttt{rfe}$. Indeed, if $a \neq e$, then this follows since TSO-coherence ensures that $\texttt{rb};\texttt{mo};\texttt{rfe};\texttt{po}$ is irreflexive. Alternatively, if $a = e$, then $\langle d, a\rangle \in \texttt{rfe}$ and $\langle a, b\rangle \in \texttt{rb}$ imply $\langle d, b\rangle \in \texttt{mo}$, which contradicts $\langle b, d\rangle \in \texttt{mo}$.

Hence, we have $\langle d, e\rangle \in \texttt{po}$, and since $\langle d, e\rangle$ is an edge on a maximal path, we also have $\langle d, e\rangle \in po|_{\text{imm}}$ and $\langle d, e\rangle \notin \texttt{mo};\texttt{rf}$. Now, TSO-coherence ensures that $\texttt{rb};\texttt{mo};[\texttt{UF}];\texttt{po}$ is irreflexive, and so $d \notin \texttt{UF}$. Hence, $d \in \texttt{W}$. It remains to show that either $\langle d, e\rangle \notin \texttt{po}|_{loc}$ or $\langle d, e\rangle \in \texttt{rf}$. Suppose that $\langle d, e\rangle \in \texttt{po}|_{loc}$. Since G is complete, there must exist some d' such that $\langle d', e\rangle \in \texttt{rf}$. Now, if $\langle d', d\rangle \in \texttt{mo}$, we would have $\langle e, d\rangle \in \texttt{rb}$, which contradicts the fact that $\texttt{rb};\texttt{hb}$ is irreflexive. Additionally, $\langle d, d'\rangle \in \texttt{mo}$ is impossible, since $\langle d, e\rangle \notin \texttt{mo};\texttt{rf}$. Hence, since $\texttt{mo}$ is total on $\texttt{WU}$, we must have $d' = d$ and so $\langle d, e\rangle \in \texttt{rf}$. □

Proof (of Proposition 2). Let $G' \in \{\mathsf{Reorder}(G, a, b), \mathsf{RemoveWR}(G, a, b)\}$ for some $a \in \texttt{W}$ and $b \in \texttt{R}$. Suppose that G' is TSO-consistent, and let rf and mo such that $G'_0 = G' \cup \{\texttt{rf} : rf\} \cup \{\texttt{mo} : mo\}$ is complete and TSO-coherent. We show that G is TSO-consistent. Consider the two cases:

- $G' = \mathsf{Reorder}(G, a, b)$. We show that $G_0 = G \cup \{\texttt{rf} : rf\} \cup \{\texttt{mo} : mo\}$ is TSO-coherent (clearly, it is complete). By definition, $G.\texttt{po} = (G'.\texttt{po} \setminus \{\langle b, a\rangle\}) \cup \{\langle a, b\rangle\}$, and $G.\texttt{C} = G'.\texttt{C}$ for every other component $\texttt{C}$. Using Proposition 8, we have to prove irreflexivity of the following relations: (i) $\texttt{mo};\texttt{rfe}$, (ii) $\texttt{mo}^?;\texttt{rfe}^?;G.\texttt{po}$, (iii) $\texttt{rb};\texttt{rfe}^?;G.\texttt{po}^?$, (iv) $\texttt{rb};\texttt{mo}$, (v) $\texttt{rb};\texttt{mo};\texttt{rfe};G.\texttt{po}$, and (vi) $\texttt{rb};\texttt{mo};[\texttt{UF}];G.\texttt{po}$. Irreflexivity of (i) and (iv) is trivial since they concern only $\texttt{mo}$, $\texttt{rfe}$, and $\texttt{rb}$ which are identical in G and G'. Similarly: (v) and (vi) are irreflexive, observing that $[\texttt{RUF}];G.\texttt{po} \subseteq [\texttt{RUF}];G'.\texttt{po}$; (iii) is irreflexive since $G.\texttt{po}|_{loc} \subseteq G'.\texttt{po}|_{loc}$, and (ii) is irreflexive since $G.\texttt{po};[\texttt{WUF}] \subseteq G'.\texttt{po};[\texttt{WUF}]$;
- $G' = \mathsf{RemoveWR}(G, a, b)$. We show that $G_0 = G \cup \{\texttt{rf} : rf \cup \{\langle a, b\rangle\}\} \cup \{\texttt{mo} : mo\}$ is TSO-coherent (clearly, it is complete). Thus, we prove irreflexivity of the following relations (note that $G_0.\texttt{mo} = G'_0.\texttt{mo}$ and $G_0.\texttt{rfe} = G'_0.\texttt{rfe}$, so $G_0.\texttt{mo};G_0.\texttt{rfe}$ is irreflexive because so is $G'_0.\texttt{mo};G'_0.\texttt{rfe}$; we remove the "$G_0.$" or "$G'_0.$" prefix for $\texttt{mo}$ and $\texttt{rfe}$):
  1. $\texttt{rfe};G_0.\texttt{po}$: irreflexive since $\texttt{rfe};G_0.\texttt{po};[\texttt{WU}] \subseteq \texttt{rfe};G'_0.\texttt{po}$, and $G'_0.\texttt{hb}$ is irreflexive.

2. $\mathtt{mo}; G_0.\mathtt{po}$: irreflexive since $[\mathtt{WUF}]; G_0.\mathtt{po}; [\mathtt{WUF}] \subseteq G'_0.\mathtt{po}$, and $\mathtt{mo}; G'_0.\mathtt{hb}$ is irreflexive.
3. $\mathtt{mo}; \mathtt{rfe}; G_0.\mathtt{po}$: irreflexive since $\mathtt{rfe}; G_0.\mathtt{po}; [\mathtt{WUF}] \subseteq \mathtt{rfe}; G'_0.\mathtt{po}$, and $\mathtt{mo}; G'_0.\mathtt{hb}$ is irreflexive.
4. $G_0.\mathtt{rb}$: irreflexive since $G_0.\mathtt{rb} \subseteq G'_0.\mathtt{rb} \cup (\{b\} \times G.\mathtt{E})$ and $\langle b, b\rangle \notin G_0.\mathtt{rb}$ (since $b \in \mathtt{R}$).
5. $G_0.\mathtt{rb}; \mathtt{rfe}$: irreflexive since $G_0.\mathtt{rb} \subseteq G'_0.\mathtt{rb} \cup (\{b\} \times G.\mathtt{E})$ and $\langle c, b\rangle \notin G_0.\mathtt{rfe}$ for every $c \in G.\mathtt{E}$.
6. $G_0.\mathtt{rb}; G_0.\mathtt{po}$: To see this suppose that $\langle c, d\rangle \in G_0.\mathtt{rb}$ and $\langle d, c\rangle \in G_0.\mathtt{po}$. Hence, $d \in \mathtt{WU}$, and so $d \neq b$. If $c \neq b$, we also have obtain $\langle c, d\rangle \in G'_0.\mathtt{rb}$ and $\langle d, c\rangle \in G'_0.\mathtt{po}$, which contradicts the fact that $G'_0.\mathtt{rb}; G'_0.\mathtt{hb}$ is irreflexive. Suppose that $c = b$. Then, by definition, since $\langle a, b\rangle \in G_0.\mathtt{rf}$, we have $\langle a, d\rangle \in \mathtt{mo}$. Hence, $a \neq d$, and since $\langle d, b\rangle \in G_0.\mathtt{po}$ and $\langle a, b\rangle \in G.\mathtt{po}|_{\mathrm{imm}}$ we also have $\langle d, a\rangle \in G'_0.\mathtt{po}$. This contradicts the irreflexivity of $G'_0.\mathtt{mo}; G'_0.\mathtt{hb}$.
7. $G_0.\mathtt{rb}; \mathtt{rfe}; G_0.\mathtt{po}$: irreflexive since

$$\mathtt{rfe}; G_0.\mathtt{po}; G_0.\mathtt{rb} \subseteq \mathtt{rfe}; ((G'_0.\mathtt{po}; G'_0.\mathtt{rb}) \cup (G'_0.\mathtt{po}; \mathtt{mo})),$$

   and both $\mathtt{rb}; G'_0.\mathtt{hb}$ and $\mathtt{mo}; G'_0.\mathtt{hb}$ are irreflexive.
8. $G_0.\mathtt{rb}; \mathtt{mo}$: holds since $[\mathtt{U}]; G_0.\mathtt{rb} = [\mathtt{U}]; G'_0.\mathtt{rb}$, and $G'_0.\mathtt{rb}; \mathtt{mo}$ is irreflexive.
9. $G_0.\mathtt{rb}; \mathtt{mo}; \mathtt{rfe}; G_0.\mathtt{po}$: irreflexive since

$$\mathtt{rfe}; G_0.\mathtt{po}; G_0.\mathtt{rb} \subseteq \mathtt{rfe}; ((G'_0.\mathtt{po}; G'_0.\mathtt{rb}) \cup (G'_0.\mathtt{po}; \mathtt{mo})),$$

   and both $G'_0.\mathtt{rb}; \mathtt{mo}; \mathtt{rfe}; G'_0.\mathtt{po}$ and $\mathtt{mo}; G'_0.\mathtt{hb}$ are irreflexive.
10. $G_0.\mathtt{rb}; \mathtt{mo}; [\mathtt{UF}]; G_0.\mathtt{po}$: irreflexive since

$$[\mathtt{UF}]; G_0.\mathtt{po}; G_0.\mathtt{rb} \subseteq [\mathtt{UF}]; ((G'_0.\mathtt{po}; G'_0.\mathtt{rb}) \cup (G'_0.\mathtt{po}; \mathtt{mo})),$$

    and both $G'_0.\mathtt{rb}; \mathtt{mo}; [\mathtt{UF}]; G'_0.\mathtt{po}$ and $\mathtt{mo}; G'_0.\mathtt{hb}$ are irreflexive. □

Proof (of Proposition 4). Let $G' = \mathsf{ReorderWR}(G, a, b)$. Then, $G'.\mathtt{po} = (G.\mathtt{po} \setminus \{\langle a, b\rangle\}) \cup \{\langle b, a\rangle\}$, and $G'.\mathtt{C} = G.\mathtt{C}$ for every other component $\mathtt{C}$. Assuming $\langle a, b\rangle \notin \mathtt{mo}; \mathtt{rf}$, it is easy to apply Proposition 8 to prove that G' is TSO-coherent. First, since $loc(a) \neq loc(b)$, we cannot have $\langle a, b\rangle \in \mathtt{rf} \cup \mathtt{rb}; \mathtt{rf}^?$, and so $\mathtt{rfe}^?; G'.\mathtt{po}$ and $\mathtt{rb}; \mathtt{rfe}^?; G'.\mathtt{po}^?$ are irreflexive. Second, irreflexivity of $\mathtt{mo}; \mathtt{rfe}^?; G'.\mathtt{po}$ follows from the facts that $\mathtt{mo}; \mathtt{rfe}^?; G.\mathtt{po}$ is irreflexive and $\langle a, b\rangle \notin \mathtt{mo}; \mathtt{rf}^?$. Finally, irreflexivity of $\mathtt{rb}; \mathtt{mo}; \mathtt{rfe}; G'.\mathtt{po}$ and $\mathtt{rb}; \mathtt{mo}; [\mathtt{UF}]; G'.\mathtt{po}$ follows from $G'.\mathtt{po}; [\mathtt{RU}] \subseteq G'.\mathtt{po}$ and the irreflexivity of $\mathtt{rb}; \mathtt{mo}; \mathtt{rfe}; G.\mathtt{po}$ and $\mathtt{rb}; \mathtt{mo}; [\mathtt{UF}]; G.\mathtt{po}$. □

Proof (of Proposition 6). Let $G' = \mathsf{Reorder}(G, a, b)$ for some a and b with $loc(a) \neq loc(b)$. Then, $G'.\mathtt{po} = (G.\mathtt{po} \setminus \{\langle a, b\rangle\}) \cup \{\langle b, a\rangle\}$, and $G'.\mathtt{C} = G.\mathtt{C}$ for every other component $\mathtt{C}$. Note that each of the relations mentioned in Definition 6 is identical in G and G'. In particular, since $loc(a) \neq loc(b)$ and $\langle a, b\rangle \notin \mathtt{deps}$, assuming (ppo-upper-bound), we have $\langle a, b\rangle, \langle b, a\rangle \notin G.\mathtt{ppo} \cup G'.\mathtt{ppo}$. Hence, G is Power-coherent iff G' is Power-coherent. □

Proof (of Proposition 7). Let G be a Power-coherent execution, and let $T = \mathtt{deps} \cup \mathtt{po}|_{loc} \cup (\mathtt{po}; [\mathtt{F}]) \cup ([\mathtt{F}]; \mathtt{po})$. T is acyclic since $T \subseteq \mathtt{po}$. Note that $[\mathtt{RU}]; T^+; [\mathtt{WU}] \subseteq \mathtt{hb}$. Indeed, if there is a T-path from $a \in \mathtt{RU}$ to $b \in \mathtt{WU}$ that contains a fence event, then $\langle a, b\rangle \in \mathtt{fence} \subseteq \mathtt{hb}$. Otherwise, $\langle a, b\rangle \in (\mathtt{deps} \cup \mathtt{po}|_{loc})^+$. By (ppo-lower-bound), we obtain that $\langle a, b\rangle \in \mathtt{ppo} \subseteq \mathtt{hb}$.

Next, we show that $T^+ \cup \mathtt{rfe}^+$ is acyclic. Since both T^+ and $\mathtt{rfe}^+$ are irreflexive and transitive, and $\mathtt{rfe}^+ \subseteq \mathtt{WU} \times \mathtt{RU}$, it suffices to show that $[\mathtt{WU}]; \mathtt{rfe}^+; T^+; [\mathtt{WU}]$ is acyclic. Now, for every $a, c \in \mathtt{WU}$ and $b \in \mathtt{RU}$ such that $\langle a, b\rangle \in \mathtt{rfe}^+$ and $\langle b, c\rangle \in T^+$, we also have $\langle a, b\rangle \in \mathtt{hb}^+$ and $\langle b, c\rangle \in \mathtt{hb}$. Hence, $[\mathtt{WU}]; \mathtt{rfe}^+; T^+; [\mathtt{WU}] \subseteq \mathtt{hb}^+$. Finally, $\mathtt{hb}^+$ is acyclic since G is Power-coherent. □

B Appendix: Unsoundness of Sequentialization in Hardware Memory Models

We present example showing that sequentialization is an unsound transformation in TSO, Power, and ARM.

B.1 TSO

Consider the IRIW (independent reads of independent writes) litmus test:

Initially, $x = y = 0$

$$\begin{array}{l} a := x; \ \#1 \\ b := y; \ \#0 \end{array} \;\Big\|\; x := 1; \;\Big\|\; y := 1; \;\Big\|\; \begin{array}{l} c := y; \ \#1 \\ d := x; \ \#0 \end{array}$$

Forbidden under TSO

By merging the first two threads and similarly the last two threads, we get a variant of the SB (store buffering) litmus test:

Initially, $x = y = 0$

$$\begin{array}{l} x := 1; \\ a := x; \ \#1 \\ b := y; \ \#0 \end{array} \;\Big\|\; \begin{array}{l} y := 1; \\ c := y; \ \#1 \\ d := x; \ \#0 \end{array}$$

Allowed under TSO

B.2 Power

Initially, $[x] = [y] = [z] = [w] = 0$

$$\begin{array}{l} [w] := 1; \\ \texttt{sync}; \\ [y] := 1; \end{array} \;\Big\|\; \begin{array}{l} a := [y]; \ \#1 \\ \texttt{if } a = 1 \texttt{ then} \\ \quad [x] := 1; \end{array} \;\Big\|\; \begin{array}{l} b := [x]; \ \#1 \\ c := [z + b - b]; \ \#0 \end{array} \;\Big\|\; \begin{array}{l} [z] := 1; \\ \texttt{sync}; \\ d := [w]; \ \#0 \end{array}$$

Forbidden under Power

By sequentializing the middle two threads, however, the behavior is allowed.

Initially, $[x] = [y] = [z] = [w] = 0$

$$\begin{array}{l} [w] := 1; \\ \texttt{sync}; \\ [y] := 1; \end{array} \;\Big\|\; \begin{array}{l} a := [y]; \ \#1 \\ \texttt{if } a = 1 \texttt{ then} \\ \quad [x] := 1; \\ b := [x]; \ \#1 \\ c := [z + b - b]; \ \#0 \end{array} \;\Big\|\; \begin{array}{l} [z] := 1; \\ \texttt{sync}; \\ d := [w]; \ \#0 \end{array}$$

Allowed under Power

B.3 ARM

Same example as for Power, with `dmb` fences instead of `sync`.

C Appendix: Power's `isync`

The following example shows that reorderings over SPower fall short to explain the weak behavior of Power in programs with control fences.

		$c_1 := x;$ // 1	$c_2 := y;$ // 1		
$w_1 := 1;$	$a_1 := x;$ // 2	$x := 2;$	$y := 2;$	$a_2 := x;$ // 2	$w_2 := 1;$
`lwsync`;	`lwsync`;	$d_1 := z_1^{\mathsf{acq}};$ // 1	$d_2 := z_2^{\mathsf{acq}};$ // 1	`lwsync`;	`lwsync`;
$z_1 := 1;$	$b_1 := w_1;$ // 0	$y := 1;$	$x := 1;$	$b_2 := w_2;$ // 0	$z_2 := 1;$

Here the C11-style acquire reads ($d_1 := z_1^{\mathsf{acq}}$ and $d_2 := z_2^{\mathsf{acq}}$) denote a read followed by a branch and a control fence, resulting in `ctrl+isync` dependency edges from the read to every `po`-subsequent memory access.

The specified weak behavior ($a_1 = a_2 = 2, b_1 = b_2 = 0, c_1 = c_2 = d_1 = d_2 = 1$) is allowed by Power, where the only Power-coherent execution showing this behavior includes a `po` $\cup$ `rf`-cycle going between the third thread and the fourth thread.

Now, due to the lightweight fences (that cannot be reordered at all) and the acquire reads (that cannot be reordered with subsequent accesses), the only sound reorderings that can be applied here are the reordering of the store to x across the acquire read in the third thread, and, symmetrically, the reordering of the store to y across the acquire read in the fourth thread. Applying the reordering in the third thread results in the following (the reordering in the fourth thread is symmetric):

		$c_1 := x;$ // 1	$c_2 := y;$ // 1		
$w_1 := 1;$	$a_1 := x;$ // 2	$d_1 := z_1^{\mathsf{acq}};$ // 1	$y := 2;$	$a_2 := x;$ // 2	$w_2 := 1;$
`lwsync`;	`lwsync`;	$x := 2;$	$d_2 := z_2^{\mathsf{acq}};$ // 1	`lwsync`;	`lwsync`;
$z_1 := 1;$	$b_1 := w_1;$ // 0	$y := 1;$	$x := 1;$	$b_2 := w_2;$ // 0	$z_2 := 1;$

However, for the reordered program, the specified behavior is not allowed under Power. The reason is that we now have a dependency from the read of z_1 to the write of x, which implies an `hb`-edge from the write of z_1 in the first thread to the write of x in the third one. In turn, we have a `prop`-edge from the write of w in the first thread to the write of x in the third one, which leads to a violation of Power's observation condition.