Memory Consistency and Program Transformations

AKSHAY GOPALAKRISHNAN, Computer Science, McGill University, Montreal, Canada
CLARK VERBRUGGE, Computer Science, McGill University, Montreal, Canada
MARK BATTY, Computer Science, University of Kent, Canterbury, United Kingdom of Great Britain and Northern Ireland

A memory consistency model specifies the allowed behaviors of shared memory concurrent programs. At the language level, these models are known to have a non-trivial impact on the safety of program optimizations. This limits the ability to rearrange/refactor code without introducing new behaviors. Existing programming language memory models try to address this by permitting more (*relaxed/weak*) concurrent behaviors, but are still unable to allow all the desired optimizations. A core problem is that *weaker* consistency models may also render optimizations unsafe, a conclusion that goes against the intuition of them allowing more behaviors. This exposes an open problem of the *compositional interaction* between memory consistency semantics and optimizations; which parts of the semantics correspond to allowing/disallowing which set of optimizations is unclear. In this work, we establish a formal foundation suitable enough to understand this compositional nature. We decompose optimizations into a finite set of elementary *effects*, over which aspects of safety can be assessed. We use this decomposition to identify a desirable compositional property (*complete*) that would guarantee the safety of optimizations from one memory model to another. We showcase its practicality by proving such a property between Sequential Consistency (SC) and SC_{RR}, the latter allowing independent read-read reordering over *SC*. Our work potentially paves way to a new design methodology of programming-language memory models, one that places emphasis on the optimizations desired to be performed.

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

Additional Key Words and Phrases: Memory consistency, compiler optimizations, correctness, compositionality

ACM Reference Format:
Akshay Gopalakrishnan, Clark Verbrugge, and Mark Batty. 2025. Memory Consistency and Program Transformations. *Form. Asp. Comput.* 37, 3, Article 22 (June 2025), 42 pages. https://doi.org/10.1145/3721143

1 Introduction

The semantics of shared memory concurrency is given by what is known a memory consistency model. They specify constraints on the visibility of actions performed by a thread/processor to another, which give the allowed concurrent behaviors of a program. To be specific, it tells us what write values a read to shared memory can observe in a concurrent execution. While the

This work is supported in by NSERC Discovery Grant RGPIN-2019-05213 and EPSRC UK Grant EP/X015076/1.
Authors' Contact Information: Akshay Gopalakrishnan, Computer Science, McGill University, Montreal, Quebec, Canada; e-mail: akshay.akshay@mail.mcgill.ca; Clark Verbrugge, Computer Science, McGill University, Montreal, Quebec, Canada; e-mail: clump@cs.mcgill.ca; Mark Batty, Computer Science, University of Kent, Canterbury, Kent, United Kingdom of Great Britain and Northern Ireland; e-mail: m.j.batty@kent.ac.uk.

ACM 1433-299X/2025/06-ART22
https://doi.org/10.1145/3721143

use of multi-core machines and software has become ubiquitous, the impact of concurrency on compilers is still being explored. Compilers play a significant role in program performance by identifying and optimizing code patterns. These optimizations save CPU cycles and memory access times. However, these program transformations (or commonly known as program/compiler optimizations) have historically been designed for sequential programs. Therefore, performing them has non-trivial consequences in a concurrent context [19]. Even simple transformations such as constant propagation or code motion [23] can be rendered unsafe: performing them results in the program exhibiting new behaviors. The goal then, for program performance, is to have programming language concurrency models that allow as many optimizations as possible. However, existing models used in practice are not able to satisfy this, and how to allow a desirable set of optimizations is not well understood [22].

A core reason revolves around the intuition behind how to allow a desired optimization. In a sequential program (single threaded), an optimization is considered unsafe if it fails to guarantee the functional behavior of the program. This is commonly referred to as ensuring *conservative correctness* for any transformation. Focusing on the possible interactions of multiple, concurrently executing code sequences, we can deem an optimization unsafe if it results the program exhibiting behaviors not intended by the programmer. *Weak* consistency models (e.g., TSO, RA, RMO) allow more concurrent behaviors. Thus, intuition dictates *weakening* the consistency semantics should be enough to allow any desired set of optimizations in addition to those guaranteed by the original model. However, this intuition turns out to be incorrect: weaker models may allow more behaviors, but not strictly more optimizations. Thus, how one might compose consistency requirements while retaining all the desired optimizations is unclear. Our work sheds light on this issue, by answering the following two fundamental questions regarding compositionality:

(1) When is the safety of an optimization implied from one model to another?
(2) Given a memory model and a desired optimization known to be unsafe, can we formally derive/specify a memory model allowing it while retaining the safety of existing ones?

1.1 Core Idea

Prior work approaches safety of optimizations by establishing semantic guarantees over execution traces [34]. However, these semantic guarantees are thread-local, and are independent of the whole program context. Relying on such guarantees restrict optimizations that are dependent on global analyses of the program being optimized. In order to understand the compositional interaction between optimizations and concurrency, we adopt an approach in line with conservative correctness. We decompose an optimization of a program into a set of syntactic-effects on sub-programs, as opposed to semantic-effects on traces. These sub-programs will collectively represent a conservative approximate of program behavior.

For instance, consider the program optimization in Figure 1, where code motion is performed, removing $w = 1$ outside the conditional block. The same code motion can be decomposed into two separate syntactic-effects, one for each conditional branch path. The "then" branch sub-program incurs adjacent reordering: $b = y$ and $w = 1$. Whereas the "else" branch sub-program incurs an introduction of write: $w = 1$.

Such a decomposition can be extended to concurrent contexts, as the example optimization in Figure 2. In each of our concurrent examples, $x, y, z, w \ldots$ represent shared memory and $a, b, c, d \ldots$ represent local memory to each thread. The write $w = 1$ is placed from thread $T1$ to thread $T2$, a form of code motion that optimally reorders work among threads. This can be decomposed into two separate syntactic-effects on the respective sub-programs, showcased by "effect1" and "effect2". The "then" and "else" branch sub-programs incur de-ordering $w = 1$ from $T2$ and ordering to $T1$.

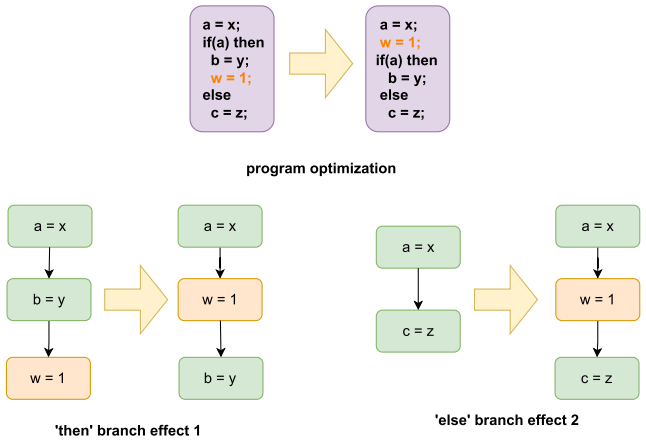

Fig. 1. Code motion equivalent to reordering (effect 1) and introduction (effect 2) on two sub-programs.

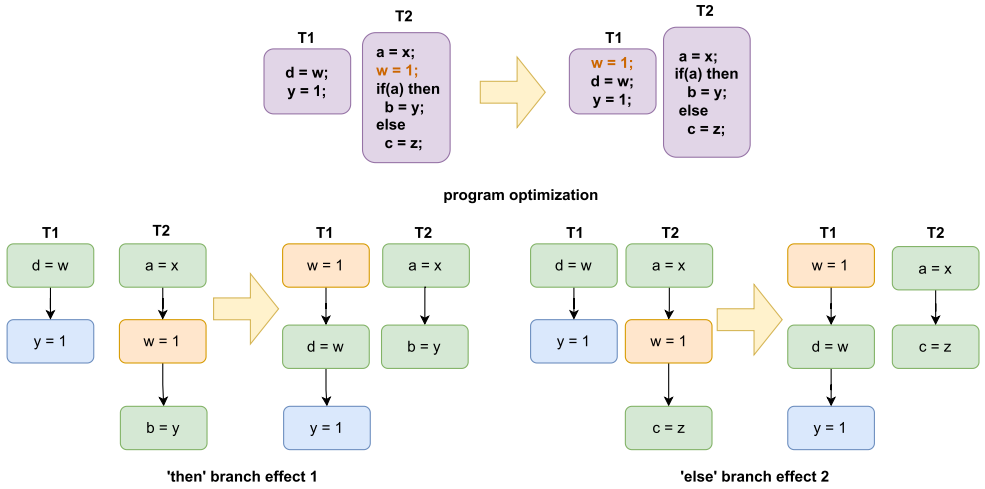

Fig. 2. Code motion equivalent to performing a de-ordering in conjunction with ordering w.r.t. threads $T1$, $T2$ on two sub-programs.

Dividing a transformation into some combination of effects on these simpler programs allows us to assess safety by removing the complexity of conditionals and loops. We can now answer the two fundamental questions using effects, with the following observation.

— An optimization is safe under given memory model if all the corresponding effects are safe.
— The safety of an optimization is preserved from one memory model to another if the safety of corresponding effects is preserved.
— Allowing a desired optimization translates to allowing the corresponding effects.

1.2 Contributions

Answering the aforementioned two fundamental questions at the level of effects results in the following formal contributions:

— We represent a conservative approximate of concurrent program behaviors, depicted by *pre-traces* and *candidate-executions* (Section 3.1).

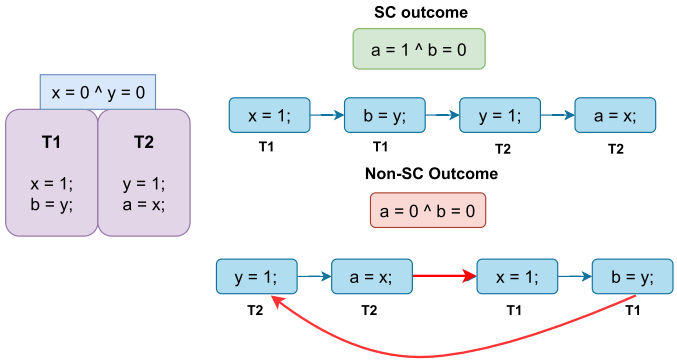

Fig. 3. The SB litmus test showcasing an allowed (green box) and forbidden (red box) outcome under *SC*.

— We use this approximation to decompose program transformations into different *transformation-effects* on each pre-trace (Section 3.3).
— We propose a methodology to derive consistency models that represent collections of desirable transformation-effects. This is followed by formally specifying the desired compositional interaction, *Complete*, which guarantees safety of transformation-effects across models (Section 4).
— We showcase the practicality of the proposed methodology by deriving model SC_{RR} using **Sequential Consistency** (***SC***) and reordering of independent reads, followed by proving it Complete for a significant set of transformation-effects safe under SC (Section 5).

Road Map. Section 2 goes over some informal background and motivation which can be skipped by readers familiar with the problem of safe optimizations and memory consistency. Section 3 goes over our view for programs as an over-approximate set of *pre-traces*, followed by the formal description of program transformations as a set of *transformation-effects*. Section 4 showcases our proposed methodology and formalizes the desired compositional interaction *Complete*. Section 5 goes over concrete models, showing the practicality in deriving models by adding desired transformations and proving the model Complete w.r.t. the original. Section 6 discusses the advantage of having such an approach for programming language memory models; one that focuses on the desirable set of compiler optimizations. Finally, Section 7 goes over related work followed by Section 8 with conclusion and potential future work. The full proofs of our results are provided in Appendices at the end.

2 Background/Motivation

Sequential interleaving or SC is the de-facto memory model relied on while designing concurrent programs [18]. Informally, it is defined as a set of all outcomes possible by an arbitrary interleaving of the sequential executions of each thread. For instance, consider the **store-buffering** (**SB**) litmus test in Figure 3. Each shared memory is initialized to 0. As per SC, several sequential interleavings are possible, each potentially giving us a different outcome. The outcome $a = 1 \wedge b = 0$ in the green box is possible due to the interleaving order shown by the blue boxes below it. T1 can perform both its actions, which can be followed by actions of T2. The outcome in the red box, however, cannot be observed under *SC*; the blue boxes below it along with the red arrows indicate a cyclic order of execution. Via *SC* reasoning, since $y = 1$ sequentially precedes the copy of x into a, a holding the value 0 implies $y = 1$ has happened before $x = 1$, thus enforcing $b = 1$.

While *SC* indeed is closer to a programmer's intuitive reasoning, its semantics have a detrimental impact on performance [2]. We direct our focus to the impact on compiler optimizations. These

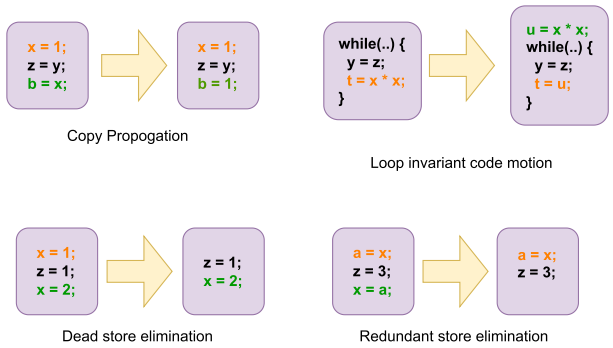

Fig. 4. Different optimizations routinely performed by compilers today.

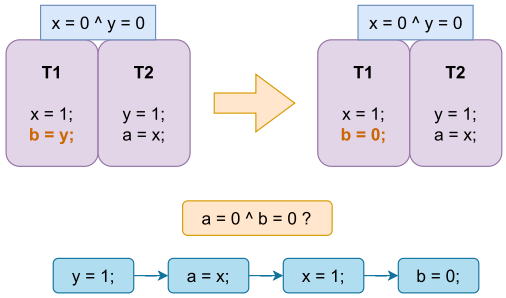

Fig. 5. The forbidden outcome $a = 0 \wedge b = 0$ permitted under *SC* after constant propagation, setting $b = 0$.

include syntactic changes that preserve semantics, such as code-motion, common sub-expression elimination, whole algorithm transformations, and so on [3, 23]. The performance of the program, then, can be attributed to effectively applying these optimizations on a program. Figure 4 showcases some common optimizations done by the compiler on our programs. Copy/constant propagation identifies redundant reads ($b = x$) from memory whose value already exists in the program ($x = 1$). Dead store elimination removes writes which have been overwritten before its value is used. Redundant store elimination removes writes which write the same current value to memory. Loop invariant code motion moves code fragments whose result do not change in any iteration outside the loop.

Many of these optimizations rely on plain code motion, which help in identifying redundancies in a thread-local (peephole) fashion. Redundancies such as these are typically identified through dataflow analysis [23], which help identify invariant properties on the information flow between two program points. For instance, the constant propagation example in Figure 4 is possible since x holding the value 1 is true at all points between the $x = 1$ statement and the $b = x$ assignment. We may also view it as a sequence of transformations based on more local relations: it is safe to reorder $b = x$ and $z = y$, and then it is easy to assert (via dataflow analysis) the value of x propagated to b has no side effect.

Traditionally, the invariants core to such optimizations are established relying on sequential (or thread-local) flow of information. These optimizations, however, can have an undesirable impact on concurrent programs. For example, applying constant propagation on the SB example in Figure 5 may determine that y holding 0 is an invariant from the initialization of y through the entire execution of T1, and thus replacing $b = y$ with $b = 0$ is valid. This makes the outcome in

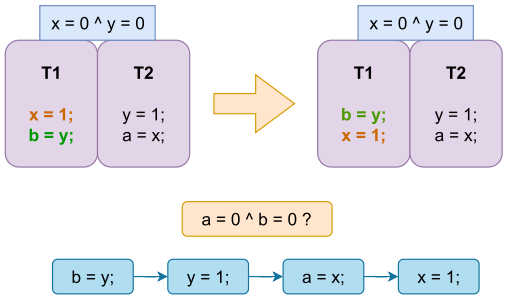

Fig. 6. The forbidden outcome $a = 0 \ \wedge \ b = 0$ permitted under *SC* after reordering $x = 1$ and $b = y$.

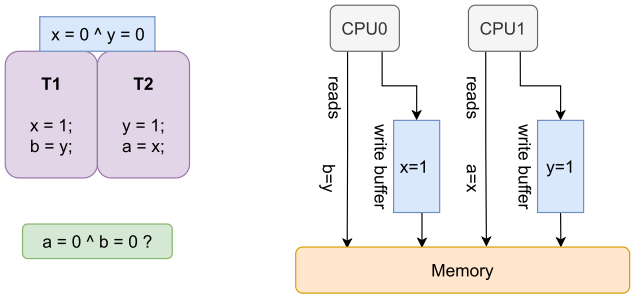

Fig. 7. The forbidden *SC* outcome $a = 0 \ \wedge \ b = 0$ for the program (left) allowed under TSO due to SB.

the "red" box of Figure 3 observable, even if SC is otherwise assumed, as shown by the sequential interleaving in blue boxes of Figure 5. In fact, even simple independent code motion can be rendered unsafe under *SC*. Revisiting the SB example in Figure 6, the compiler can see no harm (using thread-local information) in performing independent write-read reordering. This syntactic change, however, allows the outcome originally forbidden under *SC*; the blue boxes in Figure 6 show a sequential interleaving justifying the outcome.

2.1 Weak Consistency Models and Program Transformations

The impact of concurrency semantics on hardware optimizations has been studied extensively [2], resulting in weaker memory models: those that allow more hardware optimizations. One such example is **Total Store Order** (**TSO**), whose semantics represents an abstract machine using FIFO store buffers per-CPU. Informally, every write is first committed to the respective processor-specific write buffer. Reads to memory are first checked from the processor's own write-buffer and a fence/lock instruction forcing the buffer to be flushed to main memory. Under TSO, the outcome disallowed under SC can be explained via SB [24]. Figure 7 shows how the SB non-SC outcome is possible. From the lens of TSO, both the writes $x = 1$ by CPU0 and $y = 1$ by CPU1 can be buffered, while the symmetric reads of y by CPU0 and x by CPU1 can still read from main memory, thus allowing both the reads to return 0.

Intuitively, the effect of TSO write buffers is similar to program transformations (optimizations) that involve independent writes and reads to be reordered. Such reordering is unsafe under *SC*, which may compel us to conclude *TSO* permits more code transformations than *SC*. While such reordering is indeed allowed [22], the same cannot be said for rest. For instance, let us consider thread-inlining as a transformation, which sequentially merges code of two threads

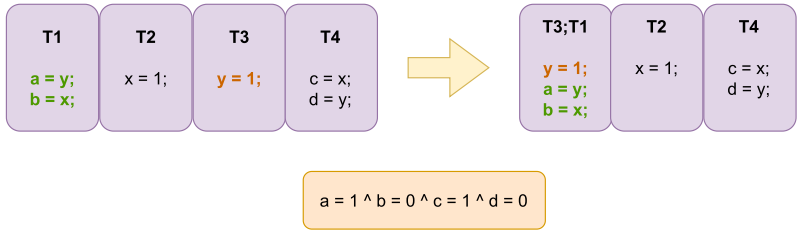

Fig. 8. Example program (left) where inlining $T1$ sequentially after $T3$ (right) permits the forbidden behavior (yellow box) under *TSO*, but not under *SC*.

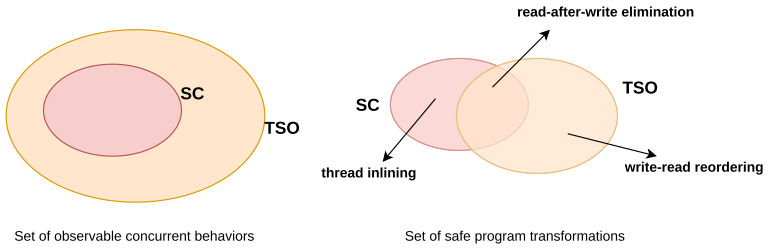

Fig. 9. Venn diagram showcasing the relation between sets of allowed concurrent outcomes (left) and program transformations (right).

into one. Although not a common optimization, and requiring some concurrency awareness, a compiler may decide to do this in order to match the amount of processors available in the target multi-core system. This optimization would avoid unnecessary context switching overhead [12], thus improving performance.

Figure 8 represents one such instance of thread-inlining. To see why this optimization is unsafe, first note the outcome in the orange box is not observed under *TSO*. Its semantics require all threads to observe modifications to memory in the same order (*multi-copy-atomic*). Thus, if $T1$ observes $y = 1$ before $x = 1$ then $T4$ is constrained by that same ordering. However, if we do inline $T1$ and $T3$, as shown on the right, the outcome in question can be observed under *TSO*: the read to y can be taken from the write buffer of $T3$; $T1$ rather than main memory, and thus $T4$ is free to observe $x = 1$ prior to $y = 1$. In contrast, it is well known that *thread inlining* is a sound transformation under *SC*; intuitively, it simply places a constraint on possible sequential interleavings of the program.

To conclude, we now have an example of a weak memory model under which we can observe more concurrent behaviors, but does *not strictly allow more safe transformations*. Figure 9 shows the picture we have for *TSO* and *SC*. The Venn diagram on the left indicates the set of concurrent behaviors observed under these two models. Whereas the right indicates our conclusion based on the set of transformations safe under them.

3 Preliminaries

We first go over our view of programs, representing program behaviors as a conservative approximate of *pre-traces* and their *candidate executions* (Section 3.1). This is intended to represent a compiler's conservative view of program executions. We then formally specify memory consistency models, adopting their declarative (axiomatic) format to filter out these execution traces (Section 3.2). This is followed by defining a *transformation-effect* over pre-traces, into which we decompose any program transformation (Section 3.3). We then formalize safety of any transformation-effects, followed by linking it to safety of program transformations (Section 3.4).

3.1 Programs, Traces, and Executions

We view concurrent programs (*prog*) as a parallel composition (||) of sequential programs (*sp*), each of which is associated with a thread id t and a sequence of actions p. Each p is composed of statements (*st*), conditional branches (if-then-else). Each *st* is either a shared memory event or a local computation ($a = e$). Memory events can either be a read from (e.g., $a = x$) or write to (e.g., $x = v$) shared memory. Write events are also associated with a value v, which can, in general, be a constant value or a local variable; for simplicity, we limit v to integers in this exposition. The complete program grammar is as shown below.

$$
\begin{aligned}
&prog := sp||prog \mid sp\\
&sp := t : p\\
&p := st \mid p;p \mid \text{if}(cond) \text{ then } \{p\} \text{ else } \{p\}\\
&st := a{=}x; \mid x{=}v; \mid a{=}e\\
&e := a \mid v\\
&cond := \text{true} \mid \text{false} \mid a == v \mid a != v\\
&\text{domains} := v \in \mathbb{Z} \mid t \in (\mathbb{N} \cup \{0\})
\end{aligned}
$$

Remark 1. Following previous work on this topic [13], we assume all loops terminate in a finite number of iterations and require static thread construction (via the || operator). Our proofs/results, however, do not depend on this assumption.

We represent initial values of shared memory as a sequence of write events syntactically ordered before all other memory accesses. Let *tid* and *mem* be mapping of an event to the thread id and memory (addresses, although in examples we will use unique variable names). Given a program in this way, we construct an abstract program graph.

Definition 3.1. An *abstract program Pr* is an abstraction of the original program, retaining the syntactic order represented by the binary relation po_{prog}, set of shared memory events St_{prog} and conditional branch points.

Figure 10 is an example of a program and its abstraction. Let $st_{prog}(Pr)$ return the set of statements and $\text{po}_{prog}(Pr)$ return the set of syntactic orders from abstract program Pr. Let St_t and po_t represent the set of events and syntactic order of thread t. Note that po_{prog} is a strict partial order.

We now take an abstract program Pr and construct a pre-trace. A pre-trace represents a possible execution trace consisting of maximal events from an Abstract Program and their syntactic order. We derive this by syntactically tracing program execution paths for each thread, non-deterministically choosing a path on every conditional branch point. Formally, we first define $St_t^{trace} \subseteq St_t$ as a subset of program statements of thread t such that $\forall a, b \neq a \in St_t^{trace}$ we have

$$
\begin{aligned}
&(a, b) \in \text{po}_t(Pr) \vee (b, a) \in \text{po}_t(Pr).\\
&(a, c) \in \text{po}_t(Pr) \implies c \in St_t^{trace}.\\
&(c, b) \in \text{po}_t(Pr) \implies c \in St_t^{trace}.
\end{aligned}
$$

We then derive $po_t^{trace} \subseteq \text{po}_t$ that represent the syntactic order between events in St_t^{trace}. Lastly, let po_{init} represent the syntactic order between initialization writes and memory events of each thread with $tid > 0$.

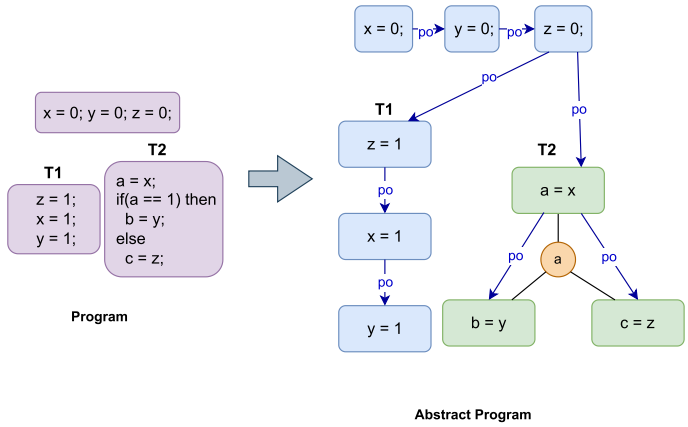

Fig. 10. Example program (left) mapped to its abstract form (right).

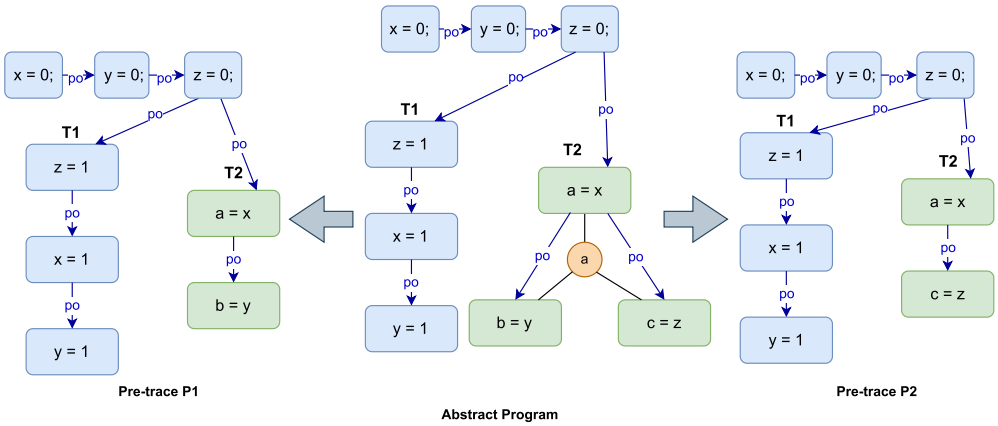

Fig. 11. Example abstract program (middle) mapped to its two possible pre-traces (left and right).

Definition 3.2. A *pre-trace* (P) is a tuple consisting for some *trace* of Pr, the St_t^{trace} and appropriate po_t^{trace} for each thread.

$$P = \left(\bigcup_{t=0}^{n} St_t^{trace}, \bigcup_{t=1}^{n} \mathrm{po}_t^{trace} \cup po_{init} \right).$$

Figure 11 shows a mapping of an abstract program to all its pre-traces.

Remark 2. Since po_t^{trace} and po_{init} are strict total orders, we omit the subscripts for po in pre-traces.

Let $st(P)$ return the set of statements and $\mathrm{po}(P)$ return the set of program orders. Further, let $r(P)$ and $w(P)$ return the set of read and write memory events of P, respectively. Let $st_{loc}(P), r_{loc}(P), w_P$ return the set of events of P operating on memory location *loc*. Let $!loc$ be those other than *loc* events.

Pre-traces still do not represent an outcome, which is generally considered to be the final state of memory (local as well as shared). To represent outcomes, we first define ***read-from*** (**rf**) as a binary relation from a write event to a read event of same memory. Next, we define ***memory-order***

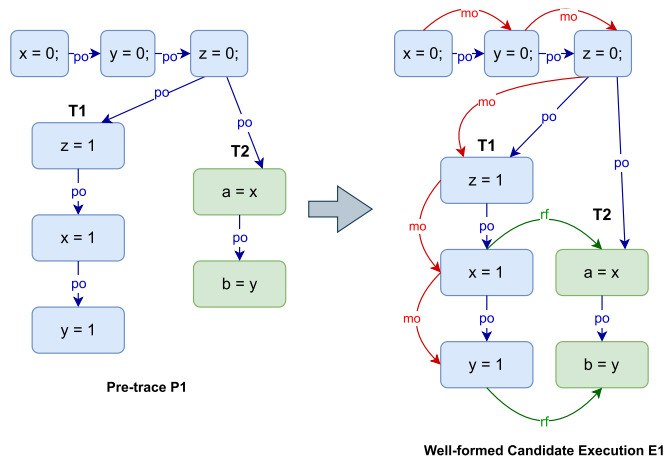

Fig. 12. Example pre-trace (left) mapped to a possible candidate execution (right).

(**mo**) to be the propagation order between write events of P. Let mo_{loc} represent the mo between writes to the same memory and $max(loc)$ give the maximal write (if exists) in mo_{loc}.

Definition 3.3. An execution is a pre-trace P augmented with non-empty rf, mo relations between events.

$$E = (P, \text{rf}, \text{mo}).$$

Let $p(E)$, $\text{rf}(E)$, $\text{mo}(E)$ be projection functions that return the pre-trace P, rf, and mo relations, respectively. We say E is *well-formed* ($\text{wf}(E)$) if every read event is associated with an rf relation.

Definition 3.4. An execution E is a candidate (or a *candidate execution* E) if

a) $\text{wf}(E)$.
b) rf^{-1} functional.
c) mo total order.
d) $\forall loc \,.\, max(loc) \neq \phi$.

where *loc* represents all possible shared memory locations.[1]

We finally define an outcome (behavior) to be the $\text{rf}(E)$ and $\text{mo}(E)$ extracted from any candidate execution E.

Let $\langle P \rangle$ represent the set of candidate executions E of given pre-trace P. Figure 12 shows one possible candidate execution for a given pre-trace P.

Remark 3. Unless explicitly stated, we refer to E as a candidate execution.

3.2 Memory Model

We have decomposed programs into a finite set of pre-traces and candidate executions. We now define a memory model, which Informally, is a set of rules that apply to candidate executions.

Definition 3.5. A *memory consistency model* M is a finite set of consistency rules. A *consistency rule* $a \in M$ is a function that maps executions to boolean values.

We utilize the memory model to give us executions (and thereby candidate executions) that adhere to its rules.

[1]The existence of a maximal event in every mo_{loc} captures the final state of memory.

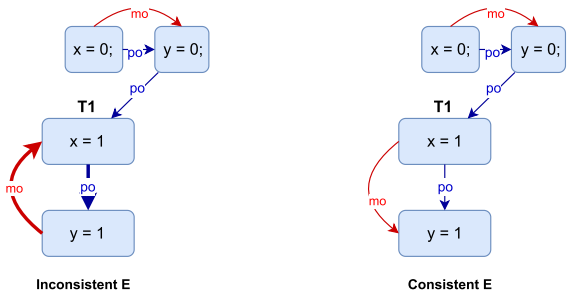

Fig. 13. Inconsistent (left) and consistent (right) executions as per memory model with consistency rule (po ∪ mo) acyclic.

Definition 3.6. An execution E is *consistent* w.r.t. memory model M, written $c_M(E)$, if all the consistency rules of M return *true*.[2]

$$c_M(E) \implies \forall a \in M \,.\, a(E) = \text{true}.$$

An execution E is *inconsistent* w.r.t. memory model M if at least one consistency rule returns *false*. As an example, suppose a memory model M has a single consistency rule: one that returns true for any execution when po ∪ mo acyclic. Figure 13 shows two candidate executions, one consistent and other inconsistent as per M.

Remark 4. In all our examples of candidate executions, we will omit showing certain relations which can be implied. For instance, we do not show all the mo and po edges in Figure 13.

Let $[\![P]\!]_M$ represent the set of candidate executions of pre-trace P given memory model A. Let $I\langle P\rangle_M$ represent the set of inconsistent candidate executions. We then have $I\langle P\rangle_M \cup [\![P]\!]_M = \langle P\rangle$.

Actual Behaviors of a Program. Note that the memory model is not enough to assert whether an outcome of the program is possible. A memory model needs to be coupled with thread-local language semantics. This will enable filtering out pre-traces and corresponding candidate executions which are not possible by a program. For instance, Figure 14 shows a pre-trace of a program tracing the false conditional path: one that cannot happen. These pre-traces can be filtered out by thread-local reasoning. Once the relevant set of pre-traces are determined, we can filter out the program's behavior as per the entire language semantics.

Remark 5. Our results are independent of the choice of thread-local semantics adopted.

3.3 Program Transformations

The set of compiler optimizations we aim to address can be placed under the umbrella of program transformations: syntactic changes to programs with the intention to preserve semantics. To relate these transformations to our view of programs, we note that each transformation at the program level has a varied effect on pre-traces, which in turn affect candidate executions. For instance, revisiting a previous example in Figure 15, reordering $w = 1$ outside the conditional block at the program level affects the pre-traces in two different ways: one as simple reordering and one as write introduction. More generally, these effects can result in adding/removal of statements (st) and modification to syntactic order (po).

[2]Note that an execution need not be a *candidate* to be consistent under a memory model.

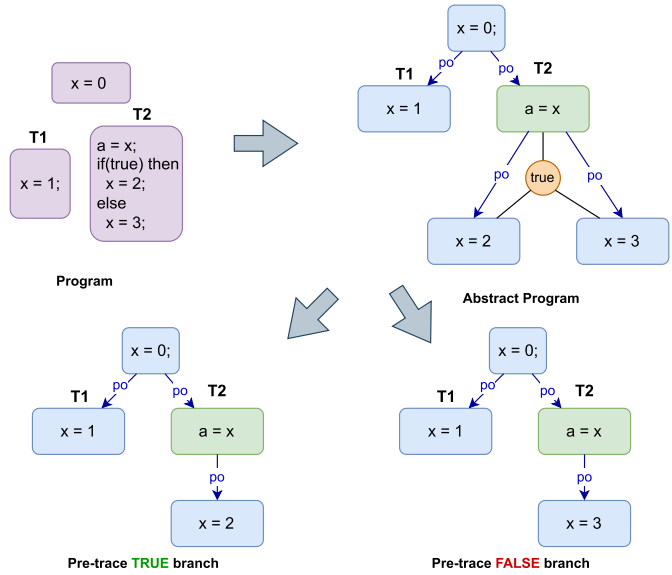

Fig. 14. The pre-trace generated for program (top left) with *false* branch (bottom right) will be filtered out by appropriate thread-local semantics.

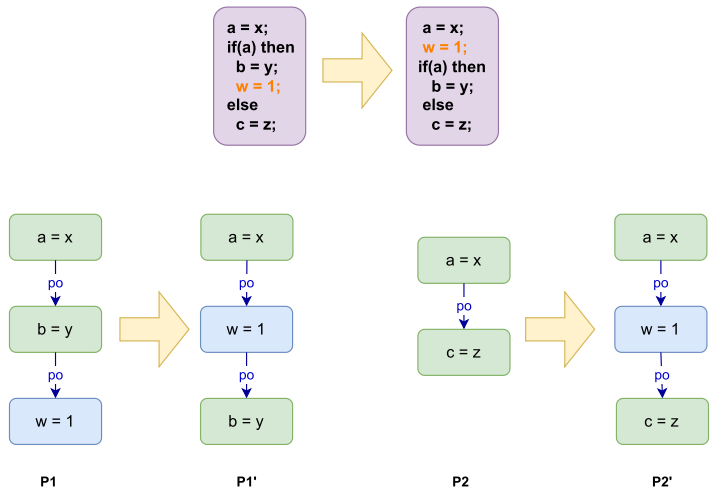

Fig. 15. Example of reordering $w = 1$ outside the conditional block in program as transformation-effects $P1 \mapsto P1'$ and $P2 \mapsto P2'$.

Definition 3.7. A *transformation-effect* tr is defined on a pre-trace $P \mapsto_{tr} P'$ as

$$P' = ((st(P) - st^-) \cup st^+,\ (\mathrm{po}(P) - \mathrm{po}^-) \cup \mathrm{po}^+).$$

where st^+, po^+ represent elements added and st^-, po^- those removed.

Going back to Figure 15, the reordering has the transformation-effects $P1 \mapsto_{tr} P1'$, $P2 \mapsto_{tr} P2'$ defined as

$$P1' = (st(P1),\ (\mathrm{po}(P1) - \{[b = y]; \mathrm{po}; [w = 1]\}) \cup \{[w = 1]; \mathrm{po}; [b = y]\}).$$
$$P2' = (st(P2) \cup \{w = 1\},\ \mathrm{po}(P2) \cup (\{[a = x]; \mathrm{po}; [w = 1]\} \cup \{[w = 1]; \mathrm{po}; [b = y]\})).$$

Remark 6. Since po is transitive, we omit mentioning certain edges added/removed as part of a transformation-effect that are implied via transitivity.

3.3.1 Identifying Relevant Pre-Trace Pairs. We mentioned above that a program transformation, in essence, has an effect on pre-traces. Definition 3.7 is generic; any pre-trace can be considered as a result of a transformation-effect on any other pre-trace. The example in Figure 15, however, specifically has just two effects $P1 \mapsto P1'$ and $P2 \mapsto P2'$. While we label these as transformed pre-traces, it is instead a mapping from pre-traces of one program to pre-traces of the transformed program. This implies the appropriate effects that constitute a program-transformation are not naive mapping of every pre-trace of the original program to every pre-trace of the transformed. More specifically, we want to ensure the transformation is described by effects $P1 \mapsto P1'$ and $P2 \mapsto P2'$, but not $P1 \mapsto P2'$ or $P2 \mapsto P1'$.

Tracking event set identities. To identify these appropriate pairs of pre-traces, we need a way to compare them. We start by associating each memory event with a unique identity. This would ensure that a transformation-effect can be derived by simply comparing the event set of two pre-traces; making it easier to identify which statements were removed/reordered/added. Having such metadata is similar to the metadata LLVM associates to the program, which the compiler can keep track of, during an optimization [10].

Tracking Conditional branch choices. Associating identities helps in precisely categorizing effects involving memory events that are congruent (e.g., two reads to the same location x). But this is not sufficient to identify the exact pairs of pre-traces in Figure 15: the metadata on events alone would still consider $P1, P2'$ to be a valid pair.

To address this, we additionally associate the conditional branch points with a unique identity, which captures the conditional expression and an identity to the branch. This ensures we know exactly which set of conditional branch points remain unchanged between the original and transformed program. With this additional metadata, we ensure that for each common branch point, the same choice (then branch or else branch) is made to generate the corresponding pre-traces. Two branch points are common if they have the same identity and same expression. This rules out the case of $P1, P2'$ being a valid pair in Figure 15.

Remark 7. Asserting the equality of two conditional expressions can be complex, incorporating thread-local semantics to identify more common branch points, but for simplicity we assume them to be just the same expression.

Figure 16 summarizes our idea of event set and conditional branch point identities. The memory events are captured by identities $A, A1, A2, A3$ and the branch point is captured by $C1$.

More generally, let B_i represent a branch point unique identity, where $id(B_i)$ is the label, $e(B_i)$ the associated conditional expression, and $c(B_i)$ the choice made. Let $cond(P)$ be the set of all conditional branch points $\{B1, B2, B3 \ldots\}$ for pre-trace P. Then, two pre-traces $P1, P2$ are comparable if the choices for every conditional branch point common to both are the same. We now finally have a process to compare pre-traces, which form a pair between which the transformation-effect occurs. Let $\langle\!\langle Pr \rangle\!\rangle$ represent the set of pre-traces for abstract program Pr.

Definition 3.8. Two pre-traces $P \in \langle\!\langle Pr \rangle\!\rangle, P' \in \langle\!\langle Pr' \rangle\!\rangle$ are *comparable* ($P \sim P'$) if for each conditional branch point common to both, the same conditional outcome is chosen.

$$\forall B \in cond(P), B' \in cond(P')\ .\ id(B) = id(B') \wedge e(B) = e(B') \implies c(B) = c(B').$$

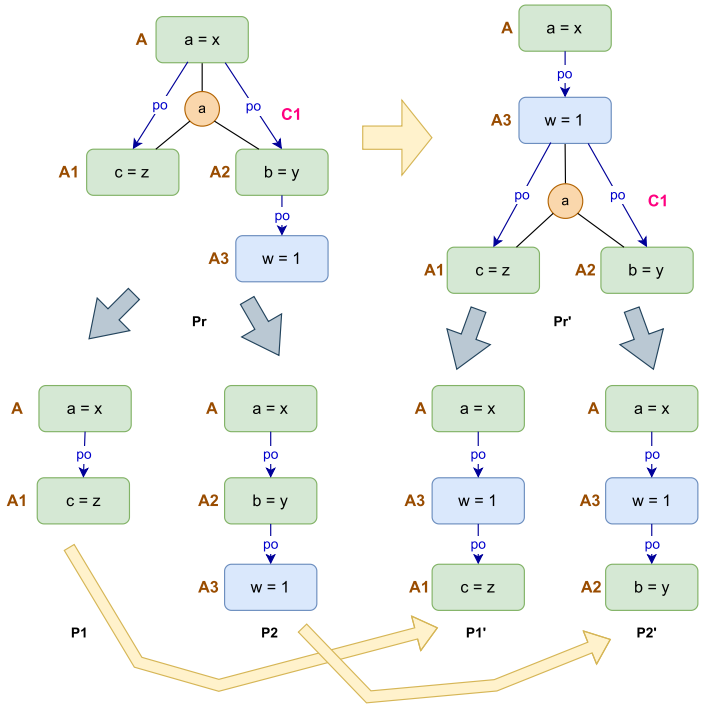

Fig. 16. Tracking event set identities ($A, A1, A2, A3$) and conditional branch points ($C1$) allows us to represent the reordering as only the two effects $P1 \mapsto P1', P2 \mapsto P2'$.

Remark 8. Every $P \mapsto_{tr} P'$ we consider henceforth inherently implies $P \sim P'$.

Given such similar pre-traces, we can extract the appropriate transformation-effect that changes one pre-trace to another.

3.4 Safety of Transformation and Its Effects

On having explained how to identify the relevant pairs of pre-traces that constitute a transformation-effect, we can now delve into their safety. In general, a program transformation is considered safe if the resultant program does not introduce additional behaviors. In our context, a behavior is synonymous to a candidate execution's mo and rf components. Therefore, we first specify how to compare them. We can compare two candidate executions, if their common set of read events have the same rf relations and their common set of writes have the same mo.

Definition 3.9. Two executions E, E' are *comparable*, written $E' \sim E$, if

— $r(p(E')) \cap r(p(E)) = codom(\mathrm{rf}(E') \cap \mathrm{rf}(E))$.

— $\forall a \in \mathrm{mo}(E)\ .\ (dom(a) \in w(p(E')) \wedge codom(a) \in w(p(E'))) \implies a \in \mathrm{mo}(E')$.

where *dom* and *codom* return the domain and co-domain, respectively, of input binary relation.

Remark 9. $\sim$ is symmetric ($E' \sim E \iff E \sim E'$).

We now specify how to compare sets of behaviors.

Definition 3.10. An execution set A is *contained* in candidate execution set B, written $A \sqsubseteq B$, if

$$\forall E' \in A\ .\ \exists E \in B \text{ s.t } E' \sim E.$$

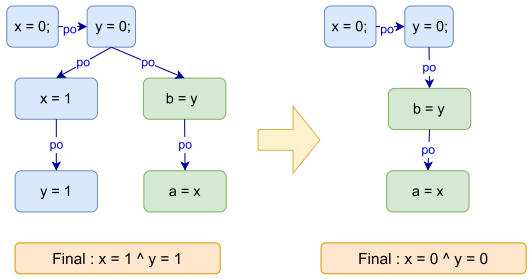

Fig. 17. Definition 3.12 considers the above effect to be safe under *SC*, thereby allowing a new behavior with different final state of memory.

We first use the above relation to constrain the transformation-effects $P \mapsto_{tr} P'$ to those that do not introduce a write-value. Introducing redundant/unused writes is often safe in sequential code, but they are more likely to be unsafe in a concurrent setting. While a redundant write may not be read/used in a thread-local context, they can easily be visible to other threads, thereby allowing new memory read values. Restricting such introductions can be expressed using $\langle\rangle$: the set of candidate executions of P' must always be contained in that of P - $\langle P' \rangle \sqsubseteq \langle P \rangle$.

Remark 10. We henceforth only consider transformation-effects that respect $\langle P' \rangle \sqsubseteq \langle P \rangle$.

We now can specify the safety of a transformation-effect. Informally, we require the effect to not introduce any additional consistent behaviors. This can be formally expressed as follows.

Definition 3.11. A transformation-effect tr on a pre-trace $P \mapsto_{tr} P'$ is *safe* under memory model M, written $psafe(M, tr, P)$, if

$$[\![P']\!]_M \sqsubseteq [\![P]\!]_M.$$

Effects Altering Final Shared Memory State. We note that Definition 3.11 would allow effects that may remove writes altering the final state of memory. For instance, even for a simple message-passing program (left) in Figure 17, transformation removing writes $x = 1$ and $y = 1$ (right) will always be considered a safe effect as per Definition 3.11 under SC. However, in reality, this alters the final state of memory from $x = 1 \wedge y = 1$ (left) to $x = 0 \wedge y = 0$ (right), which is clearly something we do not desire. To prevent this, we instead assume the final writes (maximal writes in mo_{loc}) can be all read at the end of the program execution.[3] This can be represented in the form of reads syntactically ordered after all events in a pre-trace, which are also associated with an rf relation as normal reads. Having this assumption prevents us from allowing effects of the form like in Figure 17.

3.4.1 From Safety of Effects to Transformations. Definition 3.8 gives us the pairs of pre-traces for which we need to check safety of transformation-effects, while Definition 3.11 gives us the criteria for safety of a transformation-effect. In order to determine safety of the transformation, we first note that every pre-trace of the transformed program represents one possible execution trace. In order to ensure no additional behaviors are introduced, each of these pre-traces must then have at least one matching comparable pre-trace from the original program. Then, for all such matching pairs, we would require the corresponding effects to be safe.

[3]The set of reads to the final state of memory can be a subset of shared memory, as some locations may not be used (read) by the program at all. We can leave the precise choice of final reads to the compiler.

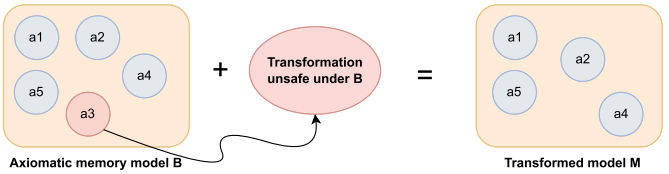

Fig. 18. For memory model B, allowing an unsafe transformation is equivalent to removing consistency rule $a3$, giving us resultant transformed model M.

Definition 3.12. A program transformation t, modifying abstract program Pr to Pr', is safe under memory model M if

— $\forall P' \in \langle\!\langle Pr' \rangle\!\rangle \;.\; \exists P \in \langle\!\langle Pr \rangle\!\rangle \;.\; P \sim P'$.
— $\forall P \in \langle\!\langle Pr \rangle\!\rangle, P' \in \langle\!\langle Pr' \rangle\!\rangle \;.\; P \sim P' \implies psafe(P, tr, M)$.

4 Methodology

With the base setup, there are two directions one can take. The first obvious one is to test existing models used in practice. However, this would require additionally mapping the instruction semantics to a common language over which we can formally specify and compare the models for our purpose [28]. While this can be done, we note that our primary goal is on understanding the interaction between memory consistency semantics and optimizations. Hence, we resort to the second approach; we derive models incrementally by adding desired transformations known to be unsafe under a base model. In this section, we lay out our methodology in deriving memory models this way, followed by formally identifying the properties desired that make it compositional w.r.t. transformations.

We first lay out the intuition to our approach, specifying the desired compositional properties that the resultant model must satisfy w.r.t. the original (Section 4.1). We then formally specify each desired property, followed by laying out in brief a strategy to prove them (Section 4.2, Section 4.3).

4.1 Proposed Approach

Recall that a transformation-effect is unsafe under memory model B because on performing such a transformation, a new behavior may be introduced. This means, the behavior in the original program was disallowed by some consistency rule (axiom/constraint) being violated. Equivalently, the behavior in the transformed program was allowed as the existing consistency rules of B are respected. To allow a transformation, then, translates to one of the following:

— Add some constraints to B, thereby prohibiting the additional behavior in the transformed program.
— Remove some constraints from B, thereby allowing the additional behavior in the original program itself.

The first approach, while feasible, may result in models that are not practical. For instance, allowing write-read reordering would translate to disallowing the interleaving behavior shown in Figure 6. The resulting model would only permit outcomes that stem from executing $T1$ entirely before $T2$ (and vice versa). Thus, allowing write-read reordering translates to reasoning with single-threaded programs.[4]

Hence, we adopt the second approach: allowing a transformation by removing constraints (or subset of them) from B. Figure 18 explains this intuition. Suppose we identify such a constraint for

[4]We also show at the end of Section 5 another scenario with similar issue.

some transformation-effect tr and remove it (ex: $a3$ in Figure 18), giving us resultant transformed model M. To ensure this is the desired model, we need to prove the following:

— The resultant model M allows more behaviors: *Weak*.
— The resultant model M allows the desired transformation-effect tr for all pre-traces: *Sound*.
— The resultant model M preserves the safety of existing transformation-effects: *Complete*.

Relational notations. Given a binary relation R, let $R^{-1}, R^{?}, R^{+}$ represent inverse, reflexive closure, and transitive closure, respectively. Let $[E]$ represent the identity relation on set E. Let $R1;R2$ represent left sequential composition of two binary relations. Let $[A];R;[B]$ represent the relation $R \cap (A \times B)$. Lastly, we say the composition $R1;R2$ *forms a cycle* or $R1;R2$ *cycle* if there exists a cyclic path given by a non-empty relation $[a];R1;[b];R2;[a]$.

Assumptions over Models. Before we formally specify and go over each of the above parts to prove, we first state our assumptions on memory models. Each memory model we consider will place a constraint prohibiting the existence of specific cycles in the graph. We assume that each such constraint of the memory model can be represented as a set of irreflexivity constraints over binary relations instead.[5] For example, the constraint $(\mathrm{po} \cup \mathrm{rf})^{+}$ acyclic can instead be specified as $(\mathrm{po} \cup \mathrm{rf})^{+}$ irreflexive. We additionally assume no constraint to be redundant: one cannot be contained within another. For example, the constraint $(\mathrm{po} \cup \mathrm{rf})^{+}$ irreflexive is contained within the constraint $(\mathrm{po} \cup \mathrm{rf} \cup \mathrm{mo})^{+}$ irreflexive.

4.2 Weak and Sound

A memory model is *weaker* than another if it allows more concurrent behaviors. This can be formally expressed as follows.

Definition 4.1. A memory model W *weaker* than memory model B ($\mathrm{weak}\langle W, B\rangle$) if every execution consistent in B is also consistent in W.

$$\forall E \,.\, c_B(E) \implies c_W(E).$$

Proving weakness for our methodology is not required, as by construction, we remove a constraint from the base model.

A transformation-effect is *sound* for a memory model if it is safe for any pre-trace. This can be formally expressed as

Definition 4.2. A transformation effect tr is *sound* w.r.t. memory model M $sound(M, tr)$ if

$$\forall P \,.\, psafe(A, tr, P).$$

Proving sound naively would require proving safety for all possible pre-traces. Fortunately, it suffices to quantify pre-traces using the constraints of given memory model, reasoning over their set of consistent/inconsistent candidate executions [34].

4.3 Complete

In Section 3.3, we saw how transformations can be decomposed as a series of transformation-effects. Therefore, in order to preserve safety of transformations from one model to another, we must preserve the safety of the composed effects.

Definition 4.3. A memory model M is *Complete* w.r.t. memory model B ($\mathrm{comp}\langle M, B\rangle$) if

$$\forall P \mapsto_{tr} P' \,.\, psafe(B, tr, P) \implies psafe(M, tr, P).$$

[5]Many existing models are generally represented in the form of acyclic constraints (e.g., C11, RC11, Java). Identifying an equivalent set of irreflexivity constraints which is suitable enough for our purpose is left to future work.

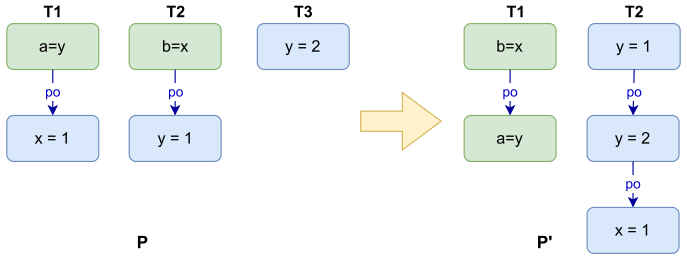

Fig. 19. Example of a non-trivial transformation-effect $P \mapsto P'$, which can be decomposed into several forms of de-ordering, reordering, and inlining transformation-effects.

Definition 4.3 quantifies over all possible transformation-effects. Therefore, a proof for it will require quantifying over them. One approach can be to decompose each transformation into a series of elementary effects (e.g., peephole optimizations), which can defined by us. However, this may turn out to be infeasible when quantifying over effects like those in Figure 19. Such a transformation involves several variants of de-ordering ($\mathrm{po}^- \neq \phi$), inlining ($\mathrm{po}^+ \neq \phi$), and reordering effects. To add, the set of effects may vary depending on how we decompose them. Isolating our concern case-wise over each set in Definition 3.7 is also not practical: even simple reordering involves both po^- and po^+ in unison.

Using models B and M. We lay out an approach to proving Complete by quantifying tr using the constraints of the models. We note that the set of transformation-effects we require proving Complete, by Definition 4.3, are those safe under B. From a contrapositive lens, we instead require the set of transformation-effects tr unsafe under M. Each of these tr unsafe under M, by Definition 3.11, tell us that some consistent execution E' in the transformed pre-trace has no comparable consistent execution E in the original. The contraposition then requires the transformation-effect to also be unsafe under B. By weak$\langle M, B\rangle$, proving it would require us to handle only the case when E' is also inconsistent under B. If this information suffices to prove that every such tr is unsafe under B, then we are done. The above idea culminates into the following theorem.

Theorem 4.4. *Consider memory models B, M s.t. weak$\langle M, B\rangle$. Consider any pre-trace P and all transformation-effects $P \mapsto_{tr} P'$ such that*

$$\exists E' \in [\![P']\!]_M \,.\, \forall E \in \langle P\rangle \,.\, E \sim E' \implies E \in I\langle P\rangle_M,$$

If $\forall tr \,.\, \neg c_{B\setminus M}(E') \implies \neg psafe(B, tr, P)$ then comp$\langle M, B\rangle$.

Proof. We prove this by contraposition. Assuming $\neg comp\langle M, B\rangle$, we have

$\exists P \mapsto_{tr} P' . \neg(psafe(B, tr, P) \implies psafe(M, tr, P))$.

$\to psafe(B, tr, P) \wedge \neg psafe(M, tr, P)$. (Definition 4.3)

$\to \neg([\![P']\!]_M \sqsubseteq [\![P]\!]_M)$. (Definition 3.11)

$\to \exists E' \in [\![P']\!]_M . \nexists E \in [\![P]\!]_M$ s.t. $(E' \sim E)$. (Definition 3.11)

$\to E \in I\langle P\rangle_M$. ($\langle P'\rangle \sqsubseteq \langle P\rangle$)

$\to E \in I\langle P\rangle_B$. (Definition 4.1)

$\to E' \in I\langle P'\rangle_B$. (Definition 3.11,4.1)

$\to \neg c_{B\setminus M}(E')$.

Which means $\forall tr \,.\, \neg c_{B\setminus M}(E') \implies \neg psafe(B, tr, P)$ is false. Hence, proved. □

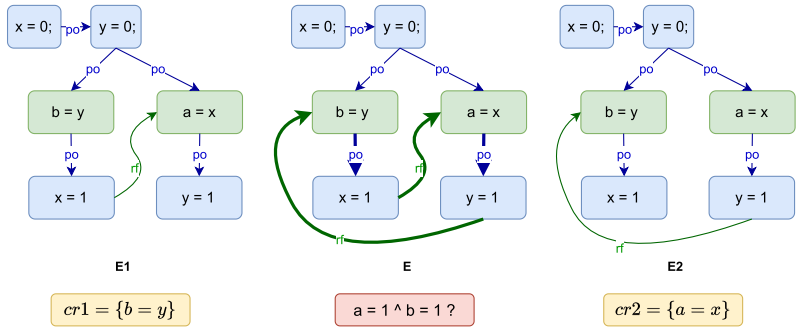

Fig. 20. Removing the rf relation with $b = y$ (left) or with $a = x$ (right) from the inconsistent execution E gives us consistent executions $E1$ and $E2$ respectively under memory model $(\text{po} \cup \text{rf})^+$ acyclic.

4.3.1 Proving Complete via Crucial Sets. Theorem 4.4 helps us prove M Complete w.r.t. B by simply identifying those transformation-effects which requires proving unsafe under base model B. On a high level, we do this by identifying another $E'_t \in [\![P']\!]_B$ which has no comparable $E_t \in [\![P]\!]_B$. In order to derive such an execution, we manipulate specific rf relations of E'. The idea is that for a certain set of E' inconsistent under B (at least under the memory models we consider), we can identify reads which, on the removal of their associated rf relation, give us a consistent execution under B (albeit not well-formed). Figure 20 shows the example of an inconsistent execution E in the middle for a memory model with constraint $(\text{po} \cup \text{rf})^+$ acyclic. We can see how removing the rf relation with $b = y$ or $a = x$ gives us resultant executions $E1$ and $E2$ both of which are consistent under the same model.

Formally, we denote the reads associated with these rf relations as *crucial* w.r.t. a memory model for a candidate execution. The collection of such reads is called a crucial set.

Definition 4.5. Given a execution E inconsistent under memory model M, a set $cr \subseteq r(p(E))$ is a *crucial set* if the execution $E' \sim E$ such that

a) $\text{rf}(E') = \text{rf}(E) \setminus \{[a]; \text{rf}; [b] \mid b \in cr\}$. b) $\text{mo}(E') = \text{mo}(E)$.

is consistent under M.

Recalling example in Figure 20, both $cr1$ and $cr2$ are crucial sets.

Remark 11. A crucial set may not always exist: the inconsistent execution in Figure 13 has no crucial set w.r.t. example memory model.

We now link crucial sets to the constraints of the memory model, which we recall will be in the form of irreflexivity constraints on binary relations. Violation of any such constraint would imply the existence of a cycle in the graph, like that in Figure 20. Therefore, the crucial reads would be those whose rf relations enable such a cycle to exist.

Definition 4.6. We define cra as a function that takes in a binary relation s between memory events in E and returns a set of reads r such that without each of its associated rf, the relation is irreflexive.

$$\forall r \in cra(s) \;.\; r \notin st(p(E)) \implies s \text{ irreflexive.}$$

With cra, we can construct crucial sets for each E' inconsistent under B, choosing at most one read from each cyclic path formed which violate the constraints of B. Recall the purpose of

identifying such a crucial set is to derive another E'_t such that $c_B(E'_t)$. We can do this if it is possible to reassign the rf relations of the reads in the crucial set to give us our desired E'_t. For the models we consider, we show this is possible by first removing all the rf relations from one of its crucial sets, followed by incrementally adding *consistent* rf relations one by one. We term this property as *piecewise consistent* that a memory model satisfies for any execution E. Formally,

Definition 4.7. A memory model M is *piecewise consistent* ($\text{pwc}(M)$), if for any execution E such that

- $\neg\text{wf}(E)$.
- rf^{-1} functional.
- $\forall loc\ .\ max(loc) \neq \phi$.
- $c_M(E)$.
- mo total order.

we have a candidate execution E_t such that

- $c_M(E_t)$.
- $\text{mo}(E) = \text{mo}(E_t)$.
- $p(E) = p(E_t)$.
- $\text{rf}(E) \subset \text{rf}(E_t)$.

We show in Section 5.3 how the above approach is used to prove Complete for a memory model M w.r.t. *SC*.

5 Toward Concrete Models

We now apply our formalism for concrete axiomatic models. First, we specify the axiomatic model of *SC* in the form of irreflexivity constraints (Section 5.1). This is followed by showcasing with our formalism why *TSO* is not Complete w.r.t. *SC* (Section 5.2). Next, we derive SC_{RR} as a model that allows independent read-read reordering over *SC*. We show that SC_{RR} is Weak, Sound, and Complete for transformation-effects not involving write-elimination (Section 5.3). Then, we show that SC_{RR} remains Weak, Sound, and Complete (barring write-eliminations) w.r.t. *SC* on inclusion of *fences* (frr) and ***read-modify-write*** (rmw) events in the language (Section 5.4). Lastly, we conclude by showing the need for barring write-eliminations, followed by directions to include them (Section 5.5).

Additional relations. First, let ***read-from-internal*** (**rfi**) represent the subset of rf such that both the write and read are of the same thread, and let ***reads-from-external*** (**rfe**) be the rest. Second, let ***happens-before*** (**hb**) be the transitive closure of program order and reads-from relations $(\text{po} \cup \text{rf})^+$. Third, let *memory-order-loc* ($\text{mo}_{|loc}$) represent the mobetween writes to same memory. Finally, let ***reads-before*** (**rb**) represent the sequential composition $\text{rf}^{-1}; \text{mo}_{|loc}$, which relates reads to writes which have taken place before it to same memory in an execution.

5.1 Sequential Consistency (SC)

We adopt the axiomatic format of *SC* as in [16], which consists of the following set of consistency rules for any execution E

a) mo strict total order.
b) hb irreflexive.
c) mo; hb irreflexive.
d) rb; hb irreflexive.
e) rb; mo; hb irreflexive.

Figure 21 shows the two executions of SB discussed earlier in Figure 3. The outcome to the left has no rule of *SC* is violated, thus deeming it consistent under *SC*. Whereas the outcome to the right is inconsistent, the cycle $[a = x]; \text{rb}; [x = 1]; \text{mo}; [y = 1]; \text{po}; [a = x]$ violates Rule (e).

We use Definition 4.6 to correlate *SC* with the notion of *crucial reads*. Recall that the cyclic path in Figure 21 right results in the sequential composition rb; mo; hb to be no longer irreflexive. Specifically, $[a = x]; \text{rb}; [x = 1]; \text{mo}; [y = 1]; \text{po}; [a = x]$ is non-empty.

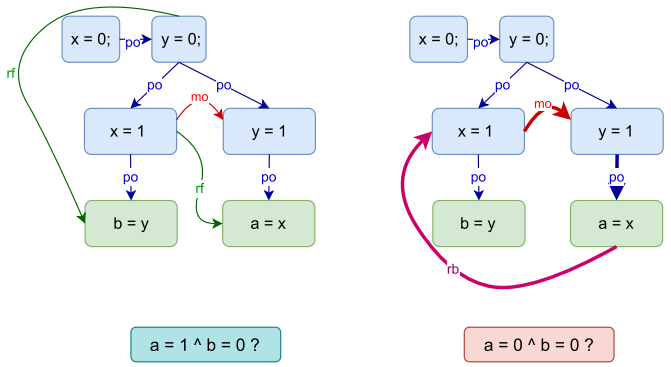

Fig. 21. Two outcomes of SB; one consistent (left) and another inconsistent (right) under *SC*.

Lemma 5.1. *For a given execution E, we have*

- $cra(mo) = \phi$.
- $cra(hb) = \{r \mid ([e]; rfe; [r]; hb; [e] \neq \phi) \vee ([e]; rfi; [r]; po; [e] \neq \phi)\}$.
- $cra(mo; hb) = \{r \mid [e]; mo; hb^?; rfe; [r]; hb; [e] \neq \phi\}$.
- $cra(rb; hb) = \{r \mid ([e]; rb; hb^?; rfe; [r]; hb; [e] \neq \phi) \vee ([r]; rb; hb; [r] \neq \phi)\}$.
- $cra(rb; mo; hb) = \{r \mid ([e]; rb; mo; hb^?; rfe; [r]; hb; [e] \neq \phi) \vee ([r]; rb; mo; hb; [r] \neq \phi)\}$.

Proof. (sketch) Consider the case for hb which is not irreflexive, i.e., represented by the relation hb cycle. Expanding hb, we get

$$
\begin{aligned}
&[e]; \text{hb}.\\
&\rightarrow [e]; \text{rfe}; [r]; \text{hb} \vee [e]; \text{rfi}; [r]; \text{po}.\\
&\rightarrow [e]; \text{rfe}; [r]; \text{po}; \text{hb}; \text{po} \vee [e]; \text{rfi}; [r]; \text{po}.
\end{aligned}
$$

Observe that for both cases, without the rf relation with r and e, the cycle will cease to exist. The sequential composition no longer exists, meaning the aforementioned hb cycle no longer exists.[6]

The other cases follow a similar line of argument. □

We refer readers to Appendix B for the proofs that the consistency rules of *SC* are not redundant, and that *SC* is piecewise consistent.

5.2 Revisiting *TSO* vs *SC*

We revisit our counter example on inlining which showcased TSO is not Complete w.r.t. *SC*. We do not derive *TSO* from *SC*, rather we use a similar existing formulation of the model in [16]. The fragment of *TSO* without fences *F* and *read-modify-writes/locks* have the following consistency rules for any execution *E*.

a) mo strict total order.
b) hb irreflexive.
c) mo; hb irreflexive.
d) rb; hb irreflexive.
e) rb; mo; rfe; po irreflexive.

The rules (a), (b), (c), (d) are the same as that for *SC*, whereas rule (e) is a subset of that in *SC*; which specifically allow the behaviors exhibited by FIFO SB of TSO. As an example, Figure 22 shows both the executions of SB shown for *SC* in Figure 21. Notice that the outcome on the right is also allowed; no rule of *TSO* is violated.

[6]For hb to be irreflexive, all such hb cycles must disappear or no longer exist.

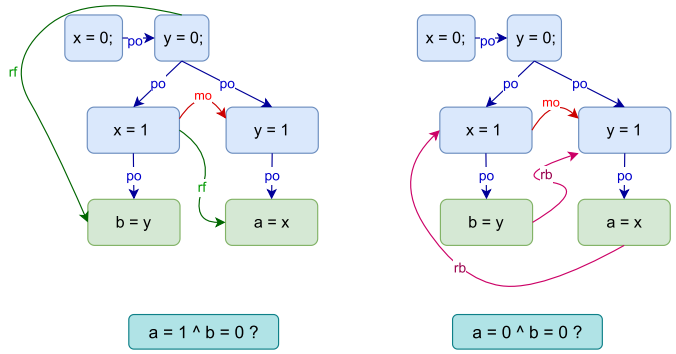

Fig. 22. Two outcomes of SB, both consistent under *TSO*.

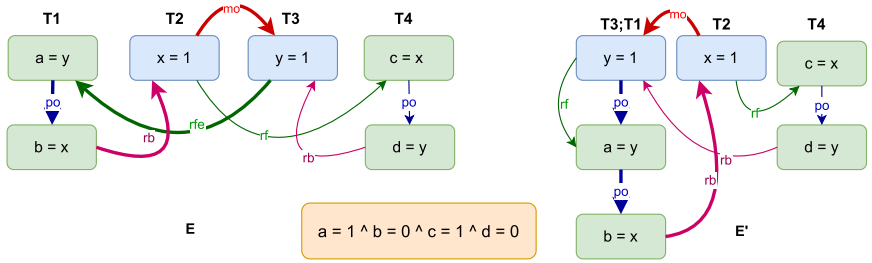

Fig. 23. For the given outcome (yellow box), E is inconsistent under both *SC* and *TSO*, but E' is only inconsistent under *SC*.

We show why *TSO* is not Complete w.r.t. *SC* using our formalism. We make use of Theorem 4.4 as follows:

- $B = SC$.
- $M = TSO$.
- $weak\langle SC, TSO\rangle$.
- $B \setminus M = \mathrm{rb}; \mathrm{mo}; \mathrm{po}$ irreflexive.

To prove Complete, we need to show all transformation-effects quantified by Theorem 4.4 are unsafe under *SC*. Even if one of these transformation-effects are safe instead, then by Definition 4.3, *TSO* is not Complete w.r.t. *SC*. Thread-inlining qualifies to be such an effect: Figure 23 shows two executions E (left) and E' as per our requirement. We can see that E (left) has $[b = x]; \mathrm{rb}; [x = 1]; \mathrm{mo}; [y = 1]; \mathrm{rfe}; [y]; \mathrm{po}$ cycle, violating rule (e) of both *SC* and *TSO*. However, for E' (right), we only have $[b = x]; \mathrm{rb}; [x = 1]; \mathrm{mo}; [y = 1]; \mathrm{po}$ cycle, which does not violate any rule of *TSO*, but violates rule (e) of *SC*. Since such an inlining-effect is sound under *SC*[7] [22], we have $\neg\mathrm{comp}\langle TSO, SC\rangle$.

5.3 SC + Reordering of Independent Read–Read (SC_{RR})

With Theorem 4.4, we identified transformation-effects using which we showed $\neg\mathrm{comp}\langle TSO, SC\rangle$. However, the main question remains; Is it practically possible to prove a given consistency model Complete w.r.t. another? If not for all transformation-effects, can we instead identify a significant set for which we can? In this section, we show that it is possible to do so. We first derive an axiomatic model from *SC* with the intent on allowing read-read reordering. We then show the resultant model is Weak, Sound, and Complete, the latter of which is restricted to effects not involving write elimination.

[7]We only require finding an instance of a pre-trace for which tr is safe under *SC*. In this case, it is safe for all pre-traces.

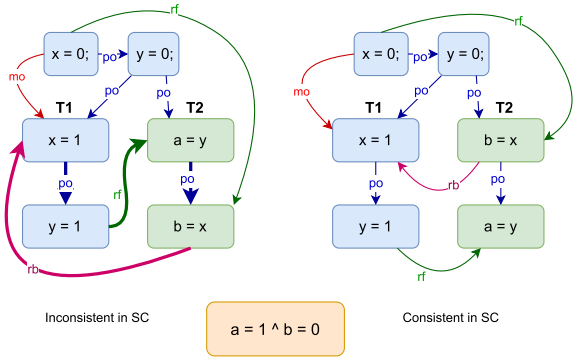

Fig. 24. The forbidden outcome under *SC* (yellow box) for the original pre-trace (left) allowed after reordering the two reads in $T2$ (right).

5.3.1 Transformation-Effect and Proposed Model. We first define reordering of adjacent independent reads as an effect. Let po_{imm} represent the program order between two adjacent events, i.e., no other memory event exists between them in program order ($[a];\text{po}_{imm};[b] \implies \nexists c\ .\ [a];\text{po};[c];\text{po};[b]$).

Definition 5.2. Consider a pre-trace P having two reads r_x, r_y such that $mem(r_x) \neq mem(r_y)$ and $[r_x];\text{po}_{imm};[r_y]$. Then, *reordering of adjacent independent reads* r_x, r_y is a transformation-effect $P \mapsto_{rr} P'$ such that

- $st^- = \phi$.
- $st^+ = \phi$.
- $\text{po}^- = \{(r_x, r_y)\}$.
- $\text{po}^+ = \{(r_y, r_x)\}$.

Transformation-effect in Definition 5.2 can be shown unsafe under *SC* using the example in Figure 24. The outcome (yellow box) in the original pre-trace is not possible under *SC*. The candidate execution (left) justifying it has $[b = x];\text{rb};[x = 1];\text{po};[y = 1];\text{rfe};[a = y];\text{po}$ cycle, thereby violating rule (d) of *SC*. However, the outcome is allowed under *SC* as a result of reordering the two adjacent reads $a = y$ and $b = x$. The corresponding candidate execution (right) shows no cycles violate the rules of *SC*.

To identify the constraint that prevents rr, note that the transformation-effect in Figure 24 was unsafe primarily due to a subset of rb;hb cycle that involved the po between the two reads (rb;hb;rfe;$[a = y];\text{po}_{imm};[b = x]$ cycle). More generally, we note the hb relation between any two independent reads plays a role in disallowing possible adjacent read–read reordering. Thus, the model SC_{RR} that allows reordering of adjacent independent reads is the following constraint on any execution E:

$$SC_{RR} = SC \setminus (\text{rb};\text{mo}^?;\text{hb}^?;\text{rfe};[r_{loc}(p(E))];\text{hb};[r_{!loc}(p(E))]\ \text{irreflexive}),$$

where $\text{mo}^?, \text{hb}^?$ here denote they are optional in the sequential composition, and *loc* represents any memory location. For brevity, we refer to the removed constraint as a_{rr}. To be specific, SC_{RR} is the derived model having the following consistency rules for any execution E.

a) mo strict and total.
b) hb irreflexive.
c) mo;hb irreflexive.
d) rb;hb $\setminus a_{rr}$ irreflexive.
e) rb;mo;hb $\setminus a_{rr}$ irreflexive.

Rules (a), (b), (c) are the same as that of *SC*, whereas rules (d) and (e) are ones which require the removal of a_{rr}.

5.3.2 Sound and Complete. We first show SC_{RR} is Sound.

Theorem 5.3. *For transformation-effect rr in Definition 5.2, we have sound*(SC_{RR}, rr).

Proof. (sketch) We prove this by contradiction.

— Suppose rr is not Sound.
— This implies there exists some pre-trace P, for which the transformation-effect $P \mapsto_{rr} P'$ is unsafe ($\neg psafe(M, rr, P)$).
— From Definition 3.11, this would mean there exists some consistent execution $E' \in [\![P']\!]_{SC_{RR}}$ for which there does not exist any comparable execution $E \in [\![P]\!]_{SC_{RR}}$.
— Since for all well-formed executions, we assume $\langle P' \rangle \sqsubseteq \langle P \rangle$, we can infer that every such comparable E is inconsistent under SC_{RR}.
— Considering r_x, r_y to be two adjacent reads which have been reordered, we can infer that $[r_x]; \mathrm{po}_{imm}; [r_y]$ is responsible in E being inconsistent.
— The po_{imm} relation between the reads plays a role in violating some rule of SC_{RR}.
— Whereas on reordering them, E' is consistent instead.

We show that this is never true; for any such E, we must also have E' inconsistent under SC_{RR}. As a sample, let us say the Rule (b), i.e., hb irreflexive does not hold in E, i.e., hb forms a cycle. Expanding hb, we get the following:

$$
\begin{aligned}
&\mathrm{hb}.\\
&\rightarrow \mathrm{hb}; [r_x]; \mathrm{po}_{imm}; [r_y]; \mathrm{hb}.\\
&\rightarrow \mathrm{hb}; [r_x]; \mathrm{po}_{imm}; [r_y]; \mathrm{po}.\\
&\rightarrow \mathrm{hb}; \mathrm{po}; [r_x]; \mathrm{po}_{imm}; [r_y]; \mathrm{po} \ \vee \ \mathrm{hb}; \mathrm{rfe}; [r_x]; \mathrm{po}_{imm}; [r_y]; \mathrm{po}.\\
&\rightarrow \mathrm{hb}; \mathrm{po}; [r_y] \ \vee \ \mathrm{hb}; \mathrm{rfe}; [r_x]; \mathrm{po}.
\end{aligned}
$$

We can see that hb forms a cycle independent of the po_{imm} between r_x and r_y, allowing E' to also be inconsistent due to Rule (b). Showing the same for all possible violated rules of SC_{RR} helps us conclude we indeed have *sound*(SC_{RR}, rr). The rest of the cases are in Appendix C. □

Next, we show that SC_{RR} is Complete w.r.t. SC for a significant set of transformation-effects. We use the result from Theorem 4.4 for our purpose as follows:

- $B = SC$.
- $M = SC_{RR}$.
- $weak\langle SC, SC_{RR} \rangle$.
- $B \setminus M = a_{rr}$.

Recall from Section 4.3 that our objective is to derive another E' of the transformed pre-trace which is consistent under SC, but has no consistent execution E from the original pre-trace. For this, we make use of cra (Definition 4.6) as follows. We first note the relation between compositions violating a_{rr} and four other sequential compositions which may not be irreflexive.

Proposition 5.4. *Let rr be the composition* $rb; mo^?; hb^?; rfe; [r_x]; hb; [r_{y \neq x}]$ *which violates constraint* a_{rr}*, i.e., it forms a cycle. Then, we have the following:*

- $cra(rr) \notin cra(rb; mo^?; po; [r_x])$.
- $cra(rr) \notin cra(rb; mo^?; po; [r_y])$.
- $cra(rr) \notin cra(rb; rfe; [r'_x]; po; [r_x])$.
- $cra(rr) \notin cra(rb; rfe; [r'_y]; po; [r_y])$.
- $cra(rr) \notin cra(mo; rfe; [r_x]; po)$.
- $cra(rr) \notin cra(mo; rfe; [r_y]; po)$.
- $cra(rr) \notin cra(rfi; [r_x]; po)$.
- $cra(rr) \notin cra(rfi; [r_y]; po)$.

We then show that for our purpose, we have at least one of the above compositions is true in E, as described in Theorem 4.4; correlating it with violating constraints common to both SC and SC_{RR}.

Lemma 5.5. *Consider transformation-effects $P \mapsto_{tr} P'$ defined by Theorem 4.4 not involving write-elimination ($st^{-} \cap w(P) = \phi$). Consider also, from Theorem 4.4, the corresponding execution $E' \in \langle P' \rangle$ which is consistent under SC_{RR} but inconsistent under SC. Then, at least one of the following is true for any $E \in \langle P \rangle$ where $E \sim E'$:*

a) $rfi; po$ *cycle.*

b) $mo; po$ *cycle.*

c) $mo; rfe; po$ *cycle.*

d) $rb; mo^{?}; po$ *cycle.*

e) $rb; rfe; po$ *cycle.*

Proof. (sketch) By Theorem 4.4, we know that any E such that $E \sim E'$ is inconsistent under SC_{RR}. Going case wise for each possible violated constraint of SC_{RR}, we show that it reduces to one of the above mentioned compositions. We refer the readers to the Appendix C for the detailed proof. □

Finally, using the above, we can construct and compare crucial sets for both E' and E, allowing us to show SC_{RR} is complete w.r.t. SC for all effects that do not involve write elimination.

Theorem 5.6. *For all transformation-effects not involving write elimination ($st^{-} \cap w(P) = \phi$), $comp\langle SC_{RR}, SC \rangle$.*

Proof. (sketch) Note that each relation that forms a cycle as specified in Lemma 5.5 also violates one of the constraints of SC. From Proposition 5.4, we can then derive a crucial set w.r.t. SC for E' and can show it isn't the same for E. Using $pwc(SC)$, we can derive a pair of candidate execution pairs $E_t \sim E'_t$ such that $E'_t \in [\![P']\!]_{SC}$ and $E_t \in I\langle P \rangle_{SC}$. Since no write-elimination occurs, by Definition 3.12 the transformation-effect tr is unsafe under SC. Finally, by Theorem 4.4, we have $comp\langle SC_{RR}, SC \rangle$. We refer the readers to Appendix C for the detailed formal proof. □

5.4 Read-Modify-Writes and Fence Instructions

So far, we have only addressed programs that have simple memory read/write events. In practice, most concurrent algorithms also rely on rmw instructions [5]. These are memory events which perform a read followed by a write in one atomic step (e.g., compare-and-swap, fetch-and-add). To add, the model SC_{RR} does not allow us to forbid any form of read-read reordering. Both these requirements can be fulfilled respectively by adding an rmw and a fence to the language.

Let $rmw(a, x, v)$ represent a read from shared memory x into thread-local variable a, followed by a write of value v to it and $rmw(P)$ return the rmw events of pre-trace P. Let f_{rr} represent the fence event and $frr(P)$ return the fence events of pre-trace P.

5.4.1 Including rmw and f_{rr} Rules. First, we consider rmw to be both reads and writes - $rmw(P) = r(P) \cap w(P)$. This ensures all existing rules of SC and SC_{RR} also apply for any rmw event. Second, we add another rule for *rmw atomicity*: no memory event must appear to occur between the read and write of this event in any execution. From [16], this requirement translates to forbidding outcomes like those in Figure 25 left. Here, we note that $a = 0$ implies the read has taken place before the write $y = 2$. However, the final value of y being 1 indicates that $y = 2$ took place before $y = 1$. Such an execution has $[rmw(a, y, 1)]; \mathrm{rb}; [y = 2]; \mathrm{mo}$ cycle. Thus, for SC and all models derived from it, the rule of atomicity would be

$$\mathrm{rb}; \mathrm{mo} \text{ irreflexive.}$$

In order to disallow reordering two independent reads under SC_{RR}, our idea is to place an rmw or fence f_{rr} event in between them. Concretely, if there is an rmw event between two read events

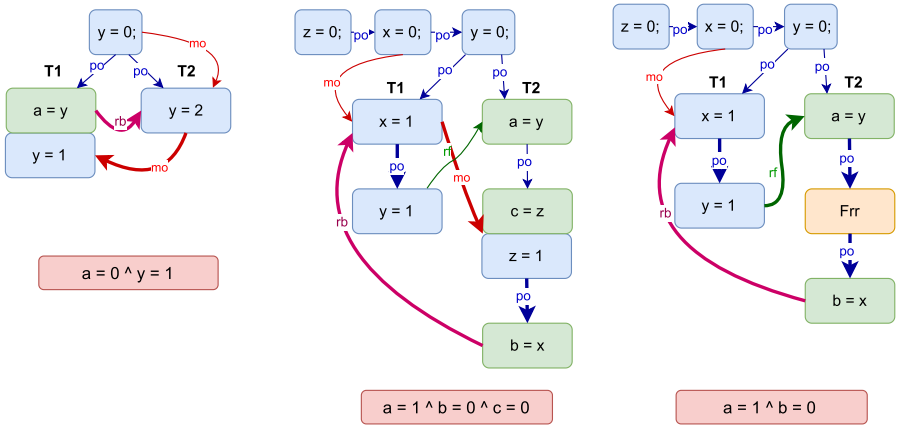

Fig. 25. The role of rmw and fences in forbidding outcomes violating atomicity (left) and preventing reordering of independent reads (middle, right).

in program order, we do not wish to permit their reordering. Since mo also involves *rmw* events, we need to add the following rule:

$$\mathrm{rb};\mathrm{mo}^{?};[rmw(p(E))];\mathrm{po};[r(p(E))] \text{ irreflexive.}$$

This translates to disallowing the outcomes like those in Figure 25 middle. The corresponding candidate execution has $[a = y];\mathrm{po};[rmw(c,z,1)];\mathrm{po};[b = x]$. The outcome $a = 1 \wedge b = c = 0$ is inconsistent under SC_{RR} as $[b = x];\mathrm{rb};[x = 1];\mathrm{mo};[rmw(c,z,1)];\mathrm{po}$ cycle violates atomicity.

Similarly, if a fence event f_{rr} exists between two reads r_y, r_x in program order, reordering the two reads should not be allowed. Using the same idea for *rmw*, the rule to add is similar to a_{rr}, except replacing *rmw* with a fence instead, giving us the following consistency rule given any execution E

$$\mathrm{rb};\mathrm{mo};\mathrm{rfe};[r_{loc}(p(E))];\mathrm{po};[frr(p(E))];\mathrm{po};[r_{!loc}(p(E))] \text{ irreflexive.}$$

We refer to the fence rule as a_{frr} for brevity. This translates to disallowing the outcomes like those in Figure 25 right. The corresponding candidate execution has $[a = y];\mathrm{po};[f_{rr}];\mathrm{po};[b = x]$, and the outcome $a = 1 \wedge b = 0$ is inconsistent under SC_{RR} as $[b = x];\mathrm{rb};[x = 1];\mathrm{hb};[a = y];\mathrm{po};[f_{rr}];\mathrm{po}$ cycle violates our new rule.

Our resultant model of SC_{RR}, with the inclusion of the above rules, is

a) mo strict and total.
b) hb irreflexive.
c) mo;hb irreflexive.
d) rb;hb $\setminus a_{rr}$ irreflexive.
e) rb;mo;hb $\setminus a_{rr}$irreflexive.
f) rb;mo irreflexive.
g) a_{frr}.

The rule for *rmw* is already a part of the above SC_{RR} if we restrict a_{rr} specifically to plain reads $(r(P) \setminus rmw(P))$. Therefore, we do not need to add any new rule for it. Note also that a_{frr} will only be required for SC_{RR}; the fence instruction is equivalent to a no-op under *SC*, making the rule redundant.

5.4.2 Weak, Sound, Complete? The addition of rmw and read-read-fences along with their respective consistency rules for *SC* and SC_{RR} do not change our results. SC_{RR} is still Weaker than *SC*; the axiom a_{frr} is redundant for *SC*. For Sound, we need to address two more cases based on the added axioms of atomicity and fences.

Theorem 5.7. *For the SC_{RR} model with the constraints (f) and (g) and transformation-effect rr as in Definition 5.2, we have sound(SC_{RR}, rr).*

Proof. Recall that in order to prove the effect $P \mapsto_{rr} P'$ is sound, we need to show that every execution $E \in \langle P \rangle$, which is inconsistent due to reads which are reordered, has a comparable execution $E' \in \langle P' \rangle$ which is also inconsistent. In addition to the existing proof cases of Theorem 5.3, two additional cases related to *rmw* and f_{rr} are to be addressed.

— Case 1: rb;mo cycle. There is no po involved in the composition of this relation. So none of the po edges matter.

— Case 2: a_{frr} violated. This means there exist reads r_x, r_y and a fence f_{rr} such that

$$\begin{aligned} &\mathrm{rb};\mathrm{mo};\mathrm{rfe};[r_y];\mathrm{po};[f_{rr}];\mathrm{po};[r_x] \text{ cycle.} \\ \rightarrow\ &\mathrm{rb};\mathrm{mo};\mathrm{rfe};[r_y];\mathrm{po}_{imm};[r_z];\mathrm{po};[f_{rr}];\mathrm{po};[r_x] \vee \\ &\mathrm{rb};\mathrm{mo};\mathrm{rfe};[r_y];\mathrm{po};[f_{rr}];\mathrm{po};[r_z];\mathrm{po}_{imm};[r_x] \text{ cycle.} \end{aligned}$$

For both cases, we note that $[r_y];\mathrm{po}_{imm};[r_z]$ as well as $[r_z];\mathrm{po}_{imm};[r_x]$ are not essential to violate a_{frr}.

Thus, by contradiction, we have $sound(SC_{RR}, rr)$. □

Show SC_{RR} with the new rules remains Complete w.r.t. *SC* involves a little more work. First, we note relations between $cra(rr)$ and some additional compositions which form a cycle.

Proposition 5.8. *Let rr be the composition rb; $mo^?$; $hb^?$; rfe; $[r_x]$; hb; $[r_{y\neq x}]$ which violates constraint a_{rr}, i.e., it represents a cyclic path. Then, we have the following:*

- $cra(rr) \nsubseteq cra(rb; mo)$.
- $cra(rr) \nsubseteq cra(mo; rf)$.

Second, we have three more compositions that may not be irreflexive in any execution *E* as specified by Theorem 4.4.

Lemma 5.9. *Consider transformation-effects $P \mapsto_{tr} P'$ defined by Theorem 4.4 not involving write-elimination ($st^- \cap w(P) = \phi$) and the corresponding execution $E' \in \langle P' \rangle$ consistent under SC_{RR} but inconsistent under SC. Then, on addition of f_{rr} and rmw events and their consistency rules, the following may also be true for any $E \in \langle P \rangle$ where $E \sim E'$:*

a) mo; rf cycle.

b) rb; mo cycle.

c) rb; mo; rfe; po; $[f_{rr}]$; po cycle.

Third, remember that proving SC_{RR} Complete w.r.t. *SC* does not involve transformation-effects with f_{rr}: they are no-ops w.r.t *SC*. Accordingly, we identify the cases when a transformation-effect involves f_{rr}.

Lemma 5.10. *For transformation-effects $P \mapsto_{tr} P'$ as defined in Theorem 4.4 with no write elimination, and with corresponding pair $E \sim E'$ and $B = SC$, $M = SC_{RR}$, we have the following:*

$$a_{frr}(E) = false \implies ((f_{rr} \in st^-) \vee ([f_{rr}]; po; [r_x] \in po^-) \vee ([r_y]; po; [f_{rr}] \in po^-)),$$

where $r_x, r_y \notin st^-$.

Finally, we show that addition of f_{rr} and *rmw* events and their consistency rules (f) and (g) does not change the result of Theorem 5.6.

Theorem 5.11. *Given SC and SC_{RR} with rmw and f_{rr} events, for all transformation-effects not involving write elimination, we have comp$\langle SC_{RR}, SC \rangle$.*

The proofs of Lemma 5.9, 5.10 and Theorem 5.11 are in Appendix D.

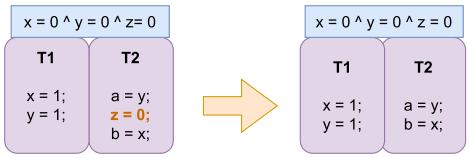

Fig. 26. The write $z = 0$ eliminated as dead-code on performing value range analysis.

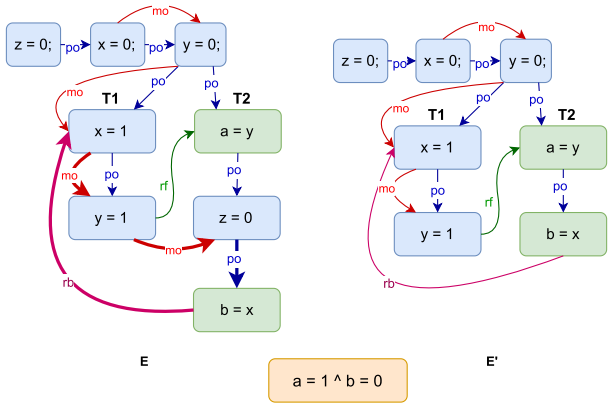

Fig. 27. The outcome $a = 1 \wedge b = 0$ (yellow box) forbidden under SC_{RR} for the original pre-trace due to E but allowed after write-elimination due to E'.

5.5 Write Elimination-Effects

We have proven SC_{RR} Complete w.r.t. SC only for transformation-effects without write-elimination. The reason for this can be explained via the counter example in Figure 26. Consider the original program on the left. The compiler, on performing value range analysis, can declare the write $z = 0$ as dead-code and remove it, resulting in the code to the right. While such a transformation-effect is safe under SC (we welcome to readers to check it themselves for all possible candidate executions), the same cannot be said for SC_{RR}.

Observe the two similar candidate executions $E \sim E'$ in Figure 27, the left of the original program's pre-trace P and the right for the transformed pre-trace P'. We can see that the behavior (yellow box) justified by E is inconsistent under SC_{RR}; the cycle $[b = x]; \mathrm{rb}; [x = 1]; \mathrm{mo}; [z = 0]; \mathrm{po}; [b = x]$ in E violates rule (e). Whereas E' is consistent under SC_{RR} for P'. To finish, note that even on changing the mo between $y = 1$ and $z = 0$, the resultant candidate execution still remains inconsistent under SC_{RR}, due to $[z = 0]; \mathrm{mo}; [y = 1]; \mathrm{rfe}; [a = y]; \mathrm{po}$ cycle violating constraint (c). Thus, by Definition 3.12, this transformation-effect is unsafe under SC_{RR}.

We now have an example where a write-elimination effect is safe under SC but unsafe under SC_{RR}. In order to allow such an effect, we may require adding another rule to SC_{RR}, which will need to prohibit the behavior represented by E'. However, this additional constraint would be prohibiting the message passing weak outcome E', effectively making reordering of independent reads unsound once again. A feasible way then to incorporate such effects would be to observe the role of a write in constraints (d) and (e) of SC_{RR}. We conjecture that it plays an additional role similar to that of constraint (g), that involving fence event f_{rr}. While elimination of a write necessarily involves removal of certain mo relations, those that are present to restrict reordering can instead be taken over by an f_{rr} event. We keep this as a potential future work.

6 Discussion

We have established a representation of program transformations which allow us to decompose them as effects across execution traces. This enabled us to reason about safety of transformations w.r.t. memory models in terms of elementary transformation-effects, allowing us to formally specify and reason about compositional property such as Complete, which guarantees safety of program transformations from one model to another. We showcased the above formalism's application comparing two models: SC_{RR} with respect to SC, showing SC_{RR} Weak, Sound, and Complete w.r.t. SC for all effects barring write elimination.

6.1 A New Property for Programming Models

Definition 3.7 provides a feasible way to represent transformations. Coupled with Theorem 4.4, we have a process of representing the space of compiler optimizations whose safety can be guaranteed from one model to another.

"DRF-SC" like property for Optimizations. Such a guarantee is analogous (and even complementary) to properties that allow programmers to rely on interleaving semantics. One such property is related to data-races, executions of programs that may lead to non-deterministic program behaviors. Models of C++, Java, and so on guarantee SC reasoning for programs that are data-race-free (DRF-SC) [1]. This allows reasoning about programs relying on sequential interleaving, setting aside much of the complexity of weaker memory semantics. However, DRF-SC may not guarantee a model is Complete w.r.t. SC: one cannot rely on SC to design safe optimizations for the weaker language model. The core intuition is that optimizations, though safe under SC, may introduce data-races when considered under the corresponding language model. Proving models Complete w.r.t. SC helps address this issue.

Note that DRF is a constraint on the shape of programs, which can be inferred using binary relations in a program's corresponding candidate execution [1]. Analogously, our constraint for Completeness is on the shape of program transformations. For instance, if a given compiler performs elimination only if memory writes are adjacent (e.g., $[x = 1];[x = 2] \mapsto [x = 2]$), then for programs that do not have adjacent writes, any optimization done by the compiler under SC can also be done under SC_{RR}. As another example, consider Corollary 6.1 as shown below, which states when a transformation-effect involves read-write reordering, when considering SC_{RR}.

Corollary 6.1. *For transformation-effects quantified by Theorem 4.4 with pair $E \sim E'$ and $B = SC$, $M = SC_{RR}$, we have the following for E if no write elimination-effect exists:*

$$(rfi; [r_x]; po \cup mo; rfe; [r_x]; po) \ \ cycle \implies [r]; po; [w] \in po^{-}.$$

where $r_x \in (r(P) \cap r(P'))$.

Via Corollary 6.1, we can infer that for programs not having read-write syntactic order, we have SC_{RR} Complete w.r.t. SC.

Extending to IR models. Our property of Complete also extends to a more general case, especially where language IR models exist. Many compiler optimizations are done relying on the IR models, isolating oneself from the complexity of the language model. The underlying assumption is that such optimizations must also be safe under source language model. This may not be true in general. If our IR model is SC and language model SC_{RR}, then removing redundant/unused writes may not be safe. In a scenario where the language model is C++, and the IR model is that of LLVM, proving $\text{comp}\langle C++, LLVM\rangle$ would guarantee isolated reasoning.

6.2 A New Perspective Toward Code Generation

Our proposed methodology of removing constraints of a model can also help identify effective compiler mappings to hardware. For instance, we can easily see $SC \setminus \mathrm{rb};\mathrm{mo};[w_{loc}(P) \setminus u(P)];\mathrm{po};[r_{!loc}(P)]$ irreflexive = TSO, which gives us

$$SC \setminus TSO = \mathrm{rb};\mathrm{mo};[w_{loc}(P) \setminus u(P)];\mathrm{po};[r_{!loc}(P)] \text{ irreflexive.}$$

This can be used to infer the following:

$$\forall E \in I\langle P\rangle_{SC} \,.\, c_{SC\setminus TSO}(E) \implies [\![P]\!]_{SC} = [\![P]\!]_{TSO}.$$

If our language model is SC, and our target hardware language to compile is TSO, then the above inference can be used for efficient code generation. If the source program does not contain any thread-local write-to-read syntactic order, the mapping of memory events is one-to-one. This is true due to the following proposition.

Proposition 6.2.

$$[w_{loc}(P) \setminus (w_{init} \cup u(P))];po;[r_{!loc}(P)] \notin po(P) \implies \forall E \in I\langle P\rangle_{SC} \,.\, c_{SC\setminus TSO}(E).$$

Otherwise, between every such a write-to-read syntactic order, a fence (or a stub rmw) event needs to exist to ensure correct compilation.

7 Related Work

The earlier implications of memory consistency models on optimizations were observed on those performed by hardware [2]. The issue was different, and entailed coming up with a formal specification of hardware memory consistency, much of which were (and still quite a bit) described in informal prose format [24, 31]. An axiomatic formulation, as opposed to operational model, greatly helped in clarifying hardware designer intentions [4]. The same axiomatic formulation helped in specifying programming language consistency models.

The implication of language consistency models was only later identified, facing along with informal prose, the non-trivial effects on compilation [7, 14, 17, 29, 35]. Earlier criticisms of Java were primarily based on informal/unclear specifications that were hard to understand, naturally leading to most JVMs violating them [30]. Optimizations like bytecode reordering, copy propagation, thread-inlining performed on Java Programs violated the specifications [20, 29]. A comprehensive study on the impact of optimizations exposed many SC optimizations were rendered simply unsafe; examples being redundant read/write eliminations/introductions and reordering independent memory accesses [32]. C++ also faced similar issues of informal/unclear specifications, much of which is now properly clarified and formalized [7]. However, the detrimental impact on optimizations were being slowly acknowledged, revealing simple optimizations such as register spilling and redundant write elimination unsafe in programs with benign data-races [9]. Testing product compilers such as GCC exposed bugs that involved optimizations like redundant eliminations, reordering atomic/non-atomic code fragments, and redundant introductions, all of which were unsafe under the C11 memory model. A comprehensive study of the common class of optimizations disallowed in the C11 memory model exposed the consequence of allowing certain causal cycles in conjunction with the restrictions on non-atomic reads on compiler optimizations [33]. Validating LLVM optimizations for compiled C11 programs also exposed existing bugs related to shared memory accesses [10], some of which are still being discovered [19]. The JavaScript memory model also was subject to similar issues, with impacts due to unclear specification on hardware mapping and optimizations [14, 35].

To add, other complications with programming language models involve the infamous **out-of-thin-air** (**OOTA**) problem; semantics permitting behaviors which should not be possible by

programs [8]. Recent work concludes that a significant part of this problem can be reduced to tracking semantic-dependencies without disallowing optimizations that remove redundancies [6]. Addressing OOTA has resulted in several solutions, yet they do not entirely solve it, permitting behaviors which are still considered OOTA or not successfully able to permit all desired optimizations. Java was one of the first languages for which a solution was proposed, involving complex reasoning to justify behaviors involving causal cycles [20]. Although this was done catering to optimizations, it still disallows thread-inlining. The *promising semantics* proved to be a good solution; an operational model associating promises to writes which will guarantee to be visible to other threads at a later time in execution [15]. Recent proposals have also seen the use of *event structures*, a representation of programs as a collection of different possible execution paths. The **Modular Relaxed Dependencies** (**MRD**) model attempts to capture semantic dependencies by calculating a *co-product* from the event structure of the program[25]. However, this notion of semantic dependency turns out to be too strict, disallowing optimizations such as thread-inlining, and even certain forms of thread-local redundant read/write eliminations. Another compelling approach is to use an event structure to describe an operational memory model with optimizations included as operational steps[26]. These are a small set of per-thread modifications to the event structure that represent thread-local reordering and elimination optimizations. Such operational steps at the level of event structures coincidentally resemble our notion of (or can be a counterpart to) transformation-effects over pre-traces. However, their formalism is specific to C++11, and it falls short while handling elimination of atomic writes [27]. While they do have a representation of global optimizations via value range analysis, other inter-thread analyses that would permit thread-inlining, reordering, elimination, and introduction of memory events is not considered. We do believe their model is appropriate to reason with JIT compilers, something our pre-trace model does not currently handle. In practice, however, relying on such an operational model would incur a steep learning curve. The representation of optimizations departs significantly from the standard view of control-flow-graph transformations.

We note that the above solutions using event structure-based models or, more generally, multi-execution memory models are to handle the problem of OOTA, as opposed to our goal with pre-traces. The pre-trace model is meant to represent a control flow graph of the program and the effects as changes to the graph itself. We argue this is closer to a compiler designer's intuition of programs they wish to optimize. Moreover, our model itself is much more conservative w.r.t. the view of program behaviors, permitting more possible outcomes than the program can exhibit (recall example in Figure 11, the reads are not associated with any write value yet). This conservative approximation is often desired to design optimizations that are portable and reliant on correctness of flow-analyses (which in our case, proving Complete between two models). While the focus of our work does not inherently deal with OOTA, we strongly believe our approach to decomposing an optimization into observable effects on pre-traces can help address this issue. We *conjecture* the problem of allowing optimizations involving redundancies can be separated from the OOTA problem, although we leave this for future work. Our view is that the idea of event structures flow naturally from trace-level reasoning of programs, whereas the idea of pre-traces and effects flow naturally from reasoning about safety of traditional compiler optimizations via abstract representation of programs. It would be unsurprising to us if there exists a convergence point where both these models meet.

While consistency models such as *SC* do render optimizations unsafe, whether they affect performance of programs is still debatable. A compelling proposal suggests relying just on *SC*, claiming much of the optimizations responsible for performance are already allowed by the model [21]. While their proposal arguably simplifies the whole problem, it requires significant changes to enable hardware track changes to shared memory state in a concurrent execution. Such

changes have not been adopted to date by commercial hardware vendors, hinting toward their impracticality.

Addressing safety of optimizations using execution traces has been done before, but aspects of completeness for which we adopt it has not been addressed. Previous contributions are on optimizations performed on data-race-free programs and those which do not introduce any races in the process [32]. Moreover, the space of these optimizations are almost always involving adjacent thread-local events. Our work intends to place no such constraint, and instead guarantee safety of any future optimization from one model to another; be it thread-local or inter-thread.

Our design of pre-trace matching for identifying correct effects is quite similar to a validator designed to verify C11 optimizations [11]. In a way, our work generalizes this idea of matching in order to create a separation between thread-local (sequential) and concurrent semantics, enabling reasoning about correctness in a disjoint fashion. An interesting observation from this work is their heuristic approach to loops, that claims utilizing just two iterations suffices in guaranteeing no false positives. We intend to investigate this claim in future, albeit from a perspective of identifying the class of memory models for which this may hold.

Previous work in explaining the interaction between memory consistency and optimizations has been done [16]. They show that weak behaviors of *TSO* can be explained via local adjacent write-read reordering and read-after-write elimination. Their work also exposes weak behaviors of models such as **release-acquire** (**RA**) that neither can be explained via elementary thread-local transformations nor thread-inlining. Their result w.r.t. models like *TSO* are orthogonal to our notion of Complete, and what we refer to as *Optimality*. Optimality requires one to ensure the derived model is not weakened too much that it allows more optimizations than desired.

8 Conclusion

In this work, we proposed a representation of program transformations suitable enough to understand its compositional relation with memory consistency semantics. We achieved this by decomposing a transformation into a series of elementary effects on pre-traces, which represent an over-approximation of program behaviors. We separated the role of sequential semantics w.r.t. safety of transformations, formalized the desired compositional nature between memory models (*Complete*), followed by showing its practicality by proving SC_{RR} Complete w.r.t. *SC*. Our contributions expose the need for a new property between memory consistency models, which may prove useful when designing future optimizations for compilers, thread-local or global. Finally, our approach paves way to a new design methodology for programming language models, one that places emphasis on desired optimizations.

We believe our approach is generalizable enough, in that language models can be derived based on a specific set of desired optimizations. However, they must be in the form of syntactic changes that represent a transformation-effect. Optimizations like register allocation and certain cases of expression simplifications will be indistinguishable as they do not represent a syntactic effect at the level of pre-traces. We also assume loops are finite/bounded, i.e., they can simply be unrolled. While we can represent loop optimizations (e.g., loop-invariant code motion, loop unrolling) as syntactic effects, asserting their safety w.r.t. unbounded loops has not been addressed.

8.1 Future Work

An immediate follow up to our current work would involve addressing elimination effects. While we were able to prove SC_{RR} Complete barring write eliminations, we suspect the role of writes also plays the same role as f_{rr}, disallowing any read–read reordering. Thus, a workaround to guarantee write-elimination effects safe from *SC* would be to replace them with f_{rr} instead (any write elimination effect would incur a fence introduction).

In the future, we want to derive models over *SC* for other forms of elementary reorderings (read-write, write-read, write-write). An immediate strong candidate would be write-read reordering, as we can compare it directly to *TSO*, potentially helping us identify the space of optimizations for which *TSO* is Complete w.r.t. *SC*. We intend to further combine these reordering models, in hopes of identifying a suitable model that allows all forms of reordering, and is Complete w.r.t. *SC* for (possibly) all transformation-effects. Some of these combinations would closely resemble models like *TSO*, *PSO RMO* which are known to be used in practice. Thus, comparing our models with them will prove beneficial. In order to make these models practical, we would also need to ensure they are weakened *Optimally*, i.e., we do not allow more optimizations than what we desire.

Obtaining a model for the exact desired set may not always be trivial, noting that allowing effects in turn allows multiple desired optimizations. For example, SC_{RR} also permits the optimization "$a = x; c = y; b = x; \mapsto\ a = x; c = y; b = a;$" forwarding the load to x past the read to y. Forbidding such an optimization would potentially require adding constraints in addition to removing them. To add, proving models Complete may also require manipulation of mo relations along with rf, especially when inconsistent executions may not have a crucial set (e.g., when an execution is inconsistent under *SC* only due to mo; po cycle).

Finally, a promising direction would be to address the long-standing the OOTA problem. As stated in related work, we *conjecture* decomposing optimizations into effects on pre-traces will help address safety. This separates the problem of designing a memory model to prohibit OOTA, allowing compilers to perform optimizations identifying redundancies as desired.

Acknowledgments

We sincerely thank the reviewers and the associate editor for their detailed feedback and suggestions.

References

[1] Sarita V. Adve and Hans-Juergen Boehm. 2010. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM* 53, 8 (2010), 90–101. DOI: https://doi.org/10.1145/1787234.1787255

[2] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *Computer* 29, 12 (1996), 66–76. DOI: https://doi.org/10.1109/2.546611

[3] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. 1986. *Compilers: Principles, Techniques and Tools*. Pearson Education, Inc, 583–706.

[4] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. DOI: https://doi.org/10.1145/2627752

[5] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. 2011. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. *SIGPLAN Not.* 46, 1 (Jan 2011), 487–498. DOI: https://doi.org/10.1145/1925844.1926442

[6] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The problem of programming language concurrency semantics. In *Proceedings of the 24th European Symposium on Programming Languages and Systems, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015*, Jan Vitek (Ed.). Lecture Notes in Computer Science, Vol. 9032, Springer, 283–307. DOI: https://doi.org/10.1007/978-3-662-46669-8_12

[7] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 55–66. DOI: https://doi.org/10.1145/1926385.1926394

[8] Hans-Juergen Boehm and Brian Demsky. 2014. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness, MSPC'14*, Jeremy Singer, Milind Kulkarni, and Tim Harris (Eds.). ACM, 7:1–7:6. DOI: https://doi.org/10.1145/2618128.2618134

[9] Hans-J. Boehm. 2011. How to miscompile programs with "benign" data races. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism (HotPar'11)*. USENIX Association, USA, 3.

[10] Soham Chakraborty and Viktor Vafeiadis. 2016. Validating optimizations of concurrent C/C++ programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO'16)*. ACM, New York, NY, USA, 216–226. DOI: https://doi.org/10.1145/2854038.2854051

[11] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.* 3, POPL, Article 70 (Jan 2019), 28 pages. DOI: https://doi.org/10.1145/3290383

[12] Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell. 2007. Context switch overheads for Linux on ARM platforms. In *Proceedings of the 2007 Workshop on Experimental Computer Science (ExpCS'07)*. ACM, New York, NY, USA, 3–es. DOI: https://doi.org/10.1145/1281700.1281703

[13] Mike Dodds, Mark Batty, and Alexey Gotsman. 2018. Compositional verification of compiler optimisations on relaxed memory. In *Proceedings of the 27th European Symposium on Programming Languages and Systems, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018*, Amal Ahmed (Ed.). Lecture Notes in Computer Science, Vol. 10801, Springer, 1027–1055. DOI: https://doi.org/10.1007/978-3-319-89884-1_36

[14] Akshay Gopalakrishnan and Clark Verbrugge. 2022. Reordering under the ECMAScript memory consistency model. In *Languages and Compilers for Parallel Computing*, Barbara Chapman and José Moreira (Eds.). Springer International Publishing, Cham, 198–214.

[15] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*. ACM, New York, NY, USA, 175–189. DOI: https://doi.org/10.1145/3009837.3009850

[16] Ori Lahav and Viktor Vafeiadis. 2016. Explaining relaxed memory models with program transformations. In *FM 2016: Proceedings of the 21st International Symposium on Formal Methods*, John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Lecture Notes in Computer Science, Vol. 9995, 479–495. DOI: https://doi.org/10.1007/978-3-319-48989-6_29

[17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 618–632. DOI: https://doi.org/10.1145/3062341.3062352

[18] Leslie Lamport. 1997. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Computers* 46, 7 (1997), 779–782. DOI: https://doi.org/10.1109/12.599898

[19] Sung-Hwan Lee. 2023. A Miscompilation Bug in Loop Invariant Code Motion Pass (LLVM). Retrieved April 2024 from https://github.com/llvm/llvm-project/issues/64188

[20] Jeremy Manson, William W. Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 378–391. DOI: https://doi.org/10.1145/1040305.1040336

[21] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A case for an SC-preserving compiler. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY, USA, 199–210. DOI: https://doi.org/10.1145/1993498.1993522

[22] Evgenii Moiseenko, Anton Podkopaev, and Dmitrij V. Koznov. 2021. A survey of programming language memory models. *Program. Comput. Softw.* 47, 6 (2021), 439–456. DOI: https://doi.org/10.1134/S0361768821060050

[23] Steven Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[24] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Lecture Notes in Computer Science, Vol. 5674, Springer, 391–407. DOI: https://doi.org/10.1007/978-3-642-03359-9_27

[25] Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular relaxed dependencies in weak memory concurrency. In *Proceedings of the 29th European Symposium on Programming Languages and Systems, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020*, Peter Müller (Ed.). Lecture Notes in Computer Science, Vol. 12075, Springer, 599–625. DOI: https://doi.org/10.1007/978-3-030-44914-8_22

[26] Jean Pichon-Pharabod and Peter Sewell. 2016. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 622–633. DOI: https://doi.org/10.1145/2837614.2837616

[27] Jean Yves Alexis Pichon-Pharabod. 2018. A no-thin-air memory model for programming languages. phdthesis. Apollo - University of Cambridge Repository. DOI: https://doi.org/10.17863/CAM.21597

[28] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.* 3, POPL, Article 69 (Jan 2019), 31 pages. DOI: https://doi.org/10.1145/3290382

[29] William W. Pugh. 1999. Fixing the Java memory model. In *Proceedings of the ACM 1999 Conference on Java Grande, JAVA'99*, Geoffrey C. Fox, Klaus E. Schauser, and Marc Snir (Eds.). ACM, 89–98. DOI: https://doi.org/10.1145/304065.304106

[30] William W. Pugh. 2000. The Java memory model is fatally flawed. *Concurr. Pract. Exp.* 12, 6 (2000), 445–455.

[31] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY, USA, 175–186. DOI: https://doi.org/10.1145/1993498.1993520

[32] Jaroslav Ševcík and David Aspinall. 2008. On validity of program transformations in the Java memory model. In *ECOOP 2008 - Proceedings of the 22nd European Conference on Object-Oriented Programming*, Jan Vitek (Ed.). Lecture Notes in Computer Science, Vol. 5142, Springer, 27–51. DOI: https://doi.org/10.1007/978-3-540-70592-5_3

[33] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 209–220. DOI: https://doi.org/10.1145/2676726.2676995

[34] Jaroslav Ševčík. 2011. Safe optimisations for shared-memory concurrent programs. *SIGPLAN Not.* 46, 6 (Jun 2011), 306–316. DOI: https://doi.org/10.1145/1993316.1993534

[35] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. 2020. Repairing and mechanising the JavaScript relaxed memory model. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. ACM, New York, NY, USA, 346–361. DOI: https://doi.org/10.1145/3385412.3385973

Appendices

A Auxiliary Observations

These will be useful for the proofs in Appendix C, D.

Every memory model we consider guarantees the cycles that violate its consistency rules in an execution are preserved on adding (pending) rf and mo relations. Formally,

Definition A.1. A memory model M is *relation-preserving* if any binary relation rel of execution E required for the constraints of M is preserved in every E' ($\mathrm{rel}(E) \subseteq \mathrm{rel}(E')$) such that

- $p(E) = p(E')$.
- $\mathrm{rf}(E) \subseteq \mathrm{rf}(E')$.
- $\mathrm{mo}(E) \subseteq \mathrm{mo}(E')$.

Crucial sets and piecewise consistency of a model can be used to derive a consistent candidate execution from an inconsistent one.

Lemma A.2. *For a given a candidate execution E and memory model M such that $\mathrm{pwc}(M)$ and $\neg c_M(E)$, if a crucial set cr is non-empty, then there exists a candidate execution E_t such that*

- $c_M(E_t)$.
- $p(E_t) = p(E)$.
- $po(p(E_t)) = po(p(E))$.
- $mo(E_t) = mo(E)$.
- $rf(E_t) \cap rf(E) = rf(E) \setminus cr$.

Proof. Using Definition 4.5, we identify crucial set cr and remove the corresponding set of rf relations to give execution E' such that $c_M(E')$ (albeit not well-formed). From $\mathrm{pwc}(M)$, we can use Definition 4.7 to construct E_t from E'. □

The following holds for all transformation-effects not involving any elimination.

Proposition A.3. *For all $P \mapsto_{tr} P'$ such that $st^- = \phi$ we have*

$$\forall E' \in \langle P' \rangle . \exists ! E \in \langle P \rangle \text{ s.t. } E \sim E'.$$

A.1 Minimal Crucial Sets and Safety

Definition A.4. A non-empty crucial set cr for E and memory model M is *minimal* if $\nexists cr'.cr' \subset cr$. Let $Cr(E, M)$ represent the set of minimal crucial sets of candidate execution E for memory model M.

We show how minimal crucial sets enable us to assert a transformation-effect without write-eliminations as unsafe.

Lemma A.5. *Consider a memory model M such that $pwc(M)$. Further, consider a pre-trace P and a transformation-effect $P \mapsto_{tr} P'$ that involves no elimination ($st^{-} = \phi$), along with two candidate executions $E \in I\langle P\rangle_M, E' \in I\langle P'\rangle_M$ such that $E \sim E'$. Then,*

— $\exists cr' \in Cr(E', M) \wedge Cr(E, M) = \phi \implies \neg psafe(M, tr, P)$.
— $\exists cr' \in Cr(E', M), cr \in Cr(E, M)$ *s.t.* $cr \nsubseteq cr' \implies \neg psafe(M, tr, P)$.

Proof. For both cases, from Lemma A.2, we can construct a consistent candidate execution E'_t from E' using cr'. However, from Definition 4.5, since cr is non-existent (or minimal), and that M is relation-preserving (Definition A.1), the resultant candidate execution $E_t \sim E'_t$ from E will still be inconsistent. Because $st^{-} = \phi$, from Proposition A.3, no other candidate execution from $\langle P\rangle$ is similar to E'_t. Thus, we have a consistent execution E'_t under M in P', for which there does not exist any similar consistent execution in the original pre-trace P. From Definition 3.11, we can infer $\neg psafe(M, tr, P)$. □

Using Lemma A.5, we can also infer transformation-effects unsafe for a subset of read-elimination.

Corollary A.6. *Consider the above setting, but with $st^{-} \cap w(P) = \phi$, with any $cr \in Cr(E, M)$ and $cr_t = cr \cap r(P')$ such that $cr_t \neq \phi$. Then,*

— $\exists cr' \in Cr(E', M) \wedge Cr(E, M) = \phi \implies \neg psafe(M, tr, P)$.
— $\exists cr' \in Cr(E', M)\ .\ cr_t \nsubseteq cr' \implies \neg psafe(M, tr, P)$.

Proof. The first case is same as the first of Lemma A.5. For the second, we first reassign the rf for the crucial reads in $cr \setminus cr_t$. This still keeps both E and E' similar and inconsistent. Without the extra reads, the case now is same as the second case in Lemma A.5. □

B Proofs for Sequential Consistency

Theorem B.1. *No consistency rule of SC is redundant.*

Proof. We prove this by contradiction. Assume some consistency rule of SC is redundant. Then

$$\exists a1, a2 \in SC \text{ s.t. } \forall E\ .\ \neg a1(E) \implies \neg a2(E).$$

We show that this is not true by construction of a candidate execution that clearly does not satisfy the implication for any given pair of rules. We take cases over $a1$, being any of the five consistency rules of SC, resulting in five examples.

a) mo strict total order.
b) mo; hb irreflexive.
c) hb irreflexive.
d) rb; hb irreflexive.
e) rb; mo; hb irreflexive.

The counter examples are shown in Figure 28 as five cases. For each, notice that exactly one consistency rule of SC is violated, but the others not. □

Theorem B.2. *SC is piecewise consistent - $pwc(SC)$.*

Proof. Consider an execution E such that $c_{SC}(E)$ and that it is not well-formed only due to the missing rf relation with r_x. The task now is to find a suitable rf for r_x such that none of the consistency rules of SC are violated, giving us our desired candidate execution E_t. We first choose a random write w_x and modify our choice based on the cycles formed violating a constraint of SC. Our modified choice is either an immediate mo_x previous or later write to w_x. We show that this process should terminate, resulting in the desired write. If the first choice violates no rule of SC, we are done. If it does, then the following cycles are possible.

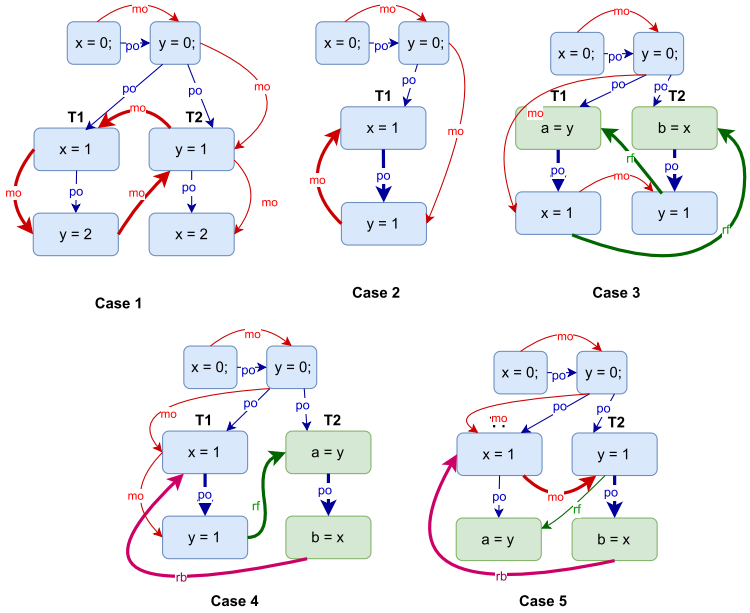

Fig. 28. Counter examples for each case of $a1$ being one of the consistency rules.

Case 1: $\mathrm{hb};[w_x];\mathrm{rf};[r_x]$ cycle or $\mathrm{mo};[w_x];\mathrm{rfe};[r_x];\mathrm{hb}$ cycle or $\mathrm{rb};\mathrm{mo}^?;[w_x];\mathrm{rfe};[r_x];\mathrm{hb}$ cycle.
- Choosing an mo later write w'_x ($[w_x];\mathrm{mo};[w'_x]$), would always get $[w_x];\mathrm{mo};[w'_x];\mathrm{rf};[r_x];\mathrm{hb}$ cycle or $\mathrm{rb};\mathrm{mo}^?;[w'_x];\mathrm{rfe};[r_x];\mathrm{hb}$ cycle.
- Hence, we choose the immediate mo previous write w'_x ($[w'_x];\mathrm{mo};[w_x]$) instead.

Case 2: $[r_x];\mathrm{rb};[w];\mathrm{mo}^?;\mathrm{hb}$ cycle.
- Choosing an mo previous write w'_x ($[w'_x];\mathrm{mo};[w_x]$), would get the same $[r_x];\mathrm{rb};\mathrm{mo}^?;\mathrm{hb}$ cycle.
- Hence, we choose the immediate mo later write w'_x ($[w_x];\mathrm{mo};[w'_x]$) instead.

Since the mo_x has a minimal (initialization) and maximal (candidate execution) element, requiring to choose only mo previous or only mo later writes will eventually get us our desired write. However, as we noted above, different cycles may require choosing either one while trying to identify our desired write. This can result in never finding a write: choosing an mo previous one can result in a cycle that requires to choose an mo later write instead. We show that such a situation would imply E inconsistent under SC without the rf relation instead, thereby violating our premise. There are three cases to consider, pairing the two cycles which would create the scenario mentioned above, while choosing either w_x or w'_x.

Case 1: $[w];\mathrm{mo};[w_x];\mathrm{rfe};[r_x];\mathrm{hb}$ cycle and $[r_x];\mathrm{rf}^{-1};[w'_x];\mathrm{mo}_{loc};[w_x];\mathrm{mo}^?;[w'];\mathrm{hb}$ cycle.
This gives us $[w_x];\mathrm{mo}^?;[w'];\mathrm{hb};\mathrm{po};[r_x];\mathrm{po};\mathrm{hb};[w];\mathrm{mo}$ cycle, or $\mathrm{mo};\mathrm{hb}$ cycle in E.

Case 2: $[w_x];\mathrm{rf};[r_x];\mathrm{hb}$ cycle and $[r_x];\mathrm{rf}^{-1};[w'_x];\mathrm{mo}_{loc};[w_x];\mathrm{mo}^?;[w'];\mathrm{hb}$ cycle.
This gives us $[w_x];\mathrm{mo}^?;[w'];\mathrm{hb};\mathrm{po};[r_x];\mathrm{po};\mathrm{hb};$ cycle, or $\mathrm{mo};\mathrm{hb}$ cycle in E.

Case 3: $\mathrm{rb};\mathrm{mo}^?;[w_x];\mathrm{rfe};[r_x];\mathrm{hb}$ cycle and $[r_x];\mathrm{rf}^{-1};[w'_x];\mathrm{mo}_{loc};[w_x];\mathrm{mo}^?;[w'];\mathrm{hb}$ cycle.
This gives us $\mathrm{rb};\mathrm{mo}^?;[w_x];\mathrm{mo}^?;[w'];\mathrm{hb};\mathrm{po};[r_x];\mathrm{po};\mathrm{hb}$ cycle, or $\mathrm{rb};\mathrm{mo}^?;\mathrm{hb}$ cycle in E.

Thus, we will always be able to find a suitable write w_x for $[w_x];\mathrm{rf};[r_x]$, resulting in E_t being consistent under SC. Hence, $pwc(SC)$ is true. □

Lemma B.3. *Given an inconsistent execution E under SC, a crucial set exists if mo is a strict total order and does not conflict po.*

Proof. Each of the consistency rules of *SC* without the constraints mentioned in the premise has a non-empty *cra* by Lemma 5.1. $cra(\mathrm{mo};\mathrm{hb})$ is non-empty as mo does not conflict with po. We can thus construct a crucial set *cr* by taking at least 1 read from each *cra* set and considering it's rf in E. □

C Proofs Regarding SC_{RR} without Fences and Read-modify-Writes

Theorem 5.3. *For transformation-effect rr in Definition 5.2, we have sound(SC_{RR}, rr).*

Proof. Continuing from the proof sketch via contradiction, note that $[r_x];\mathrm{po}_{imm};[r_y]$ must be a part of the cycle in E violating some rule of SC_{RR}. Our objective is to show that for each cycle, if E is inconsistent, then so is E' which is comparable to E.

Rule (a) violated: mo not strict.

The po between reads does not play a role in mo. Thus, mo remains non-strict even in E', thereby violating Rule (a).

Rule (b) violated: hb cycle.

$$\begin{aligned}
&\mathrm{hb};[r_x];\mathrm{po}_{imm};[r_y];\mathrm{hb}\text{ cycle.}\\
&\rightarrow \mathrm{hb};[r_x];\mathrm{po}_{imm};[r_y];\mathrm{hb}.\\
&\rightarrow \mathrm{hb};\mathrm{po};[r_x];\mathrm{po}_{imm};[r_y];\mathrm{hb} \vee \mathrm{hb};\mathrm{rfe};[r_x];\mathrm{po}_{imm};[r_y];\mathrm{po}.\\
&\rightarrow \mathrm{hb};\mathrm{po};[r_y];\mathrm{hb} \vee \mathrm{hb};\mathrm{rfe};[r_x];\mathrm{po}\text{ cycle.}
\end{aligned}$$

We can see that hb cycle exists independent of the po between r_x and r_y. Thus, hb forms a cycle in E', thereby violating Rule (b).

Rule (c) violated: mo;hb cycle.

$$\begin{aligned}
&\mathrm{mo};\mathrm{hb};[r_x];\mathrm{po}_{imm};[r_y]\text{ cycle.}\\
&\rightarrow \mathrm{mo};\mathrm{hb};[r_x];\mathrm{po}_{imm};[r_y];\mathrm{po}.\\
&\rightarrow \mathrm{mo};\mathrm{hb}^?;\mathrm{po};[r_x];\mathrm{po}_{imm};[r_y];\mathrm{hb} \vee \mathrm{mo};\mathrm{hb}^?;\mathrm{rfe};[r_x];\mathrm{po}_{imm};[r_y];\mathrm{po};\mathrm{hb}^?.\\
&\rightarrow \mathrm{mo};\mathrm{hb}^?;\mathrm{po};[r_y];\mathrm{hb} \vee \mathrm{mo};\mathrm{hb}^?;\mathrm{rfe};[r_x];\mathrm{po};\mathrm{hb}^?\text{ cycle.}
\end{aligned}$$

We can see that mo;hb cycle exists independent of the po between r_x and r_y. Thus, mo;hb forms a cycle in E', thereby violating Rule (c).

Rule (d) or (e) violated: $\mathrm{rb};\mathrm{mo}^?;\mathrm{hb} \setminus a_{rr}$ cycle. (We remove mo for the sake of simplicity.)

$$\begin{aligned}
&\mathrm{rb};\mathrm{hb};[r_x];\mathrm{po}_{imm};[r_y];\mathrm{hb}^?\text{ cycle.}\\
&\rightarrow \mathrm{rb};\mathrm{hb};\mathrm{po};[r_x];\mathrm{po}_{imm};[r_y];\mathrm{hb}^? \vee \mathrm{rb};\mathrm{hb};\mathrm{rfe};[r_x];\mathrm{po}_{imm};[r_y];\mathrm{hb}^?.\\
&\rightarrow \mathrm{rb};\mathrm{hb};[r_y];\mathrm{hb}^? \vee \mathrm{rb};\mathrm{hb};\mathrm{rfe};[r_x];\mathrm{po}_{imm};[r_y];\mathrm{po};\mathrm{hb}_? \vee a_{rr}.\\
&\rightarrow \mathrm{rb};\mathrm{hb};[r_y];\mathrm{hb}^? \vee \mathrm{rb};\mathrm{hb};\mathrm{rfe};[r_x];\mathrm{po};\mathrm{hb}^? \vee a_{rr}\text{ cycle.}
\end{aligned}$$

We can see that either rb;hb cycle exists independent of the po between r_x and r_y or the composition itself becomes part of a_{rr}, which is not a constraint of SC_{RR}. Thus, Rule (d) or (e) is violated in E'.

Hence, via contradiction we have $sound(SC_{RR}, rr)$. □

Lemma 5.5. *Consider transformation-effects $P \mapsto_{tr} P'$ defined by Theorem 4.4 not involving write-elimination ($st^- \cap w(P) = \phi$). Consider also, from Theorem 4.4, the corresponding execution $E' \in \langle P' \rangle$*

which is consistent under SC_{RR} but inconsistent under SC. Then, at least one of the following is true for any $E \in \langle P \rangle$ where $E \sim E'$:

a) $rfi; po$ *cycle.*
b) $mo; po$ *cycle.*
c) $mo; rfe; po$ *cycle.*
d) $rb; mo^?; po$ *cycle.*
e) $rb; rfe; po$ *cycle.*

Proof. From Theorem 4.4 we know $E \in I\langle P \rangle_{SC_{RR}}$ and $E' \in [\![P']\!]_{SC_{RR}}$. We also know from $st^- \cap w(P) = \phi$, mo must be a strict total order for both E and E'. We go case-wise on the remaining possible rules that could be violated for E. For each, we show that they reduce to the above claimed compositions which form a cycle. Since pre-traces are finite set of events, by monotonicity, this reduction process eventually terminates. To make the reduction process readable, we assume rr to be the composition forming a cycle, thus violating rule a_{rr}. We also denote the following symbols for the other relations that we need to show forming a cycle in E:

- **A** - rfi; po cycle.
- **B** - mo; po cycle.
- **C** - mo; rfe; po cycle.
- **D** - rb; $mo^?$; po; $[r_x]$ cycle.
- **E** - rb; rfe; $[r_x]$; po; $[r_x]$ cycle.

Rule (b) violated: hb cycle.

$\rightarrow$ hb.
$\rightarrow$ rf; po; $hb^?$.
$\rightarrow$ rfi; po $\vee$ rfe; po; $hb^?$.
$\rightarrow$ **A** $\vee$ $[w]$; rfe; po; $[w']$; rfe; po; $hb^?$.
$\rightarrow$ **A** $\vee$ $[w']$; mo; $[w]$; rfe; $[r_x]$; po $\vee$ $[w]$; mo; $[w']$; rfe; po; $hb^?$.
$\rightarrow$ **A** $\vee$ **C** $\vee$ mo; hb cycle.

Rule (c) violated: mo; hb cycle.

$\rightarrow$ mo; hb.
$\rightarrow$ mo; $hb^?$; po.
$\rightarrow$ mo; po $\vee$ mo; $[w]$; $hb^?$; $[w']$; rf; po.
$\rightarrow$ **B** $\vee$ mo; $[w']$; rf; po $\vee$ $[w]$; hb; $[w']$; mo.
$\rightarrow$ **B** $\vee$ mo; $[w']$; rfi; po $\vee$ mo; $[w']$; rfe; po $\vee$ mo; hb.
$\rightarrow$ **B** $\vee$ rfi; po $\vee$ **C** $\vee$ mo; hb.
$\rightarrow$ **B** $\vee$ **A** $\vee$ **C** $\vee$ mo; hb cycle.

Rule (d) violated: rb; hb cycle.

$\rightarrow$ rb; hb.
$\rightarrow$ rb; $hb^?$; po; $[r_x]$.
$\rightarrow$ rb; po; $[r_x]$ $\vee$ rb; $[w]$; $hb^?$; $[w']$; rf; po; $[r_x]$.
$\rightarrow$ **D** $\vee$ rb; $[w]$; mo; $[w']$; rf; po; $[r_x]$ $\vee$ mo; $[w]$; hb; $[w']$ $\vee$ rb; $[w]$; rf; po; $[r_x]$.
$\rightarrow$ **D** $\vee$ rb; mo; rfe; po; $[r_x]$ $\vee$ rb; mo; rfi; po; $[r_x]$ $\vee$ mo; hb $\vee$ rb; $[w]$; rf; po; $[r_x]$.
$\rightarrow$ **D** $\vee$ **E** $\vee$ rr $\vee$ rfi; po $\vee$ **D** $\vee$ mo; hb $\vee$ rb; po $\vee$ rfi; po $\vee$ rb; rfe; po; $[r_x]$.
$\rightarrow$ **D** $\vee$ **E** $\vee$ rr $\vee$ **A** $\vee$ mo; hb $\vee$ **D** $\vee$ **A** $\vee$ **E**.

$$\rightarrow \mathbf{D} \vee \mathbf{E} \vee \mathbf{A} \vee rr \vee \text{mo};\text{hb cycle}.$$

Rule (e) violated: rb; mo; hb cycle.

$\rightarrow$ rb; mo; hb.

$\rightarrow$ rb; mo; hb$^?$; po; $[r_x]$.

$\rightarrow$ rb; mo; $[w']$; hb$^?$; $[w]$; rf; po; $[r_x]$ $\vee$ rb; mo; po; $[r_x]$.

$\rightarrow$ rb; mo; $[w']$; mo$^?$; $[w]$; rf; po; $[r_x]$ $\vee$ mo; $[w']$; hb; $[w]$ $\vee$ **D**.

$\rightarrow$ rb; mo; $[w]$; rfe; po; $[r_x]$ $\vee$ rb; mo; $[w]$; rfi; po; $[r_x]$ $\vee$ mo; hb $\vee$ **D**.

$\rightarrow$ rb; mo; rfe; po; $[r_x]$ $\vee$ rb; mo; po; $[r_x]$ $\vee$ rfi; po $\vee$ mo; hb $\vee$ **D**.

$\rightarrow$ rb; mo; rfe; $[r_x]$; po; $[r_x]$ $\vee$ rb; mo; rfe; $[r_y]$; po; $[r_x]$ $\vee$ **D** $\vee$ **A** $\vee$ mo; hb $\vee$ **D**.

$\rightarrow$ rb; rfe; $[r_x]$; po; $[r_x]$ $\vee$ rr $\vee$ **A** $\vee$ mo; hb $\vee$ **D**.

$\rightarrow$ **E** $\vee$ rr $\vee$ **A** $\vee$ mo; hb $\vee$ **D** cycle.

The above cases show that either rr or one of the relations **A, B, C, D, E** that form a cycle (by monotonicity) must hold in E. If only the cycle rr exists in E, then by Theorem 4.4, it is not a valid transformation-effect (as E must be inconsistent under SC_{RR}). Hence proved. □

Theorem 5.6. *For all transformation-effects not involving write elimination ($st^- \cap w(P) = \phi$), comp$\langle SC_{RR}, SC\rangle$.*

Proof. From Lemma 5.5, we know at least one of the following must be true in E:

a) rfi; po cycle.
b) mo; po cycle.
c) mo; rfe; po cycle.
d) rb; mo$^?$; po cycle.
e) rb; rfe; po cycle.

Consider the above compositions as rsc. We also know that E' has only rr cycles as opposed to the above ones for E. From Lemma B.3, we know that a crucial set exists for E'. From Proposition 5.4, we know $cra(rr) \nsubseteq cra(rsc)$. Thus, we can construct a minimal crucial set cr' for E', such that

$$\nexists cr \in Cr(E, SC)\ .\ cr \subseteq cr'.$$

Since $st^- \cap w(P) = \phi$, the mo cannot change among similar executions. Therefore, there can be only more than one E if the eliminated reads are assigned some other rf relation. Depending on the transformation-effect and crucial set cr, we have three cases for this

Case 0: $\nexists cr \in Cr(E, SC)$.
Since E has no crucial set, we trivially have $\neg psafe(SC, tr, P)$.

Case 1: $\exists cr\ .\ st(r(E')) \cap cr = \phi$.
This contradicts our premise over E' and E in Theorem 4.4, as we now can have an execution E consistent under SC_{RR} and $E' \sim E$. Therefore, this case need not be considered.

Case 2: $\exists cr\ .\ st^- \cap cr \neq \phi$.
We simply remove the reads eliminated from cr, giving us cr_t. But note that we would still have $cr_t \nsubseteq cr'$ from the above, otherwise it is simply Case 1. From Corollary A.6, we have $\neg psafe(SC, tr, P)$.

Case 3: $\exists cr\ .\ st^- \cap cr = \phi$.
By Lemma A.5, we have $\neg psafe(SC, tr, P)$.

Thus, by contradiction we have comp$\langle SC_{RR}, SC\rangle$. □

D Proofs Regarding SC_{RR} on Adding Fences and Read-Modify-Writes

Lemma 5.9. *Consider transformation-effects $P \mapsto_{tr} P'$ defined by Theorem 4.4 not involving write-elimination ($st^- \cap w(P) = \phi$) and the corresponding execution $E' \in \langle P' \rangle$ consistent under SC_{RR} but inconsistent under SC. Then, on addition of f_{rr} and rmw events and their consistency rules, the following may also be true for any $E \in \langle P \rangle$ where $E \sim E'$:*

a) $mo; rf$ *cycle.* *b)* $rb; mo$ *cycle.*

c) $rb; mo; rfe; po; [f_{rr}]; po$ *cycle.*

Proof. We go case-wise on relations that do bring different possible relations forming a cycle on addition of *rmw* or *fences*.

Rule (c) violated: mo; hb cycle.

$$\rightarrow \text{mo}; \text{hb}.$$
$$\rightarrow \text{mo}; \text{rf} \cup \text{mo}; \text{po}; \text{hb}^?.$$
$$\rightarrow \text{mo}; \text{rf} \cup \text{mo}; \text{hb} \text{ cycle}.$$

Rule (f) violated: rb; mo cycle.
As is.
Rule (g) violated: rb; mo; rfe; $[r_y]$; po; $[f_{rr}]$; po; $[r_{x \neq y}]$ cycle.
As is.

The above cases show that either mo; hb cycle or one of the compositions above (by monotonicity) must hold. Thus, at least one of the above mentioned relations may also form a cycle in E. □

Lemma 5.10. *For transformation-effects $P \mapsto_{tr} P'$ as defined in Theorem 4.4 with no write elimination, and with corresponding pair $E \sim E'$ and $B = SC$, $M = SC_{RR}$, we have the following:*

$$a_{frr}(E) = false \implies ((f_{rr} \in st^-) \vee ([f_{rr}]; po; [r_x] \in po^-) \vee ([r_y]; po; [f_{rr}] \in po^-)),$$

where $r_x, r_y \notin st^-$.

Proof. From Theorem 4.4 we know $E \in I\langle P \rangle_{SC_{RR}}$ and $E' \in [\![P']\!]_{SC_{RR}}$. We also know that mo must be a strict total order for both E and E'. Consider the relation resulting in $a_{frr}(E) = false$.

$$\text{rb}; \text{mo}^?; \text{hb}^?; \text{rfe}; [r_y]; \text{po}; [f_{rr}]; \text{po}; [r_{x \neq y}] \text{ cycle}.$$

We know that the rf and mo relations cannot change. Thus, either po or some *st* needs to change. We also know that st^- cannot involve the reads r_x, r_y nor any write in $w(P)$. Therefore, the only choices remain is to either remove f_{rr} or the two po relations $[r_y]; po; [f_{rr}]$ or $[f_{rr}]; po; [r_x]$. Hence proved. □

Theorem 5.11. *Given SC and SC_{RR} with rmw and f_{rr} events, for all transformation-effects not involving write elimination, we have comp$\langle SC_{RR}, SC \rangle$.*

Proof. From Lemma 5.9, at least one of the following must be true in E:

a) mo; rf cycle. b) rb; mo cycle.

c) rb; mo; rfe; po; $[f_{rr}]$; po cycle.

First, if the cycle involves the fence f_{rr} like (c), then by Lemma 5.10, the transformation-effect also involves f_{rr}. Since f_{rr} acts as a no-op in *SC*, we do not need to consider such effects for proving Complete. We are only concerned with effects on programs with no fences involved.

Now, consider the first two cycles (a), (b) as *usc*. We know that E' has only *rr* cycles as opposed to the above for E. From Lemma B.3, we know that a crucial set exists for E'. From Proposition 5.8, we know $cra(rr) \not\subseteq cra(usc)$. Thus, there must be a minimal crucial set cr' such that

$$\nexists cr \in Cr(E, SC) \;.\; cr \subseteq cr'.$$

Since $st^{-} \cap w(P) = \phi$, the mo cannot change among similar executions. Therefore, there can be only more than one E if the eliminated reads are assigned some other rf relation. Depending on the transformation-effect and crucial set *cr*, we have three cases for this

Case 0: $\nexists cr \in Cr(E, SC)$.
Since E has no crucial set, we trivially have $\neg psafe(SC, tr, P)$.

Case 1: $\exists cr \;.\; st(r(E')) \cap cr = \phi$.
This contradicts our premise over E' and E in Theorem 4.4, as we know can have an execution E consistent under SC_{RR} and $E' \sim E$. Therefore, this case need not be considered.

Case 2: $\exists cr \;.\; st^{-} \cap cr \neq \phi$.
We simply remove the reads eliminated from *cr*, giving us cr_t. But note that we would still have $cr_t \not\subseteq cr'$ from the above, otherwise it is simply Case 1. From Corollary A.6, we have $\neg psafe(SC, tr, P)$.

Case 3: $\exists cr \;.\; st^{-} \cap cr = \phi$.
By Lemma A.5, we have $\neg psafe(SC, tr, P)$.

Thus, by contradiction, for the models *SC* and SC_{RR} with the inclusion of f_{rr} and *rmw* events, we have $\text{comp}\langle SC_{RR}, SC\rangle$. □

Corollary 6.1. *For transformation-effects quantified by Theorem 4.4 with pair $E \sim E'$ and $B = SC$, $M = SC_{RR}$, we have the following for E if no write elimination-effect exists:*

$$(rfi; [r_x]; po \cup mo; rfe; [r_x]; po) \;\; cycle \implies [r]; po; [w] \in po^{-}.$$

where $r_x \in (r(P) \cap r(P'))$.

Proof. By Lemma 5.5 we know at least one of the mentioned relations forms a cycle. We also know that none of these cycles compositions exist in E'. Therefore, we only need to check what we can remove in order to eliminate the cycle. Let us go case-wise:

Case 1: rfi; $[r_x]$; po cycle.
Since rfi; $[r_x]$ cannot change, only the po relation can change to eliminate the cycle. Since we have $[r_x]$; po; $[w_x]$, we need $[r_x]$; po; $[w_x] \in \text{po}^{-}$.

Case 2: $[w_y]$; mo; rfe; $[r_x]$; po cycle.
Since both mo and rfe cannot change from the premise, we have to change po to eliminate the cycle. Since we have $[r_x]$; po; $[w_y]$, we need $[r_x]$; po; $[w_x] \in \text{po}^{-}$.

Thus, read–write de-ordering is part of the transformation-effect. □

Received 3 June 2024; revised 13 February 2025; accepted 25 February 2025