# Verifying Local Transformations on Relaxed Memory Models

Sebastian Burckhardt[1], Madanlal Musuvathi[1], and Vasu Singh[2]

[1] Microsoft Research
[2] EPFL, Switzerland

**Abstract.** The problem of locally transforming or translating programs without altering their semantics is central to the construction of correct compilers. For concurrent shared-memory programs this task is challenging because (1) concurrent threads can observe transformations that would be undetectable in a sequential program, and (2) contemporary multiprocessors commonly use relaxed memory models that complicate the reasoning.

In this paper, we present a novel proof methodology for verifying that a local program transformation is sound with respect to a specific hardware memory model, in the sense that it is not observable in any context. The methodology is based on a structural induction and relies on a novel compositional denotational semantics for relaxed memory models that formalizes (1) the behaviors of program fragments as a set of traces, and (2) the effect of memory model relaxations as local trace rewrite operations.

To apply this methodology in practice, we implemented a semi-automated tool called Traver and used it to verify/falsify several compiler transformations for a number of different hardware memory models.

## 1 Introduction

Compilers perform a series of transformations that translate a high-level program into low-level machine instructions, while optimizing the code for performance. For correctness, these transformations must preserve the meaning for any input program. Proving the correctness of program transformations has been well studied for sequential programs [29,18,17,19].

However, concurrent shared-memory programs require additional caution because transformations that reorder, introduce, or eliminate accesses to shared memory may be observed by concurrent threads and can thus introduce subtle safety or liveness errors in an otherwise correct program. For example, the redundant read elimination shown in Fig. 1 is not safe because it leads to nontermination, and the branch consolidation in Fig. 2 is unsafe because it can lead to an assertion violation.

Typically, only a very small part of all memory accesses (namely the accesses that are used for synchronization purposes) are susceptible to such issues. However, in the absence of a whole-program-analysis or user-provided annotations,

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 104–123, 2010.

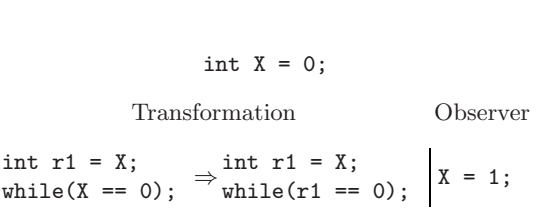


**Fig. 1.** Redundant read elimination causing nontermination

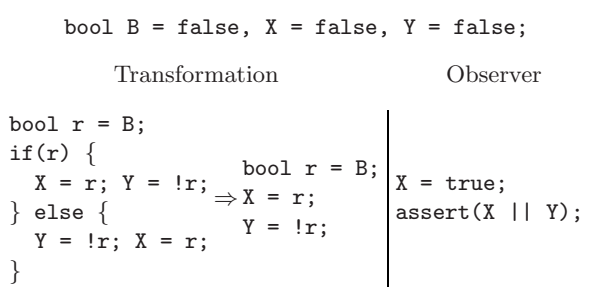


**Fig. 2.** This branch consolidation is unsafe: the assert can fail in the transformed program, but not the original program. The reason is that the transformation changes the order of the writes to `X` and `Y` in the then-branch.

we can not distinguish between data accesses and accesses that are used for synchronization. In practice, most compilers rely on the programmer to provide special type qualifiers like 'volatile' [20] or 'atomic' [4] or on custom annotations to identify synchronization accesses. Programs that correctly convey all synchronization are called 'properly labeled' [14] or 'data-race-free' [2].

There is a general understanding on how to correctly transform data-race free programs [20,4,25]. In this paper, however, we address the more conservative problem of safely transforming general programs, including programs that contain data races, or programs that are missing the annotations or types needed to identify synchronization accesses.

It may seem at first that under this conservative restriction, very few transformations would be safe. However, we can assume that programs that are designed to work on relaxed hardware memory models are resilient to certain transformations. Clearly, there is no need for a compiler to be more conservative than the hardware executing the compiled program.

For example, consider the example in Fig. 2 again, but let the execution be on a machine that relaxes write-to-write order. Now, we may argue that the transformation is indeed correct as it does not introduce new behaviors: if write-to-write order is relaxed by the hardware, the assertion violation may occur even for the original untransformed program.

For some transformations it can be rather mind-boggling to determine whether it is safe for a given architecture. For instance, by using the methodology

presented in this paper, we will prove (though not fully comprehend) that the transformation

$$\{r := A; \textbf{if } r == 0 \textbf{ then } A := 0\} \rightarrow \{r := A\}$$

is safe on a sequentially consistent machine, unsafe on a machine that relaxes write-to-read order (such as TSO), but once more safe on a machine that additionally relaxes write-to-write order (such as PSO).

Overall, we summarize our contributions as follows:

- (Section 3) We build a semantic foundation for relaxed hardware memory models. We show how many common relaxations can be explained as local rewrite operations on memory access sequences. In particular, we present a novel aggregation rule that can explain the effect of store buffers, the most common relaxation of all. Our semantics is compositional (it defines the behavior of program fragments recursively) and can model infinite executions.
- (Section 4) We present a proof methodology to verify the soundness of local program transformations over relaxed memory models, based on a notion of observations. We introduce a notion of invisible rewrite rules (Section 4.1) to reason about all possible program contexts.
- (Section 5) We show how to apply the methodology in practice by verifying/falsifying 8 program transformation for 5 different memory models (including sequential consistency), aided by a custom semi-automatic tool called Traver. Given a local program transformation and a memory model, Traver uses an automated theorem prover [11] to prove that the set of observations of the transformed program is contained in the set of observations of the original program, for all possible program contexts. Conversely, when provided with an additional falsification context, Traver can automatically show that the transformation leads to observable differences in behavior. This produces a certificate of unsoundness of the transformation.

## 2 Related Work

Our calculus and semantics, and in particular the handling of infinite executions, were inspired by Brookes' fully abstract denotational semantics for sequentially consistent programs [6]. Languages and semantics to study relaxed memory models have been developed before, in both operational style [5] and algebraic style [24]. Our work differs in that it (1) guarantees fairness for infinite executions and (2) relates to contemporary multiprocessor architectures and common program transformations.

Much prior work on hardware memory models focuses on the complex intricacies of axiomatic specifications and gives only partial formalizations (in particular, program syntax is generally ignored). Some work departs from the mainstream and uses an operational style [23] or an algebraic style [3,27] (where the algebraic style bears some similarity to our use of dynamic rewrite rules, but

does not include the important store-load aggregation rule which is crucial to correctly model contemporary hardware memory models). Recently, researchers have proposed revised axiomatic formalizations of the x86 architecture [13,22]. Our work is orthogonal: our goal is to find simple yet precise means to reason about various common hardware relaxations, rather than fully model all details of one specific hardware architecture.

Our work was partly motivated by recent work [9,26] that demonstrated the difficulty of manually verifying compiler optimizations against memory models. It is also similar to efforts on verifying the soundness of compiler transformations for language-level models (Java, DRF) [25]. Unlike the latter, however, we define soundness of transformations relative to the hardware memory model (and are thus not susceptible to whether programs are data-race-free or not), can handle infinite executions, and provide a tool that helps to automate parts of the verification/falsification effort.

## 3 Semantic Foundation

In this section, we lay the foundation for understanding hardware memory models and for reasoning about them formally. We start by demonstrating how we explain typical relaxations in the hardware using dynamic rewrite operations. We then formalize this concept by defining a simple imperative language for shared-memory programs and a compositional denotational semantics. Along the way, we discuss various challenges, such as how our semantics handles infinite executions and fairness.

We start with a quick introduction to relaxed hardware memory models, revisiting classical examples [1,14]. We use special diagrams called *derivations* to explain how to understand relaxations as a consequence of dynamic rewriting of access sequences. We distinguish three types of dynamic rewrite operations: reordering, aggregation, and splitting.

Ordering relaxations allow the hardware to execute operations in a different order than specified by the program. This can speed up execution as it allows the hardware to delay the completion of operations with high latency (such as propagating stores to a global shared memory) past subsequent operations with low latency (such as reading a locally cached value). In Fig. 3 (a) and (b), we show classic "litmus tests" to illustrate the effects of ordering relaxations. These programs distinguish syntactically between processor-local registers (lowercase identifiers) and shared memory locations (capitalized identifiers).

Not all effects can be explained by simply reordering instructions. For example, the program in Fig. 3(c) is a variation of 3(b) that shows how stored values can be visible to subsequent loads by the same processor before they have been committed to shared memory. This effect is very common and often attributed to processor-local "store buffers". We explain this effect as an aggregation of the store with the following load.

More formally, let $\langle ld\ \ L, x\rangle$ and $\langle st\ \ L, x\rangle$ represent store or load accesses from/to location $L$, with loaded/stored value of $x$. Now consider the dynamic

**(a)**

Initially: $A = B = 0$

| $P1$ | $P2$ |
|---|---|
| $A := 1$ | $r := B$ |
| $B := 1$ | $s := A$ |

Eventually: $r = 1, s = 0$

**(b)**

Initially: $A = B = 0$

| $P1$ | $P2$ |
|---|---|
| $A := 1$ | $B := 1$ |
| $r := B$ | $s := A$ |

Eventually: $r = s = 0$

**(c)**

Initially: $A = B = 0$

| $P1$ | $P2$ |
|---|---|
| $A := 1$ | $B := 1$ |
| $u := A$ | $v := B$ |
| $r := B$ | $s := A$ |

Eventually: $r = s = 0, u = v = 1$

**Fig. 3.** (a) This outcome is possible if the stores by P1 are reordered, or if the loads by P2 are reordered. (b) This outcome (known as Dekker) is possible if the stores are delayed past the loads. (c) This outcome (a variation of Dekker) is possible if stores can be both forwarded to loads and delayed past loads.

**sss** (swap store-store)
$\langle st\ L, x\rangle\langle st\ L', x'\rangle \overset{L\neq L'}{\rightarrow} \langle st\ L', x'\rangle\langle st\ L, x\rangle$
**sll** (swap load-load)
$\langle ld\ L, x\rangle\langle ld\ L', x'\rangle \rightarrow \langle ld\ L', x'\rangle\langle ld\ L, x\rangle$
**ssl** (swap store-load)
$\langle st\ L, x\rangle\langle ld\ L', x'\rangle \overset{L\neq L'}{\rightarrow} \langle ld\ L', x'\rangle\langle st\ L, x\rangle$
**sls** (swap load-store)
$\langle ld\ L, x\rangle\langle st\ L', x'\rangle \overset{L\neq L'}{\rightarrow} \langle st\ L', x'\rangle\langle ld\ L, x\rangle$
**asl** (aggregate store-load)
$\langle st\ L, x\rangle\langle ld\ L, x\rangle \rightarrow \langle st\ L, x\rangle$

| Model | Rewrite Rules | | | | |
|---|---|---|---|---|---|
| SC | (none) | | | | |
| 390 | **ssl** | | | | |
| TSO | **ssl** | **asl** | | | |
| x86-TSO | **ssl** | **asl** | | | |
| PSO | **ssl** | **asl** | **sss** | | |
| CLR | **ssl** | **asl** | **sll** | | |
| RMO | **ssl** | **asl** | **sss** | **sll**$^{\not\sim\not\sim}$ | **sls**$^{\not\sim\not\sim}$ |
| Alpha | **ssl** | **asl** | **sss** | **sll**$^{\neq}$ | **sls**$^{\not\sim\not\sim}$ |

**Fig. 4.** Dynamic rewrite operations employed by some commercial hardware memory models and by the CLR memory model. The symbols $\not\sim$, $\not\sim\not\sim$ and $\neq$ indicate that the accesses are swapped only if they are not control dependent, not data dependent, or target a different location, respectively.

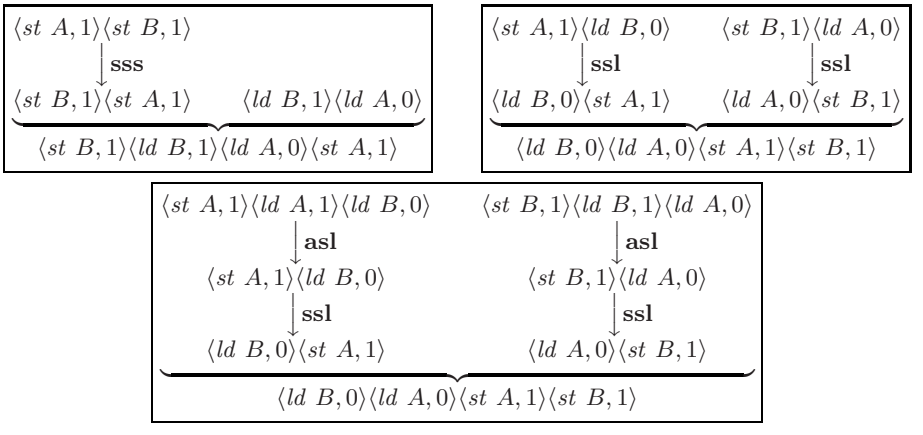


**Fig. 5. Top left**: Derivation for Fig. 3(a). P1 issues two stores that get reordered by **sss** before being interleaved with the two loads by P2. Note that we could provide an alternative derivation where the loads get reordered by **sll**. **Top right**: Derivation for Fig. 3(b). Both store-load sequences are reordered by **ssl** before being interleaved. **Bottom**: Derivation for Fig. 3(c). Both processors first aggregate the stores with the first following load by **asl**, then delay it past the second load by **ssl**.

rewrite operations in Fig. 4. All of these operations preserve the semantics of single-processor programs, as long as the conditions are observed (**asl** applies only to accesses that target the same location and store/load the same value, while **sss**, **ssl**, and **sls** apply only to accesses that target different locations).

To see how these dynamic rewrite operations can explain the examples in Fig. 3, consider the derivation diagrams in Fig. 5. Each processor first produces a sequence of memory accesses consistent with the program. These sequences are *dynamic*, as they contain data that may not be known statically (such as actual addresses and values loaded or stored), and may repeat program fragments that execute in loops. The access sequences may then be locally modified by the dynamic rewrite operations. Next, the sequences of the processors are interleaved. Informally, an interleaving shuffles the various sequences while maintaining the access order within each sequence (we give a formal definition in Section 3.2). Our derivation diagrams show which sequences are being interleaved with an underbrace. At the end of the derivation (but not necessarily before), the sequence must be *value-consistent*; that is, loaded values must be equal to the latest value stored to the same location, or the initial value if there is no preceding store.

In general, it is quite difficult to establish a precise relationship between abstract memory models (described as a collection of relaxations, in the style of [1]) and official memory model specifications of commercially available multiprocessors. However, it is possible and sensible for research purposes to model just the abstract core of such models, by focusing on the behavior of regular loads and stores. Fig. 4 shows how can model the core of many commercial hardware memory models, and even the CLR memory model, using the dynamic rewrite rules defined in Fig. 3. Our main sources for constructing this table were [16] for 390, [28] for TSO, PSO and RMO, [10] for Alpha, [22] for x86-TSO, and [7,12,21] for CLR.

Beyond simple loads and stores, all of these architectures contain additional constructs (such as locked instructions, compare-and-swaps, various memory fences, or volatile memory accesses). Many of them can be formalized using custom syntax and rewrite rules. However, for simplicity, we stick to regular loads and stores in this paper, augmented only by atomic load-stores (which offer a general method to represent synchronization operations such as locked instructions or compare-and-swap) and a full memory fence. Also, we do not currently model control or data dependencies (which would require us to follow the machine language syntax much more closely, as done in [22], for example).

Some memory models (such as PPC, ARM, RC, and PC) allow stores to be split into separate components for each processor. By combining the **asl** rule with a hierarchical cache organization, our formalism can handle a limited form of store splitting that is sufficient to explain most examples (for more detail on this topic, see [8]).

To correctly handle examples that involve synchronization with spinloops (such as Fig. 1), our formalism must handle infinite executions and model fairness conditions (e.g., the store must eventually be performed). To illustrate the subtleties of infinite rewriting, consider first the program in Fig. 6(a). If we naively

**(a)**

Initially: $A = B = r = s = 0$

| $P1$ | $P2$ |
|---|---|
| $A := 1$ | **while** $(s == 0)$ |
| **while** $(r == 0)$ | $s := A$ |
| $r := B$ | $B := 1$ |

Eventually: $P1, P2$ do not terminate

**(b)**

Initially: $A = B = r = s = 0$

| $P1$ | $P2$ |
|---|---|
| **while** $(r == 0)\{$ | **while** $(s == 0)\{$ |
| $r := B$ | $s := A$ |
| $A := 1$ | $B := 1$ |
| $B := 0$ | $A := 0$ |
| $\}$ | $\}$ |

Eventually: $P1, P2$ do not terminate

**Fig. 6.** (a) This outcome is not possible: the store by P1 has to reach P2 eventually, and vice versa. (b) This outcome is possible: both processors repeat Dekker forever.

allow infinite applications of **ssl**, the store of $A$ can be delayed past the infinite number of subsequent loads in the while loop. As a result, the program may not terminate, which we would like to disallow for the following reason. On actual hardware, stores are not retained indefinitely, so this program is guaranteed to terminate. Now consider Fig. 6(b). This program is essentially a "repeated Dekker" (Fig. 3(b)) and it is conceivable that both P1 and P2 keep executing forever. To explain such behavior, we need to apply **ssl** infinitely often.

To handle both these examples correctly, our denotational semantics uses *parallel* rewriting on infinite traces (to be formally defined in the next section).

### 3.1 A Simple Imperative Language for Shared Memory

We now proceed to formalize our description of relaxed memory models. We start by defining a simple imperative "toy" programming language that is sufficient to express the relevant concepts. It is explicitly parallel and distinguishes syntactically between shared variables (uppercase identifiers) and local variables (lowercase identifiers). All variables are mutable and lexically scoped, and must be initialized. For example, the litmus test in Fig. 3(a) looks as follows:

$$\begin{aligned}&\textbf{share } A = 0 \textbf{ in } (\textbf{share } B = 0 \textbf{ in}\\ &\qquad (\textbf{local } r = 0 \textbf{ in } (\textbf{local } s = 0 \textbf{ in}\\ &\qquad\qquad ((A := 1;\ \ B := 1) \parallel (r := B;\ \ s := A)))))\end{aligned}$$

The formal syntax is shown in Fig. 7. We let $\mathcal{L}$ be the set of shared variables (locations in shared memory), $\mathcal{R}$ be the set of processor-local variables (registers), $\mathcal{V} = \mathcal{L} \cup \mathcal{R}$ be the set of all variables, and $\mathcal{X}$ be the set of values assumed by the variables.

The (load) and (store) statements move values between local and shared variables. The (assign) statement performs computation, such as addition, on local variables. The (compare-and-swap) statement compares the values of $L$ and $r_c$, stores $r_n$ to $L$ if they are equal, and assigns the original value of $L$ to $r_r$. Note that our language does not contain lock or unlock instructions, as there is in fact no

$$
\begin{array}{lcll}
L & \in & \mathcal{L} & \text{(shared variable)} \\
r & \in & \mathcal{R} & \text{(local variable)} \\
x & \in & \mathcal{X} & \text{(value)} \\
f & : & \mathcal{X}^n \to \mathcal{X} & \text{(local computation)}, n \geq 0 \\
s & ::= & \textbf{skip} & \text{(skip)} \\
 & | & r := L & \text{(load)} \\
 & | & L := r & \text{(store)} \\
 & | & r := f(r_1, \ldots, r_n) & \text{(assign)}, n \geq 0 \\
 & | & r_r := \mathit{cas}(L, r_c, r_n) & \text{(compare and swap)} \\
 & | & \mathit{fence} & \text{(full memory fence)} \\
 & | & \mathit{get}\ r & \text{(read from console)} \\
 & | & \mathit{print}\ r & \text{(write to console)} \\
 & | & s;\ s & \text{(sequential composition)} \\
 & | & s_1 \parallel \cdots \parallel s_n & \text{(parallel composition)}, n \geq 2 \\
 & | & \textbf{if}\ r\ \textbf{then}\ s\ \textbf{else}\ s & \text{(conditional)} \\
 & | & \textbf{while}\ r\ \textbf{do}\ s & \text{(loop)} \\
 & | & \textbf{local}\ r = x\ \textbf{in}\ s & \text{(local variable declaration)} \\
 & | & \textbf{share}\ L = x\ \textbf{in}\ s & \text{(shared variable declaration)}
\end{array}
$$

**Fig. 7.** Syntax of program snippets $s$

blocking synchronization at the hardware level (blocking synchronization can be implemented using spinloops and compare-and-swap). We also include a (fence) statement to enforce a full memory fence.

The statements (get) and (print) represent simple I/O in the form of reading from or writing to an interactive console. The statements (sequential composition), (conditional) and (loop) have their usual meaning (we let the special value 0 denote false, and all others denote true). The statement (parallel composition) executes its components concurrently, and waits for all of them to finish before completing. The statements (local) and (shared) declare *mutable* variables and initialize them to the given value. Compared to *let*, as used in functional languages, they differ by (1) allowing mutation of the variable, and (2) strictly restricting the scope and lifetime to the nested snippet.

To enforce that local variables are not accessed concurrently, we define the free variables as in (Fig. 8) and call a snippet *ill-formed* if it contains a parallel composition $s_1 \parallel \cdots \parallel s_n$ such that for some $i, j$, we have $(FV(s_i) \cap FV(s_j) \cap \mathcal{R}) \neq \emptyset$, and *well-formed* otherwise. We let $\mathcal{S}$ be the set of all well-formed snippets.

Finally, we define a *program* to be a well-formed snippet $s$ with no free variables. We let $\mathcal{P}$ be the set of all programs.

Note that conventional hardware memory models consider only a restricted shape of programs (a single parallel composition of sequential processes). Our syntax is more general, as it allows arbitrary nesting of declarations and compositions. This (1) simplifies the definitions and proofs, (2) lets us perform local reasoning (because we can delimit the scope of variables), and (3) allows us to explore the implications of hierarchical memory organizations.

$$
\begin{aligned}
FV(\textbf{skip}) &= \emptyset \\
FV(r := L) &= \{r, L\} \\
FV(L := r) &= \{L, r\} \\
FV(r_0 := f(r_1 \dots r_n)) &= \{r_0, r_1, \dots r_n\} \\
FV(r_r := cas(L, r_c, r_n)) &= \{L, r_r, r_c, r_n\} \\
FV(fence) &= \emptyset \\
FV(get\ r) &= \{r\} \\
FV(print\ r) &= \{r\} \\
FV(s; s') &= FV(s) \cup FV(s') \\
FV(s_1 \parallel \dots \parallel s_n) &= FV(s_1) \cup \dots \cup FV(s_n) \\
FV(\textbf{if } r \textbf{ then } s \textbf{ else } s') &= \{r\} \cup FV(s) \cup FV(s') \\
FV(\textbf{while } r \textbf{ do } s) &= \{r\} \cup FV(s) \\
FV(\textbf{local } r = x \textbf{ in } s) &= FV(s) \setminus \{r\} \\
FV(\textbf{share } L = x \textbf{ in } s) &= FV(s) \setminus \{L\}
\end{aligned}
$$

**Fig. 8.** Definition of the set of free variables $FV(s)$ of $s$

### 3.2 Denotational Semantics

Our semantics mirror the ideas behind the derivation diagrams used in the previous section. Informally speaking, each processor generates a set of potential traces. These traces are concatenated by sequential composition, interleaved by parallel composition, and modified by the dynamic rewrite operations of the memory model. They are then filtered by requiring value consistency (*after* being interleaved and reordered).

To capture the semantics of a program or snippet more formally, we first define a set $\mathcal{B}$ of behaviors; We then recursively define the semantic function $[\![]\!]_M$ to map any snippet $s$ onto the set $[\![s]\!]_M \subset \mathcal{B}$ of its behaviors for a given memory model $M$. We represent the memory model $M$ as a set of dynamic rewrite operations, and model its effect on behaviors as a closure operator.

To capture behaviors locally, we use a combination of state valuations (to capture local state) and event traces (to capture externally visible events and accesses to shared variables). Let $Q$ be the set of local states, defined as functions $\mathcal{R} \to \mathcal{X}$, and let $Evt$ be the set of events $e$ of the form

$$e ::= \langle ld\ \ L, x\rangle \mid \langle st\ \ L, x\rangle \mid \langle ldst\ \ L, x_l, x_s\rangle \mid \langle fence\rangle \mid \langle get\ \ x\rangle \mid \langle print\ \ x\rangle.$$

We let $Evt^*$ be the set of finite event sequences (containing in particular the empty sequence, denoted $\epsilon$), we let $Evt^\omega$ be the set of infinite event sequences, and we let $Evt^\infty = Evt^* \cup Evt^\omega$ be the set of all event sequences. For two sequences $w \in Evt^*$ and $w' \in Evt^\infty$, we let $ww' \in Evt^\infty$ be the concatenation as usual. For a sequence of finite sequences $w_1, w_2, \dots \in Evt^*$, we let $w_1 w_2 \dots \in Evt^\infty$ be the concatenation (which may be finite or infinite).

We then define the set of *behaviors*

$$\mathcal{B} = (Q \times Q \times Evt^*) \cup (Q \times Evt^\infty).$$

A triple $(q, q', w)$ represents a terminating behavior that starts in local state $q$, ends in local state $q'$, and emits the finite event sequence $w$. A pair $(q, w)$

represents a nonterminating behavior that starts in local state $q$ and emits the (finite or infinite) event sequence $w$. For a set $B \subseteq \mathcal{B}$ and states $q, q' \subseteq Q$ we define the projections $[B]_{qq'} = \{w \mid (q, q', w) \in B\}$ and $[B]_q = \{w \mid (q, w) \in B\}$.

To specify dynamic rewrite operations formally, we use *rewrite rules* (as in Fig. 4) of the form $p \xrightarrow{\varphi} q$ where $p$ and $q$ are symbolic event sequences (that is, sequences of events where locations and values are represented by variables) and where $\varphi$ (if present) is a formula over the variables appearing in $p$ and $q$ which describes conditions under which the rewrite rule applies. We let $\mathcal{T}$ be the set of all such rewrite rules.

**Definition 1.** *A* memory model *is a finite set $M \subset \mathcal{T}$ of rewrite rules.*

**Definition 2.** *For a rewrite rule $t = p \xrightarrow{\varphi} q$, let $g_t \subset Evt^* \times Evt^*$ be the set of pairs $(w_1, w_2)$ such that there exists a valuation of the variables in $p, q$ for which $p = w_1$, $q = w_2$ and $\varphi$ is true. Then, define the operator $t : \mathcal{P}(Evt^*) \to \mathcal{P}(Evt^*)$ to map a set $A$ of finite event sequences to the set*

$$t(A) = \{ww_2w' \mid w, w' \in Evt^* \wedge (w_1, w_2) \in g_t \wedge ww_1w' \in A\}$$

For a set of rewrite rules $M \subset \mathcal{T}$ and a set of finite sequences $A \subset Evt^*$, we define the result of applying $M$ to $A$ as $M(A) = A \cup \bigcup_{t \in M} t(A)$. In order to apply $M$ to infinite sequences as well, we first introduce a definition for parallel rewriting. We generalize the notation for sequence concatenation to sets of sequences as usual (elementwise): for example, for $S \subseteq Evt^*$ and $S' \subseteq Evt^\infty$ we let $SS' = \{ss' \mid s \in S, s \in S'\}$.

**Definition 3.** *Let $f : \mathcal{P}(Evt^*) \to \mathcal{P}(Evt^*)$. Then we define the operators $P_f : \mathcal{P}(Evt^*) \to \mathcal{P}(Evt^*)$ and $\widehat{P_f} : \mathcal{P}(Evt^\infty) \to \mathcal{P}(Evt^\infty)$ by*

$$P_f(A) = \bigcup \{ f(A_1) \cdots f(A_n) \mid A_i \subset Evt^* \text{ such that } A_1 \cdots A_n \subseteq A\}$$
$$\widehat{P_f}(\hat{A}) = \bigcup \{ f(A_1)f(A_2)f(A_3) \cdots \mid A_i \subset Evt^* \text{ such that } A_1A_2A_3 \cdots \subseteq \hat{A}\}$$

Note that $\widehat{P_f}(\hat{A})$ may contain infinite sequences even if $\hat{A}$ does not.[1] We now show how to construct fixpoints for the effect of memory models $M \subseteq \mathcal{T}$ on behaviors.

**Definition 4.** *Let $M$ be a memory model. We define $M^* : \mathcal{P}(Evt^*) \to \mathcal{P}(Evt^*)$ and $M^\infty : \mathcal{P}(Evt^\infty) \to \mathcal{P}(Evt^\infty)$ by*

$$M^*(A) = \bigcup_{k \geq 0} M^k(A) \qquad M^\infty(\hat{A}) = \bigcup_{k \geq 0} (\widehat{P_{M^*}})^k(\hat{A})$$

*Moreover, for a set $B \subseteq \mathcal{B}$ of behaviors, define the* closure $B^M =$

$$\{(q, q, w) \mid q, q' \in Q \text{ and } w \in M^*([B]_{qq'})\} \cup \{(q, w) \mid q \in Q \text{ and } w \in M^\infty([B]_q)\}.$$

[1] for example, consider $\hat{A} = \{\epsilon\}$ and $M = \{\epsilon \to 0\}$. Then $\widehat{P_M}(\hat{A})$ contains the infinite sequence $000 \cdots$.

We can show that this is indeed a closure operation, namely, that $(B^M)^M = B^M$ (see our tech report [8] for a proof). Note that our use of parallel rewriting applies the rewrite rules in a "locally finite" manner, which is important to handle infinite executions correctly.[2]

**Definition of the Semantics.** Using the notations listed in the next paragraph, Fig. 9 shows our recursive definition of the semantic function $[\![.]\!]_M : \mathcal{S} \to \mathcal{P}(\mathcal{B})$ that assigns to each snippet $s$ the set of behaviors $[\![s]\!]_M$ that $s$ may exhibit on memory model $M$. It computes behaviors of snippets from the inside out, applying the rewrite rules at each step. Sequential composition appends the behaviors of its constituents, while parallel composition interleaves them. The behaviors of a load include all possible values it could load (because the actual value depends on the context which is not known at this point). Value consistency is enforced at the level of the shared-variable declaration, at which point we also project away accesses to that variable.[3] Fences are modeled as events that do not participate in any rewrite rules, thus enforcing ordering.

**Notations used.** For $q \in Q$, $r \in \mathcal{R}$ and $x \in \mathcal{X}$ we let $q[r \mapsto x]$ denote the function that maps $r$ to $x$, but is otherwise the same as the function $q$. For a shared variable $L \in \mathcal{L}$, let $Evt(L) \subseteq Evt$ be the set of memory accesses to $L$. For $w \in Evt^\infty$ and $i \in \mathbb{N}$, let $w[i] \in Evt$ be the event at position $i$ (starting with 1). Let $dom\ w \subseteq \mathbb{N}$ be the set of positions of $w$. For two sequences $w, w' \in Evt^\infty$ we define the set of fair interleavings $(w \# w') \subseteq Evt^\infty$ to consist of all sequences $u \in Evt^\infty$ such that there exist strictly monotonic functions $f : dom\ w \to dom\ u$ and $g : dom\ w' \to dom\ u$ satisfying $rg\ f \cap rg\ g = \emptyset$ and $rg\ f \cup rg\ g = dom\ w$, and such that $w[i] = u[f(i)]$ and $w'[i] = u[g(i)]$ for all valid positions $i$. Note that the interleaving operator $\#$ is commutative and associative. For a subset of events $C \subseteq Evt$, we define the projection function $proj_C : Evt^\infty \to Evt^\infty$ to map a sequence to the largest subsequence containing only events in $C$. We write $proj_{-L}$ short for the function $proj_{Evt \setminus Evt(L)}$ (which removes all accesses to $L$). We call a sequence $w \in Evt^\infty$ *value-consistent* with respect to a shared variable $L \in \mathcal{L}$ and an initial value $x \in \mathcal{X}$ if for each load of $L$ appearing in $w$, the value loaded matches the value of the rightmost store to $L$ that precedes the load in $w$, or the initial value $x$ if there is no such store. We let $Cons(L, x) \subseteq Evt^\infty$ be the set of all sequences that are value-consistent with respect to $L$ and $x$. Similarly, we let $Cons(L, x, x') \subseteq Evt^*$ be the set of finite sequences that are value-consistent with respect to initial and final values $x$ and $x'$ of $L$, respectively. For simplicity, we assume $\mathcal{X} = \mathbb{Z}$.

[2] For example, consider the operation **ssl** in Fig. 4 which represents the effect of stores being delayed in a buffer; while there is no bound on how long stores can be delayed, they must be eventually performed. Our formalism reflects this properly, as follows (using digits 0,1 instead of load and store events for illustration purposes). Let $A = \{1010\ldots\}$ and $M = \{10 \to 01\}$. Then $0^k 1010\ldots$ is in $M^\infty(A)$, but $000\cdots$ is not.

[3] This behavior is similar to "hide" operators in process algebras. It implies that the behaviors of a program (unlike the behaviors of snippets) contain only external events.

$$
\begin{aligned}
[\![\mathbf{skip}]\!]_M &= \{(q,q,\epsilon) \mid q \in Q\}^M \\
[\![r :=_h L]\!]_M &= \{(q, q[r \mapsto x], \langle ld\ L, x\rangle) \mid q \in Q, x \in \mathcal{X}\}^M \\
[\![L :=_h r]\!]_M &= \{(q,q,\langle st\ L, q(r)\rangle \mid q \in Q\}^M \\
[\![r_0 := f(r_1 \ldots r_n)]\!]_M &= \{(q, q[r_0 \mapsto f(q(r_1)\ldots q(r_n)], \epsilon) \mid q \in Q\}^M \\
[\![r_r := cas_h(L, r_c, r_n)]\!]_M &= \begin{pmatrix} \{(q, q[r_r \mapsto q(r_c)], \langle ldst\ L, q(r_c), q(r_n)\rangle) \mid q \in Q\} \\ \cup\, \{(q, q[r_r \mapsto x], \langle ldst\ L, x, x\rangle) \mid q \in Q, x \in \mathcal{X}, x \neq q(r_c)\} \end{pmatrix}^M \\
[\![get\ r]\!]_M &= \{(q, q[r \mapsto x], \langle get\ x\rangle) \mid q \in Q, x \in \mathcal{X}\}^M \\
[\![print\ r]\!]_M &= \{(q,q,\langle print\ q(r)\rangle \mid q \in Q\}^M
\end{aligned}
$$

$$
[\![s_1; s_2]\!]_M = \begin{pmatrix} \{(q,q',w) \mid \text{there exist } (q,q'',w_1) \in [\![s_1]\!]_M \text{ and } (q'',q',w_2) \in [\![s_2]\!]_M \text{ with } w = w_1 w_2\} \\ \cup\, \{(q,w) \mid (q,w) \in [\![s_1]\!]_M\} \\ \cup\, \{(q,w) \mid \text{there exist } (q,q',w_1) \in [\![s_1]\!]_M \text{ and } (q',w_2) \in [\![s_2]\!]_M \text{ with } w = w_1 w_2\} \end{pmatrix}^M
$$

$$
[\![s_1 \parallel \cdots \parallel s_n]\!]_M = \begin{pmatrix} \{(q,q',w) \mid \text{there exist } (q,q_i,w_i) \in [\![s_i]\!]_M \text{ for all } 1 \leq i \leq n \text{ such that} \\ \qquad w \in w_1 \# \ldots \# w_n \text{ and such that } q'(r) = q_i(r) \text{ for all } r \in FV(s_i) \text{ and} \\ \qquad q'(r) = q(r) \text{ for all } r \notin FV(s_1) \cup \ldots FV(s_n)\} \\ \cup\, \{(q,w) \mid \text{ there exist } w_1, \ldots, w_n \in Evt^\infty \text{ and a nonempty subset } D \subseteq \{1,\ldots,n\} \\ \qquad \text{such that for all } j \in D, \text{ we have a behavior } (q,w_j) \in [\![s_j]\!]_M, \\ \qquad \text{and for all } j \notin D, \text{ we have a behavior } (q,q_j,w_j) \in [\![s_j]\!]_M \text{ for some } q_j, \\ \qquad \text{and } w \in w_1 \# \ldots \# w_n\} \end{pmatrix}^M
$$

$$
[\![\mathbf{if}\ r\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\!]_M = \begin{pmatrix} \{(q,q',w) \mid (q(r) \neq 0 \wedge (q,q',w) \in [\![s_1]\!]_M) \vee (q(r) = 0 \wedge (q,q',w) \in [\![s_2]\!]_M)\} \\ \cup\, \{(q,w) \mid (q(r) \neq 0 \wedge (q,w) \in [\![s_1]\!]_M) \vee (q(r) = 0 \wedge (q,w) \in [\![s_2]\!]_M)\} \end{pmatrix}^M
$$

$$
[\![\mathbf{while}\ r\ \mathbf{do}\ s]\!]_M = \begin{pmatrix} \{(q_0, q_n, w_1 \cdots w_n) \mid \text{there exist } n \geq 0 \text{ and } q_0, \ldots, q_n \text{ such that } (q_i, q_{i+1}, w_{i+1}) \in [\![s]\!]_M \\ \qquad \text{for } 0 \leq i < n, \quad \text{and } q_0(r) \neq 0, \ldots, q_{n-1}(r) \neq 0, \text{ and } q_n(r) = 0\} \\ \cup\, \{(q_0, w_1 w_2 \cdots) \mid \exists q_1, q_2, \ldots : (q_i, q_{i+1}, w_{i+1}) \in [\![s]\!]_M \text{ and } q_i(r) \neq 0\} \\ \cup\, \{(q_0, w_1 \cdots w_n) \mid \text{there exist } n \geq 1 \text{ and } q_0, \ldots, q_{n-1} \text{ such that } q_i(r) \neq 0 \text{ for all } i \text{ and} \\ \qquad (q_i, q_{i+1}, w_{i+1}) \in [\![s]\!]_M \text{ for } 0 \leq i < n-1 \text{ and } (q_{n-1}, w_n) \in [\![s]\!]_M\} \end{pmatrix}^M
$$

$$
[\![\mathbf{local}\ L = x\ \mathbf{in}\ s]\!]_M = \begin{pmatrix} \{(q,q',w) \mid \text{ there exists a behavior } (q[r \mapsto x], q'', w) \in [\![s]\!]_M \\ \qquad \text{such that } q' = q''[r \mapsto q(r)]\} \\ \cup\, \{(q,w) \mid \text{ there exists a behavior } (q[r \mapsto x], w) \in [\![s]\!]_M\} \end{pmatrix}^M
$$

$$
[\![\mathbf{share}\ L = x\ \mathbf{in}\ s]\!]_M = \begin{pmatrix} \{(q,q',w) \mid \text{ there exists a behavior } (q,q',w') \in [\![s]\!]_M \\ \qquad \text{such that } w' \in Cons(L,x) \text{ and } w = proj_{-L}(w')\} \\ \cup\, \{(q,w) \mid \text{ there exists a behavior } (q,w') \in [\![s]\!]_M \\ \qquad \text{such that } w' \in Cons(L,x) \text{ and } w = proj_{-L}(w')\} \end{pmatrix}^M
$$

**Fig. 9.** Denotational Semantics of our Calculus, parameterized by a set $M$ of dynamic rewrite rules. An empty set $M$ represents the standard semantics (sequential consistency).

$p_1$

$$\left[\begin{array}{l}\textbf{local } r = 1 \textbf{ in}\\ \textbf{local } s = 2 \textbf{ in}\\ (\mathit{print}\ r)\ \parallel\ (\mathit{print}\ s)\end{array}\right]$$

$p_2$

$$\left[\begin{array}{l}\textbf{local } r = 1 \textbf{ in}\\ \textbf{local } s = 2 \textbf{ in}\\ \mathit{print}\ r;\\ \mathit{print}\ s\end{array}\right]$$

$p_3$

$$\left[\begin{array}{l}\textbf{local } r = 1 \textbf{ in}\\ \textbf{while } r \textbf{ do}\\ \quad \mathit{print}\ r\end{array}\right]$$

$p_4$

$$\left[\begin{array}{l}\textbf{local } r = 0 \textbf{ in}\\ \mathit{get}\ r;\\ \textbf{while } r \textbf{ do}\\ \quad \textbf{skip};\\ \mathit{print}\ r\end{array}\right]$$

**Fig. 10.** Four example programs. $p_1$ and $p_2$ always terminate, $p_3$ never terminates, and $p_4$ sometimes terminates. $p_1$ can be soundly transformed to $p_2$, but not vice versa.

## 4 Verifying Local Program Transformations

In this section, we present our methodology for verifying the soundness of local program transformations on a chosen hardware memory model. We start with a general definition of what can be observed about a program execution. Next, we show how to prove that a local, static program transformation is unobservable (and thus sound) if its effect on dynamic traces can be captured by *invisible rewrite rules* on those traces (Section 4.1) . For each memory model, we present a list of invisible rewrite rules and describe how we proved invisibility.

For our purposes, the observable behavior of a program includes (1) whether the program terminates or diverges, and (2) the sequence of externally visible events (that is, interactions of the program with the environment). We formalize this by defining the subset $\mathit{Ext} \subset \mathit{Evt}$ of externally visible events and the set $\mathcal{O}$ of observations as

$$\begin{aligned}\mathit{Ext} &= \{\langle \mathit{get}\ n\rangle \mid n \in \mathbb{Z}\} \cup \{\langle \mathit{print}\ n\rangle \mid n \in \mathbb{Z}\}\\ \mathcal{O} &= \{u \mid u \in \mathit{Ext}^*\} \cup \{\nabla u \mid u \in \mathit{Ext}^\infty\}\end{aligned}$$

An observation of the form $u$ represents a terminating execution that produces the finite event sequence $u$; an observation of the form $\nabla u$ represents a nonterminating execution that produces the (finite or infinite) sequence $u$. For example, the program $p_1$ in Fig. 10 has two possible observations, $\langle \mathit{print}\ 1\rangle\langle \mathit{print}\ 2\rangle$ and $\langle \mathit{print}\ 2\rangle\langle \mathit{print}\ 1\rangle$; the program $p_2$ has one possible observation, $\langle \mathit{print}\ 1\rangle\langle \mathit{print}\ 2\rangle$; the program $p_3$ has one possible observation, $\nabla\langle \mathit{print}\ 1\rangle^\omega$; and the program $p_4$ has the set $\{\langle \mathit{get}\ 0\rangle\langle \mathit{print}\ 0\rangle\} \cup \{\nabla\langle \mathit{get}\ n\rangle \mid n \neq 0\}$ of observations.

Using the semantics established in the previous section, we now formally define the set of observations of a program $p$ on a memory model $M$ as follows:

$$\begin{aligned}\mathit{obs}_M(p) = &\{u \mid \exists (q, q', w) \in [\![p]\!]_M : u = \mathit{proj}_{\mathit{Ext}}(w)\}\\ \cup\ &\{\nabla u \mid \exists (q, w) \in [\![p]\!]_M : u = \mathit{proj}_{\mathit{Ext}}(w)\}\end{aligned}$$

For programs $p, p' \in \mathcal{P}$, we let $\langle p \Rightarrow p'\rangle$ represent the *global transformation* of $p$ into $p'$. We then define a global transformation $\langle p \Rightarrow p'\rangle$ to be *sound* for memory model $M$ if it does not introduce any new observations, that is, $\mathit{obs}_M(p') \subseteq \mathit{obs}_M(p)$.

Note that we consider it acceptable if the transformed program has *fewer* observations than the original one. For example, we would consider it o.k. to

$$\begin{aligned}
\textbf{edl}\ (\text{eliminate double load}) &: \langle ld\ L, x\rangle\langle ld\ L, x\rangle \to \langle ld\ L, x\rangle \\
\textbf{eds}\ (\text{eliminate double store}) &: \langle st\ L, x\rangle\langle st\ L, x'\rangle \to \langle st\ L, x'\rangle \\
\textbf{ecs}\ (\text{eliminate confirmed store}) &: \langle st\ L, x\rangle\langle ld\ L, x\rangle \to \langle st\ L, x\rangle \\
\textbf{asl}\ (\text{aggregate store-load}) &: \langle st\ L, x\rangle\langle ld\ L, x\rangle \to \langle st\ L, x\rangle \\
\textbf{iil}\ (\text{invent irrelevant load}) &: \epsilon \to \langle ld\ L, *\rangle \\
\textbf{eil}\ (\text{eliminate irrelevant load}) &: \langle ld\ L, *\rangle \to \epsilon
\end{aligned}$$

**Fig. 11.** A list of rewrite rules that are invisible for certain memory models. The last two contain wildcards; the meaning is that those rules apply to sets of behaviors, rather than indidvidual behaviors.

transform program $p_1$ to program $p_2$ in Fig. 10, which essentially reduces the nondeterministic choices available to the scheduler in scheduling the two print statements. An external entity interacting with the program cannot conclusively detect that a transformation took place. The reason is that schedulers are free to favor certain schedules over others (as long as the schedules themselves are fair). Therefore, an observer can not tell whether the reduction in schedules is caused by the transformation or by a whim of the scheduler.

In this work, we focus on local transformations, that is, transformations of components whose context is not known. See Fig. 12 for 8 examples of local transformations. More formally, we define a *program context* to be a "program with a hole []", defined syntactically as follows:

$$\begin{aligned}
c ::= \quad & [\,] \quad | \quad c\ ;\ s \quad | \quad s\ ;\ c \quad | \quad \textbf{local}\ r = x\ \textbf{in}\ c \quad | \quad \textbf{share}\ L = x\ \textbf{in}\ c \\
& | \quad \textbf{while}\ r\ \textbf{do}\ c \quad | \quad \textbf{if}\ r\ \textbf{then}\ c\ \textbf{else}\ s \quad | \quad \textbf{if}\ r\ \textbf{then}\ s\ \textbf{else}\ c \\
& | \quad s_1 \parallel \cdots \parallel s_{k-1} \parallel c \parallel s_{k+1} \parallel \cdots \parallel s_n \qquad (\text{where } 1 \le k \le n)
\end{aligned}$$

For a context $c$ and snippet $s$, we let $c[s]$ be the snippet obtained by replacing the hole in $c$ with $s$. For two snippets $s, s' \in \mathcal{S}$, we let $\langle s \to s'\rangle$ be a *local transformation*. We say a local transformation $\langle s \to s'\rangle$ *induces* a global transformation $\langle p \Rightarrow p'\rangle$ if there exists a context $c$ such that $p = c[s]$, $p' = c[s']$, and we say a local transformation is *sound* if all induced global transformations are sound.

### 4.1 Invisible Rewrite Rules

To determine whether a local transformation $\langle s \to s'\rangle$ (such as shown in Fig. 12) is sound, we can compare the set of behaviors $[\![s]\!]_M$ and $[\![s']\!]_M$. Because our denotational semantics is defined recursively, it is quite obvious that $[\![s']\!]_M = [\![s]\!]_M$ implies $obs_M(c[s']) = obs_M(c[s])$ in any context $c$, and thus that the transformation is sound. Unfortunately, not all transformations are that simple to prove, because a transformation can be sound even if $[\![s']\!]_M \neq [\![s]\!]_M$ (our semantics is not fully abstract).[4]

[4] For example, consider the "redundant read-after-read elimination" transformation from Fig. 12, and consider $M = SC = \emptyset$. Clearly, the sets $[\![s']\!]_M$ and $[\![s]\!]_M$ are not the same and not contained in each other (all behaviors of $[\![s']\!]_M$ contain one fewer load). Nevertheless, this transformation is actually safe, because the removal of the read can not be observed by any context.

$$\begin{aligned}
\text{(load reordering)}\quad & \{\textbf{if } r \textbf{ then } \{s := A;\ t := B\} \textbf{ else } \{t := B;\ s := A\}\} \\
& \rightarrow \{s := A;\ t := B\} \\
\text{(store reordering)}\quad & \{\textbf{if } r \textbf{ then } \{A := s;\ B := t\} \textbf{ else } \{B := t;\ A := s\}\} \\
& \rightarrow \{A := s;\ B := t\} \\
\text{(irrelevant read elim.)}\quad & \{\textbf{local } r = 0 \textbf{ in } \{r := A; \textbf{ if } r \textbf{ then } \{B := s\} \textbf{ else } \{B := s\}\}\} \\
& \rightarrow \{B := s\} \\
\text{(irrelevant read introd.)}\quad & \{\textbf{if } r \textbf{ then local } s = 0 \textbf{ in } \{s := A;\ B := s\}\} \\
& \rightarrow \{\textbf{local } s = 0 \textbf{ in } \{s := A;\ \textbf{if } r \textbf{ then } B := s\}\}\}
\end{aligned}$$

$$\begin{aligned}
\text{(redundant read-after-read elim.)}\quad & \{r := A;\ b := A\} \rightarrow \{r := A;\ b := r\} \\
\text{(redundant read-after-write elim.)}\quad & \{A := r;\ s := A\} \rightarrow \{A := r;\ s := r\} \\
\text{(redundant write-before-write elim.)}\quad & \{A := r;\ A := s\} \rightarrow \{A := s\} \\
\text{(redundant write-after-read elim.)}\quad & \{r := A; \textbf{if } r == 0 \textbf{ then } A := 0\} \rightarrow \{r := A\}
\end{aligned}$$

**Fig. 12.** Some examples of local transformations [26]. The snippets follow the syntax defined in §3.1, with $\mathcal{L} = \{A, B, \dots\}$ and $\mathcal{R} = \{r, s, t, \dots\}$.

To handle a larger generality of transformations, we introduce the concept of "invisible" rewrite rules on dynamic traces. Essentially, we show that certain dynamic rewrite operations never alter the set of observations. In particular, any rewrite rule that is already part of the memory model is invisible. In general, there can be many more such rules, however. Consider the rules shown in Fig. 11. All of these rules are "invisible" on at least some of the memory models.

More formally, we say a local transformation $\langle s \rightarrow s' \rangle$ is *covered* by a set of rewrite rules $D$ if

$$[\![s']\!]_M \subseteq f_D([\![s]\!]_M),$$

where the operator $f_D : \mathcal{P}(\mathcal{B}) \rightarrow \mathcal{P}(\mathcal{B})$ on behaviors is defined as parallel rewriting[5]

$$[f_D(B)]_{qq'} = P_D([B]_{qq'}) \qquad [f_D(B)]_q = \widehat{P_D}([B]_q).$$

The following definition and theorem relate how invisibility provides the means to prove the soundness of a local transformation by showing that it is covered by some set $D$ of invisible rules.

**Definition 5 (Invisibility).** *Let $D$ be a set of rewrite rules, and let $M$ be a memory model. We say $D$ is* invisible *on $M$ if it is the case that any local transformation that is covered by $D$ is sound for $M$. We say an individual rule $d$ is invisible on $M$ if the set $\{d\}$ is invisible on $M$.*

**Theorem 1.** *The dynamic rewrite rules **edl**, **eds**, **ecs**, **asl**, **eil**, and **iil** are invisible on SC, the rules **edl**, **eds**, **eil**, and **iil** are invisible on TSO,* 390 *and PSO, the rules **edl**, **eds**, **eil**, and **iil** are invisible on CLR, and the set $\{$**eds**, **ecs**$\}$ is invisible on PSO.*

[5] Recall our earlier definition of $[X]_{qq'} = \{w \mid (q, q', w) \in X\}$ and $[X]_q = \{w \mid (q, w) \in X\}$ for a set $X \subset \mathcal{B}$ of behaviors.

The proof of Thm. 1 is based on structural induction, and is available in our tech report [8]. However, walking through the entire proof whenver we wish to enlarge the list of rules or memory models in Thm. 1 is unpractical. Thus, we have broken out a set of conditions that are sufficient to prove invisibility, and can be checked with relative ease.

**Theorem 2 (Simple Conditions for Invisibility).** *Let $M \subseteq \mathcal{T}$ be a memory model, and let $D \subseteq \mathcal{T}$ be a set of rewrite rules. Then the following conditions are sufficient to guarantee that $D$ is invisible on $M$:*

1. ***(Commutativity)**. $m(P_D(A)) \subseteq P_D(M^*(A))$ for all $m \in M$ and $A \subseteq Evt^*$.*
2. ***(Atomicity)**. if $(S_1, S_2) \in G_d$ for some $d \in D$, then all sequences in $S_2$ are of length 0 or 1.*
3. ***(Value Consistency)** if $(S_1, S_2) \in G_d$ for some $d \in D$, and $w_2 \in S_2 \cap Cons(L, x, x')$ for some $L$, $x$, and $x'$, then there exists a $w_1 \in S_1 \cap Cons(L, x, x')$ such that $proj_{-L}(\{w_2\}) \in D(proj_{-L}(\{w_1\}))$.*
4. ***(External Consistency)** if $(S_1, S_2) \in G_d$ for some $d \in D$, then $proj_{Ext}(S_2) \subseteq proj_{Ext}(S_1)$.*

We illustrate the use of these conditions by walking through one case, namely $M = 390 = \{\mathbf{ssl}\}$ and $D = \{\mathbf{edl}\}$. Atomicity is immediate (the right-hand side of **edl** is a single event). Value Consistency is straightforward because the left- and right-hand side of **edl** are functionally equivalent, and the projection $proj_{-L}$ will either map them both to $\epsilon$ or both to themselves. External Consistency is trivial as **edl** does not contain external events. Commutativity requires some work. To show that $\mathbf{ssl}(P_{\mathbf{edl}}(A)) \subset P_{\mathbf{edl}}(\mathbf{ssl}^*(A))$ for all A, we examine $\mathbf{ssl}(P_{\mathbf{edl}}(A))$ and think about all possible scenarios where **ssl** rewrites modified positions of a parallel application of **edl** (if it rewrites only unmodified positions, it clearly commutes with $P_{\mathbf{edl}}$). Thinking about this scenario (matching the left-hand side of **ssl** with the right-hand side of **edl**), we can single out the following situation

$$\begin{aligned} &\langle st\ L, x\rangle\langle ld\ L', x'\rangle\langle ld\ L', x'\rangle \in A \\ &\langle st\ L, x\rangle\langle ld\ L', x'\rangle \in P_D(A) \\ &\langle ld\ L', x'\rangle\langle st\ L, x\rangle \in \mathbf{ssl}(P_{\mathbf{edl}}) \end{aligned}$$

Now we understand that starting with the same first line, we can get to the same last line by first applying **ssl** *twice* and then applying $P_{\mathbf{edl}}$:

$$\begin{aligned} &\langle st\ L, x\rangle\langle ld\ L', x'\rangle\langle ld\ L', x'\rangle \in A \\ &\langle ld\ L', x'\rangle\langle st\ L, x\rangle\langle ld\ L', x'\rangle \in \mathbf{ssl}(A) \\ &\langle ld\ L', x'\rangle\langle ld\ L', x'\rangle\langle st\ L, x\rangle \in \mathbf{ssl}(\mathbf{ssl}(A)) \\ &\qquad \langle ld\ L', x'\rangle\langle st\ L, x\rangle w' \in P_{\mathbf{edl}}(\mathbf{ssl}(\mathbf{ssl}(A))) \end{aligned}$$

which implies the claim.

## 5 Application

To simplify the task of proving or refuting soundness, we automated some parts of the proof by developing a tool called Traver, written in F# and using the

| **transformation name** (see Fig. 12) | *SC* | 390 | *TSO* | *PSO* | *CLR* |
|---|---|---|---|---|---|
| (load reordering) | × | × | × | × | √ |
| (store reordering) | × | × | × | √ | × |
| (irrelevant read elim.) | √ (**eil**) | √ (**eil**) | √ (**eil**) | √ (**eil**) | √ (**eil**) |
| (irrelevant read intr.) | √ (**iil**) | √ (**iil**) | √ (**iil**) | √ (**iil**) | √ (**iil**) |
| (red. read-after-read elim.) | √ (**edl**) | √ (**edl**) | √ (**edl**) | √ (**edl**) | √ (**edl**) |
| (red. wr.-bef.-wr. elim.) | √ (**eds**) | √ (**eds**) | √ (**eds**) | √ (**eds**) | √ (**eds**) |
| (red. read-after-wr. elim.) | √ (**asl**) | × | √ | √ | √ |
| (red. wr.-after-read elim.) | √ (**ecs**) | × | × | √ (**eds**, **ecs**) | × |

**Fig. 13.** Soundness results for the examples from Fig. 12. For sound transformations (marked by √), we list the set $D$ of invisible rules employed by the proof. For unsound transformations (marked by ×), we show example derivations in Fig. 14. All results were validated by our tool.

automated theorem prover Z3 [11]. It operates in one of two modes, verification or falsification.

- In verification mode, Traver takes as input a local transformation $\langle s \rightarrow s' \rangle$, a memory model $M$, and a set $D$ of invisible rewrite rules supplied by the user. It then executes both $s$ and $s'$ symbolically to obtain symbolic representations of their behaviors, and attempts to prove that $D$ covers $\langle s \rightarrow s' \rangle$ by computing the closure of $[\![s]\!]_M$ under $D$ and checking whether it contains $[\![s']\!]_M$. If successful, soundness is established. Otherwise, the result is inconclusive,and Traver reports a behavior in the set difference to the user (which can be inspected to find new candidates for invisible rules that may help to prove soundness, or provide ideas on how to falsify the transformation).
- In falsification mode, Traver takes as input a local transformation $\langle s \rightarrow s' \rangle$, a memory model $M$, and a context $c$ (which may contain several threads). It then computes the closure of $c[s']$ and $c[s]$ under interleavings under $M$, and solves for a behavior of $c[s']$ that is not observationally equivalent to any behavior in $c[s]$ (assuming that *all* initial and final values of all variables are being observed) . If such a behavior is found, soundness has been successfully refuted. Otherwise, the result is inconclusive.

For both modes, the snippets $s, s'$ are supplied to Traver using a sugared syntax, which makes it very easy to try out many different local transformations (however, we currently support loop-free snippets without parallel composition only). The model $M$ is specified by selecting a subset of the rewrite rules in Fig. 4 and Fig. 11 (not including rewrite rules that are conditional on control or data dependencies).

Using our tool, we successfully proved or refuted soundness of the 8 transformations in Fig. 12 for the memory models SC, 390, TSO,[6] PSO, and CLR as defined in Fig.4. The total time needed by the tool to prove/refute all examples is about 15 seconds. The results are shown in Fig. 13.

[6] Note that the results for TSO also apply for x86-TSO and for x86-IRIW

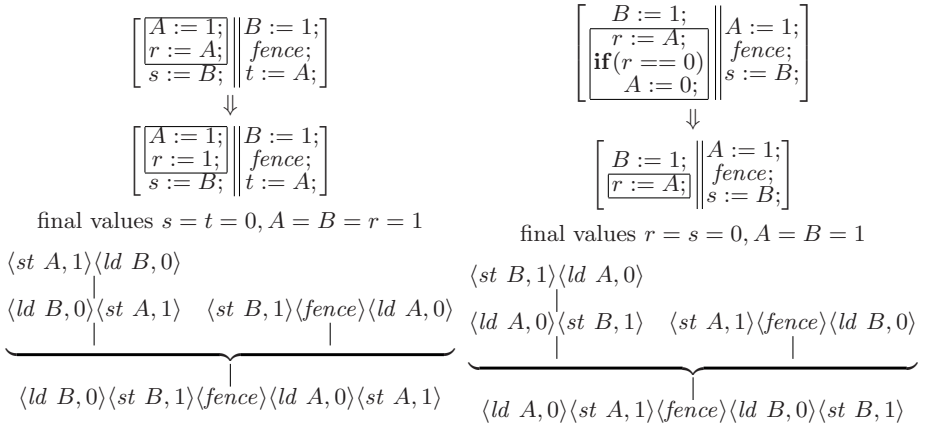


**Fig. 14. (Left.)** Derivation showing that the redundant-read-after-write-elimination is not sound on 390. **(Right.)** Derivation showing that the redundant-write-after-read-elimination is not sound on 390, on *TSO*, and on *CLR*. **(Both.)** We show the original program, the transformed program, and an execution of the transformed program that is not possible on the original program. All shared variables and registers are initially zero.

As expected, the first two transformations (load-reordering, store-reordering) are unsound for all models except models that specifically relax load-load order or store-store order.

The next four transformations (irrelevant-read-elimination, irrelevant-read-introduction, redundant-read-after-read-elimination, and redundant-write-before-write-elimination) are sound for all memory models. The last two transformations proved more interesting. For both, we were able to prove that they are sound on *SC*. However, they exhibit some surprising behavior on relaxed memory models.

- The redundant-read-after-write-elimination is unsound on 390. Fig. 14 (left) shows a derivation to explain this effect. Intuitively, the sequence $\{A := r;\ s := A\}$ has a fence-like effect on 390 which is lost by the transformation. However, on memory models that also support store-load forwarding (**asl**), this transformation is sound.
- The redundant-write-after-read elimination is unsound on 390, *TSO*, and *CLR*, but sound on *PSO*. Fig. 14 (right) shows a derivation to explain this effect. Intuitively, the reason is that because the transformed snippet is a simple load, it can be swapped with a preceding store if the rule **ssl** is part of the memory model. This would not be possible with the original code unless the memory model also contains the rule **sss** which in turn sheds some light on why this transformation is sound for *PSO*.

We believe it would have been very difficult to correctly determine soundness of these transformations (in particular the last two) or to discover the derivations that explain the effects without our proof methodology.

## 6 Conclusion and Future Work

Our experience with Traver has successfully demonstrated the power of formalism and automation in discovering corner cases where normal intuition fails. We believe that the proof methodology and the tool presented in the paper have many more uses in the future. Of particular interest are (1) verifying translations involving different memory models (between different architectures, or between different intermediate representations), and (2) extending our methodology to transformations involving higher-level synchronization such as locks, semaphores, or sending and receiving messages on channels.

## References

1. Adve, S., Gharachorloo, K.: Shared memory consistency models: a tutorial. Computer 29(12), 66–76 (1996)
2. Adve, S., Hill, M.: A unified formalization of four shared-memory models. IEEE Trans. Parallel Distrib. Syst. 4(6), 613–624 (1993)
3. Arvind, Maessen, J.-W.: Memory model = instruction reordering + store atomicity. In: ISCA, pp. 29–40 (2006)
4. Boehm, H.-J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: Programming Language Design and Implementation (PLDI), pp. 68–78 (2008)
5. Boudol, G., Petri, G.: Relaxed memory models: an operational approach. In: Principles of Programming Languages, POPL (2009)
6. Brookes, S.: Full abstraction for a shared variable parallel language. In: LICS, pp. 98–109 (1993)
7. Brumme, C.: cbrumme's weblog, `http://blogs.gotdotnet.com/cbrumme/archive/2003/05/17/51445.aspx`
8. Burckhardt, S., Musuvathi, M., Singh, V.: Verification of compiler transformations for concurrent programs. Technical Report MSR-TR-2008-171, Microsoft Research (2008)
9. Cenciarelli, P., Sibilio, E.: The java memory model: Operationally, denotationally, axiomatically. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 331–346. Springer, Heidelberg (2007)
10. Compaq Computer Corporation. Alpha Architecture Reference Manual, 4th edn. (January 2002)
11. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
12. Duffy, J.: Joe Duffy's Weblog, `http://www.bluebytesoftware.com/blog/2007/11/10/CLR20MemoryModel.aspx`
13. Sarkar, S., et al.: The semantics of x86-CC multiprocessor machine code. In: Principles of Programming Languages, POPL (2009)
14. Gharachorloo, K.: Memory Consistency Models for Shared-Memory Multiprocessors. PhD thesis, University of Utah (2005)
15. Intel Corporation. Intel 64 Architecture Memory Ordering White Paper (August 2007)
16. International Business Machines Corporation. z/Architecture Principles of Operation, 1st edn. (December 2000)

17. Klein, G., Nipkow, T.: A machine-checked model for a java-like language, virtual machine, and compiler. ACM Transactions on Programming Languages and Systems 28(4), 619–695 (2006)
18. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. In: Programming Language Design and Implementation (PLDI), pp. 220–231 (2003)
19. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Principles of programming languages (POPL), pp. 42–54 (2006)
20. Manson, J., Pugh, W., Adve, S.: The Java memory model. In: Principles of Programming Languages (POPL), pp. 378–391 (2005)
21. Morrison, V.: Understand the impact of low-lock techniques in multithreaded apps. MSDN Magazine 20(10) (October 2005)
22. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO (extended version). Technical Report UCAM-CL-TR-745, Univ. of Cambridge (2009)
23. Park, S., Dill, D.L.: An executable specification, analyzer and verifier for RMO (relaxed memory order). In: Symposium on Parallel Algorithms and Architectures (SPAA), pp. 34–41 (1995)
24. Saraswat, V., Jagadeesan, R., Michael, M., von Praun, C.: A theory of memory models. In: PPoPP 2007: Principles and practice of parallel programming, pp. 161–172 (2007)
25. Sevcik, J.: Program Transformations in Weak Memory Models. PhD thesis, University of Edinburgh (2008)
26. Sevcik, J., Aspinall, D.: On validity of program transformations in the Java memory model. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 27–51. Springer, Heidelberg (2008)
27. Shen, X., Arvind, Rudolph, L.: Commit-reconcile & fences (crf): A new memory model for architects and compiler writers. In: ISCA, pp. 150–161 (1999)
28. Weaver, D., Germond, T. (eds.): The SPARC Architecture Manual Version 9. PTR Prentice Hall, Englewood Cliffs (1994)
29. Young, W.D.: A mechanically verified code generator. Journal of Automated Reasoning 5(4), 493–518 (1989)