Brookes is Relaxed, Almost![⋆]

Radha Jagadeesan, Gustavo Petri, and James Riely

School of Computing, DePaul University

Abstract. We revisit the Brookes [1996] semantics for a shared variable parallel programming language in the context of the Total Store Ordering (TSO) relaxed memory model. We describe a denotational semantics that is fully abstract for Brookes' language and also sound for the new commands that are specific to TSO. Our description supports the folklore sentiment about the simplicity of the TSO memory model.

1 Introduction

Sequential Consistency (SC), defined by Lamport [1979], enforces total order on memory operations — reads and writes to the memory — respecting the program order of each individual thread in the program. Operationally, SC is realized by traditional interleaving semantics, where shared memory is represented as a map from locations to values. For such an operational semantics, Brookes [1996] describes a fully abstract denotational view that identifies a process with its transition traces. This technique supports several approaches to program logics for shared memory concurrent programs based on separation logic (see Reynolds [2002] for an early survey). For example, O'Hearn [2007] and Brookes [2007] develop the semantics of Concurrent Separation Logic (CSL), an adaptation of separation logic to reason about concurrent threads operating on shared memory. CSL has been used to prove correctness of several concurrent data structures; for example, [Parkinson et al., 2007] and [Vafeiadis and Parkinson, 2007]. Similarly, Brookes [1996] gives the foundation for refinement approaches to prove the correctness of concurrent data structures such as in [Turon and Wand, 2011].

There are at least two motivations to consider memory models that are *weaker*, or more *relaxed*, than SC: First, modern multicore architectures permit executions that are not sequentially consistent. Second, SC disables some common compiler optimizations for sequential programs, such as the reordering of independent statements. This has led to a large body of work on on relaxed memory models; Adve and Gharachorloo [1996] and Adve and Boehm [2010] provide a tutorial introduction with detailed bibliography on architectures and their impact on language design.

The operational semantics of programming languages in the presence of such relaxed memory models has now been explored. For example, Boudol and Petri [2009] explore the operational semantics of a process language with write buffers; Sevcík et al. [2011] explore the operational semantics of CLight executing with the TSO memory model; and Jagadeesan et al. [2010] describe the operational semantics of an object language under the Java Memory Model (JMM) of Manson et al. [2005].

[⋆] Research supported by NSF 0916741.

However, what has not been investigated in the literature is the denotational semantics of a language with a relaxed memory execution model. We solve this open problem in this paper.

1.1 Overview of The Paper

Our investigations are carried out in the context of the TSO memory model described in SPARC [1994], recently proposed as the model of x86 architectures by Sewell et al. [2010]. In TSO, each sequential thread carries its own write buffer that serves as the initial target of the writes executed by the thread. Thus, TSO permits executions that are not possible with SC.

To illustrate this relaxed behavior let us consider the canonical example depicted in 1 below. We have two sequential threads running in parallel. The left thread, with code ($x := 1;\ y$), sets x to 1 and then reads y returning the value read. The thread on the right, with code ($y := 1;\ x$), sets y to 1 and then reads and returns x. We consider that the initial state has $x = y = 0$. In the SC model, the execution where both threads read 0 is impermissible. It is however achieved by the following TSO execution with write buffers. Below we depict the initial configuration, where both threads have empty buffers (indicated by $\emptyset$) and the memory state is denoted by $\{x := 0, y := 0\}$.

$$\big(\{x := 0, y := 0\},\ \ \langle \emptyset, x := 1; y\rangle \parallel \langle \emptyset, y := 1; x\rangle\big) \tag{1}$$

The writes performed by a thread go into its write buffer (rather than the shared memory). Thus, the above process configuration can evolve to

$$\big(\{x := 0, y := 0\},\ \ \langle [x := 1], y\rangle \parallel \langle [y := 1], x\rangle\big)$$

where $\langle [x := 1], y\rangle$ stands for the thread with local buffer containing the assignment of 1 to x, which is not visible to the other thread, and similarly for $\langle [y := 1], x\rangle$. Now, both reads can return the value from the shared store, which is 0.

Of course, the usual SC executions are also available in a TSO model, which we demonstrate an example execution where both reads yield 1 starting from the initial process configuration. From the intermediate configuration above, both buffer updates can nondeterministically move into memory before the reads execute. Then, we get:

$$\big(\{x := 1, y := 1\},\ \ \langle \emptyset, y\rangle \parallel \langle \emptyset, x\rangle\big)$$

leading to an execution where both reads yield 1.

We provide a precise formalization of the denotational semantics for the language of Brookes [1996] in the context of the TSO memory model. Our model includes the characteristic mfence instructions of TSO, which terminates only when the local buffer of the thread executing the instruction is empty.

Our formalization satisfies the Data Race Free (DRF) property Adve and Hill [1990]. Informally, a program is DRF if no SC execution of the program leads to a state in which a write happens concurrently with another operation on the same location. A DRF *model* requires that the programmer view of computation coincides with SC for programs that satisfy the DRF property.

Let us review [Brookes, 1996] before adapting it to a TSO setting. We use the metavariable s to stand for a shared memory, that is a partial map of variables to values, and C for commands (possibly partially executed). Brookes [1996] views the denotation of a command, $\mathscr{T}[\![C]\!]$, as a set of completed transition traces, ranged by the metavariable α, and with the form $\alpha = (s_0, s'_0) \cdot (s_1, s'_1) \dots (s_n, s'_n)$. These traces describe the interaction between a *system* and its *environment*, where the following conditions hold.

– The execution starts with the command under consideration, so $C_0 = C$.
– Transitions from s_k to s'_k model a *system step*, i.e. $\forall k \in [0,n] \; . \; s_k, C_k \longrightarrow s'_k, C_{k+1}$.
– Transitions from s'_k to s_{k+1} model an *environment step*.
– The transition trace represents a terminated execution, so $C_n = \mathsf{skip}$.

As in any sensible semantics, skip must be a unit for sequential composition.

$$\mathsf{skip};\, C;\, \mathsf{skip} \;\equiv\; C \tag{2}$$

This equation motivates the *stuttering* and *mumbling* closure properties. Closure by stuttering accommodates the case when the system does not move at all but just observes the current state, i.e. $s'_i = s_i$. Closure by mumbling permits the combination of system steps that have no intervening environment step.

We can now describe the model for TSO. The type of command denotations, $\mathscr{T}[\![C]\!]$, changes to a function that takes an input buffer b and yields a set of pairs of the form $\langle \alpha, b' \rangle$ where α is a transition trace as before, and b' is the resulting buffer. The pair $\langle \alpha, b' \rangle$ is to be understood as follows, where we use P's as metavariables for threads, and letting $\alpha = (s_0, s'_0) \cdot (s_1, s'_1) \dots (s_n, s'_n)$.

– The execution of the command starts with the input buffer b, so $P_0 = \langle b, C \rangle$.
– The state pairs still represent system steps, i.e. $\forall k \in [0,n] \; . \; s_k, P_k \longrightarrow s'_k, P_{k+1}$.
– The change from s'_k to s_{k+1} still represents an environment step.
– The transition trace represents a terminated execution leaving b' as the resulting buffer, so $P_n = \langle b', \mathsf{skip} \rangle$. Thus, the pending updates in the resulting buffer b' are yet to reach the shared memory even though there is no command left to be executed.

Our TSO semantics has analogues of the stuttering and mumbling properties for the same reasons as discussed above. In addition, it has two buffer closure properties.

Buffer update closure. Consider the program skip. Executions in $\mathscr{T}[\![\mathsf{skip}]\!](b)$ can result in a smaller buffer b', because buffer updates can propagate into the shared memory. Furthermore, the change from b to b' can be done piecemeal, one buffer update at a time. Thus, skip should permit any executions of $\mathtt{upd}(b)$ defined as the stuttering and mumbling closure of the set

$$\left\{ \langle (s_0, s'_0) \cdots (s_n, s'_n), b' \rangle \mid b = [x_0 := v_0, \dots, x_n := v_n] +\!\!+\, b' \;\&\; \forall i \in [0,n] \; . \; s'_i = s_i[x_i := v_i] \right\}$$

Each step in the above trace corresponds to the addition of one buffer update into memory. Mumbling closure introduces the possibility of multiple buffer updates in one atomic step.

To validate Equation 2, *buffer-update closure* permits data to potentially move from the buffers to shared state before and after any command executes:

$$\frac{\langle \alpha_1, b_1 \rangle \in \mathtt{upd}(b), \langle \alpha_2, b_2 \rangle \in \mathscr{T}[\![C]\!](b_1), \langle \alpha_3, b' \rangle \in \mathtt{upd}(b_2)}{\langle \alpha_1 \cdot \alpha_2 \cdot \alpha_3, b' \rangle \in \mathscr{T}[\![C]\!](b)}$$

Buffer reduction closure. The program $(x := 1; x := 1)$ simulates the program $(x := 1)$ (taking the two steps uninterruptedly), whereas the converse is not true. In buffer terms, this motivates the idea that two identical contiguous writes can be replaced by one copy of the write without leading to any new behaviors. We formalize this notion of buffer simulation as a binary relation $b_1 \rhd b'$ and demand:

$$\frac{\langle \alpha, b_1 \rangle \in \mathscr{T}[\![C]\!](b),\ b_1 \rhd b'}{\langle \alpha, b' \rangle \in \mathscr{T}[\![C]\!](b)}$$

Results. We present the following results.

– We describe operational and denotational semantics for the language that accommodate the extra executions permitted by TSO.

– We prove that our denotational semantics is fully abstract when we observe termination of programs.

– We use the model to identify some equational principles that hold for parallel programs under the TSO memory model.

Our results provide some formal validation for the "folklore" sentiment about the simplicity of the TSO memory model.

Organization of paper. We eschew a separate related works section since we cite the related work in context. In Section 2 we discuss the transition system for the programming language. We develop the model theory in Section 3, and prove the correspondence between operational and denotational semantics in Section 4. In Section 5, we illustrate the differences from Brookes [1996] by describing some laws that hold for programs. More detailed proof sketches are found in a fuller version of the paper.[1]

2 Operational Semantics

We assume disjoint sets of *variables*, x, y and *values*, v. The only values we consider are natural numbers. In conditionals, we interpret non-zero (resp. zero) integers as true (resp. false). As usual we denote by $\mathsf{FV}(C)$ the set of *free variables* of command C.

$$\begin{array}{rcll}
E & ::= & x \mid v \mid E_1 + E_2 \mid \neg E \mid \cdots & \text{(Expression)} \\
C, D & ::= & \mathsf{skip} \mid x := E \mid C; D \mid C \parallel D \mid \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ D & \text{(Command)} \\
 & \mid & \mathsf{while}\ E\ \mathsf{do}\ C \mid \mathsf{local}\ x\ \mathsf{in}\ C \mid \mathsf{await}\ E\ \mathsf{then}\ C \mid \mathsf{mfence} & \\
P, Q & ::= & \langle b, C \rangle \mid P; D \mid P \parallel Q \mid \mathsf{new}\ x := v\ \mathsf{in}\ P & \text{(Process)} \\
\mathbb{P}, \mathbb{Q} & ::= & [-] \mid \mathbb{P}; D \mid \mathbb{P} \parallel Q \mid P \parallel \mathbb{Q} \mid \mathsf{new}\ x := v\ \mathsf{in}\ \mathbb{P} & \text{(Process context)}
\end{array}$$

A *buffer*, $b \in \mathsf{Buff}$, is a list of variable/value pairs, with Buff the domain of all buffers. If $b = [x_1 := v_1, \ldots, x_n := v_n]$, then $dom(b) \triangleq \{x_1, \ldots, x_n\}$. We write $+\!\!+$ for concatenation, $\emptyset$ for the empty buffer and $b|_x$ for the buffer that results from removing x from b. We consider buffer rewrites $(\rhd : \mathsf{Buff} \times \mathsf{Buff})$ that can merge contiguous identical writes, e.g. $[x_1 := v_1, \ldots, x_n := v_n, x_n := v_n] \rhd [x_1 := v_1, \ldots, x_n := v_n]$.

[1] http://fpl.cs.depaul.edu/jriely/papers/2011brookes.pdf

Definition 1. *The relation* $\rhd : \mathsf{Buff} \times \mathsf{Buff}$ *is defined inductively as follows.*

$$\frac{}{(\forall x, v).\ [x := v, x := v] \rhd [x := v]} \qquad \frac{}{b \rhd b} \qquad \frac{b \rhd b_1, b_1 \rhd b'}{b \rhd b'} \qquad \frac{b_1 \rhd b'_1, b_2 \rhd b'_2}{b_1 +\!\!+ b_2 \rhd b'_1 +\!\!+ b'_2}$$

A *memory*, $s \in \Sigma$, is a partial map from variables to values, where Σ is the domain of all memories. We adopt several notation conventions for partial maps: if $s = \{x_1 := v_1, \ldots, x_n := v_n\}$, then $dom(s) \triangleq \{x_1, \ldots, x_n\}$. We write $s[x := v]$ for the memory s with the value of reference x substituted for v, and $s[b]$ to denote the memory which results from applying the updates contained in b from left to right.

As usual, we suppose a semantic function which maps expressions to functions from memories to values (notation $[\![E]\!]s = v$). In the forthcoming transition rules, the memory passed to this function is already updated with (any) relevant buffer.

$$\frac{s(x) = v}{[\![x]\!](s) = v} \qquad \frac{[\![E_1]\!](s) = v_1,\ [\![E_2]\!](s) = v_2}{[\![E_1 + E_2]\!](s) = v_1 + v_2} \qquad \ldots$$

In this paper, we consider that expressions evaluate atomically, following the first language considered in Brookes [1996]. There are two standard approaches to formalizing finer grain semantics; either 1. a compilation of complex expressions to a sequence of simpler commands that only perform a single read or add local variables, or 2. a direct formalization in terms of a transition system as done in the later sections of Brookes [1996]. Our presentation can accommodate either of these changes. We elide details in the interest of space.

Each sequential thread has its own buffer. A process is a parallel composition of sequential threads. A *configuration* is a memory/process pair. Figure 1 defines the evaluation relation $(s, P \longrightarrow s', P')$; where $\longrightarrow^*$ is the reflexive and transitive closure of the relation $\longrightarrow$. $C\{\!\![{}^{y}\!/\!{}_{x}]\!\!\}$ denotes the command derived from C by replacing every occurrence of x with y.

The buffers grow larger in ASSIGN that adds a new update to the buffer, and grow smaller in COMMIT that moves thread local buffer updates into the shared memory. CTXT-BUF allows contiguous and identical updates in the buffer to be removed.

The command skip captures our notion of termination. For example, in SKIP-SEQ, the succeeding command moves into the evaluation context when the preceding process evaluates to skip. When a process terminates, its associated buffer is not necessarily empty; e.g. when $x := E$ terminates, the update to x might still be in the buffer and not yet reflected in the shared memory.

The rule FENCE implements mfence as an assertion that can terminate only when the threads buffer is empty; e.g. $x := E; \mathsf{mfence}$ terminates only when the update to x has been moved to the shared memory, thus making it visible to every other parallel thread.

The rule PAR-CMD enables the initiation of a parallel composition only when the buffer is empty. This restriction is in conformance with Appendix J of SPARC [1994] to ensure that the newly created threads can be scheduled on different processors. For similar reasons, SKIP-PAR ensures that a parallel composition terminates only when the buffers of both parallel processes are empty.

$$\frac{}{s,\, \langle b, \mathsf{while}\ E\ \mathsf{do}\ C\rangle \longrightarrow s,\, \langle b, \mathsf{if}\ E\ \mathsf{then}\ (C; \mathsf{while}\ E\ \mathsf{do}\ C)\ \mathsf{else}\ \mathsf{skip}\rangle}\ (\textsc{while})$$

$$\frac{\llbracket E\rrbracket(s[b]) \neq 0}{s,\, \langle b, \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ D\rangle \longrightarrow s,\, \langle b, C\rangle}\ (\textsc{then}) \qquad \frac{\llbracket E\rrbracket(s[b]) = 0}{s,\, \langle b, \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ D\rangle \longrightarrow s,\, \langle b, D\rangle}\ (\textsc{else})$$

$$\frac{y \notin dom(b) \cup \mathsf{FV}(C)}{s,\, \langle b, \mathsf{local}\ x\ \mathsf{in}\ C\rangle \longrightarrow s,\, \mathsf{new}\ y := 0\ \mathsf{in}\ \langle b, C\{\![{}^{y}\!/\!_{x}]\!\}\rangle}\ (\textsc{local})$$

$$\frac{\llbracket E\rrbracket s \neq 0 \quad s,\, \langle \emptyset, C\rangle \longrightarrow^{*} s',\, \langle \emptyset, \mathsf{skip}\rangle}{s,\, \langle \emptyset, \mathsf{await}\ E\ \mathsf{then}\ C\rangle \longrightarrow s',\, \langle \emptyset, \mathsf{skip}\rangle}\ (\textsc{await}) \qquad \frac{\llbracket E\rrbracket(s[b]) = v}{s,\, \langle b, x := E\rangle \longrightarrow s,\, \langle b +\!\!+ [x := v], \mathsf{skip}\rangle}\ (\textsc{assign})$$

$$\frac{}{s,\, \langle [x := v] +\!\!+ b, C\rangle \longrightarrow s[x := v],\, \langle b, C\rangle}\ (\textsc{commit}) \qquad \frac{}{s,\, \langle \emptyset, \mathsf{mfence}\rangle \longrightarrow s,\, \langle \emptyset, \mathsf{skip}\rangle}\ (\textsc{fence})$$

$$\frac{}{s,\, \langle \emptyset, (C \parallel D)\rangle \longrightarrow s,\, \langle \emptyset, C\rangle \parallel \langle \emptyset, D\rangle}\ (\textsc{par-cmd}) \qquad \frac{}{s,\, \langle \emptyset, \mathsf{skip}\rangle \parallel \langle \emptyset, \mathsf{skip}\rangle \longrightarrow s,\, \langle \emptyset, \mathsf{skip}\rangle}\ (\textsc{skip-par})$$

$$\frac{s, P \longrightarrow s', P'}{s, P \parallel Q \longrightarrow s', P' \parallel Q}\ (\textsc{ctxt-left}) \qquad \frac{s, Q \longrightarrow s', Q'}{s, P \parallel Q \longrightarrow s', P \parallel Q'}\ (\textsc{ctxt-right})$$

$$\frac{}{s,\, \mathsf{new}\ y := v\ \mathsf{in}\ \langle b, \mathsf{skip}\rangle \longrightarrow s,\, \langle b|_{y}, \mathsf{skip}\rangle}\ (\textsc{skip-new}) \qquad \frac{}{s,\, \langle \emptyset, \mathsf{skip}\rangle; D \longrightarrow s,\, \langle \emptyset, D\rangle}\ (\textsc{skip-seq})$$

$$\frac{b \rhd b'}{s,\, \langle b, C\rangle \longrightarrow s',\, \langle b', C\rangle}\ (\textsc{ctxt-buf}) \qquad \frac{s,\, \langle b, C\rangle \longrightarrow s,\, \langle b', C'\rangle}{s,\, \langle b, C; D\rangle \longrightarrow s',\, \langle b', C'; D\rangle}\ (\textsc{ctxt-cmd})$$

$$\frac{s, P \longrightarrow s', P'}{s, P; D \longrightarrow s', P'; D}\ (\textsc{ctxt-seq})$$

$$\frac{s[y := v], P \longrightarrow s', P' \qquad s'(y) = v'}{s,\, \mathsf{new}\ y := v\ \mathsf{in}\ P \longrightarrow s'[y := s(y)],\, \mathsf{new}\ y := v'\ \mathsf{in}\ P'}\ (\textsc{ctxt-new})$$

Fig. 1: Evaluation: $(s, P \longrightarrow s', P')$

Our sole use of the local construct is to provide a model of thread-local registers in the special case when C is a sequential thread. However, our more general formalization permits the description of state that is shared among parallel processes. The process context new $y := v$ in $\mathbb{P}$ carries the shared state of this variable. The hypothesis on the initial buffer in LOCAL ensures that any mfence in C do not affect the global x. The renaming ensures that the updates of CTXT-NEW do not affect the global x. SKIP-NEW discards any remaining updates to the local y. The commands IF and WHILE are standard. The AWAIT construct from Brookes [1996] is a conditional critical region. It provides atomic protection to the entire command C which is expected to be a series of assignments. The compare-and-set instruction of TSO architectures is programmable as follows:

$$\mathtt{cas}(x, v, w) = \mathsf{await}\ 1\ \mathsf{then}\ \mathsf{if}\ x = v\ \mathsf{then}\ x := w\ \mathsf{else}\ x := v;$$

The other atomic instructions of TSO are programmable similarly. Following the semantics of cas in x86-TSO Owens et al. [2009], AWAIT ensures that the buffers are

$$\begin{bmatrix} \mathit{flag}_0 := 1; \\ \textsf{if } \mathit{flag}_1 = 0 \textsf{ then} \\ \quad \mathit{CS_0} \end{bmatrix} \| \begin{bmatrix} \mathit{flag}_1 := 1; \\ \textsf{if } \mathit{flag}_0 = 0 \textsf{ then} \\ \quad \mathit{CS_1} \end{bmatrix}$$

(a) Dekker Mutual Exclusion

$$\begin{bmatrix} \mathit{data} := 1; \\ \mathit{flag} := 1 \end{bmatrix} \| \begin{bmatrix} \textsf{local } r \textsf{ in} \\ \quad \textsf{if } \mathit{flag} = 0 \textsf{ then} \\ \quad\quad r := \mathit{data} \end{bmatrix}$$

(b) Safe Publication

Fig. 2: Examples of TSO Programs

empty before and after the command executes and prevents buffer updates from other threads as the LOKD modifier of Owens et al. [2009].

While TSO does not directly support such multi-instruction atomic conditional critical regions, our semantics continues to be sound for a traditional TSO programming model, only providing the simpler `cas` and the single-word atomics alluded to above. We use this construct to permit a direct comparison with Brookes [1996] and use it (as Brookes [1996]) to construct discriminating contexts in the proof of full abstraction.

Figure 2 presents some standard TSO examples. Dekker's mutual exclusion algorithm (2a) fails under TSO. In initial memories that contain 0 for $flag_0$ and $flag_1$, the initial write of both threads can be put in their internal buffers, remaining unaccessible to the other thread while the reads can proceed before the updates are performed. Thus, both threads can get values 0 for their respective reads and execute their critical sections concurrently. On the other hand, the standard safe publication idiom of Figure 2b is safe under TSO, since the updates of $flag$ and $data$ will proceed in order. Thus, if $flag$ is seen to have value 1 in the right thread to the right, the update of 1 on $data$ has also propagated to the memory.

We end this section by remarking that our programming language satisfies the standard DRF guarantee, following traditional proofs, e.g. see Adve and Gharachorloo [1996], Boudol and Petri [2009], Owens et al. [2009].

3 Denotational Semantics

We use α, β etc. for elements of $(\Sigma \times \Sigma)^\star$, the sequences of state pairs, and ε for the empty trace. We will consider $\mathscr{P}((\Sigma \times \Sigma)^\star)$, the powerset of sequences of state pairs, with the subset ordering. Similar assumptions are made for $\mathscr{P}((\Sigma \times \Sigma)^\star \times \mathsf{Buff})$, ranged by the metavariable $\mathscr{U}$. Commands yield functions in $\mathsf{Buff} \to \mathscr{P}((\Sigma \times \Sigma)^\star \times \mathsf{Buff})$.

Definition 2. *For any $b \in \mathsf{Buff}$, define $\mathscr{T}[\![C]\!](b) \in \mathscr{P}((\Sigma \times \Sigma)^\star \times \mathsf{Buff})$ as follows.*

$$\mathscr{T}[\![C]\!](b) = \big\{ \langle (s_0, s_0') \cdot \ldots \cdot (s_n, s_n')), b' \rangle \mid \forall k \in [0, n-1] \;.\; s_k, P_k \longrightarrow^* s_k', P_{k+1} \;\&\; P_0 = \langle b, C \rangle \;\&\; P_n = \langle b', \mathsf{skip} \rangle \big\}$$

Thus, we only consider transition traces where the residual left of the command is skip, albeit with potentially unfinished buffer updates.

As in [Brookes, 1996], the transition traces are closed under stuttering and mumbling, to capture the reflexivity and transitivity of the operational transition relation.

$$\frac{\langle \alpha \cdot \beta, b \rangle \in \mathscr{U}}{\langle \alpha \cdot (s,s) \cdot \beta, b \rangle \in \mathscr{U}} \ \textsc{Stuttering} \qquad \frac{\langle \alpha \cdot (s,s') \cdot (s',s'') \cdot \beta, b \rangle \in \mathscr{U}}{\langle \alpha \cdot (s,s'') \cdot \beta, b \rangle \in \mathscr{U}} \ \textsc{Mumbling}$$

Let $\mathscr{U} \in \mathscr{P}((\Sigma \times \Sigma)^\star \times \mathsf{Buff})$, we define $\mathscr{U}^\ddagger$ to be the smallest set containing $\mathscr{U}$ such that is stuttering and mumbling closed.

Definition 3. *Define* $\mathtt{upd}(b)$ *to be the stuttering and mumbling closure of*

$$\left\{\langle (s_0,s'_0) \cdots (s_n,s'_n), b'\rangle \mid b = [x_0 := v_0, \ldots, x_n := v_n] \mathbin{+\!\!+} b' \;\&\; \forall k \in [0,n] \,.\, s'_k = s_k[x_k := v_k]\right\}$$

And then we can deduce the inclusion: $\forall b \in \mathsf{Buff} \,.\, \mathtt{upd}(b) \subseteq \mathscr{T}[\![\mathsf{skip}]\!](b)$.

We now let $f : \mathsf{Buff} \to \mathscr{P}((\Sigma \times \Sigma)^\star \times \mathsf{Buff})$, and consider the following closure properties.

$$\frac{\langle \alpha_1, b_1\rangle \in \mathtt{upd}(b),\ \langle \alpha_2, b_2\rangle \in f(b_1),\ \langle \alpha_3, b'\rangle \in \mathtt{upd}(b_2)}{\langle \alpha_1 \cdot \alpha_2 \cdot \alpha_3, b'\rangle \in f(b)}\ \text{BUFF-UPD}$$

$$\frac{\langle \alpha, b_1\rangle \in f(b),\ b_1 \rhd b'}{\langle \alpha, b'\rangle \in f(b)}\ \text{BUFF-RED}$$

Definition 4. *Let* $f : \mathsf{Buff} \to \mathscr{P}((\Sigma \times \Sigma)^\star \times \mathsf{Buff})$. *Then* $f^\dagger$ *is the smallest function (in the pointwise order) such that:*
1. *For all b,* $f(b)$ *is stuttering and mumbling closed.*
2. f *is buffer-update and buffer-reduction closed.*

If $f = f^\dagger$, we say f is closed. Any command yields a closed function.

Lemma 5. *For every command C,* $(\mathscr{T}[\![C]\!])^\dagger = \mathscr{T}[\![C]\!]$.

The following auxiliary definitions enable us to describe the equations satisfied by the transition traces semantics. Let h be a partial function from buffers to sets of transition traces such that $\forall b \in \mathsf{Buff} \,.\, (\exists b_1 \in \mathit{dom}(h) \,.\, (\exists b' \in \mathsf{Buff} \,.\, b = b' \mathbin{+\!\!+} b_1))$; then, there is a unique closed function that contains h. Formally, we overload the closure notation and write:

$$h^\dagger = \lambda b. \{\langle \alpha \cdot \beta, b'\rangle \mid \langle \alpha, b_1\rangle \in \ \mathtt{upd}(b), \langle \beta, b'\rangle \in h(b_1)\}^\dagger$$

We define the operator $\parallel : (\Sigma \times \Sigma)^\star \times (\Sigma \times \Sigma)^\star \to \mathscr{P}^+((\Sigma \times \Sigma)^\star)$ that yields the set of all interleavings of its arguments. We write it infix and define it inductively.

$$\alpha \parallel \varepsilon = \{\alpha\} \qquad \frac{\beta \in \alpha_1 \parallel \alpha_2}{\beta \in \alpha_2 \parallel \alpha_1} \qquad \frac{\beta \in \alpha_1 \parallel \alpha_2}{(s_0,s'_0) \cdot \beta \in ((s_0,s'_0) \cdot \alpha_1) \parallel \alpha_2}$$

We say that the system does not alter x in $(s_0,s'_0) \cdots (s_n,s'_n)$ if $\forall k \in [1,n] \,.\, s_k(x) = s'_k(x)$ and we use $(\Sigma \times \Sigma)^\star_{x+}$ for the set of such transition sequences. We say that the environment does not alter x in $(s_0,s'_0) \cdots (s_n,s'_n)$, if $\forall k \in [1,n-1] \,.\, s'_k(x) = s_{k+1}(x)$ and we use $(\Sigma \times \Sigma)^\star_{x-}$ for the set of such transition sequences. We write $\alpha|_x = \beta|_x$ if traces α and β are identical except for the values of reference x. We write $\mathsf{Buff}|_x$ for the set of buffers that do not have x in their domain. We let $[\![E_{=0}]\!] = \lambda b. \{\langle (s,s), b\rangle \mid [\![E]\!](s[b]) = 0\}^\dagger$ and similarly for $[\![E_{\neq 0}]\!]$.

The transition traces semantics from Definition 2 satisfies the equations of Figure 3.

$$
\begin{array}{rcl}
[\![\mathsf{skip}]\!] & = & \lambda b\ .\ \{\langle \varepsilon, b\rangle\}^\dagger \\
[\![C;D]\!] & = & \lambda b\ .\ \{\langle \alpha\cdot\beta, b'\rangle \mid \exists\, b_1 \in \mathsf{Buff}\ .\ \langle \alpha, b_1\rangle \in [\![C]\!](b), \langle \beta, b'\rangle \in [\![D]\!](b_1)\}^\dagger \\
[\![\mathsf{mfence}]\!] & = & \lambda b\ .\ \{\langle \alpha, \emptyset\rangle \in [\![\mathsf{skip}]\!](b)\} \\
[\![x := E]\!] & = & \lambda b\ .\ \{\langle (s,s), b +\!\!+ [x := v]\rangle \mid [\![E]\!](s[b]) = v\}^\dagger \\
[\![\mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ D]\!] & = & [\![E_{=0}]\!];[\![D]\!] \cup [\![E_{\neq 0}]\!];[\![C]\!] \\
[\![\mathsf{while}\ E\ \mathsf{do}\ C]\!] & = & ([\![E_{\neq 0}]\!];[\![C]\!])^\star;[\![E_{=0}]\!] \\
[\![\mathsf{await}\ E\ \mathsf{then}\ C]\!] & = & \lambda b \in \{\emptyset\}.\ \{\langle (s,s'), \emptyset\rangle \mid [\![E]\!](s) \neq 0, \langle (s,s'), \emptyset\rangle \in [\![C]\!](\emptyset)\}^\dagger \\
[\![C_1 \,\|\, C_2]\!] & = & \lambda b \in \{\emptyset\}\ .\ \{\langle \beta, \emptyset\rangle \mid \beta \in \beta_1 \,\|\, \beta_2,\ \forall i \in [1,2]\ .\ \langle \beta_i, \emptyset\rangle \in [\![C_i]\!](\emptyset)\}^\dagger \\
[\![\mathsf{local}\ x\ \mathsf{in}\ C]\!] & = & \lambda b \in \mathsf{Buff}|_x\ .\ \{\langle \beta, b'|_x\rangle \mid \beta \in (\Sigma\times\Sigma)^\star_{x^+}, \\
 & & \qquad \exists\, \langle \beta_1, b'\rangle \in [\![C]\!](b)\ .\ \beta_1 \in (\Sigma\times\Sigma)^\star_{x^-}\ \&\ \beta|_x = \beta_1|_x\}^\dagger
\end{array}
$$

Fig. 3: Denotational semantics of TSO + await

Lemma 6. *For every command C, $[\![C]\!] = \mathscr{T}[\![C]\!]$*

Proof. The proof is straightforward by induction on the command C. Let us analyze some cases:

- $C \equiv \mathsf{skip}$: $[\![\mathsf{skip}]\!](b_1) = \lambda b\ .\ \{\langle \varepsilon, b\rangle\}^\dagger(b_1) = \mathsf{upd}(b_1) = \mathscr{T}[\![\mathsf{skip}]\!]^\dagger(b_1) = \mathscr{T}[\![\mathsf{skip}]\!](b_1)$ where the last step is given by Lemma 5.
- $C \equiv \mathsf{mfence}$: $[\![\mathsf{mfence}]\!](b_1) = \lambda b\ .\ \{\langle \alpha, \emptyset\rangle \in [\![\mathsf{skip}]\!](b)\}(b_1) = \{\langle \alpha, \emptyset\rangle | \langle \alpha, \emptyset\rangle \in \mathsf{upd}(b_1)\} = \mathscr{T}[\![\mathsf{mfence}]\!](b_1)^\dagger = \mathscr{T}[\![\mathsf{mfence}]\!](b_1)$ where the last two steps are given by mumbling and stuttering closure and Lemma 5.
- $C \equiv x := E$: $[\![x := E]\!](b_1) = \lambda b\ .\ \{\langle (s,s), b +\!\!+ [x := v]\rangle \mid [\![E]\!](s[b]) = v\}^\dagger(b_1) = \{\langle \alpha\cdot (s,s)\cdot\beta, b''\rangle | \langle \alpha, b'\rangle \in \mathsf{upd}(b_1) \& [\![E]\!](s[b']) = v \& \langle \beta, b''\rangle \in \mathsf{upd}(b' +\!\!+ [x := v])\}^\ddagger = \mathscr{T}[\![x := E]\!]^\dagger(b_1) = \mathscr{T}[\![x := E]\!](b_1)$.
- $C \equiv C_0;C_1$: $[\![C_0;C_1]\!] = \lambda b\ .\ \{\langle \alpha\cdot\beta, b'\rangle \mid \exists\, b_1 \in \mathsf{Buff}\ .\ \langle \alpha, b_1\rangle \in [\![C_0]\!](b), \langle \beta, b'\rangle \in [\![C_1]\!](b_1)\}^\dagger = \lambda b\ .\ \{\langle \alpha\cdot\beta, b'\rangle \mid \exists\ b_1 \in \mathsf{Buff}\ .\ \langle \alpha, b_1\rangle \in \mathscr{T}[\![C_0]\!](b), \langle \beta, b'\rangle \in \mathscr{T}[\![C_1]\!](b_1)\}^\dagger = \mathscr{T}[\![C_0;C_1]\!]$ where the final step is given by Lemma 5.

Other cases are routine. □

In this light, we are able to freely interchange $[\![C]\!]$ and $\mathscr{T}[\![C]\!]$ in the rest of this paper.

4 Full Abstraction

We follow Brookes [1996] as closely as possible in this section in order to highlight the differences caused by TSO.

The input-output relation of a program is defined using only the shared memory, i.e. the program is started with an empty buffer and the output state is observed when the buffer is empty.

Definition 7 (Input-Output IO). *For every command C it holds*

$$\mathsf{IO}[\![C]\!] = \{(s,s') \mid \langle (s,s'), \emptyset\rangle \in \mathscr{T}[\![C]\!](\emptyset)\}$$

Definition 8. *The trace $\alpha = (s_0, s'_0) \cdots (s_n, s'_n)$ is Interference Free (IF) if and only if for all $i \in [0, n-1]$ we have $s'_i = s_{i+1}$.*

Notice that every $(s, s') \in \mathsf{IO}[\![C]\!]$ arises from the mumbling closure of IF traces.

We add the following notations for technical convenience:

$$\mathsf{IO}[\![C]\!]/_s = \{(s, s') \mid (s, s') \in \mathsf{IO}[\![C]\!]\}$$
$$[\![C]\!](b)/_s = \{\langle \alpha, b' \rangle \mid \langle \alpha, b' \rangle \in [\![C]\!](b) \ \& \ \alpha = (s, s') \cdot \alpha'\}$$

Definition 9. *The operational ordering compares the IO relation of commands in all possible command contexts $\mathbb{C}[-]$,*

$$C \leqslant_{IO} D \iff \forall \mathbb{C}[-], s \,.\, \mathsf{FV}(\mathbb{C}[C]) \cup \mathsf{FV}(\mathbb{C}[D]) \subseteq \mathsf{dom}(s) \Rightarrow \mathsf{IO}[\![\mathbb{C}[C]]\!]/_s \subseteq \mathsf{IO}[\![\mathbb{C}[D]]\!]/_s$$

Definition 10. *There is a natural ordering induced by the denotational semantics,*

$$C \sqsubseteq D \iff \forall b, s \,.\, \mathsf{FV}(C) \cup \mathsf{FV}(D) \cup \mathsf{dom}(b) \subseteq \mathsf{dom}(s) \Rightarrow [\![C]\!](b)/_s \subseteq [\![D]\!](b)/_s$$

In the rest of this section, we prove that $\sqsubseteq$ and $\leqslant_{IO}$ coincide.

From Figure 3, it is evident that all the program combinators are monotone with respect to set inclusion. Thus, we deduce the following lemma.

Lemma 11 (Compositional Monotonicity). *For all commands C and D it holds*

$$C \sqsubseteq D \Rightarrow \forall \mathbb{C}[-] \,.\, \mathbb{C}[C] \sqsubseteq \mathbb{C}[D]$$

Since $[\![]\!]$ and $\mathscr{T}[\![]\!]$ coincide by Lemma 6 we obtain:

Corollary 12 (Adequacy). *For all commands C and D*

$$C \sqsubseteq D \Rightarrow C \leqslant_{IO} D$$

We now introduce some macros that we will use for the following developments. For any memories s, s' and buffer b, there is evidently an expression $\mathtt{IS}_s$ such that

$$[\![\mathtt{IS}_s]\!](s'[b]) \neq 0 \ \Leftrightarrow \ \mathsf{dom}(s) = \mathsf{dom}(s') \ \& \ (\forall x \in \mathsf{dom}(s) \,.\, s(x) = s'[b](x))$$

Moreover, for all memories s, s' and buffer b, there is evidently a program consisting of a sequence of assignments MAKE_s such that

$$s', \langle b, \mathsf{MAKE}_s \rangle \longrightarrow^* s, \langle \emptyset, \mathsf{skip} \rangle$$

Finally, for each buffer b, there is evidently a program consisting of a sequence of assignments MAKE_b such that for any s, b'

$$s, \langle b', \mathsf{MAKE}_b \rangle \longrightarrow^* s[b'], \langle b, \mathsf{skip} \rangle$$

The program MAKE_b can be used to encode input buffers as a command context.

Lemma 13. *For any command C and buffers b and b' we have*

$$[\![\mathsf{MAKE}_{b'}; C]\!](b) = [\![C]\!](b +\!\!+ b')$$

Proof (SKETCH). By induction on the length of b'. The base case is immediate and the inductive case follows from the definition of sequential composition.

Corollary 14. *For all C_1 and C_2,*

$$C_1 \not\sqsubseteq C_2 \Rightarrow \exists C, s \,.\, [\![C;C_1]\!](\emptyset)/_s \not\subseteq [\![C;C_2]\!](\emptyset)/_s$$

Proof. If $[\![C_1]\!](b)/_s \not\subseteq [\![C_2]\!](b)/_s$, choose $C = \mathsf{MAKE}_b$.

For the proof of our main result we will need to encode a context that simulates the environment of an arbitrary trace α. To that end we define the following program.

Definition 15. *Given $\alpha = (s_0, s'_0) \cdots (s_n, s'_n)$, define the command* $\mathsf{SIMULATE}_\alpha$ *as*

$$\begin{aligned}\mathsf{SIMULATE}_\alpha = &\ \mathsf{await}\ \mathtt{IS}_{s_0}\ \mathsf{then\ skip};\\ &\ \mathsf{await}\ \mathtt{IS}_{s'_0}\ \mathsf{then}\ \mathsf{MAKE}_{s_1};\\ &\ \mathsf{await}\ \mathtt{IS}_{s'_1}\ \mathsf{then}\ \mathsf{MAKE}_{s_2};\\ &\ \ldots\\ &\ \mathsf{await}\ \mathtt{IS}_{s'_{n-1}}\ \mathsf{then}\ \mathsf{MAKE}_{s_n}\end{aligned}$$

Intuitively, $[\![\mathsf{SIMULATE}_\alpha]\!]$ is given by the closure of the single trace that is "complementary" to α. Formally,

$$[\![\mathsf{SIMULATE}_\alpha]\!] = \lambda b \in \{\emptyset\}\ .\{\langle (s'_0, s_1) \cdot (s'_1, s_2) \cdots (s'_{n-1}, s_n), \emptyset\rangle\}^\dagger$$

Lemma 16. *Given α as in Definition 15, letting $\{flag, finish\}$ be disjoint from $\mathsf{FV}(C) \cup \mathsf{dom}(b) \cup \bigcup_i (dom(s_i) \cup dom(s'_i))$, and considering the command context,*

$$\mathbb{C}[-] = flag := 0; finish := 0; \left(\begin{array}{l}\mathsf{MAKE}_{b_0};\\ [-]\end{array} \quad \| \quad \mathsf{SIMULATE}_\alpha\right)$$

we obtain $\langle \alpha, b\rangle \in [\![\mathsf{MAKE}_{b_0};C]\!](\emptyset) \iff \langle \alpha_0 \cdot (s_0, s'_n) \cdot \alpha_1, \emptyset\rangle \in [\![\mathbb{C}[C]]\!](\emptyset)$, where $\langle \alpha', \emptyset\rangle \in [\![\mathsf{SIMULATE}_\alpha]\!](\emptyset)$, $(s_0, s'_n) \in (\alpha \,\|\, \alpha')^\ddagger$, $\langle \alpha_0, \emptyset\rangle \in \mathtt{upd}([flag := 0, finish := 0])$, and $\langle \alpha_1, \emptyset\rangle \in \mathtt{upd}(b)$.

Thus, the lemma characterizes the IF traces where the final state before flushing the final buffer b is s'_n, the first state is s_0 and the trace terminates by flushing the buffer b. The variables $flag$ and $finish$ play essentially no role in this lemma and are included only to accommodate the use-case later.

The proof follows Brookes [1996]. For the forward direction, if $\langle \alpha, b\rangle \in [\![C]\!](\emptyset)$, the following IF trace:

$$\langle (s_0, s'_0) \cdot (s'_0, s_1) \cdot (s_1, s'_1) \cdot (s'_1, s_2) \cdots (s'_{n-1}, s_n) \cdot (s_n, s'_n), b\rangle$$

is in $[\![\mathbb{C}[C]]\!](\emptyset)$ by interleaving. Thus, by mumbling closure, $\langle (s_0, s'_n), b\rangle \in [\![\mathbb{C}[C]]\!](\emptyset)$. Conversely, $\langle (s_0, s'_n), b\rangle \in [\![\mathbb{C}[C]]\!](\emptyset)$ for some b only if there is some β that can be interleaved with $(s'_0, s_1) \cdot (s'_1, s_2) \cdots (s'_{n-1}, s_n)$ to fill up the gaps between s_i and s'_i for all i. Such a trace yields α by stuttering and mumbling.

A significant difference from Brookes [1996] is that we need to check that the final buffers – since they are part of the trace semantics – coincide. To that end, we define the following program CHECK_b that "*observes*" all the updates of buffer b as they are performed one by one into the memory.

Definition 17. *For a buffer $b = [x_1 := v_1, \ldots, x_n := v_n]$ and memories s and $\bar{s}$, define:*

$$\begin{aligned}\mathsf{CHECK}_{b,s,\bar{s}} = {} & \mathsf{await}\ \mathtt{IS}_s\ \mathsf{then}\ \mathsf{MAKE}_{\bar{s}};\\ & \mathsf{await}\ \mathtt{IS}_{\bar{s}[x_1:=v_1]}\ \mathsf{then}\ \mathsf{MAKE}_{\bar{s}};\\ & \ldots\\ & \mathsf{await}\ \mathtt{IS}_{\bar{s}[x_n:=v_n]}\ \mathsf{then}\ \mathsf{MAKE}_{\bar{s}}\end{aligned}$$

Informally, the program CHECK_b starts by replacing the state s for a state $\bar{s}$. In our use case, *$a\bar{m}em$* maps every variable to values that do not appear in the trace generating the state s. The await commands are intended to observe each update from the buffer b of another thread. Upon observing each update in state $\bar{s}$ that state is reinitialized to observe the following buffer update.

Lemma 18. *Let $\{flag, finish\}$ be disjoint from $\mathsf{FV}(C) \cup \mathsf{dom}(b) \bigcup_i (dom(s_i) \cup dom(s'_i))$. Let $\bar{s}$ be any memory such that the range of $\bar{s}$ is disjoint from the range of s and b. Consider the command context*

$$\mathbb{C}[-] = flag := 0; finish := 0;\\ \left(\begin{array}{lcl} & & D;\\ \mathsf{MAKE}_{b_0}; & & \mathsf{await}\ 1\ \mathsf{then}\ flag := 1;\\ [-]; & \parallel & \mathsf{await}\ 1\ \mathsf{then}\ flag := 0;\\ \mathsf{if}\ flag\ \mathsf{then}\ finish := 1 & & \mathsf{CHECK}_{b,s,\bar{s}};\\ & & \mathsf{await}\ \mathtt{IS}_{\bar{s}[finish:=1]}\ \mathsf{then\ skip}\end{array}\right)$$

Then there exist α_0, and α_1 such that $\langle \alpha_0, b\rangle \in [\![C]\!](b_0)$ and $\langle \alpha_1, \varepsilon\rangle \in [\![D]\!](\varepsilon)$ with $(s_0, s) \in (\alpha_0 \| \alpha_1)^\ddagger$ if and only if $\mathsf{IO}[\![\mathbb{C}[C]]\!](\emptyset) \neq \emptyset$.

Proof (SKETCH). Let $\langle \alpha_0, b\rangle \in [\![C]\!](b_0)$ and $\langle \alpha_1, \emptyset\rangle \in [\![D]\!]$ such that $(s_0, s) \in (\alpha_0 \parallel \alpha_1)^\ddagger$. Consider the execution given by the following interleaving:

- obviously we start by executing the initial assignments of $flag$ and $finish$, which are updated before spawning the new threads,
- C, D execute with appropriate an interleaving to yield the shared memory s and a buffer b' for the thread on the left of the parallel component and an empty buffer for the thread on the right (by the semantics of await), where $b' \triangleright b$,
- we then execute the first await on the right hand of the parallel composition to set $flag$ in shared memory,
- the left thread of the parallel composition observes the update on $flag$ and sets $finish$ and this update is added to the buffer of the left hand thread,
- the await on the right hand thread executes unsetting $flag$ in shared memory,
- $\mathsf{CHECK}_{b,s,\bar{s}}$ terminates successfully since the individual awaits can be interleaved with the propagation of buffer updates from b into the shared memory,
- the update to $finish$ moves into shared memory from the buffer of left thread. Since b was exhausted in the previous step, there is no change in shared memory on $dom(\bar{s})$,
- the final await in the right thread terminates successfully because $finish$ is set and the state remains at $\bar{s}[finish := 1]$.

The order is essentially forced on every terminating execution by the synchronization of this context. In order for the right thread to terminate, $finish$ needs to be set by the left thread. The conditional that sets $finish$ in the left thread is constrained to execute between the two updates of $flag$ (within awaits on the right thread). Thus, the full interleaving of C, D has terminated executing the commands before the start of the execution of $\mathsf{CHECK}_{b,s,\bar{s}}$. In order for $\mathsf{CHECK}_{b,s,\bar{s}}$ to terminate successfully, the memory has to coincide with s and it needs to be possible to see the buffer updates of b in sequence. The check that memory is left unaltered at $\bar{s}$ in the final await ensures that there were no further buffer updates caused by the left thread after the termination of $\mathsf{CHECK}_{b,s,\bar{s}}$. □

Lemma 19. $C_1 \not\sqsubseteq C_2 \Rightarrow C_1 \not\leqslant_{IO} C_2$

*Proof (*SKETCH*).* We have to construct a command context to distinguish the IO behavior of C_1, C_2. By Corollary 14, we can assume that $[\![\mathsf{MAKE}_{b_0}; C_1]\!](\emptyset) \not\subseteq [\![\mathsf{MAKE}_{b_0}; C_2]\!](\emptyset)$. Now let $\langle \alpha, b\rangle \in [\![C_1]\!](\emptyset) \setminus [\![C_2]\!](\emptyset)$. Consider the program context

$$\begin{aligned}\mathbb{C}[-] = {} & flag := 0;\\ & finish := 0;\\ & \left(\begin{array}{lcl} & & \mathsf{SIMULATE}_\alpha;\\ \mathsf{MAKE}_{b_0}; & & \mathsf{await}\ 1\ \mathsf{then}\ flag := 1;\\ [-]; & \| & \mathsf{await}\ 1\ \mathsf{then}\ flag := 0;\\ \mathsf{if}\ flag\ \mathsf{then}\ finish := 1 & & \mathsf{CHECK}_{b,s,\bar{s}};\\ & & \mathsf{await}\ \mathtt{IS}_{\bar{s}[finish:=1]}\ \mathsf{then\ skip}\end{array}\right)\end{aligned}$$

where $flag$, $finish$, $\bar{s}$, s and b satisfy the naming constraints of Lemmas 18 and 16. Since $\langle \alpha, b\rangle \in [\![C_1]\!](b_0)$, we use the forward direction of Lemmas 16 and 18 to deduce that $\mathsf{IO}[\![\mathbb{C}[C_1]]\!](\emptyset) \neq \emptyset$. Let $\mathsf{IO}[\![\mathbb{C}[C_2]]\!](\emptyset) \neq \emptyset$. Then there are α_0 and α' with $\langle \alpha_0, b\rangle \in [\![C_2]\!](b_0)$ and $\langle \alpha', \emptyset\rangle \in [\![\mathsf{SIMULATE}_\alpha]\!]$ such that $(s_0, s) \in \{\alpha_0 \| \alpha'\}^\dagger$. So by Lemma 18 $\langle \alpha, b\rangle \in [\![C_2]\!](b_0)$, which is a contradiction. □

Combining Corollary 12 and Lemma 19, we deduce that the denotational semantics $[\![]\!]$ is inequationally fully abstract.

Theorem 20 (Full Abstraction). *For any commands C and D we have*

$$C \sqsubseteq D \iff C \leqslant_{IO} D$$

A simple corollary of the proof of Lemma 19 is that it suffices to consider simple contexts to prove inter-substitutivity of programs. For a given D and a given b, consider:

$$\begin{aligned}\mathbb{C}^b_D[-] = {} & flag := 0; finish := 0;\\ & \left(\begin{array}{lcl} \mathsf{MAKE}_b; & & \\ [-]; & \| & D\\ \mathsf{if}\ flag\ \mathsf{then}\ finish := 1 & & \end{array}\right)\end{aligned}$$

where $flag$, $finish$, satisfy the naming constraints of Lemmas 18 and 16. Then:

$$C_1 \sqsubseteq C_2 \iff (\forall D, b)\ \mathsf{IO}[\![\mathbb{C}^b_D[C_1]]\!] \neq \emptyset \Rightarrow \mathsf{IO}[\![\mathbb{C}^b_D[C_2]]\!] \neq \emptyset$$

This validates the folklore analysis of TSO programs using only sequential testers in parallel.

In Figure 4b we can see that the standard Independent Reads and Independent Writes (IRIW) example is safe in TSO. However, if we consider a simple inlining of that example, as shown in Figure 4a we can simulate the effects of the reordering of reads by forwarding values from the local thread. To see this, we can see that both read that are put in registers r_0 and r_2 can be served from the local buffers *before* the buffers are actually updated, and therefore the subsequent reads that are written in registers r_1 and r_3 can bypass the previous writes pending on buffers, and obtain the initial value in memory of 0.

5 Examples & Laws

We examine some laws of parallel programming under a TSO memory model, and consider some standard TSO examples from the perspective of the denotational semantics introduced in Section 3.

Laws of parallel programming. Most of the laws inherited from Brookes [1996] hold in our setting.

$$\begin{aligned}
\mathsf{skip};C &\equiv C \equiv C;\mathsf{skip} && (1)\\
(C_1;C_2);C_3 &\equiv C_1;(C_2;C_3) && (2)\\
C_1 \| C_2 &\equiv C_2 \| C_1 && (3)\\
(C_1 \| C_2) \| C_3 &\equiv C_1 \| (C_2 \| C_3) && (4)\\
(\mathsf{if}\ E\ \mathsf{then}\ C_0\ \mathsf{else}\ C_1);C &\equiv \mathsf{if}\ E\ \mathsf{then}\ C_0;C\ \mathsf{else}\ C_1;C && (5)\\
\mathsf{while}\ E\ \mathsf{do}\ C &\equiv \mathsf{if}\ E\ \mathsf{then}\ (C;\mathsf{while}\ E\ \mathsf{do}\ C)\ \mathsf{else}\ \mathsf{skip} && (6)
\end{aligned}$$

In (1) and (2) we see that sequential composition is associative with unit skip. Laws (3) and (4) say that parallel composition is commutative and associative. However, skip is not a unit for parallel composition in general, since parallel composition requires flushing the buffers before spawning the threads and when synchronizing them at the end. Instead what holds is:

$$\mathsf{skip} \| C \equiv (\mathsf{mfence};C;\mathsf{mfence})$$

Law (5) implies that sequential composition distributes into conditionals, and finally law (6) is the usual unrolling law for while loops. Also, The usual laws for local variables hold. If x is not free in C then:

$$\begin{aligned}
\mathsf{local}\ x\ \mathsf{in}\ C &\equiv C\\
\mathsf{local}\ x\ \mathsf{in}\ C;D &\equiv C;\mathsf{local}\ x\ \mathsf{in}\ D\\
\mathsf{local}\ x\ \mathsf{in}\ (C \| D) &\equiv C \| \mathsf{local}\ x\ \mathsf{in}\ D
\end{aligned}$$

Thread inlining. Thread inlining is always sound in Brookes [1996], where for example the following rule holds

$$x{:=}y;C \sqsubseteq x{:=}y; \| C$$

In our setting however, this equation holds only if C does not read reference x. In the case where C reads x, C in the left hand side can potentially access newer local updates that are not available globally. In this case, a mfence is needed to validate the equation, i.e.:

$$x{:=}y;\mathsf{mfence};\ C \sqsubseteq x{:=}y \| C$$

Commutation of independent statements. The TSO memory model permits reads to move ahead of previous writes on independent references. This is generally seen with the following example. Using the denotational semantics, we are able to prove the inequality, and moreover the denotations imply the existence of counterexamples to show that the inequality cannot be strengthened to an equality. Thus we get:

$$\begin{bmatrix}\mathsf{local}\ r\ \mathsf{in}\\ \quad r := y;\\ \quad x := 1;\\ \quad z := r;\end{bmatrix} \sqsubseteq \begin{bmatrix}\mathsf{local}\ r\ \mathsf{in}\\ \quad x := 1;\\ \quad r := y;\\ \quad z := r;\end{bmatrix}$$

In the light of the remark following the full abstraction theorem we can give a proof of this example by showing that for any parallel component we can provide a simulation from the program to the left to the one on the right.

Proof. Let b, D, Q be such that

$$s, \mathsf{new}\ r := 0\ \mathsf{in}\ \langle b, r := y; x := 1; z := r; D\rangle \parallel Q \longrightarrow\ s', \mathsf{skip}$$

for some s'. It suffices to show:

$$s, \mathsf{new}\ r := 0\ \mathsf{in}\ \langle b, x := 1; r := y; z := r\rangle \parallel Q \longrightarrow\ s', \mathsf{skip}$$

We construct a simulation relation $\mathscr{R}$ between the configurations that result from s, $\mathsf{new}\ r := 0\ \mathsf{in}\ \langle b, r := y; x := 1; z := r; D\rangle \parallel Q$ and those arising from s, $\mathsf{new}\ r := 0\ \mathsf{in}\ \langle b, x := 1; r := y; z := r;\rangle \parallel Q$.
Define a function $\mapsto$ on the derivatives that result from the left threads,

$$\mathsf{new}\ r := 0\ \mathsf{in}\ \langle b, r := y; x := 1; z := r; D\rangle \mapsto \mathsf{new}\ r := 0\ \mathsf{in}\ \langle b, x := 1; r := y; z := r; D\rangle$$

$$\left.\begin{array}{r}\mathsf{new}\ r := v\ \mathsf{in}\ \langle b +\!\!+ [r := v], x := 1; z := r; D\rangle\\ \mathsf{new}\ r := v\ \mathsf{in}\ \langle \emptyset, x := 1; z := r; D\rangle\\ \mathsf{new}\ r := v\ \mathsf{in}\ \langle b +\!\!+ [r := v, x := 1], z := r; D\rangle\\ \mathsf{new}\ r := v\ \mathsf{in}\ \langle [x := 1], z := r; D\rangle\}\end{array}\right\} \mapsto \mathsf{new}\ r := 0\ \mathsf{in}\ \langle b +\!\!+ [x := 1; r := v], z := r; D\rangle$$

$$P \mapsto P$$

then $\mathscr{R}$ is defined by substituting the left thread by its image under $\mapsto$, i.e.

$$s, P \parallel Q' \quad \mathscr{R} \quad s, \mapsto (P) \parallel Q'$$

Since configurations related by $\mathscr{R}$ have identical memory and coincide for the thread on the right, $\mathscr{R}$ preserves the behavior of threads on the right. Also, $\mapsto$ ensures that reads in the left thread are preserved. It is thus immediate that $\mathscr{R}$ is a simulation. □

Alternatively, we can give a proof only relying on the denotational semantics.

Proof. We first consider $[\![r := y; x := 1]\!](b)$ where $r \notin dom(b)$. There are three cases, depending on how many of the two updates have reached the memory from the buffer.

– (1) Corresponds to the case that neither of the two updates has made it into the memory from the buffer.

– (2) Corresponds to the case that the update to r has made it into memory.

– (3) Corresponds to the case that both updates are now our of the buffer and into memory.

$$\begin{aligned}[\![r:=y;x:=1]\!](b)=&\{\langle\alpha\cdot\alpha',b''\cdot[r\leftarrow s'_0[b'](y)]\cdot[x\leftarrow 1]\rangle \mid \langle\alpha,b'\rangle\in\mathtt{upd}(b)\ \& &(1)\\ &\qquad \alpha=\alpha_0\cdot(s_0,s'_0)\ \&\ \langle\alpha',b''\rangle\in\mathtt{upd}(b')\}\\ &\cup\{\langle\alpha\cdot\alpha',[x\leftarrow 1]\rangle \mid \langle\alpha,b'\rangle\in\mathtt{upd}(b)\ \&\ \alpha=\alpha_0\cdot(s_0,s'_0)\ \& &(2)\\ &\qquad \langle\alpha',\emptyset\rangle\in\mathtt{upd}(b'\cdot[r\leftarrow s'_0[b'](y)])\}\\ &\cup\{\langle\alpha\cdot\alpha',\varepsilon\rangle \mid \langle\alpha,b'\rangle\in\mathtt{upd}(b)\ \&\ \alpha=\alpha_0\cdot(s_0,s'_0)\ \& &(3)\\ &\qquad \langle\alpha',\emptyset\rangle\in\mathtt{upd}(b'\cdot[r\leftarrow s'_0[b'](y)]\cdot[x\leftarrow 1])\}\end{aligned}$$

In the case that b does not contain pending updates on r, the value for r has to come from $r:=y$. This simplifies the calculation of $[\![r:=y;x:=1;z:=r]\!](b)$: we unconditionally deduce an update of $[z\leftarrow s'_0[b'](y)$, for some intermediate state s'_0. This update gets placed at the end of the buffer and reaches the shared state r in the state in case the buffer is empty.

Similarly, consider the semantics of $[\![x:=1;r:=y]\!](b)$ where $r\notin dom(b)$. Again, there are three cases, depending on how many of the two updates have made it from the buffer to memory.

– (1') Corresponds to the case that neither of the two updates has made it into the memory from the buffer.

– (2') Corresponds to the case that the update to x has made it into memory.

– (3') Corresponds to the case that both updates are now our of the buffer and into memory.

$$\begin{aligned}[\![x:=1;r:=y]\!](b)=&\{\langle\alpha\cdot\alpha',b''\cdot[x\leftarrow 1]\cdot[r\leftarrow s'_0[b'](y)]\rangle \mid \langle\alpha,b'\rangle\in\mathtt{upd}(b)\ \& &(1')\\ &\qquad \alpha=\alpha_0\cdot(s_0,s'_0)\ \&\ \langle\alpha',b''\rangle\in\mathtt{upd}(b')\}\\ &\cup\{\langle\alpha\cdot\alpha',[r\leftarrow s'_0[b'](y)]\rangle \mid \langle\alpha,b'\rangle\in\mathtt{upd}(b\cdot[x\leftarrow 1])\ \& &(2')\\ &\qquad \alpha=\alpha_0\cdot(s_0,s'_0)\ \&\ \langle\alpha',\emptyset\rangle\in\mathtt{upd}(b')\}\\ &\cup\{\langle\alpha\cdot\alpha',\varepsilon\rangle \mid \alpha\in\mathtt{upd}b\cdot[x\leftarrow 1],b'\ \&\ \alpha=\alpha_0\cdot(s_0,s'_0)\ \& &(3')\\ &\qquad \langle\alpha',\emptyset\rangle\in\mathtt{upd}(b'\cdot[r\leftarrow s'_0[b'](y)])\}\end{aligned}$$

In the case that b does not contain pending updates on r, the value for r has to come from $r:=y$. This simplifies the calculation of $[\![x:=1;r:=y;z:=r]\!](b)$: we unconditionally deduce an update of $[z\leftarrow s'_0[b'](y)$, for some intermediate state s'_0. This update gets placed at the end of the buffer and reaches the shared state r in the state in case the buffer is empty.

$[\![r:=y;x:=1;z:=r]\!](b)$ only depends on $[\![r:=y;x:=1]\!](b)$ and similarly $[\![x:=1;r:=y;z:=r]\!](b)$ only depends on $[\![x:=1;r:=y]\!](b)$. Consider $[\![\mathsf{local}\ r\ \mathsf{in}\ r:=y;x:=1;z:=r]\!](b)$ and $[\![\mathsf{local}\ r\ \mathsf{in}\ x:=1;r:=y;z:=r]\!](b)$. We reason about the $(\cdot|_r)$ versions of these sets.

– Sets (1) and $(1')$ are identical except for the order of the last two updates in the buffers, so $(1)|_r=(1')|_r$.

– (2) and $(2')$ are different since the former contains, in its last step, an update of r not present in the traces of $(2')$, and the latter contains an update of x, in its last step, not present in (2). However, every trace in $(2)|_r$ coincides with some trace in $(1')|_r$ since the read of y is performed before the update of x and the operator $(\cdot)|_r$ eliminates internal steps of r as well as pending updates on r in the final buffer.

$$\begin{bmatrix}\mathsf{local}\ r_0, r_1\ \mathsf{in}\\ x := 1;\\ r_0 := x;\\ r_1 := y\end{bmatrix} \parallel \begin{bmatrix}\mathsf{local}\ r_2, r_3\ \mathsf{in}\\ y := 1;\\ r_2 := y;\\ r_3 := x\end{bmatrix}$$

Possible: $r_0 = r_2 = 1$ & $r_1 = r_3 = 0$

(a) Buffer Forwarding

$$[x := 1] \parallel [y := 1] \parallel \begin{bmatrix}\mathsf{local}\ r_0, r_1\ \mathsf{in}\\ r_0 := x;\\ r_1 := y\end{bmatrix} \parallel \begin{bmatrix}\mathsf{local}\ r_2, r_3\ \mathsf{in}\\ r_2 := y;\\ r_3 := x\end{bmatrix}$$

Impossible: $r_0 = r_2 = 0$ & $r_1 = r_3 = 1$

(b) IRIW

Fig. 4: TSO Examples

– Sets (3) and $(3')$ are only equal for the cases where the buffer b' is not ε, that is when the read of y is performed before the update of $[x \leftarrow 1]$. The cases where the read of y is performed after the update of x in $(3')$ are not present in (3), since in (3) the read of y is forced to happen before the update of x. The traces of $(3)|_r$ are included in $(2')|_r \cup (3')|_r$.

□

Our denotational model does not record reads. However, the order in which the read of y and the update of x are performed can be observed by the value written to r in some cases. For example, in traces of the form $\alpha_0 \cdot (s_0, s_0[x \leftarrow 1]) \cdot \alpha_1$ if the value written to r does not happen in any state of α_1, the read was actually performed before the update of x. In the above example, these traces are only possible in the program at the right in the traces of $(2')$ and $(3')$. In particular, this implies that $[\![\mathsf{local}\ r\ \mathsf{in}\ x := 1; r := y; z := r]\!](b) \not\sqsubseteq [\![\mathsf{local}\ r\ \mathsf{in}\ r := y; x := 1; z := r]\!](b)$, meaning that

$$\begin{bmatrix}\mathsf{local}\ r\ \mathsf{in}\\ x := 1;\\ r := y;\\ z := r;\end{bmatrix} \not\sqsubseteq \begin{bmatrix}\mathsf{local}\ r\ \mathsf{in}\\ r := y;\\ x := 1;\\ z := r;\end{bmatrix}$$

In general, TSO does not permit writes of independent references or reads of independent reference to commute. However, a special case of this latter class of transformation can be modeled by the capability of reading one threads own writes (as show in the example of Figure 4a). Notice in particular that the example in Figure 4a is a case of inlining of the standard IRIW example (shown in Figure 4b), which provides evidence of our previous claim that inlining is not a legal TSO transformation in general. Our denotational semantics is able to explain this relaxed behavior by means of the inequalities below. In particular, the one on the right can be proved using the inequality discussed above and the one on the left.

$$\begin{bmatrix}\mathsf{local}\ r\ \mathsf{in}\\ x := 1;\\ r := 1;\\ z := r\end{bmatrix} \sqsubseteq \begin{bmatrix}\mathsf{local}\ r\ \mathsf{in}\\ x := 1;\\ r := x;\\ z := r\end{bmatrix} \qquad \begin{bmatrix}\mathsf{local}\ r_1, r_2\ \mathsf{in}\\ r_2 := y;\\ x := 1;\\ r_1 := 1;\\ z_0 := r_1; z_1 := r_2\end{bmatrix} \sqsubseteq \begin{bmatrix}\mathsf{local}\ r_1, r_2\ \mathsf{in}\\ x := 1;\\ r_1 := x;\\ r_2 := y;\\ z_0 := r_1; z_1 := r_2\end{bmatrix}$$

6 Conclusion

We describe how to modify the Brookes semantics for a shared variable parallel programming language Brookes [1996] to address the TSO relaxed memory model. We view our results as the foundations towards two developments: (a) separation logics for relaxed memory models, and (b) refinement theory for relaxed memory models.

References

S. V. Adve and H.-J. Boehm. Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM*, 53:90–101, August 2010. ISSN 0001-0782.

S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *ISCA*, pages 2–14, 1990.

G. Boudol and G. Petri. Relaxed memory models: an operational approach. In *POPL*, pages 392–403, 2009.

S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3): 227–270, 2007.

S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996.

R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *ESOP*, pages 307–326, 2010.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *POPL*, pages 378–391, 2005.

P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375 (1-3):271–307, 2007.

S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: x86-TSO. In *TPHOL*, pages 391–407, 2009.

M. J. Parkinson, R. Bornat, and P. W. O'Hearn. Modular verification of a non-blocking stack. In *POPL*, pages 297–302, 2007.

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *POPL*, pages 43–54, 2011.

P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7): 89–97, 2010.

Inc. CORPORATE. SPARC. *The SPARC Architecture Manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

A. J. Turon and M. Wand. A separation logic for refining concurrent objects. *SIGPLAN Not.*, 46:247–258, January 2011. ISSN 0362-1340.

V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.