Fences in Weak Memory Models: Appendix

Jade Alglave\textsuperscript{1}, Luc Maranget\textsuperscript{1}, Susmit Sarkar\textsuperscript{2}, and Peter Sewell\textsuperscript{2}

\textsuperscript{1} INRIA  \textsuperscript{2} University of Cambridge

Abstract. We present an axiomatic framework, implemented in Coq, to define weak memory models \emph{w.r.t.} several parameters: local reorderings of reads and writes, and visibility of inter and intra processor communications through memory, including full store atomicity relaxation. Thereby, we give a formal hierarchy of weak memory models, in which we provide a formal study of what should be the action and placement of fences to restore a given model such as \emph{SC} from a weaker one. Finally, we provide a tool, \emph{diy}, that tests a given machine to determine the architecture it exhibits. We detail the results of our experiments on Power and the model we extract from it. This identified an implementation error in Power 5 memory barriers (for which IBM is providing a software workaround); our results also suggest that Power 6 does not suffer from this problem.

We present here additional data and explanations, as well as the proof sketches of the results that appear in the paper. Moreover, our results are entirely implemented in the Coq proof assistant: the development is available at http://moscova.inria.fr/~alglave/wmm/.

1 Tables of notations

Communication relaxations

<table>
<thead>
<tr>
<th>diy relaxation</th>
<th>Relation</th>
<th>Source</th>
<th>Target</th>
<th>Processor</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rfe</td>
<td>rfe</td>
<td>W</td>
<td>R</td>
<td>Different</td>
<td>Same</td>
</tr>
<tr>
<td>Rfi</td>
<td>rfi</td>
<td>W</td>
<td>R</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>Wse</td>
<td>wse</td>
<td>W</td>
<td>W</td>
<td>Different</td>
<td>Same</td>
</tr>
<tr>
<td>Wsi</td>
<td>wsi</td>
<td>W</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>Fre</td>
<td>fre</td>
<td>R</td>
<td>W</td>
<td>Different</td>
<td>Same</td>
</tr>
<tr>
<td>Fri</td>
<td>fri</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
</tbody>
</table>

Program order relaxations

\emph{Program order proper}
<table>
<thead>
<tr>
<th>diy relaxation</th>
<th>Relation</th>
<th>Source</th>
<th>Target</th>
<th>Processor</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>PosRR</td>
<td>$\bigcup_{\ell} \xrightarrow{\text{pos}} \cap (\mathbb{R}_L \times \mathbb{R}_L)$</td>
<td>R</td>
<td>R</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>PodRR</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{pod}} \cap (\mathbb{R}<em>{\ell_1} \times \mathbb{R}</em>{\ell_2})$</td>
<td>R</td>
<td>R</td>
<td>Same</td>
<td>Diff</td>
</tr>
<tr>
<td>PosRW</td>
<td>$\bigcup_{\ell} \xrightarrow{\text{pos}} \cap (\mathbb{R}_L \times \mathbb{W}_L)$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>PodRW</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{pod}} \cap (\mathbb{R}<em>{\ell_1} \times \mathbb{W}</em>{\ell_2})$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Diff</td>
</tr>
<tr>
<td>PosWW</td>
<td>$\bigcup_{\ell} \xrightarrow{\text{pos}} \cap (\mathbb{W}_L \times \mathbb{W}_L)$</td>
<td>W</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>PodWW</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{pod}} \cap (\mathbb{W}<em>{\ell_1} \times \mathbb{W}</em>{\ell_2})$</td>
<td>W</td>
<td>W</td>
<td>Same</td>
<td>Diff</td>
</tr>
<tr>
<td>PosWR</td>
<td>$\bigcup_{\ell} \xrightarrow{\text{pos}} \cap (\mathbb{W}_L \times \mathbb{R}_L)$</td>
<td>W</td>
<td>R</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>PodWR</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{pod}} \cap (\mathbb{W}<em>{\ell_1} \times \mathbb{R}</em>{\ell_2})$</td>
<td>W</td>
<td>R</td>
<td>Same</td>
<td>Diff</td>
</tr>
</tbody>
</table>

### Dependencies

<table>
<thead>
<tr>
<th>diy relaxation</th>
<th>Relation</th>
<th>Source</th>
<th>Target</th>
<th>Processor</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>DpsR</td>
<td>$\bigcup_{\ell} \xrightarrow{\text{dps}} \cap (\mathbb{R}_L \times \mathbb{R}_L)$</td>
<td>R</td>
<td>R</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>DpdR</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{dps}} \cap (\mathbb{R}<em>{\ell_1} \times \mathbb{R}</em>{\ell_2})$</td>
<td>R</td>
<td>R</td>
<td>Same</td>
<td>Diff</td>
</tr>
<tr>
<td>DpsW</td>
<td>$\bigcup_{\ell} \xrightarrow{\text{dps}} \cap (\mathbb{R}_L \times \mathbb{W}_L)$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>DpdW</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{dps}} \cap (\mathbb{R}<em>{\ell_1} \times \mathbb{W}</em>{\ell_2})$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Diff</td>
</tr>
</tbody>
</table>

### Barriers relaxations: base cases

#### Sync

<table>
<thead>
<tr>
<th>diy relaxation</th>
<th>Relation</th>
<th>Source</th>
<th>Target</th>
<th>Processor</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyncsRR</td>
<td>$\bigcup_{\ell} \xrightarrow{\text{syncc}} \cap (\mathbb{R}_L \times \mathbb{R}_L)$</td>
<td>R</td>
<td>R</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>SyncdRR</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{syncc}} \cap (\mathbb{R}<em>{\ell_1} \times \mathbb{R}</em>{\ell_2})$</td>
<td>R</td>
<td>R</td>
<td>Same</td>
<td>Diff</td>
</tr>
<tr>
<td>SyncsRW</td>
<td>$\bigcup_{\ell} \xrightarrow{\text{syncc}} \cap (\mathbb{R}_L \times \mathbb{W}_L)$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>SyncdRW</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{syncc}} \cap (\mathbb{R}<em>{\ell_1} \times \mathbb{W}</em>{\ell_2})$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Diff</td>
</tr>
<tr>
<td>SyncsWR</td>
<td>$\bigcup_{\ell} \xrightarrow{\text{syncc}} \cap (\mathbb{W}_L \times \mathbb{R}_L)$</td>
<td>W</td>
<td>R</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>SyncdWR</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{syncc}} \cap (\mathbb{W}<em>{\ell_1} \times \mathbb{R}</em>{\ell_2})$</td>
<td>W</td>
<td>R</td>
<td>Same</td>
<td>Diff</td>
</tr>
<tr>
<td>SyncsWW</td>
<td>$\bigcup_{\ell} \xrightarrow{\text{syncc}} \cap (\mathbb{W}_L \times \mathbb{W}_L)$</td>
<td>W</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>SyncdWW</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{syncc}} \cap (\mathbb{W}<em>{\ell_1} \times \mathbb{W}</em>{\ell_2})$</td>
<td>W</td>
<td>W</td>
<td>Same</td>
<td>Diff</td>
</tr>
</tbody>
</table>

#### LwSync

<table>
<thead>
<tr>
<th>diy relaxation</th>
<th>Relation</th>
<th>Source</th>
<th>Target</th>
<th>Processor</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>LwSyncsRR</td>
<td>$\bigcup_{\ell} \xrightarrow{\text{lwsyncc}} \cap (\mathbb{R}_L \times \mathbb{R}_L)$</td>
<td>R</td>
<td>R</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>LwSyncdRR</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{lwsyncc}} \cap (\mathbb{R}<em>{\ell_1} \times \mathbb{R}</em>{\ell_2})$</td>
<td>R</td>
<td>R</td>
<td>Same</td>
<td>Diff</td>
</tr>
<tr>
<td>LwSyncsRW</td>
<td>$\bigcup_{\ell} \xrightarrow{\text{lwsyncc}} \cap (\mathbb{R}_L \times \mathbb{W}_L)$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>LwSyncdRW</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{lwsyncc}} \cap (\mathbb{R}<em>{\ell_1} \times \mathbb{W}</em>{\ell_2})$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Diff</td>
</tr>
<tr>
<td>LwSyncsWR</td>
<td>$\bigcup_{\ell} \xrightarrow{\text{lwsyncc}} \cap (\mathbb{W}_L \times \mathbb{R}_L)$</td>
<td>W</td>
<td>R</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>LwSyncdWR</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{lwsyncc}} \cap (\mathbb{W}<em>{\ell_1} \times \mathbb{R}</em>{\ell_2})$</td>
<td>W</td>
<td>R</td>
<td>Same</td>
<td>Diff</td>
</tr>
<tr>
<td>LwSyncsWW</td>
<td>$\bigcup_{\ell} \xrightarrow{\text{lwsyncc}} \cap (\mathbb{W}_L \times \mathbb{W}_L)$</td>
<td>W</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>LwSyncdWW</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{lwsyncc}} \cap (\mathbb{W}<em>{\ell_1} \times \mathbb{W}</em>{\ell_2})$</td>
<td>W</td>
<td>W</td>
<td>Same</td>
<td>Diff</td>
</tr>
</tbody>
</table>
# Barriers relaxations: A-cumulativity

### Sync

<table>
<thead>
<tr>
<th>diy relaxation</th>
<th>Relation</th>
<th>Source</th>
<th>Target</th>
<th>Processor</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACSyncsRR</td>
<td>$\cup \ell \xrightarrow{\text{rfc}} (\rightarrow \cap (\mathbb{R}<em>\ell \times \mathbb{R}</em>\ell))$</td>
<td>R</td>
<td>R</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>ACSyncdRR</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{rfc}} (\rightarrow \cap (\mathbb{R}<em>\ell \times \mathbb{R}</em>\ell))$</td>
<td>R</td>
<td>R</td>
<td>Same</td>
<td>Diff</td>
</tr>
<tr>
<td>ACSyncsRW</td>
<td>$\cup \ell \xrightarrow{\text{rfc}} (\rightarrow \cap (\mathbb{R}<em>\ell \times \mathbb{W}</em>\ell))$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>ACSyncdRW</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{rfc}} (\rightarrow \cap (\mathbb{R}<em>\ell \times \mathbb{W}</em>\ell))$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Diff</td>
</tr>
</tbody>
</table>

### LWSync

<table>
<thead>
<tr>
<th>diy relaxation</th>
<th>Relation</th>
<th>Source</th>
<th>Target</th>
<th>Processor</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACLwSyncsRR</td>
<td>$\cup \ell \xrightarrow{\text{rfc}} (\rightarrow \cap (\mathbb{R}<em>\ell \times \mathbb{R}</em>\ell))$</td>
<td>R</td>
<td>R</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>ACLwSyncdRR</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{rfc}} (\rightarrow \cap (\mathbb{R}<em>\ell \times \mathbb{R}</em>\ell))$</td>
<td>R</td>
<td>R</td>
<td>Same</td>
<td>Diff</td>
</tr>
<tr>
<td>ACLwSyncsRW</td>
<td>$\cup \ell \xrightarrow{\text{rfc}} (\rightarrow \cap (\mathbb{R}<em>\ell \times \mathbb{W}</em>\ell))$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>ACLwSyncdRW</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{rfc}} (\rightarrow \cap (\mathbb{R}<em>\ell \times \mathbb{W}</em>\ell))$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Diff</td>
</tr>
</tbody>
</table>

# Barriers relaxations: B-cumulativity

### Sync

<table>
<thead>
<tr>
<th>diy relaxation</th>
<th>Relation</th>
<th>Source</th>
<th>Target</th>
<th>Processor</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCSyncsRW</td>
<td>$\cup \xrightarrow{\text{sync}} (\rightarrow \cap (\mathbb{R}<em>\ell \times \mathbb{W}</em>\ell)); \xrightarrow{\text{rfc}}$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>BCSyncdRW</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{sync}} (\rightarrow \cap (\mathbb{R}<em>\ell \times \mathbb{W}</em>\ell)); \xrightarrow{\text{rfc}}$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Diff</td>
</tr>
<tr>
<td>BCSyncsWW</td>
<td>$\cup \xrightarrow{\text{sync}} (\rightarrow \cap (\mathbb{W}<em>\ell \times \mathbb{W}</em>\ell)); \xrightarrow{\text{rfc}}$</td>
<td>W</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>BCSyncdWW</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{sync}} (\rightarrow \cap (\mathbb{W}<em>\ell \times \mathbb{W}</em>\ell)); \xrightarrow{\text{rfc}}$</td>
<td>W</td>
<td>W</td>
<td>Same</td>
<td>Diff</td>
</tr>
</tbody>
</table>

### LWSync

<table>
<thead>
<tr>
<th>diy relaxation</th>
<th>Relation</th>
<th>Source</th>
<th>Target</th>
<th>Processor</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCLwSyncsRW</td>
<td>$\cup \xrightarrow{\text{lwsync}} (\rightarrow \cap (\mathbb{R}<em>\ell \times \mathbb{W}</em>\ell)); \xrightarrow{\text{rfc}}$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>BCLwSyncdRW</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{lwsync}} (\rightarrow \cap (\mathbb{R}<em>\ell \times \mathbb{W}</em>\ell)); \xrightarrow{\text{rfc}}$</td>
<td>R</td>
<td>W</td>
<td>Same</td>
<td>Diff</td>
</tr>
<tr>
<td>BCLwSyncsWW</td>
<td>$\cup \xrightarrow{\text{lwsync}} (\rightarrow \cap (\mathbb{W}<em>\ell \times \mathbb{W}</em>\ell)); \xrightarrow{\text{rfc}}$</td>
<td>W</td>
<td>W</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>BCLwSyncdWW</td>
<td>$\bigcup_{\ell_1 \neq \ell_2} \xrightarrow{\text{lwsync}} (\rightarrow \cap (\mathbb{W}<em>\ell \times \mathbb{W}</em>\ell)); \xrightarrow{\text{rfc}}$</td>
<td>W</td>
<td>W</td>
<td>Same</td>
<td>Diff</td>
</tr>
</tbody>
</table>

## 2 Properties of validity

From our definition of validity arises a simple notion of comparison among architectures. $A_1 \leq A_2$ means that $A_1$ is weaker than $A_2$:

**Definition 1 (Weaker).**

$$A_1 \leq A_2 \triangleq (\text{ppo}_1 \leq \text{ppo}_2) \land (\text{grf}_1 \leq \text{grf}_2)$$
2.1 Validity is decreasing
We prove validity of an execution to be decreasing w.r.t the weaker predicate; thus, a weak architecture exhibits at least all the behaviours authorised by a stronger one:

Theorem 1 (Validity is decreasing).
\[ \forall A_1 A_2 \ (A_1 \leq A_2) \Rightarrow (\forall X, A'_2 \text{ valid}(X) \Rightarrow A'_1 \text{ valid}(X)) \]

Proof (in Coq). From \( A_1 \leq A_2 \), we immediately have \( A_1.ghb \subseteq A_2.ghb \), thus if \( A_2.ghb \) is acyclic, so is \( A_1.ghb \). □

2.2 Monotonicity of validity
Programs running on an architecture \( A_1 \) exhibit executions that would be valid on a stronger architecture \( A_2 \); we characterise all such executions as follows:

\[ A_1.\text{check}_{A_2}(X) \triangleq \text{acyclic}(\text{grf}_{A_2} \cup \text{ws} \cup \text{fr} \cup \text{ppo}_{A_2}) \]

Theorem 2 (Characterisation).
\[ \forall A_1 A_2 \ (A_1 \leq A_2) \Rightarrow (\forall X, (A'_1 \text{ valid}(X) \land A_1.\text{check}_{A_2}(X)) \iff A'_2 \text{ valid}(X)) \]

Proof (in Coq).
⇒ \( X \) being valid on \( A_1 \), we have all requirements – well formedness and uniproc – to guarantee it is valid on \( A_2 \), except the last predicate, which holds by the hypothesis \( \text{check}_{A_2} \).
⇐ \( X \) being valid on \( A_2 \) gives us all requirements – well formedness and uniproc – to guarantee its validity on \( A_1 \) except the last one. As \( A_1 \leq A_2 \), we know that \( A_1.ghb \subseteq A_2.ghb \) (lemma \( \text{ghb} \text{incl} \)), thus the acyclicity requirement for \( A_1.ghb \) holds if \( A_2.ghb \) is acyclic. □

3 Examples of models in our framework
3.1 SC
Definitions
\[ Sc \triangleq (MM, \text{rf}, \emptyset) \]
\[ \text{hb-seq} \triangleq \text{ws} \cup \text{fr} \cup \text{rf} \]

Sc characterisation
Corollary 1 (Sc characterisation).
\[ \forall A, (A \leq Sc) \Rightarrow (\forall X, \text{valid } AX \land A.\text{check}_{Sc} X \iff Sc.\text{ valid } X) \]

Proof (in Coq).
⇒ as \( \text{po} \cup \text{hb-seq} = Sc.ghb \), this is a direct consequence of thm. 2.
⇐ as \( A \leq Sc \), this is a direct consequence of thm. 1. □
Sc is SC. SC has been defined in [1] as follows:

[...] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Thus an SC execution is a total order $\text{ex} \rightarrow$ consistent with the program order:

$$\text{seq}(\text{ex} \rightarrow) \triangleq \text{total-order}(\text{ex} \rightarrow, \mathcal{E}) \land \mathcal{P}_o \subseteq \text{ex} \rightarrow$$

The implicit execution model of [1] states that a read $r$ takes its value from the most recent write that precedes it in $\text{ex} \rightarrow$. Writing $\text{pw}(o, r)$ for the set of writes to the same location that precede $r$ according to partial order $o$, we extract our $\text{rf}$ relation from $\text{ex} \rightarrow$ as follows:

$$\text{pw}(\text{ex} \rightarrow, r) \triangleq \{w \mid \text{loc}(w) = \text{loc}(r) \land w \text{ex} \rightarrow r\}$$

$$\text{SC. rf}(\text{ex} \rightarrow) \triangleq \{(w, r) \mid w = \max_{\text{ex} \rightarrow} \left(\text{pw}(\text{ex} \rightarrow, r)\right)\}$$

We extract write serialisation as well, and one of our execution witnesses from $\text{ex} \rightarrow$, which we use to show that the SC definition from [1] is equivalent to ours:

$$\text{SC. ws}(\text{ex} \rightarrow) \triangleq \bigcup_{\ell} (\mathcal{W}_\ell \times \mathcal{W}_\ell) \cap \text{ex} \rightarrow$$

$$\text{SC. wit}(\text{ex} \rightarrow) \triangleq (\mathcal{E}, \mathcal{P}_o, \text{SC. rf}(\text{ex} \rightarrow), \text{SC. ws}(\text{ex} \rightarrow))$$

**Theorem 3 (Sc is SC).**

$$\forall X, \text{Sc. valid}(X) \Leftrightarrow \exists \text{ex} \rightarrow, \text{seq}(\text{ex} \rightarrow) \land \text{SC. wit}(\text{ex} \rightarrow) = X$$

**Proof (in Coq).**

$\Rightarrow$ from $X$ being valid on $\text{Sc}$, we have acyclic($\text{ghb} \rightarrow$), that is acyclic($\text{hb-seq} \cup \mathcal{P}_o$) on $\text{Sc}$, which gives us an equivalent SC execution by Cor. 1.

$\Leftarrow$ from $\text{ex} \rightarrow$, we produce a SC. wit which is valid by Thm. 1.

\[\square\]

### 3.2 TSO

**Definitions**

$$\text{ppo-tso} \rightarrow \triangleq \mathcal{R} \cup \mathcal{W}$$

$$T_{so}^x \triangleq (\text{ppo-tso} \rightarrow, \text{rfe}, \emptyset)$$

$$\text{hb-tso} \rightarrow \triangleq \text{ws} \rightarrow \cup \text{fr} \rightarrow \cup \text{rfe}$$
**Tso characterisation** By Thm. 2 we show the following criterion – where \( \text{hb-tso} \) is \( \rightarrow_{\text{ws}} \cup \rightarrow_{\text{fr}} \cup \rightarrow_{\text{rf}} \) – characterises valid executions that are Tso\(^*\) on any \( A \leq \text{Tso} \):

\[
A \cdot \text{check}_{\text{Tso}}(X) \triangleq \text{acyclic}(\rightarrow_{\text{ppo-tso}} \cup \rightarrow_{\text{hb-tso}})
\]

**Corollary 2 (Tso characterisation).**

\[
\forall A, (A \leq \text{Tso}) \Rightarrow (\forall X, \text{valid } AX \land A \cdot \text{check}_{\text{Tso}} X \iff \text{valid } \text{Tso}(X))
\]

**Proof (in Coq).**

\[
\Rightarrow \text{As } \rightarrow_{\text{ppo-tso}} \cup \rightarrow_{\text{hb-tso}} = \text{Tso} \cdot \rightarrow_{\text{ghb}}, \text{this is a direct consequence of thm. 2.}
\]

\[
\Leftarrow \text{as } A \leq \text{Tso, this is a direct consequence of thm. 1.} \quad \Box
\]

Tso is TSO

**Equivalence with the native model** Sparc [3, V. 8, Appendix K] formally defines a TSO execution as a partial order \( \rightarrow_{\text{ex}} \) on memory events constrained by some axioms that constrain \( \rightarrow_{\text{ex}} \). We formulate those as follows:

\[
\text{ptso}(\rightarrow_{\text{ex}}) \triangleq \text{partial-order}(\rightarrow_{\text{ex}}, E) \land \text{RM} \subseteq \rightarrow_{\text{ex}} \land \text{WW} \subseteq \rightarrow_{\text{ex}} \land \exists \text{tso} \rightarrow \text{tso} \subseteq \rightarrow_{\text{ex}} \land \text{total-order}(\rightarrow_{\text{tso}}, \text{W})
\]

Note that \( \text{tso} \rightarrow \) above is the postulated total order on stores that justify the name TSO. An additional axiom, Value, where \( L_a \) (resp. \( S_a \)) is the notation of [3] for a load (resp. a store) with location \( a \):

\[
\text{Val}(L_a) = \text{Val}\left(\text{max}_{\rightarrow_{\text{ex}}} \{S_a \mid S_a \rightarrow_{\text{ex}} L_a \lor S_a \rightarrow_{\text{po}} L_a\}\right)
\]

[...] states that the value of a data load is the value written by the most recent store to that location. Two terms combine to define the most recent store. The first corresponds to stores by other processors, while the second corresponds to stores by the processor that issued the load.

We relate Sparc TSO and our Tso by interpreting the Value axiom as specifying \( \rightarrow_{\text{rf}}, \) and extract \( \rightarrow_{\text{ws}} \) from \( \rightarrow_{\text{ex}}, \) as we did in the Sc case:

\[
\text{TSO} \cdot \text{rf}(\rightarrow_{\text{ex}}) \triangleq \{(w, r) \mid w = \text{max}_{\rightarrow_{\text{ex}}} (\text{pw}(\rightarrow_{\text{ex}} \cup \rightarrow_{\text{po}}, r))\}
\]

\[
\text{TSO} \cdot \text{ws}(\rightarrow_{\text{ex}}) \triangleq \{(w_1, w_2) \mid \exists l, (w_1, w_2) \in (\text{W}_l \times \text{W}_l) \land w_1 \rightarrow_{\text{ex}} w_2\}
\]

\[
\text{TSO} \cdot \text{wit}(\rightarrow_{\text{ex}}) \triangleq (E, \rightarrow_{\text{po}}, \text{TSO} \cdot \text{rf}(\rightarrow_{\text{ex}}), \text{SC} \cdot \text{ws}(\rightarrow_{\text{ex}}))
\]

**Theorem 4 (Tso is TSO).**

\[
\forall X, \text{Tso}^* \cdot \text{valid}(X) \iff \exists \rightarrow_{\text{ex}}, \text{ptso}(\rightarrow_{\text{ex}}) \land \text{TSO} \cdot \text{wit}(\rightarrow_{\text{ex}}) = X
\]

\(^1\) We omit the axioms *Atomicity* and *Termination.*
Proof (in Coq).
⇒ from $X$ being valid on $Tso$, we have $\text{acyclic}^{\text{ghb}}$, that is $\text{acyclic}^{\text{ghb-tso} \cup \text{po-tso}}$ on $Tso$, which gives us an equivalent $TSO$ execution by Cor. 2.
⇐ from $\text{ex}$, we produce a $TSO$. wit which is valid by Thm. 1.

4 Barriers

From now on, we note $A$. $\text{hb-seq}$ for $\text{?rfe}$ $\rightarrow \cup \text{ws} \rightarrow \text{fr}$ $\rightarrow \text{fr}$ $\rightarrow \text{fr}$ $\rightarrow \text{fr}$. We trivially have $A$. $\text{hb-seq}$ $\rightarrow \subseteq A$. $\text{hb-seq}$, and that $\text{hb-seq}$ is transitive.

A key lemma is that forall relation $r \rightarrow$ over events, such that there is a cycle in $\text{hb-seq}$ $\rightarrow \cup r \rightarrow$, then there is a cycle in $\text{hb-seq}$ $\rightarrow (\text{+} r \rightarrow)$.

4.1 Barriers guarantee

Consider two architectures $A_1 \leq A_2$. We define the predicate $\text{fb}$ (fully barriered) on $A_1 \leq A_2$, where $\text{rfi} \rightarrow \text{rfi}$ is the set difference, and $x \rightarrow z, x \rightarrow z \wedge z \rightarrow y$ stands for the sequence:

$$A_1.\text{fb}_{A_2}(X) \triangleq (\text{ppo} \rightarrow \text{grf} \rightarrow \text{rfi} \rightarrow \text{ppo} \rightarrow) \subseteq \text{ab}$$

We prove that the above condition on $\text{ab}$ suffices to restore $A_2$ from $A_1$:

Theorem 5 (Barriers guarantee).

$$\forall A_1, A_2, (A_1 \leq A_2) \Rightarrow (\forall X, \text{valid}(X) \wedge A_1.\text{fb}_{A_2}(X) \Rightarrow A_2.\text{valid}(X))$$

Proof (in Coq).

Let relation $\text{hb}^r_{A_2}$ be $\text{?rfe} \cup (\text{ws} \rightarrow \text{fr}) \cup (\text{fr} \rightarrow \text{fr} \rightarrow \text{fr} \rightarrow \text{fr} \rightarrow \text{fr} \rightarrow)$. We prove that $\text{hb}^r_{A_2} \cup \text{fr} \cup \text{fr} \cup \text{ppo}$ is acyclic if and only if $R_2 = \text{hb}^r_{A_2}$. Then, to prove the theorem it suffices to prove that $\text{ghb}^r_{A_1}$ is cyclic whenever $R_2$ is.

Let us consider a cycle in $R_2$. Because $m_1 \rightarrow m_2 \rightarrow m_3$ implies $m_1 \rightarrow m_3$, we can safely assume that no $\text{fr} \rightarrow$ sequence ends by a $\text{rfi} \rightarrow$ step. Then, as no two $\rightarrow$ steps can follow one another, condition $A_1.\text{fb}_{A_2}(X)$ suffices to guarantee the inclusion $R_2 \subseteq (\text{ghb}^r_{A_1})^+$ and conclude.

4.2 Considering a weaker barriers guarantee

When two architectures $A_2$ and $A_1$ have the same policy $w.r.t.$ the store atomicity and store buffer relaxations, which we model by $\text{hrf} = \text{grf}$, there is no need for a barrier as powerful as above to restore $A_2$ from $A_1$: a barrier that only orders the events that surround it statically—that is, a non cumulative barrier, action we model by $\text{non-cumul}(X, \text{fenced}) \triangleq \text{fenced}$—is enough. Consider
the wfb predicate, which states that the barriers provided by $A_1$ maintain the pairs that are preserved in the program order on $A_2$ but not on $A_1$:

$$A_1. \text{wfb}_{A_2}(X) \triangleq \texttt{ppo}_{A_2} \subseteq \texttt{ab}_1$$

The same guarantee applies if $A_2$ hinders the store buffer relaxation by its preserved program order, i.e. when $\texttt{rfi} \subseteq \texttt{ppo}_2$—which is particular to $Sc$:

**Theorem 6 (Non cumulative barriers guarantee).**

$$\forall A_1, A_2, ((A_1 \leq A_2) \land (\texttt{ext}_1 = \texttt{ext}_2) \lor (\texttt{rfi} \subseteq \texttt{ppo}_2)) \Rightarrow$$

$$(\forall X, A_1. \text{valid}(X) \land A_1. \text{wfb}_{A_2}(X) \Rightarrow A_2. \text{valid}(X))$$

**Proof (in Coq).** As for Thm. 5 we need prove $R_2 = (\texttt{hb}_{\texttt{sc}} \subseteq \texttt{ghb}_{\texttt{ps}} \subseteq \texttt{ghb}_{\texttt{tso}})$. First, by hypothesis $\texttt{ext}_1 = \texttt{ext}_2$, we get $\texttt{hb}_{\texttt{sc}} \subseteq \texttt{ghb}_{\texttt{ps}}$. Second, we prove $\texttt{hb}_{\texttt{sc}} \subseteq \texttt{ghb}_{\texttt{tso}}$ by considering each step in $\texttt{ghb}_{\texttt{ps}}$. For a particular step $m_1 \texttt{ppo}_{\texttt{ps}} \Rightarrow m_2$, we have either $m_1 \texttt{ppo}_{\texttt{ps}} \Rightarrow m_1 \texttt{ghb}_{\texttt{ps}} \Rightarrow m_2$, or $m_1 \texttt{ppo}_{\texttt{ps}} \Rightarrow m_1 \texttt{ghb}_{\texttt{ps}} \Rightarrow m_2$. While for all $\texttt{rfi}$ steps, we conclude either by $\texttt{rfi} \subseteq \texttt{ghb}_{\texttt{ps}}$—when $\texttt{int}_{\texttt{ps}} = \texttt{int}_{\texttt{tso}} = \texttt{true}$, or by $\texttt{rfi} \subseteq \texttt{ppo}_{\texttt{ps}}$. \qed

**From Tso to Sc** As $\texttt{rfc}$ are global in both Tso and Sc, and Sc hinders the store buffering relaxation by its $\texttt{ppo}$ definition, it suffices by Thm. 6 to fence all pairs in $\texttt{ppo}_{\texttt{ps}} \setminus \texttt{ppo}_{\texttt{tso}} = \texttt{WR}$ (where $\texttt{WR} \triangleq ((\texttt{W} \times \texttt{R}) \cap \texttt{po})$) to restore Sc from Tso:

**Corollary 3 (Barriers restoring Sc from Tso).**

$$\forall X, (Tso. \text{valid}(X) \land \text{non-cumul}(X, \texttt{WR}) \subseteq \texttt{ab}_{\texttt{tso}}) \Rightarrow Sc. \text{valid}(X))$$

**From Pso to Tso** We comment here on the two definitions of PSO given in Sparc documentations [3]. We adapt the definition of [3, V8] to our framework:

$$\texttt{Pso}^{\prime}. \text{Arch} \triangleq (\lambda X. \texttt{RM}, \texttt{rfc}, \lambda X. \emptyset)$$

Tso and Pso agree on both the store atomicity and the store buffering relaxations, which allows us to apply Thm. 6; $\texttt{Pso}^{\prime}$ is restored from $\texttt{Pso}$ by inserting non cumulative barriers between all $\texttt{ppo}_{\texttt{ps}} \setminus \texttt{ppo}_{\texttt{pso}} = \texttt{WW}$ pairs. Indeed, TSO is obtained from PSO by adding StoreStore barriers after each write [3, V9].

### 5 diy

#### 5.1 Cycles generation

A type edge represents the arrows of an execution witness. It can be in $\texttt{rf}, \texttt{ws}$, $\texttt{fr}$: their source and target share the same location and have the appropriate
directions (e.g. \( rf \rightarrow \) has a write as source and a read as target). Similarly to \( rfi \rightarrow \) and \( rfe \rightarrow \), we define \( wsi \rightarrow \) and \( wse \rightarrow \), \( fri \rightarrow \) and \( fre \rightarrow \). An edge can also be in \( po \rightarrow \), \( dp \rightarrow \) or \( fenced \rightarrow \), where we need to specify the source and target directions (except for \( dp \rightarrow \) whose source is a read by definition), and if they share the same location.

We specify a concrete syntax for the edges: thus \( Rfe \) represents a \( rfe \rightarrow \) arrow, \( Fre \) a \( fre \rightarrow \) arrow, \( PosRW \) specifies a \( po \rightarrow \) arrow with a read (R) as source, a write (W) as target, knowing these events share the same location (s), and \( DpdR \) a \( dp \rightarrow \) arrow with a read (R) as target, with different (d) source and target locations.

We define two relations \( relax \rightarrow \) and \( safe \rightarrow \), whose default values are respectively \( po \cup rfi \rightarrow \) and \( ws \cup fri \cup dp \cup fenced \rightarrow \). They can be modified by an input of the user: e.g. on a x86 machine, which has a Tso model [2], \( rfe \rightarrow \) can be specified as safe.

We show a cycle in \( hb-seq \rightarrow \cup po \rightarrow \) is a cycle in \( (relax) + \rightarrow \) and \( (safe) + \rightarrow \). Thus, a test violating \( Sc \) corresponds to a cycle in this second relation. But we do not generate cycles with more than one relaxation: if a test exhibits a relaxation, we cannot decide which one is responsible. A cycle \( Rfe; DpsW; Fre \) determines whether \( rfe \rightarrow \) is global or not since \( dp \rightarrow \) and \( fr \rightarrow \) are safe, whereas \( Rfe; DpsW; Rfi; Fre \) is inconclusive as it exercises both \( rfe \rightarrow \) and \( rfi \rightarrow \). We also generate the cycles exhibiting no relaxation (in \( (safe) + \)), to test our assumption \( safe \rightarrow \) is global.

## 5.2 Code generation

ddy interprets a sequence of edges as a cycle from which it computes a litmus test or fails. One of its execution witnesses includes a cycle compliant with the input edges sequence. The final condition is a conjunction of equalities over the final state of registers and memory locations characterising this cycle.

Test generation performs the following successive steps.

1. We map the edges sequence to a circular double-linked list, whose cells represent memory events, with direction, location, and value fields. An additional field records the edge starting from the event. This list represents the input cycle and appear in at least one of the execution witnesses of the produced test.

2. A linear scan sets the events directions, by comparing each target direction with the following source direction. When equal, the in-between cell direction is set to the common value; otherwise (e.g. \( Rfe; Rfe \)), the generation fails.

3. If no event \( e \) has an incoming edge specifying a location change, generation fails. Otherwise, a linear scan starting from \( e \) sets the locations. We reject the cycles where \( e \) and its predecessor have different locations (e.g. \( Rfe; PodRW \)).

4. We cut the input cycle into maximal sequences of events with the same location, each being scanned \( w.r.t. \) the cycle order: we give the value 1 to the first write in this sequence, 2 to the second one, \( etc \). Thus the values also reflect the write serialisation order for the specified location.
5. **Significant reads** are the sources of $\text{fr} \rightarrow$ edges and the targets of $\text{rf} \rightarrow$ edges. They are associated with the write on the other side of the edge. In the $\text{fr} \rightarrow$ case, the value of the read is the one of its associated write. In the $\text{rf} \rightarrow$ case, the value of the read is the value of the predecessor of its associated write in $\text{ws} \rightarrow$, i.e. by construction the value of its associated write minus 1. Non significant reads do not appear in the test condition.

6. We cut the cycle into maximal sequences of events from the same processor, each being scanned, generating load instructions to (resp. stores from) fresh registers for reads (resp. writes). We insert code implementing a dependency in front of events targeting $\text{dp} \rightarrow$ and the appropriate barrier instruction for events targeting $\text{fenced} \rightarrow$ edges. We build initial state at this step: stores and loads take their addresses from fresh registers whose content (a memory location) is defined in the initial state. Part of the final condition is also built: for any significant read with value $v$ resulting in a load instruction to register $r$, we add the equality $r = v$.

7. We complete the final condition to characterise write serialisations. The write serialisation for a given location $x$ is defined by the sequence of values $0$ (initial value of $x$), $\ldots$, $n$, where $n$ is the last value allocated for location $x$ at step 4. If $n$ is 0 or 1 then no addition to the final condition needs to be performed, because the write serialisation is either a singleton or a pair. If $n$ is 2, we add the equality $x = 2$. Otherwise ($n > 2$), we add an observer to the program, i.e. we add a thread performing $n$ loads from $x$ to $r_1$, $\ldots$, $r_n$ and add the equalities $r_1 = 1 \land \ldots \land r_n = n$ to the final condition.

References