Robustness between Weak Memory Models

Soham Chakraborty
EEMCS, TU Delft
Email: s.s.chakraborty@tudelft.nl

***Abstract*—**

Robustness of a concurrent program ensures that its behaviors on a weak concurrency model are indistinguishable from those on a stronger model. Enforcing robustness is particularly useful when porting or migrating applications between architectures. Existing tools mostly focus on ensuring sequential consistency (SC) robustness which is a stronger condition and may result in unnecessary fences.

To address this gap, we analyze and enforce robustness between weak memory models, more specifically for two mainstream architectures: x86 and ARM (versions 7 and 8). We identify robustness conditions and develop analysis techniques that facilitate porting an application between these architectures. To the best of our knowledge, this is the first approach that addresses robustness between the hardware weak memory models.

We implement our robustness checking and enforcement procedure as a compiler pass in LLVM and experiment on a number of standard concurrent benchmarks. In almost all cases, our procedure terminates instantaneously and insert significantly less fences than the naive schemes that enforce SC-robustness.

I. Introduction

Robustness analysis checks whether a program running on a weak memory consistency model demonstrates only the behaviors that are allowed by a stronger model. Robust programs can therefore be seamlessly migrated from one model to another as far as their concurrent behaviors are concerned. If a program is not robust, we can insert fences to enforce robustness.

Robustness analysis is especially beneficial in porting applications [1, 2] where it is crucial to preserve the observable behaviors of a running application. For instance, consider the porting of an application written for x86 to ARM. Since the x86 model is stronger than the ARM models (x86 exhibits less behavior), x86-robustness abstracts the underlying ARM machine specification to an outside observer. Consider the following programs where initially $X = Y = 0$.

$$\begin{array}{l} X = 1; \\ a = Y; \end{array} \left\| \begin{array}{l} Y = 1; \\ b = X; \end{array} \right. \quad \textsf{(SB)} \qquad \Bigg| \qquad \begin{array}{l} a = X; \\ Y = 1; \end{array} \left\| \begin{array}{l} b = Y; \\ X = 1; \end{array} \right. \quad \textsf{(LB)}$$

Both x86 and ARM allow same set of concurrent executions in the SB program and hence indistinguishable on x86 and ARM. Therefore SB can be ported seamlessly between these architectures. Now consider the porting of the LB program from x86 to ARM. x86 disallows $a = b = 1$ but ARM allows the outcome. Hence the LB program in ARM is not x86-robust. To enforce x86-robustness we insert fences in both threads and restrict the $a = b = 1$ outcome.

Checking and enforcing robustness to a stronger but non-SC model from a weaker model can play a key role in migrating programs between architectures having weak concurrency models. Existing SC-robustness approaches may not provide an optimal solution as they check a stronger constraint and hence may introduce additional fences. For example, if we use an SC-robustness checker for SB, it identifies that the $a = b = 0$ outcome is allowed on ARM but disallowed in SC. Hence the analyzer inserts two full fences (`DMB` in ARMv7 and `DMBFULL` in ARMv8) between the memory accesses in both threads which are unnecessary in this case.

To address this scenario we propose robustness analysis and enforcement between weak memory models of two mainstream architectures: x86 and ARM (version 8 and 7). As ARMv8 is a stronger model than ARMv7, we also study ARMv8-robustness for ARMv7 to enable application porting between these ARM models. We also check SC-robustness in x86, ARMv8, ARMv7 and restrict relaxed memory behaviors.

In this paper we propose M-K robustness where M is a stronger model than K and M can also be a non-SC model unlike existing approaches in [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]. We propose the M-K robustness conditions in §III and prove their correctness [15]. Our proposed M-K robustness conditions ensure that if a K-consistent execution satisfies the M-K condition then the execution is also M-consistent. We check if certain memory access pairs are appropriately ordered in a K-consistent execution so that the execution shows no weaker behavior. Otherwise we insert fences to enforce order and restrict the weaker behaviors. However, as fences are costly, we investigate if it is possible to weaken the robustness constraints for the memory access pairs which are on same-location or are ordered by dependencies. We observe that these relations suffice in x86 and ARMv8, but the results in ARMv7 are counter-intuitive.

- We note that dependency based ordering *preserved-program-order* (ppo) is not strong enough to ensure robustness in ARMv7. Consider the following ARMv7 program.

  $$\begin{array}{l} a = T; \\ X = a; \end{array} \left\| X = 2; \right\| \begin{array}{l} b = X; \\ Y = b; \end{array} \left\| \begin{array}{l} c = Y; \\ Z = c; \end{array} \right\| Z = 1; \left\| \begin{array}{l} d = Z; \\ T = d; \end{array} \right. \quad \textsf{(WP)}$$

  The execution in Fig. 4 exhibits non-SC behavior though all the memory access pairs result in ppo relations due to data dependencies. Even an intermediate full fence in one of these threads cannot restrict the relaxed behavior.
- We evaluate the role of same-location program-order relation in defining robustness conditions. On ARMv7, same-location read-write access pair is unordered (see ARM-Weak [16] example in Fig. 2). Yet if all *external-program-orders* (see §III) are on same-location or have intermediate fences then the program exhibits only SC behavior.

In §IV we propose static analyses to check if a program is M-K robust based on the respective conditions. Otherwise we insert fences to enforce robustness. These analyses are computed in polynomial time as shown in § IV-C unlike the robustness checkers which explore program executions and are of significantly higher computational complexity.

The robustness checking procedures analyze the programs with thread functions. In these programs each thread function may result in any number of concurrent threads in an execution. Thus our analysis is parameterized by the thread functions and the analyses are applicable to all the programs having same thread functions.

We have implemented the analyses procedures in a tool called *Fency* based on LLVM [17] and have evaluated on several well known concurrent programs [8, 14]. We compare the SC-x86 robustness analysis of *Fency* to existing SC-TSO robustness results of Trencher [8] that explore program executions by model checkers. Yet, *Fency* is quite precise and matches Trencher in most of the programs. Moreover, *Fency* does not use external model checkers or SAT/SMT solvers and therefore is significantly fast in most of the cases.

We also compare *Fency* to a *naive* fence insertion scheme that do not use robustness analysis. *Fency* inserts significantly fewer fences than the naive scheme in several benchmarks. Moreover, empirical evaluations show that if a model W is weaker than M then ensuring W-K robustness often requires fewer fences than ensuring M-K robustness. Thus precise robustness analysis is indeed beneficial for many cases instead of using SC-robustness checkers.

Outline and Contributions. §II reviews the concurrency models. §III proposes the M-K robustness conditions. §IV explains our approach to check and enforce robustness. §V examine the experimental results. §VI discusses the related work and we conclude in §VII. The proofs and additional details are in the supplementary material [15].

II. Concurrency Models

In this section we review SC, x86, ARMv8, and ARMv7 concurrency. For all models we follow a common syntax.

$$
\begin{aligned}
E &::= r \,|\, v \,|\, E+E \,|\, E*E \,|\, E \leq E \,|\, \cdots \\
C &::= \mathsf{skip} \,|\, C;C \,|\, t=E \,|\, t=X \,|\, X=E \,|\, \texttt{RMW}(X,E,E) \\
&\quad\;\; |\, \mathsf{Fence} \,|\, \texttt{RMW}(X,E) \,|\, \mathbf{br}\ \mathit{label} \,|\, \mathbf{br}\ \mathit{label}\ \mathit{label} \,|\, \cdots \\
P &::= X=v;\cdots X=v;\{C \parallel \cdots \parallel C\}
\end{aligned}
$$

An expression E results from thread-local temporary (t), value (v), and arithmetic operations (E). Command $t = X$ returns the value of a shared memory location X to a thread-local register r and $X = E$ writes the evaluation of expression E to X. The $\texttt{RMW}(X, E_r, E_w)$ atomically compares the values of X and E_r; if equal then X is written to the value of E_w and set r. If the value of X is not equal to the value of E_r then the RMW fails. Command $\texttt{RMW}(X, E_r)$ atomically updates the value of X with the value of E_r and returns the value of X to r. A failed RMW performs only read access. A fence orders certain memory accesses. We use conditional and unconditional branches for program's control flow. Finally, a program consists of a set of initialization writes followed by a parallel composition of thread commands. Unless otherwise mentioned, the initializations set all memory locations to zero.

A. Program Semantics and Execution Graphs

We follow the axiomatic models for all architectures [18, 19, 20, 21, 22, 23, 24, 25, 26]. In these axiomatic models a program's semantics is defined by a set of consistent executions. An execution consists of a set of events and relations.

Event. An event $\langle \mathsf{id}, \mathsf{tid}, \mathsf{lab}\rangle$ consists of unique identifier id, thread identifier $\mathsf{tid} \in \mathsf{N}$, and a label lab based on the respective executed memory or fence access. A label is of the form $\langle \mathsf{op}, \mathsf{loc}, \mathsf{val}\rangle$ where op, loc, and val are operation type, location, and read or written value.

Preliminaries. Given a binary relation P on events, $\mathsf{dom}(P)$ and $\mathsf{codom}(P)$ are its domain and its range. P^{-1}, $P^{?}$, P^{+}, and P^{*} are inverse, reflexive, transitive, and reflexive-transitive closures of P respectively. P_ℓ denotes P related event pairs on same locations i.e. $P_\ell \triangleq \{(e, e') \in P \mid e.\mathsf{loc} = e'.\mathsf{loc}\}$ and $P_{\neq\ell} \triangleq P \setminus P_\ell$ denote the P related event pairs on different locations. $\mathsf{imm}(P)$ defines the immediate P relation, i.e. $\mathsf{imm}(P) \triangleq \exists a, b.\ P(a, b) \wedge \nexists c.\ P(a, c) \wedge R(c, b)$. $P\,;S$ is the relational composition of the binary relations P and S. Finally, $[A]$ is an identity relation on a set A.

R, W, and F are the set of read, write, and fence events. The events are related by primitive relations: strict partial order program-order (po) captures the syntactic order among the events, reads-from (rf) relates a write event to a read event that justifies its read value, and strict total order coherence-order (co) relates same-location writes.

Execution. An execution is of the form $\mathsf{X} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{co}\rangle$ where $\mathsf{X.E}$ is the set of events in X. The set of po, rf, and co relations between the events in $\mathsf{X.E}$ are $\mathsf{X.po}$, $\mathsf{X.rf}$, and $\mathsf{X.co}$. Execution X is *well-formed* if $\mathsf{X.po}$ is total in each thread and every read reads-from some write, i.e. $\mathsf{X.R} \subseteq \mathsf{codom}(\mathsf{X.rf})$.

We derive a number of relations from these primitive relations. Relation $\mathsf{rmw} \subseteq \mathsf{imm}(\mathsf{po}) \cap ([\mathsf{R}] \times [\mathsf{W}])_\ell$ denotes atomic update where a read has an immediate po-successor write on the same location. The non-rmw read and write events are load (Ld) and store (St) events.

$$\mathsf{Ld} \triangleq \mathsf{R} \setminus \mathsf{dom}(\mathsf{rmw}) \qquad\qquad \mathsf{St} \triangleq \mathsf{W} \setminus \mathsf{codom}(\mathsf{rmw})$$

A successful RMW generates an rmw and a failed RMW generates a Ld event. We use $a \cdot b \triangleq [\{a\}]; \mathsf{imm}(\mathsf{po}); [\{b\}]$ to denote that a and b are immediate po related events.

Relation WR denotes a write-read event pair on different locations that does not have any intermediate rmw.

$$\mathsf{WR} \triangleq ([\mathsf{W}]; \mathsf{po}_{\neq\ell}; [\mathsf{R}]) \setminus (\mathsf{po}; \mathsf{rmw}; \mathsf{po})$$

The from-read (fr) relation relates a pair of same-location read and write events r and w where r reads-from a write w' which is co-before w, that is, $\mathsf{fr} \triangleq \mathsf{rf}^{-1}; \mathsf{co}$. For example, in Fig. 1a the $\mathsf{R}(X, 0)$ and $\mathsf{W}(X, 1)$ events are in fr relation.

We categorize the relations as external and internal based on whether the events are also in po relation. Considering rf, co, and fr relations rfi, coi, fri and rfe, coe, fre denote the internal and external relations respectively.

$$\mathsf{rfe} \triangleq \mathsf{rf} \setminus \mathsf{po} \qquad \mathsf{coe} \triangleq \mathsf{co} \setminus \mathsf{po} \qquad \mathsf{fre} \triangleq \mathsf{fr} \setminus \mathsf{po}$$
$$\mathsf{rfi} \triangleq \mathsf{rf} \cap \mathsf{po} \qquad \mathsf{coi} \triangleq \mathsf{co} \cap \mathsf{po} \qquad \mathsf{fri} \triangleq \mathsf{fr} \cap \mathsf{po}$$

For example, the rf and fr edges in Fig. 1a edges are rfe and fre edges respectively. Based on the rfe, coe, and fre we define *extended-coherence-order* (eco) on same location events: $\mathsf{eco} \triangleq (\mathsf{rfe} \cup \mathsf{coe} \cup \mathsf{fre})^+$.

Consistency Axioms. An axiomatic model is defined by a set of axioms. An execution is consistent in a model if it satisfies all its axioms. An axiom violation can be captured by a cycle on the respective execution graph.

B. Formal Models

Now we move to the axiomatic definitions based on various relations. We elide some definitions here due to space constraint which we discuss in the technical appendix [15].

In these models a store access writes value v on location x and generates an event with label $\mathsf{W}(x, v)$. A load access reads value v from x and generates an event with label $\mathsf{R}(x, v)$. A successful RMW on x reads value v' and writes value v to generate a pair of $\mathsf{R}(x, v')$ and $\mathsf{W}(x, v)$ events that are in rmw relation. A failed RMW generates an $\mathsf{R}(x, v')$ event. The full fences in x86, ARMv8, and ARMv7 are MFENCE, DMBFULL, and DMB respectively. A full fence generate an event with label F. ARM architectures also provides ISB fence to order a pair of reads. In ARMv7 an ISB access along with control (cmp) and jump (bc) instructions generate cmp; bc; ISB that result in $\mathsf{ctrl}_{\mathtt{ISB}}$ between a pair of read events in an execution [19]. In ARMv8 an ISB generates an ISB event.

ARMv8 Specific Accesses. In addition, ARMv8 has synchronizing memory accesses such as release write, acquire read, and acquirePC load which are denoted by events with label $\mathsf{L}(x, v)$, $\mathsf{A}(x, v)$, and $\mathsf{Q}(x, v)$. ARMv8 also provide DMBLD and DMBST fences that generate F_{LD}, and F_{ST} events. Finally, $\mathsf{L} \subseteq \mathsf{W}$, $\mathsf{A} \subseteq \mathsf{R}$, $\mathsf{Q} \subseteq \mathsf{Ld} \subseteq \mathsf{R}$, and F, F_{LD}, F_{ST} are the set of release, acquire, acquirePC, and full, load, store fence events.

All these models satisfy coherence and atomicity properties.

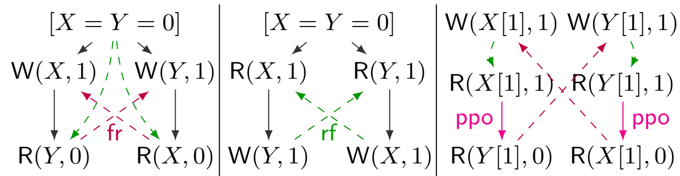
Coherence. The property enforces SC per location i.e. in an execution all accesses on same memory locations are totally ordered. A complete execution graph X satisfies coherence if $\mathsf{X.po}_\ell \cup \mathsf{X.rf} \cup \mathsf{X.co} \cup \mathsf{X.fr}$ is acyclic.

Atomicity. An execution X violates atomicity if there is an intermediate write on same location between rmw related read and write events. In that case $\mathsf{X.fre}(r, w)$ and $\mathsf{X.coe}(w', w)$ hold where r and w are X.rmw-related events and w' is another write on the same location as r and w.

SC. An well-formed execution X is SC when:

- $(\mathsf{X.po} \cup \mathsf{X.rf} \cup \mathsf{X.fr} \cup \mathsf{X.co})$ is acyclic (SC)
- $\mathsf{X.rmw} \cap (\mathsf{X.fre}; \mathsf{X.coe}) = \emptyset$ (atomicity)

The executions in Fig. 1 are inconsistent in SC. For example, the SB execution has $\mathsf{po} \cup \mathsf{fr}$ cycle. Note that coherence constraint is included in (SC) axiom as $\mathsf{po}_\ell \subseteq \mathsf{po}$ holds and therefore if $(\mathsf{X.po} \cup \mathsf{X.rf} \cup \mathsf{X.fr} \cup \mathsf{X.co})$ is acyclic then $(\mathsf{X.po}_\ell \cup \mathsf{X.rf} \cup \mathsf{X.fr} \cup \mathsf{X.co})$ is also acyclic.



(a) SB (b) LB (c) IRIW

Fig. 1: Distingushing executions: SB execution is disallowed in SC but allowed in x86 and ARM. SC and x86 disallow LB execution but ARM models allow it. IRIW execution is disallowed in SC, x86, ARMv8, but allowed in ARMv7.

x86. Relation x86-preserved-program-order (xppo) orders read-read, read-write, write-write access pairs. Relation implied signifies that an intermediate rmw or F acts as a full fence. Based on these relations x86 defines x86-happens-before (xhb). Finally, x86 defines its consistency constraints for a well-formed execution.

- $\mathsf{X.po}_\ell \cup \mathsf{X.rf} \cup \mathsf{X.fr} \cup \mathsf{X.co}$ is acyclic (sc-per-loc)
- $\mathsf{X.rmw} \cap (\mathsf{X.fre}; \mathsf{X.coe}) = \emptyset$ (atomicity)
- X.xhb is acyclic where (GHB)
  - $\mathsf{xhb} \triangleq \mathsf{xppo} \cup \mathsf{implied} \cup \mathsf{rfe} \cup \mathsf{fr} \cup \mathsf{co}$ where
  - $\mathsf{xppo} \triangleq ((\mathsf{W} \times \mathsf{W}) \cup (\mathsf{R} \times \mathsf{W}) \cup (\mathsf{R} \times \mathsf{R})) \cap \mathsf{po}$
  - $\mathsf{implied} \triangleq \mathsf{po}; [\mathsf{dom}(\mathsf{rmw}) \cup \mathsf{F}] \cup [\mathsf{codom}(\mathsf{rmw}) \cup \mathsf{F}]; \mathsf{po}$

x86 satisfies coherence and atomicity by (sc-per-loc) and (atomicity) axioms respectively. Axiom (GHB) ensures a global order based on xhb relation. The model allows Fig. 1a but disallows the executions in Figs. 1b and 1c.

ARMv8. In ARMv8 relation observed-by ($\mathsf{obs} \subseteq \mathsf{eco}$) relates same-location external events. Relation atomic-ordered-by ($\mathsf{aob} \subseteq \mathsf{po}_\ell$) orders events based on rmw and acquire or acquirePC events. The dependency-ordered-before (dob) captures dependency based ordering between events e.g. $\mathsf{data} \cup \mathsf{addr} \subseteq \mathsf{dob}$. Relation barrier-ordered-by (bob) orders events by fences and stronger memory accesses as follows.

$$\mathsf{bob} \triangleq \mathsf{po}; [\mathsf{F}]; \mathsf{po} \cup [\mathsf{R}]; \mathsf{po}; [\mathsf{F}_{\mathrm{LD}}]; \mathsf{po} \cup [\mathsf{W}]; \mathsf{po}; [\mathsf{F}_{\mathrm{ST}}]; \mathsf{po}; [\mathsf{W}]$$
$$\cup [\mathsf{L}]; \mathsf{po}; [\mathsf{A}] \cup \mathsf{po}; [\mathsf{L}] \cup [\mathsf{A} \cup \mathsf{Q}]; \mathsf{po} \cup \mathsf{po}; [\mathsf{L}]; \mathsf{coi}$$

A full fence orders all accesses, a load fence orders a read with its successors, and a store fence orders a pair of writes. A release access is ordered with its predecessors and an acquire or acquirePC is ordered with its successors. Release and acquire accesses are ordered. Finally, (a, b) is ordered if b is a write and there is an intermediate release store on the same-location as b. Based on these relations ARMv8 defines *Ordered-before* (ob) order: $\mathsf{ob} \triangleq (\mathsf{obs} \cup \mathsf{dob} \cup \mathsf{aob} \cup \mathsf{bob})^+$. A well-formed ARMv8 execution X is consistent when:

- $\mathsf{X.po}_\ell \cup \mathsf{X.rf} \cup \mathsf{X.co} \cup \mathsf{X.fr}$ is acyclic (internal)
- $\mathsf{X.rmw} \cap (\mathsf{X.fre}; \mathsf{X.coe}) = \emptyset$ (atomicity)
- X.ob is irreflexive (external)

These axioms allow the executions in Figs. 1a and 1b but disallows the execution in Fig. 1c by the (external) axiom.

$$a = X;\; X = 1; \;\Big\|\; Y = X; \;\Big\|\; X = Y;$$

(ARM-Weak)

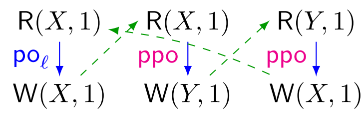


Fig. 2: Outcome $a = 1$ is allowed in ARMv7.

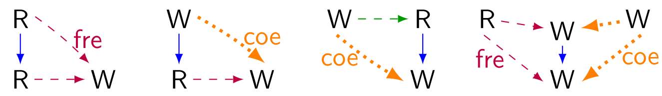


Fig. 3: Coherence ensures $\mathsf{eco};\mathsf{epo}_\ell \cup \mathsf{epo}_\ell;\mathsf{eco} \subseteq \mathsf{eco}$.

ARMv7. ARMv7 orders memory accesses in a thread by *preserved-program-order* (ppo) based on dependencies or $\mathsf{fence} \subseteq \mathsf{po};[\mathsf{F}];\mathsf{po}$ relation. ARMv7 also defines happens-before (ahb) and propagation ($\mathsf{prop} \subseteq R_1;\mathsf{fence};R_2$) relations that can order events across threads. Finally a well-formed ARMv7 execution X is consistent when:

- $(\mathsf{X.po}_\ell \cup \mathsf{X.rf} \cup \mathsf{X.fr} \cup \mathsf{X.co})$ is acyclic. (sc-per-loc)
- $\mathsf{X.rmw} \cap (\mathsf{X.fre};\mathsf{X.coe}) = \emptyset$ (atomicity)
- $\mathsf{X.fre};\mathsf{X.prop};\mathsf{X.ahb}^*$ is irreflexive. (observation)
- $(\mathsf{X.co} \cup \mathsf{X.prop})$ is acyclic. (propagation)
- $\mathsf{X.ahb}$ is acyclic. (no-thin-air)

Axiom (observation) constrains the set of writes from which reads may read-from; if a write w is in $\mathsf{prop};\mathsf{ahb}^*$ relation with a same-location read r then r does not read from w' which is co-before w. (propagation) ensures that prop does not contradict co and (no-thin-air) constrain causality cycle.

ARMv7 allows the executions in Fig. 1 including IRIW with $a = c = 1, b = d = 0$ outcome in the following program.

$$X[1] = 1; \;\Big\|\; \begin{array}{l} a = X[1]; \\ b = Y[a]; \end{array} \;\Big\|\; \begin{array}{l} c = Y[1]; \\ d = X[c]; \end{array} \;\Big\|\; Y[1] = 1; \quad \text{(IRIW)}$$

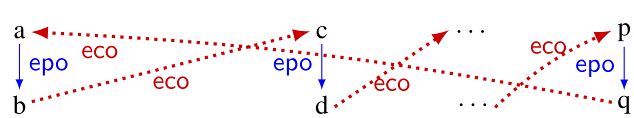
In addition read-write accesses on same-location can be unordered in ARMv7. As a result, the ARM-Weak program in Fig. 2 has an execution with $a = 1$ outcome.

III. Robustness Analysis and Enforcement

In this section we first define M-K robustness and then propose the M-K robustness conditions.

Definition 1. *A program is M-K robust if all its K-consistent executions are also M-consistent.*

Suppose a K-consistent execution X violates an axiom from M-consistency. The violation results in a cycle in X. If the cycle contains no po edge then it is formed by rfe, fre, and coe edges on same location events. The cycle also violates coherence. This is not possible as execution X is K-consistent and all K models we are considering satisfy coherence. So the cycle consists of a set of po-edges along with the eco edges between them. We define these po edges as *external-program-order* (epo) i.e. $\mathsf{epo} \triangleq \mathsf{po} \cap (\mathsf{codom}(\mathsf{eco}) \times \mathsf{dom}(\mathsf{eco}))$.



Thus we represent an axiom violation as a $(\mathsf{epo};\mathsf{eco})^+$ cycle where all the epo edges on the cycle are not sufficiently ordered. To enforce order we insert fences to strengthen these epo edges and restrict a cycle to enforce M-K robustness.

Theorem 1. *A program $\mathbb{P}$ is M-K robust if in all its K-consistent execution X, $\mathsf{X.epo} \subseteq \mathsf{X}.R$ holds where R is defined as M-K condition as follows.*

(SC-x86)	$\mathsf{xppo} \cup \mathsf{po}_\ell \cup \mathsf{implied};\mathsf{po}^?$
(SC-ARMv8)	$\mathsf{po}_\ell \cup (\mathsf{aob} \cup \mathsf{dob} \cup \mathsf{bob})^+$
(x86-ARMv8)	$\mathsf{po}_\ell \cup (\mathsf{aob} \cup \mathsf{bob} \cup \mathsf{dob})^+ \cup \mathsf{WR}$
(SC-ARMv7)	$\mathsf{po}_\ell \cup \mathsf{fence}$
(x86-ARMv7)	$\mathsf{po}_\ell \cup \mathsf{fence} \cup \mathsf{WR}$
(ARMv8-ARMv7)	$\mathsf{po}_\ell \cup [\mathsf{W}];\mathsf{po} \cup \mathsf{fence}$

Next, we explain the M-K conditions for the concurrency models. The correctness proofs for these robustness conditions are in the technical appendix [15].

A. Robustness of x86 Programs

From the SC-x86 condition in Theorem 1, relation xppo orders read-read, read-write, and write-write pairs. So if an x86 execution violates SC-x86 robustness then it contains a $(\mathsf{epo};\mathsf{eco})^+$ cycle with one or multiple epo edges that are in WR relation. If it is on same location then there is an alternative $(\mathsf{eco};\mathsf{epo})^+$ cycle as shown in Fig. 3 that also denote the violation. The $\mathsf{implied};\mathsf{po}^?$ relation can order a write-read pair by intermediate rmw or F.

Consider the SB execution from Fig. 1a in x86. The epo edges do not satisfy SC-x86 condition and the execution is non-SC. If we insert fences between the store-load pairs in each thread then the program exhibits only SC behaviors.

B. Robustness of ARMv8 Programs

SC-ARMv8 Robustness. Suppose an ARMv8 execution contains a $(\mathsf{epo};\mathsf{eco})^+$ cycle that violates SC-ARMv8 robustness. If an epo_ℓ edge is on the cycle then as shown in Fig. 3 there is an alternative $(\mathsf{epo};\mathsf{eco})^+$ cycle without the edge.

Now consider an $(\mathsf{epo};\mathsf{eco})^+$ cycle where each epo on the cycle is in $(\mathsf{aob} \cup \mathsf{bob} \cup \mathsf{dob})^+$ relation. In that case $((\mathsf{aob} \cup \mathsf{bob} \cup \mathsf{dob})^+;\mathsf{eco})^+$ cycle implies an ob cycle which is not possible as an ARMv8 consistent execution satisfies (external). The epo edges in SB and LB executions in Fig. 1 do not satisfy the SC-ARMv8 condition. The executions are allowed in ARMv8 but not in SC.

x86-ARMv8 Robustness. The x86-ARMv8 robustness condition orders all epo relations except WR pairs as WR is also unordered in x86. Hence an ARMv8 execution exhibits only x86 behavior if the x86-ARMv8 condition holds. Consider the SB execution from Fig. 1a in ARMv8; both the epo edges are also in WR and the execution is x86 consistent.

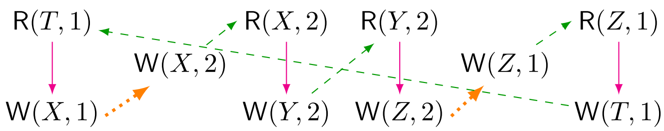


Fig. 4: ARMv7 allows the execution of the WP program.

```
SB2(p){
1. if (a)  X = 1;
2. else Y = 1;
3. a = Y;
4. b = X; }
```

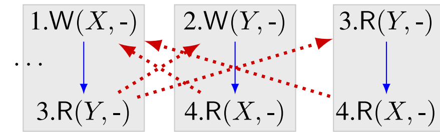


Fig. 5: Subgraph of SB2 MPG with potential epo and eco edges. SB2(true) || SB2(false) violates SC-x86 robustness.

C. Robustness of ARMv7 Programs

SC-ARMv7 Robustness. The ARMv7 model uses po_ℓ and fence relations to order epo edges for SC-ARMv7 robustness.

The ppo and po_ℓ do not guarantee SC-ARMv7 robustness as shown in the execution in Fig. 2. If we insert fences in the second and third threads the execution is disallowed in ARMv7 and the resulting program is SC-ARMv7 robust.

Moreover, ppo relations in all epo edges do not ensure SC behavior in an execution. For instance, the WP program execution in Fig. 4 is non-SC even though the epo edges are ppo-ordered. Note that, even if we insert an intermediate DMB in one of the threads the cycle is still possible in ARMv7.

x86-ARMv7 Robustness. To ensure x86-robustness, ARMv7 orders all epo relations except write-read pairs. Consider the SB program execution in Fig. 1a where the epo edges are WR pairs and the execution is consistent in both ARMv7 and x86.

ARMv8-ARMv7 Robustness. ARMv8-ARMv7 robustness requires to order all $\mathsf{epo}_{\neq\ell}$ relations except write-read and write-write pairs. In this case also ppo relation cannot order $\mathsf{epo}_{\neq\ell}$ edges. Hence the cycle in the ARMv7 execution in Fig. 4 is disallowed in ARMv8 as it is an ob cycle.

IV. Checking and Enforcing Robustness

In this section we lift the semantic notion of M-K robustness to the program syntax and propose static analyses to check and enforce robustness in the following steps.

1) *Identify program components which may run concurrently.* We consider fork-join parallelism and identify the thread functions where each function may create multiple threads.
2) *Memory-access pair graph construction.* We identify the memory accesses in thread functions and construct a memory-access pair graph (MPG) that captures the potential epo and eco edges in the executions.
3) *Checking robustness.* If an MPG contains a cycle then we check whether each access pair on the cycle is ordered. If so then all K-consistent execution of the program preserve M-K robustness condition and as a result all K consistent executions of these programs are also M consistent.
4) *Enforcing robustness.* If the memory access pairs on the cycle are not ordered we insert appropriate fences between the memory access pairs. These fences disallow these cycle in the executions in the K consistency model and in turn enforce M-K robustness.

A. MPG Construction

Let $\{f_1, f_2, \ldots, f_n\}$ be the set of thread functions in a program that may run in parallel. Let $\mathcal{C} = \langle \mathcal{V}, \mathcal{E} \rangle$ be a control flow graph (CFG) of a thread function where $\mathcal{C}.\mathcal{V}$ are the instruction nodes and $\mathcal{C}.\mathcal{E}$ are the set of control flow edges. We analyze the thread functions' CFGs to construct an MPG.

Helper Definitions. We define following helper conditions.

- $\mathsf{CFG}(f)$ returns the control-flow-graph of a function f.
- $\mathsf{mayAA}(i, j)$ checks if i and j may access same location.
- $\mathsf{ac}(\mathcal{C}, A)$ returns the primitives in $\mathcal{C}$ which create A events or rmw relations i.e. $\mathsf{ac}(\mathcal{C}, A) \triangleq \{i \mid [\![i]\!] \in A\}$. In this case $\mathsf{ac}(\mathcal{C}, \mathsf{rmw})$ returns the accesses that create RMW primitives.
- $\mathcal{P}(\mathcal{C}, i, j)$ checks if there is a path from i to j on the control flow graph $\mathcal{C}$ i.e. $\mathcal{P}(\mathcal{C}, i, j) \triangleq (i, j) \in [\mathcal{C}.\mathcal{V}]; \mathcal{C}.\mathcal{E}^+; [\mathcal{C}.\mathcal{V}]$.
- $\mathsf{MM}(\mathcal{C})$ returns the set of memory access pairs in a control flow graph $\mathcal{C}$ where the second access is reachable from the first access. These pairs depict the potential epo edges i.e. $\mathsf{MM}(\mathcal{C}) \triangleq \{(i, j) \mid i, j \in \mathsf{ac}(\mathcal{C}, \mathsf{W} \cup \mathsf{R}) \wedge \mathcal{P}(\mathcal{C}, i, j)\}$.

Definition 2. *An MPG is of the form $\mathbb{G} = \langle \mathbb{V}, \mathbb{E} \rangle$ where $\mathbb{G}.\mathbb{V}$ is the set of shared memory access pairs and $\mathbb{G}.\mathbb{E}$ denote the set of edges between the nodes. An edge from $(a, b) \in \mathbb{G}.\mathbb{V}$ to $(c, d) \in \mathbb{G}.\mathbb{V}$ implies that b and c may access same location.*

Procedure BuildG in Fig. 6 constructs an MPG. In BuildG line 2-4 appends the memory access pairs from $\mathsf{CFG}(f_1), \mathsf{CFG}(f_1), \ldots, \mathsf{CFG}(f_n)$ to $\mathbb{V}$. Line 5-8 compute the $\mathbb{G}.\mathbb{E}$ edges. An edge between (a, b) and (c, d) denotes that $\mathsf{mayAA}(b, c)$ holds. Note that we also create $\mathbb{G}.\mathbb{E}$ edges between access pairs from the same thread function. It is because multiple concurrent threads may execute same thread function and access pairs from a function may result in events which are concurrent in an execution. In this case we effectively analyze all programs of the form $f_1 \mid\mid \cdots f_1 \mid\mid \cdots \mid\mid f_n \cdots \mid\mid f_n$.

B. Checking robustness on MPG

A cycle in MPG $\mathbb{G}$ implies a potential $(\mathsf{epo}; \mathsf{eco})^+$ cycle in an execution. $\mathsf{Cy}(\mathbb{G})$ returns the set of access pairs that may create cycle(s) in the MPG $\mathbb{G}$ i.e.

$$\begin{aligned}\mathsf{Cy}(\mathbb{G}) \triangleq \{&n \mid n \in \mathbb{G}.\mathbb{V} \wedge \exists m, o \in \mathbb{G}.\mathbb{V}.\\ &m \neq n \wedge o \neq n \wedge \mathbb{G}.\mathbb{E}(m, n) \wedge \mathbb{G}.\mathbb{E}(n, o)\}\end{aligned}$$

We do create any self loop in $\mathbb{G}$ on n. A self loop on n implies that n may create concurrent event pair (p, q) and (r, s) in an execution where $\mathsf{eco}(q, r)$ or $\mathsf{eco}(p, s)$ holds which implies $(p, q), (r, s) \in \mathsf{po}_\ell$. However, po_ℓ is included in all M-K robustness condition and therefore multiple event pairs from n does not create any new robustness violation.

If $\mathsf{Cy}(\mathbb{G})$ has any unordered access pair following respective Ord condition then we report M-K robustness violation.

example. Consider the $\textsf{SB2}$ function in Fig. 5. The program $\textsf{SB2(true)} \parallel \textsf{SB2(false)}$ violates SC-x86 robustness due to an execution where $\textsf{R}(Y, 0)$ and $\textsf{R}(X, 0)$ is possible in the first and second threads respectively. We construct the MPG from $\{1, 2, 3, 4\}$ accesses. The subgraph in Fig. 5 contains a cycle of $(1, 3)$ and $(2, 4)$ that depicts SC-x86 robustness violation.

1) Defining Ord Conditions

To define an $\textsf{Ord}$ condition we use the following definitions.

- $\textsf{mustAA}(i, j)$ checks if i and j always access same location.
- Procedure $\textsf{getG}(i)$ returns the CFG $\mathcal{C}$ of instruction i.
- $\mathcal{P}_{\textsf{nf}}$ checks if there exist any path from i to j on the CFG $\mathcal{C}$ without passing through a fence in F. Else in all executions the events from i and j are ordered by a set of fences.

$$\mathcal{P}_{\textsf{nf}}(\mathcal{C}, i, j, F) \triangleq \mathcal{P}(\langle \mathcal{C}.\mathcal{V} \setminus F, \mathcal{C}.\mathcal{E} \setminus B\rangle, i, j)$$
$$\text{where } B = (G.\mathcal{V} \times F) \cup (F \times G.\mathcal{V})$$

- $\textsf{isW}(i)$ and $\textsf{isR}(i)$ check if the access i is write and read respectively.
- $\textsf{isWR}(\mathcal{C}, i, j)$ checks if i and j are write-read pair which may access different locations without any intermediate `RMW`. In an execution i and j may create a $\textsf{WR}$ relation.

$$\begin{aligned}\textsf{isWR}(\mathcal{C}, i, j) \triangleq\, & \textsf{isW}(i) \wedge \textsf{isR}(j) \wedge \neg\textsf{mustAA}(i, j) \\ & \wedge \exists u\ (u \in \textsf{ac}(\mathcal{C}, \textsf{rmw}) \\ & \wedge \mathcal{P}(\mathcal{C}, i, u) \wedge \mathcal{P}(\mathcal{C}, u, j))\end{aligned}$$

x86. The $\textsf{Ord}$ condition for SC-x86 robustness is as follows.

$$\begin{aligned}\textsf{Ord}(\textsf{SC}, \textsf{x86}, \mathcal{C}, i, j) \triangleq\, & \textsf{isR}(i) \vee \textsf{isW}(j) \vee \textsf{mustAA}(i, j) \\ & \vee \neg\mathcal{P}_{\textsf{nf}}(\mathcal{C}, i, j, \textsf{ac}(\mathcal{C}, \textsf{F}))\end{aligned}$$

The $\textsf{isR}(i)$ and $\textsf{isW}(j)$ conditions ensure $\textsf{xppo}$ relations between the events generated from i and j. $\textsf{mustAA}(i, j)$ checks if i and j generated events pairs are in $\textsf{epo}_\ell$ relation. The $\mathcal{P}_{\textsf{nf}}$ condition checks if there are intermediate fences between i and j generated events in all executions. The $\textsf{Ord}$ condition is satisfied in $\textsf{LB}$ and $\textsf{IRIW}$ but violated in the $\textsf{SB}$ program.

In x86 a successful `RMW` results in $\textsf{rmw}$ which acts as an intermediate fence. But a failed `RMW` generates a read event only and it does not act as a fence. Therefore an `RMW` operation between a pair of memory access does not ensure that the access pair is ordered in all execution. However, if an `RMW` is used in an *wait*-loop where the loop terminates only when the `RMW` is successful then the `RMW` in the *wait*-loop acts as a fence in all x86 terminating executions. For these programs we strengthen SC-x86 robustness checking condition as follows.

$$\begin{aligned}\textsf{SOrd}(\textsf{SC}, \textsf{x86}, i, j) \triangleq\, & \textsf{isR}(i) \vee \textsf{isW}(j) \vee \textsf{mustAA}(i, j) \\ & \vee \neg\mathcal{P}_{\textsf{nf}}(\mathcal{C}, i, j, \textsf{ac}(\mathcal{C}, \textsf{F} \cup \textsf{rmw}))\end{aligned}$$

ARMv8(A8). $\textsf{isL}(i)$, $\textsf{isA}(i)$, $\textsf{isAQ}(i)$ check if an access i is a release, acquire, acquire/acquirePC respectively. $\textsf{isLA}(i, j)$ holds for a release, acquire access pair (i, j). $\textsf{Lcoi}(i)$ returns the set of release-writes that access same-location as i. $RA(\mathcal{C}, i)$ returns the set of acquire-reads that is reachable from i through some release-writes.

$$\begin{aligned}RA(\mathcal{C}, i) \triangleq\, & \{a \mid \textsf{isA}(a) \wedge \neg\mathcal{P}_{\textsf{nf}}(\mathcal{C}, i, a, \textsf{ac}(\mathcal{C}, \textsf{L}))\} \\ \textsf{Lcoi}(\mathcal{C}, i) \triangleq\, & \{w \mid \textsf{isL}(w) \wedge \textsf{mustAA}(w, i)\}\end{aligned}$$

We now define the $\textsf{Ord}$ condition for SC-ARMv8 robustness where $B \triangleq \textsf{ac}(\mathcal{C}, \textsf{F}) \cup \textsf{RA}(i)$. It results in $B_F = \textsf{po}; [\textsf{F}]; \textsf{po} \cup \textsf{po}; [\textsf{L}]; \textsf{po}[\textsf{A}]; \textsf{po} \subseteq \textsf{bob}$ that acts as a fence on an $\textsf{epo}$. Moreover we define $\textsf{isRR}(i, j) \triangleq \textsf{isR}(i) \wedge \textsf{isR}(j)$, $\textsf{isRW}(i, j) \triangleq \textsf{isR}(i) \wedge \textsf{isW}(j)$, $\textsf{isWW}(i, j) \triangleq \textsf{isW}(i) \wedge \textsf{isW}(j)$.

$$\textsf{Ord}(\textsf{SC}, \textsf{A8}, \mathcal{C}, i, j) \triangleq \textsf{mustAA}(i, j) \qquad (1)$$
$$\vee(\neg\mathcal{P}_{\textsf{nf}}(\mathcal{C}, i, j, B)) \vee \textsf{isLA}(i, j) \vee \textsf{isAQ}(i) \vee \textsf{isL}(j) \qquad (2)$$
$$\vee(\textsf{isRR}(i, j) \wedge \neg\mathcal{P}_{\textsf{nf}}(\mathcal{C}, i, j, B \cup \textsf{ac}(\mathcal{C}, \textsf{F}_{\text{LD}}))) \qquad (3)$$
$$\vee(\textsf{isRW}(i, j) \wedge \neg\mathcal{P}_{\textsf{nf}}(\mathcal{C}, i, j, B \cup \textsf{ac}(\mathcal{C}, \textsf{F}_{\text{LD}}) \cup \textsf{Lcoi}(\mathcal{C}, j))) \qquad (4)$$
$$\vee(\textsf{isWW}(i, j) \wedge \neg\mathcal{P}_{\textsf{nf}}(\mathcal{C}, i, j, B \cup \textsf{ac}(\mathcal{C}, \textsf{F}_{\text{ST}}) \cup \textsf{Lcoi}(\mathcal{C}, j))) \qquad (5)$$

The definition ensures that the generated events from i and j are in (1) $\textsf{po}_\ell$ or in one of the following $\textsf{bob}$ relations: (2) $B_F \cup [\textsf{L}]; \textsf{po}; [\textsf{A}] \cup [\textsf{A} \cup \textsf{Q}]; \textsf{po} \cup \textsf{po}; [\textsf{L}]$, (3) $B_F \cup [\textsf{R}]; \textsf{po}; [\textsf{F}_{\text{LD}}]; \textsf{po}$, (4) $B_F \cup [\textsf{R}]; \textsf{po}; [\textsf{F}_{\text{LD}}]; \textsf{po} \cup \textsf{po}; [\textsf{L}]; \textsf{coi}$, (5) $B_F \cup [\textsf{W}]; \textsf{po}; [\textsf{F}_{\text{ST}}]; \textsf{po}; [\textsf{W}] \cup \textsf{po}; [\textsf{L}]; \textsf{coi}$. The overall condition ensures SC-ARMv8 robustness. The condition is satisfied in $\textsf{IRIW}$ but violated in $\textsf{SB}$ and $\textsf{LB}$.

The $\textsf{dob}$ and $\textsf{aob}$ relations also order memory accesses. From the definition $\textsf{aob} \subseteq \textsf{po}_\ell$ which is already captured by (1). We do not include $\textsf{dob}$ in the $\textsf{Ord}$ condition as a dependency can be optimized away after the robustness analysis which may result in a non-robust program even when we report the original program to be robust.

Next, we define x86-ARMv8 robustness condition where an (i, j) access pair is ordered or may generate a $\textsf{WR}$ pair.

$$\textsf{Ord}(\textsf{x86}, \textsf{A8}, \mathcal{C}, i, j) \triangleq \textsf{Ord}(\textsf{SC}, \textsf{A8}, \mathcal{C}, i, j) \vee \textsf{isWR}(\mathcal{C}, i, j)$$

$\textsf{SB}$ and $\textsf{IRIW}$ satisfy the condition but $\textsf{LB}$ violates it.

ARMv7(A7). We define the $\textsf{Ord}$ condition to ensure the SC-ARMv7 robustness condition in all ARMv7 executions. Then we extend the $\textsf{Ord}$ for SC-ARMv7 to define the $\textsf{Ord}$ conditions for x86-ARMv7 and ARMv8-ARMv7 robustness.

$$\begin{aligned}\textsf{Ord}(\textsf{SC}, \textsf{A7}, \mathcal{C}, i, j) &\triangleq \textsf{mustAA}(i, j) \vee (\neg\mathcal{P}_{\textsf{nf}}(\mathcal{C}, i, j, \textsf{ac}(\mathcal{C}, \textsf{F}))) \\ \textsf{Ord}(\textsf{x86}, \textsf{A7}, \mathcal{C}, i, j) &\triangleq \textsf{Ord}(\textsf{SC}, \textsf{A7}, \mathcal{C}, i, j) \vee \textsf{isWR}(\mathcal{C}, i, j) \\ \textsf{Ord}(\textsf{A8}, \textsf{A7}, \mathcal{C}, i, j) &\triangleq \textsf{Ord}(\textsf{SC}, \textsf{A7}, \mathcal{C}, i, j) \vee \textsf{isW}(i)\end{aligned}$$

The memory access pairs in the $\textsf{LB}$ program satisfies the ARMv8-ARMv7, and the $\textsf{SB}$ program satisfies the x86-ARMv7, ARMv8-ARMv7 conditions.

2) Robustness Analysis and Enforcement Procedure

The $\textsf{MKRobust}$ procedure in Fig. 6 checks M-K robustness on an MPG $\mathbb{G}$: (line 3) we first compute $\textsf{Cy}(\mathbb{G})$. (line 4-7) if an access pair (a, b) in $\textsf{Cy}(\mathbb{G})$ is on a cycle then we check if (a, b) is ordered by the $\textsf{Ord}$ condition. (line 8) returns the unordered memory access pairs O.

If O is empty then the program is M-K robust. Else $\textsf{Enforce}$ procedure insert appropriate fences to enforce robustness. Procedure $\textsf{getF}$ returns a fence based on the access type a and

procedure BuildG($\{f_1, \ldots, f_n\}$)
  for $f \in \{f_1, \ldots, f_n\}$ **do**
    $\mathcal{C} \leftarrow \mathsf{CFG}(f)$;
    $\mathbb{V} \leftarrow \mathbb{V} \cup \mathsf{MM}(\mathcal{C})$;
  for $(a, b) \in \mathbb{V}$ **do**
    for $(c, d) \in \mathbb{V}$ **do**
      if $\mathsf{mayAA}(b, c)$ **then**
        $\mathbb{E} \leftarrow \mathbb{E} \cup \{(a, b), (c, d)\}$;
  return $\langle \mathbb{V}, \mathbb{E} \rangle$;
end procedure

procedure MKRobust(M, K, $\mathbb{G}$)
  $O \leftarrow \emptyset$;
  $AB \leftarrow \mathsf{Cy}(\mathbb{G})$;
  for $(a, b) \in AB$ **do**
    $\mathcal{C} \leftarrow \mathsf{getG}(b)$;
    if $\neg\mathsf{Ord}(\mathsf{M},\mathsf{K},\mathcal{C}, a, b)$ **then**
      $O \leftarrow O \cup \{(a, b)\}$;
  return O;
end procedure

procedure Enforce(K, O)
  $H \leftarrow \emptyset$;
  for $(a, b) \in O$ **do**
    if $b \notin H$ **then**
      $f \leftarrow \mathsf{getF}(K, a, b)$;
      $\mathsf{insertF}(\mathsf{getG}(b), a, b, f)$;
      $H \leftarrow H \cup \{b\}$;
end procedure

$\mathbb{G} \leftarrow \mathsf{BuildG}(\{f_1, \ldots, f_n\})$; $O \leftarrow \mathsf{MKRobust}(M, K, \mathbb{G})$; $\mathsf{Enforce}(K, O)$;

Fig. 6: Static M-K robustness analysis and enforcement.

procedure getF(K, a, b)
  if $K == \mathsf{x86}$ **then** **return** new(`MFENCE`);
  if $K == \mathsf{A7}$ **then** **return** new(`DMB`);
  if $K == \mathsf{A8}$ **then**
    if $\mathsf{isW}(a) \land \mathsf{isR}(b)$ **then** **return** new(`DMBFULL`);
    if $\mathsf{isW}(a) \land \mathsf{isW}(b)$ **then** **return** new(`DMBST`);
    if $\mathsf{isR}(a)$ **then** **return** new(`DMBLD`);
end procedure

procedure insertF($\mathcal{C}, a, b, f$)
  $\mathcal{V}' \leftarrow \mathcal{C}.\mathcal{V} \cup \{f\}$;
  $\mathcal{E}_1 \leftarrow \mathcal{C}.\mathcal{E} \cup \{(f, b)\}$
  $\mathcal{E}' \leftarrow \mathcal{E}_1 \cup \{(e, f) \mid \mathcal{C}.\mathcal{E}^+(e, b)\} \cup \{(f, e) \mid \mathcal{C}.\mathcal{E}^+(b, e)\}$
  return $\langle \mathcal{V}', \mathcal{E}' \rangle$;
end procedure

Fig. 7: Procedure getF and insertF.

b in the memory model K. Procedure insertF inserts the fence f between a and b. Note that one inserted fence may order multiple access pairs. These methods are defined in Fig. 7. In case of x86 and ARM programs we insert `MFENCE` and `DMB` respectively. In ARMv8 we first insert `DMBFULL` followed by `DMBLD` and then `DMBST` fences.

C. *Complexity of Robustness*

To analyze the complexity of the robustness algorithm we analyze the main procedures: BuildG, MKRobust, and Enforce which perform MM, $\mathcal{P}_{\mathsf{nf}}$, and Cy computations. Given a program with n statements, the number of shared memory accesses and control flow edges are bound by n and n^2 respectively. Hence MM contain maximum n^2 elements and $\mathcal{P}_{\mathsf{nf}}$ computation is bound by traversing n^2 edges. So procedure BuildG constructs an MPG graph with maximum $|\,\mathsf{MM}\,| = n^2$ nodes and $|\,\mathsf{MM}\,|^2 = n^4$ edges. Hence Cy computation traverses maximum n^4 edges. In procedure MKRobust, for each node in MPG, we check (i) if it is on the cycle by computing Cy (ii) if yes then it performs $\mathcal{P}_{\mathsf{nf}}$ computation for the memory access pair. Hence MKRobust overall incurs $n^2 * (n^4 + n^2) = n^6 + n^4$ computation. Next, procedure Enforce takes maximum n^2 computation for each access pair in MM and for overall incurs maximum $n^2 * |\,\mathsf{MM}\,| = n^4$ computation. Hence, the robustness checking and enforcement computation is bounded by $\mathsf{O}(n^6)$ which is polynomial in terms of the program size.

V. Experimental Evaluation

Implementation. We implement the robustness analysis and enforcement techniques in *Fency* (for FENCe analYsis) as LLVM compiler passes for x86, ARMv8, and ARMv7 programs. We leverage the existing analyses in LLVM. The CFG analyses are used to define MM, Path, $\mathcal{P}$, and $\mathcal{P}_{\mathsf{nf}}$ conditions. We define the mayAA and mustAA conditions using memory operand type and alias analyses provided in LLVM.

We run the analyses on a MacOS machine having a 2.4GHz 8-Core Intel i9 processor with 64 GB RAM.

Benchmarks. We analyze a number of well-known concurrent algorithms and data structures [14, 27] including global barrier (Barrier) construct, mutual exclusion algorithms (by Dekker, Peterson, and Lamport), different lock algorithms (e.g. Spinlock, Seqlock, Ticketlock), non-blocking write protocol (NBW), read-copy-update (RCU) programs, work-stealing queue in Cilk, and ChaseLev dequeue. These programs use C11 [28, 29] atomic accesses extensively. The release-acquire(RA)/TSO/SC versions indicate the memory model for which the respective version is developed. The number of lines in the LLVM IR (.ll) files vary between 100-400 which indicate the approximate size of an analyzed CFG.

Naive fence insertion scheme. We compare *Fency* to a naive scheme which does not use robustness information in fence insertion. The naive scheme works as follows.

- Eliminate existing fences in concurrent threads.
- Enforce robustness by fence insertion in concurrent threads.
  - (x86) Insert `MFENCE` after load, store, and `RMW` accesses.
  - (ARMv8) Insert `DMBLD` after non-acquire loads and `DMBFULL` for other memory accesses.
  - (ARMv7) Insert `DMB` after all memory accesses.

A. *Experimental Results*

In Figs. 8 and 9 we report the results of some benchmarks. The full results are in the supplementary material [15]. For comparison we also provide the number of fences required by

Prog.	SC-x86		Trencher	
	result	⟨sec	result	⟨sec
Barrier	6|0✗2	⟨0.005	✗2	⟨0.004
Dekker-TSO	20|4✓0	⟨0.002	✓0	⟨0.007
Peterson-SC	14|0✗2	⟨0.004	✗2	⟨0.013
Lamport-SC	17|0✗4	⟨0.019	✗4	⟨0.107
Spinlock	14|0✓0	⟨0.004	✓0	⟨0.007
Ticketlock	12|0✓0	⟨0.004	✓0	⟨0.006
Seqlock	7|0✓0	⟨0.004	✓0	⟨0.582
RCU-offline	33|4✗3	⟨0.038	✗-	⟨0.246
Cilk-TSO	22|2✓0	⟨0.011	✗0	⟨2.039
Cilk-SC	22|0✓0	⟨0.010	✓2	⟨6.322

Prog.	ARMv7					
	SC		x86		ARMv8	
	result	⟨sec	result	⟨sec	result	⟨sec
Barrier	6|2✗2	⟨0.012	6|2✓0	⟨0.002	6|2✓0	⟨0.002
Dekker-TSO	20|8✗6	⟨0.003	20|8✗6	⟨0.007	20|8✗6	⟨0.009
Peterson-SC	14|0✗12	⟨0.002	14|0✗10	⟨0.002	14|0✗8	⟨0.003
Lamport-SC	17|7✗10	⟨1.699	17|7✗8	⟨1.659	17|7✗5	⟨1.698
Spinlock	18|12✓0	⟨0.141	18|12 ✓0	⟨0.133	18|12✓0	⟨0.133
Ticketlock	14|8✓0	⟨0.025	14|8✓0	⟨0.022	14|8✓0	⟨0.023
Seqlock	9|6✗2	⟨0.006	9|6✗2	⟨0.002	9|6✗2	⟨0.002
RCU-offline	36|19✗17	⟨0.335	36|19✗15	⟨0.334	36|19✗10	⟨0.339
Cilk-TSO	33|10✗6	⟨2.455	33|10✗6	⟨2.411	33|10✗6	⟨2.427
Cilk-SC	33|8✗7	⟨2.445	33|8✗7	⟨2.410	33|8✗7	⟨2.411

Fig. 8: Robustness analyses and enforcement for x86 and ARMv7 programs.

the naive schemes as well as the results from state-of-the-art x86-robustness checker Trencher [8].

Intrpreting the Results. The (SC-K) entries in the tables are of the form (a|b(✓/✗) c ⟨ d) where

- 'a': number of fences required by naive scheme.
- 'b': number of existing fences in the program.
- 'c': number of fences inserted by proposed scheme.
- '✓/✗' symbol denotes if a program is M-K robust or not.
- 'd': time taken by the robustness pass in seconds.

In ARMv8 we show total number of `DMB`(FULL/LD/ST) fences. We use $\#(a\text{-}(b{+}c))$ less fences than the naive schemes e.g. from Fig. 8 the Barrier program requires 6-(0+2)=4 less fences than the naive scheme to enforce SC-x86 robustness.

For Trencher we analyze the encoded programs taken from [14]. We report if the program is SC-x86 robust (✓/✗), number of inserted fences (i.e. 'c') and the execution time (i.e. 'd'). Trencher fence insertion does not terminate for RCU-offline.

1) Checking Robustness

x86 programs. We report the SC-x86 robustness analysis results of *Fency* in Fig. 8 (and in [15]) and compare the results from Trencher. on the corresponding programs.

The SC-x86 robustness analysis in *Fency* is quite precise and agrees to Trencher in all cases except Lamport-RA, Lamport-TSO, and Cilk-SC programs. Lamport-(RA/TSO) have unordered write-read pairs that generate WR relations and hence *Fency* report SC-robustness violation though these access pairs never execute concurrently in any x86 execution. Moreover, in most cases *Fency* insert same number of fences as Trencher.

We note a subtle case in Cillk-SC. It has an access sequence $a = \mathsf{R}_{\text{RLX}}(T); \mathsf{W}_{\text{RLX}}(T, a\text{-}1); \mathsf{R}_{\text{RLX}}(H)$. Trencher reports SC-violation due to the WR pair. However, LLVM combines the load and store of T and create an atomic fetch-and-sub: $a = \mathsf{R}_{\text{RLX}}(T); \mathsf{W}_{\text{RLX}}(T, a\text{-}1) \rightsquigarrow a = \mathsf{fsub}(T, 1)$. Hence the resulting x86 program ensures SC-robustness which *Fency* reports correctly.

We also note the execution time of *Fency* and of Trencher. Trencher incurs significantly more time for the Seqlock, Cilk-TSO, Cilk-SC programs and does not terminate for RCU-offline fence insertion. Trencher exhibits comparable efficiency in certain programs e.g. Spinlock, Ticketlock. However, in these programs also if we increase the number of threads by replicating the thread functions then Trencher incurs orders of seconds to check and enforce robustness. At the same time Trencher inserts more fences. On the other hand, the analyses in *Fency* are parameterized by thread functions and therefore are unaffected by the number of executing threads.

Prog.	ARMv8			
	SC		x86	
	result	⟨sec	result	⟨sec
Barrier	6|2✗2	⟨0.009	6|2✗0	⟨0.007
Dekker-TSO	20|8✗4	⟨0.007	20|8✗4	⟨0.011
Peterson-SC	14|0✗11	⟨0.001	14|0 ✗10	⟨0.001
Lamport-SC	17|7✗9	⟨0.007	17|7✗9	⟨0.008
Spinlock	18|12✗4	⟨0.017	18|12 ✗4	⟨0.009
Ticketlock	14|8✗2	⟨0.006	14|8✗2	⟨0.007
Seqlock	9|6✗2	⟨0.002	9|6 ✗2	⟨0.005
RCU-offline	35|16✗17	⟨0.157	35|16 ✗19	⟨0.160
Cilk-TSO	33|10✗7	⟨0.025	33|10 ✗7	⟨0.024
Cilk-SC	33|8✗8	⟨0.011	33|8✗8	⟨0.012

Fig. 9: Robustness analyses & enforcement in ARMv8.

ARMv8 programs. In Fig. 9 (and in [15]) we report the robustness results of the ARMv8 programs. The ARMv8 programs violate SC and x86 robustness as the programs contain independent memory accesses on different locations which are unordered in ARMv8.

As ARMv8 is weaker than x86, the programs (e.g. Barrier) which violate SC-x86 robustness also violate SC-ARMv8 robustness. Moreover, there are programs which are SC-x86 robust but violates SC-ARMv8 robustness such as dekker-TSO. These programs violate both SC-ARMv8 and x86-ARMv8 robustness due to unordered accesses that result in $[\mathsf{R}]; \mathsf{po}_{\neq\ell}; [\mathsf{R}]$ or $[\mathsf{W}]; \mathsf{po}_{\neq\ell}; [\mathsf{W}]$ relation in an execution. These access pairs are ordered in x86 but not in ARMv8 and hence violate x86-ARMv8 robustness.

Robustness of ARMv7 programs. In general the ARMv7 programs violate robustness when x86 or ARMv8 are not robust as shown in Fig. 8 (and in [15]). However, C11 release/acquire/SC accesses which generate full fences in ARMv7 and synchronizing accesses in ARMv8 which act as half fences. As a result, in some programs the ARMv7 version enforce stronger ordering than the ARMv8 version. Hence the ARMv7 programs are robust unlike the ARMv8 programs. For example, Consider the C11 event (without read/written values) sequences from Spinlock and Ticketlock programs and their C11 to ARMv8 and ARMv7 mappings [30].

$$\mathsf{R}(X) \cdot \mathsf{W}_{\mathrm{SC}}(Y) \cdot \mathsf{R}(Z) \mapsto \mathsf{R}(X) \cdot \mathsf{L}(Y) \cdot \mathsf{R}(Z) \quad \text{(C-v8)}$$
$$\mathsf{R}(X) \cdot \mathsf{W}_{\mathrm{SC}}(Y) \cdot \mathsf{R}(Z) \mapsto \mathsf{R}(X) \cdot \mathsf{F} \cdot \mathsf{W}(Y) \cdot \mathsf{F} \cdot \mathsf{R}(Z) \quad \text{(C-v7)}$$

The reads are unordered in ARMv8 and may violate SC-ARMv8. The ARMv7 event sequence is ordered by fences that leads to SC-ARMv7 robustness.

The Barrier (and Peterson-RA-b) program violates SC-ARMv7 due to unordered store-load pairs, but satisfies x86 and ARMv8 robustness. Some ARMv7 programs violate SC, x86, ARMv8 robustness due to unordered read-read pairs.

2) Enforcing robustness.

In most of the programs enforcing weaker model requires less number of inserted fences. However, certain ARMv8 programs (e.g. lamport-SC) incur less fences to enforce SC-ARMv8 than x86-ARMv8. Consider the ARMv8 sequence $\mathsf{W}(X) \cdot \mathsf{R}(X) \cdot \mathsf{R}(Y) \cdot \mathsf{W}(Y)$ that may violate SC-ARMv8 and x86-ARMv8. To ensure SC-ARMv8 we insert a `DMBFULL` that results in $\mathsf{W}(X) \cdot \mathsf{R}(X) \cdot \mathsf{F} \cdot \mathsf{R}(Y) \cdot \mathsf{W}(Y)$ sequence. To ensure x86-ARMv8 we insert a `DMBLD` and a `DMBST` to generate a $\mathsf{W}(X) \cdot \mathsf{R}(X) \cdot \mathsf{F}_{\mathrm{LD}} \cdot \mathsf{R}(Y) \cdot \mathsf{F}_{\mathrm{ST}} \cdot \mathsf{W}(Y)$ sequence.

3) Performance of Robustness Analyses

We have already compared the execution times of SC-x86 robustness analysis in *Fency* and Trencher. In case of ARM program versions *Fency* incurs less than a second except for ARMv7 Cilk-(TSO/SC) programs. The timings of *Fency* analyses vary among different program versions. It is because LLVM may optimize a program differently for different architectures. So the number of memory accesses (parameter 'a' in Figs. 8 and 9) and the number of memory access pairs vary. Moreover, the CFGs in different architectures also differ which affect the $\mathcal{P}_{\mathsf{nf}}$ and Cy computations.

VI. Related Work

SC-robustness is studied against TSO [3, 4, 5, 6, 7, 8, 9, 10], PSO [11, 12], POWER [13], and Release-Acquire [14] models by exploring possible executions using model checking tools. On the contrary, we analyze and transform programs as LLVM passes without exploring program executions.

[8] check and enforce SC-robustness for parameterized programs for any number of threads. It reduces the robustness checking problem to parameterized reachability analysis on possible executions. Instead, our approach is static and parameterized over the thread functions for any number of threads.

PORTHOS [31] checks portability of a program from one model to another, particularly from POWER to TSO by encoding models in SAT/SMT solvers. On the contrary, we check robustness or portability of ARM models which are different from POWER. In addition, our analysis enable fence insertion to enforce robustness unlike PORTHOS.

A number of approaches [32, 8, 33, 34, 35, 18, 6, 11] propose fence insertion to ensure SC. Among these fence insertion schemes our approach is closer to static approaches [34, 18, 35]. [18] use delay-set analysis to ensure SC for weak memory programs. [35] proved that identifying minimal set of fences is NP-hard and proposed minimal fence insertion based on control flow analysis. Similar to [35], we analyze control flow graph without exploring the executions.

[32] checks SC-robustness against x86 and POWER, and restore SC by inserting lock-unlock or `RMW` constructs. [34] proposed fence insertion in POWER to strengthen a program to release/acquire semantics which has same ordering constraints between memory accesses as TSO. On the contrary, we propose M-K robustness; we define robustness conditions for ARMv7 and ARMv8 programs and show that ppo is not sufficient to enforce SC in ARMv7. Moreover, we analyze parameterized programs unlike these approaches.

We extend abstract event graph (AEG) from [34] and propose memory pair graph in our analyses. An AEG captures the possible execution graphs statically for a given set of threads and statically detect possible robustness-violating cycles which may occur in an execution. The proposed memory-access pair graph (MPG) also considers that the program is parameterized where each thread function may create multiple threads and hence construct the event graph on all memory access pairs from all threads. Then similar to AEG we statically detect possible robustness-violating cycles on MPG. However, our fence insertion may not be optimal; identifying optimal fence insertion is an well studied problem [35, 18, 34] which we will pursue in the context of M-K robustness.

VII. Conclusion and Future Work

In this paper we identify robustness conditions for x86, ARMv8, and ARMv7 relaxed memory models. Based on these identified conditions we check M-K robustness. If robustness is violated we insert appropriate fences to enforce robustness. We implement our approach as LLVM compiler passes and evaluate the efficiency on a number of well-known concurrent algorithms and data structures.

Going forward we want to extend the analyses to other concurrency features in x86 and ARM models [36]. We would also like to extend these analyses to other architectures such as RISC-V [37] and Power [38].

References

[1] A. Barbalace, M. L. Karaoui, W. Wang, T. Xing, P. Olivier, and B. Ravindran, "Edge computing: the case for heterogeneous-isa container migration," in *VEE'20*, 2020, pp. 73–87.

[2] A. Barbalace, R. Lyerly, C. Jelesnianski, A. Carno, H. Chuang, V. Legout, and B. Ravindran, “Breaking the boundaries in heterogeneous-isa datacenters,” in *ASPLOS 2017*, 2017, pp. 645–659.

[3] S. Burckhardt and M. Musuvathi, “Effective program verification for relaxed memory models,” in *CAV'08*, 2008, pp. 107–120.

[4] A. Bouajjani, R. Meyer, and E. Möhlmann, “Deciding robustness against total store ordering,” in *ICALP'11*, 2011, pp. 428–440.

[5] J. Burnim, K. Sen, and C. Stergiou, “Sound and complete monitoring of sequential consistency for relaxed memory models,” in *TACAS'11*, 2011, pp. 11–25.

[6] A. Linden and P. Wolper, “A verification-based approach to memory fence insertion in relaxed memory systems,” in *SPIN'11*, 2011, pp. 144–160.

[7] A. Gotsman, M. Musuvathi, and H. Yang, “Show no weakness: Sequentially consistent specifications of tso libraries,” 2012.

[8] A. Bouajjani, E. Derevenetc, and R. Meyer, “Checking and enforcing robustness against TSO,” in *ESOP 2013*, 2013, pp. 533–553.

[9] P. A. Abdulla, M. F. Atig, and T.-P. Ngo, “The best of both worlds: Trading efficiency and optimality in fence insertion for tso,” in *Programming Languages and Systems*. Springer Berlin Heidelberg, 2015, pp. 308–332.

[10] A. Bouajjani, C. Enea, S. O. Mutluergil, and S. Tasiran, “Reasoning about tso programs using reduction and abstraction,” in *CAV'18*, 2018, pp. 336–353.

[11] A. Linden and P. Wolper, “A verification-based approach to memory fence insertion in pso memory systems,” in *TACAS'13*, 2013, pp. 339–353.

[12] P. A. Abdulla, M. F. Atig, M. Lång, and T. P. Ngo, “Precise and sound automatic fence insertion procedure under pso,” in *Networked Systems*, 2015, pp. 32–47.

[13] E. Derevenetc and R. Meyer, “Robustness against power is pspace-complete,” in *ICALP'14*, ser. LNCS, vol. 8573, 2014, pp. 158–170.

[14] O. Lahav and R. Margalit, “Robustness against release/acquire semantics,” in *PLDI 2019*, 2019, pp. 126–141.

[15] S. Chakraborty, “Technical appendix.” 2021, available at https://www.st.ewi.tudelft.nl/sschakraborty/mkrobustness.html.

[16] A. Podkopaev, O. Lahav, and V. Vafeiadis, “Promising compilation to armv8,” in *ECOOP'17*, 2017.

[17] “The LLVM compiler infrastructure,” http://llvm.org/.

[18] D. E. Shasha and M. Snir, “Efficient and correct execution of parallel programs that share memory,” *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 2, pp. 282–312, 1988.

[19] J. Alglave, L. Maranget, and M. Tautschnig, “Herding cats: modelling, simulation, testing, and data-mining for weak memory,” *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, pp. 7:1–7:74, 2014.

[20] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, “Repairing sequential consistency in C/C++11,” in *PLDI 2017*, 2017, pp. 618–632, technical Appendix Available at https://plv.mpi-sws.org/scfix/full.pdf.

[21] J. Alglave and L. Maranget, “herd7 consistency model simulator,” http://diy.inria.fr/www/.

[22] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell, “Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8,” *PACMPL*, vol. 2, no. POPL, pp. 19:1–19:29, 2018.

[23] “SC cat file,” 2021, available at https://github.com/herd/herdtools7/blob/master/herd/libdir/sc.cat.

[24] “x86 cat file,” 2021, available at https://github.com/herd/herdtools7/blob/master/herd/libdir/x86tso.cat.

[25] “Armv8 cat file,” 2021, available at https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat.

[26] “Armv7 cat file,” 2021, available at https://github.com/herd/herdtools7/blob/master/herd/libdir/arm.cat.

[27] B. Norris and B. Demsky, “CDSChecker: Checking concurrent data structures written with C/C++ atomics,” in *OOPSLA'13*, 2013.

[28] ISO/IEC 9899, “Programming language C,” 2011.

[29] ISO/IEC 14882, “Programming language C++,” 2011.

[30] “C/C++11 mappings to processors,” https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html.

[31] H. Ponce de León, F. Furbach, K. Heljanko, and R. Meyer, “Portability analysis for weak memory models. porthos: One tool for all models,” 08 2017.

[32] J. Alglave and L. Maranget, “Stability in weak memory models,” in *CAV 2011*, 2011, p. 50–66.

[33] F. Liu, N. Nedev, N. Prisadnikov, M. Vechev, and E. Yahav, “Dynamic synthesis for relaxed memory models,” in *PLDI '12*, 2012, pp. 429–440.

[34] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl, “Don't sit on the fence: A static analysis approach to automatic fence insertion,” *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 2, pp. 6:1–6:38, 2017.

[35] J. Lee and D. A. Padua, “Hiding relaxed memory consistency with a compiler,” *IEEE Transactions on Computers*, vol. 50, no. 8, pp. 824–833, 2001.

[36] J. Alglave, W. Deacon, R. Grisenthwaite, A. Hacquard, and L. Maranget, “Armed cats: Formal concurrency modelling at arm,” vol. 43, no. 2, 2021.

[37] “RISC-V specification.” https://riscv.org/technical/specifications/.

[38] “Powerisa public.v3.1.” https://wiki.raptorcs.com/wiki/File:PowerISA_public.v3.1.pdf.

Appendix

This is the technical appendix of the article "Robustness between Weak Memory Models." It contains the proofs used in this paper.

CONTENTS

- Appendix A reviews SC, x86, ARMv8, and ARMv7 memory models.
- Appendix B contains the proof of SC-x86 robustness.
- Appendix C contains the proofs of SC-ARMv8 and x86-ARMv8 robustness.
- Appendix D contains the proofs of SC-ARMv7, x86-ARMv7, and ARMv8-ARMv7 robustness.
- Appendix E contains the additional experimental results.

A. Concurrency Models of Architectures

Events. In all these model we use the following events.

- Read event $\mathsf{R}(x, v)$ reads value v from location x.
- Write event $\mathsf{W}(x, v)$ writes value v to location x.
- Fence event F.
- Non-rmw load event $\mathsf{Ld}(x, v) \subseteq \mathsf{R}(x, v)$ and in this case $\mathsf{Ld} \triangleq \mathsf{R} \setminus \mathsf{dom}(\mathsf{rmw})$.
- Non-rmw store event $\mathsf{St}(x, v) \subseteq \mathsf{W}(x, v)$ and in this case $\mathsf{St} \triangleq \mathsf{W} \setminus \mathsf{codom}(\mathsf{rmw})$.

Common relations in these models

- Program-order po.
- Reads-from rf.
- Coherence-order co.
- From-read fr.
- Read-modify-write $\mathsf{rmw} \subseteq \mathsf{imm}(\mathsf{po}) \cap ([\mathsf{R}] \times [\mathsf{W}])_\ell$

1) x86 Model.

Relations.

- Relation *x86-preserved-program-order*(xppo).

$$\mathsf{xppo} \triangleq ((\mathsf{W} \times \mathsf{W}) \cup (\mathsf{R} \times \mathsf{W}) \cup (\mathsf{R} \times \mathsf{R})) \cap \mathsf{po}$$

- Relation $\mathsf{implied}$ for fences.

$$\mathsf{implied} \triangleq \mathsf{po}; [\mathsf{dom}(\mathsf{rmw}) \cup \mathsf{F}] \cup [\mathsf{codom}(\mathsf{rmw}) \cup \mathsf{F}]; \mathsf{po}$$

- Relation *x86-happens-before*(xhb).

$$\mathsf{xhb} \triangleq \mathsf{xppo} \cup \mathsf{implied} \cup \mathsf{rfe} \cup \mathsf{fr} \cup \mathsf{co}$$

Consistency Constraints. A well-formed x86 execution X is consistent when:

- $\mathsf{X.po}_\ell \cup \mathsf{X.rf} \cup \mathsf{X.fr} \cup \mathsf{X.co}$ is acyclic. (sc-per-loc)
- $\mathsf{X.rmw} \cap (\mathsf{X.fre}; \mathsf{X.coe}) = \emptyset$. (atomicity)
- $\mathsf{X.xhb}$ is acyclic. (GHB)

2) ARMv7 Model

Relations. ARMv7 model defines following relations on events.

Dependency relations.

- $\mathsf{data} \subseteq \mathsf{R} \times \mathsf{W}$ captures the data dependencies of write events.
- $\mathsf{addr} \subseteq \mathsf{R} \times (\mathsf{R} \cup \mathsf{W})$ captures the address dependencies of memory access events.
- $\mathsf{ctrl} \subseteq \mathsf{R} \times \mathsf{E}$ captures the control dependencies of events. Fence `ISB` constructs `CBISB` primitive which results in $\mathsf{ctrl}_{\mathtt{ISB}}$ relation in an execution. Moreover, $\mathsf{ctrl}; \mathsf{po} \subseteq \mathsf{ctrl}$ and $\mathsf{ctrl}_{\mathtt{ISB}}; \mathsf{po} \subseteq \mathsf{ctrl}_{\mathtt{ISB}}$ holds in ARMv7 model.

Relation read-different-writes and detour

- Relation *read-different-writes*(rdw) relates two same-location po-related reads which reads-from different writes.

$$\mathsf{rdw} \triangleq (\mathsf{fre}; \mathsf{rfe}) \cap \mathsf{po}$$

- Relation detour relates a write w to a read r which are po-related and r reads-from a write w' which is coe-after w.

$$\mathsf{detour} \triangleq (\mathsf{coe}; \mathsf{rfe}) \cap \mathsf{po}$$

In ARMv7 a memory access event consist of *init* and *commit* steps. The *init* and *commit* steps of different events are related by ii_0, ci_0, ic_0, cc_0.

$$\mathsf{ii}_0 \triangleq \mathsf{addr} \cup \mathsf{data} \cup \mathsf{rdw} \cup \mathsf{rfi} \qquad \mathsf{ic}_0 \triangleq \emptyset$$
$$\mathsf{ci}_0 \triangleq \mathsf{ctrl}_{\mathtt{ISB}} \cup \mathsf{detour} \qquad \mathsf{cc}_0 \triangleq \mathsf{data} \cup \mathsf{ctrl} \cup \mathsf{addr}; \mathsf{po}^?$$

Based on these components ii, ic, ci, cc relations can be defined from the following sequential compositions and the constraints ([20, F.1]).

$$\mathsf{xy} \triangleq \bigcup_{n \geq 1} \mathsf{x}^1\mathsf{y}_0^1; \mathsf{x}^2\mathsf{y}_0^2; \cdots \mathsf{x}^n\mathsf{y}_0^n \text{ where}$$

- $\mathsf{x}, \mathsf{y}, \mathsf{x}^1 \cdots \mathsf{x}^n, \mathsf{y}^1 \cdots \mathsf{y}^n \in \{\mathsf{i}, \mathsf{c}\}$.
- If $\mathsf{x} = \mathsf{c}$ then $\mathsf{x}^1 = \mathsf{c}$.
- For every $1 \leq k \leq n-1$, if $\mathsf{y}^k = \mathsf{c}$ then $\mathsf{x}^{k+1} = \mathsf{c}$.
- If $\mathsf{y} = \mathsf{i}$ then $\mathsf{y}^n = \mathsf{i}$.

Finally, ARMv7 defines preserved-program-order $\mathsf{ppo} \subseteq \mathsf{po}$ relation.

$$\mathsf{ppo} \triangleq [\mathsf{R}]; \mathsf{ii}; [\mathsf{R}] \cup [\mathsf{R}]; \mathsf{ic}; [\mathsf{W}]$$

- Relation $\mathsf{fence} \subseteq \mathsf{po}$ also orders memory access pairs. Note that unlike x86, rmw does not act as a fence in ARMv7.

$$\mathsf{fence} \triangleq [\mathsf{R} \cup \mathsf{W}]; \mathsf{po}; [\mathsf{F}]; \mathsf{po}; [\mathsf{R} \cup \mathsf{W}]$$

- Relation happens-before(ahb). A pair of ppo or fence related events in the same thread or a pair of rfe related events in different threads are in happens-before(ahb) relation.

$$\mathsf{ahb} \triangleq \mathsf{ppo} \cup \mathsf{fence} \cup \mathsf{rfe}$$

- Relation propagation(prop) defines the order in which writes propagate to other threads.

$$\mathsf{prop} \triangleq \mathsf{prop}_1 \cup \mathsf{prop}_2 \text{ where}$$
$$\mathsf{prop}_1 \triangleq [\mathsf{W}]; \mathsf{rfe}^?; \mathsf{fence}; \mathsf{ahb}^*; [\mathsf{W}]$$
$$\mathsf{prop}_2 \triangleq (\mathsf{coe} \cup \mathsf{fre})^?; \mathsf{rfe}^?; (\mathsf{fence}; \mathsf{ahb}^*)^?; \mathsf{fence}; \mathsf{ahb}^*$$

Note that $\mathsf{prop}_1 \subseteq \mathsf{prop}_2$ in ARMv7.

Consistency Constraints. A well-formed ARMv7 execution X is consistent when:

- $(\mathsf{X.po}_\ell \cup \mathsf{X.rf} \cup \mathsf{X.fr} \cup \mathsf{X.co})$ is acyclic. (sc-per-loc)
- $\mathsf{X.fre}; \mathsf{X.prop}; \mathsf{X.ahb}^*$ is irreflexive. (observation)
- $(\mathsf{X.co} \cup \mathsf{X.prop})$ is acyclic. (propagation)
- $\mathsf{X.rmw} \cap (\mathsf{X.fre}; \mathsf{X.coe}) = \emptyset$. (atomicity)
- $\mathsf{X.ahb}$ is acyclic. (no-thin-air)

3) ARMv8 Model

ARMv8 Specific Events.

- Acquire-read $\mathsf{A}(x, v)$ reads value v from location x with acquire mode.
- AcquirePC-read $\mathsf{Q}(x, v)$ reads value v from location x with acquirePC mode.
- release-write $\mathsf{L}(x, v)$ writes value v to location x with release mode.
- Fences: In additon to full fence F, ARMv8 provides load fence $\mathsf{F}_{\mathtt{LD}}$, store fence $\mathsf{F}_{\mathtt{ST}}$, and control fence `ISB` events.

Relations. ARMv8 defines various relations.

Relations among events.

$$\mathsf{A} \subseteq \mathsf{R}, \mathsf{Q} \subseteq \mathsf{R}, \mathsf{L} \subseteq \mathsf{W}$$

Dependency relations are same as ARMv7 except ISB is an event in ARMv8.

- $\mathsf{data} \subseteq \mathsf{R} \times \mathsf{W}$ captures the data dependencies of write events.
- $\mathsf{addr} \subseteq \mathsf{R} \times (\mathsf{R} \cup \mathsf{W})$ captures the address dependencies of memory access events.
- $\mathsf{ctrl} \subseteq \mathsf{R} \times \mathsf{E}$ captures the control dependencies of events. Moreover, $\mathsf{ctrl}; \mathsf{po} \subseteq \mathsf{ctrl}$ and $\mathsf{ctrl}_{\mathtt{ISB}}; \mathsf{po} \subseteq \mathsf{ctrl}_{\mathtt{ISB}}$ holds in ARMv7 model.

Relation coherence-after(ca) orders read or write to a fr-after or co-after write.

$$\mathsf{ca} \triangleq \mathsf{fr} \cup \mathsf{co}$$

Relation observed-by(obs) constitute of thread external fre, fre, and coe relations.

$$\mathsf{obs} \triangleq \mathsf{rfe} \cup \mathsf{coe} \cup \mathsf{fre}$$

Relation atomic-ordered-by(aob) is derived from rmw and rfi relations.

$$\mathsf{aob} \triangleq \mathsf{rmw} \cup [\mathsf{range}(\mathsf{rmw})]; \mathsf{rfi}; [\mathsf{A} \cup \mathsf{Q}]$$

Relation dependency-ordered-before(dob) captures dependency based ordering among a pair of events in a thread

$$\begin{aligned}\mathsf{dob} \triangleq\ & \mathsf{addr} \cup \mathsf{data} \cup \mathsf{ctrl}; [\mathsf{W}] \cup (\mathsf{ctrl} \cup (\mathsf{addr}; \mathsf{po})); [\mathtt{ISB}]; \mathsf{po}; [\mathsf{R}] \\ & \cup \mathsf{addr}; \mathsf{po}; [\mathsf{W}] \cup (\mathsf{ctrl} \cup \mathsf{data}); \mathsf{coi} \cup (\mathsf{addr} \cup \mathsf{data}); \mathsf{rfi}\end{aligned}$$

Relation barrier-ordered-by(bob) orders events by fences or results from stronger memory accesses.

$$\begin{aligned}\mathsf{bob} \triangleq\ & \mathsf{po}; [\mathsf{F}]; \mathsf{po} \cup [\mathsf{L}]; \mathsf{po}; [\mathsf{A}]; \cup [\mathsf{R}]; \mathsf{po}; [\mathsf{F}_{\mathrm{LD}}]; \mathsf{po} \\ & \cup [\mathsf{A} \cup \mathsf{Q}]; \mathsf{po} \cup [\mathsf{W}]; \mathsf{po}; [\mathsf{F}_{\mathrm{ST}}]; \mathsf{po}; [\mathsf{W}] \cup \mathsf{po}; [\mathsf{L}] \cup \mathsf{po}; [\mathsf{L}]; \mathsf{coi}\end{aligned}$$

Relation *ordered-before*(ob) is a transitive closure of obs, aob, dob, and bob relations.

$$\mathsf{ob} \triangleq (\mathsf{obs} \cup \mathsf{dob} \cup \mathsf{aob} \cup \mathsf{bob})^{+}$$

Consistency Constraints. A well-formed ARMv8 execution X is consistent when:

- $\mathsf{X}.\mathsf{po}_{\ell} \cup \mathsf{X}.\mathsf{ca} \cup \mathsf{X}.\mathsf{rf}$ is irreflexive. (internal)
- $\mathsf{X}.\mathsf{ob}$ is irreflexive. (external)
- $\mathsf{X}.\mathsf{rmw} \cap (\mathsf{X}.\mathsf{fre}; \mathsf{X}.\mathsf{coe}) = \emptyset$ (atomicity)

B. Proof of SC robustness against x86

In this case $R = \mathsf{xppo} \cup \mathsf{po}_\ell \cup \mathsf{implied}; \mathsf{po}^?$.

Proof. Both SC and x86 satisfies atomicity.

It remains to show $(\mathsf{X.po} \cup \mathsf{X.rf} \cup \mathsf{X.fr} \cup \mathsf{X.co})$ is acyclic by contradiction.

Assume $(\mathsf{X.po} \cup \mathsf{X.rf} \cup \mathsf{X.fr} \cup \mathsf{X.co})$ creates a cycle.

If the cycle has one or no epo edge then the cycle violates (sc-per-loc).

Otherwise, the cycle contains two or more epo edges.

It implies a $(\mathsf{X.epo}; \mathsf{X.eco})^+$ cycle.

It implies a $((\mathsf{X.xppo} \cup \mathsf{X.po}_\ell \cup \mathsf{X.implied}; \mathsf{X.po}^?); \mathsf{X.eco})^+$ cycle.

Considering incoming and outgoing eco edges to po_ℓ:

- $\mathsf{X.rfe}; [\mathsf{R}]; \mathsf{X.po}_\ell; [\mathsf{R}]; \mathsf{X.fre} \subseteq \mathsf{X.co}$
- $[\mathsf{W}]; \mathsf{X.po}_\ell; [\mathsf{R}]; \mathsf{X.fre} \subseteq \mathsf{X.co}$
- $\mathsf{X.rfe}; [\mathsf{R}]; \mathsf{X.po}_\ell; [\mathsf{W}] \subseteq \mathsf{X.co}$
- $(\mathsf{X.fre} \cup \mathsf{X.coe}); [\mathsf{W}]; \mathsf{X.po}_\ell; [\mathsf{W}] \subseteq \mathsf{X.fre} \cup \mathsf{X.coe}$

We also know that $\mathsf{co} = \mathsf{coi} \cup \mathsf{coe} \subseteq [\mathsf{W}]; \mathsf{po}_\ell; [\mathsf{W}] \cup \mathsf{eco}$. Therefore applying the above reductions we derive a $((\mathsf{X.xppo} \cup \mathsf{X.po}_\ell \cup \mathsf{X.implied}; \mathsf{X.po}^?); \mathsf{X.eco})^+$ cycle.

It implies a $((\mathsf{X.xppo} \cup \mathsf{X.implied}; \mathsf{X.po}^?); \mathsf{X.eco})^+$ cycle.

It implies a $((\mathsf{X.xppo} \cup \mathsf{X.implied}); \mathsf{X.eco})^+$ cycle by analyzing possible cases of $\mathsf{implied}; \mathsf{po}$.

However, this is not possible as X satisfies (GHB). Hence a contradiction and $(\mathsf{X.po} \cup \mathsf{X.rf} \cup \mathsf{X.fr} \cup \mathsf{X.co})$ is acyclic. □

C. SC, x86 robustness against ARMv8

1) Proof of SC robustness against ARMv8

In this case $R = \mathsf{po}_\ell \cup (\mathsf{aob} \cup \mathsf{dob} \cup \mathsf{bob})^+$.

Proof. Both SC and ARMv8 satisfies atomicity.

It remains to show $(\mathsf{X.po} \cup \mathsf{X.rf} \cup \mathsf{X.fr} \cup \mathsf{X.co})$ is acyclic by contradiction.

Assume $(\mathsf{X.po} \cup \mathsf{X.rf} \cup \mathsf{X.fr} \cup \mathsf{X.co})$ creates a cycle.

If the cycle has one or no epo edge then the cycle violates (sc-per-loc).

Otherwise, the cycle contains two or more epo edges.

It implies $(\mathsf{X.epo}; \mathsf{X.eco})^+$ creates a cycle.

It implies $((\mathsf{X.po}_\ell \cup (\mathsf{X.aob} \cup \mathsf{X.bob} \cup \mathsf{X.bob})^+); \mathsf{X.eco})^+$ creates a cycle.

Considering $\mathsf{X.po}_\ell$ with incoming and outgoing $\mathsf{X.eco}$, possible cases:

(1) $[\mathsf{R}]; \mathsf{X.po}_\ell; [\mathsf{R}]; \mathsf{X.fre}; [\mathsf{W}] \subseteq [\mathsf{R}]; \mathsf{X.fre}$

(2) $[\mathsf{W}]; \mathsf{X.po}_\ell; [\mathsf{R}]; \mathsf{X.fre}; [\mathsf{W}] \subseteq [\mathsf{W}]; \mathsf{X.coe}$

(3) $[\mathsf{W}]; \mathsf{X.rfe}; [\mathsf{R}]; \mathsf{X.po}_\ell; [\mathsf{W}] \subseteq [\mathsf{W}]; \mathsf{X.coe}$

(4) $(\mathsf{X.coe} \cup \mathsf{X.fre}); [\mathsf{W}]; \mathsf{X.po}_\ell; [\mathsf{W}] \subseteq \mathsf{X.coe} \cup \mathsf{X.fre}$

Therefore a $((\mathsf{X.po}_\ell \cup (\mathsf{X.aob} \cup \mathsf{X.bob} \cup \mathsf{X.bob})^+); \mathsf{X.eco})^+$ cycle implies $((\mathsf{X.aob} \cup \mathsf{X.bob} \cup \mathsf{X.bob})^+; \mathsf{X.eco})^+$ cycle.

It implies an $\mathsf{X.ob}$ cycle which violates (external) and therefore a contradiction. □

2) Proof of x86 robustness against ARMv8

In this case $R = \mathsf{po}_\ell \cup (\mathsf{aob} \cup \mathsf{bob} \cup \mathsf{dob})^+ \cup \mathsf{WR}$

Proof. Both x86 and ARMv8 satisfies atomicity.

It remains to show $(\mathsf{X.po} \cup \mathsf{X.rf} \cup \mathsf{X.fr} \cup \mathsf{X.co})$ is acyclic by contradiction.

Assume $\mathsf{X.xppo} \cup \mathsf{X.implied} \cup \mathsf{X.rfe} \cup \mathsf{X.fr} \cup \mathsf{X.co}$ edges create a cycle.

It implies a $\mathsf{X.xppo} \cup \mathsf{X.implied} \cup \mathsf{X.rfe} \cup \mathsf{X.fre} \cup \mathsf{X.coe}$ cycle as $\mathsf{fri} \cup \mathsf{coi} \subseteq \mathsf{xppo}$.

If the cycle has one or no epo edge then the cycle violates (sc-per-loc).

Otherwise, the cycle contains two or more epo edges.

It implies a $(\mathsf{X.epo} \cup \mathsf{X.eco})$ cycle where $\mathsf{epo} = (\mathsf{X.xppo} \cup \mathsf{X.implied})^+$.

Now consider the possible cases of epo:

Case $[\mathsf{R}]; \mathsf{X.epo}; [\mathsf{X.R}]$**:** Following the M-K-robustness condition it implies $\mathsf{X.po}_\ell \cup (\mathsf{X.aob} \cup \mathsf{X.dob} \cup \mathsf{X.bob})^+$.

Let $[\mathsf{R}]; \mathsf{X.epo}; [\mathsf{X.R}] \subseteq \mathsf{X.po}_\ell$. Considering the outgoing fre edge: $[\mathsf{R}]; \mathsf{X.po}_\ell; [\mathsf{R}]; \mathsf{X.fre} \subseteq \mathsf{X.fre}$.

It implies an alternative $(\mathsf{X.epo} \cup \mathsf{X.eco})$ cycle where all $[\mathsf{R}]; \mathsf{X.epo}; [\mathsf{X.R}] \subseteq (\mathsf{X.aob} \cup \mathsf{X.dob} \cup \mathsf{X.bob})^+$ hold.

Case $[\mathsf{R}]; \mathsf{X.epo}; [\mathsf{X.W}]$**:**

Following the M-K-robustness condition it implies $\mathsf{X.po}_\ell \cup (\mathsf{X.aob} \cup \mathsf{X.dob} \cup \mathsf{X.bob})^+$.

Let $[\mathsf{R}]; \mathsf{X.epo}; [\mathsf{X.W}] \subseteq \mathsf{X.po}_\ell$. Considering the incoming rfe edge: $[\mathsf{W}]; \mathsf{X.rfe}; [\mathsf{R}]; \mathsf{X.po}_\ell; [\mathsf{W}] \subseteq \mathsf{X.coe}$.

It implies an alternative $(\mathsf{X.epo} \cup \mathsf{X.eco})$ cycle where all $[\mathsf{R}]; \mathsf{X.epo}; [\mathsf{X.W}] \subseteq (\mathsf{X.aob} \cup \mathsf{X.dob} \cup \mathsf{X.bob})^+$ hold.

Case $[\mathsf{W}]; \mathsf{X.epo}; [\mathsf{X.W}]$**:**

Following the M-K-robustness condition it implies $\mathsf{X.po}_\ell \cup (\mathsf{X.aob} \cup \mathsf{X.dob} \cup \mathsf{X.bob})^+$.

Let $[\mathsf{W}]; \mathsf{X.epo}; [\mathsf{X.W}] \subseteq \mathsf{X.po}_\ell$. Considering the incoming fre edge: $[\mathsf{R}]; \mathsf{X.fre}; [\mathsf{W}]; \mathsf{X.po}_\ell; [\mathsf{W}] \subseteq \mathsf{X.fre}$.

It implies an alternative $(\mathsf{X.epo} \cup \mathsf{X.eco})$ cycle where all $[\mathsf{R}]; \mathsf{X.epo}; [\mathsf{X.W}] \subseteq (\mathsf{X.aob} \cup \mathsf{X.dob} \cup \mathsf{X.bob})^+$ hold.

Case $[\mathsf{W}]; \mathsf{X.epo}; [\mathsf{X.R}]$**:**

Following the M-K-robustness condition it implies $\mathsf{X.po}_\ell \cup (\mathsf{X.aob} \cup \mathsf{X.dob} \cup \mathsf{X.bob})^+ \cup \mathsf{WR}$.

Subcase $[\mathsf{W}]; \mathsf{X.epo}; [\mathsf{X.R}] \subseteq \mathsf{X.po}_\ell$**:** Considering the outgoing fre edge: $[\mathsf{W}]; \mathsf{X.po}_\ell; [\mathsf{R}]; \mathsf{X.fre} \subseteq \mathsf{X.coe}$.

It implies an alternative $(\mathsf{X.epo} \cup \mathsf{X.eco})$ cycle where all $[\mathsf{W}]; \mathsf{X.epo}; [\mathsf{X.R}] \subseteq (\mathsf{X.aob} \cup \mathsf{X.dob} \cup \mathsf{X.bob})^+ \cup \mathsf{WR}$ hold.

Subcase $[\mathsf{W}]; \mathsf{X.epo}; [\mathsf{X.R}] \subseteq \mathsf{X.WR}$**:**

We know that $[\mathsf{W}]; \mathsf{X.epo}; [\mathsf{X.R}] \subseteq (\mathsf{X.xppo} \cup \mathsf{X.implied})^+$.

We consider two possibilities:

- $[\mathsf{W}]; \mathsf{X.epo}; [\mathsf{X.R}]$ has no intermediate $\mathsf{X.implied}$.
  In this case $[\mathsf{W}]; \mathsf{X.epo}; [\mathsf{X.R}] \subseteq \mathsf{X.xppo}^+$. We know $\mathsf{xppo} \not\subseteq [\mathsf{W}]; \mathsf{po}; [\mathsf{R}]$. Moreover, $\mathsf{xppo}; \mathsf{xppo} \subseteq ([\mathsf{R}]; \mathsf{X.po} \cup \mathsf{X.po}; [\mathsf{W}]); ([\mathsf{R}]; \mathsf{X.po} \cup \mathsf{X.po}; [\mathsf{W}]) \not\subseteq [\mathsf{W}]; \mathsf{po}; [\mathsf{R}]$.

Hence $[W]; X.epo; [X.R] \not\subseteq X.xppo^+$.

- Otherwise.
  $[W]; X.epo; [X.R] \subseteq [W]; (X.xppo \cup X.implied)^*; X.implied; (X.xppo \cup X.implied)^*; [R]$
  Possible cases following the definition of WR:
  - $[W]; X.epo; [X.R] \subseteq [W]; X.po; [F]; po; [R]$. It implies $[W]; X.epo; [X.R] \subseteq X.bob$.
  - $[W]; X.epo; [X.R] \subseteq [W]; po; [dom(rmw)]$.
    It implies $[W]; X.epo; [X.R] \subseteq [W]; po; [F]; X.po; [R] \subseteq X.bob$.
  - $[W]; X.epo; [X.R] \subseteq [codom(rmw)]; X.po; [R]$.
    It implies $[W]; X.epo; [X.R] \subseteq [W]; po; [F]; X.po; [R] \subseteq X.bob$.

  Therefore $[W]; X.epo; [X.R] \subseteq (X.aob \cup X.dob \cup X.bob)^+$ hold.

It implies a $(X.aob \cup X.dob \cup X.bob)^+; X.eco)^+$ cycle.

It implies a X.ob cycle which is a contradiction.

Therefore X is x86-consistent and satisfies x86-ARMv8 robustness. □

D. SC, x86, ARMv8 robustness against ARMv7

1) Proof of SC robustness against ARMv7

In this case $R = po_\ell \cup fence$.

Proof. Both SC and ARMv7 satisfies atomicity.

It remains to show that $(X.po \cup X.rf \cup X.fr \cup X.co)$ is acyclic by contradiction.

Assume $(X.po \cup X.rf \cup X.fr \cup X.co)$ creates a cycle. If the cycle has one or no epo edge then the cycle violates (sc-per-loc). Otherwise, the cycle contains two or more epo edges.

It implies $(X.epo; X.eco)^+$ creates a cycle.

It implies $((X.po_\ell \cup X.fence); X.eco)^+$ creates a cycle.

Considering the incoming and outgoing edges for $X.po_\ell$:

- $[R]; X.po_\ell; [R]; X.fre \subseteq [R]; X.fre$
- $[W]; X.rfe; [R]; X.po_\ell; [W] \subseteq [W]; X.coe$
- $(X.fre \cup X.coe); [W]; X.po_\ell; [W] \subseteq (X.fre \cup X.coe)$
- $[W]; X.po_\ell; [R]; X.fre \subseteq [W]; X.coe$

It implies $(X.fence; X.eco)^+$ creates a cycle.

Now we consider $[codom(fence)]; X.eco; [dom(fence)]$ path.

Possible cases:

- $[R]; X.eco; [R]$: It implies $X.fre; X.rfe$.
- $[R]; X.eco; [W]$: It implies $X.fre$.
- $[W]; X.eco; [W]$: It implies $X.coe$.
- $[W]; X.eco; [R]$: It implies $X.coe; X.rfe$.

Thus an $(X.fence; X.eco)^+$ cycle implies

a $((X.coe \cup X.fre); X.rfe^?; X.fence)^+$ cycle.

It implies a $prop^+$ cycle which violates (propagation).

Hence a contradiction and therefore SC is preserved.

□

2) Proof of x86 robustness against ARMv7

In this case $R = po_\ell \cup fence \cup WR$.

Proof. Both x86 and ARMv7 satisfies atomicity.

It remains to show $(X.po \cup X.rf \cup X.fr \cup X.co)$ is acyclic by contradiction.

Assume $X.xppo \cup X.implied \cup X.rfe \cup X.fr \cup X.co$ edges create a cycle.

It implies a $X.xppo \cup X.implied \cup X.rfe \cup X.fre \cup X.coe$ cycle as $fri \cup coi \subseteq xppo$.

If the cycle has one or no epo edge then the cycle violates (sc-per-loc).

Otherwise, the cycle contains two or more epo edges.

It implies a $(X.epo \cup X.eco)$ cycle where $epo = (X.xppo \cup X.implied)^+$.

Now consider the possible cases of epo:

Case $[R]; X.epo; [X.R]$**:** Following the M-K-robustness condition it implies $[R]; X.epo; [X.R] \subseteq X.po_\ell \cup X.fence$.

Let $[R]; X.epo; [X.R] \subseteq X.po_\ell$. Considering the outgoing fre edge: $[R]; X.po_\ell; [R]; X.fre \subseteq X.fre$.

It implies an alternative $(X.epo \cup X.eco)$ cycle where all $[R]; X.epo; [X.R] \subseteq X.fence$ hold.

Case $[R]; X.epo; [X.W]$**:**

Following the M-K-robustness condition it implies $[R]; X.epo; [X.W] \subseteq X.po_\ell \cup X.fence$.

Let $[R]; X.epo; [X.W] \subseteq X.po_\ell$. Considering the incoming rfe edge: $[W]; X.rfe; [R]; X.po_\ell; [W] \subseteq X.coe$.

It implies an alternative $(X.epo \cup X.eco)$ cycle where all $[R]; X.epo; [X.W] \subseteq X.fence$ hold.

Case $[W]; X.epo; [X.W]$**:**

Following the M-K-robustness condition it implies $[W]; X.epo; [X.W] \subseteq X.po_\ell \cup X.fence$.

Let $[W]; X.epo; [X.W] \subseteq X.po_\ell$. Considering the incoming fre edge: $[R]; X.fre; [W]; X.po_\ell; [W] \subseteq X.fre$.

It implies an alternative $(X.epo \cup X.eco)$ cycle where all $[R]; X.epo; [X.W] \subseteq X.fence$ hold.

Case $[W]; X.epo; [X.R]$**:**

Following the M-K-robustness condition it implies $[W]; X.epo; [X.R] \subseteq X.po_\ell \cup \subseteq X.fence \cup WR$.

Subcase $[W]; X.epo; [X.R] \subseteq X.po_\ell$**:** Considering the outgoing fre edge: $[W]; X.po_\ell; [R]; X.fre \subseteq X.coe$.

It implies an alternative $(X.epo \cup X.eco)$ cycle where all $[W]; X.epo; [X.R] \subseteq (X.aob \cup X.dob \cup X.bob)^+ \cup WR$ hold.

Subcase $[W]; X.epo; [X.R] \subseteq X.WR$**:**

We know that $[W]; X.epo; [X.R] \subseteq (X.xppo \cup X.implied)^+$.

We consider two possibilities:

- $[W]; X.epo; [X.R]$ has no intermediate X.implied.
  In this case $[W]; X.epo; [X.R] \subseteq X.xppo^+$. We know $xppo \not\subseteq [W]; po; [R]$. Moreover, $xppo; xppo \subseteq ([R]; X.po \cup X.po; [W]); ([R]; X.po \cup X.po; [W]) \not\subseteq [W]; po; [R]$.

Hence $[\mathsf{W}]; \mathsf{X}.\mathsf{epo}; [\mathsf{X}.\mathsf{R}] \not\subseteq \mathsf{X}.\mathsf{xppo}^+$.

- Otherwise.
  $[\mathsf{W}]; \mathsf{X}.\mathsf{epo}; [\mathsf{X}.\mathsf{R}] \subseteq [\mathsf{W}]; (\mathsf{X}.\mathsf{xppo} \cup \mathsf{X}.\mathsf{implied})^*; \mathsf{X}.\mathsf{implied}; (\mathsf{X}.\mathsf{xppo} \cup \mathsf{X}.\mathsf{implied})^*; [\mathsf{R}]$
  Possible cases following the definition of WR:
  - $[\mathsf{W}]; \mathsf{X}.\mathsf{epo}; [\mathsf{X}.\mathsf{R}] \subseteq [\mathsf{W}]; \mathsf{X}.\mathsf{po}; [\mathsf{F}]; \mathsf{po}; [\mathsf{R}]$. It implies $[\mathsf{W}]; \mathsf{X}.\mathsf{epo}; [\mathsf{X}.\mathsf{R}] \subseteq \mathsf{X}.\mathsf{fence}$.
  - $[\mathsf{W}]; \mathsf{X}.\mathsf{epo}; [\mathsf{X}.\mathsf{R}] \subseteq [\mathsf{W}]; \mathsf{po}; [\mathsf{dom}(\mathsf{rmw})]$.
    It implies $[\mathsf{W}]; \mathsf{X}.\mathsf{epo}; [\mathsf{X}.\mathsf{R}] \subseteq [\mathsf{W}]; \mathsf{po}; [\mathsf{F}]; \mathsf{X}.\mathsf{po}; [\mathsf{R}] \subseteq \mathsf{X}.\mathsf{fence}$.
  - $[\mathsf{W}]; \mathsf{X}.\mathsf{epo}; [\mathsf{X}.\mathsf{R}] \subseteq [\mathsf{codom}(\mathsf{rmw})]; \mathsf{X}.\mathsf{po}; [\mathsf{R}]$.
    It implies $[\mathsf{W}]; \mathsf{X}.\mathsf{epo}; [\mathsf{X}.\mathsf{R}] \subseteq [\mathsf{W}]; \mathsf{po}; [\mathsf{F}]; \mathsf{X}.\mathsf{po}; [\mathsf{R}] \subseteq \mathsf{X}.\mathsf{fence}$.

  Therefore $[\mathsf{W}]; \mathsf{X}.\mathsf{epo}; [\mathsf{X}.\mathsf{R}] \subseteq \mathsf{X}.\mathsf{fence}$ hold.

It implies a $(\mathsf{X}.\mathsf{fence}; \mathsf{X}.\mathsf{eco})^+$ cycle.
It implies a $\mathsf{X}.\mathsf{prop}^+$ cycle which is a contradiction.
Therefore X is x86-consistent and satisfies x86-ARMv7 robustness. □

3) Proof of ARMv8 robustness against ARMv7

In this case $R = \mathsf{po}_\ell \cup [\mathsf{W}]; \mathsf{po} \cup \mathsf{fence}$.

Proof. We show X is ARMv8 consistent.
(internal)
Assume a $(\mathsf{X}.\mathsf{po}_\ell \cup \mathsf{X}.\mathsf{fr} \cup \mathsf{X}.\mathsf{co} \cup \mathsf{X}.\mathsf{rf})$ cycle.
However, X satisfies (sc-per-loc) and hence a contradiction.
Therefore, X satisfies (internal).

(external)
Assume a $\mathsf{X}.\mathsf{ob}$ cycle.
It implies $(\mathsf{X}.\mathsf{obs}; (\mathsf{X}.\mathsf{aob} \cup \mathsf{X}.\mathsf{bob} \cup \mathsf{X}.\mathsf{dob}))^+$ creates cycle.
From the definition,
$(\mathsf{X}.\mathsf{aob} \cup \mathsf{X}.\mathsf{bob} \cup \mathsf{X}.\mathsf{dob}) \subseteq \mathsf{po}_\ell \cup \mathsf{fence} \cup [\mathsf{W}]; \mathsf{po}$ and therefore
$((\mathsf{X}.\mathsf{rfe} \cup \mathsf{X}.\mathsf{coe} \cup \mathsf{X}.\mathsf{fre}); (\mathsf{X}.\mathsf{po}_\ell \cup \mathsf{fence}))^+$ creates cycle.
It implies prop creates a cycle which violates (propagation).
Therefore a contradiction and X satisfies (external).

(atomicity)
ARMv7 execution X satisfies (atomicity).

Therefore X has only ARMv8 execution.

□

E. Experimental Results

In this section we show the detailed experimental results on a set of concurrency benchmarks in Figs. 10 and 11.

Prog.	SC-x86		Trencher	
	result	⟨sec	result	⟨sec
Barrier	6|0✗2	⟨0.005	✗2	⟨0.004
Dekker-TSO	20|4✓0	⟨0.002	✓0	⟨0.007
Dekker-SC	20|0✗6	⟨0.013	✗4	⟨0.046
Peterson-RA	14|4✓0	⟨0.003	✓0	⟨0.007
Peterson-RA-b	21|0✗6	⟨0.002	✗2	⟨0.006
Peterson-RA-d	14|0✓0	⟨0.002	✓0	⟨0.006
Peterson-TSO	12|2✓0	⟨0.005	✓0	⟨0.009
Peterson-SC	14|0✗2	⟨0.004	✗2	⟨0.013
Lamport-RA	18|4✗2	⟨0.009	✓0	⟨0.017
Lamport-TSO	17|2✗2	⟨0.005	✓0	⟨0.014
Lamport-SC	17|0✗4	⟨0.019	✗4	⟨0.107
Spinlock	14|0✓0	⟨0.004	✓0	⟨0.007
Spinlock4	28|0✓0	⟨0.001	✓0	⟨0.006
Ticketlock	12|0✓0	⟨0.004	✓0	⟨0.006
Ticketlock4	24|0✓0	⟨0.003	✓0	⟨0.006
Seqlock	7|0✓0	⟨0.004	✓0	⟨0.582
NBW	20|0✓0	⟨0.009	✓0	⟨0.009
RCU	27|0✗4	⟨0.008	NA	
RCU-offline	33|4✗3	⟨0.038	✗-	⟨0.246
Cilk-TSO	22|2✓0	⟨0.011	✓0	⟨2.039
Cilk-SC	22|0✓0	⟨0.010	✗2	⟨6.322
Chaselev-RA	11|3✓0	⟨0.009	*	
Chaselev-TSO	13|1✗1	⟨0.008	✓0	⟨0.006
Chaselev-SC	13|0✗2	⟨0.007	*	

Prog.	ARMv8			
	SC		x86	
	result	⟨sec	result	⟨sec
Barrier	6|2✗2	⟨0.009	6|2✗0	⟨0.007
Dekker-TSO	20|8✗4	⟨0.007	20|8✗4	⟨0.011
Dekker-SC	20|4✗9	⟨0.008	20|4✗7	⟨0.003
Peterson-RA	14|4✗8	⟨0.006	14|4 ✗8	⟨0.005
Peterson-RA-b	24|18✗9	⟨0.035	24|18✗6	⟨0.027
Peterson-RA-d	16|6✗10	⟨0.011	16|6✗10	⟨0.008
Peterson-TSO	12|2✗6	⟨0.012	12|2✗6	⟨0.009
Peterson-SC	14|0✗11	⟨0.001	14|0 ✗10	⟨0.001
Lamport-RA	18|12✗5	⟨0.016	18|12 ✗5	⟨0.007
Lamport-TSO	17|9✗7	⟨0.013	17|9 ✗7	⟨0.008
Lamport-SC	17|7✗9	⟨0.007	17|7✗9	⟨0.008
Spinlock	18|12✗4	⟨0.017	18|12 ✗4	⟨0.009
Spinlock4	36|24✗8	⟨0.007	36|24 ✗8	⟨0.008
Ticketlock	14|8✗2	⟨0.006	14|8✗2	⟨0.007
Ticketlock4	28|16✗3	⟨0.001	28|16 ✗3	⟨0.002
Seqlock	9|6✗2	⟨0.002	9|6 ✗2	⟨0.005
NBW	24|6✗11	⟨0.011	24|6✗11	⟨0.007
RCU	27|10✗12	⟨0.010	27|10 ✗8	⟨0.008
RCU-offline	35|16✗17	⟨0.157	35|16 ✗19	⟨0.160
Cilk-TSO	33|10✗7	⟨0.025	33|10 ✗7	⟨0.024
Cilk-SC	33|8✗8	⟨0.011	33|8✗8	⟨0.012
Chaselev-RA	13|5✗1	⟨0.010	13|5✗1	⟨0.004
Chaselev-TSO	15|3✗7	⟨0.010	15|3 ✗7	⟨0.003
Chaselev-SC	15|2✗8	⟨0.011	15|2✗9	⟨0.003

Fig. 10: Robustness analyses and enforcement for x86 and ARMv8 programs. "-" indicates nontermination of the analysis and "*" indicates the analysis fails on the respective programs. NA indicates that the encoding is not available in [14].

<table>
<tr><th rowspan="3">Prog.</th><th colspan="6">ARMv7</th></tr>
<tr><th colspan="2">SC</th><th colspan="2">x86</th><th colspan="2">ARMv8</th></tr>
<tr><th>result</th><th>⟨ms</th><th>result</th><th>⟨ms</th><th>result</th><th>⟨ms</th></tr>
<tr><td>Barrier</td><td>6|2✗2</td><td>⟨0.012</td><td>6|2✓0</td><td>⟨0.002</td><td>6|2✓0</td><td>⟨0.002</td></tr>
<tr><td>Dekker-TSO</td><td>20|8✗6</td><td>⟨0.003</td><td>20|8✗6</td><td>⟨0.007</td><td>20|8✗6</td><td>⟨0.009</td></tr>
<tr><td>Dekker-SC</td><td>20|4✗10</td><td>⟨0.003</td><td>20|4✗10</td><td>⟨0.004</td><td>20|4✗10</td><td>⟨0.004</td></tr>
<tr><td>Peterson-RA</td><td>14|4✗8</td><td>⟨0.003</td><td>14|4✗8</td><td>⟨0.002</td><td>14|4✗8</td><td>⟨0.002</td></tr>
<tr><td>Peterson-RA-b</td><td>27|18✗6</td><td>⟨0.075</td><td>27|18 ✓3</td><td>⟨0.074</td><td>27|18 ✓2</td><td>⟨0.072</td></tr>
<tr><td>Peterson-RA-d</td><td>18|6✗10</td><td>⟨0.008</td><td>18|6✗10</td><td>⟨0.005</td><td>18|6✗8</td><td>⟨0.008</td></tr>
<tr><td>Peterson-TSO</td><td>12|2✗8</td><td>⟨0.003</td><td>12|2✗8</td><td>⟨0.001</td><td>12|2✗4</td><td>⟨0.003</td></tr>
<tr><td>Peterson-SC</td><td>14|0✗12</td><td>⟨0.002</td><td>14|0✗10</td><td>⟨0.002</td><td>14|0✗8</td><td>⟨0.003</td></tr>
<tr><td>Lamport-RA</td><td>18|12✗6</td><td>⟨3.137</td><td>18|12✗4</td><td>⟨3.108</td><td>18|12✗2</td><td>⟨3.105</td></tr>
<tr><td>Lamport-TSO</td><td>17|9✗8</td><td>⟨1.602</td><td>17|9✗6</td><td>⟨1.596</td><td>17|9✗3</td><td>⟨1.592</td></tr>
<tr><td>Lamport-SC</td><td>17|7✗10</td><td>⟨1.699</td><td>17|7✗8</td><td>⟨1.659</td><td>17|7✗5</td><td>⟨1.698</td></tr>
<tr><td>Spinlock</td><td>18|12✓0</td><td>⟨0.141</td><td>18|12 ✓0</td><td>⟨0.133</td><td>18|12✓0</td><td>⟨0.133</td></tr>
<tr><td>Spinlock4</td><td>36|24✓0</td><td>⟨0.729</td><td>36|24✓0</td><td>⟨0.699</td><td>36|24✓0</td><td>⟨0.742</td></tr>
<tr><td>Ticketlock</td><td>14|8✓0</td><td>⟨0.025</td><td>14|8✓0</td><td>⟨0.022</td><td>14|8✓0</td><td>⟨0.023</td></tr>
<tr><td>Ticketlock4</td><td>28|16✓0</td><td>⟨0.372</td><td>28|16✓0</td><td>⟨0.366</td><td>28|16✓0</td><td>⟨0.372</td></tr>
<tr><td>Seqlock</td><td>9|6✗2</td><td>⟨0.006</td><td>9|6✗2</td><td>⟨0.002</td><td>9|6✗2</td><td>⟨0.002</td></tr>
<tr><td>NBW</td><td>24|7✗9</td><td>⟨0.144</td><td>24|7✗9</td><td>⟨0.142</td><td>24|7✗8</td><td>⟨0.139</td></tr>
<tr><td>RCU</td><td>27|10✗14</td><td>⟨1.376</td><td>27|10✗14</td><td>⟨1.367</td><td>27|10✗8</td><td>⟨1.361</td></tr>
<tr><td>RCU-offline</td><td>36|19✗17</td><td>⟨0.335</td><td>36|19✗15</td><td>⟨0.334</td><td>36|19✗10</td><td>⟨0.339</td></tr>
<tr><td>Cilk-TSO</td><td>33|10✗6</td><td>⟨2.455</td><td>33|10✗6</td><td>⟨2.411</td><td>33|10✗6</td><td>⟨2.427</td></tr>
<tr><td>Cilk-SC</td><td>33|8✗7</td><td>⟨2.445</td><td>33|8✗7</td><td>⟨2.410</td><td>33|8✗7</td><td>⟨2.411</td></tr>
<tr><td>Chaselev-RA</td><td>15|5✗2</td><td>⟨ 0.013</td><td>15|5✗2</td><td>⟨ 0.002</td><td>15|5✗2</td><td>⟨ 0.003</td></tr>
<tr><td>Chaselev-TSO</td><td>17|3✗7</td><td>⟨ 0.012</td><td>17|3✗7</td><td>⟨ 0.001</td><td>17| 3 ✗7</td><td>⟨ 0.002</td></tr>
<tr><td>Chaselev-SC</td><td>17|2✗8</td><td>⟨ 0.012</td><td>17|2✗8</td><td>⟨ 0.001</td><td>17|2✗8</td><td>⟨ 0.002</td></tr>
</table>

Fig. 11: Robustness analyses and enforcement for ARMv7 programs.