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 na"ive 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 \( X = Y = 0 \).

\[
\begin{align*}
X &= 1; & Y &= 1; & (SB) & a &= X; & Y &= 1; & b &= Y; & X &= 1; & (LB)
\end{align*}
\]

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 7 and 8). 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, 8, 9, 10, 11, 12, 13, 14]. We propose the \( M \)-K robustness conditions in \$\text{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.

\[
\begin{align*}
\begin{array}{l}
a = T; \quad & X = 2; \quad b = X; \quad & \text{WP} \\
a = a; \quad & X = X; \quad & Z = c; \quad & Z = 1; \quad & d = Z; \quad & T = d;
\end{array}
\end{align*}
\]

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 [10], example in Fig. 5). Yet if all external-program-orders (see \$\text{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 $M$-robust based on the respective conditions. Otherwise we insert fences to enforce robustness. These analyses are computed in polynomial time as shown in §V unlike the robustness checkers which explore program execution 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 execution 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 $W$-robustness often requires fewer fences than ensuring $M$-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 $M$-robustness conditions. §IV explains our approach to check and enforce robustness. §V examines 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.

$$E ::= r \mid v \mid E + E \mid E * E \mid E \leq E \mid \cdots$$

$$C ::= \text{skip} \mid C \mid t = E \mid t = X \mid X = E \mid \text{RMW}(X, E, E) \mid C \mid \text{Fence}(\text{RMW}(X, E)) \mid \text{br label} \mid \text{br label label} \mid \cdots$$

$$P ::= x = v; \cdots X = v; \{C \parallel \cdots \parallel C\}$$

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 $\text{RMW}(X, E_r, E_w)$ 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 $\text{RMW}$ fails. Command $\text{RMW}(X, E_r)$ atomically updates the value of $X$ with the value of $E_r$ and returns the value of $X$ to $r$. A failed $\text{RMW}$ 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 execution. An execution consists of a set of events and relations.

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

**Preliminaries.** Given a binary relation $P$ on events, $\text{dom}(P)$ and $\text{codom}(P)$ are its domain and its range. $P^{-1}$, $P'$, $P^+$, and $P^*$ are inverse, reflexive, transitive, and reflexive-transitive closures of $P$ respectively. $P_t$ denotes $P$ related event pairs on same locations i.e. $P_t \triangleq \{ (e,e') \in P \mid e.loc = e'.loc \}$ and $P_{\#t} \triangleq P \setminus P_t$ denotes the $P$ related event pairs on different locations. $\text{imm}(P)$ defines the immediate $P$ relation, i.e. $\text{imm}(P) \triangleq \exists a, b. P(a,b) \land \forall c. P(a,c) \land R(c,b)$. $P ; S$ 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 $X = \langle E, po, rf, co \rangle$ where $X.E$ is the set of events in $X$. The set of po, rf, and co relations between the events in $X.E$ are $X.po, X.rf$, and $X.co$. Execution $X$ is well-formed if $X.po$ is total in each thread and every read reads-from some write, i.e. $X.R \subseteq \text{codom}(X.F)$.

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

$$Ld \triangleq R \setminus \text{dom}(\text{rmw}) \quad St \triangleq W \setminus \text{codom}(\text{rmw})$$

A successful $\text{RMW}$ generates an $\text{rmw}$ and a failed $\text{RMW}$ generates a $Ld$ event. We use $a \cdot b \triangleq \{ [a] ; \text{imm}(po) ; \{ [b] \} \}$ 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 $\text{rmw}$.

$$WR \triangleq ([| W |]_r ; po_{\#t} ; R) \setminus (po ; \text{rmw} ; po)$$

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, $fr \triangleq rf^{-1} ; co$. For example, in Fig. 1a the $R(X,0)$ and $W(X,1)$ 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.

\[ \text{rfi} \triangleq \text{rf} \land \text{po} \]
\[ \text{coe} \triangleq \text{co} \land \text{po} \]
\[ \text{fr} \triangleq \text{rf} \land \text{po} \]
\[ \text{fr} \triangleq \text{fr} \land \text{po} \]

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: eco \( \triangleq (\text{rfe} \cup \text{coe} \cup \text{fre})^+ \).

**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 \( W(x, v) \). A load access reads value \( v \) from \( x \) and generates an event with label \( R(x, v) \). A successful \( \text{RMW} \) on \( x \) reads value \( v' \) and writes value \( v \) to generate a pair of \( R(x, v') \) and \( W(x, v) \) events that are in \( \text{rmw} \) relation. A failed \( \text{RMW} \) generates an \( R(x, v') \) event. The full fences in x86, ARMv8, and ARMv7 are \( \text{MFENCE}, \text{DMFULL}, \) and \( \text{DMB} \) 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 ctrl123 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 L(x, v), A(x, v), and Q(x, v). ARMv8 also provide DMB and DMST fences that generate F_{LD} and F_{ST} events. Finally, L \( \subseteq W \), A \( \subseteq Q \), L \( \subseteq D \) \( \subseteq L \), 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 \( X, \text{po} \cup X, \text{rf} \cup X, \text{co} \cup X, \text{fr} \cup X, \text{fr} \) is acyclic.

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

**SC.** An well-formed execution \( X \) is SC when:

- \( X, \text{po} \cup X, \text{rf} \cup X, \text{fr} \cup X, \text{co} \) is acyclic (SC)
- \( X, \text{rmw} \cap (X, \text{fre} \cup X, \text{coe} \) is acyclic (atomicity)

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

\[ [X = Y = 0] \]
\[ W(x, 1) \]
\[ W(Y, 1) \]
\[ (a) \text{SB} \]
\[ [X = Y = 0] \]
\[ R(x, 1) \]
\[ R(Y, 1) \]
\[ (b) \text{LB} \]
\[ W(X[1], 1) \]
\[ W(Y[1], 1) \]
\[ R(X[1], 1) \]
\[ R(Y[1], 1) \]
\[ (c) \text{IRIW} \]

Fig. 1: Distinguishing 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.

- \( X, \text{po} \cup X, \text{rf} \cup X, \text{fr} \cup X, \text{co} \) is acyclic (sc-per-loc)
- \( X, \text{rmw} \cap (X, \text{fre} \cup X, \text{coe}) \) is acyclic (atomicity)
- \( X, \text{xhb} \) is acyclic where (GHB)
  - \( \text{xhb} \triangleq \text{xppo} \cup \text{fr} \cup \text{co} \)
  - \( \text{xppo} \triangleq ((W \times W) \cup (R \times W) \cup (R \times R)) \cap \text{po} \)
  - \( \text{implied} \triangleq \text{po} \cup \text{dom}(\text{rmw} \cup F) \cup \text{codom}(\text{rmw} \cup F) \cup \text{po} \)

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

\[ \text{bob} \triangleq \text{po} \cup F \cup \text{po} \cup R \cup \text{F}_{LD} \cup \text{po} \cup W \cup \text{po} \cup F_{ST} \cup \text{po} \cup W \cup \text{L} \cup \text{po} \cup [A] \cup \text{po} \cup [L] \cup \text{A} \cup \text{Q} \cup \text{po} \cup [L] \cup \text{coi} \]

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: ob \( \triangleq (\text{obs} \cup \text{dob} \cup \text{aob} \cup \text{bob})^+ \). A well-formed ARMv8 execution \( X \) is consistent when:

- \( X, \text{po} \cup X, \text{rf} \cup X, \text{fr} \cup X, \text{co} \) is acyclic (internal)
- \( X, \text{rmw} \cap (X, \text{fre} \cup X, \text{coe}) \) is acyclic (atomicity)
- \( X, \text{ob} \) is irreflexive (external)

These axioms allow the executions in Figs. [1a] and [1b] but disallows the execution in Fig. [1c] by the (external) axiom.
ARMv7. ARMv7 orders memory accesses in a thread by preserved-program-order (ppo) based on dependencies or fence ⊆ po; [F]; po relation. ARMv7 also defines happens-before (ahb) and propagation (prop ⊆ R; fence; R2) relations that can order events across threads. Finally a well-formed ARMv7 execution X is consistent when:

- \((X, \text{po}_X \cup X, \text{rf} \cup X, \text{fr} \cup X, \text{co})\) is acyclic. (sc-per-loc)
- \(X, \text{rmw} \cap (X, \text{fre}; X, \text{coe}) = \emptyset\) (atomicity)
- \(X, \text{fre}; X, \text{prop}; X, \text{ahb}\) is irreflexive. (observation)
- \((X, \text{co} \cup X, \text{prop})\) is acyclic. (propagation)
- \(X, \text{ahb}\) is acyclic. (no-thin-air)

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

ARMv7 allows the executions in Fig. 1 including [IRIV] with \(a = c = 1, b = d = 0\) outcome in the following program.

\[
\begin{align*}
X[1] = 1; & & a = X[1]; & & c = Y[1]; & & Y'[1] = 1; & & \text{(IRIV)}
\end{align*}
\]

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 coe edges between them. We define these po edges as external-program-order (epo) i.e. 

\[
\text{apo} \triangleq \text{po} \cap (\text{codom}(\text{coe}) \times \text{dom}(\text{coe})).
\]

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

**Theorem 1.** A program \(P\) is \(M-K\) robust if in all its \(K\)-consistent execution \(X, X, \text{apo} \subseteq X, R\) holds where \(R\) is defined as \(M-K\) condition as follows.

\[
\begin{align*}
\text{(SC-x86)} & \cup \text{apo} \cup \text{po}; \text{implied}; \text{po}^2 \\
\text{(SC-ARMv8)} & \cup \text{apo} \cup (\text{abo} \cup \text{dob} \cup \text{bob})^+ \\
\text{(x86-ARMv8)} & \cup \text{apo} \cup (\text{abo} \cup \text{bob} \cup \text{dob})^+ \cup \text{WR} \\
\text{(SC-ARMv7)} & \cup \text{po}; \text{fence} \\
\text{(x86-ARMv7)} & \cup \text{po}; \text{fence} \cup \text{WR} \\
\text{(ARMv8-ARMv7)} & \cup \text{po}; \text{fence} \cup \text{WR}
\end{align*}
\]

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 \(\text{ppo}\) orders read-read, read-write, and write-write pairs. So if an x86 execution violates SC-x86 robustness then it contains a \((\text{apo}; \text{coe})\) cycle with one or multiple \(\text{apo}\) edges that are in WR relation. If it is on same location then there is an alternative \((\text{eco}; \text{coe})\) cycle as shown in Fig. 3 that also denote the violation. The \(\text{implied}; \text{po}^2\) relation can order a write-read pair by intermediate \text{rmw} or \text{F}.

Consider the SB execution from Fig. 1a in x86. The \(\text{apo}\) 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 \((\text{apo}; \text{eco})^+\) cycle that violates SC-ARMv8 robustness. If an \(\text{apo}\) edge is on the cycle then as shown in Fig. 3 there is an alternative \((\text{eco}; \text{eco})^+\) cycle without the edge.

Now consider an \((\text{eco}; \text{eco})^+\) cycle where each \(\text{eco}\) on the cycle is in \((\text{abo} \cup \text{bob} \cup \text{dob})^+\) relation. In that case \((\text{abo} \cup \text{bob} \cup \text{dob})^+; \text{eco}^+\) cycle implies an ob cycle which is not possible as an ARMv8 consistent execution satisfies (external). The \(\text{apo}\) 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 \(\text{apo}\) 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 \(\text{apo}\) edges are also in WR and the execution is x86 consistent.
C. Robustness of ARMv7 Programs

SC-ARMv7 Robustness. The ARMv7 model uses \textit{ppo} and \textit{fence} relations to order \textit{epo} edges for SC-ARMv7 robustness.

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

x86-ARMv7 Robustness. To ensure x86-robustness, ARMv7 orders all \textit{epo} relations except write-read pairs. Consider the \textit{SB} program execution in Fig. 1a where the \textit{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 \textit{epo} relations except write-read and write-write pairs. In this case also \textit{ppo} relation cannot order \textit{epo} 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 \textit{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 \textit{epo} and \textit{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 \textit{K}-consistent execution of the program preserve \textit{M-K} robustness condition and as a result all \textit{K} consistent executions of these programs are also \textit{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 \textit{K} consistency model and in turn enforce \textit{M-K} robustness.

A. MPG Construction

Let \{\textit{f1}, \textit{f2}, ..., \textit{fn}\} be the set of thread functions in a program that may run in parallel. Let \(\langle V, E \rangle\) be a control flow graph (CFG) of a thread function where \(C.V\) are the instruction nodes and \(C.E\) are the set of control flow edges. We analyze the thread functions’ CFGs to construct an MPG.

Helper Definitions. We define following helper conditions.

- \(\text{CFG}(f)\) returns the control-flow-graph of a function \(f\).

- \(\text{mayAA}(i,j)\) checks if \(i\) and \(j\) may access same location.

- \(\text{ac}(C,A)\) returns the primitives in \(C\) which create \(A\) events or \textit{rmw} relations i.e. \(\text{ac}(C,A) \triangleq \{i \mid [i] \in A\}\). In this case \(\text{ac}(C, \textit{rmw})\) returns the accesses that create \textit{rmw} primitives.

- \(\mathcal{P}(C,i,j)\) checks if there is a path from \(i\) to \(j\) on the control flow graph \(C\) i.e. \(\mathcal{P}(C,i,j) \triangleq \{i,j \mid j \in [C.V] ; C.E^+ ; [C.V] \}\).

- \(\text{MM}(C)\) returns the set of memory access pairs in a control flow graph \(C\) where the second access is reachable from the first access. These pairs depict the potential \textit{epo} edges i.e. \(\text{MM}(C) \triangleq \{i,j \mid i,j \in \text{ac}(C,W \cup R) \land (P(C,i,j))\}\).

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

Procedure BuildG in Fig. 6 constructs an MPG. In BuildG line 2-4 append the memory access pairs from \(\text{CFG}(f_1), \text{CFG}(f_1), ..., \text{CFG}(f_n)\) to \(V\). Line 5-8 compute the \(G.E\) edges. An edge between \((a,b)\) and \((c,d)\) denotes that \(\text{mayAA}(b,c)\) holds. Note that we also create \(G.E\) 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 \(f_1 || \cdots || f_i || \cdots || f_n\).

B. Checking robustness on MPG

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

\[\text{Cy}(G) \triangleq \{n \mid n \in G.V \land \exists m,o \in G.V, \ m \neq n \land o \neq n \land G.E(m,n) \land G.E(n,o)\}\]

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

If \(\text{Cy}(G)\) has any unordered access pair following respective \textit{Ord} condition then we report \textit{M-K} robustness violation.
example. Consider the SB2 function in Fig. 5. The program SB2(true) || SB2(false) violates SC-x86 robustness due to an execution where R(Y, 0) and 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 Ord condition we use the following definitions.

- mustAA(i, j) checks if i and j always access same location.
- Procedure getG(i) returns the CFG C of instruction i.
- \(P_{nf}\) checks if there exist any path from i to j on the CFG 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.

\[ P_{nf}(C, i, j, F) \triangleq P((C \backslash F, C, E \backslash B), i, j) \]

- isW(i) and isR(i) check if the access i is write and read respectively.
- isWR(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 WR relation.

\[ isWR(C, i, j) \triangleq isW(i) \land isR(j) \land \neg mustAA(i, j) \land \exists u \in ac(C, rmw) \land P(C, i, u) \land P(C, u, j) \]

x86. The Ord condition for SC-x86 robustness is as follows.

\[ \text{Ord(SC, x86, C, i, j)} \triangleq \text{isR(i)} \lor \text{isW(j)} \lor \text{mustAA(i, j)} \]

The is(i) and isW(j) conditions ensure xpp relation between the events generated from i and mustAA(i, j) checks if i and j generated events pairs are in \(epo\) relation. The \(P_{nf}\) condition checks if there are intermediate fences between i and j generated events in all executions. The Ord condition is satisfied in \(SB\) and \(RIW\) but violated in the \(SB\) program.

In x86 a successful RMW results in 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 a 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.

\[ \text{SOrd(SC, x86, C, i, j)} \triangleq \text{isR(i)} \lor \text{isW(j)} \lor \text{mustAA(i, j)} \lor \neg P_{nf}(C, i, j, ac(C, F \cup rmw)) \]

2) Robustness Analysis and Enforcement Procedure

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

If \(O\) is empty then the program is \(M-K\) robust. Else Enforce procedure insert appropriate fences to enforce robustness. Procedure getF returns a fence based on the access type \(a\) and
We define the robustness algorithm as follows.

**Procedure:** BuildG \{f_1, \ldots, f_n\}

1. for \( f \in \{f_1, \ldots, f_n\} \) do
2. \( C \leftarrow \text{CFG}(f); \)
3. \( \forall \leftarrow \forall \cup \text{MM}(C); \)
4. for \((a, b) \in \forall \) do
5. \( (c, d) \in \forall \) do
6. if mayAA(b, c) then
7. \( E \leftarrow E \cup \{(a, b), (c, d)\}; \)
8. return \((V, E)\);
9. end procedure
10. \( G \leftarrow \text{BuildG}\{f_1, \ldots, f_n\}; \)
11. \( O \leftarrow \text{MKRobust}(M, K, G); \)
12. \( \text{Enforce}(K, O); \)

**Procedure:** getF(K, a, b)

1. if \( K == \text{x86} \) then return new(MFENCE);
2. if \( K == \text{A7} \) then return new(DMB);
3. if \( K == \text{A8} \) then
4. if isW(a) \& isR(b) then return new(DMBFULL);
5. if isW(a) \& isW(b) then return new(DMBST);
6. if isR(a) then return new(DMBLD);
7. end procedure

**Procedure:** insertF(C, a, b, f)

1. \( \forall' \leftarrow C, \forall \cup \{f\}; \)
2. \( E_1 \leftarrow C, E \cup \{(f, b)\}; \)
3. \( E' \leftarrow E_1 \cup \{(c, f) | C, E^+(c, b) \} \cup \{(f, c) | C, E^+(b, e) \}; \)
4. return \((\forall', E')\);
5. end procedure

**Procedure:** Enforce(K, O)

1. \( H \leftarrow \emptyset; \)
2. for \((a, b) \in O\) do
3. if \( b \notin H \) then
4. \( f \leftarrow \text{getF}(K, a, b); \)
5. insertF(getF(b), a, b, f);
6. \( H \leftarrow H \cup \{b\}; \)
7. end procedure

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

Fig. 7: Procedure getF and insertF.

### 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, \( P_n \), and \( P_m \) 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 [13, 27] including global barrier (Barrier) construct, mutual exclusion algorithms (by Dekker, Peterson, and Lamport), different lock algorithms (e.g., Spinlock, Spinlock, 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
the naive schemes as well as the results from state-of-the-art x86-robustness checker Trencher [8].

**Interpreting 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.
- ‘ヽ/ˊx’ 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 \((a-(b+c))\) 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 ( checkpoints), 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-SC, 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 ordered write-read pairs that generate TSO, and Cilk-SC programs. Lamport-(RA/TSO) have unverified results due to the barriers on the corresponding programs. Lamport-SC and 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.

**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-SC. These programs violate both SC-ARMv8 and x86-ARMv8 robustness due to unordered accesses that result in \(R:\{p\,d\}; \{R\} \) or \(W:\{p\,d\}; \{W\}\) relation in an execution. These access pairs are ordered in x86 but not in ARMv8 and hence violate x86-ARMv8 robustness.

---

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

**Fig. 9:** Robustness analyses & enforcement in ARMv8.
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].

\[
\begin{align*}
R(X) \cdot W_{SC}(Y) \cdot R(Z) & \rightarrow R(X) \cdot L(Y) \cdot R(Z) \quad (C-v8) \\
R(X) \cdot W_{SC}(Y) \cdot R(Z) & \rightarrow R(X) \cdot F \cdot W(Y) \cdot F \cdot R(Z) \quad (C-v7)
\end{align*}
\]

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 \( W(X) \cdot R(X) \cdot R(Y) \cdot W(Y) \) that may violate SC-ARMv8 and x86-ARMv8. To enforce SC-ARMv8 we insert a DMBFULL that results in \( W(X) \cdot R(X) \cdot F \cdot R(Y) \cdot W(Y) \) sequence. To ensure x86-ARMv8 we insert a DMBLD and a DMBST to generate \( W(X) \cdot R(X) \cdot F_{LD} \cdot R(Y) \cdot F_{ST} \cdot W(Y) \) sequence.

3) Performance of Robustness Analyses

We have already compared the execution times of SC-x86 robustness analysis in Fence and Trencher. In case of ARM program versions Fence incurs less than a second except for ARMv7 Cilk-(TSO/SC) programs. The timings of Fence 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 \( P_{nf} \) 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 [32,33,34,35,36,37] propose fence insertion to ensure SC. Among these fence insertion schemes our approach is closer to static approaches [34,35,36,37] 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 [35,38,34] 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
