Robustness between Weak Memory Models

Soham Chakraborty

EEMCS, TU Delft

FMCAD 2021

What is Weak Memory Model?

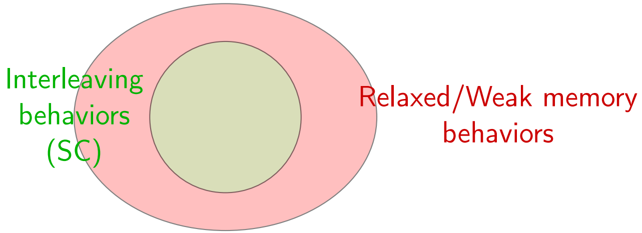
Traditionally: concurrency = thread interleaving

Interleaving
behaviors
(SC)

What is Weak Memory Model?

Traditionally: concurrency = thread interleaving

Reality: more behaviors than thread interleaving



Example: SB

$$X = Y = 0;$$

$$\begin{array}{l|l} (1)\ X = 1; & (3)\ Y = 1; \\ (2)\ a = Y; & (4)\ b = X; \end{array}$$

Behaviors:	SC	Interleavings
$a = 1, b = 1$	✓	1-3-2-4, 3-1-4-2, ...
$a = 0, b = 1$	✓	1-2-3-4
$a = 1, b = 0$	✓	3-4-1-2
$a = 0, b = 0$	✗	-

Example: SB

$$X = Y = 0;$$

$$\begin{array}{l} (1)\ X = 1; \\ (2)\ a = Y; \end{array} \;\Big\|\; \begin{array}{l} (3)\ Y = 1; \\ (4)\ b = X; \end{array}$$

Behaviors:	SC	x86
$a = 1, b = 1$	✓	✓
$a = 0, b = 1$	✓	✓
$a = 1, b = 0$	✓	✓
$a = 0, b = 0$	✗	✓

Example: SB+mfences

$$X = Y = 0;$$

$X = 1;$	$Y = 1;$
MFENCE;	MFENCE;
$a = Y;$	$b = X;$

Behaviors:	SC	x86
$a = 1, b = 1$	✓	✓
$a = 0, b = 1$	✓	✓
$a = 1, b = 0$	✓	✓
$a = 0, b = 0$	✗	✗

Examples: SB+mfences and SB

$X = Y = 0;$

$X = 1;$ MFENCE; $a = Y;$	$Y = 1;$ MFENCE; $b = X;$

Behaviors:	SC	x86
$a = 1, b = 1$	✓	✓
$a = 0, b = 1$	✓	✓
$a = 1, b = 0$	✓	✓
$a = 0, b = 0$	✗	✗

$X = Y = 0;$

$X = 1;$ $a = Y;$	$Y = 1;$ $b = X;$

Behaviors:	SC	x86
$a = 1, b = 1$	✓	✓
$a = 0, b = 1$	✓	✓
$a = 1, b = 0$	✓	✓
$a = 0, b = 0$	✗	✓

Some (not all) programs exhibit additional behaviors on weaker models

Checking (SC) Robustness

Check: For a given program P, and a memory model K:
Does running P on K have *extra behavior* w.r.t. SC?

Checking (SC) Robustness

Check: For a given program P, and a memory model $K = x86$:
Does running P on K have *extra behavior* w.r.t. SC?

Example:

$$X = Y = 0;$$

$$\begin{array}{l||l} X = 1; & Y = 1; \\ a = Y; & b = X; \end{array}$$

Violates SC-robustness

Checking and Enforcing (SC) Robustness

Check: For a given program P, and a memory model $K = x86$:
Does running P on K have *extra behavior* w.r.t. SC?

Enforce (if program P violates SC-robustness on K):
Transform P to P' such that P' is SC-robust.

Example:

$$X = Y = 0; \qquad\qquad\qquad\qquad X = Y = 0;$$

$$\begin{array}{l\|l} X = 1; & Y = 1; \\ a = Y; & b = X; \end{array} \quad \Rightarrow \quad \begin{array}{l\|l} X = 1; & Y = 1; \\ \text{MFENCE;} & \text{MFENCE;} \\ a = Y; & b = X; \end{array}$$

Violates SC-robustness   Enforce SC-robustness

Checking and Enforcing (SC) Robustness

Check: For a given program P, and a memory model $K = x86$:
Does running P on K have *extra behavior* w.r.t. SC?

Enforce (if program P violates SC-robustness on K):
Transform P to P' such that P' is SC-robust.

Example:

$$\begin{array}{c} X = Y = 0; \\ \begin{array}{l\|l} X = 1; & Y = 1; \\ a = Y; & b = X; \end{array} \end{array} \quad \Rightarrow \quad \begin{array}{c} X = Y = 0; \\ \begin{array}{l\|l} X = 1; & Y = 1; \\ \text{MFENCE;} & \text{MFENCE;} \\ a = Y; & b = X; \end{array} \end{array}$$

Violates SC-robustness

Enforce SC-robustness

Enable translation of a program from model K to SC

This Paper

Checking and enforcing robustness of
x86 and ARM (Version 8 and 7) concurrent programs

SC-Robustness

For a given program P, and a memory model K:
Does running P on K have *extra behavior* w.r.t. SC?

$$\Downarrow$$

M-K Robustness

For a given program P, and two memory models M and K: Does running P on K have *extra behavior* w.r.t. M?

M-K Robustness

For a given program P, and two memory models M and K: Does running P on K have *extra behavior* w.r.t. M?

Existing approaches: M=sequential consistency (SC)

$\downarrow$ *M*-*K* $\rightarrow$	x86	ARMv8	ARMv7
SC	✓	?	?
x86	-	?	?
ARMv8	-	-	?

M-K Robustness

For a given program P, and two memory models M and K: Does running P on K have *extra behavior* w.r.t. M?

Existing approaches: M=sequential consistency (SC)

$\downarrow$ *M*-*K* $\rightarrow$	x86	ARMv8	ARMv7
SC	✓	✓	✓
x86	-	?	?
ARMv8	-	-	?

M-K Robustness

For a given program P, and two memory models M and K: Does running P on K have *extra behavior* w.r.t. M?

Existing approaches: M=sequential consistency (SC)

$\downarrow$ M-K $\rightarrow$	x86	ARMv8	ARMv7
SC	✓	✓	✓
x86	-	✓	✓
ARMv8	-	-	✓

x86 to ARM Translation

$$X = Y = 0;$$

$$\begin{array}{l||l} X = 1; & Y = 1; \\ a = Y; & b = X; \end{array}$$

Behaviors:	SC	x86	ARM
$a = 1, b = 1$	✓	✓	✓
$a = 0, b = 1$	✓	✓	✓
$a = 1, b = 0$	✓	✓	✓
$a = 0, b = 0$	✗	✓	✓

The program is x86-ARM robust

SC-robustness for ARM is too strong for x86 to ARM translation

- The inserted DMBFULL fences are redundant

x86 to ARM Translation

$X = Y = 0;$

$X = 1;$	$Y = 1;$
DMBFULL	DMBFULL
$a = Y;$	$b = X;$

Behaviors:	SC	x86	ARM
$a = 1, b = 1$	✓	✓	✓
$a = 0, b = 1$	✓	✓	✓
$a = 1, b = 0$	✓	✓	✓
$a = 0, b = 0$	✗	-	✗

The program is x86-ARM robust

SC-robustness for ARM is too strong for x86 to ARM translation

- The inserted DMBFULL fences are redundant

M-K Robustness

For a given program P, and two memory models M and K: Does running P on K have *extra behavior* w.r.t. M?

Existing approaches: M=sequential consistency (SC)

$\downarrow$ *M-K* $\rightarrow$	x86	ARMv8	ARMv7
SC	✓	✓	✓
x86	-	✓	✓
ARMv8	-	-	✓

Proposed Approach:

1. Identify *M-K* robustness conditions
2. Statically analyze if a program is M-K robust
3. If not: Insert appropriate fences to enforce robustness

SB Execution Graph

$$X = Y = 0;$$

$$\begin{array}{l|l} X = 1; & Y = 1; \\ a = Y; & b = X; \end{array}$$

Outcome:

$a = b = 0$

SB Execution Graph

$$X = Y = 0;$$

$$\begin{array}{l||l} X = 1; & Y = 1; \\ a = Y; & b = X; \end{array}$$

Outcome:
$a = b = 0$

$$\begin{tikzcd}
 & {[X = Y = 0]} \arrow[dl, dotted, bend right, "\text{co}"'] \arrow[dl] \arrow[dr, dotted, bend left, "\text{co}"] \arrow[dr] \arrow[ddl, dashed, "\text{rf}"] \arrow[ddr, dashed, "\text{rf}"'] & \\
\mathrm{W}(X,1) \arrow[dd, "\text{po}"'] & & \mathrm{W}(Y,1) \arrow[dd, "\text{po}"] \\
 & & \\
\mathrm{R}(Y,0) \arrow[uurr, dashed] & \text{fre} & \mathrm{R}(X,0) \arrow[uull, dashed]
\end{tikzcd}$$

po: program order

rf: reads-from

co: coherence-order

fr: from-read

rfe: external-reads-from

coe: external-coherence-order

fre: external-from-read

SB Execution Graph

$$X = Y = 0;$$

$$\begin{array}{l||l} X = 1; & Y = 1; \\ a = Y; & b = X; \end{array}$$

$$a = b = 0$$

$$\begin{tikzcd}
 & {[X = Y = 0]} \arrow[dl] \arrow[dr] & \\
\mathrm{W}(X, 1) \arrow[d, "\mathrm{po}"'] & & \mathrm{W}(Y, 1) \arrow[d, "\mathrm{po}"'] \\
\mathrm{R}(Y, 0) \arrow[urr, dashed] & \mathrm{fre} & \mathrm{R}(X, 0) \arrow[ull, dashed]
\end{tikzcd}$$

SC-robustness violation by $\mathrm{po} \cup \mathrm{fre}$ cycle

Some Definitions

$$X = Y = 0;$$

$$\begin{array}{l||l} X = 1; & Y = 1; \\ a = Y; & b = X; \end{array}$$

$$a = b = 0$$

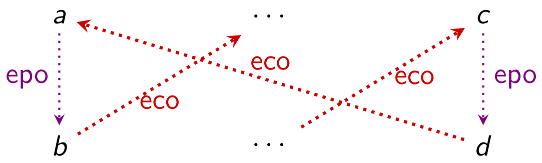
$$\begin{tikzcd}
 & {[X = Y = 0]} \arrow[dl] \arrow[dr] & \\
\mathrm{W}(X,1) \arrow[d, dotted, "\mathrm{epo}"'] & & \mathrm{W}(Y,1) \arrow[d, dotted, "\mathrm{epo}"'] \\
\mathrm{R}(Y,0) \arrow[urr, dotted, "\mathrm{eco}"] & & \mathrm{R}(X,0) \arrow[ull, dotted]
\end{tikzcd}$$

- $\mathrm{eco} = (\mathrm{rfe} \cup \mathrm{coe} \cup \mathrm{fre})^{+}$ and
- $\mathrm{epo} = \mathrm{po} \cap (\mathrm{codom}(\mathrm{eco}) \times \mathrm{dom}(\mathrm{eco}))$

Proposed Technique

An axiom violation implies a cycle on the execution graph

An axiom violating cycle is of the form:



where at least one epo is *unordered*

***M*-*K* Robustness violating cycle:**

allowed in model *K* but disallowed in model *M*

Enforcing Robustness: Breaking the Cycle

Enforce ordering on epo edges

Possible ways to order memory access pairs in architectures:

- Memory accesses are ordered
- Preserved-program-orders based on dependencies
- Same location memory accesses
- Intermediate fences

Orderings in Model K

Model K $\Rightarrow$ $\Downarrow$ Ordering constraints	x86	ARMv8	ARMv7
Regular Memory accesses	✓		
synchronizing memory accesses	-		
Dependency based ordering	-		
Same location access pairs	✓		
Intermediate fences	✓		

Orderings in Model K

Model K $\Rightarrow$ $\Downarrow$ Ordering constraints	x86	ARMv8	ARMv7
Regular Memory accesses	✓	✗	
synchronizing memory accesses	-	✓	
Dependency based ordering	-	✓	
Same location access pairs	✓	✓	
Intermediate fences	✓	✓	

Orderings in Model K

Model K $\Rightarrow$ $\Downarrow$ Ordering constraints	x86	ARMv8	ARMv7
Regular Memory accesses	✓	✗	✗
synchronizing memory accesses	-	✓	-
Dependency based ordering	-	✓	✓
Same location access pairs	✓	✓	✗
Intermediate fences	✓	✓	✓

ARMv7 Concurrency

Same location read-write accesses are not always ordered

$$X = Y = 0;$$

$$\begin{array}{l|l|l} a = X; & & \\ & Y = X; & X = Y; \\ X = 1; & & \end{array}$$

ARMv7 allows the following execution

$$\begin{tikzcd}
\mathrm{R}(X,1) \arrow[d, "\mathrm{po}_\ell"'] & \mathrm{R}(X,1) \arrow[d, "\mathrm{data}"] & \mathrm{R}(Y,1) \arrow[d, "\mathrm{data}"] \\
\mathrm{W}(X,1) \arrow[ur, dashed, "\mathrm{rfe}"'] & \mathrm{W}(Y,1) \arrow[ur, dashed, "\mathrm{rfe}"'] & \mathrm{W}(X,1) \arrow[ull, dashed, "\mathrm{rfe}"]
\end{tikzcd}$$

Yet po_ℓ is included in SC-ARMv7 condition

ARMv7 Concurrency

Dependencies are not strong enough relation (unlike ARMv8)

$$X = T; \parallel X = 2; \parallel Y = X; \parallel Z = Y; \parallel Z = 1; \parallel T = Z;$$

ARMv7 allows the following execution

$$\begin{tikzcd}
\mathrm{R}(T,1) \arrow[dd] & & \mathrm{R}(X,2) \arrow[dd] & \mathrm{R}(Y,2) \arrow[dd] & & \mathrm{R}(Z,1) \arrow[dd] \\
& \mathrm{W}(X,2) \arrow[ur, dashed] & & & \mathrm{W}(Z,1) \arrow[ur, dashed] & \\
\mathrm{W}(X,1) \arrow[ur, dashed] & & \mathrm{W}(Y,2) \arrow[uur, dashed] & \mathrm{W}(Z,2) \arrow[ur, dashed] & & \mathrm{W}(T,1) \arrow[uulllll, dashed]
\end{tikzcd}$$

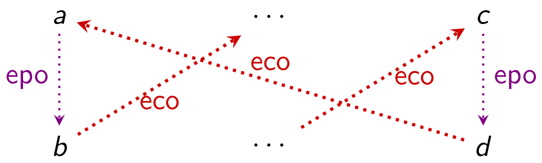
The execution is NOT SC-ARMv7 even if all epo edges are ppo

Robustness Conditions

Conditions for M-K Robustness: all epo edges are ordered

Model K $\Rightarrow$ / $\Downarrow$ Ordering constraints	x86	ARMv8	ARMv7
Regular Memory accesses	✓	✗	✗
synchronizing memory accesses	-	✓	-
Dependency based ordering	-	✓	✓ ✗
Same location access pairs	✓	✓	✗ ✓
Intermediate fences	✓	✓	✓

Static Robustness Checking



Static checking of the semantic robustness property

Steps:

- Identify program components which may run concurrently
  - Thread functions which may create multiple threads

Static Robustness Checking

$$\begin{tikzcd}
a \arrow[d, dotted, "\mathsf{epo}"'] & \cdots & c \arrow[d, dotted, "\mathsf{epo}"] \\
b \arrow[ru, dotted, "\mathsf{eco}"] & \cdots \arrow[ru, dotted, "\mathsf{eco}"] & d \arrow[llu, dotted, "\mathsf{eco}"']
\end{tikzcd}$$

Static checking of the semantic robustness property

Steps:

- Identify program components which may run concurrently

- Construct memory-access pair graph (MPG)

Static Robustness Checking

$$\begin{tikzcd}
a \arrow[d, dotted, "\text{epo}"'] & \cdots & c \arrow[d, dotted, "\text{epo}"] \\
b \arrow[ur, dotted, "\text{eco}"'] & \cdots \arrow[ur, dotted, "\text{eco}"] & d \arrow[ull, dotted, "\text{eco}"']
\end{tikzcd}$$

Static checking of the semantic robustness property

Steps:

- Identify program components which may run concurrently

- Construct memory-access pair graph (MPG)

- Identify the access pairs on the cycle

Static Robustness Checking

$$\begin{tikzcd}
a \arrow[d, dotted, "\text{epo}"'] & \cdots & c \arrow[d, dotted, "\text{epo}"] \\
b \arrow[ur, dotted, "\text{eco}"'] & \cdots \arrow[ur, dotted, "\text{eco}"] & d \arrow[ull, dotted, "\text{eco}"']
\end{tikzcd}$$

Static checking of the semantic robustness property

Steps:

- Identify program components which may run concurrently

- Construct memory-access pair graph (MPG)

- Identify the access pairs on the cycle

- Check if any access pair on the cycle may create an unordered epo

Implementation and Experiments

Fency: a tool for static robustness analysis and enforcement

- x86, ARMv8, ARMv7 programs
- Based on LLVM code generation phase
- Parameterized programs

Experiments

- Several concurrent data structures and algorithms
- Compared to Trencher: an existing SC-x86 robustness analyzer

Fency vs Naive for SC-x86 robustness

Fency ensures SC-x86 robustness with less fences

Prog.	Naive	Fency
Barrier	6	2
Dekker-TSO	20	4
Peterson-SC	14	2
Lamport-SC	17	4
Spinlock	14	0
Ticketlock	12	0
Seqlock	7	0
RCU-offline	33	7
Cilk-TSO	22	2
Cilk-SC	22	0

Fency vs Trencher for SC-x86 robustness

Checking results and # inserted fences

Prog.	Fency		Trencher	
Barrier	✗	2	✗	2
Dekker-TSO	✓	0	✓	0
Peterson-SC	✗	2	✗	2
Lamport-SC	✗	4	✗	4
Spinlock	✓	0	✓	0
Ticketlock	✓	0	✓	0
Seqlock	✓	0	✓	0
RCU-offline	✗	3	✗	-
Cilk-TSO	✓	0	✓	0
Cilk-SC	✓	0	✗	2

Analysis time: Fency vs Trencher

Prog.	Fency		Trencher	
	result	⟨seconds	result	⟨seconds
Barrier	✗ |2	⟨0.005	✗ |2	⟨0.004
Dekker-TSO	✓ |0	⟨0.002	✓ |0	⟨0.007
Peterson-SC	✗ |2	⟨0.004	✗ |2	⟨0.013
Lamport-SC	✗ |4	⟨0.019	✗ |4	⟨0.107
Spinlock	✓ |0	⟨0.004	✓ |0	⟨0.007
Ticketlock	✓ |0	⟨0.004	✓ |0	⟨0.006
Seqlock	✓ |0	⟨0.004	✓ |0	⟨0.582
RCU-offline	✗ |3	⟨0.038	✗ |-	⟨0.246
Cilk-TSO	✓ |0	⟨0.011	✓ |0	⟨2.039
Cilk-SC	✓ |0	⟨0.010	✗ |2	⟨6.322

Other Observations from Empirical Evaluation

Most of the ARM (8 and 7) programs violate robustness criteria

- Independent memory access pairs are unordered

Other Observations from Empirical Evaluation

Most of the ARM (8 and 7) programs violate robustness criteria

- Independent memory access pairs are unordered

Enforcing non-SC robustness often requires less fences than enforcing SC-robustness.

- Robustness analyses between weak memory models are useful !

Conclusion and Future Work

Robustness analysis and enforcement

- x86, ARMv8, ARMv7 programs

Fency: static robustness checking and enforcement

Available at:

https://www.st.ewi.tudelft.nl/sschakraborty/Fency-FMCAD21.zip

Going forward:

- New architectures, features, precise and scalable analysis tools

Conclusion and Future Work

Robustness analysis and enforcement

- x86, ARMv8, ARMv7 programs

Fency: static robustness checking and enforcement

Available at:

https://www.st.ewi.tudelft.nl/sschakraborty/Fency-FMCAD21.zip

Going forward:

- New architectures, features, precise and scalable analysis tools

Thank you !