

Compositional Reasoning for WMMs

COV889 Course Presentation

Ramneet Singh
Mrunmayi Bhalerao

IIT Delhi

May 2023

- 1 Introduction
- 2 Abstract Language
- 3 Basic Proof System
- 4 Multicopy Atomic Memory Models
- 5 Non-Multicopy Atomic Memory Models

Section 1

Introduction

Rely/Guarantee Reasoning

- Verification of concurrent programs with shared resources is challenging due to **combinatorial explosion**
- **Abstraction to the rescue!**
- Everything the environment can do: \mathcal{R}
- Everything you can do: \mathcal{G}

$$\mathcal{R}, \mathcal{G} \vdash P\{c\}Q$$

- *Compositional!*

Extension to Weak Memory Models

- Judgements using earlier techniques are valid under **sequentially consistent semantics**
 - Can be directly used for data-race free code executing on weak memory models
 - But, lots of code has data races! `seqlock`,
`java.util.concurrent.ConcurrentLinkedQueue` ...
- How do we extend them to weak memory models?
- What If: We could find a condition under which sequentially consistent rely-guarantee reasoning can be *soundly* preserved

$$(\vdash P\{c\}Q) \wedge ?? \implies \vdash P\{c_{WM}\}Q$$

- Benefits:
 - Reuse existing verification techniques
 - Deal with the complexity of weak memory separately as a *side-condition*

What are WMMs anyway?

- Relaxing the memory consistency guarantees provided by hardware enables optimisations
 - Store forwarding (will see later)
 - Write buffers
- (Part 1) **Multicopy Atomic**: One thread's stores become observable to all other threads at the same time.
 - x86-TSO, ARMv8, RISC-V
- (Part 2) **Non-Multicopy Atomic**: Each component has its own *view* of the global memory.
 - Older ARM versions, POWER, C11
- Challenge: Two types of interference now – **Inter-Thread** + **Intra-Thread** (due to reordering)
- How will we deal with this? ...

Teaser

- We want a compositional approach through thread-local reasoning.
- Exploit the reordering semantics of Colvin and Smith: **multicopy atomic memory models can be captured in terms of instruction reordering.**
 - Combinatorial explosion? (n reorderable instructions in a thread $\implies n!$ behaviours)
 - Introduce *reordering interference freedom* between $(\frac{n(n-1)}{2})$ pairs of instructions (Stay tuned...)
- In non-multicopy atomic WMMs, there is **no global shared state(!)**
 - Judgement for each thread is applicable to *its view* (depends on *propagation* of writes by hardware)
 - How do we know it holds in other threads' views?
 - Represent the semantics using reordering **between** different threads
 - No longer compositional? Hardest part of the talk – *global reordering interference freedom*: use the rely abstraction to represent reorderings between threads

Section 2

Abstract Language

Syntax

- Individual (atomic) instructions α
- Commands (or programs)

$$c := \epsilon \mid \alpha \mid c_1; c_2 \mid c_1 \sqcap c_2 \mid c^* \mid c_1 \parallel c_2$$

- Iteration, choice are non-deterministic
- Empty program ϵ represents termination

Semantics: Configurations

- *Configuration* (c, σ) of a program
 - Command c to be executed
 - State σ (map from variables to values)
- *Action Step*: Performed by component, changes state

$$(c, \sigma) \xrightarrow{as} (c', \sigma') \iff \exists \alpha. c \mapsto_{\alpha} c' \wedge (\sigma, \sigma') \in \text{beh}(\alpha)$$

- *Silent Step*: Performed by component, doesn't change state

$$(c_1 \sqcap c_2, \sigma) \rightsquigarrow (c_1, \sigma) \quad (c_1 \sqcap c_2, \sigma) \rightsquigarrow (c_2, \sigma)$$

$$(c^*, \sigma) \rightsquigarrow (\epsilon, \sigma) \quad (c^*, \sigma) \rightsquigarrow (c; c^*, \sigma)$$

- *Program Step*: Action Step or Silent Step
- *Environment Step*: Performed by environment, changes state.
 $(c, \sigma) \xrightarrow{es} (c, \sigma')$

Semantics: Commands

- Each atomic instruction α has a relation $\text{beh}(\alpha)$ (over pre- and post-states) specifying its behaviour
- Program execution is defined by a small-step semantics over commands
- Iteration, non-deterministic choice are dealt with at a higher level (see next slide)

$$\frac{}{\alpha \mapsto_{\alpha} \epsilon}$$

$$\frac{c_1 \mapsto_{\alpha} c'_1}{c_1; c_2 \mapsto_{\alpha} c'_1; c_2}$$

$$\frac{c_1 \mapsto_{\alpha} c'_1}{c_1 \parallel c_2 \mapsto_{\alpha} c'_1 \parallel c_2}$$

$$\frac{c_2 \mapsto_{\alpha} c'_2}{c_1 \parallel c_2 \mapsto_{\alpha} c_1 \parallel c'_2}$$

Section 3

Basic Proof System

Definitions

- Associate a verification condition $vc(\alpha)$ with each instruction α :
Provides finer-grained control (just set to \top if not needed)

- Hoare triple

$$P\{ \alpha \}Q \stackrel{\text{def}}{=} P \subseteq vc(\alpha) \cap \{ \sigma \mid \forall \sigma', (\sigma, \sigma') \in \text{beh}(\alpha) \implies \sigma' \in Q \}$$

- A rely-guarantee pair $(\mathcal{R}, \mathcal{G})$ is well-formed if

- \mathcal{R} is reflexive and transitive
- \mathcal{G} is reflexive

- Stability of predicate P under rely condition \mathcal{R}

$$\text{stable}_{\mathcal{R}}(P) \stackrel{\text{def}}{=} P \subseteq \{ \sigma \in P \mid \forall \sigma', (\sigma, \sigma') \in \mathcal{R} \implies \sigma' \in P \}$$

- Instruction α satisfies guarantee condition \mathcal{G}

$$\text{sat}(\alpha, \mathcal{G}) \stackrel{\text{def}}{=} \{ \sigma \mid \forall \sigma', (\sigma, \sigma') \in \text{beh}(\alpha) \implies (\sigma, \sigma') \in \mathcal{G} \}$$

- Now introduce rely/guarantee judgements at three levels

Instruction Level (\vdash_a)

$$\mathcal{R}, \mathcal{G} \vdash_a P\{c\}Q \stackrel{\text{def}}{=} \text{stable}_{\mathcal{R}}(P) \wedge \text{stable}_{\mathcal{R}}(Q) \wedge \text{vc}(\alpha) \subseteq \text{sat}(\alpha, \mathcal{G}) \wedge P\{c\}Q$$

- Interplay between environmental interference and pre-, post-conditions handled through stability

Component Level (\vdash_c)

$$\text{Atom} \frac{\mathcal{R}, \mathcal{G} \vdash_a P\{\alpha\}Q}{\mathcal{R}, \mathcal{G} \vdash_c P\{\alpha\}Q}$$

$$\text{Seq} \frac{\mathcal{R}, \mathcal{G} \vdash_c P\{c_1\}M \quad \mathcal{R}, \mathcal{G} \vdash_c M\{c_2\}Q}{\mathcal{R}, \mathcal{G} \vdash_c P\{c_1; c_2\}Q}$$

$$\text{Choice} \frac{\mathcal{R}, \mathcal{G} \vdash_c P\{c_1\}Q \quad \mathcal{R}, \mathcal{G} \vdash_c P\{c_2\}Q}{\mathcal{R}, \mathcal{G} \vdash_c P\{c_1 \sqcap c_2\}Q}$$

$$\text{Iteration} \frac{\mathcal{R}, \mathcal{G} \vdash_c P\{c\}P \quad \text{stable}_{\mathcal{R}}(P)}{\mathcal{R}, \mathcal{G} \vdash_c P\{c^*\}Q}$$

$$\text{Conseq} \frac{\mathcal{R}, \mathcal{G} \vdash_c P\{c\}Q \quad P' \subseteq P \quad \mathcal{R}' \subseteq \mathcal{R} \quad Q \subseteq Q' \quad \mathcal{G} \subseteq \mathcal{G}'}{\mathcal{R}', \mathcal{G}' \vdash_c P'\{c\}Q'}$$

Global Level (\vdash)

- Global satisfiability needs component satisfiability + **interference check**

$$\text{Comp} \frac{\mathcal{R}, \mathcal{G} \vdash_c P\{c\}Q \quad \text{rif}(\mathcal{R}, \mathcal{G}, c)}{\mathcal{R}, \mathcal{G} \vdash P\{c\}Q}$$

- Usual parallel rule

$$\text{Par} \frac{\mathcal{R}_1, \mathcal{G}_1 \vdash_c P_1\{c_1\}Q_1 \quad \mathcal{R}_2, \mathcal{G}_2 \vdash_c P_2\{c_2\}Q_2 \quad \mathcal{G}_2 \subseteq \mathcal{R}_1 \quad \mathcal{G}_1 \subseteq \mathcal{R}_2}{\mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2 \vdash P_1 \wedge P_2\{c_1 \parallel c_2\}Q_1 \wedge Q_2}$$

Section 4

Multicopy Atomic Memory Models

Reordering Semantics: Basics

- Multicopy atomic memory models can be characterised using a *reordering* relation \leftarrow over pairs of instructions in a component
- \leftarrow is syntactically derivable based on the specific memory model.
E.g., in ARMv8
 - Two instructions which don't access (read or write) a common variable can be reordered
 - Various types of memory barriers prevent reordering
- *Forwarding* is another complication
 - $\beta = x := 3; \alpha = y := x$. Can forward the value 3 to y , losing dependence between α, β .
 - $x := 3 ; y := x \implies y := 3 ; x := 3$
 - Denote α with the value written in an earlier instruction forwarded to it as $\alpha_{<\beta>}$.
- Forwarding may continue arbitrarily and can span multiple instructions

Reordering Semantics: Formal

- $\alpha_{<c>}$: cumulative forwarding effects of the instructions in command c on α
- Ternary relation $\gamma < c < \alpha$: Reordering of instruction α prior to command c , with cumulative forwarding effects producing γ .
- Definition by induction

$$\alpha_{<\beta>} < \beta < \alpha \stackrel{\text{def}}{=} \beta \leftrightarrow \alpha_{<\beta>}$$

$$\alpha_{<c_1; c_2>} < c_1; c_2 < \alpha \stackrel{\text{def}}{=} \alpha_{<c_1; c_2>} < c_1 < \alpha_{<c_2>} \wedge \alpha_{<c_2>} < c_2 < \alpha$$

- Example: $\alpha = (y := x), \beta = (x := 3), \gamma = (z := 5)$.
 $\alpha_{<\beta>} = (y := 3), \alpha_{<\gamma; \beta>} = (y := 3)$.

$$y := 3 < x := 3 < y := x \quad y := 3 < z := 5 ; x := 3 < y := x$$

- Can execute an instruction which occurs later in the program if reordering and forwarding can bring it (in its new form γ) to the beginning

$$\text{Reorder} \frac{c_2 \mapsto_{\alpha} \quad \gamma < c < \alpha}{c_1; c_2 \mapsto_{\gamma} c_1; c'_2}$$

Reordering Interference Freedom

- Insight: Any valid reordering will **preserve thread-local semantics**, thus may only invalidate reasoning when **observed by the environment**.
 - Abstraction to the rescue again! Observed by environment $\implies \mathcal{G}$ violated, or \mathcal{R} not strong enough
- Three Levels: Instructions, Commands, Program

RIF: Instructions

- Two instructions are *reordering interference free*: Reasoning over them in their original order is sufficient to include reordered behaviour.

$$\begin{aligned} \text{rif}_a(\mathcal{R}, \mathcal{G}, \beta, \alpha) &\stackrel{\text{def}}{=} \forall P, Q, M. \mathcal{R}, \mathcal{G} \vdash_a P\{\beta\}M \wedge \mathcal{R}, \mathcal{G} \vdash_a M\{\alpha\}Q \\ &\implies \exists M'. \mathcal{R}, \mathcal{G} \vdash_a P\{\alpha_{<\beta>}\}M' \wedge \mathcal{R}, \mathcal{G} \vdash_a M'\{\beta\}Q \end{aligned}$$

RIF: Commands

- Command c is *reordering interference free* from α under \mathcal{R}, \mathcal{G} if the reordering of α over each instruction of c is reordering interference free, *including those variants produced by forwarding*.

$$\text{rif}_c(\mathcal{R}, \mathcal{G}, \beta, \alpha) \stackrel{\text{def}}{=} \text{rif}_a(\mathcal{R}, \mathcal{G}, \beta, \alpha)$$

$$\text{rif}_c(\mathcal{R}, \mathcal{G}, c_1; c_2, \alpha) \stackrel{\text{def}}{=} \text{rif}_c(\mathcal{R}, \mathcal{G}, c_1, \alpha_{<c_2>}) \wedge \text{rif}_c(\mathcal{R}, \mathcal{G}, c_2, \alpha)$$

RIF: Programs

- Program c is *reordering interference free* if and only if **all possible reorderings** of its instructions over the respective prefixes are reordering interference free.

$$\text{rif}(\mathcal{R}, \mathcal{G}, c) \stackrel{\text{def}}{=} \forall \alpha, r, c'. c \mapsto_{\alpha <_r} c' \implies \text{rif}_c(\mathcal{R}, \mathcal{G}, r, \alpha) \wedge \text{rif}(\mathcal{R}, \mathcal{G}, c')$$

- Observe: Checking $\text{rif}(\mathcal{R}, \mathcal{G}, c)$ amounts to
 - Checking $\text{rif}_a(\mathcal{R}, \mathcal{G}, \beta, \alpha)$ for all pairs of instructions β, α that can reorder in c
 - Including those pairs for which α is a new instruction generated through forwarding

Gameplan

- ① Compute all pairs of reorderable instructions (β, α) .
- ② Demonstrate reordering interference freedom for as many of these pairs as possible (using $\text{rif}_a(R, G, \beta, \alpha)$).
- ③ If rif_a cannot be shown for some pairs
 - introduce memory barriers to prevent their reordering or
 - modify the verification problem such that their reordering can be considered benign
- ④ Verify the component in isolation, using standard rely/guarantee reasoning with an assumed sequentially consistent memory model.

For a thread with n reorderable instructions,

$n!$ Possible Behaviours $\longrightarrow n(n-1)/2$ rif_a checks

Thanks for staying tuned :)

Section 5

Non-Multicopy Atomic Memory Models

Non-Multicopy Atomic Memory Models

- There is **no shared state** that all components agree on throughout execution, invalidating a core assumption of standard rely/guarantee reasoning.

Write History Semantics: Representation

- Each component is associated with a unique identifier.
- Shared memory state is represented as a list of variable writes $\langle w_1, w_2, w_3, \dots \rangle$, with metadata to indicate which components have performed and observed particular writes.
- The order of events in this write history provides an *overall order* to the system's events, with those later in the list being the most recent.
- Each $w_i = (x \mapsto v)_{rds}^{wr}$ where
 - x is a variable
 - v is a value
 - $\text{writer}((x \mapsto v)_{rds}^{wr}) = wr$ is the writer component's identifier
 - $\text{readers}((x \mapsto v)_{rds}^{wr}) = rds$ is the set of component identifiers that have observed the write
 - $\text{var}((x \mapsto v)_{rds}^{wr}) = x$

Write History Semantics: Manipulation

- Divide instructions into two types: *global* and *local*. Global instructions α are:
 - Store $(x := v)_i$, Load $[x = v]_i$, Memory barrier fence $_i$, Skip instruction (corresponding to some internal step)
- Behaviour of these instructions is formalised as (for skip it's just *id*):

$$\text{beh}((x := v)_i) = \{ (h \circ h', h \circ (x \mapsto v)_{\{i\}}^i \circ h') \mid$$

$$\forall w \in h'. \text{writer}(w) \neq i \wedge (\text{var}(w) = x \implies i \notin \text{readers}(w))$$

$$\text{beh}([x = v]_i) = \{ (h \circ (x \mapsto v)_r^j \circ h', h \circ (x \mapsto v)_r^j \circ h') \mid$$

$$\forall w \in h'. (\text{var}(w) = x \implies i \notin \text{readers}(w))$$

$$\text{beh}(\text{fence}_i) = \{ (h, h) \mid \forall w \in h. (i \in \text{readers}(w) \implies \forall y. y \in \text{readers}(w))$$

- Propagations of writes are modelled as environment effects and can take place at any point during the execution.

$$\text{prp} = \{ (h \circ (x \mapsto v)_r^j \circ h', h \circ (x \mapsto v)_{r \cup \{i\}}^j \circ h') \mid$$

$$i \notin r \wedge \forall w \in h. (\text{var}(w) = x \implies i \in \text{readers}(w)) \}$$

More Notation

- New constructor in the language: $\text{comp}(i, m, c)$ indicating a component with
 - identifier i
 - local state m
 - command c
- Assume a local behaviour relation lbeh such that $(m, \alpha', m') \in \text{lbeh}(\alpha)$ if executing α
 - changes the local state from m to m'
 - corresponds to the global instruction α'

$$\text{comp}(i, m, c) \mapsto_{\alpha'_i} \text{comp}(i, m', c') \iff c \mapsto_{\alpha} c' \wedge (m, \alpha', m') \in \text{lbeh}(\alpha)$$

- Go from local semantics/reasoning to global semantics/reasoning using comp and lbeh .
 - Constraint: systems are constructed as the parallel composition of a series of comp commands.
 - Trivial support for local state (e.g., hardware registers).

Meaning of Judgement

- If there is no global state, what does $\mathcal{R}, \mathcal{G} \vdash P\{c\}Q$ (for a component i with command c) mean?
- For a set of components I , write history h , for all variables x , $\text{view}_I(h, x) = v$ iff

$$h = h' \circ (x \mapsto v)_r^w \circ h'' \wedge I \subseteq r \wedge \forall w_i \in h''. \text{var}(w_i) = x \implies I \not\subseteq \text{readers}(w_i)$$

- For all executions of c
 - If
 - the execution operates on a write history h such that $\text{view}_i(h) \in P$
 - all propagations to i modify view_i in accordance with \mathcal{R}
 - Then i will
 - modify view_i in accordance with G
 - given termination, will end with a write history h such that $\text{view}_i(h) \in Q$
- This state mapping allows for rely/guarantee judgements **over individual components** to be trivially lifted from a standard memory model to their respective views of a write history.

Parallel Composition: Taming the Beast

- Parallel composition is complicated: **Need to relate differing components views.**
- *If* the execution of an instruction α by some component i satisfies its guarantee specification G_i in state h ,

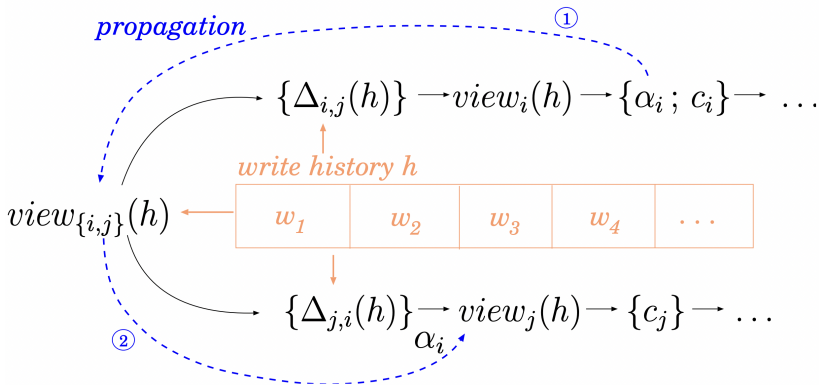
$$\text{view}_i(h) \in \text{sat}(\alpha, G_i)$$

- *Then* the effects of propagating α 's writes to some other component j will satisfy its rely specification R_j in its view,

$$\text{view}_j(h) \in \text{sat}(\alpha, R_j)$$

- Insight: It is possible to relate the views of two components by only considering the **difference in their observed writes**, i.e., the writes one component has observed but the other has not.

Travelling Between Components



- Aim to demonstrate rely/guarantee compatibility when propagating an instruction α from component i to component j
 - Given: component i executes α such that $view_i(h) \in \text{sat} \alpha, G_i$.
 - Step 1: Show that α can be executed in the shared view, i.e., $view_{\{i,j\}}(h) \in \text{sat}(\alpha, G_i)$.
 - Step 2: Show that α can be executed in component j 's view, i.e.,

Endgame

- Define a compatibility relation

$$\text{compat}(\mathcal{G}_i, \mathcal{R}_i, \mathcal{R}_j) \hat{=} \forall x, v \cdot wp(\mathcal{R}_i \cap \mathcal{R}_j \cap \text{id}_x, \text{sat}(x := v, \mathcal{G}_i)) \subseteq \text{sat}(x := v, \mathcal{R}_j)$$

- Modify the rules for parallel composition (note that we need separate relies and guarantees for each component because demonstrating compat requires pairwise checking)

$$[\text{Comp}'] \frac{\mathcal{R}, \mathcal{G} \vdash_c P\{c\}Q \quad \text{rif}(\mathcal{R}, \mathcal{G}, c)}{[i \rightarrow \mathcal{R}], [i \rightarrow \mathcal{G}] \vdash P\{\text{comp}(i, m, c)\}Q}$$

$$[\text{Par}'] \frac{\begin{array}{l} \mathcal{R}_1, \mathcal{G}_1 \vdash P_1\{c_1\}Q_1 \quad \mathcal{R}_2, \mathcal{G}_2 \vdash P_2\{c_2\}Q_2 \quad \text{disjoint}(\mathcal{R}_1, \mathcal{R}_2) \\ \forall i \in \text{dom}(\mathcal{R}_1) \cdot \forall j \in \text{dom}(\mathcal{R}_2) \cdot \text{compat}(\mathcal{G}_1(i), \mathcal{R}_1(i), \mathcal{R}_2(j)) \\ \forall i \in \text{dom}(\mathcal{R}_2) \cdot \forall j \in \text{dom}(\mathcal{R}_1) \cdot \text{compat}(\mathcal{G}_2(i), \mathcal{R}_2(j), \mathcal{R}_1(j)) \end{array}}{\mathcal{R}_1 \uplus \mathcal{R}_2, \mathcal{G}_1 \uplus \mathcal{G}_2 \vdash P_1 \cap P_2\{c_1 \parallel c_2\}Q_1 \cap Q_2}$$