What's Decidable about Causally Consistent Shared Memory?

Ori Lahav  
TEL AVIV UNIVERSITY

Udi Boker  
IDC HERZLIYA

OWLS  
January 20, 2021
Safety Verification

\[ X := 1; \]
\[ \text{repeat} \]
\[ \quad a := Y; \]
\[ \text{until} \ (a = 0); \]
\[ Z := 0; \]
\[ \text{assert} \ (Z = 0); \]
\[ X := 0; \]

\[ Y := 1; \]
\[ \text{repeat} \]
\[ \quad b := X; \]
\[ \text{until} \ (b = 0); \]
\[ Z := 1; \]
\[ \text{assert} \ (Z = 1); \]
\[ Y := 0; \]
Simple Solution

For programs with a **bounded data domain**, this problem is clearly **decidable**:

- Reduction to reachability in finite-state systems
- PSPACE-complete
Sequential Consistency (SC)

- We assumed the classical shared-memory model.
- But it is **unrealistic**: for better performance/scalability/availability/fault-tolerance shared-memory implementations provide weaker semantics.
- Similar situation in *distributed data-stores*
Weaker Memory Models

- x86-TSO
- POWER
- ARM
- RISC-V
- C/C++11
- many many more...

Decidable
[Atig, Bouajjani, Burckhardt, Musuvathi. POPL’2010]
[Abdulla, Atig, Bouajjani, Ngo. CUNCUR’2016]

Undecidable
[Abdulla, Arora, Atig, Krishna. PLDI’2019]

- Even for the Release/Acquire fragment:
  memory_order_release & memory_order_acquire
- Reduction from Post correspondence problem
Causal Consistency

- A classical model originated from *replicated data stores*:
  - nodes may disagree on the order of some operations
  - consensus on the order of “causally related” operations
- Relatively simple and intuitive but more scalable than SC
Safety Verification

\[
X := 1;
\text{repeat}
\quad a := Y; \quad \text{// 0}
\text{until (a = 0);}\\
Z := 0;\\
\text{assert (Z = 0);}\\
X := 0;
\]

\[
Y := 1;
\text{repeat}
\quad b := X; \quad \text{// 0}
\text{until (b = 0);}\\
Z := 1;\\
\text{assert (Z = 1);}\\
Y := 0;
\]

Both can read 0 in the same execution!
Is it a Problem?


- How come airplanes don’t crash?
- There are ways to demand sequential consistency when we need it.
- We often don’t need sequential consistency in its full power.

We have to define and understand the semantics of shared-memory concurrency.
Flag-Based Synchronization

\[ X = Y = 0 \]

\[ Y := 1 \quad \quad a := X \quad // \quad 1 \]
\[ X := 1 \quad \quad b := Y \quad // \quad 0 \]

This behavior is forbidden under causal consistency
Formal Semantics

Defined **declaratively** using **execution graphs**

\[
\text{happens-before} = (\text{program-order} \cup \text{reads-from})^+ 
\]

\[
\begin{align*}
X &= Y = 0 \\
Y := 1 &\quad \| \quad a := X /// 1 \\
X := 1 &\quad \| \quad b := Y /// 0
\end{align*}
\]

inconsistent execution graph

disallowed program outcome
\[ X := 1; \]
\[
\text{repeat} \quad a := Y; \\
\text{until } (a = 0); \quad \text{repeat} \quad b := X; \\
Z := 0; \quad \text{until } (b = 0); \quad Z := 1; \\
\text{assert } (Z = 0); \quad \text{assert } (Z = 1); \]

The execution graph is consistent. The annotated outcome is allowed.
Non-Multi-Copy-Atomicity

Weaker than x86-TSO: different threads can observe writes in different orders.

The execution graph is consistent.

The annotated outcome is allowed.

The program-order reads-from relationship is shown in the diagram.
What about **concurrent writes**?

The execution graph is consistent.

The annotated outcome is allowed.
What about **concurrent writes**?

- Nothing
- Make sure that they are totally ordered
  - Weak Release/Acquire (WRA)
  - Release/Acquire (RA)
  - Strong Release/Acquire (SRA)
Three Variants

Weak Release/Acquire (WRA)

Release/Acquire (RA)

Strong Release/Acquire (SRA)

∃ total order on writes to the same location (modification-order) s.t.:
WRA is strictly weaker than RA

WRA is strictly weaker than RA

w.l.o.g

W x 1 → R x 1 → W x 2

R x 1 → R x 2 → R x 1

∀ X 1, X 2, X 1 → X 2

WRA

RA

program-order

reads-from

modification-order

happens-before
RA is strictly weaker than SRA

\[
\begin{align*}
X &= 1 \\
Y &= 2 \\
a &= Y \div 1
\end{align*}
\quad\quad
\begin{align*}
Y &= 1 \\
X &= 2 \\
b &= X \div 1
\end{align*}
\]

RA is strictly weaker than SRA

reads-from

program-order

modification-order
Causal Consistency

Sequential Consistency

x86-TSO

Causal Consistency

Distributed data-stores
POWER architecture [L, Giannarakis, Vafeiadis. POPL'16]

C/C++11
Java 9

CC in [Bouajjani, Enea, Guerraoui, Hamza. POPL'17]
[Kokologiannakis, L, Sagonas, Vafeiadis. POPL'18]
Write-Write-Race Freedom

Theorem

\[ \text{Prog has no WW-races under SRA} \implies [\text{Prog}]_{\text{WRA}} = [\text{Prog}]_{\text{RA}} = [\text{Prog}]_{\text{SRA}} \]
Results

The verification problems under SRA and WRA are decidable.

- In contrast with RA [Abdulla, Arora, Atig, Krishna. PLDI’2019]
- To obtain this result we develop a new semantics for SRA and WRA.

Corollary

The verification problem under RA is decidable for write-write-race-free programs.
Lower Complexity Bound

• For causal consistency, the problem is non-primitive recursive

• We can simulate a lossy FIFO channel machine
  • as for x86-TSO  
  [Atig, Bouajjani, Burckhardt, Musuvathi. POPL’2010]
Even when the program is finite state, its synchronization with a causally consistent memory is infinite state.
• To establish **decidability**:

• We use the framework of **well-structured transition systems** (WSTS) [Abdullah] [Finkel, Schnoebelen] …

• **Challenge**: find a WSTS equivalent to a casually consistent memory
Backward Reachability

Input: LTS \((Q, Q_0, \rightarrow)\), state \(q_{\text{bad}}\)
Output: is \(q_{\text{bad}}\) reachable?

\[
S := \{q_{\text{bad}}\} \\
\text{repeat} \\
\quad S_{\text{prev}} := S \\
\quad S := S \cup \text{pre}(S) \\
\text{until } (S = S_{\text{prev}}) \\
\text{return } (Q_0 \cap S \neq \emptyset)
\]

To make this an algorithm we need to work with \textit{finitely representable sets} & guarantee termination
Well-Structured Transition Systems

• Equip the transition system with a **well quasi-order** $\leq$

• Work with **upward closed** sets of states represented by their finite basis

$\leq$ should be compatible with $\rightarrow$

$$
q_1' \xrightarrow{*} q_2' \\
\forall l \quad q_1' \leq q_1 \\
q_1 \rightarrow q_2
$$

Compatibility is guaranteed in “lossy” systems

$$
\begin{align*}
q' & \geq q \\
\text{(lose)} & \quad \frac{q'}{q'} \rightarrow q
\end{align*}
$$

• **Challenge:** characterize casually consistent shared memory as a **lossy** system
Causal Consistency as a WSTS

Two critical obstacles:

- Partial order embedding is not a well-quasi-order
- Execution histories are not lossy
Key Idea

Record the threads’ potentials in memory states:

*what possible sequences of reads each thread can execute?*
Lossy SRA

\[ Q = \text{Threads} \rightarrow \{Rxv \mid x \in \text{Var}, v \in \text{Val}\}^* \]

\[ Q_0 = \text{Threads} \rightarrow \{Rx0 \mid x \in \text{Var}\}^* \]

\[ q \preceq q' \iff \forall \tau \in \text{Threads} . q(\tau) \sqsubseteq q'(\tau) \]
\[ \begin{align*}
X & := 1; \\
\text{repeat} & \\
\quad a & := Y; \\
\text{until} & \ (a = 0); \\
Z & := 0; \\
\text{assert} & \ (Z = 0); \\
\end{align*} \]

\[ \begin{align*}
Y & := 1; \\
\text{repeat} & \\
\quad b & := X; \\
\text{until} & \ (b = 0); \\
Z & := 1; \\
\text{assert} & \ (Z = 1); \\
\end{align*} \]
Potential Maintenance for SRA

- **Lose step**
  - Remove some elements from the potentials

- **Read steps**
  - Precondition: first element in \( \tau \)'s potential is \( R_{xv} \)

- **Write steps**
  - Precondition: no \( R_{x_\_} \) in \( \tau \)'s potential
  - All threads may get new options \( R_{xv} \)
  - \( \rightarrow \) where?

\( \varepsilon \)  
\( \tau : R_{xv} \)  
\( \tau : W_{xv} \)
Where?

\[
\begin{align*}
X &= Y = 0 \\
X &= 1
\end{align*}
\]

\[
\begin{align*}
a &= X // 1 \\
b &= Y // 0
\end{align*}
\]

This transition **should be allowed**

\[
\begin{align*}
X &= Y = 0 \\
Y &= 1 \\
X &= 1
\end{align*}
\]

\[
\begin{align*}
a &= X // 1 \\
b &= Y // 0
\end{align*}
\]

This transition **should not be allowed**
Every sequence of reads that thread $\pi$ can perform after reading from a certain write executed by thread $\tau$ could be performed by thread $\tau$ immediately after it executed the write.
### Where?

<table>
<thead>
<tr>
<th>X = Y = 0</th>
<th>a = X // 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = 1</td>
<td>b = Y // 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X = Y = 0</th>
<th>a = X // 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y = 1</td>
<td>b = Y // 0</td>
</tr>
<tr>
<td>X = 1</td>
<td></td>
</tr>
</tbody>
</table>

---

**Thread 1 can read Y=0**

**Thread 1 cannot read Y=0**

---

**This transition should be allowed**

**This transition should not be allowed**
More Details

- **Multiple lists** per thread
- **Writer thread’s id** in “read options”
- Additional flags to handle **RMWs** (atomic Read-Modify-Writes)

\[
\begin{align*}
\text{WRITE} & \quad \forall \tau \in \text{Tid}, L' \in B'(\pi). \exists n \geq 0, u_1, \ldots, u_n, L_0, \ldots, L_n. \\
& \quad L' = L_0 \cdot (\tau, x, v_w, u_1) \cdot L_1 \cdot \ldots \cdot (\tau, x, v_w, u_n) \cdot L_n \\
& \quad \wedge L_0 \cdot \ldots \cdot L_n \in B(\pi) \wedge L_1 \cdot \ldots \cdot L_n \in B(\tau) \\
& \quad \wedge v_0 \in L_1 \cdot \ldots \cdot L_n. \text{loc}(o) \neq x \\
& \quad \wedge \forall o \in L_0. \text{loc}(o) = x \implies \pi \neq \tau \wedge \text{rmw}(o) = R \\
\end{align*}
\]

\[
\begin{align*}
B & \xrightarrow{\tau, W(x, o_u)}_{\text{ioSRA}} B' \\
\text{RMW} & \quad \text{loc}(o) = x \quad \text{val}(o) = v_R \\
& \quad \tau_{\text{rmw}}(o) = \text{RMW} \\
& \quad B = B_{\text{mid}}[\tau \mapsto o \cdot B_{\text{mid}}(\tau)] \\
& \quad B_{\text{mid}} \xrightarrow{\tau, W(x, o_u)}_{\text{ioSRA}} B' \\
& \quad B \xrightarrow{\tau, \text{RW}(x, o_u)}_{\text{ioSRA}} B' \\
\end{align*}
\]

\[
\begin{align*}
\text{READ} & \quad \text{loc}(o) = x \quad \text{val}(o) = v_R \\
& \quad B = B'[\tau \mapsto o \cdot B'(\tau)] \\
& \quad B \xrightarrow{\tau, R(x, o_k)}_{\text{ioSRA}} B' \\
\end{align*}
\]

\[
\begin{align*}
\text{LOWER} & \quad B' \subseteq B \\
& \quad B \xrightarrow{\varepsilon}_{\text{ioSRA}} B' \\
\end{align*}
\]
Multiple Lists per Thread

\[
\begin{array}{c|c|c|c|c|c}
\text{x := 0} & \text{y := 0} & \text{z := 0} & \text{d\_1 := x \text{ // 1}} & \text{e\_1 := y \text{ // 1}} & \text{f\_1 := z \text{ // 1}} \\
\text{x := 1} & \text{y := 1} & \boxed{\text{z := 1}} & \text{d\_2 := y \text{ // 1}} & \text{e\_2 := z \text{ // 1}} & \text{f\_2 := x \text{ // 1}} \\
\text{a\_1 := z \text{ // 1}} & \text{b\_1 := x \text{ // 1}} & \text{c\_1 := y \text{ // 1}} & \text{d\_3 := z \text{ // 0}} & \text{e\_3 := x \text{ // 0}} & \text{f\_3 := y \text{ // 0}} \\
\hline
\text{a\_2 := y \text{ // 0}} & \text{b\_2 := z \text{ // 0}} & \text{c\_2 := x \text{ // 0}} & & & \\
\end{array}
\]
Writer Thread in “Read Options”

\[
\begin{align*}
x &:= 0 \quad | \quad y := 0 \quad | \quad a := z \quad \parallel 1 \\
x &:= 1 \quad | \quad y := 1 \quad | \quad w := 1 \\
z &:= 1 \quad | \quad z := 1 \quad | \quad b := x \quad \parallel 0 \\
\end{align*}
\]

\[\begin{array}{c}
Rz1 \\
Ry0
\end{array}\quad \begin{array}{c}
Rz1 \\
Ry0
\end{array}\quad \begin{array}{c}
Rz1 \\
Rx0
\end{array}\quad \begin{array}{c}
Rz1 \\
Rx0
\end{array}\]

\[1 : Rz1 \quad \parallel \quad 2 : Rz1\]

Now, since the potential of thread \(0\) is used both for reading and writing, it “synchronizes” with the immediate future of \(0\) and retains all \(0\) reads.

Hence, to allow the addition of a read option \(\mathcal{R}\), we need to (i) extend the \(\mathcal{R}\) type of \(\mathcal{H}\) to accommodate read lists, and (ii) add a read option list to the state of each thread. This guarantees that after zero or more \(\mathcal{R}\) phases, the \(\mathcal{R}\) value is the first occurrence of \(\mathcal{R}\) in the thread's history.

What went wrong? The problem arises when \(\mathcal{R}\) phases are performed by \(\mathcal{R}\) threads with both read and write options. Together with location and value, read options allow the \(\mathcal{R}\) threads to specify a read that is necessarily included in the \(\mathcal{R}\) list.

Another complication arises due to the fact that read options may include the thread identify of the \(\mathcal{R}\) thread. For instance, \(\mathcal{R}\) threads with both read and write options may perform \(\mathcal{R}\) operations without \(\mathcal{R}\) threads being present. The \(\mathcal{R}\) threads are allowed to access the \(\mathcal{R}\) list.

Consider the following program:

\[\mathcal{R}\]
Three Variants

Weak Release/Acquire (WRA)

Release/Acquire (RA)

Strong Release/Acquire (SRA)

∃ total order on writes to the same location (modification-order) s.t.
Potential-Based System for WRA

Write steps

Precondition: no \(Rx\) in \(\tau\)'s potential

All threads may get new options \(Rx\)

\[\tau : W_{xv}\]

\[\begin{align*}
  x &= 1 \\
  x &= 2 \\
  x &= 3
\end{align*}\]

\[\begin{align*}
  a &= x /\!\!/ 1 \\
  x &= 3 \\
  a &= x /\!\!/ 1
\end{align*}\]
Potential-Based System for WRA

Use "write options" to mark when writes are allowed

+ Simple constraints on where read options are added wrt write options

```
X = 1     X = 2     X = 3
WX
RX1
RX2
RX1
```

```
X = 1     X = 2     a = x // 1
X = 3
RX1
RX2
RX1
```
Every sequence of reads and writes that thread \( \tau \) can perform after reading from a certain write executed by thread \( \pi \) could be performed by thread \( \tau \) immediately after it executed the write.
Thread 1 cannot write to X and then read X=1

This transition is disallowed
The potential-based memory systems are equivalent to the SRA/WRA systems.

When synchronized with a (finite-state) concurrent program, the potential-based memory systems form WSTS.
Results

Theorem

The verification problems under SRA and WRA are decidable.

Corollary

The verification problem under RA is decidable for write-write-race-free programs.
Research Questions

• Useful implementation

• RA without RMWs?

• Other models and extensions of causal consistency (get closer to RA?)

• Parametrized programs

• Use the potential-based semantics for other verification approaches
Interested in concurrency & verification? I’m looking for students / postdocs!