Verification under causally consistent shared memory

Ori Lahav

TEL AVIV UNIVERSITY

November 14, 2019
About me

2010 — 2013
Ph.D.  
Logic in computer science  
Advisor: A. Avron

2013 — 2014
Postdoctoral researcher  
Program verification  
Host: M. Sagiv

2014 — 2017
Postdoctoral researcher  
Weak memory models  
Hosts: V. Vafeiadis, D. Dreyer

2017 —
Faculty member  
Tel Aviv University

Main areas of research:  
• Programming languages theory  
• Verification  
• Concurrency  
• Relaxed memory models

Teaching next semester:  
• Foundations of programming languages (0368-3241)  
• Undergraduate seminar on formal theory of concurrency (0368-3114)
Shared-Memory Concurrency

- **Concurrent** modules interact by reading and writing **shared** objects in **memory**:
  
  \[
  \text{store}(x,v) \\
  r := \text{load}(x) \\
  \text{atomic read-modify-write}
  \]

- What is the **semantics**?

- The classical model is **“sequential consistency”**
Sequential Consistency

- Simple and intuitive
- But **unrealistic**: To improve performance, no modern hardware provides sequential consistency
- Similar to *distributed data-stores*
Example

Initially  $X = Y = 0$.

$X := 1; \quad Y := 1;$

$a := Y; \quad b := X;$

Finally $a = 1 \lor b = 1$?
int main () {
    int cnt = 0;
    do {
        X = 0; Y = 0;
        thread first(thread1);
        thread second(thread2);
        first.join();
        second.join();
        cnt++;
    } while (a != 0 || b != 0);
    printf("%d\n",cnt);
    return 0;
}

int X, Y, a, b;

void thread1() {
    X = 1;
    a = Y;
}

void thread2() {
    Y = 1;
    b = X;
}
Is it a Problem?

https://en.wikipedia.org/wiki/Dekker%27s_algorithm

• 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.
Causal Consistency

• A model that is **weaker** than sequential consistency

• Originated from *replicated data stores*:
  • allows nodes to disagree on the relative order of some operations
  • require global consensus on the order of “causally related” operations

• Easier to implement efficiently (weaker than **x86-TSO**)

• More **scalable** than sequential consistency

• Provided by **C/C++11** and **Java 9** in the form of: **release/acquire**

• Defined **declaratively** using **execution graphs**
Declarative Concurrency Models

- Possible program behaviours are represented by a directed graph

\[
\begin{align*}
x &= y = 0 \\
x &= 1 & a &= y \quad // 1 \\
y &= 1 & b &= x \quad // 0 \\
\end{align*}
\]

- The model imposes mathematical constraints, defining consistent execution graphs
Happens-Before

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

- Threads \textit{synchronize} by reading from other threads

- Condition: \textbf{happens-before} should be a partial order

\( \text{hb} \) is irreflexive

The execution graph is \textit{inconsistent}.

\begin{itemize}
  \item \rightarrow \quad \text{program-order}
  \item \cdashrightarrow \quad \text{reads-from}
\end{itemize}
Reading “most recent” value

A thread may **not** read from a write if it is **hb**-aware of an **hb**-later write to the same variable.

**inconsistent** execution graph
**disallowed** program outcome
What to do about *concurrent writes*?

What to do about *concurrent writes*?

```
x = 1  |  a = x // 1  |  c = x // 2  |  x = 2
b = x // 2  |  d = x // 1
```

```
| W x 1 | R x 1 | R x 2 | W x 2 |
```

```
| R x 2 | R x 1 |
```

program-order

reads-from
What to do about *concurrent writes*?

- Nothing
- Make sure all threads agree on their order
  - Weak Release/Acquire (WRA)
  - Release/Acquire (RA)
  - Strong Release/Acquire (SRA)
WRA

- Allows rather strange behaviors:

\[
\begin{align*}
x &= 1 \\
a &= x \ // 1 \\
b &= x \ // 2 \\
c &= x \ // 2 \\
d &= x \ // 1 \\
x &= 2
\end{align*}
\]

- But, often suffices in concurrent programs

- *Concurrent writes are rare*
**Idea:** use another order to *globally* decide the order of writes to each variable

**modification-order** - total order on writes to the same variable

`mo` should agree with `hb`

A thread may *not* read from a write if it is `hb`-aware of an `mo`-later write
Example

```
x = 1  
| a = x // 1  
| b = x // 2  
| c = x // 2  
| d = x // 1  
| x = 2
```

W.l.o.g...

- **W**rite × 1
- **R**ead × 1
- **R**ead × 2
- **W**rite × 2

- **WRA**
- **RA**

→ **program-order**
→ **reads-from**
→ **modification-order**
→ **happens-before**
Quiz 1/4

The execution graph is consistent.

The annotated outcome is allowed.

- → program-order
- → reads-from
- → modification-order
The execution graph is **inconsistent**.

The annotated outcome is **disallowed**.

---

**program-order**

----

**reads-from**

-----

**modification-order**
The execution graph is **consistent**.

The annotated outcome is **allowed**.

---

program-order

reads-from

modification-order

---
Quiz 4/4

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

The execution graph is **consistent**.

The annotated outcome is **allowed**.

---

program-order

---

reads-from

---

modification-order
In RA we required **local** agreement between hb and mo.

SRA requires a **global** agreement.

The previous example is *allowed by RA* but *disallowed by SRA*.

SRA is a standard causal consistency model in replicated data stores.
Verification
Verification

• We assume that the memory provides causal consistency.

• How can we verify that our program is correct?
  • Never crashes
  • Provides mutual exclusion
  • Correctly implements a concurrent data structure
  • ...

Verification

- Full guarantees
- Bounded guarantees
- Testing

- Manual verification
- Interactive verification
- Automatic verification

Beware of bugs in the above code; I have only proved it correct, not tried it.

(Donald Knuth)
Approaches for Concurrent Programs

- **Unbounded verification**
  - Represent the semantics as a (possibly infinite) transition system and verify its properties

- **Bounded model checking**
  - Unroll the loops to get a loop-free program and do exhaustive search

- **Program logics**
  - Use a deductive system to reason about the program

- **Robustness**
  - Verification under weak model = verification under sequential consistency + robustness
Unbounded Verification

Decision problem

Input: a concurrent program $P$ over a finite data domain + a safety assertion

Question: does $P$ satisfy the assertion when running under a causally consistent shared memory?

Initially $X = Y = 0$.

$X := 1; \quad Y := 1; \quad a := Y; \quad b := X;$

Finally $a = 1 \lor b = 1$?

Difficulty: programs may have loops
Unbounded Verification

• For sequential consistency, this problem is clearly decidable:
  • Represent the whole system as a finite automaton
  • Check whether it can reach an “error” state
  • PSPACE-complete

• For weaker models (in particular, causal consistency) this is not so simple:
  • The transition system is infinite state!
Negative Results

• For RA, the problem is **undecidable**
  [Abdulla, Arora, Atig, Krishna. PLDI’2019]

  • Requires Read-Modify-Write instructions
  • Reduction from “Post correspondence problem”

<table>
<thead>
<tr>
<th></th>
<th>bba</th>
<th>ab</th>
<th>bba</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bb</td>
<td>aa</td>
<td>bb</td>
<td>baa</td>
</tr>
</tbody>
</table>

28
Positive Results

• For **SRA** and **WRA**, the problem is **decidable**
  [Lahav and Boker. Submitted for publication]

• Use the framework of **well-structured transition systems** (WSTS)
• **Challenge**: find a WSTS equivalent to a casually consistent memory
• **Idea**: the state consists of the “**potential of each thread**”:
  what sequences of operations each thread can do?

using (+1,+2,+3) (-1,-10,+3) (-1,+2,0) (-2,+4,+16)
can (0,0,0) reach (at least) (16,12,39) without going below 0?
Open questions

- **RA** without RMWs
- Other models and extensions of causal consistency
- Useful implementation
Bounded Verification

Goal:

Given a *loop-free* program (usually after loop unrolling), *exhaustively* verify that *all its runs* do not violate the safety assertions.
Partial Order Reduction

Consider, first, **sequential consistency**:

- Naively checking all (exponentially many) traces is **infeasible**
- Many traces are **equivalent** with respect to the program’s safety
- **Execution graphs** track less redundant order and represent large equivalence classes
Counterintuitively, causal consistency makes the problem easier!

- Causally consistent execution graph are easier to efficiently explore
  - "prefix-determinedness": only the “happens-before"-prefix of a read determines the values it may read
  - allows to safely revert earlier choices

- Easier to check: given an execution graph level (program-order and reads-from only):
  - Checking for SC-consistency is NP-complete
  - Checking for (W)RA-consistency is in PTIME
Program Logics

• Derivation rules that provide reasoning principles for (concurrent) programs.

\[
\frac{\{P\} C_1\{Q\} \quad \{Q\} C_2\{R\}}{\{P\} C_1; C_2\{R\}} \\
\frac{\{P \land b\} C\{P\}}{\{P\} \text{while } b \text{ do } C\{P \land \neg b\}}
\]
Owicki-Gries Logic

\{ y = 0 \}

\{ T \}
\{ x := 42 \}
\{ x = 42 \}
\{ y := 1 \}
\{ T \}
\{ y \neq 0 \rightarrow x = 42 \}
\{ y = 0 \}
\{ y \neq 0 \rightarrow x = 42 \}
\{ \text{skip} \}
\{ x = 42 \}
\{ a := x \}
\{ a = 42 \}

Non-interference

\[ R \land x \vdash R \{ u/x \} \text{ for every:} \]
- assertion \( R \) in one proof outline
- assignment \( x := u \) with precondition \( P \) in the other proof outline

\( \vdash \)
\{ P \}
\{ x := u \}
\{ R \}
\{ \}
Unsoundness under Causal Consistency

\[ \{ x = 0 \land b = 2 \} \]

\[
\begin{align*}
\{ \top \} \\
x := 1 \\
x = 1 \\
a := y \\
x = 1 \land (y = 1 \rightarrow a = 1 \lor b = 1 \lor b = 2) \\
a = 1 \lor b = 1
\end{align*}
\]

\[
\begin{align*}
\{ b = 2, x \neq 2 \} \\
y := 1 \\
y = 1, x \neq 2 \\
b := x \\
y = 1, b \neq 2
\end{align*}
\]

[Lahav, Vafeiadis ICALP’14]
Robustness

• Many (useful) programs are *robust* against causal consistency:

• All program behaviours allowed by *causally consistent* memory are also allowed by *sequentially consistent* memory

\[
\text{verification under causal consistency} = \text{verification under sequential consistency} + \text{robustness}
\]
We developed a **sound and precise reduction** from robustness against RA to a **verification problem under SC**.

Idea: run an *instrumented program* under SC that *monitors* whether some step is allowed under causal consistency but not by sequential consistency.

Theorem

Execution-graph robustness against Release/Acquire is **decidable** and PSPACE-complete.
Beyond Causal Consistency
Beyond Causal Consistency

• Often we can get along with *weaker guarantees* than causal consistency
  
  • E.g., when some variables are not used for synchronization.

• Full models provide a spectrum of accesses:
C/C++11

memory_order_seq_cst (sc)
full memory fence

memory_order_release
write (rel1)
no fence (x86); lwsync (PPC)

memory_order_acquire
read (acq)
no fence (x86); isync (PPC)

memory_order_relaxed (rlx)
no fence

Non-atomic (na)
no fence, races are errors

memory_order_seq_cst

full memory fence

memory_order_release
write (rel1)
no fence (x86); lwsync (PPC)

memory_order_acquire
read (acq)
no fence (x86); isync (PPC)

memory_order_relaxed (rlx)
no fence
Non-atomic (na)
no fence, races are errors

[[-] : CExp \rightarrow P \{ \text{expr} : \text{Val} \cup \{ \_ \}, A : P_{\text{Aname}}, \text{lab} : \text{Act}, \text{sb} : P_{\text{A} \times A}, \text{lst} : A \}

[\forall x \in \{t, [t, [a, b, a]], a \in \text{Aname} \land \text{lab}(a) = \text{skip} \}

[aallocate] \{ ([t, [a, lab, b, a, a]]) = \text{Aname} \land \text{lab}(a) = \text{skip} \}

\{[\forall x \in \{[t, [a, \text{lab}, 0, a, a]], a \in \text{Aname} \land \text{lab}(a) = W_{x(v, v')} \}

[CAS] \{[t, [v, l, \text{lab}, 0, a, a]], a \in \text{Aname} \land \text{lab}(a) = W_{x(v, v')} \}

[CAS] \{[t, [v, l, \text{lab}, 0, a, a]], a \in \text{Aname} \land \text{lab}(a) = W_{x(v, v')} \}

ket \{ x = E \text{ in } E \text{ ] } \{ x \rightarrow A \text{, } b, \text{lab}, \text{sb}, \text{lst}, \text{lab} \} \} \{ \{ A, x \text{, lab}, \text{sb}, \text{lst}, \text{lab} \} \} \}

[\text{repeat } E \text{ end] } \{ \{ \text{repeat } E \text{ end] } \{ [x \rightarrow A \text{, } b, \text{lab}, \text{sb}, \text{lst}, \text{lab} \} \} \}

Axioms satisfied by consistent C11 executions,

\(\begin{align*}
\text{isSeqCst} &\quad \Rightarrow \text{isAcqCst} \quad \text{(ConsistentMO)} \\
\text{isSeqCst} &\quad \Rightarrow \text{isRelaxCst} \quad \text{(ConsistentSC)} \\
\text{isSeqCst} &\quad \Rightarrow \text{isReadCst} \quad \text{(ConsistentRR)} \\
\text{isSeqCst} &\quad \Rightarrow \text{isRelaxCst} \quad \text{(ConsistentRR)} \\
\text{isSeqCst} &\quad \Rightarrow \text{isWriteCst} \quad \text{(ConsistentRR)} \\
\text{isSeqCst} &\quad \Rightarrow \text{isRelaxCst} \quad \text{(AtomicRMW)} \\
\text{isSeqCst} &\quad \Rightarrow \text{isRelaxCst} \quad \text{(ConsistentAlloc)}
\end{align*}\)

![Figure 2. Semantics of closed program expressions.](image)

![Figure 3. Axioms satisfied by consistent C11 executions.](image)

![Figure 4. Sample executions violating coherency conditions (Batty et al. 2011).](image)
What makes a good programming language memory model?

- Formal and comprehensive
- Not too weak (good for programmers)
- Not too strong (good for hardware)
- Admit optimisations (good for compilers)
- **Allow verification**
The Problem of Programming Language
Concurrency Semantics

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod,
and Peter Sewell

University of Cambridge

“Disturbingly, 40+ years after the first relaxed-memory hardware was introduced (the IBM 370/158MP), the field still does not have a credible proposal for the concurrency semantics of any general-purpose high-level language that includes high performance shared-memory concurrency primitives. This is a major open problem for programming language semantics.”
Key Tool: Coq

- A framework for (mechanized) *proofs and programs*
- Main achievements:
  - the Four Colour Theorem
  - the Feit-Thompson Theorem
  - CompCert compiler
- ACM Software System Award 2014

[POPL’19] Bridging the Gap Between Programming Languages and Hardware Weak Memory Models
Podkopaev, Lahav, Vafeiadis

More in PL course next semester
Conclusion

• Verification of concurrent programs running under *casually consistent memory* poses interesting theoretical and practical *challenges*.

• It also provides *opportunities*: sometimes it is *easier* than verification under *sequential consistency* (e.g., dynamic partial order reduction).

• Idea: PL concurrency semantics should be designed with *verification* considerations in mind.