Maximal Causal Models

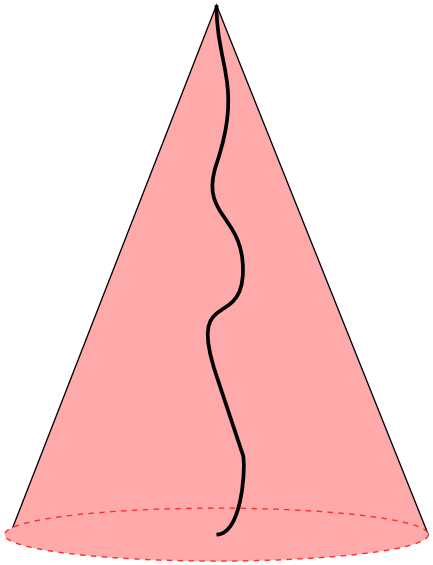
for Sequentially Consistent Systems

Traian Florin Șerbănuță, Feng Chen, and Grigore Roșu

University of Illinois at Urbana-Champaign
University Alexandru Ioan Cuza of Iași

January 14, 2014

Causal models for concurrent executions

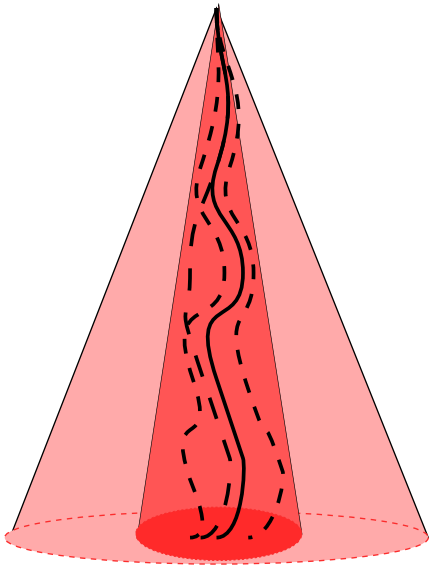


Problem: The space of all possible executions too big to test/explore

- Due to nondeterminism of thread interleavings

Question: Can we observe in an execution more than it appears?

Causal models for concurrent executions



Problem: The space of all possible executions too big to test/explore

- Due to nondeterminism of thread interleavings

Question: Can we observe in an execution more than it appears? Yes!

- Extract a causal model of the execution
- Analyze the model for potential bad behaviors

Causal model

- Record the execution as a series of events (concurrent operations)
  - Event: (Thread, operation, object, value, code location, . . .)
  - Operation type:
    - read, write, acquire, release, semaphoreUp, semaphoreDown,
    - wait, notify, spawn, join, . . .
- Use causal information to relax the total order of events recorded.

Existing causal models

Happens-before [Lamport, 1979] All operations on the same object are kept in the same order as in the observed execution

Happens-before with locksets [O'Callahan&Choi, PPoPP'03] Allow permutation of synchronized blocks

Weak happens-before [Sen&al., FMOODS'05] Preserve only the write-read dependency; locks are considered memory locations.

Weak happens-before with locksets [Wang&Stoller, PPoPP'06]

Existing causal models

Happens-before [Lamport, 1979] All operations on the same object are kept in the same order as in the observed execution

Happens-before with locksets [O'Callahan&Choi, PPoPP'03] Allow permutation of synchronized blocks

Weak happens-before [Sen&al., FMOODS'05] Preserve only the write-read dependency; locks are considered memory locations.

Weak happens-before with locksets [Wang&Stoller, PPoPP'06]

Question:

How much further can one go?

In this talk

A novel causal model

- Based on axioms for sequential consistency
- Making no assumption about the code generating the execution
- Sound by definition
- Maximal among sound causal models

Concurrent objects and serial specification

[Herlihy&Wing, 1990]

Concurrent objects

Entities exporting certain operations which any thread can perform on them.

Serial Specification

The legal behavior of a concurrent object in isolation.

Concurrent objects — Shared memory locations

Operations

- read the value of the location
- write a value at the location

Serial specification

Each read yields the value written by the latest previous write

Concurrent objects — Mutexes

Operations

- acquire the mutex
- release the mutex

Serial specification

- At any moment $\#acquire - \#release \in \{0, 1\}$
- All consecutive acquire-release pairs share the same thread

Concurrent objects — Semaphores

Operations

- V – increase number of available resources
- P – decrease number of available resources

Serial specification

At any moment $\#V \geq \#P$

Sequential consistency definitions

[Attiya&Welch, 1994]

Legal trace

Projection of trace on any concurrent object satisfies its serial specification.

Interleaving of a trace

Projection of the trace on any thread is the same as in the original trace.

$$\tau\restriction_t = \sigma\restriction_t$$

Sequentially consistent trace

A trace admitting a legal interleaving.

Our approach — Axioms for feasibility

Hypothesis

The legal traces of a sequentially consistent system obey two basic laws:

Law 1: Prefix Closedness (P.C.)

Events are indivisible and are generated in execution order

- If $\tau_1\tau_2$ feasible, then τ_1 also feasible

Law 2: Local determinism (L.D.)

Execution of an operation is determined by the previous events in the same thread and it can happen at any consistent moment after they were generated.

Local determinism — Example

A program generating trace

$$(t_1, \mathit{write}, l_1, 0)(t_1, \mathit{read}.l_1, 0)(t_1, \mathit{write}, l_2, 7)(t_2, \mathit{write}, l_1, 1)$$

can also generate (✓) or not (✗) the following

- ✓ the empty trace and all other prefixes (P.C.)

Local determinism — Example

A program generating trace

$$(t_1, \mathit{write}, l_1, 0)(t_1, \mathit{read}.l_1, 0)(t_1, \mathit{write}, l_2, 7)(t_2, \mathit{write}, l_1, 1)$$

can also generate (✓) or not (✗) the following

- ✓ the empty trace and all other prefixes (P.C.)
- ✓ $(t_2, \mathit{write}, l_1, 1)$ (L.D.)
- ✗ $(t_1, \mathit{read}, l_1, 0)$ or $(t_1, \mathit{write}, l_2, 7)$

Local determinism — Example

A program generating trace

$$(t_1, write, l_1, 0)(t_1, read.l_1, 0)(t_1, write, l_2, 7)(t_2, write, l_1, 1)$$

can also generate (✓) or not (✗) the following

- ✓ the empty trace and all other prefixes (P.C.)
- ✓ $(t_2, write, l_1, 1)$ (L.D.)
- ✗ $(t_1, read, l_1, 0)$ or $(t_1, write, l_2, 7)$
- ✓ $(t_1, write, l_1, 0)(t_2, write, l_1, 1)$ (L.D.)
- ✓ $(t_2, write, l_1, 1)(t_1, write, l_1, 0)$ (L.D.)

Local determinism — Example

A program generating trace

$$(t_1, write, l_1, 0)(t_1, read.l_1, 0)(t_1, write, l_2, 7)(t_2, write, l_1, 1)$$

can also generate (✓) or not (✗) the following

- ✓ the empty trace and all other prefixes (P.C.)
- ✓ $(t_2, write, l_1, 1)$ (L.D.)
- ✗ $(t_1, read, l_1, 0)$ or $(t_1, write, l_2, 7)$
- ✓ $(t_1, write, l_1, 0)(t_2, write, l_1, 1)$ (L.D.)
- ✓ $(t_2, write, l_1, 1)(t_1, write, l_1, 0)$ (L.D.)
- ✓ $(t_2, write, l_1, 1)(t_1, write, l_1, 0)(t_1, read.l_1, 0)$ (L.D.)
- ✓ ✓ $(t_1, write, l_1, 0)(t_2, write, l_1, 1)(t_1, read.l_1, 1)$ (L.D.)

Local determinism — Example

A program generating trace

$$(t_1, write, l_1, 0)(t_1, read.l_1, 0)(t_1, write, l_2, 7)(t_2, write, l_1, 1)$$

can also generate (✓) or not (✗) the following

- ✓ the empty trace and all other prefixes (P.C.)
- ✓ $(t_2, write, l_1, 1)$ (L.D.)
- ✗ $(t_1, read, l_1, 0)$ or $(t_1, write, l_2, 7)$
- ✓ $(t_1, write, l_1, 0)(t_2, write, l_1, 1)$ (L.D.)
- ✓ $(t_2, write, l_1, 1)(t_1, write, l_1, 0)$ (L.D.)
- ✓ $(t_2, write, l_1, 1)(t_1, write, l_1, 0)(t_1, read.l_1, 0)$ (L.D.)
- ✓ ✓ $(t_1, write, l_1, 0)(t_2, write, l_1, 1)(t_1, read.l_1, 1)$ (L.D.)
- ✓ $(t_2, write, l_1, 1)(t_1, write, l_1, 0)(t_1, read.l_1, 0)(t_1, write, l_2, 7)$ (L.D.)
- ✗ ✗ $(t_1, write, l_1, 0)(t_2, write, l_1, 1)(t_1, read.l_1, 1)(t_1, write, l_2, 7)$

Our causal model — The feasibility closure

Definition of feasible(τ)

The smallest set of traces containing τ and being closed under

- prefix closedness
- local determinism

The model is sound (by definition)

Because the sequential consistent system itself obeys the same laws.

Partial characterization

Theorem

feasible(τ) contains the legal prefixes of all interleavings of τ

Application

- Proving soundness for existing causal models
- By showing their traces are legal interleaving prefixes
- Check the article/tech. report for details

Complete characterization

- feasible(τ) = all the extended legal prefixes of interleavings of τ
- Extended — the final read for each thread can read a different value
- Example: $(t_1, write, l_1, 0)(t_2, write, l_1, 1)(t_1, read.l_1, 1)(t_2, write, l_3, 9)$ for

  $(t_1, write, l_1, 0)(t_1, read.l_1, 0)(t_1, write, l_2, 7)(t_2, write, l_1, 1)(t_2, write, l_3, 9)$

Maximality

Sketch of proof

Idea

Given a non-τ-feasible trace τ', find a program generating τ but not τ'.

- Build a canonical program p_τ generating τ
  - In a simple language whose constructs map to the concurrent operations
- Reduce to a trace σe which is not τ-feasible, but σ is
- If $\sigma\restriction_{thread(e)}$ prefix of $\tau\restriction_{thread(e)}$ then use program p_τ;
- Else the last event e' in $\sigma\restriction_{thread(e)}$ is a read with a different value
- If so, modify the canonical program generating τ
  - Insert a conditional right after the event generating e'

Maximality — possible applications

- Proving soundness for new models by proving them submodels
- Give unifying definitions to causal properties
  - Causal dataraces
  - Causal atomicity / serializability

Conclusion

- Maximal and sound causal model for sequentially consistent systems
- Useful to prove soundness for existing (and future) models
- Allowing unifying definitions of causal properties
- Potentially practical by itself
  - Formalized as-is in SMT solvers and used for finding dataraces [Said&al., NFM'11]

Conclusion

- Maximal and sound causal model for sequentially consistent systems
- Useful to prove soundness for existing (and future) models
- Allowing unifying definitions of causal properties
- Potentially practical by itself
  - Formalized as-is in SMT solvers and used for finding dataraces [Said&al., NFM'11]

Open Question

- How about other memory models?
  - Would the axiomatic approach still work?