Indistinguishability: Friend and Foe of Concurrent Data Structures

Hagit Attiya

CS, Technion
• **Uncertainty** is a main obstacle for designing correct applications in concurrent systems

• Formally captured by *indistinguishability*, so arguing about it gives us important insights

• Three examples
  - The good (helpful) 😊
  - The bad (limiting) 😞
  & The ???... 😊
Traces of a Concurrent System
Projecting on Thread-Local Views

- read X → write X
- CAS
- read X → write Y
- failed CAS
Indistiguishability: Same Local Views

- read X
- write X
- CAS
- read X
- write Y
- failed CAS
Indistinguishable Traces: Same Local Views
Indistinguishable Traces: Same Local Views
1. Reductions for Serializability 😊

Static analysis of concurrent data structures, by **sequential reductions**:

Consider only **sequential traces**, and deduce properties in **all traces**

Attiya, Ramalingam, Rinetzky: Sequential verification of serializability. POPL 2010
Serializability

interleaved trace

operation

local views

complete non-interleaved trace

[Papadimitriou '79]
Serializability $\iff$ Sequential Reduction

Concurrent serializable code $M$, local property $\phi$

- Holds in a trace iff holds in all indistinguishable traces

[Panadimitriou '79] easily imply

$\phi$ holds in all traces of $M$ iff $\phi$ holds in all complete non-interleaved traces of $M$

How to check $M$ is serializable, w/o considering all traces?
Disciplined Programming with Locks

**Locking protocol** ensures conflict serializability
- two-phase locking (2PL), tree locking (TL),
  (dynamic) DAG locking

Verify that M **respects** a local locking protocol
- Depending only on thread’s local variables & global variables locked by it
- Not centralized concurrency control monitor!

Considering only non-interleaved traces
Our Contribution: First Step

Complete non-interleaved traces of M

A local conflict serializable locking policy is respected in all traces iff it is respected in all non-interleaved traces

A local property holds in all traces iff it holds in all non-interleaved traces
Reduction to Non-Interleaved Traces: Idea

Let $\sigma$ be the \textbf{shortest} trace that violates the locking protocol $LP$

$\Rightarrow \sigma'$ follows $LP$, guarantees conflict-serializability

\[ \sigma \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad (t,e) \]

\[ \sigma' \]

\[ \sigma' \]
Reduction to Non-Interleaved Traces: Idea

Let $\sigma$ be the **shortest** trace that violates the locking protocol $LP$
$\Leftrightarrow \sigma'$ follows $LP$, guarantees conflict-serializability
$\Leftrightarrow \exists$ non-interleaved trace **indistinguishable** from $\sigma'$

\[
\begin{align*}
\sigma & \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow (t,e) \\
\sigma' & \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \\
\sigma'_{ni} & \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow
\end{align*}
\]
Reduction to Non-Interleaved Traces: Idea

Let $\sigma$ be the **shortest** trace that violates the locking protocol $LP$.

$\Rightarrow$ $\sigma'$ follows $LP$, guarantees conflict-serializability.

$\Rightarrow$ $\exists$ non-interleaved trace **indistinguishable** from $\sigma'$

$\sigma$: 

$\sigma'$: 

$\sigma_{ni}$: 

$\Rightarrow$ $\exists$ non interleaved trace (indistinguishable from $\sigma$)

where $LP$ is violated.
Further reduction

**Almost-complete non-interleaved traces**

A *local conflict serializable locking policy* is respected in all traces iff it is respected in all almost-complete non-interleaved traces

Need to argue about termination
2. When are barriers necessary?

Expensive memory ordering should be enforced in order to ensure correctness of certain concurrent data structures.

Attiya, Guerraoui, Hendler, Kuznetsov, Michael, Vechev: Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. POPL 2011
The Result & Its Scope

• Concurrent data types:
  – **Strongly non-commutative** operations
    • Operations A and B s.t. A influences B, and B influences A
    • E.g., two deq operations, counters, hash tables, trees,…
  – Serializable solo-terminating implementations

• Mutual exclusion

Any concurrent program for these problems must use **read-after-write** unless it has
**atomic-write-after-read**
What this Means?

Multicores issue memory accesses **out of order**, to compensate for slow writes.

In particular (and very common)

**Issue a read before an earlier write**, if they access different locations.
Avoiding Out-of-Order Execution

Insert read-after-write (RAW) fence

Use atomic-write-after-read (AWAR)
E.g., CAS, test&set, fetch&add,…

RAW fences / AWAR are ~60 slower than (remote) memory accesses
Proof: Must Write

If a deq does not write, it does not influence other operations.
Proof: Must Write

If a deq does not write, it does not influence other operations.

Indistinguishable from a trace where deq’s are exchanged (and 1 is returned twice)
Proof: Must Also Read

If a deq does not read, it is not influenced by other operations

Indistinguishable from a trace where deq’s are exchanged
Close-Up on the 1st Dequeue

deq

1

first shared write
Close-Up on the 1\textsuperscript{st} Dequeue

\[ \text{deq} \quad \begin{array}{c} \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \end{array} \quad 1 \]

\text{write} \quad X

no read from \( Y \neq X \)
Covering Leads to Indistinguishability

No legal serialization (1 is dequeued twice)
3. Substituting TM for atomic blocks

Opaque transactional memory is equivalent to atomic blocks in concurrent programs

Programming with Atomic Blocks

\begin{align*}
g &: = 0; \\
r &: = \text{atomic} \\
\quad x &: = A.\text{write}(2); \\
\quad y &: = B.\text{write}(4);} \\
\text{If } (r = \text{commit}) \text{ then} \\
\quad g &: = 1 \\
\text{else } e &: = x \\
s &: = \text{abort} \\
\text{while } (s \neq \text{commit}) \text{ do} \\
\quad s &: = \text{atomic} \\
\quad \quad u &: = A.\text{read}(); \\
\quad \quad v &: = B.\text{read}(); \\
\quad z &: = g; \\
\text{if } (z = 1) \text{ then} \\
\quad \quad \text{three} &: = 6 / (v - u)
\end{align*}
Programming with Atomic Blocks

Design for an abstract Transactional Memory, assuming **code blocks** that execute **atomically**

\[
g := 0; \\
r := \text{atomic}\{ \\
    x := A.\text{write}(2); \\
    y := B.\text{write}(4); \\
\}; \\
\text{If } (r = \text{commit}) \text{ then} \\
    g := 1 \\
\text{else } e := x
\]

\[
s := \text{abort} \\
\text{while } (s \neq \text{commit}) \text{ do} \\
    s := \text{atomic}\{ \\
        u := A.\text{read}(); \\
        v := B.\text{read}(); \\
    \}; \\
z := g; \\
\text{if } (z = 1) \text{ then} \\
    \text{three} := 6 / (v - u)
\]
Programming with Atomic Blocks

Execute with a concrete TM implementation, replacing atomic blocks with transactions

```
g := 0;
r := atomic{
x := A.write(2);
y := B.write(4)};
If (r = commit) then
  g := 1
else e := x
```

```
s := abort
while (s ≠ commit) do
  s := atomic{
    u := A.read();
v := B.read()};
z := g;
  if (z = 1) then
    three := 6 / (v - u)
```

$\text{TM}_c$
Concrete TMs

$TM_c$ is a library for read, write, commit, ...

**History**: invocations and responses between the program and the $TM_c$

$(t_1, \text{begintx})(t_1, \text{ok}) \ldots (t_2, \text{call.f}(3))(t_1, \text{tryCommit})(t_2, \text{ret}(3)) \ (t_1, \text{abort}) \ldots$
TM Consistency Conditions

Restrict the possible histories, e.g.

- **Opacity** [Guerraoui & Kapalka, ’08]
- **Virtual World Consistency** [Imbs et al. ’09]
- **TMS** [Doherty et al. ’09]

But which of them is **THE RIGHT ONE**?

(t1,begintx)(t1,ok) ... (t2,call.f(3))(t1,tryCommit)(t2,ret(3)) (t1,abort) ...
TM Consistency Conditions

Restrict the possible histories, e.g.

- **Opacity** [Guerraoui & Kapalka, ’08]
- **Virtual World Consistency** [Imbs et al. ’09]
- **TMS** [Doherty et al. ’09]

But which of them is **THE RIGHT ONE**?

- Ensures $\text{TM}_C$ replaces $\text{TM}_A$ correctly (**soundness**)
- Enforces minimal restrictions (**completeness**)

**Observational refinement:** Programs (in some set) have the same **views** under $\text{TM}_C$ and $\text{TM}_A$
Opacity Relation $H \subseteq_{\text{OPS}} S$

History $S$ preserves **per-thread order** and **order of non-overlapping** transactions in history $H$. 

---

**Diagram:**

- $T_1$: $w_1(x,1)$, $r_2(x,1)$, $c_2$.
- $T_2$: $r_3(x,0)$, $w_3(x,3)$.
- $T_3$: $r_3(x,0)$, $w_3(x,3)$.

---

- $T_1$: $w_1(x,1)$, $c_1$.
- $T_2$: $r_2(x,1)$, $c_2$.
- $T_3$: $w_3(x,3)$, $c_3$. 

---

- $T_1$: $w_1(x,1)$, $c_1$.
- $T_2$: $r_2(x,1)$, $c_2$.
- $T_3$: $w_3(x,3)$, $c_3$. 

---

- $T_1$: $w_1(x,1)$, $c_1$.
- $T_2$: $r_2(x,1)$, $c_2$.
- $T_3$: $w_3(x,3)$, $c_3$. 

Soundness: $H \subseteq_{OP} S \Rightarrow$

Observational Refinement

History $S$ preserves **per-thread order** and **order of non-overlapping** transactions in history $H$

- no nesting
- no privatization
- finite histories
Soundness: Proof Outline

[Fix a program and an initial state...]

Consider a trace \( \sigma \) of \( \text{TM}_C \) with history \( H \), and assume \( H \sqsubseteq_{\text{OPS}} S \) for some history \( S \) of \( \text{TM}_A \)

Construct a trace \( \tau' \approx \tau \) of \( \text{TM}_A \)

\( \Rightarrow \) Every view observed when running the program with \( \text{TM}_C \) is also observed with \( \text{TM}_A \)
How to Construct $\tau'$ From $\tau$?

From the trace $\tau$ of $\text{TMC}_C$ & the history $S$ of $\text{TM}_A$
construct a trace $\tau' \approx \tau$ of $\text{TM}_A$

Can gather together events of each atomic block
(between start & end of a transaction) since

- No access to global variables inside atomic blocks,
  only to transactional variables
- Changes to transactional variables impact other
  threads only at the end of a block
Completeness: $\equiv_{OP}$ is Necessary

- Construct a program $P_H$ for every history $H$
- Real-time order in every trace of $P_H$ must agree with the real-time order of the transactions in $H$
Summary

Indistinguishability partitions computations into classes

Reduce the difficulty of designing / verifying concurrent programs by picking / constructing a representative computation from each class to verify, or to show it violates desired properties

You can do it too...