The Semantics of Weak Persistency

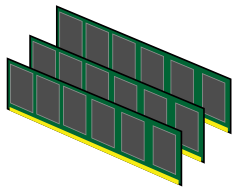
Viktor Vafeiadis

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

CALCO, 3 September 2021

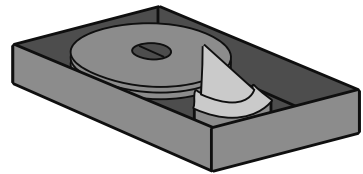
ERC-COG-2020 grant PERSIST, Mar'21–Feb'26

Traditional computer storage



RAM -
RANDOM ACCESS MEMORY
(MEMORY)

- fast byte-size access
- volatile storage



HDD -
HARD DISK DRIVE
(STORAGE)

- slow block-size access
- durable storage

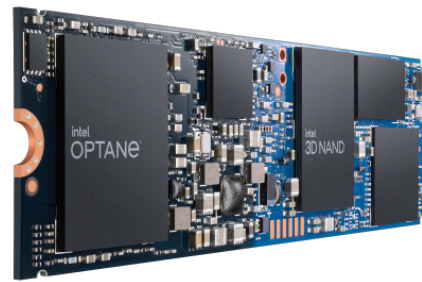
Non-volatile memory (NVM)

Fast byte-size access

- 3-10x slower than RAM
- 100x faster that HDD

Durable storage

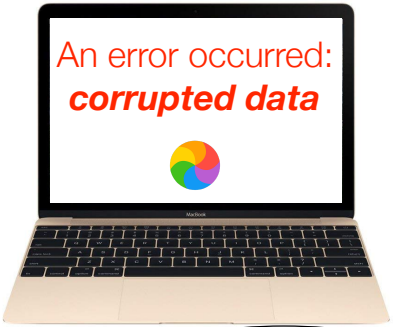
- Larger than RAM
- More energy-efficient than RAM



Verification is paramount!

Verification is paramount!

Verification is paramount!



Have you tried
turning it off and on again?

Verification is paramount!

Nothing changed!

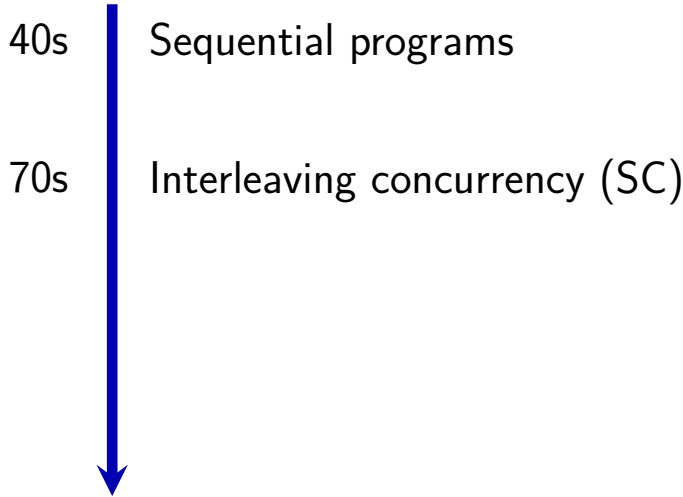
Roadmap

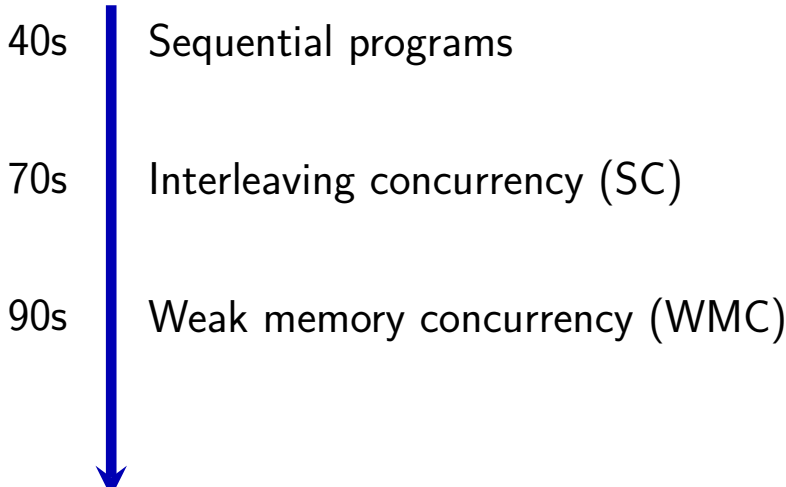
- What **semantics** do programs have?
  (operational, declarative/axiomatic)
- How can we **avoid** weak behaviours?
  (flushes, fences)
- When is a persistent algorithm **correct**?
  (invariants, persistent serializability/linearizability)
- What **techniques** can we use to establish correctness?
  (program logics, model checking)

Weak persistency semantics

40s

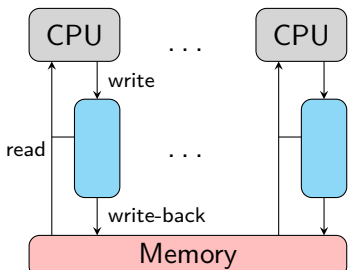
Sequential programs


40s
Sequential programs
70s
Interleaving concurrency (SC)


40s
Sequential programs
70s
Interleaving concurrency (SC)
90s
Weak memory concurrency (WMC)

Weak memory consistency

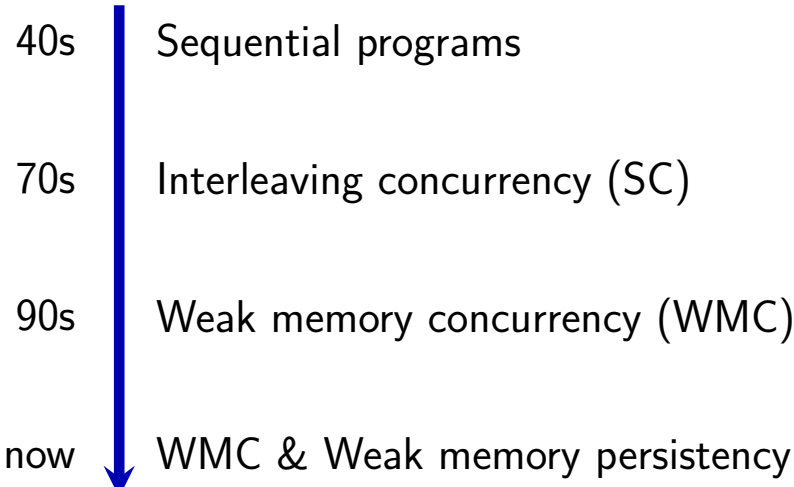
The x86-TSO model



Store buffering (SB)

Initially, $x = y = 0$.

$x := 1;$	$\|\|$	$y := 1;$
$a := y \quad // 0$	$\|\|$	$b := x \quad // 0$


40s
Sequential programs
70s
Interleaving concurrency (SC)
90s
Weak memory concurrency (WMC)
now
WMC & Weak memory persistency

Weak persistency

```
// x=y=0
x := 1;
y := 1;
```

```
// recovery routine
// x=y=1  OR  x=y=0  OR  x=1;y=0  OR  x=0;y=1
```

Weak persistency

```
// x=y=0
x := 1;
y := 1;
```

```
// recovery routine
// x=y=1  OR  x=y=0  OR  x=1;y=0  OR  x=0;y=1
```

!! Execution continues ***ahead of persistence***
— ***asynchronous*** persists

Weak persistency

```
// x=y=0
x := 1;
y := 1;
```

```
// recovery routine
// x=y=1   OR   x=y=0   OR   x=1;y=0   OR   x=0;y=1
```

!! Execution continues ***ahead of persistence***
— ***asynchronous*** persists

!! Writes may persist ***out of order***
— ***relaxed*** persists

Weak persistency

Consistency Model

the ***order*** in which writes
are ***made visible*** to other threads

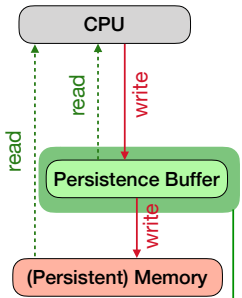
Persistency Model

the ***order*** in which writes
are ***persisted*** to NVM

NVM Semantics

Consistency + Persistency Model

Basic persistency model



`x:=1` : add `x:=1` to p-buffer

`a:=x` : if p-buffer contains `x`, read latest entry
else read from memory

p-buffer lost; memory ***retained***

Unbuffered at ***non-deterministic*** points in time

Buffering & unbuffering orders may disagree

Relaxed persists

```
// x=0;y=0
x := 1;
y := 1;
```

```
// recovery routine
// x=1;y=1 OR x=0;y=0 OR x=1;y=0 OR x=0;y=1
```

!! ***out of order persists***

Explicit persists?

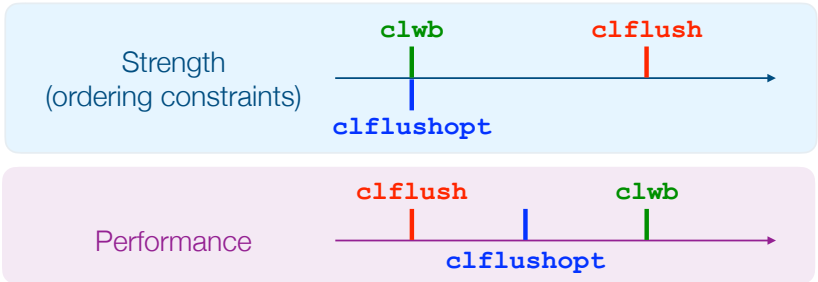
```
// x=0;y=0
x := 1;
persist x;
y := 1;
```

```
// recovery routine
// x=1;y=1 OR x=0;y=0 OR x=1;y=0 OR x=0;y=1
```

!! *out of order* ***persists***

☛ ***explicit persists?***

x86: clwb, clflushopt, clflush



- clwb and clflushopt: **same ordering** constraints
- clwb **does not invalidate** cache line
- clflushopt **invalidates** cache line
- clflush: **strongest** ordering constraints; **invalidates** cache line

Strong persists: **clflush**

```
// x=0;y=0
x := 1;
clflush x;
y := 1;
```

```
// recovery routine
// x=1;y=1 OR x=0;y=0 OR x=1;y=0 OR x=0;y=1
```

Weak persists: `clflushopt` & `clwb`

```
// x=0;y=0
x := 1;
clflushopt x / clwb x;
y := 1;
```

```
// recovery routine
// x=1;y=1 OR x=0;y=0 OR x=1;y=0 OR x=0;y=1
```

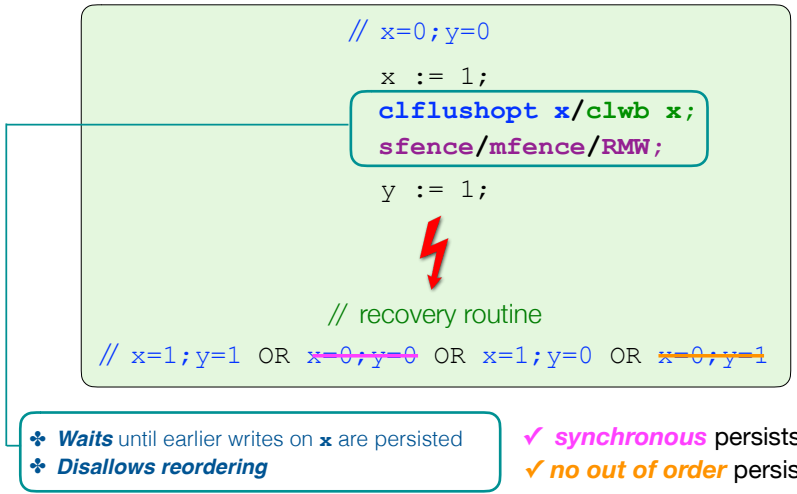
Weak persists: `clflushopt` & `clwb`

```
// x=0;y=0
x := 1;
clflushopt x / clwb x;
```

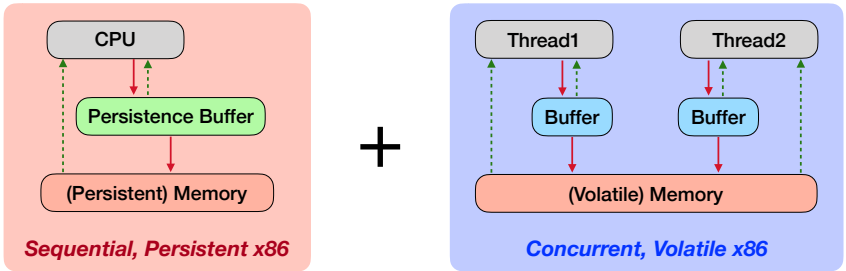
weak explicit persists of x86
are
asynchronous
and can themselves
persist out of order !

```
// x=1;y=
=0;y=1
```

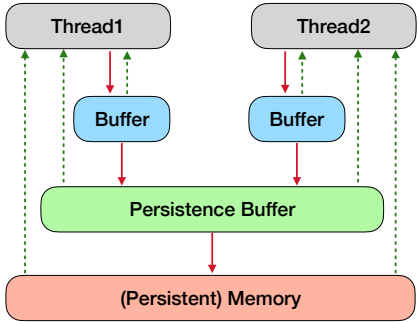
Solution: Persist sequences



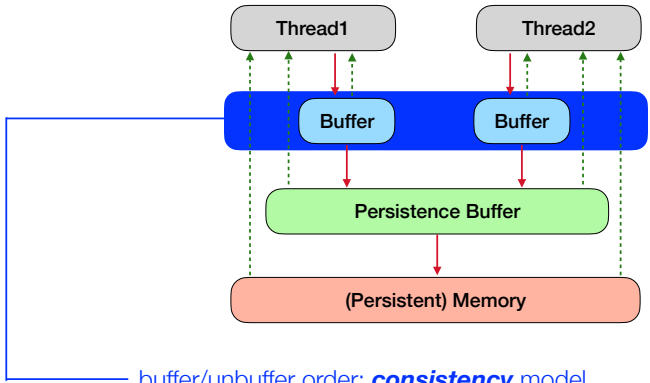
Adding weak memory consistency



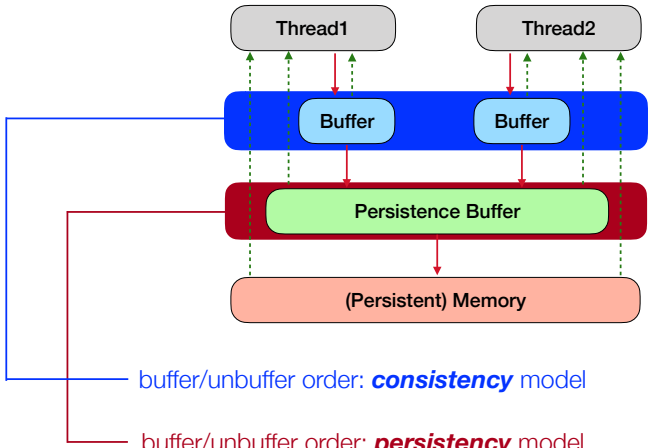
Persistent x86 (Px86)



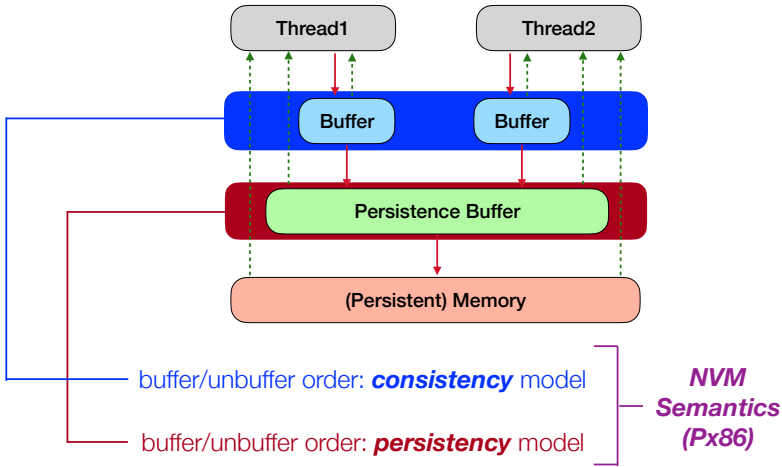
Persistent x86 (Px86)



Persistent x86 (Px86)

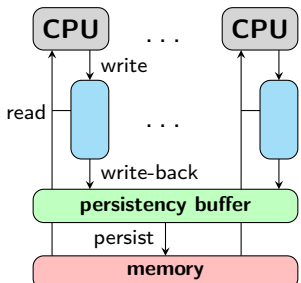


Persistent x86 (Px86)



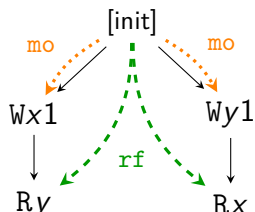
Two equivalent formal models

Operational



and

Declarative/Axiomatic



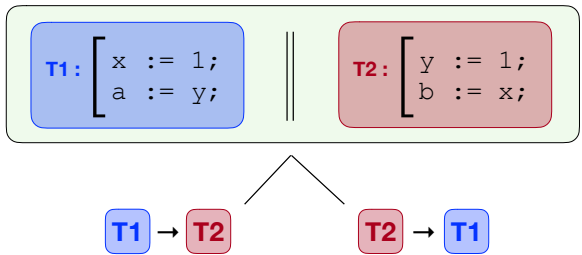
Roadmap

- ✓ What **semantics** do programs have?
  (operational, declarative/axiomatic)
- ✓ How can we **avoid** weak behaviours?
  (flushes, fences)
- ▶ When is a persistent algorithm **correct**?
  (invariants, persistent serializability/linearizability)
- ▶ What **techniques** can we use to establish correctness?
  (program logics, model checking)

Persistent serializability (PSER)

Serialisability (SER)

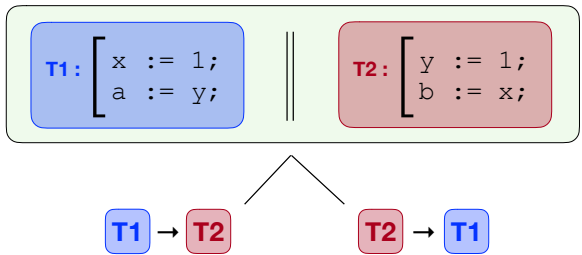
All transactions appear to ***execute*** in a sequential order



Persistent Serialisability (PSER)

All transactions appear to ***execute*** in a sequential order

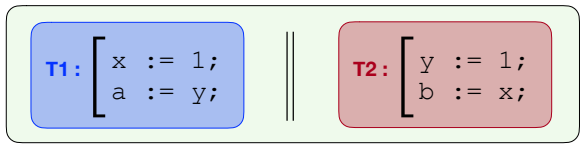
A prefix of transactions appears to ***persist*** in the ***same sequential order***

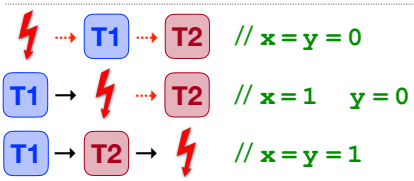


Persistent Serialisability (PSER)

All transactions appear to ***execute*** in a sequential order

A prefix of transactions appears to ***persist*** in the ***same sequential order***

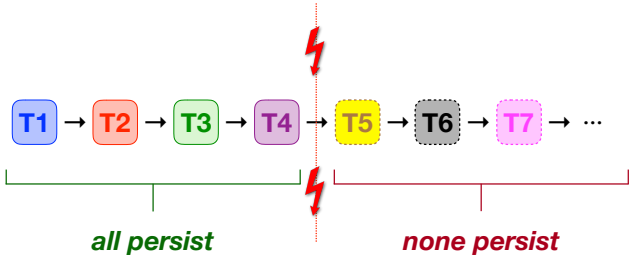




Persistent Serialisability (PSER)

All transactions appear to ***execute*** in a sequential order

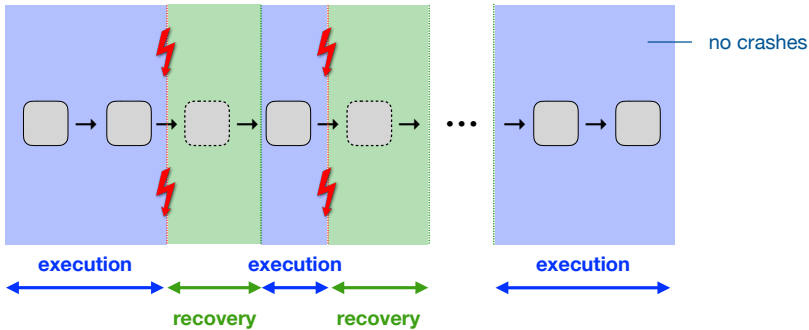
A prefix of transactions appears to ***persist*** in the ***same sequential order***



Persistent Serialisability (PSER)

All transactions appear to ***execute*** in a sequential order

A prefix of transactions appears to ***persist*** in the ***same sequential order*** in ***each era***



Persistent Serialisability (PSER)

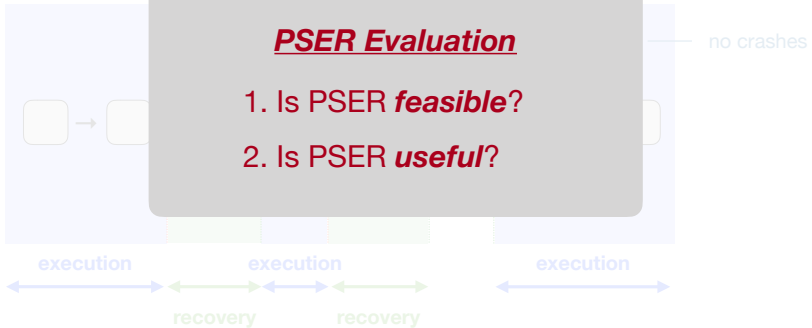
All transactions appear to ***execute*** in a sequential order

A prefix of transactions appears to ***persist*** in the ***same sequential order*** in ***each era***

Persistent Serialisability (PSER)

All transactions appear to ***execute*** in a sequential order

A prefix of transactions appears to ***persist*** in the ***same sequential order*** in ***each era***



PSER Evaluation

1. Is PSER ***feasible***?
2. Is PSER ***useful***?

Is PSER ***Feasible***?

✓ PSER ***implementation*** in ***x86, ARM***

Take ***SER*** Implementation — e.g. 2-PL

- ✢ add code for ***persistence***
- ✢ add code to ***log metadata*** for ***recovery***
- ✢ add ***recovery mechanism***

⇒

recovery mechanism

check log for ***incomplete*** transactions:

either ***complete***
or ***rollback***

Is PSER ***Useful***?

Given library *L* (e.g. queue library):

1. Take ***any*** correct ***sequential*** implementation of *L*
2. wrap each operation in a PSER transaction

⇒ ***correct, concurrent*** & ***persistent*** implementation of *L*

```
enq(q,v)=
  <enq_body>

deq(q)=
  <deq_body>
```

sequential queue imp.

```
enq(q,v)=
  pser{
    <enq_body> }

deq(q)=
  pser{
    <deq_body> }
```

correct
concurrent & ***persistent***
queue imp.

Roadmap

- ✓ What **semantics** do programs have?
  (operational, declarative/axiomatic)
- ✓ How can we **avoid** weak behaviours?
  (flushes, fences)
- ✓ When is a persistent algorithm **correct**?
  (invariants, persistent serializability/linearizability)
- ▶ What **techniques** can we use to establish correctness?
  (program logics, model checking)

Program logics for persistency

POG: Persistent Owicki-Gries reasoning

Instrument program with additional variables:

- volatile x_{v} – the latest observable value of x
- persisted x_{p} – the persisted value of x

Use OGRA, a slight weakening of Owicki-Gries that is sound under x86-TSO

$x := 1$

clflush x

$y := 1$

See OOPSLA'20; the presentation here is somewhat simplified

POG: Persistent Owicki-Gries reasoning

Instrument program with additional variables:

- volatile x_{v} – the latest observable value of x
- persisted x_{p} – the persisted value of x

Use OGRA, a slight weakening of Owicki-Gries that is sound under x86-TSO

$x := 1$

clflush x

$y := 1$

Parallel context:

$x_p := x_v$

$y_p := y_v$

See OOPSLA'20; the presentation here is somewhat simplified

POG: Persistent Owicki-Gries reasoning

Instrument program with additional variables:

- volatile x_{v} – the latest observable value of x
- persisted x_{p} – the persisted value of x

Use OGRA, a slight weakening of Owicki-Gries that is sound under x86-TSO

$x_v := 1$

$x_p := x_v$

$y_v := 1$

Parallel context:

$x_p := x_v$

$y_p := y_v$

See OOPSLA'20; the presentation here is somewhat simplified

POG: Persistent Owicki-Gries reasoning

Instrument program with additional variables:

- volatile x_{v} – the latest observable value of x
- persisted x_{p} – the persisted value of x

Use OGRA, a slight weakening of Owicki-Gries that is sound under x86-TSO

$\{x_v = 0 \wedge x_p = 0 \wedge y_v = 0 \wedge y_p = 0\}$

$x_v := 1$

$x_p := x_v$

$y_v := 1$

Parallel context:

$x_p := x_v$

$y_p := y_v$

See OOPSLA'20; the presentation here is somewhat simplified

POG: Persistent Owicki-Gries reasoning

Instrument program with additional variables:

- volatile x_{v} – the latest observable value of x
- persisted x_{p} – the persisted value of x

Use OGRA, a slight weakening of Owicki-Gries that is sound under x86-TSO

$\{x_v = 0 \wedge x_p = 0 \wedge y_v = 0 \wedge y_p = 0\}$
$x_v := 1$
$\{x_v = 1 \wedge x_p \in \{0, 1\} \wedge y_v = 0 \wedge y_p = 0\}$
$x_p := x_v$

$y_v := 1$

Parallel context:

$x_p := x_v$
$y_p := y_v$

See OOPSLA'20; the presentation here is somewhat simplified

POG: Persistent Owicki-Gries reasoning

Instrument program with additional variables:

- volatile x_{v} – the latest observable value of x
- persisted x_{p} – the persisted value of x

Use OGRA, a slight weakening of Owicki-Gries that is sound under x86-TSO

$$\{x_v = 0 \wedge x_p = 0 \wedge y_v = 0 \wedge y_p = 0\}$$
$$x_v := 1$$
$$\{x_v = 1 \wedge x_p \in \{0, 1\} \wedge y_v = 0 \wedge y_p = 0\}$$
$$x_p := x_v$$
$$\{x_v = 1 \wedge x_p = 1 \wedge y_v = 0 \wedge y_p = 0\}$$
$$y_v := 1$$

Parallel context:

$$x_p := x_v$$
$$y_p := y_v$$

See OOPSLA'20; the presentation here is somewhat simplified

POG: Persistent Owicki-Gries reasoning

Instrument program with additional variables:

- volatile x_{v} – the latest observable value of x
- persisted x_{p} – the persisted value of x

Use OGRA, a slight weakening of Owicki-Gries that is sound under x86-TSO

$$
\begin{aligned}
&\{x_v = 0 \wedge x_p = 0 \wedge y_v = 0 \wedge y_p = 0\}\\
&x_v := 1\\
&\{x_v = 1 \wedge x_p \in \{0,1\} \wedge y_v = 0 \wedge y_p = 0\}\\
&x_p := x_v\\
&\{x_v = 1 \wedge x_p = 1 \wedge y_v = 0 \wedge y_p = 0\}\\
&y_v := 1\\
&\{x_v = 1 \wedge x_p = 1 \wedge y_v = 1 \wedge y_p \in \{0,1\}\}
\end{aligned}
$$

Parallel context:

$$
\begin{aligned}
x_p &:= x_v\\
y_p &:= y_v
\end{aligned}
$$

See OOPSLA'20; the presentation here is somewhat simplified

Model checking for persistency

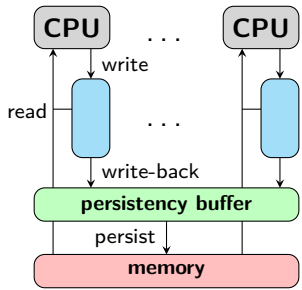
Software model checking

Given a program P and a property Φ:
Check that all executions of P satisfy Φ.

- P - shared-memory concurrency + flushes/fences
- Φ - safety assertion, invariant over persisted state
- 'Stateless' enumeration of all reachable program states

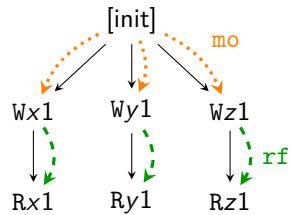
Which kind of model to use?

Operational



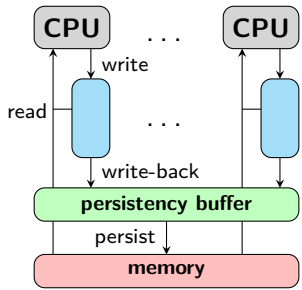
or

Declarative



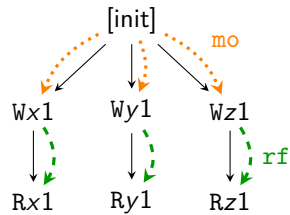
Which kind of model to use?

Operational



or

Declarative

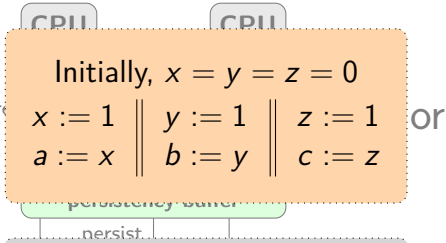


- ✓ Easy to build an interpreter
- ✗ Tied to a specific model
- ✗ Many redundant executions

- ✗ Not immediately executable
- ✓ Largely model-agnostic
- ✓ Almost no redundancy

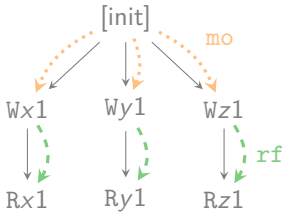
Which kind of model to use?

Operational



or

Declarative



✓ [illegible]

✗ Tied to a specific model

✗ Many redundant executions

✗ Not immediately executable

✓ Largely model-agnostic

✓ Almost no redundancy

Using declarative models pays off!

RMEM benchmarks

	RMEM	GenMC
DQ/211-2-1	172.34	0.17
DQ-opt/211-2-1	770.45	0.17
STC/210-011-000	1100.35	0.06
STC-opt/210-011-000	1154.68	0.10
QU/100-100-010	1099.20	0.06
QU-opt/100-100-010	*	0.06

Verification times in seconds

- RMEM — operational ARM model with a few optimisations
- GenMC — declarative memory model

Basic model checking algorithm

Construct all consistent execution graphs incrementally

- Fix insertion order (e.g. increasing thread ID order)

See POPL'18 and PLDI'19 papers for more details

Basic model checking algorithm

Construct all consistent execution graphs incrementally

- Fix insertion order (e.g. increasing thread ID order)

$$x := 1 \parallel a := x$$

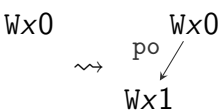
$\mathtt{W}x0$

Basic model checking algorithm

Construct all consistent execution graphs incrementally

- Fix insertion order (e.g. increasing thread ID order)

$$x := 1 \parallel a := x$$



See POPL'18 and PLDI'19 papers for more details

Basic model checking algorithm

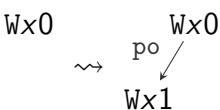
Construct all consistent execution graphs incrementally

- Fix insertion order (e.g. increasing thread ID order)

When adding a read r:

- Consider all possible writes that r could read from.

$$x := 1 \;\|\; a := x$$



See POPL'18 and PLDI'19 papers for more details

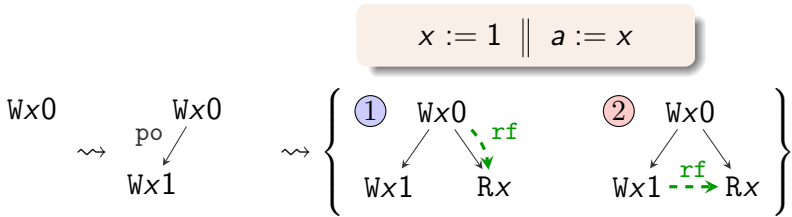
Basic model checking algorithm

Construct all consistent execution graphs incrementally

- Fix insertion order (e.g. increasing thread ID order)

When adding a read r:

- Consider all possible writes that r could read from.



See POPL'18 and PLDI'19 papers for more details

Basic model checking algorithm

Construct all consistent execution graphs incrementally

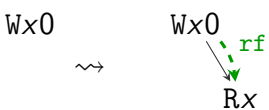
- Fix insertion order (e.g. increasing thread ID order)

When adding a read r:

- Consider all possible writes that r could read from.

$$x := 1 \parallel a := x$$

Add $a := x$ first



See POPL'18 and PLDI'19 papers for more details

Basic model checking algorithm

Construct all consistent execution graphs incrementally

- Fix insertion order (e.g. increasing thread ID order)

When adding a read r:

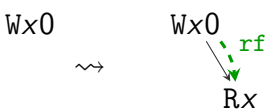
- Consider all possible writes that r could read from.

When adding a write w:

- Revisit existing reads to instead read from w.

$$x := 1 \parallel a := x$$

Add $a := x$ first



See POPL'18 and PLDI'19 papers for more details

Basic model checking algorithm

Construct all consistent execution graphs incrementally

- Fix insertion order (e.g. increasing thread ID order)

When adding a read r:

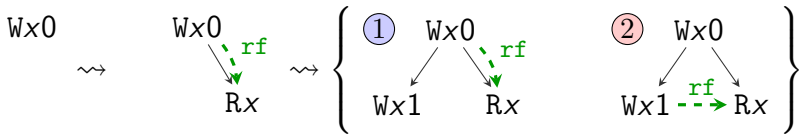
- Consider all possible writes that r could read from.

When adding a write w:

- Revisit existing reads to instead read from w.

$x := 1 \parallel a := x$

Add $a := x$ first



See POPL'18 and PLDI'19 papers for more details

Checking persistency assertions

$$x := 1 \parallel y := 2 \parallel z := 3$$

Check that $y \neq 1$ after a crash

Checking persistency assertions

$$x := 1 \parallel y := 2 \parallel z := 3$$

Check that $y \neq 1$ after a crash

Simple approach

- Enumerate all post-crash states of a program
- Check that each persisted state satisfies the assertion

Total p-ordering

- In which order did the durable events persist?
- Take any prefix of that order
- In total, $4 \times 3! =$ **24 cases**

Checking persistency assertions

$$x := 1 \parallel y := 2 \parallel z := 3$$

Check that $y \neq 1$ after a crash

Simple approach

- Enumerate all post-crash states of a program
- Check that each persisted state satisfies the assertion

Total p-ordering

- In which order did the durable events persist?
- Take any prefix of that order
- In total, $4 \times 3! =$ **24 cases**

Partial p-ordering

- Which durable events have persisted?
- Their relative ordering is irrelevant
- In total, $2^3 =$ **8 cases**

Checking persistency assertions

$$x := 1 \parallel y := 2 \parallel z := 3$$

Check that $y \neq 1$ after a crash

Simple approach

- Enumerate all post-crash states of a program
- Check that each persisted state satisfies the assertion

Total p-orderin

The assertion mentions only the variable y; whether $x := 1$ and $z := 3$ persist is irrelevant

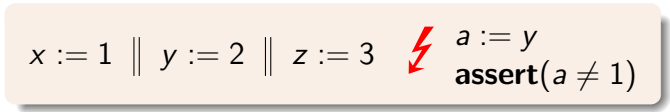
- In which or… events persist?
- Take any prefix of that order
- In total, $4 \times 3! =$ **24 cases**

…ents have persisted?

- Their relative ordering is irrelevant
- In total, $2^3 =$ **8 cases**

Recovery observer

Treat the persistency assertions as an additional thread



. . . with somewhat different consistency axioms

- Enforce snapshot atomicity
  (all recovery reads of y read from the same write)
- Observing a durable event $\Rightarrow$ observe all events persisted before it

In our example, there are only **2 execution graphs**

See POPL'21 paper on PerSeVerE for more details

Summary

- ✓ What **semantics** do programs have?
  (operational, declarative/axiomatic)
- ✓ How can we **avoid** weak behaviours?
  (flushes, fences)
- ✓ When is a persistent algorithm **correct**?
  (invariants, persistent serializability/linearizability)
- ✓ What **techniques** can we use to establish correctness?
  (program logics, model checking)