On Parallel Snapshot Isolation and Release/Acquire Consistency

Azalea Raad Ori Lahav Viktor Vafeiadis

MPI-SWS
Tel Aviv University

Thursday 19 April
ESOP 2018
Thessaloniki, Greece

What is STM?

Concurrency Control via ***transactions***

- ***atomic*** unit of work (set of operations) on shared data
- ***all-or-nothing***

```
// x = y = 0

T: [ x := 1;
     y := 1;

// x = y = 0    OR    x = y = 1
```

Which STM?

Strong consistency - inefficient

- serialisability
- strong serialisability
- …

Weak consistency

- snapshot isolation (SI)
- parallel snapshot isolation (PSI)
- …

Which STM?

Strong consistency - inefficient

- serialisability
- strong serialisability
- …

Weak consistency

- snapshot isolation (SI)
- **parallel snapshot isolation (PSI)**
  - ➡ efficient, monotonic
- …

STM Context

- Shared memory setting

  (with ***weak*** memory consistency)

- ***Mixed*** accesses to shared data

  (transactional and non-transactional)

- ***Cannot instrument*** non-transactional accesses

  (weak isolation)

PSI STM Desiderata

PSI STM Desiderata

- *Declarative* semantics

- Reference implementation (*operational* semantics)
  - ➡ ***Sound***: Behaviours(imp) ⊆ Behaviours(spec)
  - ➡ ***Complete***: Behaviours(spec) ⊆ Behaviours(imp)

PSI STM Desiderata

- *Declarative* semantics

- Reference implementation (*operational* semantics)
  - ➡ ***Sound***: Behaviours(imp) ⊆ Behaviours(spec)
  - ➡ ***Complete***: Behaviours(spec) ⊆ Behaviours(imp)

- *Declarative* semantics with ***mixed*** accesses

- Reference implementation with ***mixed*** accesses (*operational* semantics)
  - ➡ ***Sound***: Behaviours(imp) ⊆ Behaviours(spec)
  - ➡ ***Complete***: Behaviours(spec) ⊆ Behaviours(imp)

PSI STM Desiderata

✓ *Declarative* semantics

✘ Reference implementation (*operational* semantics)

- ➡ ***Sound***: Behaviours(imp) ⊆ Behaviours(spec)
- ➡ ***Complete***: Behaviours(spec) ⊆ Behaviours(imp)

✘ *Declarative* semantics with ***mixed*** accesses

✘ Reference implementation with ***mixed*** accesses (operational semantics)

- ➡ ***Sound***: Behaviours(imp) ⊆ Behaviours(spec)
- ➡ ***Complete***: Behaviours(spec) ⊆ Behaviours(imp)

PSI STM Desiderata

✓ *Declarative* semantics

✓ Reference implementation (*operational* semantics)

- ➡ ***Sound***: Behaviours(imp) ⊆ Behaviours(spec)
- ➡ ***Complete***: Behaviours(spec) ⊆ Behaviours(imp)

✓ *Declarative* semantics with ***mixed*** accesses ⟹ ***RPSI*** (‘robust’ PSI)

✓ Reference implementation with ***mixed*** accesses (operational semantics)

- ➡ ***Sound***: Behaviours(imp) ⊆ Behaviours(spec)
- ➡ ***Complete***: Behaviours(spec) ⊆ Behaviours(imp)

PSI STM Desiderata

✓ *Declarative* semantics

✓ Reference implementation (*operational* semantics)

- ➡ ***Sound***: Behaviours(imp) ⊆ Behaviours(spec)
- ➡ ***Complete***: Behaviours(spec) ⊆ Behaviours(imp)

✓ *Declarative* semantics with ***mixed*** accesses ⟹ ***RPSI*** (‘robust’ PSI)

✓ Reference implementation with ***mixed*** accesses (operational semantics)

- ➡ ***Sound***: Behaviours(imp) ⊆ Behaviours(spec)
- ➡ ***Complete***: Behaviours(spec) ⊆ Behaviours(imp)

What is PSI?

r1

x=y=0

r2

x=y=0

r3

x=y=0

What is PSI?

r1

x=y=0

r2

x=y=0

r3

x=y=0

```
[ x := 1;
```

What is PSI?

r1

x=y=0

r2

x=y=0

r3

x=y=0

S1: x=y=0

```
[ x := 1;
```

What is PSI?

r1

x=y=0

r2

x=y=0

r3

x=y=0

S1: x=y=0

```
[ x := 1;
```

C1: x=1

What is PSI?

r1

x=y=0

r2

x=y=0

r3

x=y=0

S1: x=y=0

```
[ x := 1;
```

C1: x=1

ww-conflict? no

What is PSI?

r1

x=y=0

r2

x=1;y=0

r3

x=y=0

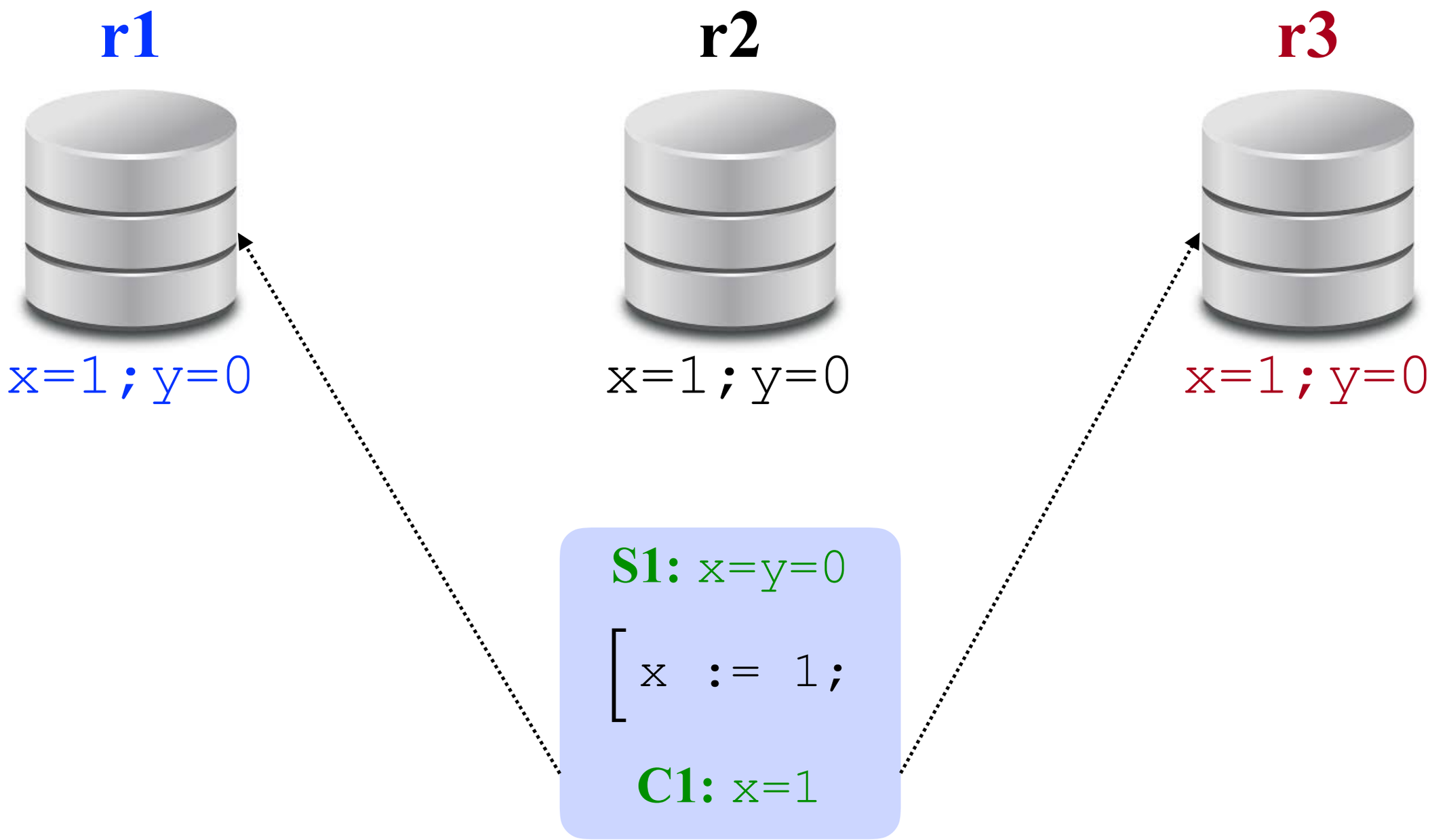
S1: x=y=0

[x := 1;

C1: x=1

ww-conflict? no

What is PSI?



PSI Litmus Tests

Write skew / SB

T1 :

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

‖

T2 :

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

r1

x=y=0

r2

x=y=0

All variables are initially 0; comments specify the values read

PSI Litmus Tests

Write skew / SB

T1 :

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

‖

T2 :

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

r1

x=y=0

r2

x=y=0

```
x := 1;
a := y;
```

PSI Litmus Tests

Write skew / SB

T1 :

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

‖

T2 :

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

r1

x=y=0

r2

x=y=0

S1: x=y=0

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

C1: x=1

PSI Litmus Tests

Write skew / SB

T1 :

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

‖

T2 :

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

r1

x=y=0

r2

x=y=0

S1: x=y=0

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

C1: x=1

S2: x=y=0

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

C2: y=1

PSI Litmus Tests

Write skew / SB

T1 :

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

‖

T2 :

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

r1

x=y=0

r2

x=y=0

S1: x=y=0

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

C1: x=1

ww-conflict? no

ww-conflict? no

S2: x=y=0

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

C2: y=1

PSI Litmus Tests

Write skew / SB

T1 :

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

‖

T2 :

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

r1

x=1;y=1

r2

x=1;y=1

S1: x=y=0

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

C1: x=1

ww-conflict? no

ww-conflict? no

S2: x=y=0

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

C2: y=1

PSI Litmus Tests

Long fork / IRIW

T1 :

```
x := 1;
```

T2 :

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

T3 :

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

T4 :

```
y := 1;
```

r1

x=y=0

r2

x=y=0

r3

x=y=0

r4

x=y=0

All variables are initially 0; comments specify the values read

PSI Litmus Tests

Long fork / IRIW

T1 :

```
[ x := 1;
```

T2 :

```
[ a := x; //1
  b := y; //0
```

T3 :

```
[ c := y; //1
  d := x; //0
```

T4 :

```
[ y := 1;
```

r1

x=y=0

r2

x=y=0

r3

x=y=0

r4

x=y=0

S1: x=y=0

```
[ x := 1;
```

C1: x=1

PSI Litmus Tests

Long fork / IRIW

T1 :

```
[ x := 1;
```

T2 :

```
[ a := x; // 1
  b := y; // 0
```

T3 :

```
[ c := y; // 1
  d := x; // 0
```

T4 :

```
[ y := 1;
```

r1

x=y=0

r2

x=y=0

r3

x=y=0

r4

x=y=0

S1: x=y=0

```
[ x := 1;
```

C1: x=1

S4: x=y=0

```
[ y := 1;
```

C4: y=1

PSI Litmus Tests

Long fork / IRIW

T1 :

```
[ x := 1;
```

T2 :

```
[ a := x; //1
  b := y; //0
```

T3 :

```
[ c := y; //1
  d := x; //0
```

T4 :

```
[ y := 1;
```

r1

x=y=0

r2

x=y=0

r3

x=y=0

r4

x=y=0

S1: x=y=0

```
[ x := 1;
```

C1: x=1

ww-conflict? no

S4: x=y=0

```
[ y := 1;
```

C4: y=1

ww-conflict? no

PSI Litmus Tests

Long fork / IRIW

T1 :

```
[ x := 1;
```

T2 :

```
[ a := x; //1
  b := y; //0
```

T3 :

```
[ c := y; //1
  d := x; //0
```

T4 :

```
[ y := 1;
```

r1

x=1;y=0

r2

x=y=0

r3

x=y=0

r4

x=0;y=1

S1: x=y=0

```
[ x := 1;
```

C1: x=1

ww-conflict? no

S4: x=y=0

```
[ y := 1;
```

C4: y=1

ww-conflict? no

PSI Litmus Tests

Long fork / IRIW

T1 :

```
[ x := 1;
```

T2 :

```
[ a := x; //1
  b := y; //0
```

T3 :

```
[ c := y; //1
  d := x; //0
```

T4 :

```
[ y := 1;
```

r1

x=1;y=0

r2

x=1;y=0

r3

x=y=0

r4

x=0;y=1

1

S1: x=y=0

```
[ x := 1;
```

C1: x=1

ww-conflict? no

S4: x=y=0

```
[ y := 1;
```

C4: y=1

ww-conflict? no

PSI Litmus Tests

Long fork / IRIW

T1 :

```
[ x := 1;
```

T2 :

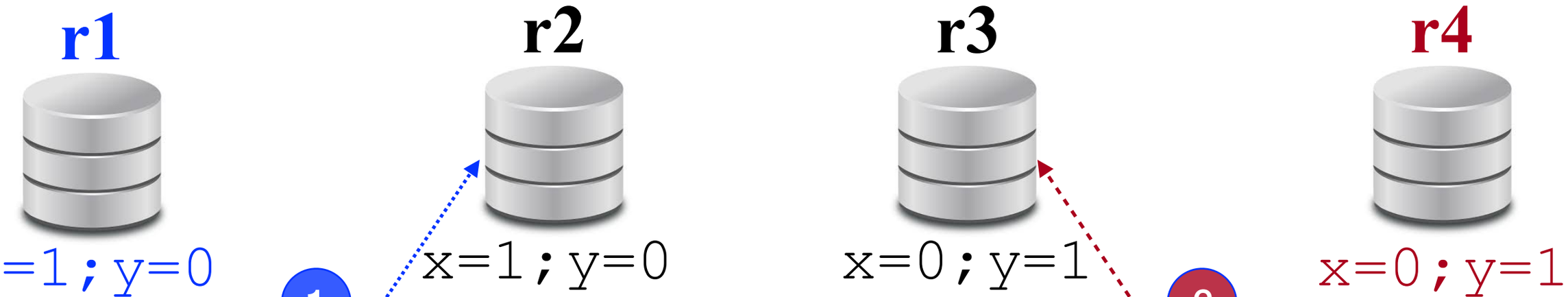
```
[ a := x; //1
  b := y; //0
```

T3 :

```
[ c := y; //1
  d := x; //0
```

T4 :

```
[ y := 1;
```



ww-conflict? no

ww-conflict? no

PSI Litmus Tests

Long fork / IRIW

T1 :

```
[ x := 1;
```

T2 :

```
[ a := x; //1
  b := y; //0
```

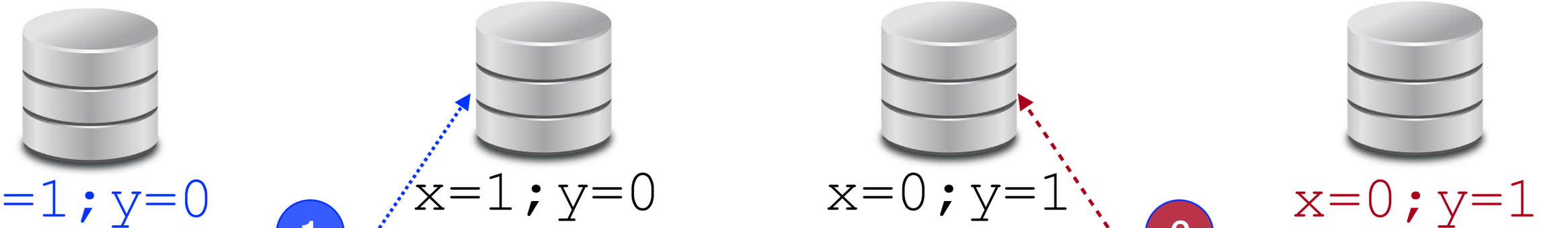
T3 :

```
[ c := y; //1
  d := x; //0
```

T4 :

```
[ y := 1;
```

r1 **r2** **r3** **r4**



S1: x=y=0

```
[ x := 1;
```

C1: x=1

ww-conflict? no

S2: x=1;y=0

T2

C2: null

S3: x=0;y=1

T3

C3: null

S4: x=y=0

```
[ y := 1;
```

C4: y=1

ww-conflict? no

PSI Litmus Tests

Long fork / IRIW

T1 :

```
[ x := 1;
```

T2 :

```
[ a := x; //1
  b := y; //0
```

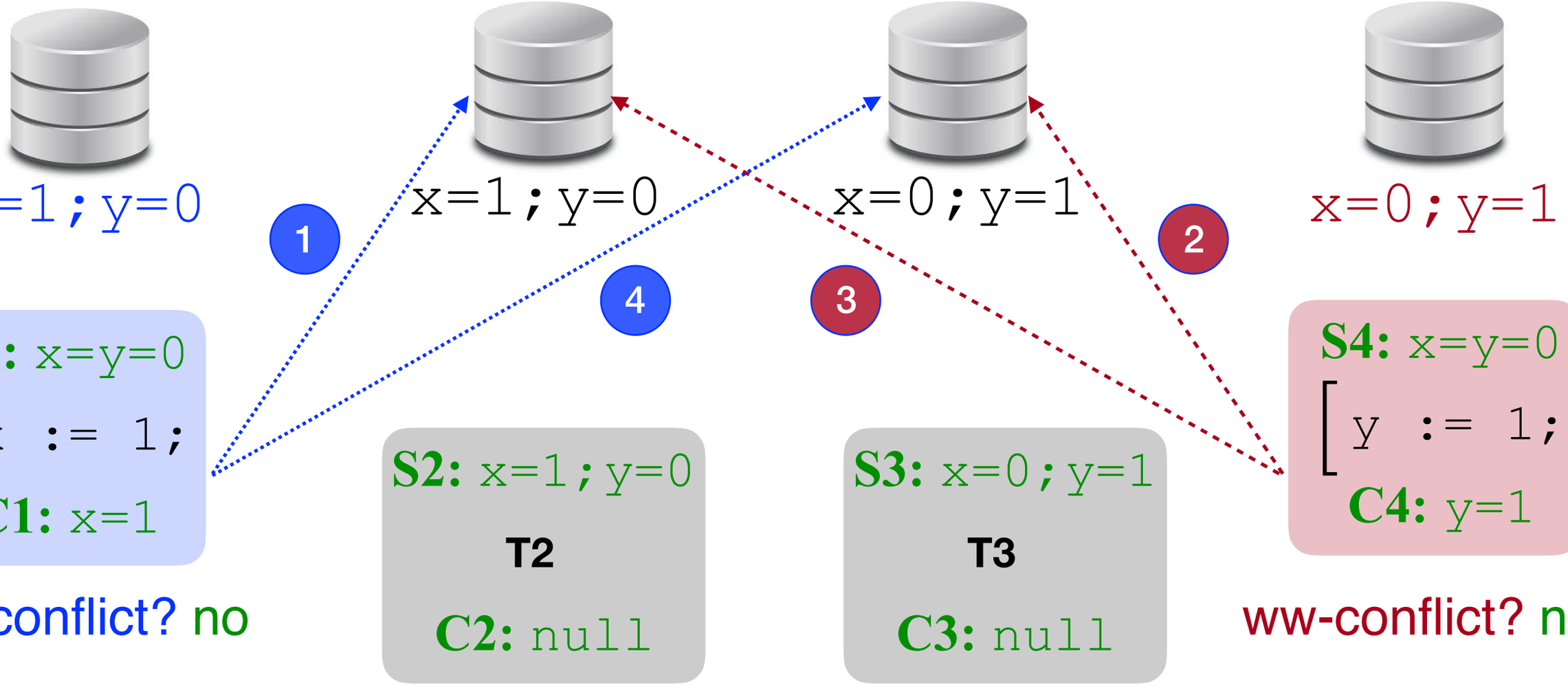
T3 :

```
[ c := y; //1
  d := x; //0
```

T4 :

```
[ y := 1;
```

r1 **r2** **r3** **r4**



Reasoning about
replicas and **propagation order**
is difficult

Reasoning about

replicas and **propagation order**

is difficult

⇒

Lock-based

PSI reference implementation

in

which memory model ?

PSI in ***which*** Memory Model?

- Sequential Consistency (SC) ?

PSI in ***which*** Memory Model?

✘ Sequential Consistency (SC) — too strong!

Long fork / IRIW

T1 :

```
x := 1;
```

T2 :

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

T3 :

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

T4 :

```
y := 1;
```

Reasoning about

replicas and **propagation order**

is difficult

⇒

Lock-based

PSI reference implementation

in

C11 **release/acquire** fragment

Which Locks for PSI?

Which Locks for PSI?

✗ ***Global*** lock

➡ *disjoint* accesses allowed

Which Locks for PSI?

✗ ***Global*** lock

➡ *disjoint* accesses allowed

- ***Per-location*** locks

Which Locks for PSI?

✗ ***Global*** lock

➡ *disjoint* accesses allowed

✗ ***Per-location*** locks

➡ concurrent reads allowed

Which Locks for PSI?

✗ ***Global*** lock

➡ *disjoint* accesses allowed

✗ ***Per-location*** locks

➡ concurrent reads allowed

- ***Per-location MRSW*** (multiple-readers-single-writer) locks

Which Locks for PSI?

✗ ***Global*** lock

➡ *disjoint* accesses allowed

✗ ***Per-location*** locks

➡ concurrent reads allowed

✗ ***Per-location MRSW*** (multiple-readers-single-writer) locks

➡ readers should *not synchronise* (e.g. IRIW)

Long fork / IRIW

T1 :

```
[ x := 1;
```

‖

T2 :

```
[ a := x; // 1
  b := y; // 0
```

‖

T3 :

```
[ c := y; // 1
  d := x; // 0
```

‖

T4 :

```
[ y := 1;
```

Which Locks for PSI?

✗ ***Global*** lock

➡ *disjoint* accesses allowed

✗ ***Per-location*** locks

➡ concurrent reads allowed

✗ ***Per-location MRSW*** (multiple-readers-single-writer) locks

➡ readers should *not synchronise* (e.g. IRIW)

✓ *Per-location* ***sequence*** locks

Sequence Locks

x_lock version *even* : lock free

x_lock version *odd* : lock taken

Sequence Locks

x_lock version *even* : lock free

x_lock version *odd* : lock taken

```
wlock(x){
 retry:
   vx := x_lock;
   if (is-odd(vx))
     goto retry;
   if(!CAS(x_lock, vx, vx+1))
     goto retry;
}

wunlock(x){
   vx := x_lock;
   x_lock := vx + 1;
}
```

Sequence Locks

x_lock version *even* : lock free

x_lock version *odd* : lock taken

```
snapshot(x){
 retry:
  // tentative version

    vx := x_lock;
    while (is-odd(vx))
      skip;

  // read x
    sx := x;
  // validate version

    if (vx != x_lock)
      goto retry;
}
```

Sequence Locks

x_lock version *even* : lock free

x_lock version *odd* : lock taken

```
snapshot(x){
 retry:
  // tentative version

    vx := x_lock;
    while (is-odd(vx))
      skip;

  // read x
    sx := x;
  // validate version

    if (vx != x_lock)
      goto retry;
}
```

```
snapshot(S){
 retry:
  // tentative versions
  for (x in S) {
    vx := x_lock;
    while (is-odd(vx))
      skip;
  }
  // read S
  for (x in S) sx := x;
  // validate versions
  for (x in S)
    if (vx != x_lock)
      goto retry;
}
```

PSI Reference Implementation

```
PSI(T) {
  // lock write set
  for(x∈WS) {
    wlock(x);
    if (x ∈RS) sx := x;
  }
  snapshot(RS\WS);
  ⟦ T ⟧ ;
  // unlock write set
  for(x∈WS) wunlock(x);
}
```

$$[\![a := x]\!] = a := sx$$

$$[\![x := a]\!] = x := a;\ sx := a$$

$$[\![S1;S2]\!] = [\![S1]\!] ; [\![S2]\!]$$

…

PSI Reference Implementation

```
PSI(T) {
  // lock write set
  for(x∈WS) {
    wlock(x);
    if (x ∈RS) sx := x;
  }
  snapshot(RS\WS);
  ⟦ T ⟧ ;
  // unlock write set
  for(x∈WS) wunlock(x);
}
```

```
⟦a := x⟧ = a := sx

⟦x := a⟧ = x := a; sx := a

⟦S1;S2⟧ = ⟦S1⟧ ; ⟦S2⟧

...
```

- ➡ ***Sound***: Behaviours(imp) ⊆ Behaviours(PSI_spec)
- ➡ ***Complete***: Behaviours(PSI_spec) ⊆ Behaviours(imp)

What about **mixed** accesses?

RPSI = PSI + **mixed** accesses

What about **mixed** accesses?

$$\text{acyclic}(\text{rpsi-hb}_{loc} \cup \text{mo} \cup \text{rb})$$

with

$$\text{rpsi-hb}; \text{rpsi-hb} \subseteq \text{rpsi-hb} \quad (\textsc{trans})$$

$$\text{po} \cup \text{rf} \cup \text{mo}_{\mathsf{T}} \subseteq \text{rpsi-hb} \quad (\textsc{psi-hb})$$

$$[E \setminus \mathcal{T}]; \text{rf}; \text{st} \subseteq \text{rpsi-hb} \quad (\textsc{nt-rf})$$

$$\text{st}; ([\mathcal{W}]; \text{st}; (\text{rpsi-hb} \setminus \text{st}); \text{st}; [\mathcal{R}])_{loc}; \text{st} \subseteq \text{rpsi-hb} \quad (\textsc{t-rf})$$

RPSI = PSI + **mixed** accesses

What about **mixed** accesses?

⇒

Lock-based

RPSI reference implementation

in

C11 **release/acquire** fragment

RPSI = PSI + **mixed** accesses

RPSI: PSI + Mixed Accesses

MP

```
x := 1;          ||   T:
y := 1;          ||   [ a := y; // 1
                 ||     b := x; // 0
```

RPSI: PSI + Mixed Accesses

MP

```
x := 1;          ‖  T:
y := 1;          ‖  [ a := y; // 1
                 ‖    b := x; // 0
```

```
snapshot(S){
  // tentative versions
 retry:
  for (x in S) {
    vx := x_lock;
    while (is-odd(vx))
       skip;
  }
  // read S
  for (x in S) sx := x;
  // validate versions
  for (x in S)
    if (vx != x_lock)
      goto retry;
}
```

RPSI: PSI + Mixed Accesses

MP

```
x := 1;          ||  T:
y := 1;          ||  [ a := y; // 1
                 ||    b := x; // 0
```

```
snapshot_RPSI(S){
  // tentative versions
 retry:
  for (x in S) {
    vx := x_lock;
    while (is-odd(vx))
       skip;
  }
  // read S
  for (x in S) sx := x;
  // validate versions & values
  for (x in S)
    if (vx != x_lock && sx != x)
      goto retry;
}
```

Solution: read every location ***twice***!

RPSI: PSI + Mixed Accesses

MP

x

y

1

0

Caveat: non-transactional writes with ***same value*** ***cannot race*** with transactions

```
snapshot_RPSI(S){
  // tentative versions
 retry:
  for (x in S) {
    vx := x_lock;
    while (is-odd(vx))
       skip;
  }
  // read S
  for (x in S) sx := x;
  // validate versions & values
  for (x in S)
    if (vx != x_lock && sx != x)
      goto retry;
}
```

Solution: read every location ***twice***!

RPSI Reference Implementation

```
RPSI(T) {
  // lock write set
  for(x∈WS) {
    wlock(x);
    if (x ∈RS) sx := x;
  }
  snapshot_RPSI(RS\WS);
  ⟦ T ⟧ ;
  // unlock write set
  for(x∈WS) wunlock(x);
}
```

```
⟦a := x⟧ = a := sx

⟦x := a⟧ = x := a; sx := a

⟦S1;S2⟧ = ⟦S1⟧ ; ⟦S2⟧

          ...
```

- ➡ ***Sound***: Behaviours(imp) ⊆ Behaviours(RPSI_spec)
- ➡ ***Complete***: Behaviours(RPSI_spec) ⊆ Behaviours(imp)

Conclusions

Conclusions

✓ ***Sound*** & ***complete*** reference implementation for ***PSI***

✓ *Declarative* ***RPSI*** semantics with ***mixed*** accesses

✓ ***Sound*** & ***complete*** reference implementation for ***RPSI***

Conclusions

✓ ***Sound*** & ***complete*** reference implementation for ***PSI***

✓ *Declarative* ***RPSI*** semantics with ***mixed*** accesses

✓ ***Sound*** & ***complete*** reference implementation for ***RPSI***

➡ PSI under ***other*** weak memory models

➡ ***Program Logics*** for STMs

Conclusions

✓ ***Sound*** & ***complete*** reference implementation for ***PSI***

✓ *Declarative* ***RPSI*** semantics with ***mixed*** accesses

✓ ***Sound*** & ***complete*** reference implementation for ***RPSI***

➡ PSI under ***other*** weak memory models

➡ ***Program Logics*** for STMs

Thank you for listening!