Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine

Michael Bond

Milind Kulkarni — Purdue

Ohio State

Man Cao

Meisam Fathi Salmi

Jipeng Huang — Microsoft

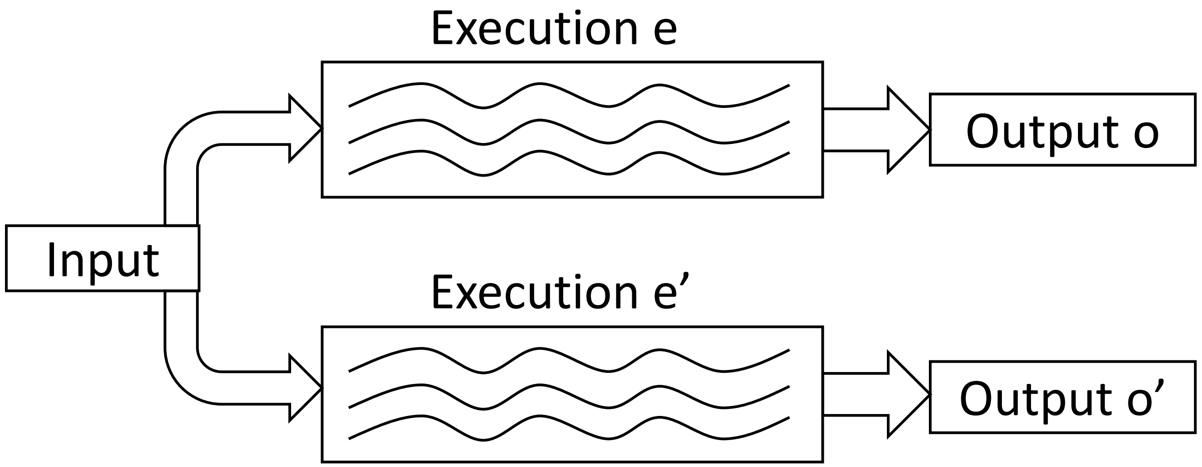
Nondeterminism is problematic

- Reproduce and Debug

- Replication for fault-tolerance

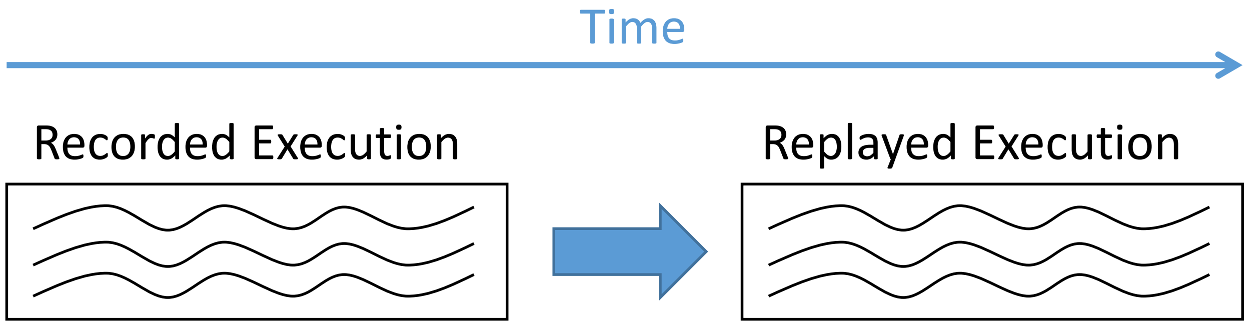
- Record and Replay!
  - RecPlay, M. Ronsse, et al, 1999
  - Respec, D. Lee, et al, 2010
  - DoublePlay, K. Veeraraghavan, et al, 2011
  - Chimera, D. Lee, et al, 2012
  - CLAP, J. Huang, et al, 2013

  And many others…



Record and Replay Types

- Offline
  - Debugging

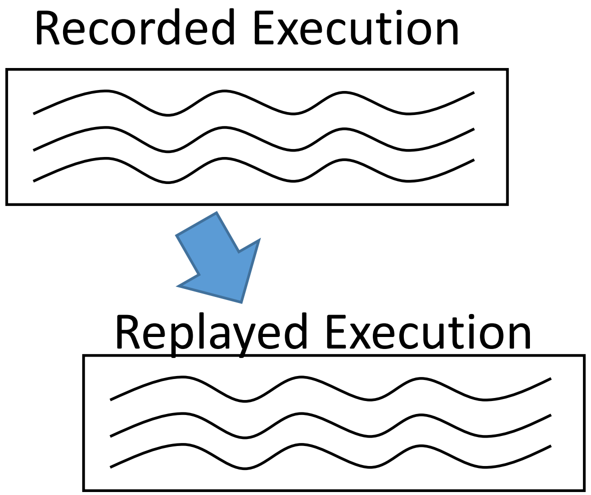


Time

Recorded Execution

Replayed Execution

- Online
  - Fault tolerance
  - Distribute dynamic analysis



Recorded Execution

Replayed Execution

Record and Replay Challenges

- Single-thread, *external* sources of nondeterminism
    - I/O, time, SysCall, etc.
    - Garbage collection, hash code
    - Adaptive compilation and dynamic classloading
        - Even harder for metacircular JVM (Jikes RVM)

- Multithreaded, *internal* nondeterminism
    - Thread interleaving
    - Hard and expensive to capture or control

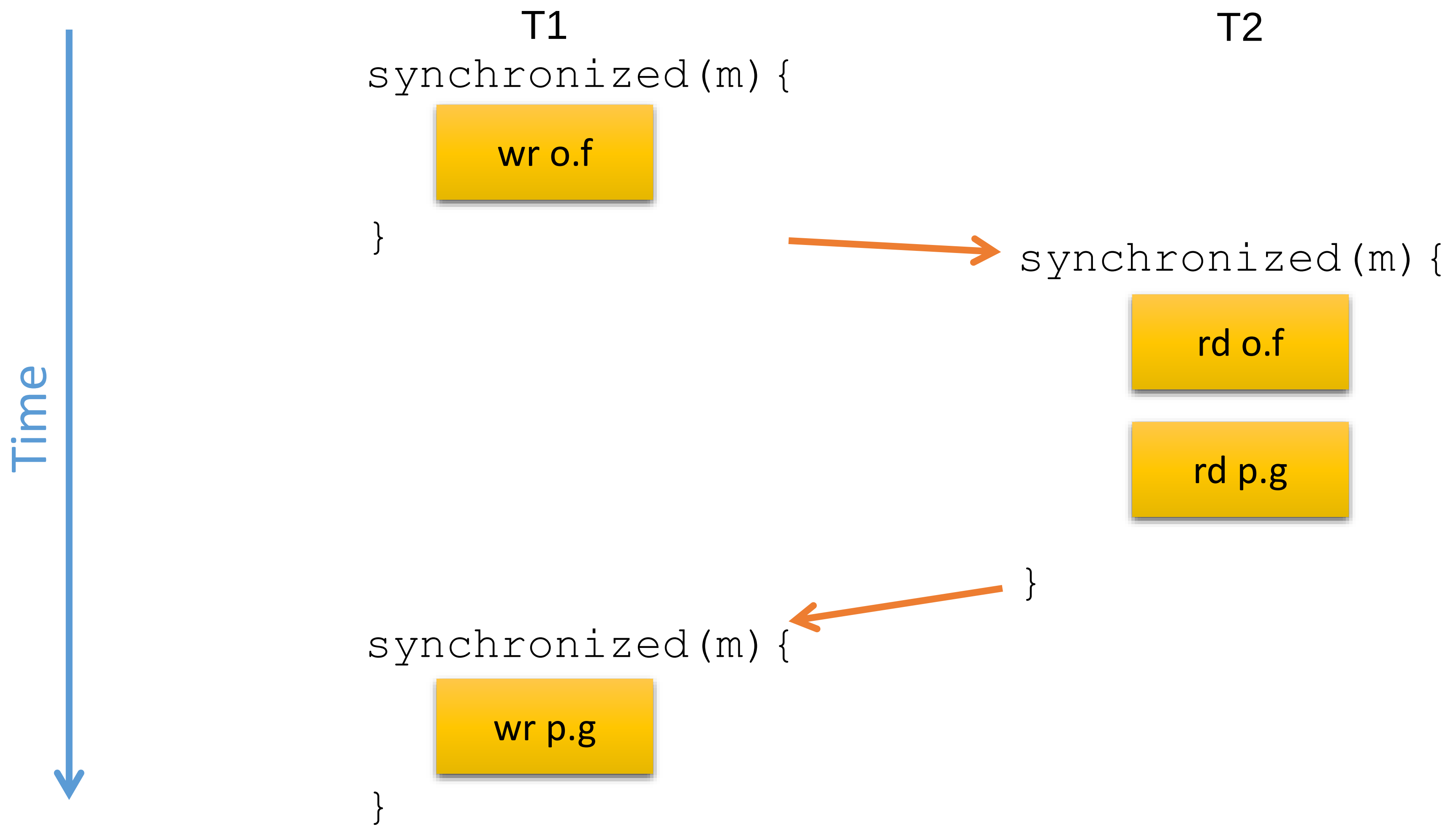
Contributions

Handle both nondeterminisms

- External (VM, system, I/O)
  - Non-trivial engineering effort
  - Novel methodology to sidestep nondeterminisms
    - Fork-and-recompile
- Internal (multithreading)
  - Two dynamic analyses
    - RECORD
    - REPLAY
  - Low overhead
  - Fewer limitations

Handling internal nondeterminism

Easy case: data-race-free execution



Time

T1

```
synchronized(m){
    wr o.f
}
```

T2

```
synchronized(m){
    rd o.f
    rd p.g
}
```

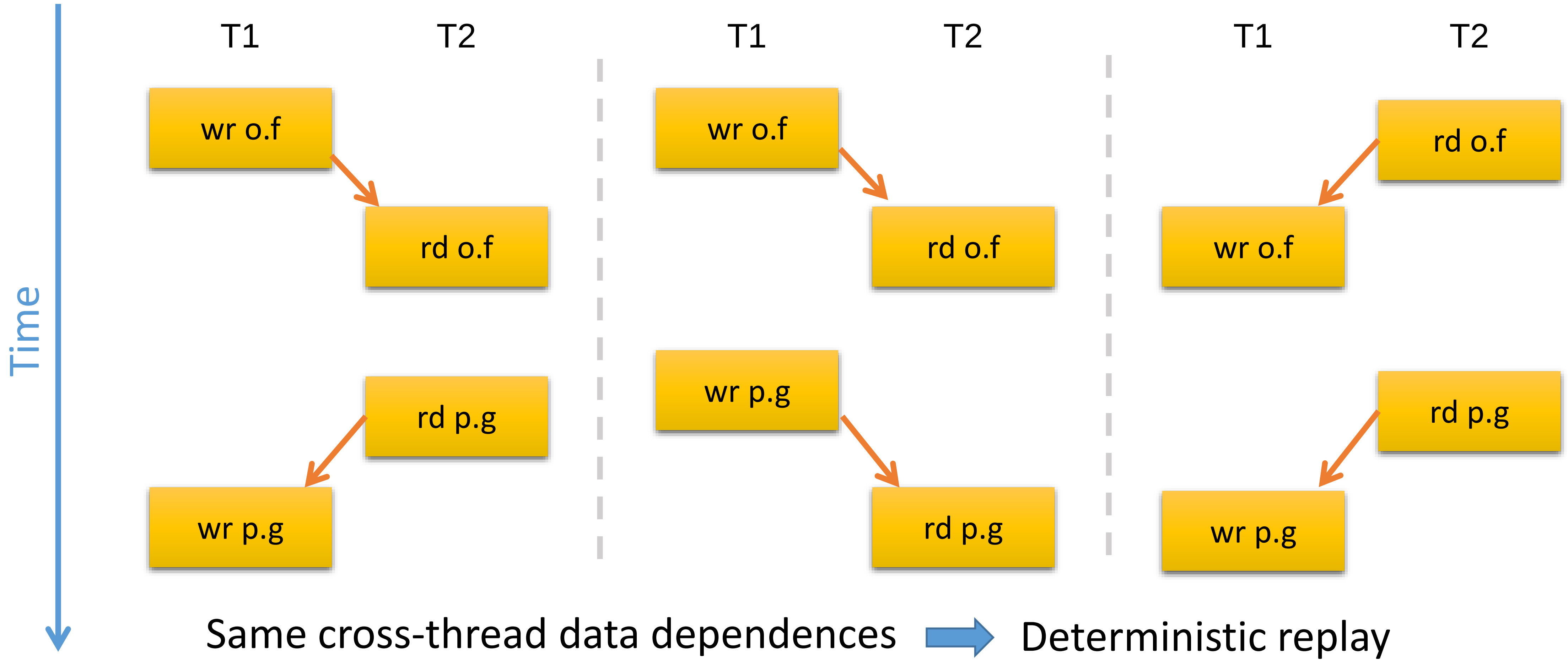
T1

```
synchronized(m){
    wr p.g
}
```

Hard case: executions with data races

- Unfortunately, most real-world programs have data races
  - Instrument all potentially racy memory accesses to capture *cross-thread data dependences*
  - Add many synchronization operations, very expensive!

Cross-thread data dependences



Same cross-thread data dependences ➡ Deterministic replay

Limitations of existing multithreaded record and replay approaches

- High overhead for recording cross-thread dependences
- OR do not handle racy execution
- OR support only offline or online replay, but not both
- OR rely on speculation and extra cores
- OR need custom hardware

Our approach overcomes all of these limitations at the same time!

RECORD

- Builds on our prior work “Octet”
- Octet tracks cross-thread dependences at object granularity

- Each object has an ownership state
    - Analogous to cache coherence protocol

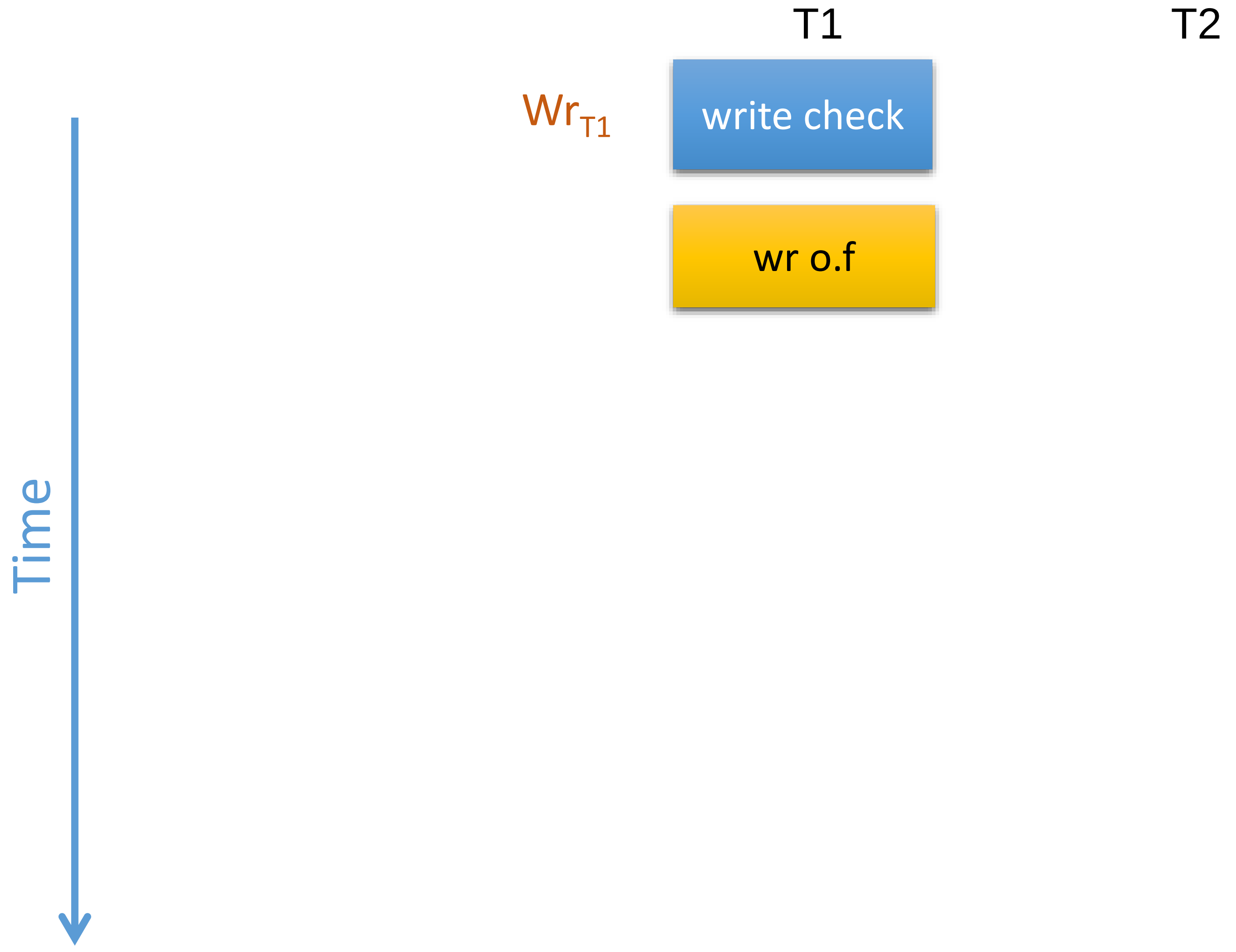
- For simplicity, consider

$$o.state \in \{ Wr_T , Rd_T \}$$

- Cross-thread dependence => state transition

State transition example

Initially o.state = Wr_{T1}, p.state = Rd_{T2}



T1

T2

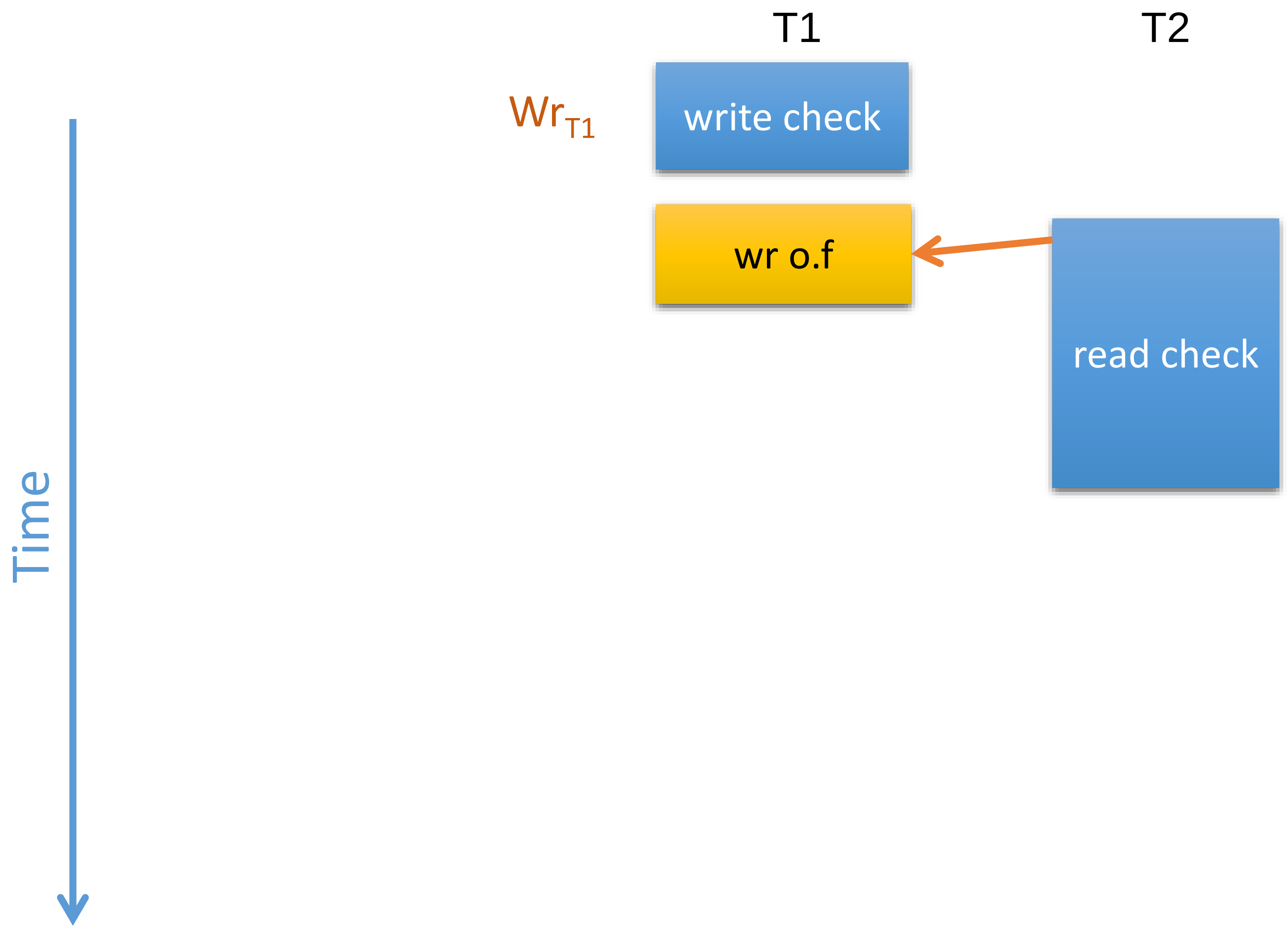
Wr_{T1}

write check

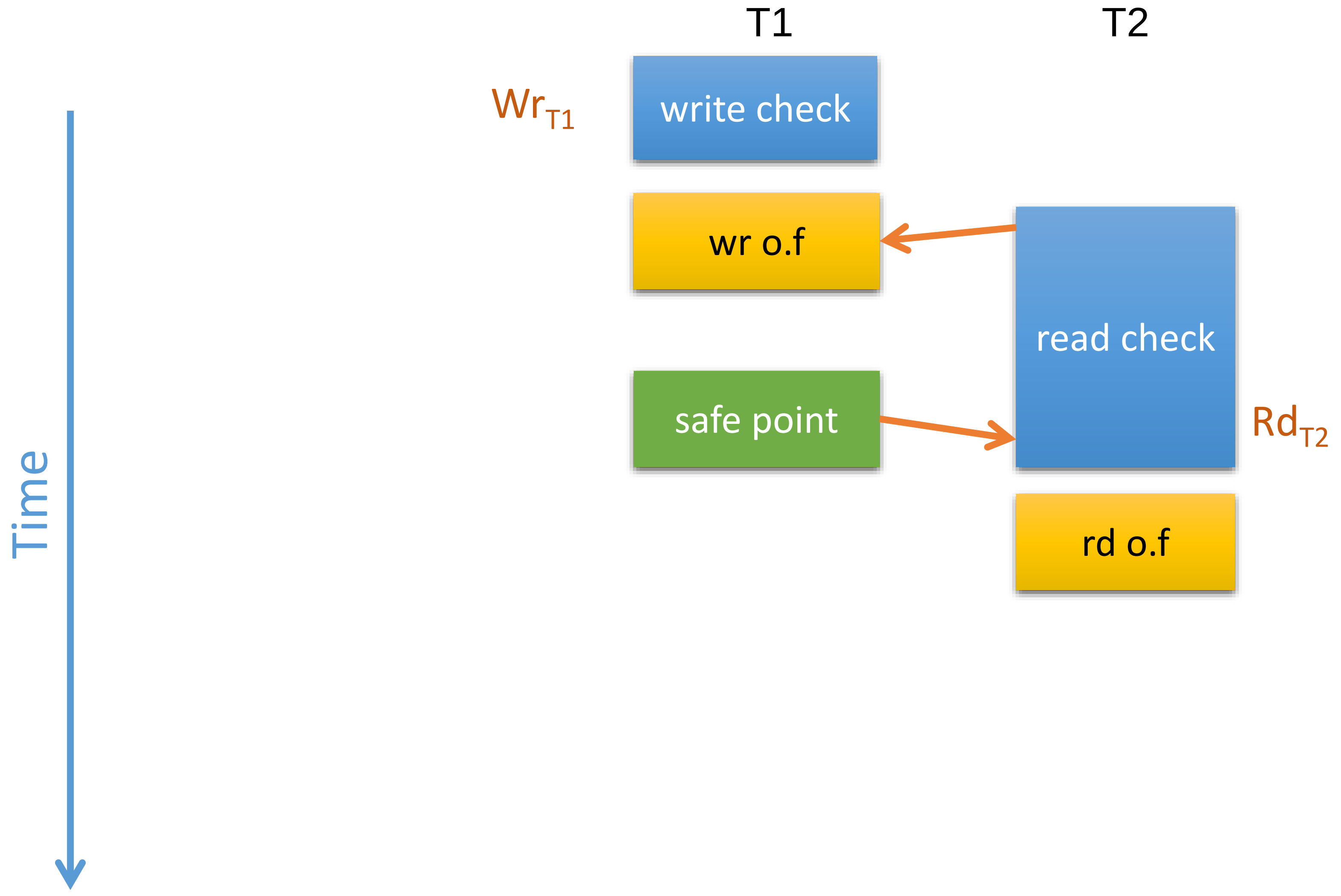
wr o.f

Time

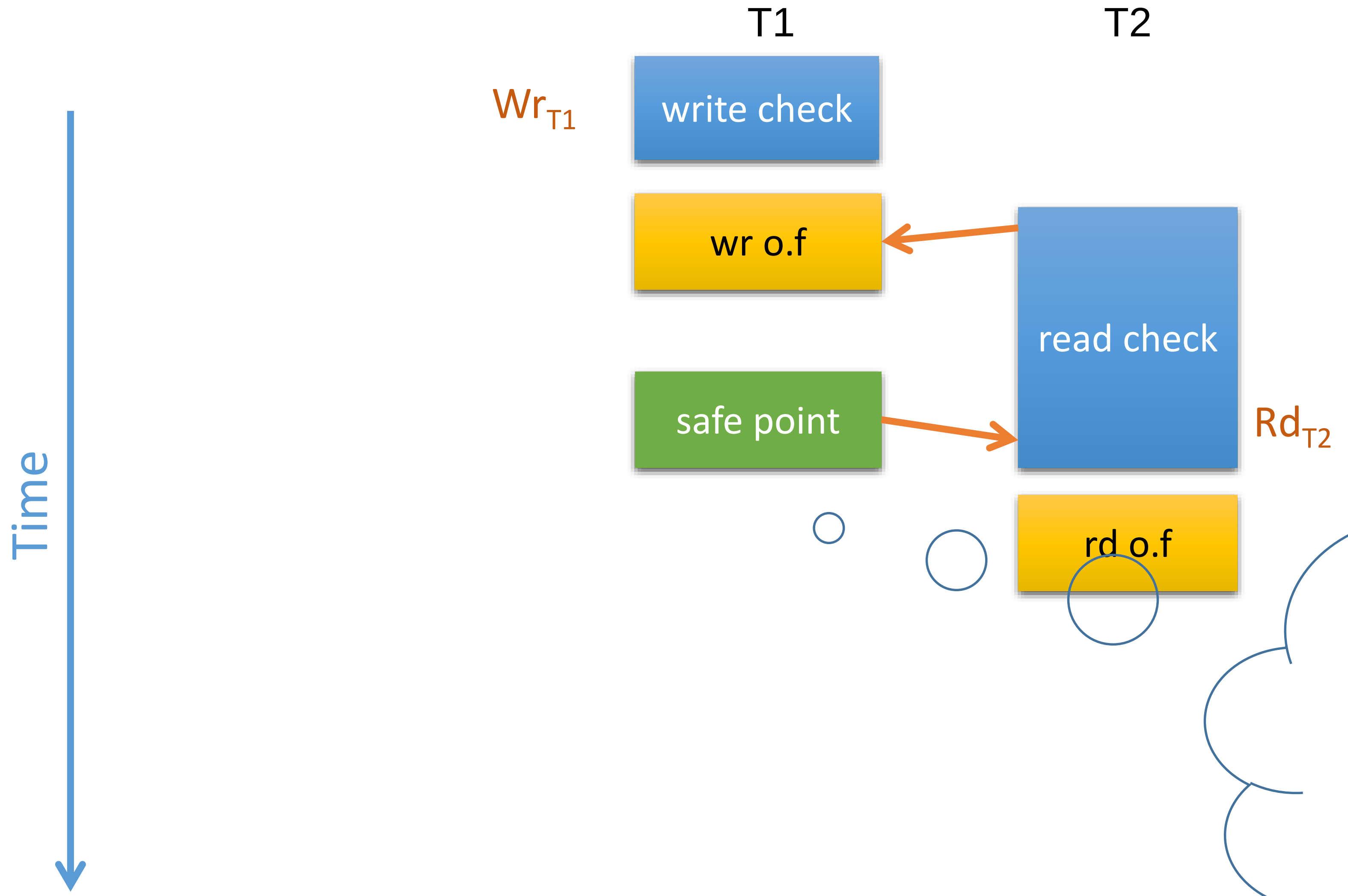
State transition example



State transition example

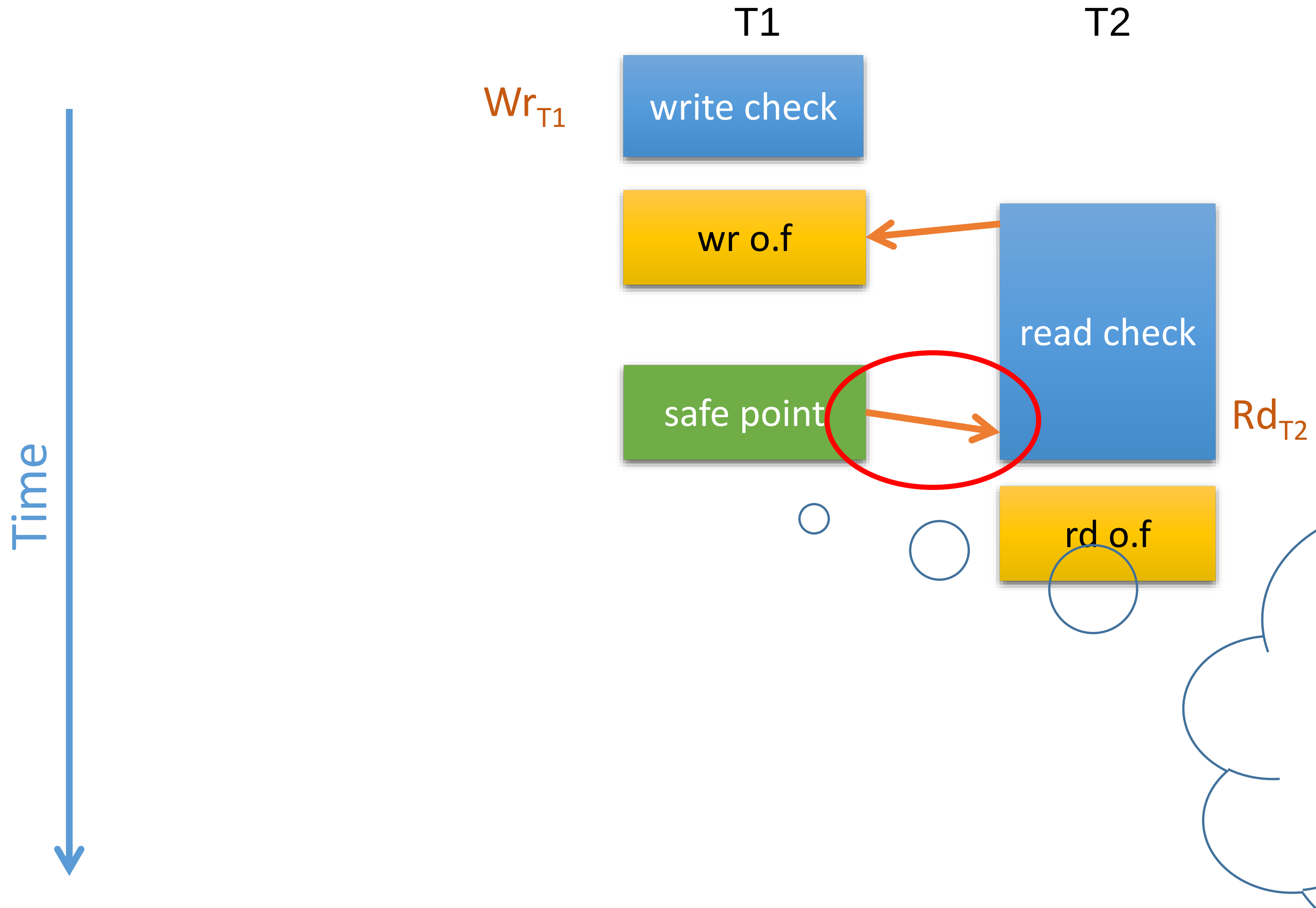


State transition example



How to record this cross-thread dependence?

State transition example



How to record this cross-thread dependence?

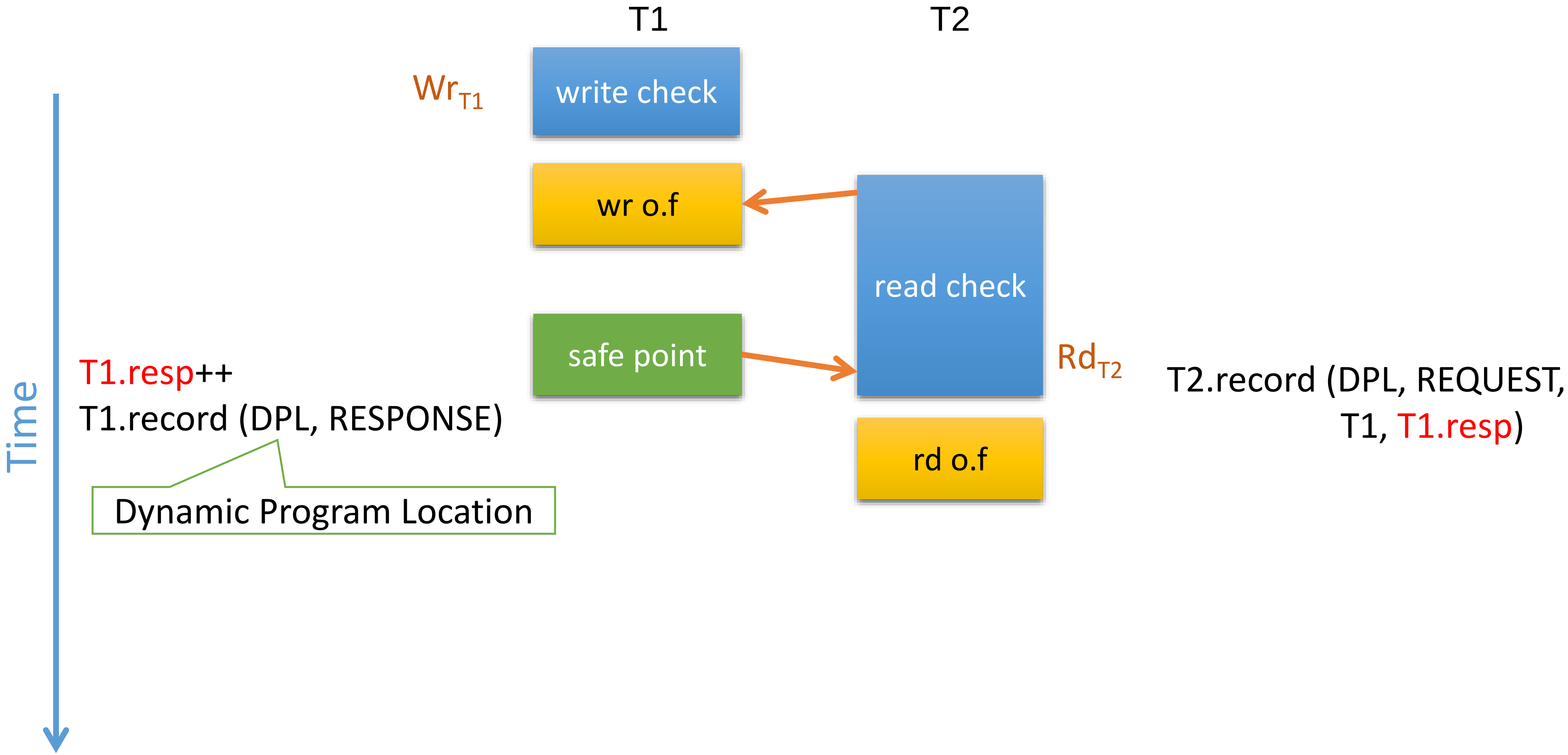
RECORD Design

- What?
  - Response edge
  - Per-thread response counter

- Where in execution?
  - Dynamic Program Location (DPL)

- Per-thread log file

RECORD example



T1

T2

Wr_{T1}

write check

wr o.f

read check

safe point

Rd_{T2}

rd o.f

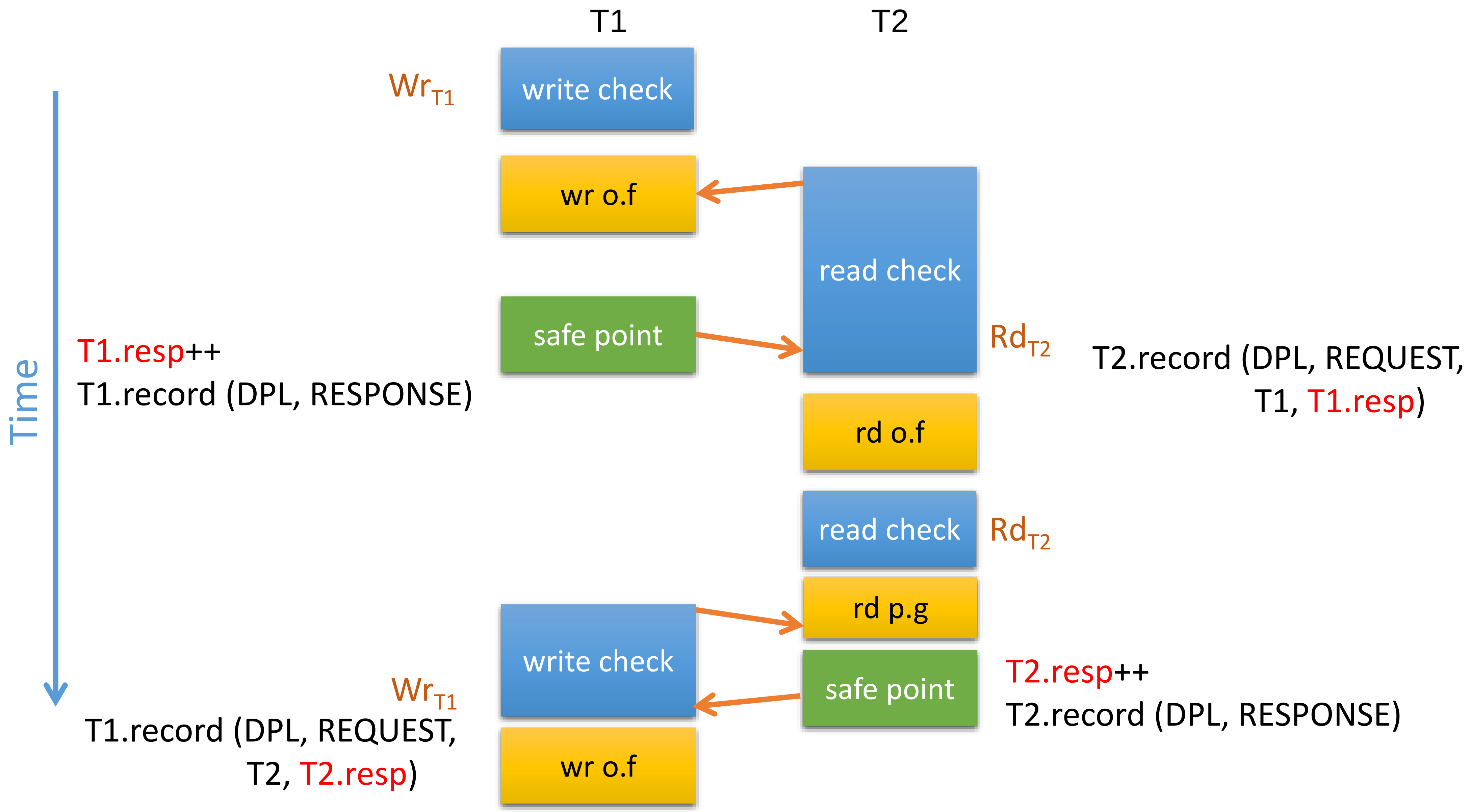
Time

T1.resp++
T1.record (DPL, RESPONSE)

Dynamic Program Location

T2.record (DPL, REQUEST,
T1, T1.resp)

RECORD example



T1

T2

Wr_{T1} write check

wr o.f

read check Rd_{T2}

safe point

T1.resp++
T1.record (DPL, RESPONSE)

T2.record (DPL, REQUEST,
T1, T1.resp)

rd o.f

read check Rd_{T2}

rd p.g

Wr_{T1} write check

safe point

T2.resp++
T2.record (DPL, RESPONSE)

T1.record (DPL, REQUEST,
T2, T2.resp)

wr o.f

Time

RECORD observation

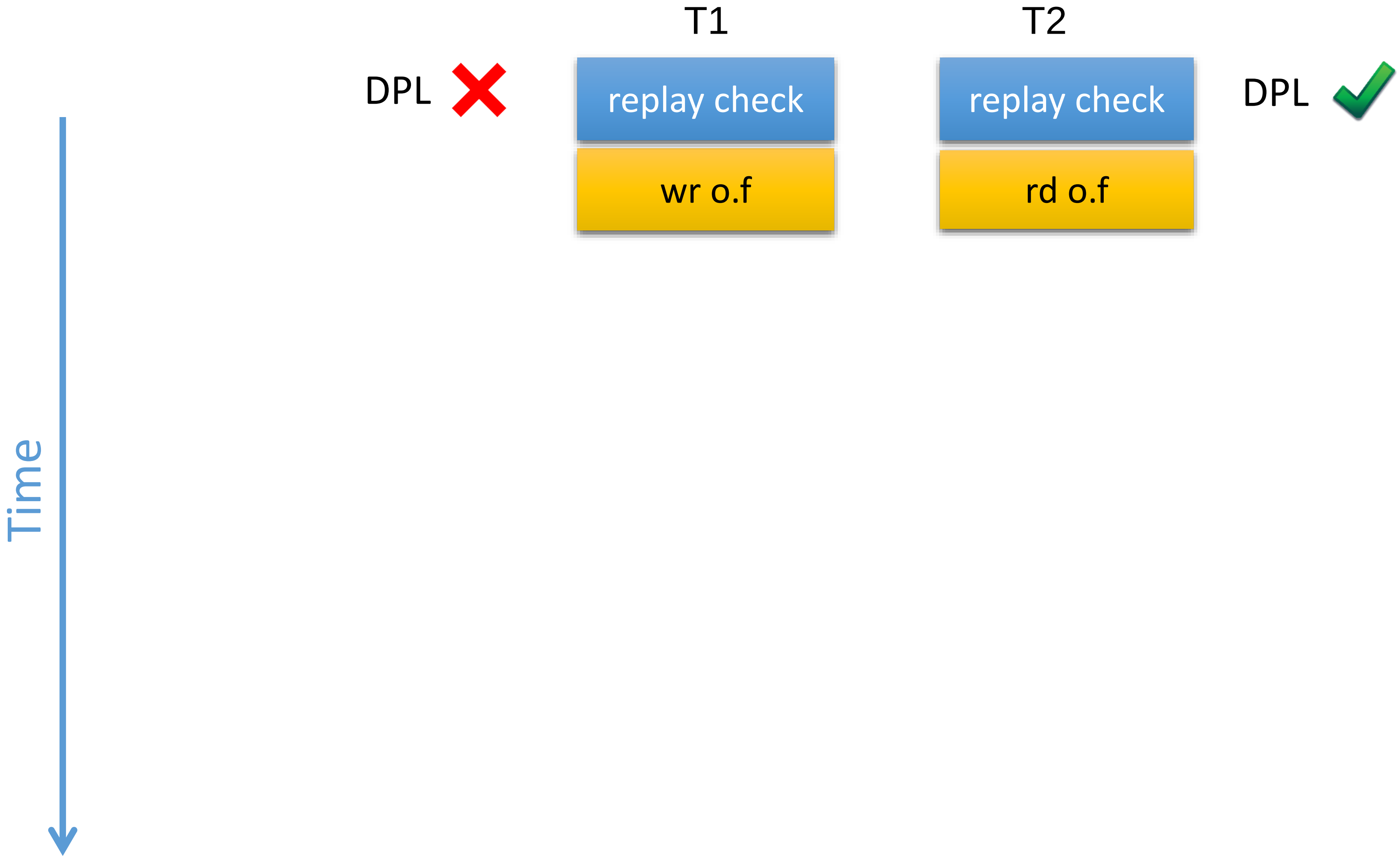
- Adds low overhead
    - Most accesses (>99%) are same-state
    - Only one load and one if check

REPLAY

- Goal: enforce recorded edges

- Instrument every possible edge source and sink
  - Check if DPL matches
- Edge source
  - Increment counter
- Edge sink
  - Wait for counter to reach recorded value

REPLAY example



T1

T2

DPL ✗

replay check

wr o.f

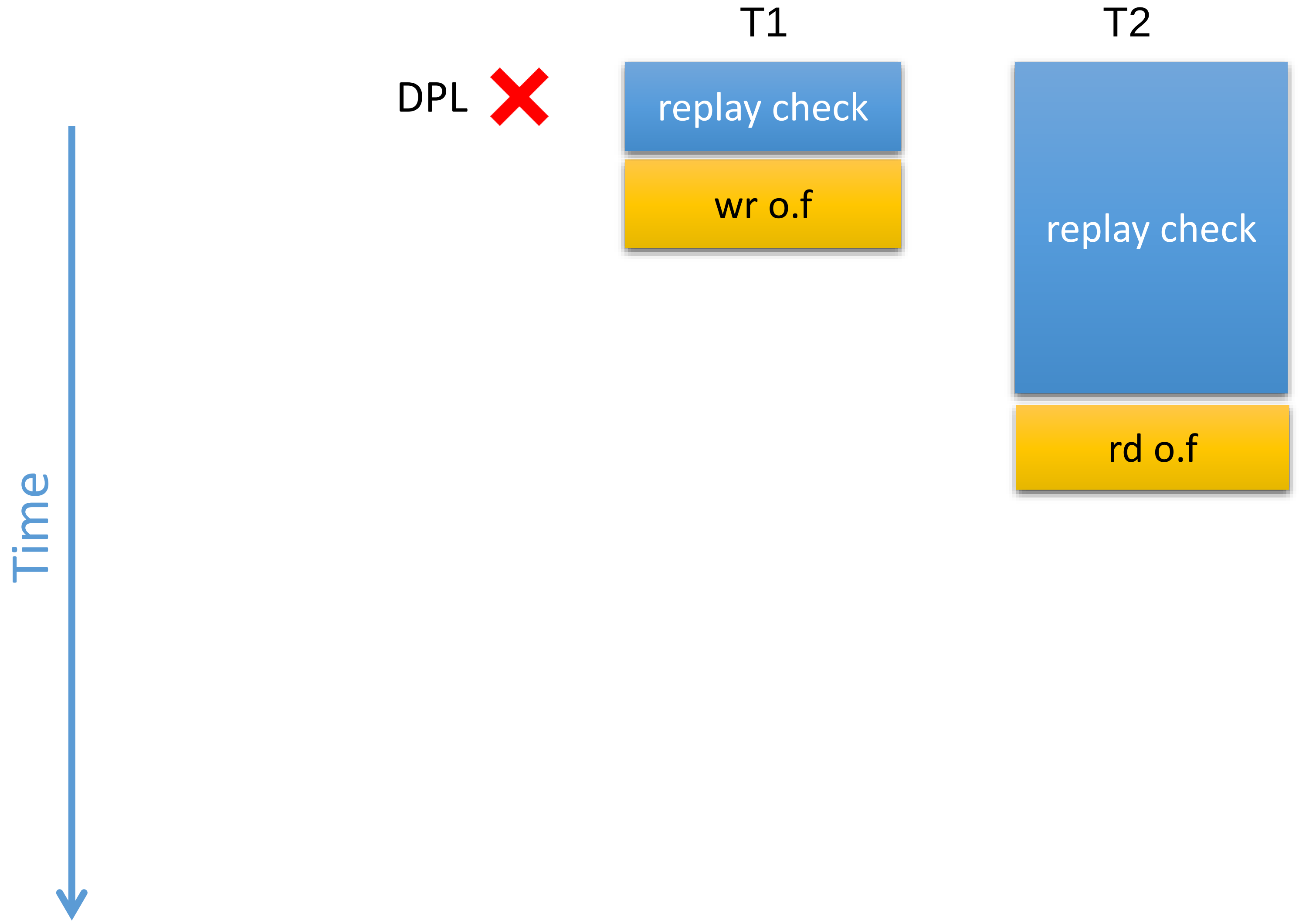
replay check

rd o.f

DPL ✓

Time

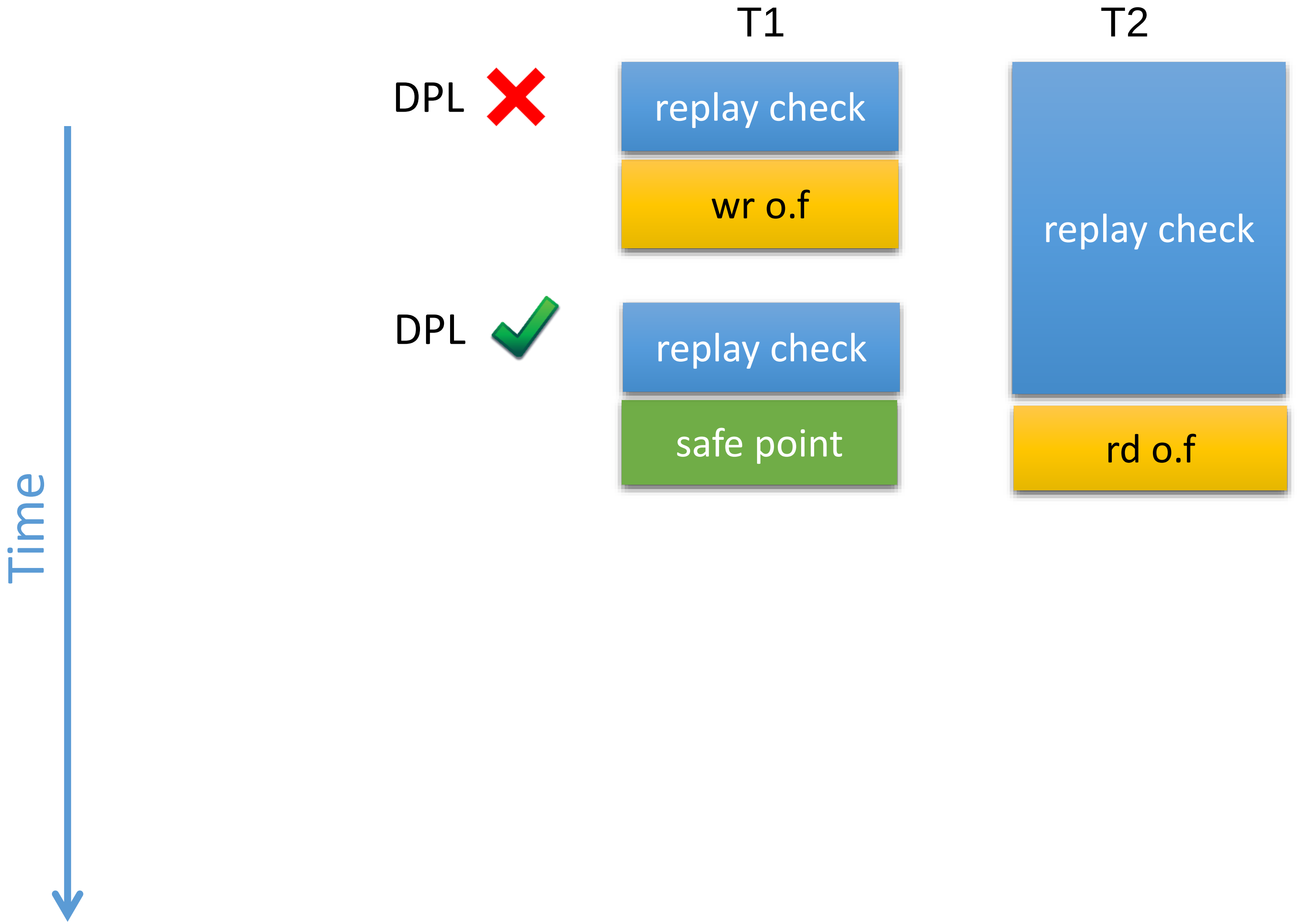
REPLAY example



DPL ✓

```
while(T1.resp < recorded_T1_resp) {
  mem_fence
}
mem_fence
```

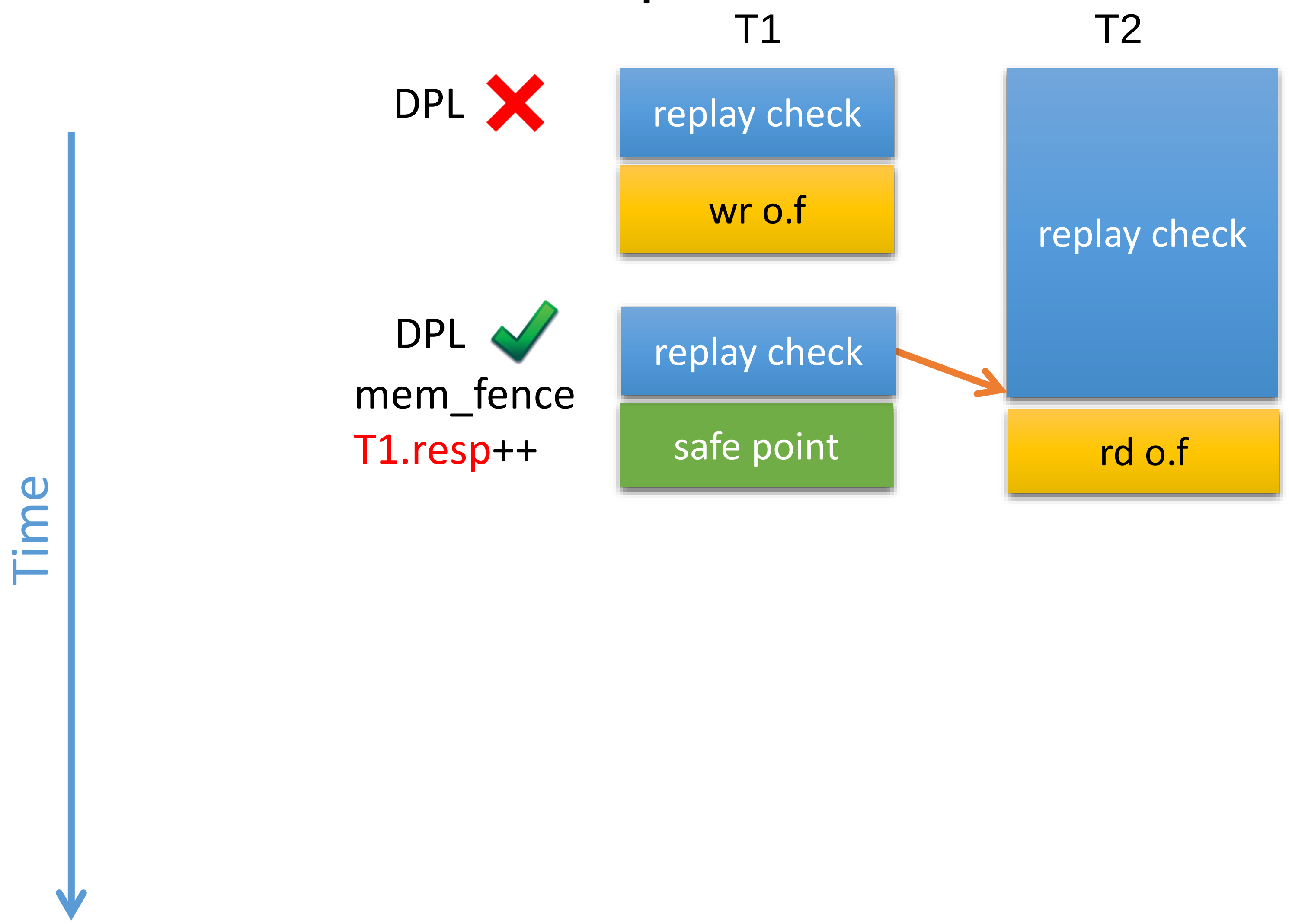
REPLAY example



DPL ✓

```
while(T1.resp < recorded_T1_resp) {
  mem_fence
}
mem_fence
```

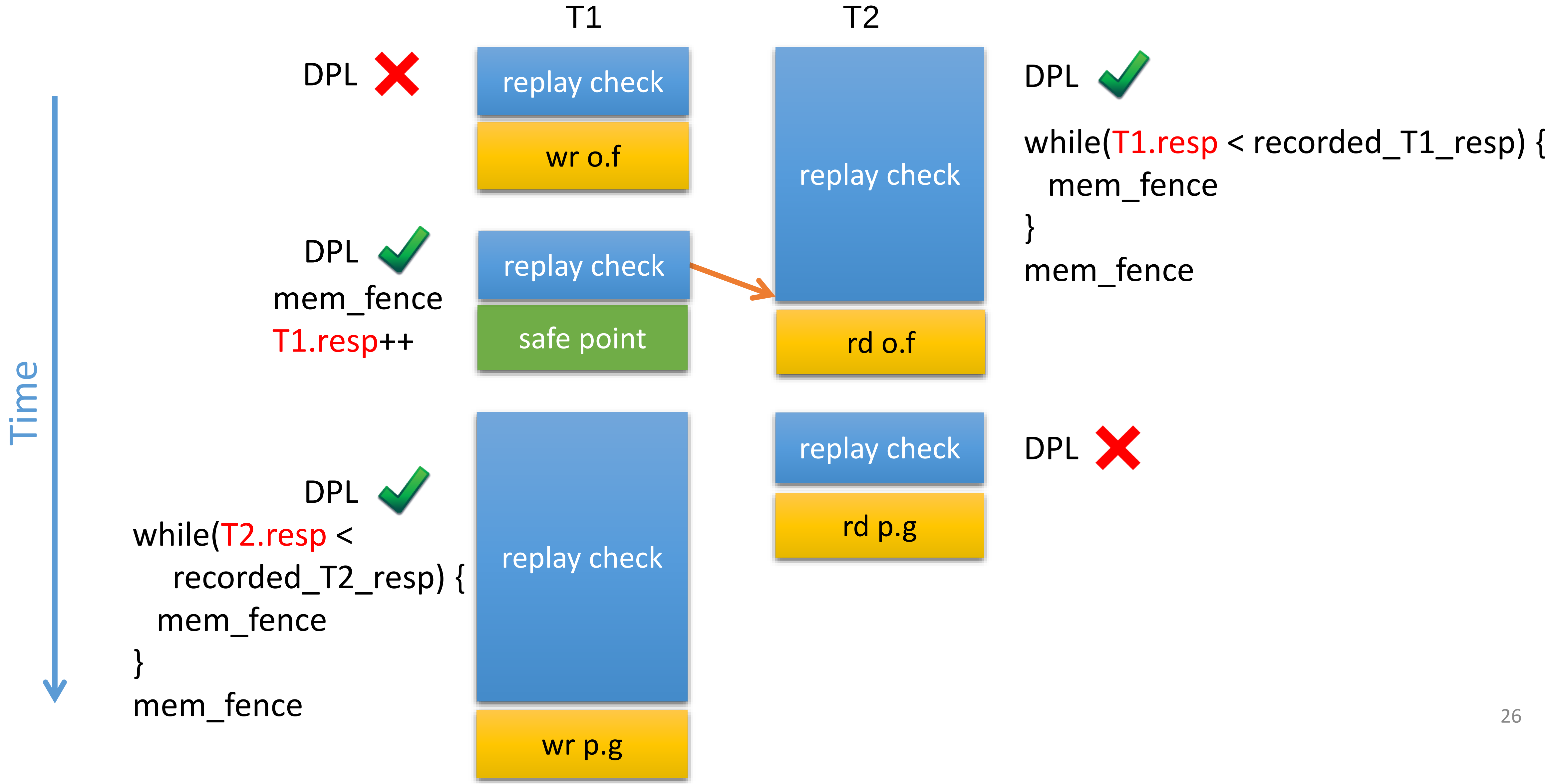
REPLAY example



DPL ✓

```
while(T1.resp < recorded_T1_resp) {
  mem_fence
}
mem_fence
```

REPLAY example



DPL ✗

DPL ✓

```
mem_fence
T1.resp++
```

DPL ✓

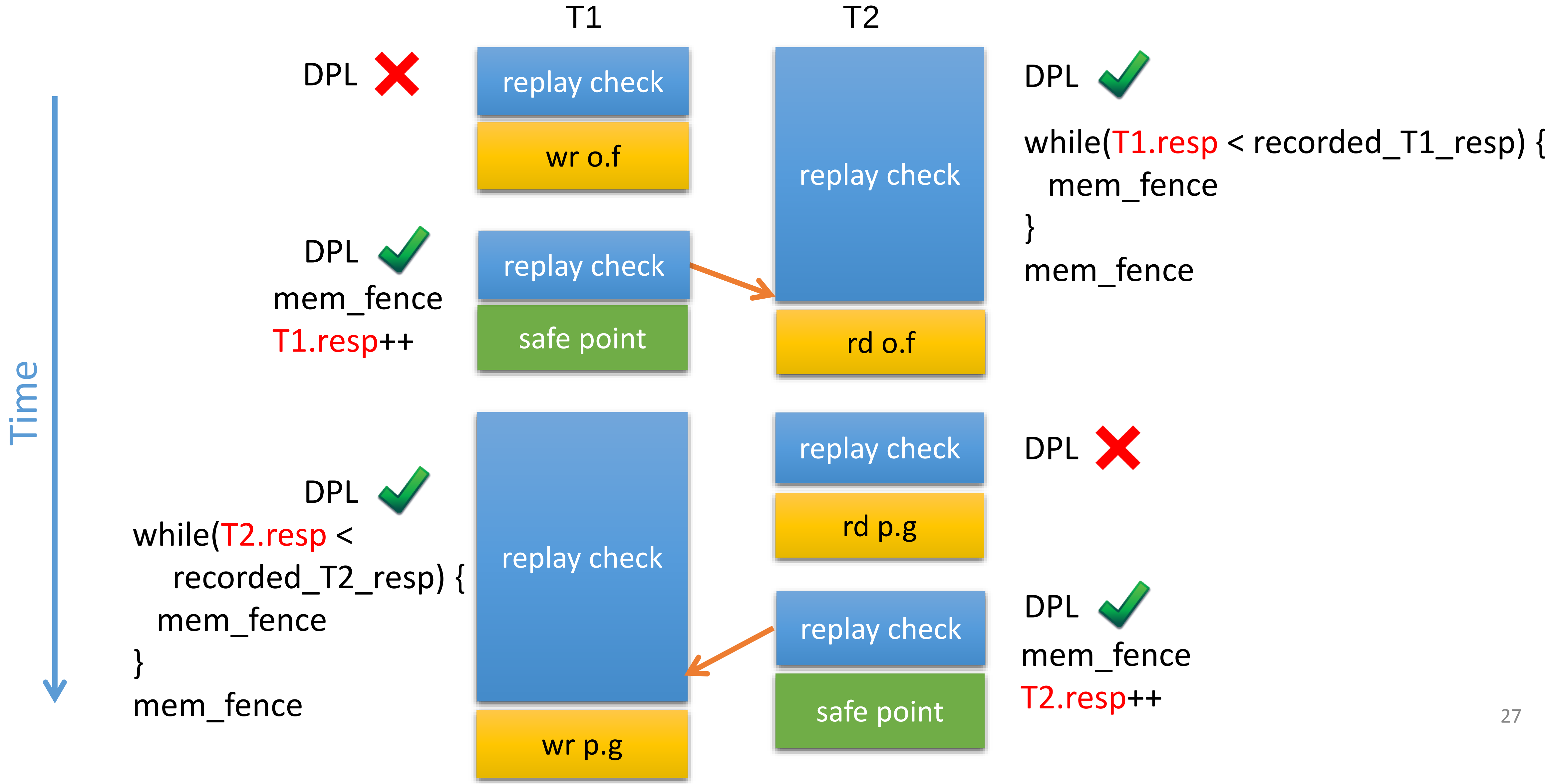
```
while(T2.resp <
    recorded_T2_resp) {
  mem_fence
}
mem_fence
```

DPL ✓

```
while(T1.resp < recorded_T1_resp) {
  mem_fence
}
mem_fence
```

DPL ✗

REPLAY example



Time

T1

T2

DPL ✗

replay check

wr o.f

replay check

DPL ✓

```
while(T1.resp < recorded_T1_resp) {
  mem_fence
}
mem_fence
```

DPL ✓

mem_fence

T1.resp++

replay check

safe point

rd o.f

replay check

DPL ✗

rd p.g

DPL ✓

```
while(T2.resp <
    recorded_T2_resp) {
  mem_fence
}
mem_fence
```

replay check

replay check

safe point

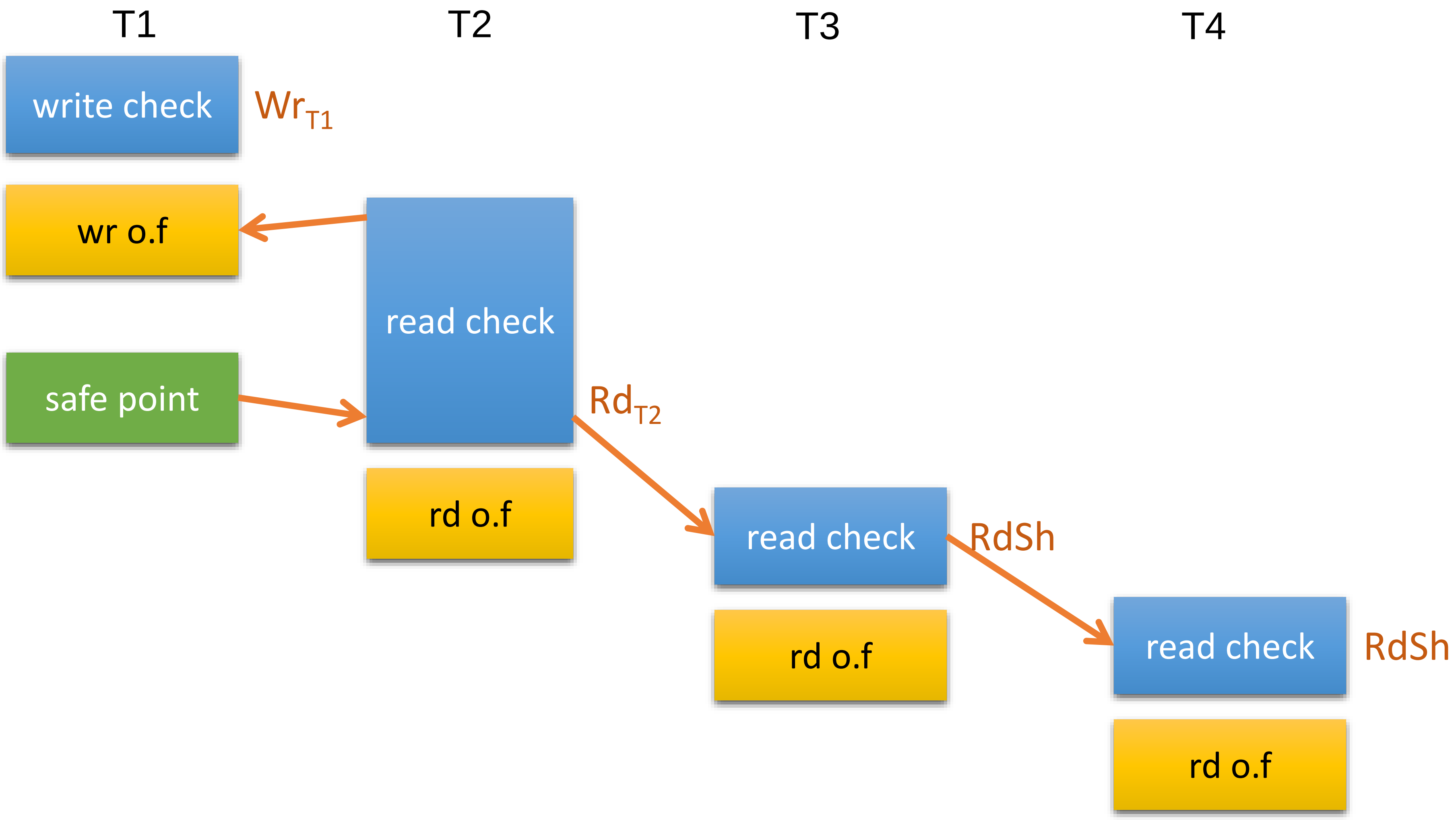
DPL ✓

mem_fence

T2.resp++

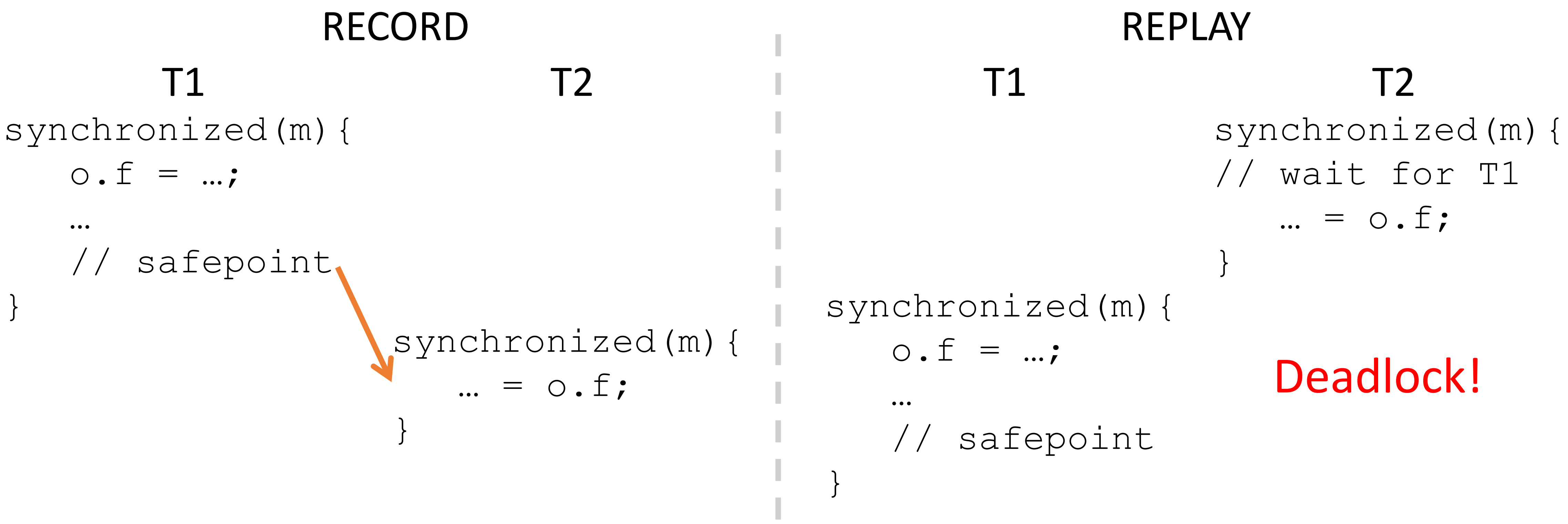
wr p.g

A more complicated case



REPLAY observations

- Do not track object’s state
  - Only per-thread or global counters
- How about program synchronization?



RECORD

```
        T1                              T2
synchronized(m){
   o.f = …;
   …
   // safepoint
}
                                   synchronized(m){
                                      … = o.f;
                                   }
```

REPLAY

```
        T1                              T2
                                   synchronized(m){
                                   // wait for T1
                                      … = o.f;
                                   }
synchronized(m){
   o.f = …;
   …
   // safepoint
}
```

Deadlock!

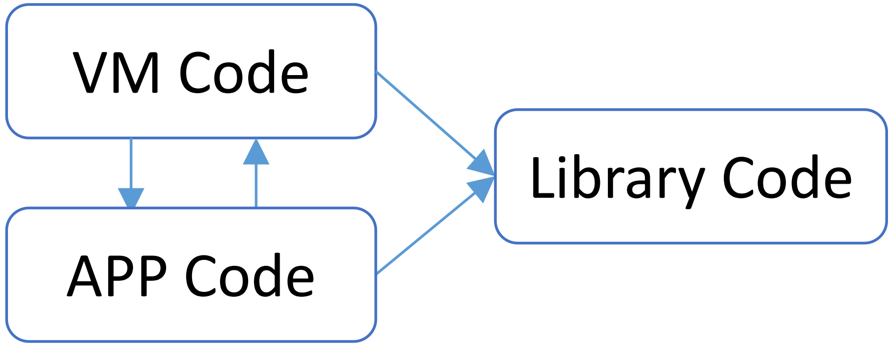
Elide program synchronization

- Necessary
  - RECORD does not track synchronization
  - Otherwise deadlock
- Side effect: more parallelism (see Evaluation)

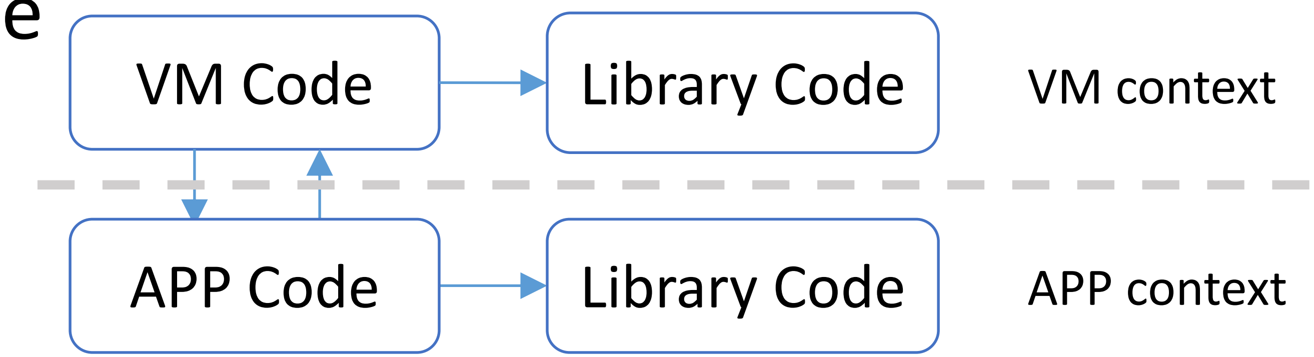
Handling external nondeterminism

Goal: Application-level determinism

- No need to track JVM's cross-thread dependences!
  - Jikes RVM



- Contexts for compiled code

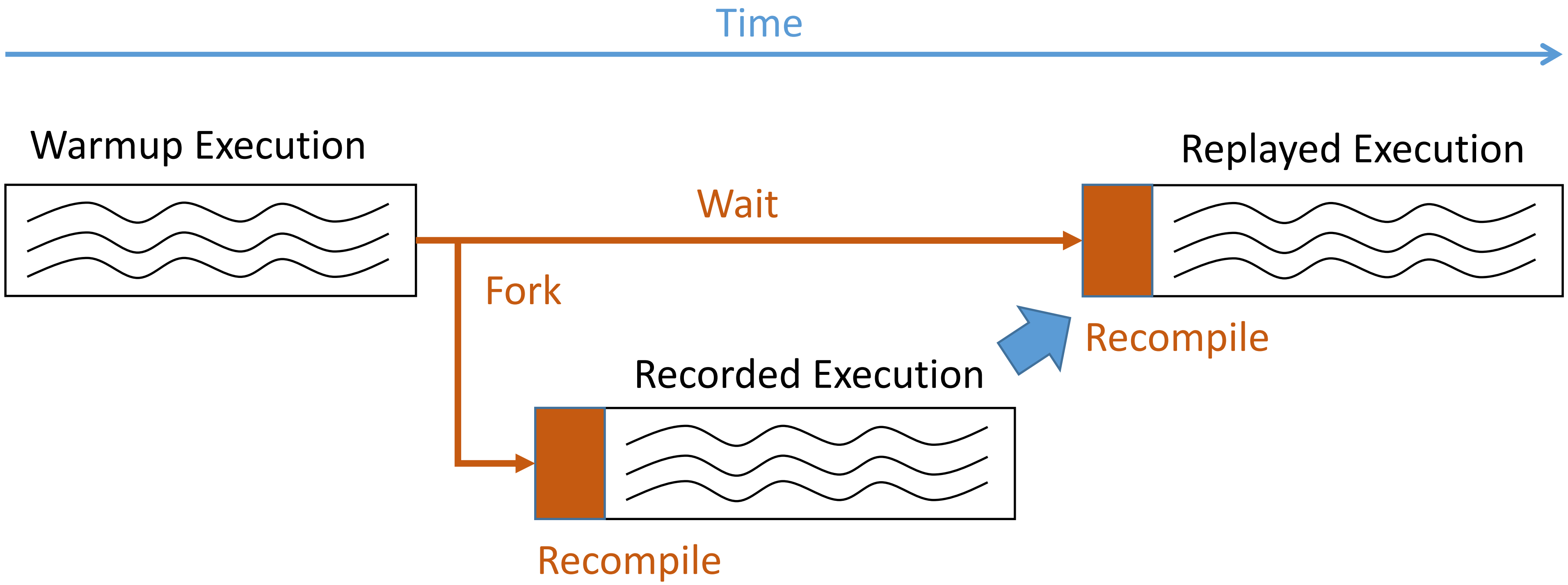


External nondeterminisms

- Handled nondeterminisms
  - Stop-the-world GC, record and replay DPLs of GC points
  - Deterministic hash code
  - Deterministic “logical time”
  - Deterministic I/O
  - etc.

- Adaptive compilation and dynamic classloading
  - Most challenging (esp. in Jikes)!
  - Fork-and-recompile

Fork-and-recompile



Evaluation

- Implementation in Jikes RVM
  - Publicly available (http://sourceforge.net/p/jikesrvm/research-archive/49/)
- DaCapo 2006 & 2009, SPEC JBB 2000 & 2005
- 64 cores (AMD Opteron 6272)

REPLAY Efficacy

Benchmark	REPLAY Success Rate
	Default w/ value logging
hsqldb6	100%
lusearch6	100%
xalan6	100%
avrora9	100%
jython9	100%
luindex9	100%
lusearch9	100%
pmd9	100%
sunflow9	100%
xalan9	60%
pjbb2000	100%
pjbb2005	100%

REPLAY Efficacy

Benchmark	REPLAY Success Rate	
	Default w/ value logging	Ignore HB edge w/ value logging
hsqldb6	100%	0%
lusearch6	100%	0%
xalan6	100%	0%
avrora9	100%	0%
jython9	100%	0%
luindex9	100%	0%
lusearch9	100%	0%
pmd9	100%	0%
sunflow9	100%	0%
xalan9	60%	0%
pjbb2000	100%	0%
pjbb2005	100%	0%

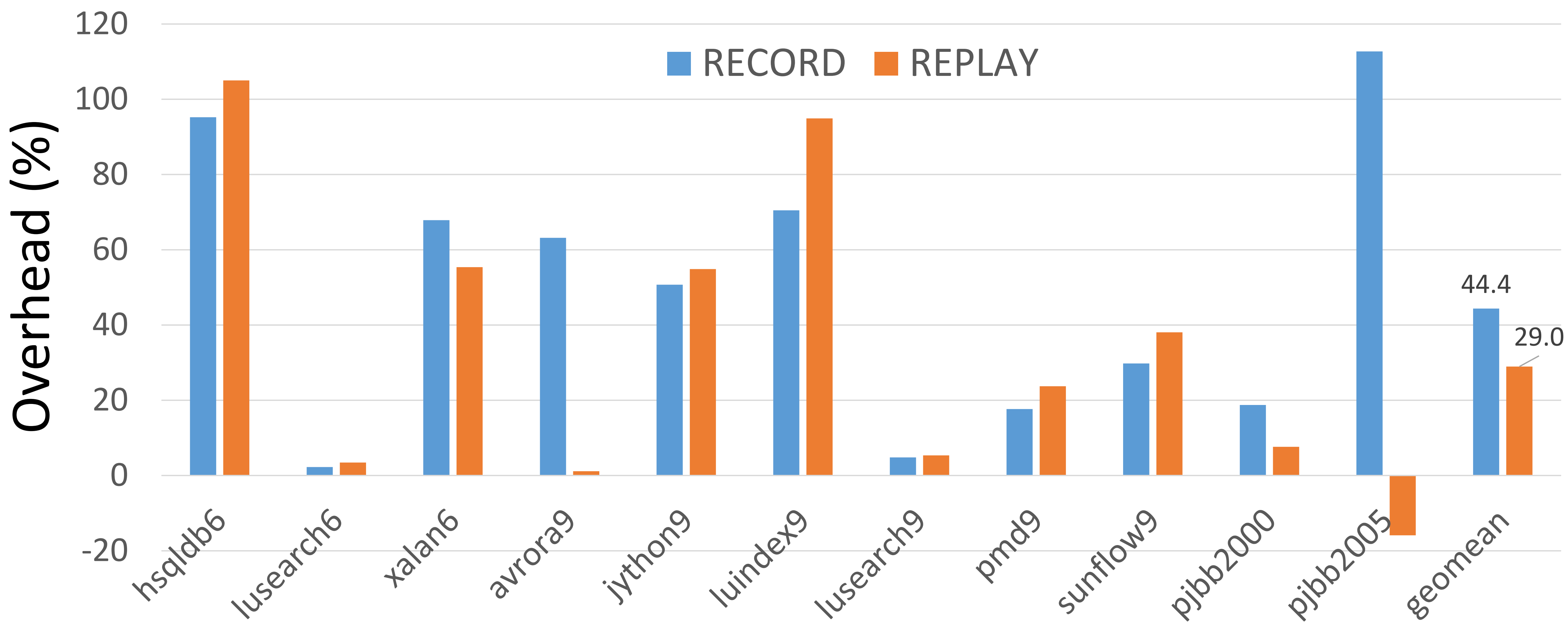
RECORD logging throughput

Benchmark	RECORD Logging MB/s
hsqldb6	0.7
lusearch6	<0.1
xalan6	7.7
avrora9	2.5
jython9	<0.1
luindex9	<0.1
lusearch9	<0.1
pmd9	0.1
sunflow9	0.1
xalan9	9.7
pjbb2000	1.1
pjbb2005	4.8

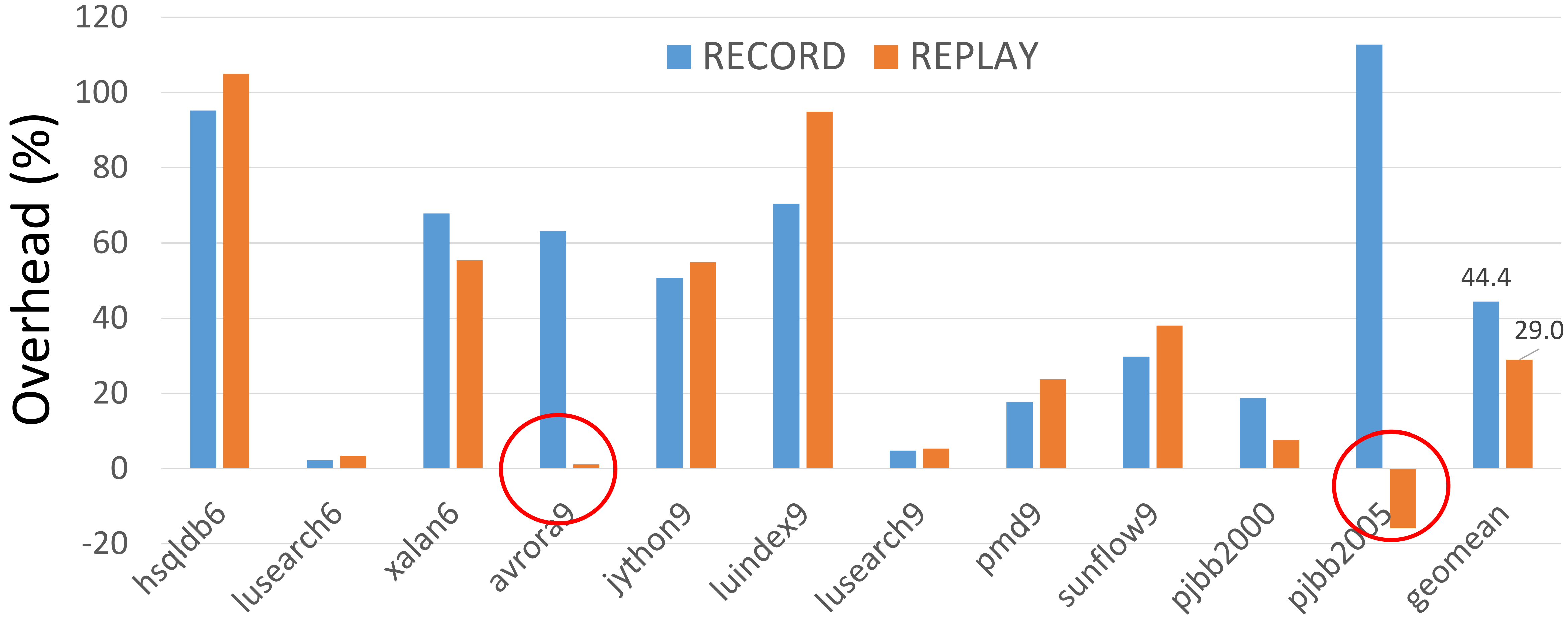
RECORD logging throughput

Benchmark	RECORD Logging MB/s
hsqldb6	0.7
lusearch6	<0.1
xalan6	7.7
avrora9	2.5
jython9	<0.1
luindex9	<0.1
lusearch9	<0.1
pmd9	0.1
sunflow9	0.1
xalan9	9.7
pjbb2000	1.1
pjbb2005	4.8

Performance



Performance



Conclusion

- Handle both external and internal nondeterminisms
    - Metacircular JVM
    - fork-and-recompile

- Efficient record and replay
    - Low overhead in RECORD
    - More parallelism in REPLAY

- Overcome many limitations simultaneously
    - Online/offline, software-only, no speculation, etc.