Correctness of concurrent objects on weak memory models

Graeme Smith

The University of Queensland

With various collaborators:

John Derrick, Sheffield, UK
Brijesh Dongol, Brunel, UK
Lindsay Groves, VUW, NZ
Kirsten Winter, UQ
Rob Colvin, UQ

Concurrent objects

- Data structures, etc., designed to be accessed simultaneously by multiple threads
- Synchronisation — coarse-grained, fine-grained, non-blocking

Goal: prove correct wrt abstract specification where all operations are atomic.

Example: reader-writer object for a pair of variables $x1$ and $x2$

```
int x1:=0,x2:=0;

write(int d1,d2){ x1,x2:=d1,d2;}

read(){ return(x1,x2);}
```

Concurrent objects

Example: reader-writer object for a pair of variables $x1$ and $x2$

- can be implemented using a Linux seqlock

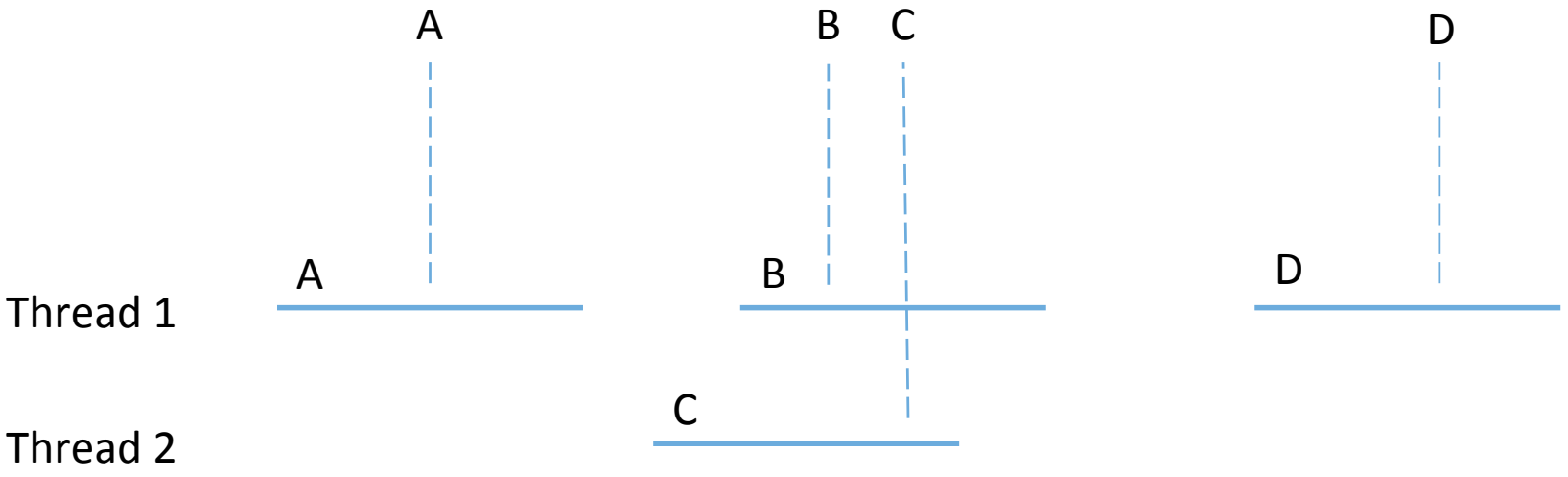
```
int x1:=0,x2:=0,c:=0;

write(int d1,d2){
   acquire;
   c++;
   x1:=d1;
   x2:=d2;
   c++;
   release;
}
```

```
read(){
   int c0,d1,d2;
   do{
     do{
       c0:=c
     }while(c0%2!=0)
     d1:=x1;
     d2:=x2;
   }while(c!=c0)
   return(d1,d2);
}
```

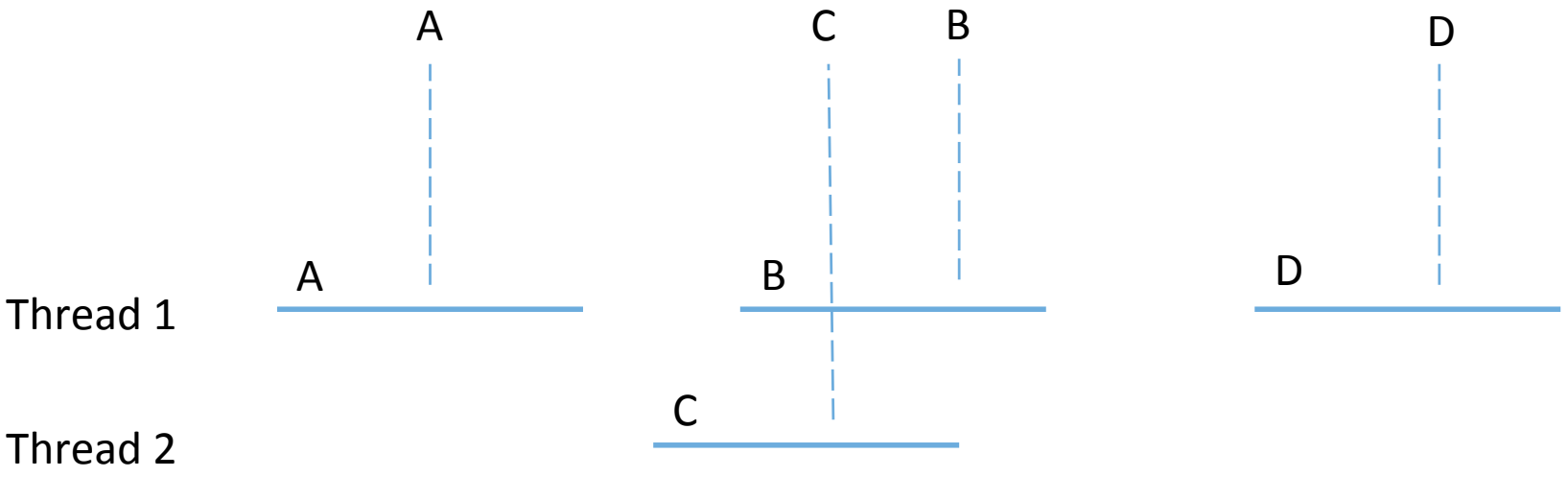
Linearizability

- Standard correctness criterion for concurrent objects
- Proposed by Herlihy and Wing (1989)
- Compositional
- Relation between sequential specification histories and concurrent implementation histories



Linearizability

- Standard correctness criterion for concurrent objects
- Proposed by Herlihy and Wing (1989)
- Compositional
- Relation between sequential specification histories and concurrent implementation histories



Linearizability

- A history is a finite sequence of invocations and responses, e.g.,

$$h = \langle\, \mathit{inv}((p, \texttt{write}), (1,2)),\ \mathit{inv}((q, \texttt{read}), \bot),\ \mathit{resp}((p, \texttt{write}), \bot),\ \mathit{resp}((q, \texttt{read}), (1,2)) \,\rangle$$

- Assuming operations are distinct, we define $<_h$ such that

$$op_1 <_h op_2 \mathrel{\widehat{=}} \exists\, m, n, in, out \bullet m < n \wedge h(m) = \mathit{resp}(op_1, out) \wedge h(n) = \mathit{inv}(op_2, in)$$

- A history is completed when, for each invocation without a response, we
  - add a response at the end of the history
  - remove the invocation
- A object C is linearizable wrt a specification A iff for each history h of C there is a history hs of A such that
  - hs is a permutation of a completion of h
  - $<_h \subseteq <_{hs}$

Linearizability vs trace refinement

- Trace refinement
  - Back (1990)
  - Abadi and Lamport (1991)
- Linearizability is weaker than trace refinement
  - Only partial correctness, e.g., `read` and `write` could be implemented as `while(true){}`
  - Only finite histories considered, e.g., *C* which outputs *true* an infinite number of times, linearizes with *A* which outputs *true* any finite number of times
  - Overlapping operations can be reordered

```
int a:=0; // shared variable

op1() {
   if (a=0) return 0;
   return 2;
}
```

```
op2() {
   a:=1;
   return 1;
}
```

Linearizability vs trace refinement

- Linearizability = observational refinement
  - Filipović, O'Hearn, Rinetzsky and Yang (2010)
  - no program calling an object can distinguish it's behaviour from the specification

The program

```
x:=op1() || y:=op2()
```

is equivalent to

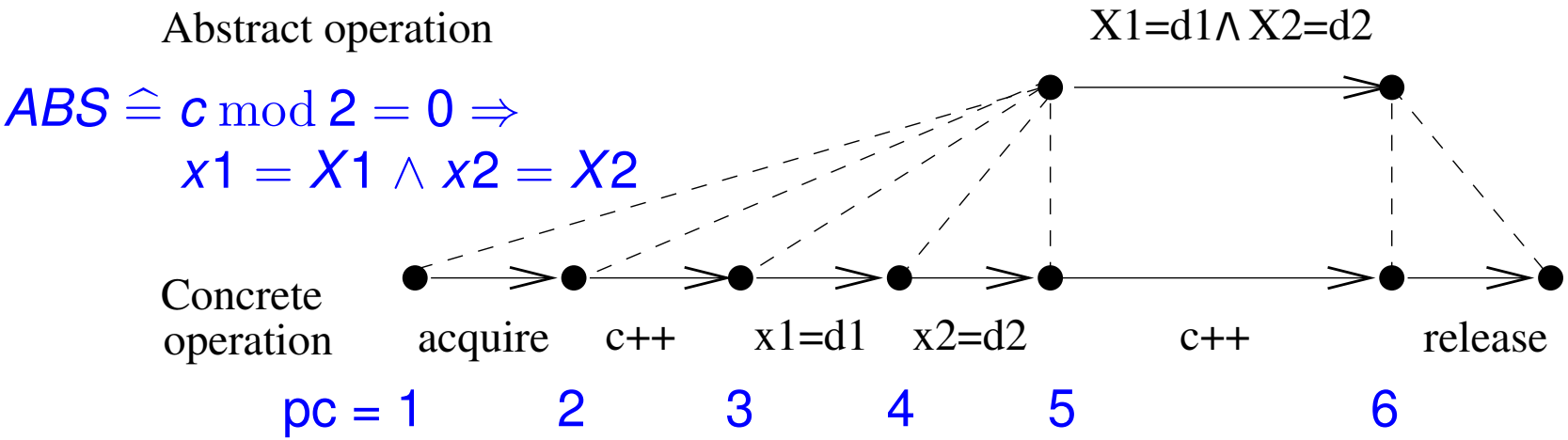
```
r1:=op1(); x:=r1 || r2:=op2(); y:=r2
```

whose behaviour `r2:=op2(); r1:=op1(); x:=r1; y:=r2` will result in observation of $x = 2$ before $y = 1$.

```
int a:=0; // shared variable
op1() {
   if (a=0) return 0;
   return 2;
}
```

```
op2() {
   a:=1;
   return 1;
}
```

Simulation-based proof method

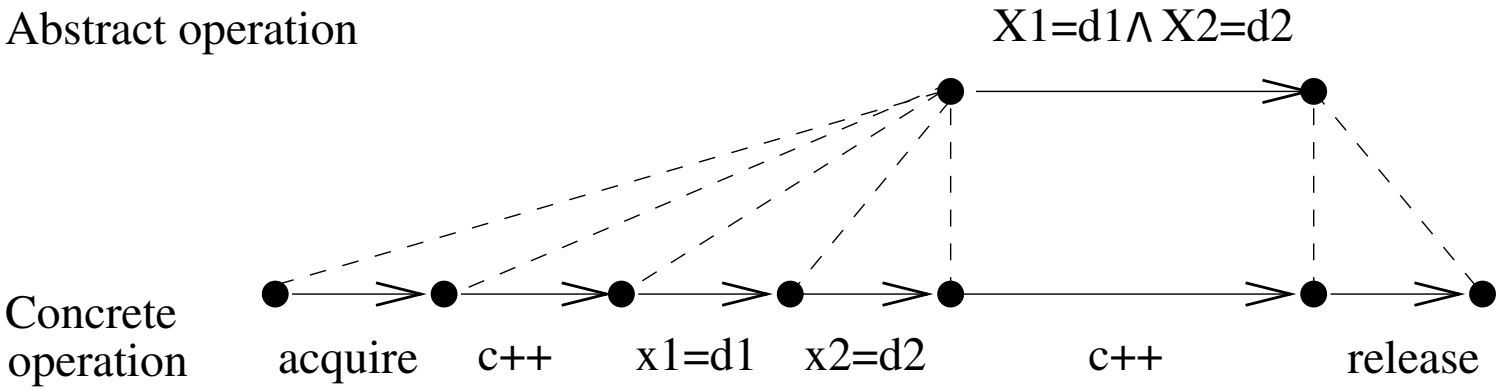


- thread-local — proofs consider 1 thread in isolation
- step-local — proofs consider 1 program step in isolation

$$INV \;\widehat{=}\; (pc = 5 \Rightarrow c \bmod 2 \neq 0 \wedge x1 = d1 \wedge x2 = d2) \wedge \ldots$$
$$(pc = 4 \Rightarrow c \bmod 2 \neq 0 \wedge x1 = d1) \wedge \ldots$$

$$D \;\widehat{=}\; (pc \in 2\,..\,6 \Rightarrow pc_q \notin 2\,..\,6) \wedge \ldots$$

Simulation rules



- non-linearization point

  $\forall\, a, c, c' \cdot ABS \wedge INV \wedge COp \Rightarrow ABS' \wedge INV'$

- linearization point

  $\forall\, a, a', c, c' \cdot ABS \wedge INV \wedge COp \wedge AOp \Rightarrow ABS' \wedge INV'$

- non-interference

  $\forall\, a, c, c', c_q \cdot ABS \wedge INV \wedge INV_q \wedge D \wedge COp \Rightarrow INV'_q \wedge D'$

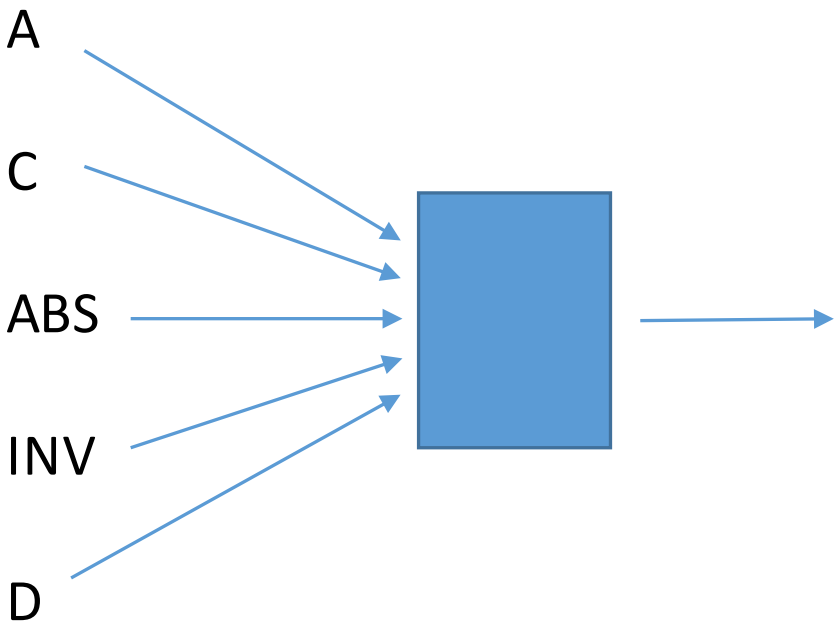
- initialisation

  $\forall\, a, c, c_q \cdot CInit \wedge CInit_q \wedge AInit \Rightarrow ABS \wedge INV \wedge D$

Tool support

- theorem proving (KIV, Isabelle)
- model checking (TLC, NuSMV)
- A number of approaches for model checking linearizability exist
  - check scenarios consisting of a finite number of threads calling a finite number of operations
  - number of threads limited to between 2 and 6
  - number of operations calls can be as limited as 2 or 3
- Our approach — thread-local and step-local
  - arbitrary number of threads
  - (potentially) arbitrary number of operation calls

Model checking approach



$M_1 \models \Box\,(ABS \wedge INV)$ simulation

$M_1 \models \Box\,(INV_q \wedge D)$ non-interference

$M_2 \models (ABS \wedge INV \wedge D)$ initialisation

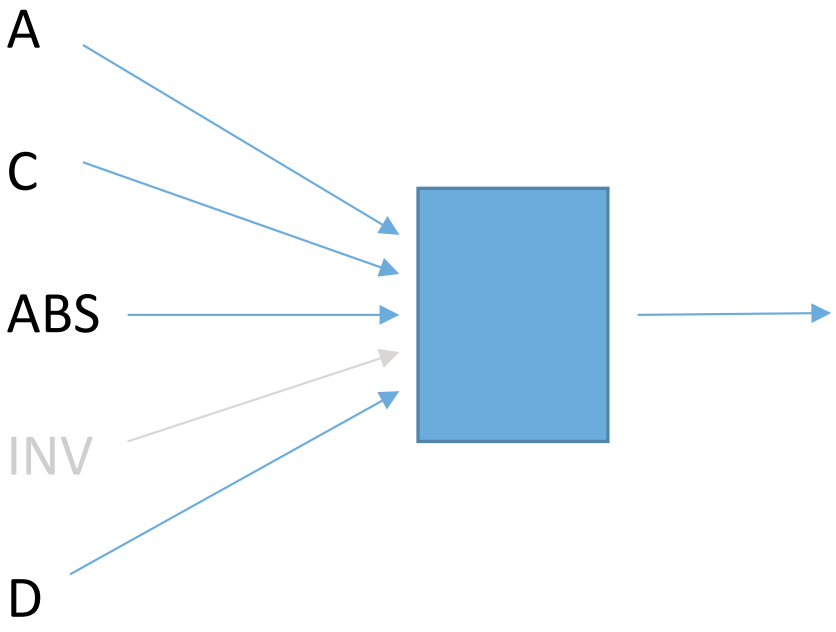
where $M1$ is

$Init \mathrel{\widehat{=}} \neg\, stop \wedge ABS \wedge INV \wedge INV_q$
$Next \mathrel{\widehat{=}} (COp_1 \vee COp_2 \vee \ldots \vee COp_n) \wedge stop'$

and $M2$ is

$Init \mathrel{\widehat{=}} \neg\, stop \wedge AInit \wedge CInit \wedge CInit_q$
$Next \mathrel{\widehat{=}} (COp_1 \vee COp_2 \vee \ldots \vee COp_n) \wedge stop'$

Model checking approach



$M_1 \models \Box\,(ABS \wedge INV)$ simulation

$M_1 \models \Box\,(INV_q \wedge D)$ non-interference

$M_2 \models (ABS \wedge INV \wedge D)$ initialisation

where $M1$ is

$Init \mathrel{\widehat{=}} \neg\, stop \wedge ABS \wedge INV \wedge INV_q$
$Next \mathrel{\widehat{=}} (COp_1 \vee COp_2 \vee \ldots \vee COp_n) \wedge stop'$

and $M2$ is

$Init \mathrel{\widehat{=}} \neg\, stop \wedge AInit \wedge CInit \wedge CInit_q$
$Next \mathrel{\widehat{=}} (COp_1 \vee COp_2 \vee \ldots \vee COp_n) \wedge stop'$

Weak memory models

- High-level programming languages
  - Java
  - C/C++
  - to allow compiler optimisations
- Multicore architectures
  - TSO — used by Intel, AMD
  - Power — used by IBM
  - ARM — used in most mobile devices
  - to improve run-time efficiency
- Allow program instructions to appear as if they occurred out of order

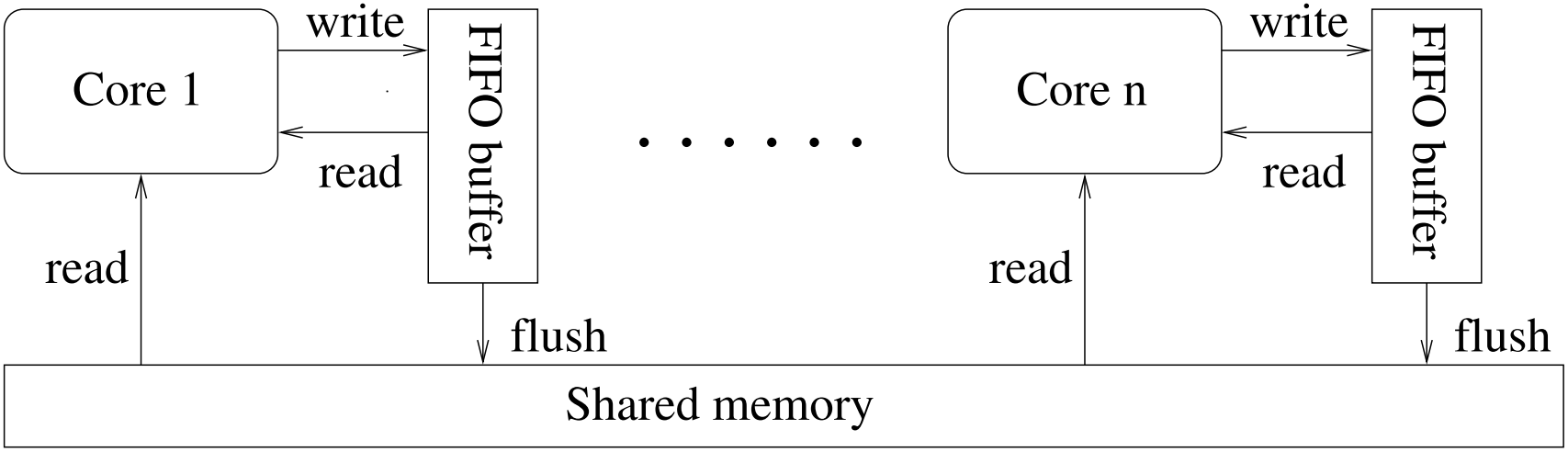
Power and ARM

- Observable behaviour can be derived via systematic testing (litmus tests)
  - Maranget, Sarkar, Sewell (2012)

Thread 1	Thread 2
$x := 1$	$r1 := y$
$y := 1$	$r2 := x$
Initial state: $x = 0 \wedge y = 0$	
Allowed: $r1 = 1 \wedge r2 = 0$	

For example, the above result occurred 1.7G/167G times on Power 7 architecture (1%), and 61K/552M times on ARM APQ8060 (0.01%).

TSO (Total Store Order)



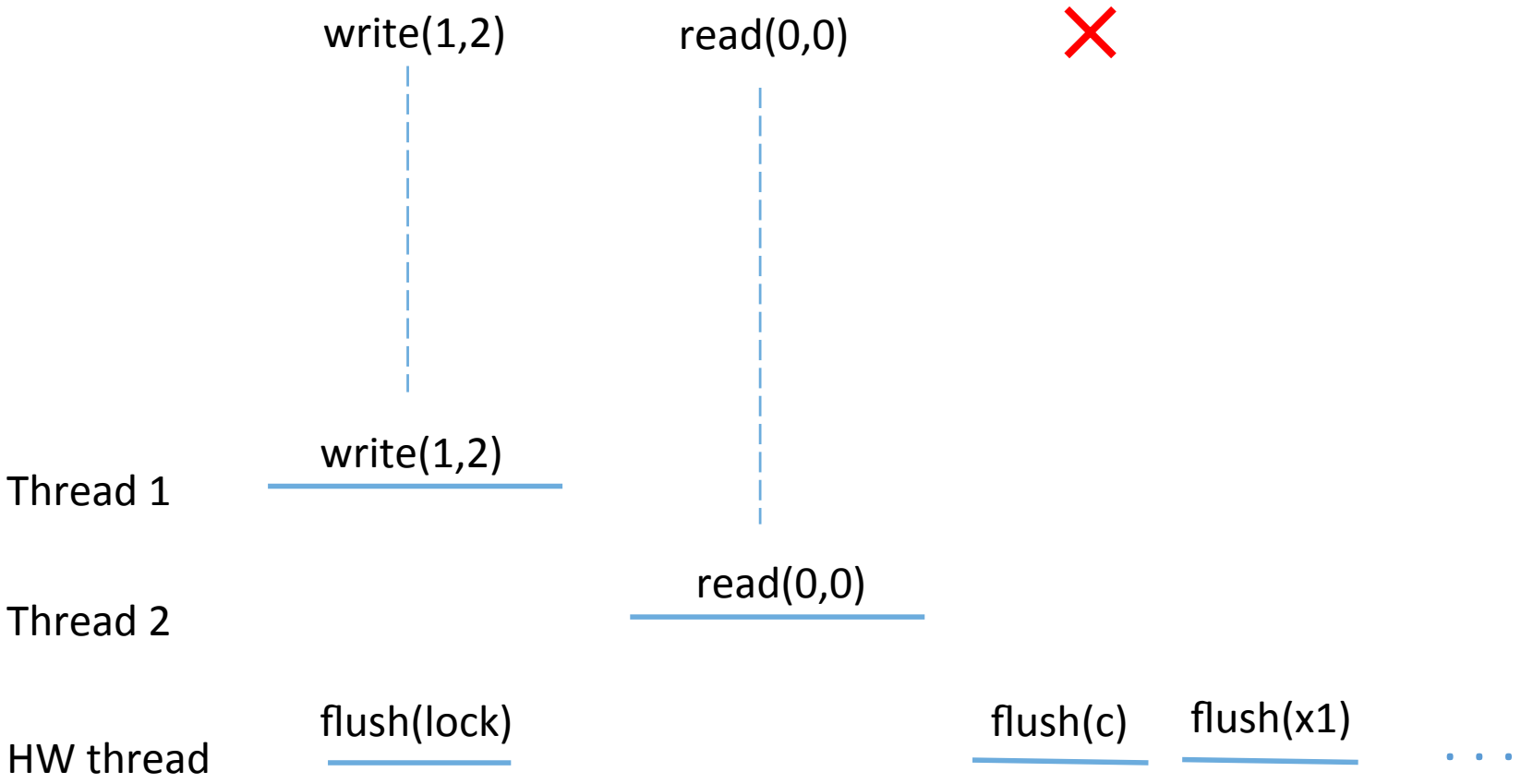
Thread 1 write(1,2)

Thread 2 read(0,0)

HW thread flush(lock) flush(c) flush(x1) . . .

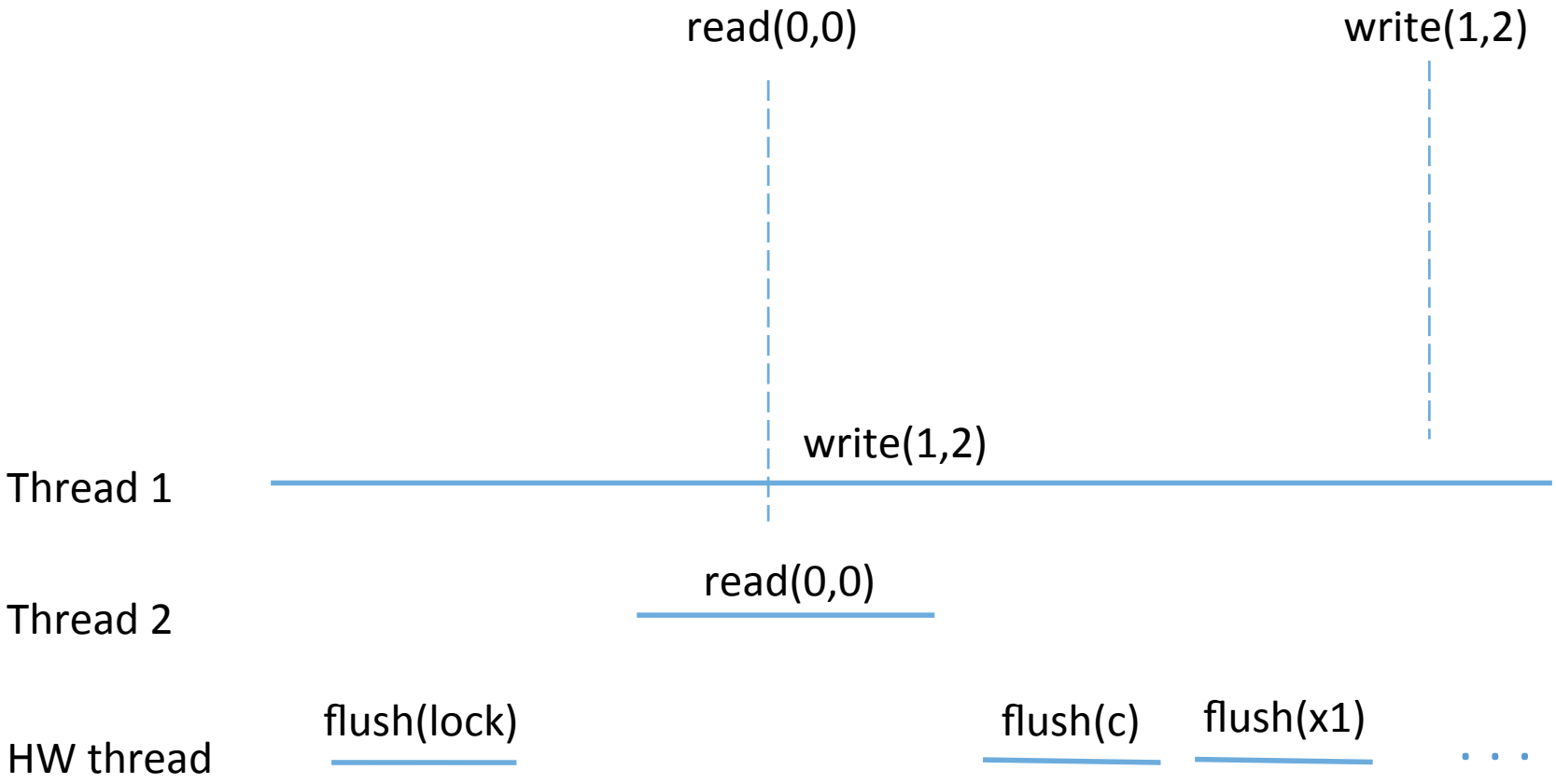
Linearizability

When threads run independently, a client program cannot see relative order of events in different threads



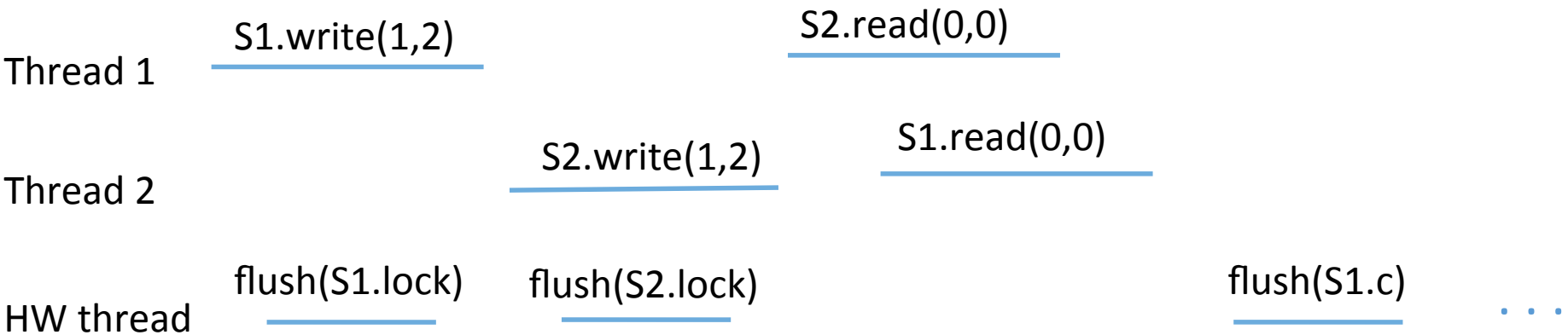
TSO-linearizability

An operation remains active (and hence may take affect) up to the time of the last flush of a value written by the operation



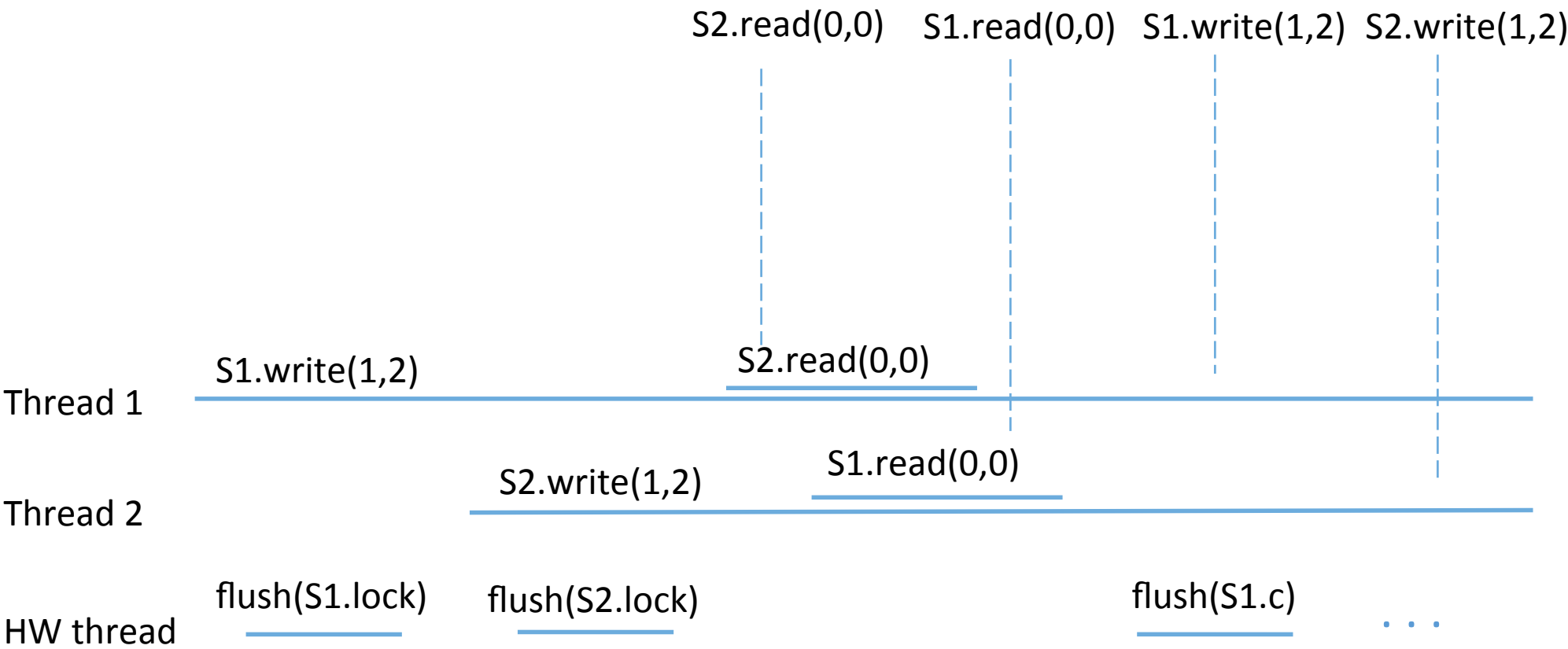
TSO-linearizability

An operation remains active (and hence may take affect) up to the time of the last flush of a value written by the operation



TSO-linearizability

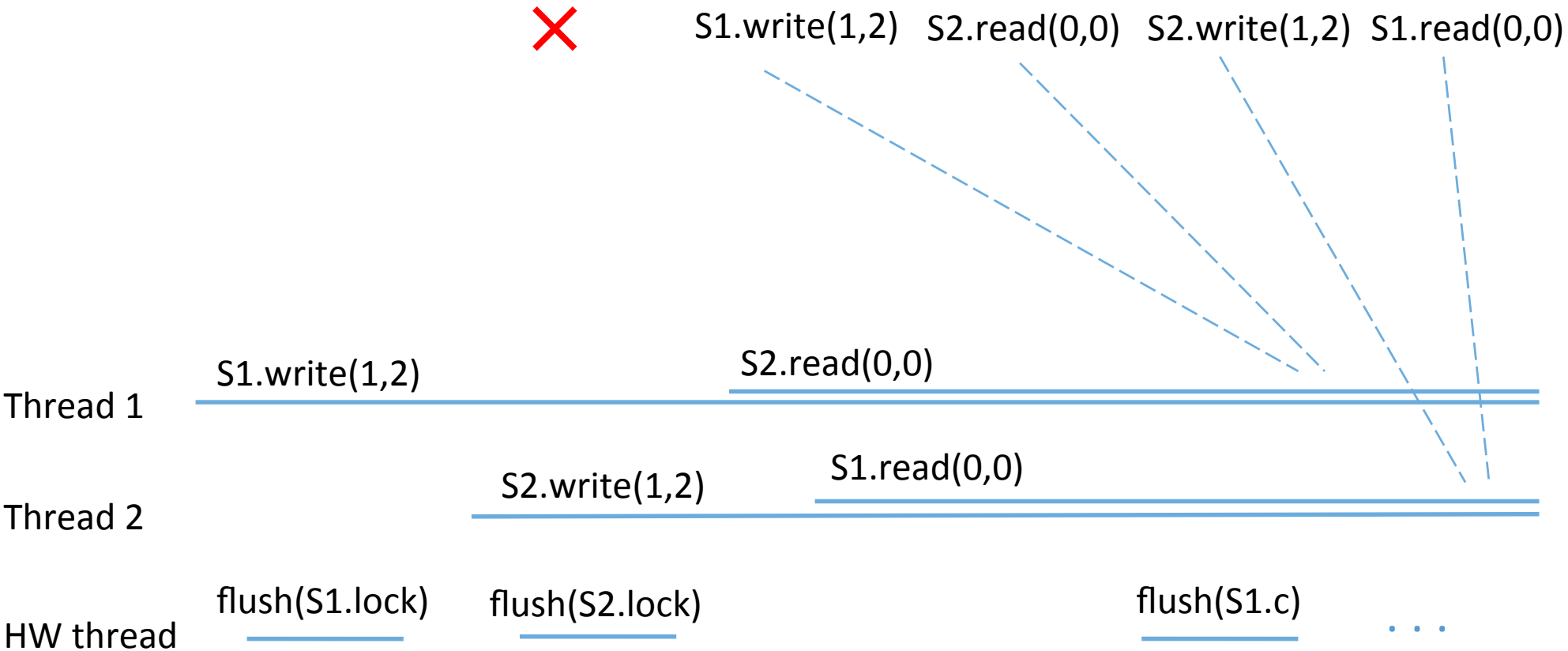
An operation remains active (and hence may take affect) up to the time of the last flush of a value written by the operation



... not sound

TSO-linearizability + sequential consistency

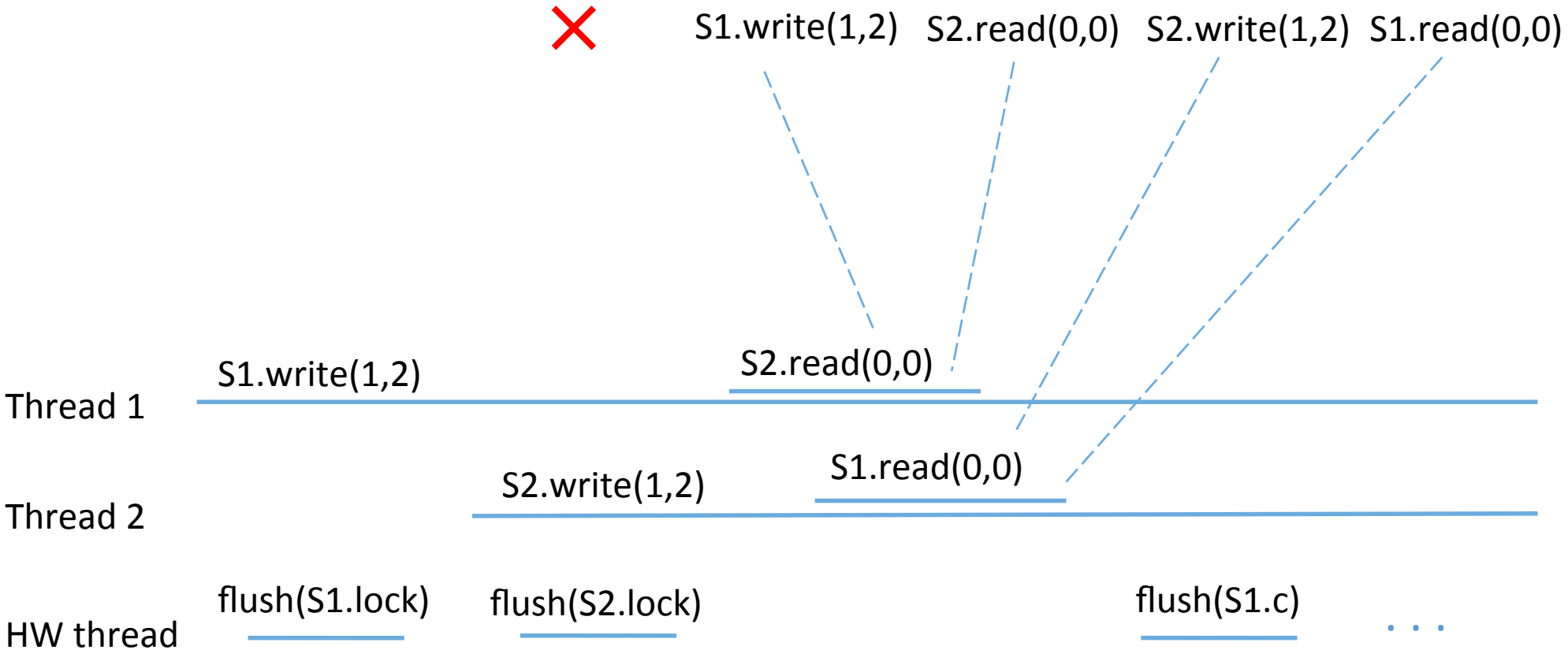
Sequential consistency requires same-thread operations linearize in the order that they are invoked



... but not compositional

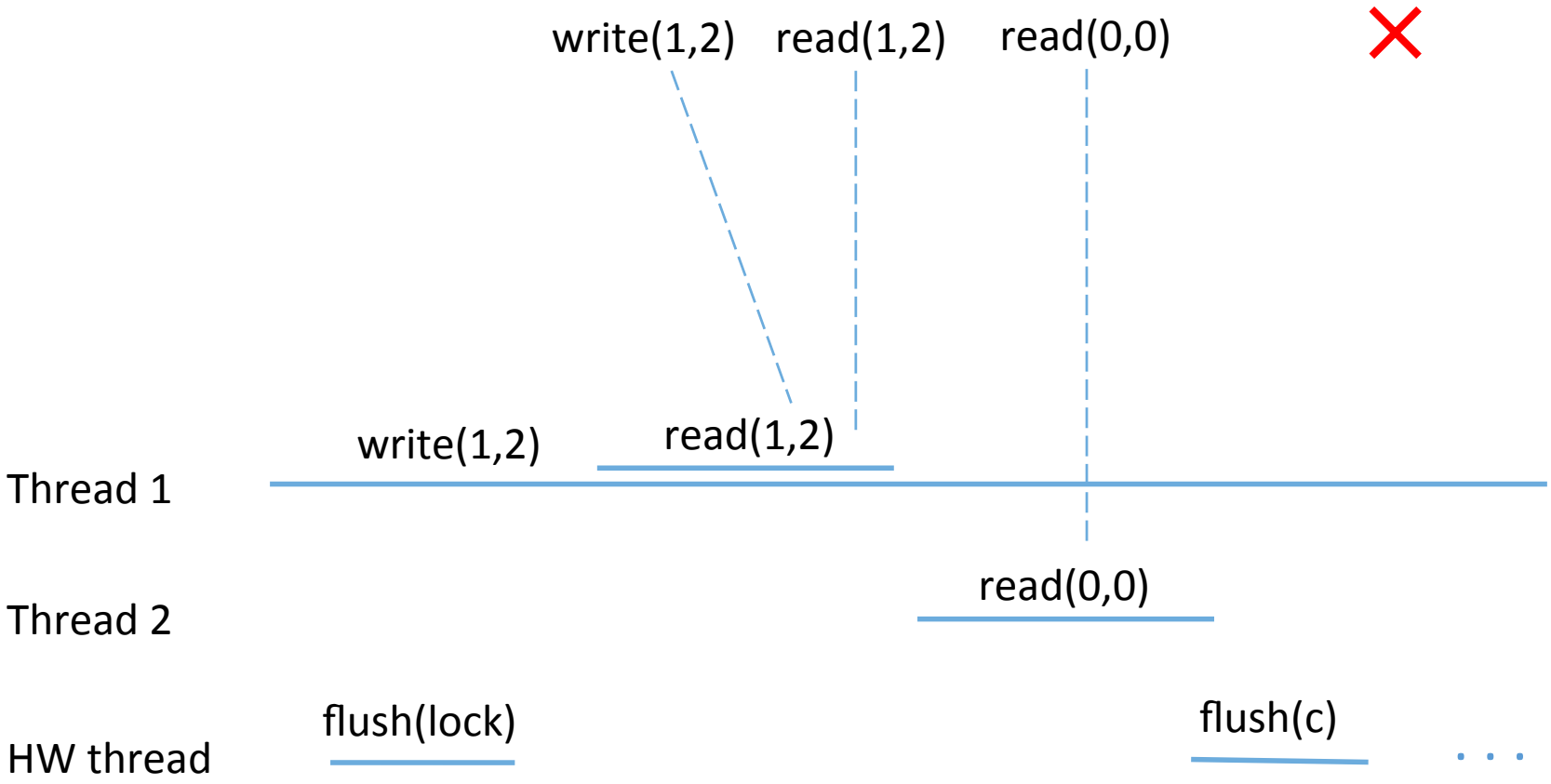
Latest approach – last Tuesday

Operations linearize before the linearization of the next same-thread operation which can be observed by the program



Latest approach – last Tuesday

Seems to be compositional – failure of running example to linearize is due to seqlock not linearizing



ARM and Power

- Reordering
  - instructions in a thread may be reordered
  - sequential semantics should be preserved
  - order of accesses to a given shared variable should be preserved
- Speculative execution
  - branches may be executed before the branch condition is evaluated
  - results of such execution cannot be committed until the condition is evaluated
  - allows reordering of instructions after the branch with instructions before
- Lack of multi-copy atomicity
  - thread A may see thread B's writes to global variables before thread C sees them

Reordering

- Reordering
  - instructions in a thread may be reordered
  - sequential semantics should be preserved
  - order of accesses to a given shared variable should be preserved

$$(a;\ c) \xrightarrow{a} c$$

$$\frac{c \xrightarrow{b} c' \qquad a \overset{R}{\Leftarrow} b}{(a;\ c) \xrightarrow{\tilde{b}^{a}} (a;\ c')}$$

where $a \overset{R}{\Leftarrow} b$ means b can be reordered before a

and $\tilde{b}^{a}$ means b takes into account the effect of a (even though a has not yet occurred globally) — *forwarding*

Reordering in TSO

$$\textbf{store}\ x, v \overset{R}{\Leftarrow} \textbf{load}\ r, y$$

$$\textbf{store}\ x, v \overset{R}{\Leftarrow} \textbf{reg}\ r, v'$$

$$a \overset{R}{\not\Leftarrow} b \qquad \text{otherwise}$$

$$\overbrace{\textbf{load}\ r, x}^{\textbf{store}\ x,v} = \textbf{reg}\ r, v$$

Reordering in ARM and Power

$$\textbf{store}\ x, v \overset{R}{\Leftarrow} \textbf{load}\ r, y$$

$$\textbf{store}\ x, v \overset{R}{\Leftarrow} \textbf{store}\ y, w \qquad \text{iff } x \neq y$$

$$\textbf{load}\ r_1, x \overset{R}{\Leftarrow} \textbf{load}\ r_2, y \qquad \text{iff } r_1 \neq r_2, x \neq y$$

$$\textbf{load}\ r, x \overset{R}{\Leftarrow} \textbf{store}\ y, v \qquad \text{iff } x \neq y$$

$$\textbf{guard}\ b \overset{R}{\Leftarrow} \textbf{load}\ r, x \qquad \text{iff } r \text{ nfi } b$$

$$\textbf{store}\ x, v \overset{R}{\Leftarrow} \textbf{guard}\ b$$

$$\textbf{load}\ r, x \overset{R}{\Leftarrow} \textbf{guard}\ b \qquad \text{iff } r \text{ nfi } b$$

$$\textbf{guard}\ b_1 \overset{R}{\Leftarrow} \textbf{guard}\ b_2$$

Speculative execution

- Speculative execution
  - branches may be executed before the branch condition is evaluated
  - results of such execution cannot be committed until the condition is evaluated
  - allows reordering of instructions after the branch with instructions before

Branch point: $(\textbf{guard}\ b;\ c_1)\ []\ (\textbf{guard}\ \neg b;\ c_2)$

$$\frac{c \xrightarrow{\textbf{guard}\ b} c' \qquad eval_\sigma(b) \equiv true}{\sigma \bullet c \xrightarrow{\tau} \sigma \bullet c'}$$

Lack of multi-copy atomicity

- Lack of multi-copy atomicity
  - thread A may see thread B's writes to global variables before thread C sees them

$$w ::= (x, v, nl) \qquad\qquad Q ::= w^*$$

Load:

$$\frac{c \xrightarrow{n:\,\mathbf{load}\ r,x} c' \qquad \forall (y, w, nl) \in Q_1 \bullet y \neq x \vee n \notin nl}{Q_1 \frown (x, v, nl) \frown Q_2 \bullet c \xrightarrow{n:\,\mathbf{reg}\ r,v} Q_1 \frown (x, v, nl') \frown Q_2 \bullet c'}$$

where $n \in nl \Rightarrow nl' = nl$ and $n \notin nl \Rightarrow nl' = nl \frown n$

Lack of multi-copy atomicity

- Lack of multi-copy atomicity
  - thread A may see thread B's writes to global variables before thread C sees them

$$w ::= (x, v, nl) \qquad\qquad Q ::= w^*$$

Store:

$$\frac{c \xrightarrow{n:\ \mathbf{store}\ x,v} c' \qquad \forall\, w \in Q_1 \bullet (x, v, n) \overset{R}{\Leftarrow} w}{Q_1 \frown Q_2 \bullet c \xrightarrow{n:\ \mathbf{store}\ x,v} Q_1 \frown (x, v, n) \frown Q_2 \bullet c'}$$

where $(x, v, n) \overset{R}{\Leftarrow} (y, w, nl)$
iff $head\ nl \neq n \wedge (x = y \Rightarrow n \notin nl)$

Fences

$$\textbf{fence} \overset{R}{\not\Leftarrow} a$$

$$a \overset{R}{\not\Leftarrow} \textbf{fence}$$

$$\textbf{cfence} \overset{R}{\not\Leftarrow} \textbf{load}\ r, x \qquad\qquad \textbf{guard}\ b \overset{R}{\not\Leftarrow} \textbf{cfence}$$

$$\frac{c \overset{n:\,\textbf{fence}}{\longrightarrow} c'}{Q \bullet c \overset{n:\,\textbf{fence}}{\longrightarrow} flush_n(Q) \bullet c'}$$

$$flush_n(\langle\,\rangle) = \langle\,\rangle$$
$$flush_n((x, v, nl) \frown Q) = (x, v, nl) \frown flush_n(Q) \qquad \text{if } n \notin nl$$
$$flush_n((x, v, nl) \frown Q) = (x, v, nl \frown nl') \frown flush_n(Q) \qquad \text{if } n \in nl$$

where nl' contains all thread ids not in nl.

Validation

Passed 347 / 348 litmus tests for ARMv8

Passed 757 / 758 litmus tests for Power 7

Failed test (on both architectures):

$$\textbf{store}\ x, 1;\ \textbf{fence};\ \textbf{store}\ y, 1\ ||$$
$$\textbf{load}\ r0, y;\ \textbf{eor}\ r1, r0, r0;\ \textbf{add}\ r1, r1, 1;\ \textbf{store}\ z, r1;\ \textbf{store}\ z, 2;$$
$$\textbf{load}\ r3, z;\ (\textbf{if}\ r3 = r3\ \textbf{then}\ \mathit{noOp}\ \textbf{else}\ \mathit{noOp});\ \textbf{cfence};\ \textbf{load}\ r4, x$$

Our semantics allows result $z = 2 \wedge r0 = 1 \wedge r4 = 0$. That is, last statement of process 2 occurs before first statement.