From C/C++11 to POWER and ARM: What is Shared-Memory Concurrency, Anyway?

Susmit Sarkar

University of St Andrews

MMnet, Heriot Watt

May, 2013

Shared Memory Concurrency: Since 1962

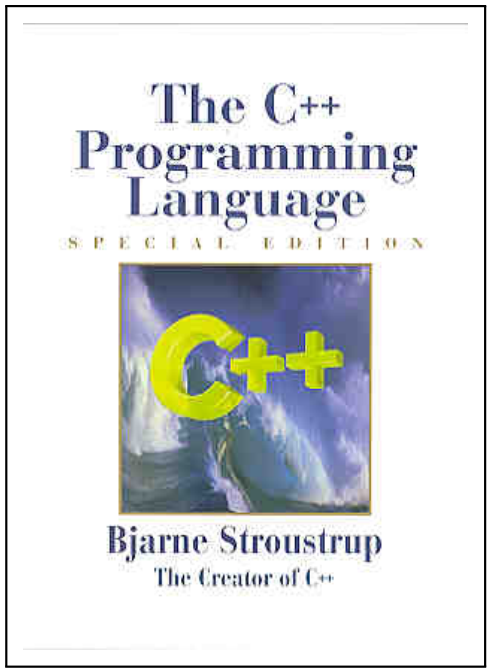
Burroughs D825 (first multiprocessing computer)

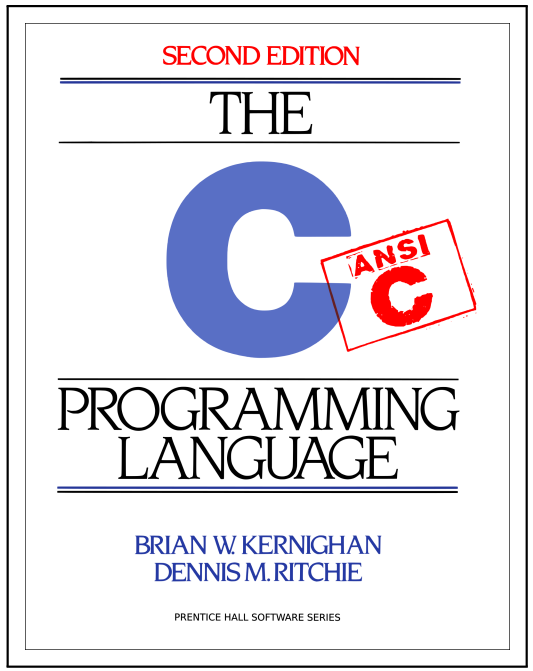
Outstanding features include truly modular hardware with parallel processing throughout.

FUTURE PLANS

The complement of compiling languages is to be expanded.

And Since 2011: In C/C++





ISO C/C++11: introduces a new concurrency model

Example: Message Passing

Initially: d = 0; f = 0;	
Thread 0	Thread 1

`d = 1;`
`f = 1;`
	`while (f == 0)`
	`{};`
	`r = d;`

Finally: r = 0 ??	

- Programmer would hope this is Forbidden

Example: Message Passing (racy)

Initially: d = 0; f = 0;	
Thread 0	Thread 1

`d = 1;`
`f = 1;`
	`while (f == 0)`
	`  {};`
	`r = d;`

Finally: r = 0 ??	

- Programmer would hope this is Forbidden
- In C/C++11, this has undefined semantics
- Data race on d and f variables

C11: A Data Race Free Model

Idea: Programmer mistake to write Data Races

Basis of C11 Concurrency

Example (contd.): mark atomics

Mark atomic variables (accesses have memory order parameter)

Initially: atomic d = 0; f = 0;	
Thread 0	Thread 1

`d.store(1,sc);`
`f.store(1,sc);`
	`while (f.load(sc) == 0)`
	`  {};`
	`r = d.load(sc);`

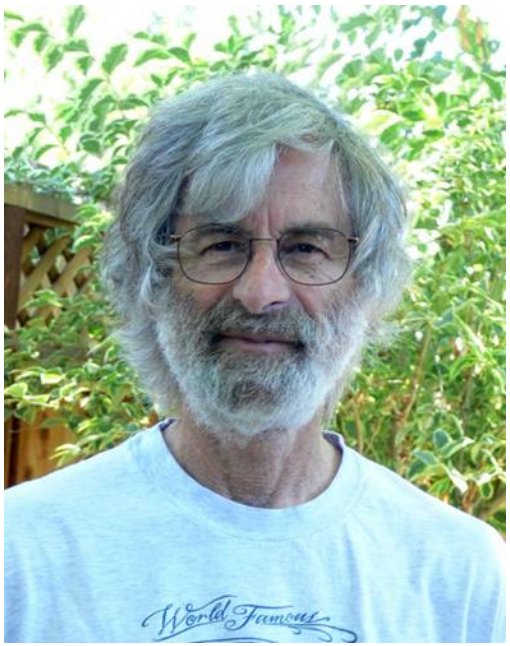
Finally: r = 0 ??	

Races on Atomic Accesses ignored (now have defined semantics)

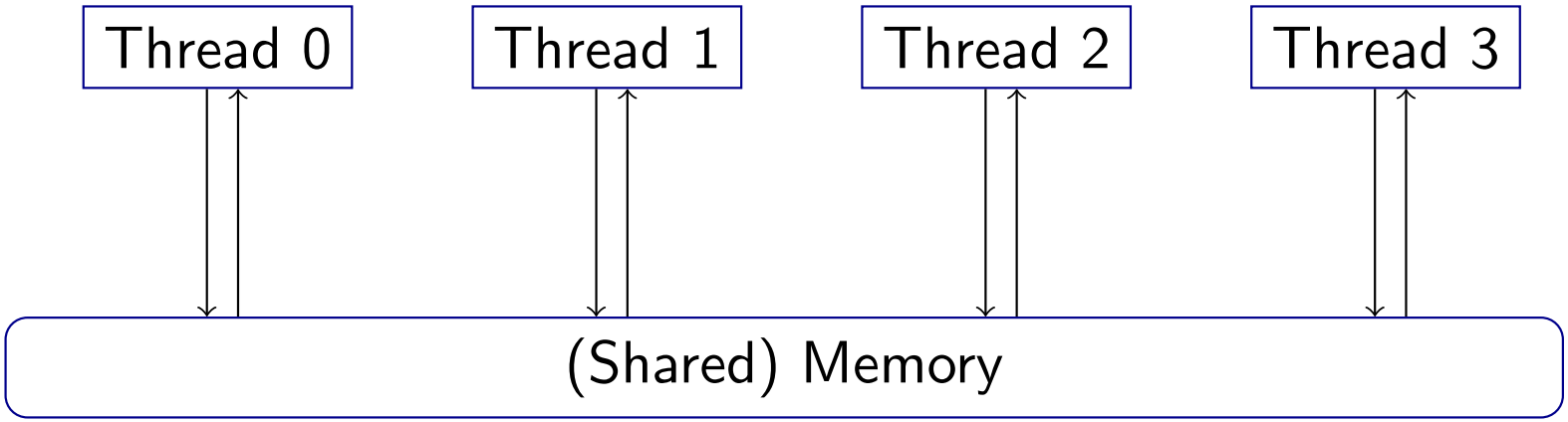
Shared Memory Concurrency

- Multiple *threads* with a single *shared memory*
- Question: How do we reason about it?
- Answer [1979]: Sequential Consistency

> *. . . the result of any execution is the same as if the operations of all the processors were executed in some sequential order, respecting the order specified by the program.*
>
> [Lamport, 1979]

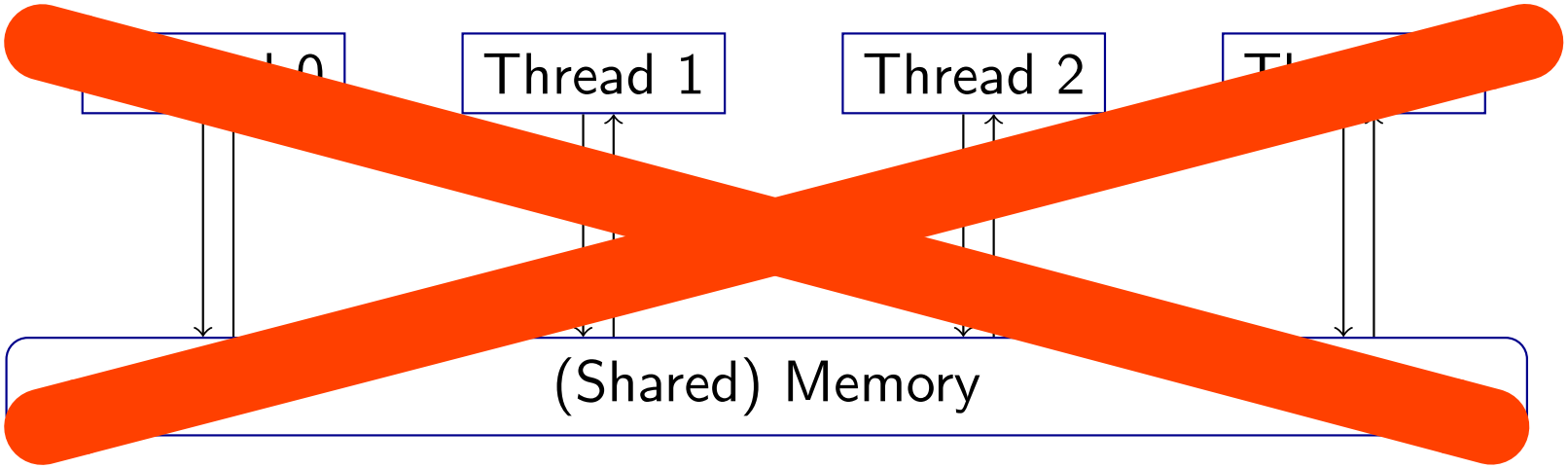


Sequential Consistency



- Traditional assumption (concurrent algorithms, semantics, verification): Sequential Consistency (SC)
- Implies: can use interleaving semantics

Sequential Consistency



- Traditional assumption (concurrent algorithms, semantics, verification): Sequential Consistency (SC)
- Implies: can use interleaving semantics

- **False** on modern (since 1972) multiprocessors, *or* with optimizing compilers

Our world is *not* SC

Not since IBM System 370/158MP (1972)

Our world is *not* SC

Not since IBM System 370/158MP (1972)

. . . . . Nor in x86, ARM, POWER, SPARC, Itanium, . . .

. . . . . Nor in C, C++, Java, . . .

Example (contd.): mark atomics relaxed

Mark atomic variables as relaxed (a memory-order parameter)

Initially: atomic d = 0; f = 0;	
Thread 0	Thread 1

`d.store(1,rlx);`
`f.store(1,rlx);`
	`while (f.load(rlx) == 0)`
	`{};`
	`r = d.load(rlx);`

Finally: r = 0 ??	

- (Forbidden on SC)

Example (contd.): mark atomics relaxed

Mark atomic variables as relaxed (a memory-order parameter)

Initially: atomic d = 0; f = 0;	
Thread 0	Thread 1

`d.store(1,rlx);`
`f.store(1,rlx);`
	`while (f.load(rlx) == 0)`
	`{};`
	`r = d.load(rlx);`

Finally: r = 0 ??	

- (Forbidden on SC)
- Defined, and possible, in C/C++11
- Allows for hardware (and compiler) optimisations

C11 Concurrency: An Axiomatic Model

- Complete executions are considered
  (threadwise operational, reading arbitrary values)
- Relations defined over memory events (*e.g.* happens-before)
- Predicate says whether execution is consistent
- Further, no consistent execution should have races

Example (contd.): release-acquire synchronization

Mark release stores and acquire loads

Initially: atomic d = 0; f = 0;	
Thread 0	Thread 1

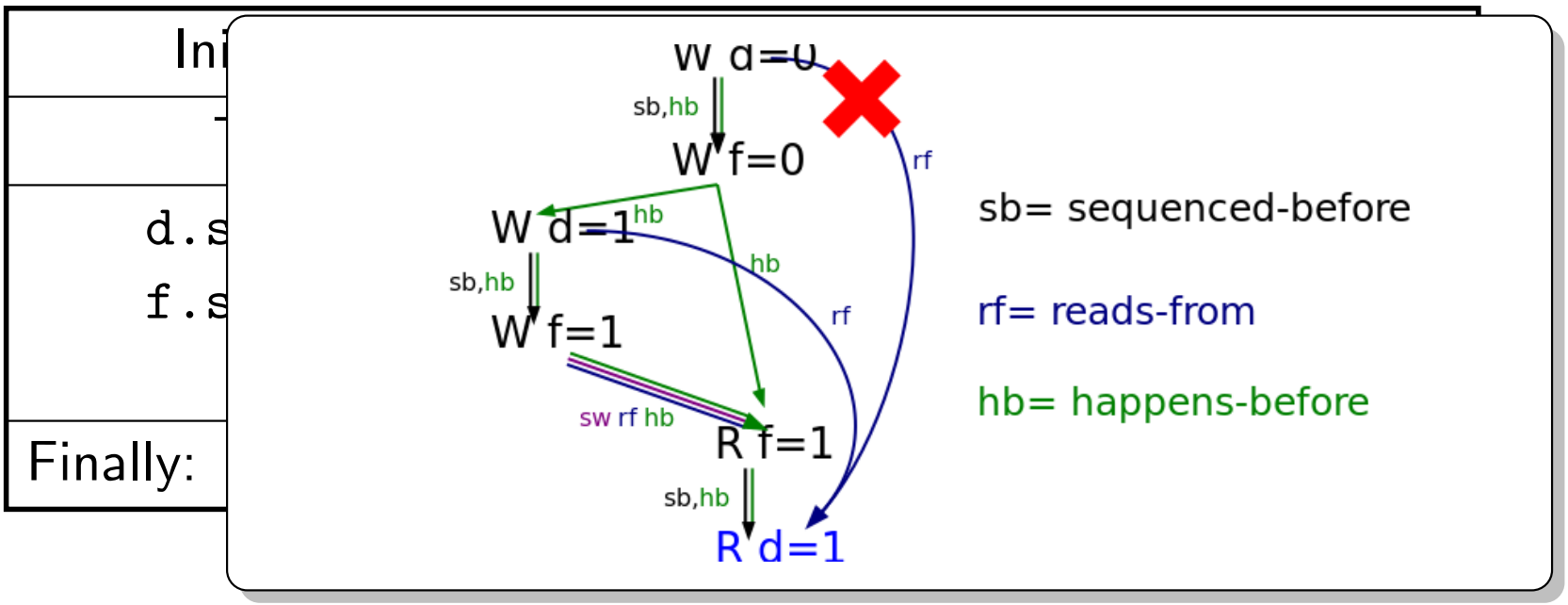
`d.store(1,rlx);`
`f.store(1,rel);`
	`while (f.load(acq) == 0)`
	`  {};`
	`r = d.load(rlx);`

Finally: r = 0 ??	

- (Forbidden on SC)
- Forbidden in C/C++11 due to release-acquire synchronization
- Implementation must ensure result not observed

Example (contd.): release-acquire synchronization

Mark release stores and acquire loads



- (Forbidden on SC)
- Forbidden in C/C++11 due to release-acquire synchronization
- Implementation must ensure result not observed

Implementation of acquire/release on POWER

Initially: d = 0; f = 0;	
Thread 0	Thread 1

`st d 1;`
`lwsync;`
`st f 1;`
	`loop: ld f rtmp;`
	`cmp rtmp 0;`
	`beq loop;`
	`isync;`
	`ld d r;`

Finally: r = 0 ??	

- Forbidden (and not observed) on POWER7, and ARM
- `lwsync` prevents write reordering
- `control dependency` with `isync` prevents read speculation

Correct implementations of C/C++ on hardware

- Can it be done?
  - . . . on highly relaxed hardware?
- What is involved?
  - Mapping new constructs to assembly
  - Optimizations: which ones legal?

Correct implementations of C/C++ on hardware

- Can it be done?
  - . . . on highly relaxed hardware? e.g. POWER/ARM
- What is involved?
  - Mapping new constructs to assembly
  - Optimizations: which ones legal?

Implementing C/C++11 on POWER: Pointwise Mapping

C/C++11 Operation	POWER Implementation
Store (non-atomic)	`st`
Load (non-atomic)	`ld`
Store relaxed	`st`
Store release	`lwsync; st`
Store seq-cst	`lwsync; st`
Load relaxed	`ld`
Load consume	`ld` (and preserve dependency)
Load acquire	`ld; cmp; bc; isync`
Load seq-cst	`hwsync; ld; cmp; bc; isync`
Fence acquire	`lwsync`
Fence release	`lwsync`
Fence seq-cst	`hwsync`
CAS relaxed	`_loop: lwarx; cmp; bc _exit;` `stwcx.; bc _loop; _exit:`
CAS seq-cst	`hwsync; _loop: lwarx; cmp; bc _exit;` `stwcx.; bc _loop; isync; _exit:`
...	...

(From Paul McKenney and Raul Silvera)

Implementing C/C++11 on POWER: Pointwise Mapping

C/C++11 Operation	POWER Implementation
Store (non-atomic)	`st`
Load (non-atomic)	`ld`
Store relaxed	`st`
Store release	`lwsync; st`
Store seq-cst	`lwsync; st`
Load relaxed	`ld`
Load consume	`ld` (and preserve dependency)
Load acquire	`ld; cmp; bc; isync`
Load seq-cst	`hwsync; ld; cmp; bc; isync`
Fence acquire	`lwsync`
Fence release	`lwsync`
Fence seq-cst	`hwsync`
CAS relaxed	`_loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; _exit:`
CAS seq-cst	`hwsync; _loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; isync; _exit:`
...	...

Is that mapping correct?

(From Paul McKenney and Raul Silvera)

Implementing C/C++11 on POWER: Pointwise Mapping

C/C++11 Operation	POWER Implementation
Store (non-atomic)	`st`
Load (non-atomic)	`ld`
Store relaxed	`st`
Store release	`lwsync; st`
Store seq-cst	`lwsync; hwsync; st`
Load relaxed	`ld`
Load consume	`ld` (and preserve dependency)
Load acquire	`ld; cmp; bc; isync`
Load seq-cst	`hwsync; ld; cmp; bc; isync`
Fence acquire	`lwsync`
Fence release	`lwsync`
Fence seq-cst	`hwsync`
CAS relaxed	Answer: No!
CAS seq-cst	`hwsync; _loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; isync; _exit:`
...	...

(From Paul McKenney and Raul Silvera)

Implementing C/C++11 on POWER: Pointwise Mapping

C/C++11 Operation	POWER Implementation
Store (non-atomic)	`st`
Load (non-atomic)	`ld`
Store relaxed	`st`
Store release	`lwsync; st`
Store seq-cst	`hwsync; st`
Load relaxed	`ld`
Load consume	`ld (and preserve dependency)`
Load acquire	
Load seq-cst	
Fence acquire	`lwsync`
Fence release	`lwsync`
Fence seq-cst	`hwsync`
CAS relaxed	
CAS seq-cst	`hwsync; _loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; isync; _exit:`
...	...

Is that mapping correct?

Answer: Yes!

(From Paul McKenney and Raul Silvera)

Implementing C/C++11 on POWER: Pointwise Mapping

C/C++11 Operation	POWER Implementation
Store (non-atomic)	`st`
Load (non-atomic)	`ld`
Store relaxed	`st`
Store release	`lwsync; st`
Store seq-cst	`hwsync; st`
Load relaxed	`ld`
Load consume	`ld` (and preserve dependency)
L	
L	
Fence acquire	`lwsync`
Fence release	`lwsync`
Fence seq-cst	`hwsync`
CAS relaxed	
CAS seq-cst	`hwsync; _loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; isync; _exit:`
...	...

Is that the only correct mapping?

Answer: No!

(From Paul McKenney and Raul Silvera)

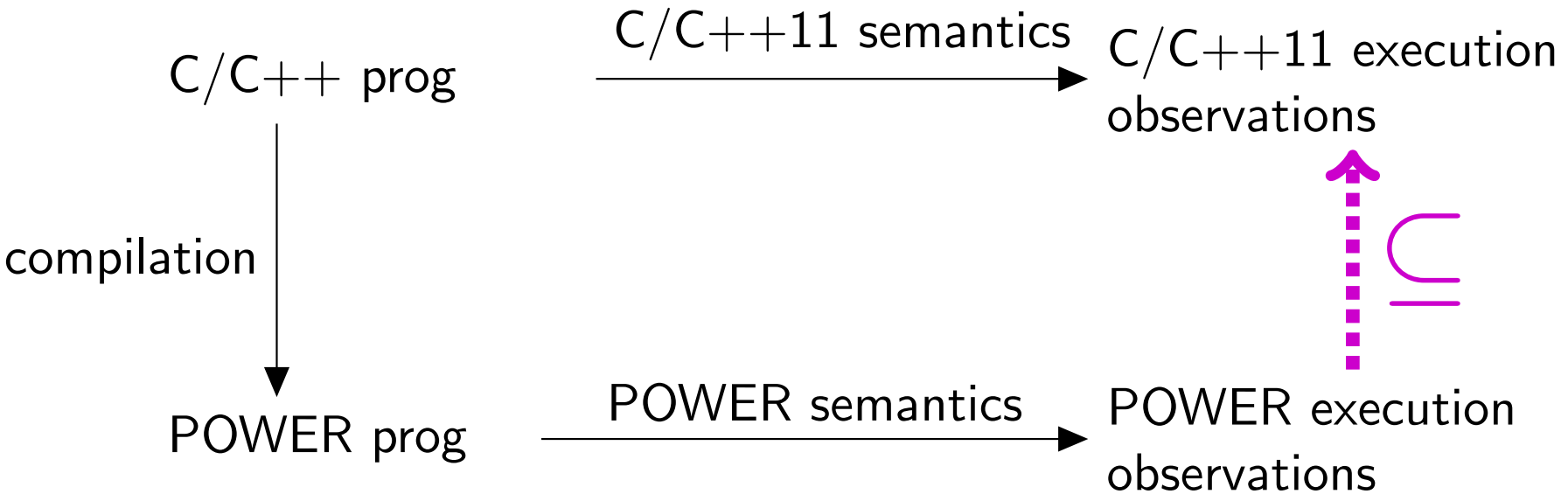
Implementing C/C++11 on POWER: Pointwise Mapping

C/C++11 Operation	POWER Implementation	Alternative
Store (non-atomic)	`st`	
Load (non-atomic)	`ld`	
Store relaxed	`st`	
Store release	`lwsync; st`	
Store seq-cst	`hwsync; st`	`hwsync; st; hwsync;`
Load relaxed	`ld`	
Load consume	`ld` (and preserve dependency)	
Load acquire	`ld; cmp; bc; isync`	
Load seq-cst	`hwsync; ld; cmp; bc; isync`	`ld; hwsync`
Fence acquire	`lwsync`	
Fence release	`lwsync`	
Fence seq-cst	`hwsync`	
CAS relaxed	`_loop: lwarx; cmp; bc _exit;` `stwcx.; bc _loop; _exit:`	
CAS seq-cst	`hwsync; _loop: lwarx; cmp; bc _exit;` `stwcx.; bc _loop; isync; _exit:`	
...	...	

All compilers must agree for separate compilation

Implementing C/C++11 on POWER *correctly*

Theorem: For any sane, non-optimising compiler following the mapping:



- Showed previous mapping incorrect
- Easily adapt proof for an alternative mapping

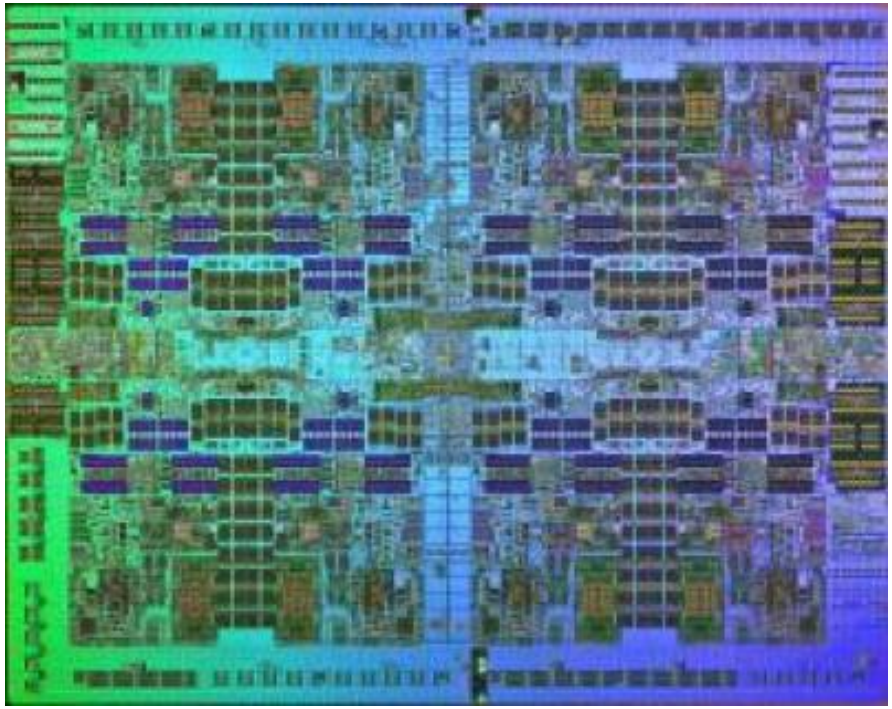
Benefits of a formal proof

Reasoning about industrial-strength concurrency

Enables:

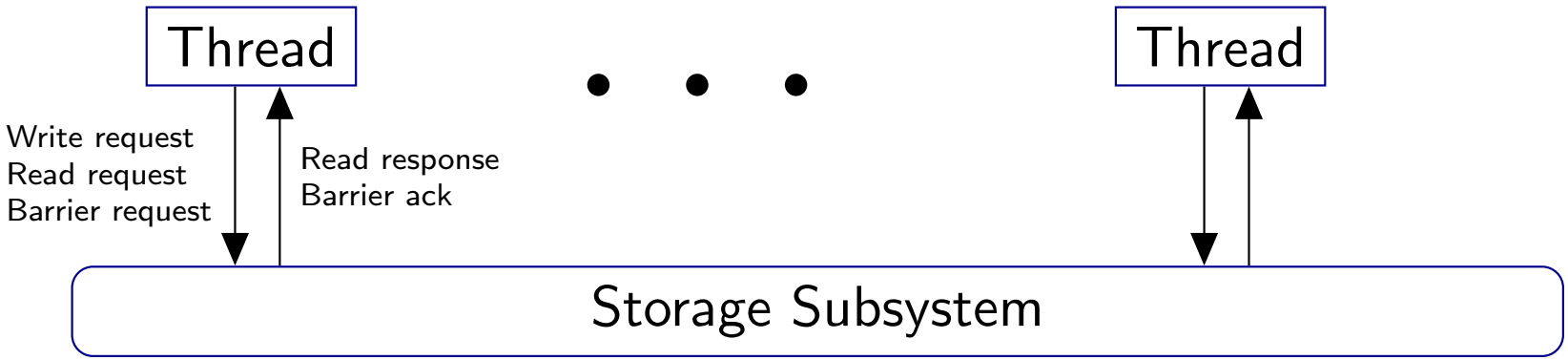
- Confidence in C/C++ and Power concurrency models
- Confidence in compiler implementations [gcc]
- Reasoning about C/C++ and Power
- (Path to) Reasoning about ARM ??

POWER: Hardware Modeling



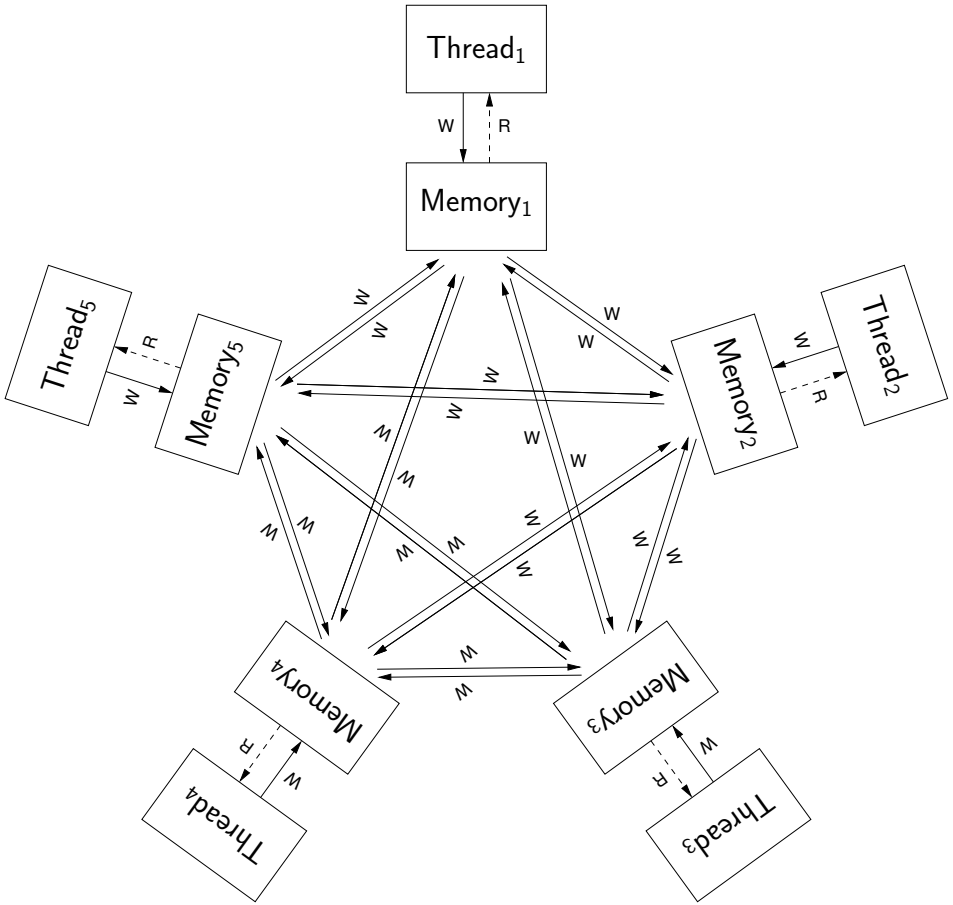
- Hard to see an axiomatic characterisation
- Model the microarchitecture (operational model)
- But, have to be *abstract*

POWER operational model



- Operational model of POWER [PLDI'11]
- Abstract view of microarchitecture
  - Abstract (topology-independent) Storage Subsystem
  - Speculation in threads visible
- Labelled transition systems, synchronising on messages
- 2500 lines of formal mathematics, described in 3 pages of prose

Topology-Independent Storage Subsystem



- Do not expose topology
- Equivalently: Copy of memory per thread
- Have to take into account barriers/ordering instructions

Cumulativity: Programming on many threads

Initially:	d = 0; f = 0;	
Thread 0	Thread 1	Thread 2

`st d 1`	`ld` r_d `d`
	`lwsync`
	`st f 1`
		`loop: ld` r_1 `f;`
		`cmp` r_1 `1;`
		`beq loop;`
		`isync;`
		`ld r` r_2`;`

Finally: $r_d = 1 \wedge r_1 = 1 \wedge r = 0$??		

The lwsync is cumulative: it keeps the stores in order for all threads
Flipping the dependency and barrier does *not* recover SC

A *(slightly)* More Complex Example

Initially: data = 0; flag = 0;	
Thread 0	Thread 1

`data = 1;`
`lwsync;`
`flag = 1;`
	`while (flag == 0)`
	`{};`
	`tmp = 1;`
	r_1 `= tmp;`
	`r = data +` $(r_1 \oplus r_1)$`;`

Finally: r = 0 ??	

- Is that behaviour Allowed? Observable?

A *(slightly)* More Complex Example

Initially: data = 0; flag = 0;	
Thread 0	Thread 1

`data = 1;`
`lwsync;`
`flag = 1;`
	`while (flag == 0)`
	`{};`
	`tmp = 1;`
	`r1 = tmp;`
	`r = data + (r1 ⊕ r1);`

Finally: r = 0 ??	

- Is that behaviour Allowed? Observable?
- Observed on Power7; Allowed by the model

Overall Model Size

Explanation in $\sim$3 pages of prose

- Microarchitectural intuitions
- No extraneous concrete details

$\sim$2500 lines of machine-processed math

- In LEM [ITP'11], a simple new semantic metalanguage
- Can extract executable code, and theorem-prover code

- With OCaml harness: interactive and exhaustive checker
- Compilable to browser!

Validating the model

- Extract executable code from definition, exhaustively enumerate possible behaviours of tests
- Run many iterations of tests on real hardware (Power G5, 6, 7)

Excerpt of results:

Test	Model	POWER 6	POWER 7
WRC+sync+addr	Forbid	ok 0 / 16G	ok 0 / 110G
WRC+data+sync	Allow	ok 150k / 12G	ok 56k / 94G
PPOCA	Allow	unseen 0 / 39G	ok 62k / 141G
PPOAA	Forbid	ok 0 / 39G	ok 0 / 157G
LB	Allow	unseen 0 / 31G	unseen 0 / 176G

- Agreed with key IBM Power designers/architects

Validating the model

- Extract executable code from definition, exhaustively enumerate possible behaviours of tests
- Run many iterations of tests on real hardware (Power G5, 6, 7)

Excerpt of results:

Test	Model	POWER 6	POWER 7
WRC+sync+addr	Forbid	ok 0 / 16G	ok 0 / 110G
WRC+data+sync	Allow	ok 150k / 12G	ok 56k / 94G
PPOCA	Allow	unseen 0 / 39G	ok 62k / 141G
PPOAA	Forbid	ok 0 / 39G	ok 0 / 157G
LB	Allow	unseen 0 / 31G	unseen 0 / 176G

- Agreed with key IBM Power designers/architects

Validating the model

- Extract executable code from definition, exhaustively enumerate possible behaviours of tests
- Run many iterations of tests on real hardware (Power G5, 6, 7)

Excerpt of results:

Test	Model	POWER 6	POWER 7
WRC+sync+addr	Forbid	ok 0 / 16G	ok 0 / 110G
WRC+data+sync	Allow	ok 150k / 12G	ok 56k / 94G
PPOCA	Allow	unseen 0 / 39G	ok 62k / 141G
PPOAA	Forbid	ok 0 / 39G	ok 0 / 157G
LB	Allow	unseen 0 / 31G	unseen 0 / 176G

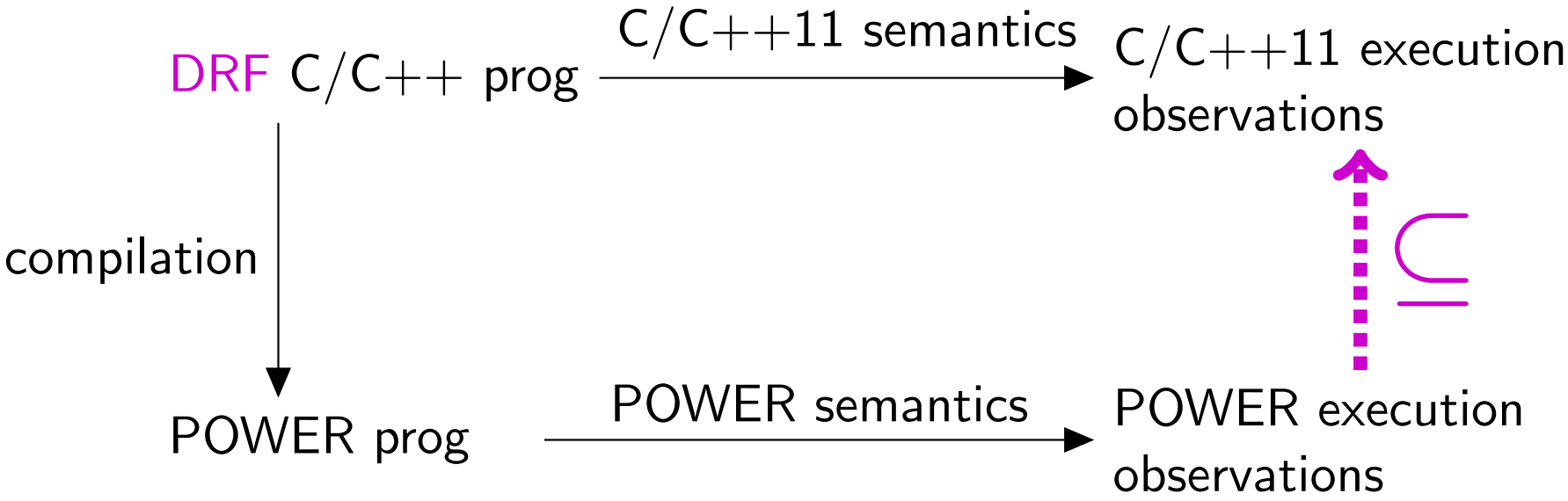
- Agreed with key IBM Power designers/architects

C/C++11 Implementation Proof

And Its Consequences

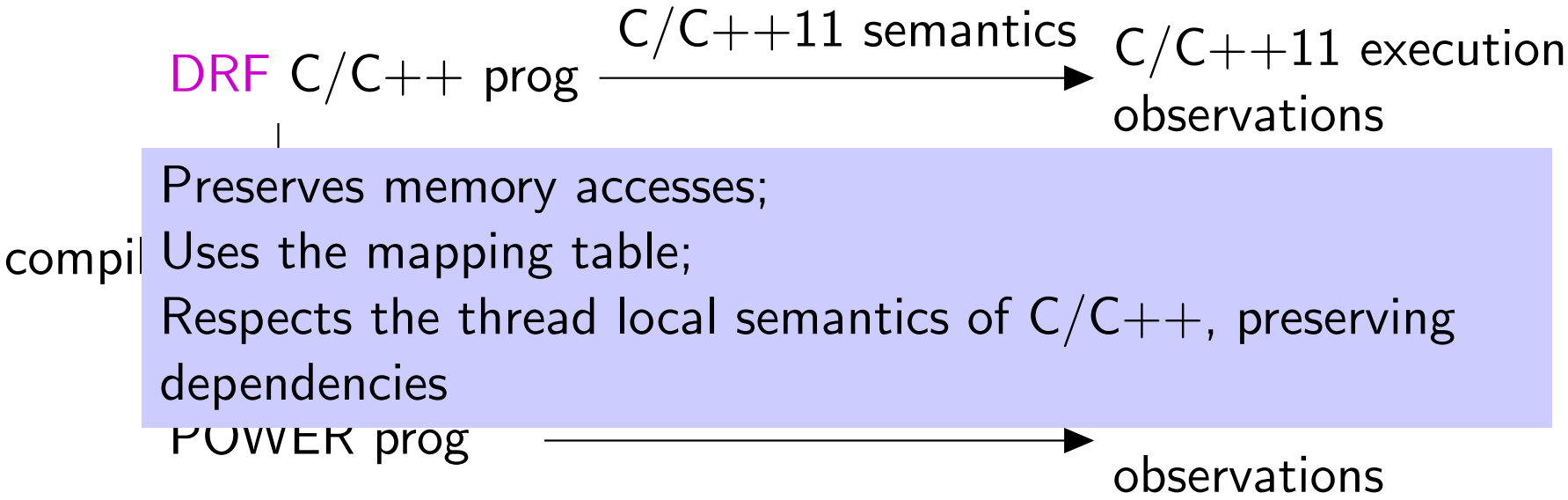
Proof outline

Theorem: For any sane, non-optimising compiler following the mapping:



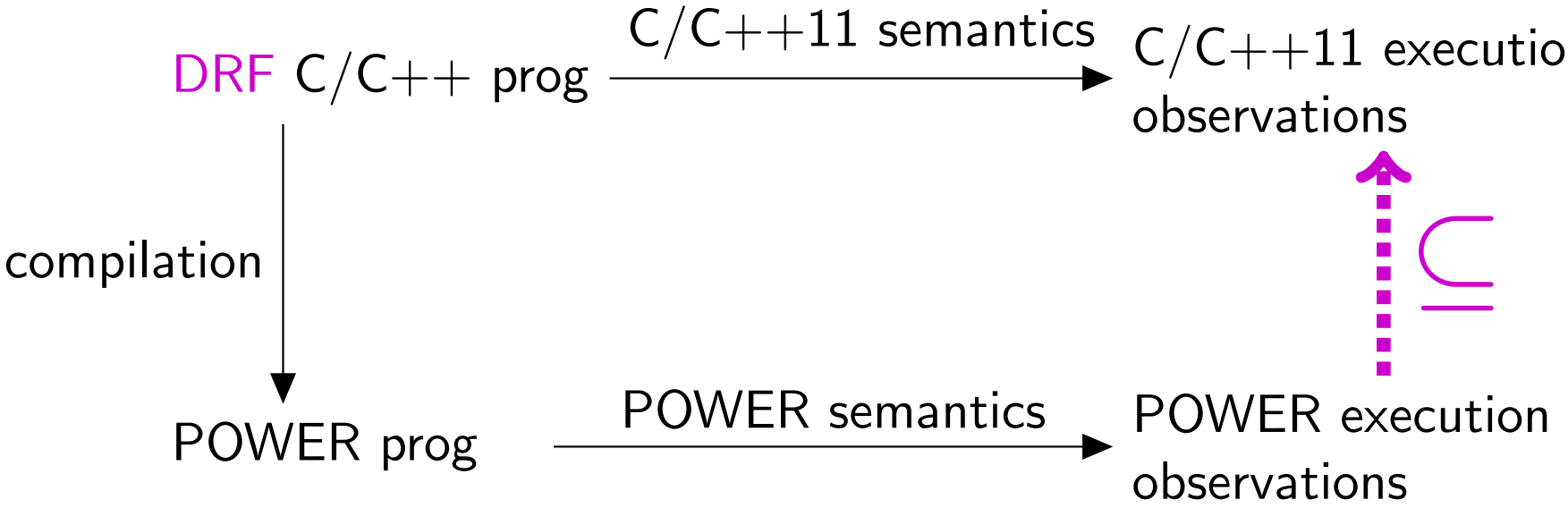
Proof outline

Theorem: For any sane, non-optimising compiler following the mapping:



Proof outline

Theorem: For any sane, non-optimising compiler following the mapping:



From POWER trace, build key relations (happens-before, SC order)

Required properties from abs. machine properties

If trace looks like it produces data race, build the C/C++ data race for contradiction

Building up happens-before (outline)

C11	Power correspondence
Base case: release-acquire	lwsync and isync
Transitive (multiple rel/acq)	Cumulativity of lwsync
Release-consume with dependencies	lwsync and dependencies
Special rules for CAS	coherence-point reasoning
...	...

Using Proofs for Hardware Design

- Previously, similar C11 proof for x86-TSO
  - There, much simpler
- What properties of Hardware were necessary?
- Turns out: x86 Compare-and-Swap have strong properties
- Weakening guarantees: Better implementation, just as good programming [PLDI'13]

Using Proofs for Hardware Design (2)

Initially: data = 0; flag = 0;	
Thread 0	Thread 1

`data = 1;`
`sync;`
`flag = 1;`
	`while (flag == 0)`
	`{};`
	`atomically (flag = 2);`
	r_1 `= flag;`
	`r = data +` ($r_1 \oplus r_1$)`;`

Finally: r = 0 ??	

- Is that Allowed? Observable?

Using Proofs for Hardware Design (2)

Initially: data = 0; flag = 0;	
Thread 0	Thread 1

`data = 1;`
`sync;`
`flag = 1;`
	`while (flag == 0)`
	`{};`
	`atomically (flag = 2);`
	r_1 `= flag;`
	`r = data +` ($r_1 \oplus r_1$)`;`

Finally: r = 0 ??	

- Is that Allowed? Observable?
- C11/C++11 mapping would break (and no good way of fixing)
- Fortunately, current hardware does not do this
- . . . and now we know why future hardware should not

Conclusion

Reasoning about industrial-strength concurrency

- Correct compilation of C/C++ concurrency primitives on Power
- Confidence in both models
- Compiler implementation relevance
- Isolate relevant properties of h/w (Path to Hardware Design)
- Reasoning about machine code at C/C++ level

Thank You!

More details at:

`http://www.cl.cam.ac.uk/~pes20/cppppc`

Understanding POWER Multiprocessors [PLDI'11]
Clarifying and Compiling C/C++ Concurrency: From C++11 to POWER [POPL'12]
Synchronising C/C++ and POWER [PLDI'12]
Fast RMWs for TSO: Semantics and Implementation [PLDI'13]

The `ppcmem` tool at:

`http://www.cl.cam.ac.uk/~pes20/ppcmem`

Model Excerpt

Propagate write to another thread

The storage subsystem can propagate a write w (by thread tid) that it has seen to another thread tid', if:

- the write has not yet been propagated to tid';
- w is coherence-after any write to the same address that has already been propagated to tid'; and
- all barriers that were propagated to tid before w (in $s.events\_propagated\_to\ (tid)$) have already been propagated to tid'.

Action: append w to $s.events\_propagated\_to\ (tid')$.

Explanation: This rule advances the thread tid' view of the coherence order to w, which is needed before tid' can read from w, and is also needed before any barrier that is in tid's view after w (has w in its "Group A") can be propagated to tid'.

Model Excerpt

Propagate write to another thread

```
let write_announce_cand m s w tid' =
  (w IN s.writes_seen) &&
  (tid' IN s.threads) &&
  (not (List.mem (SWrite w) (s.events_propagated_to tid'))) &&
  (forall (w' IN s.writes_seen).
    if List.mem (SWrite w') (s.events_propagated_to tid') && w.w_addr = w'.w_addr
    then (w',w) IN s.coherence
    else true) &&
  (forall (b IN barriers_seen s).
    if (ordered_before_in (s.events_propagated_to w.w_thread)
          (SBarrier b) (SWrite w))
    then List.mem (SBarrier b) (s.events_propagated_to tid') else true)

let write_announce_action s w tid' =
  let events_propagated_to' = funupd s.events_propagated_to tid'
                                (add_event (s.events_propagated_to tid') (SWrite w)
  <| s with events_propagated_to = events_propagated_to' |>
```