False Concurrency

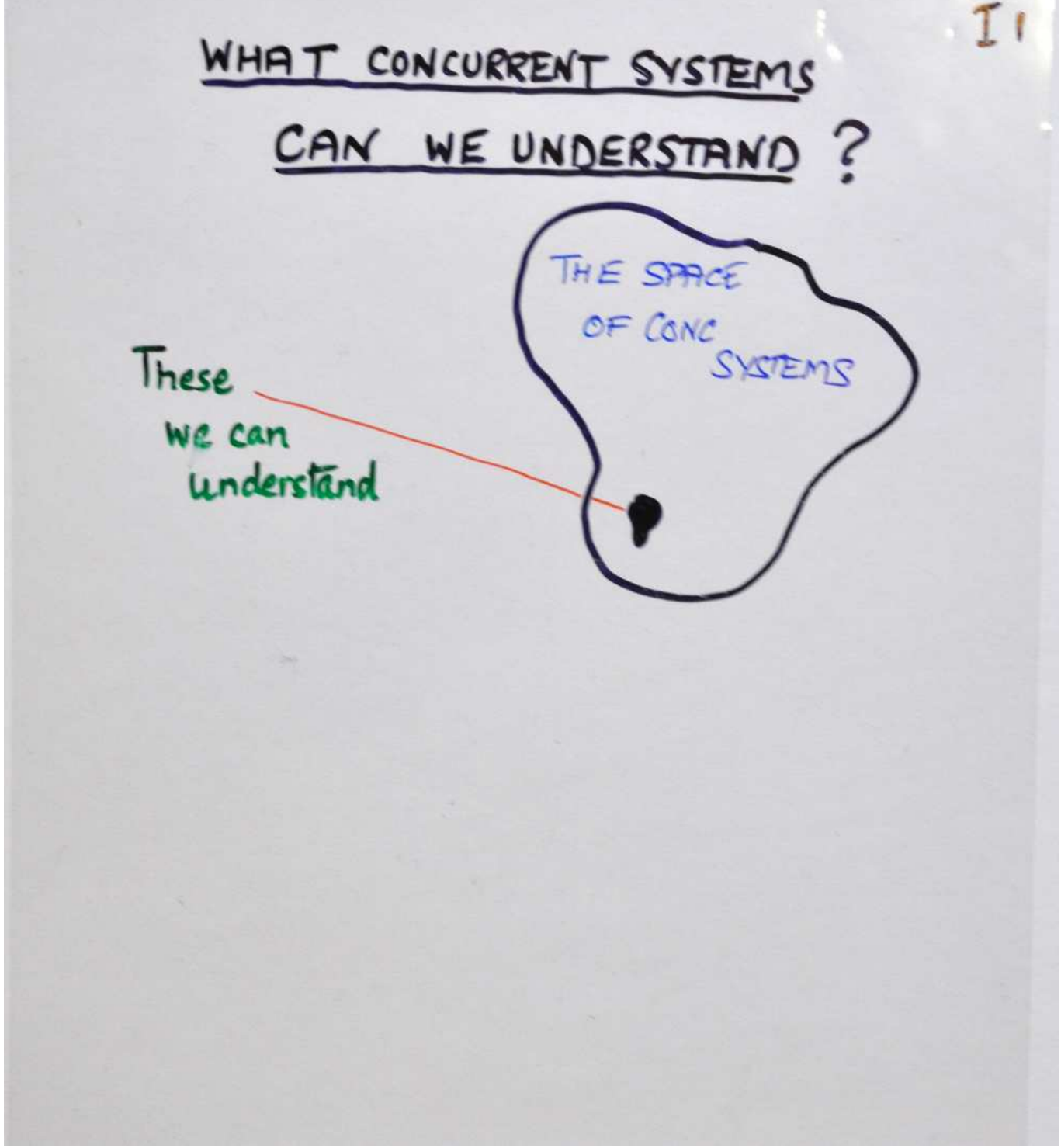
and the

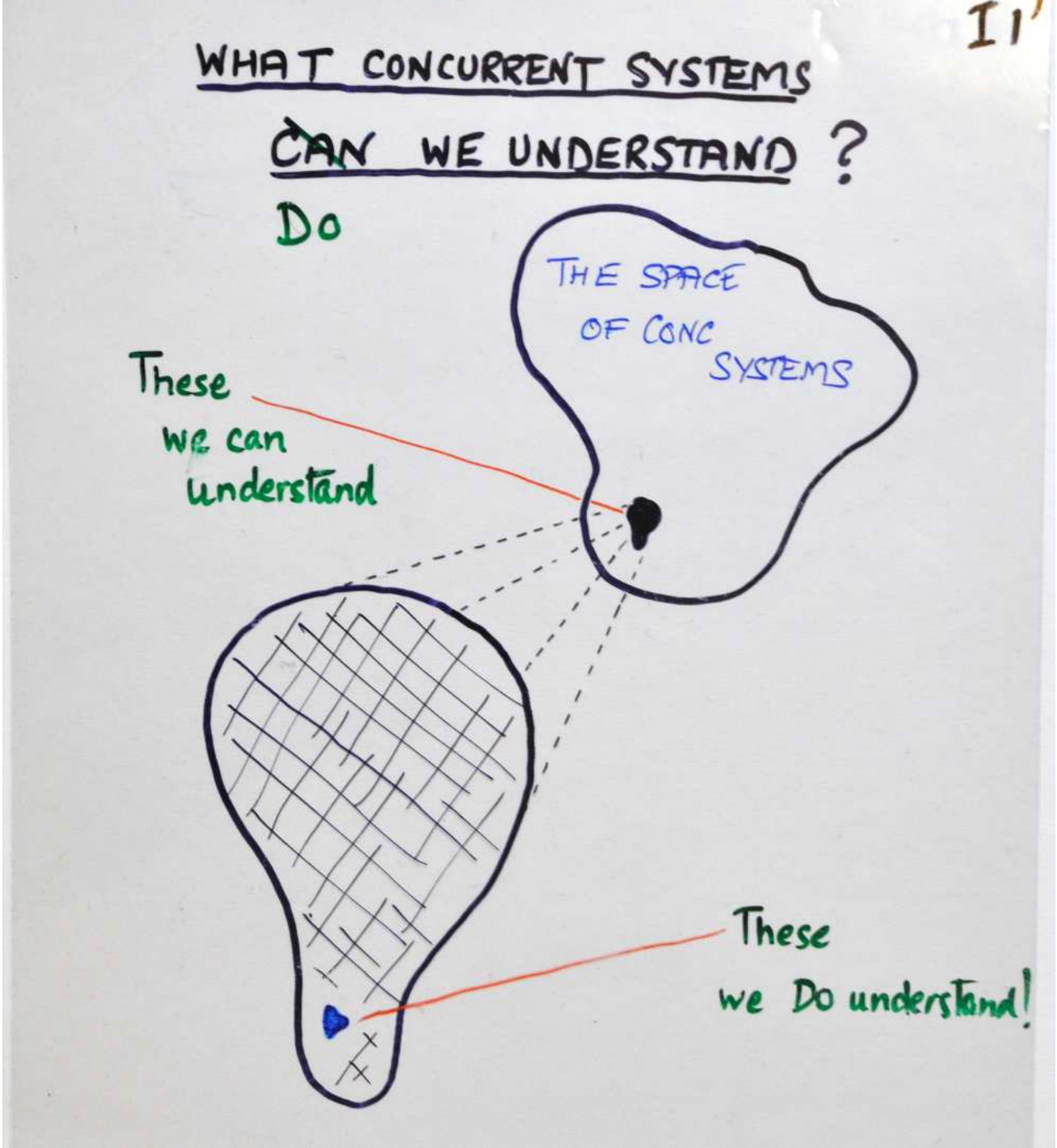
Foundations of Computer Science

Peter Sewell

University of Cambridge

Milner Symposium
Edinburgh, 15–18 April 2012


I1
WHAT CONCURRENT SYSTEMS
CAN WE UNDERSTAND ?
THE SPACE
OF CONC
SYSTEMS
These
we can
understand


WHAT CONCURRENT SYSTEMS
CAN WE UNDERSTAND ?
Do
THE SPACE
OF CONC
SYSTEMS
These
we can
understand
These
we Do understand!

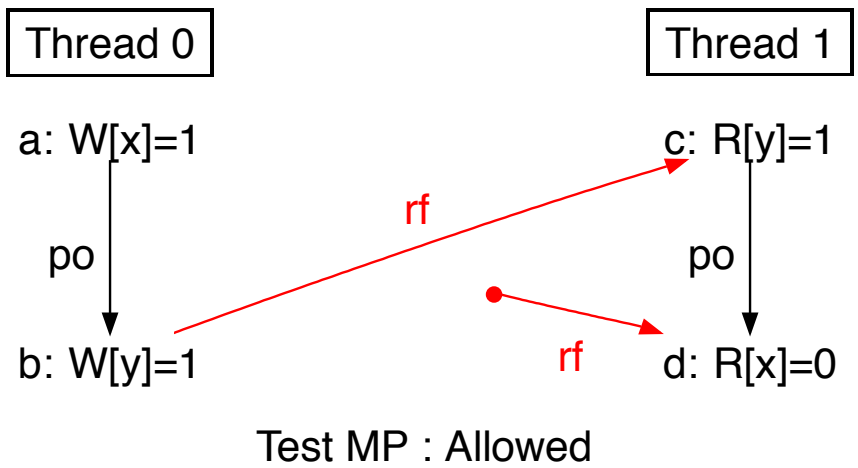
Message passing example

```
int x = 0;          bool y = false;
```

// sender thread	// receiver thread
x = 1;	while (y==false) {};
y = true;	print(x);

Good behaviour: Prints 1

Bad behaviour: Prints 0



Test MP : Allowed

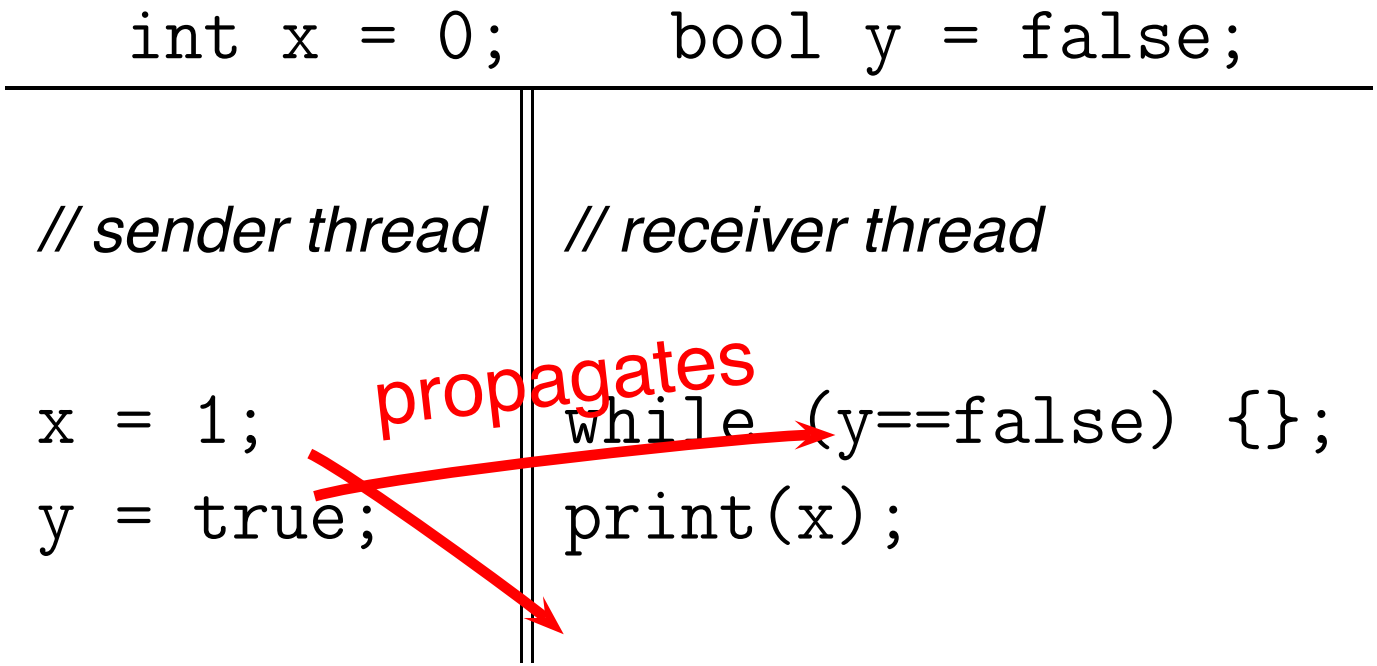
Message passing failure: ‘smart’ compiler

```
int x = 0;          bool y = false;
```

// sender thread	*// receiver thread*
	`int local = x;`
`x = 1;`	`while (y==false) {};`
`y = true;`	`print(local);`

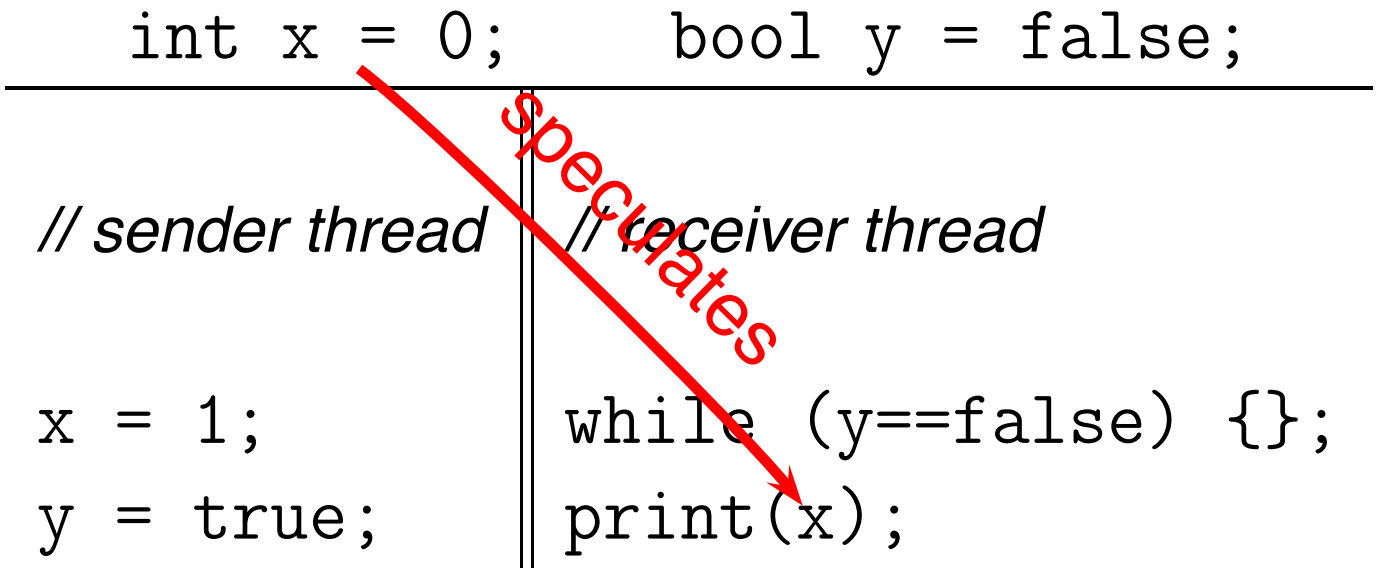
Bad behaviour: Sometimes prints 0

Message passing failure: ‘smart’ processor



Bad behaviour: Sometimes prints 0

Message passing failure: ‘smart’ processor



Bad behaviour: Sometimes prints 0

Message passing fix (1): Power/ARM

`int x = 0;`	`bool y = false;`
// sender thread	*// receiver thread*
`x = 1;`	`while (y==false) {};`
`lwsync/DMB;`	`isync/ISB;`
`y = true;`	`print(x);`

Good behaviour: Always prints 1

Message passing fix (2): Power/ARM

```
int x = 0;     int *y = NULL;
```

// sender thread	*// receiver thread*
`x = 1;`	`int *local = y;`
`lwsync/DMB;`	`while (local == NULL) {local = y;};`
`y = &x;`	`print(*local);`

(address dependency, from value of one read to address of the next)

Good behaviour: Always prints 1

Basic Question

What *is* the concurrency semantics of Power/ARM processors?

We've built a model...

[Susmit Sarkar, Jade Alglave, Luc Maranget, Derek Williams, Sewell]

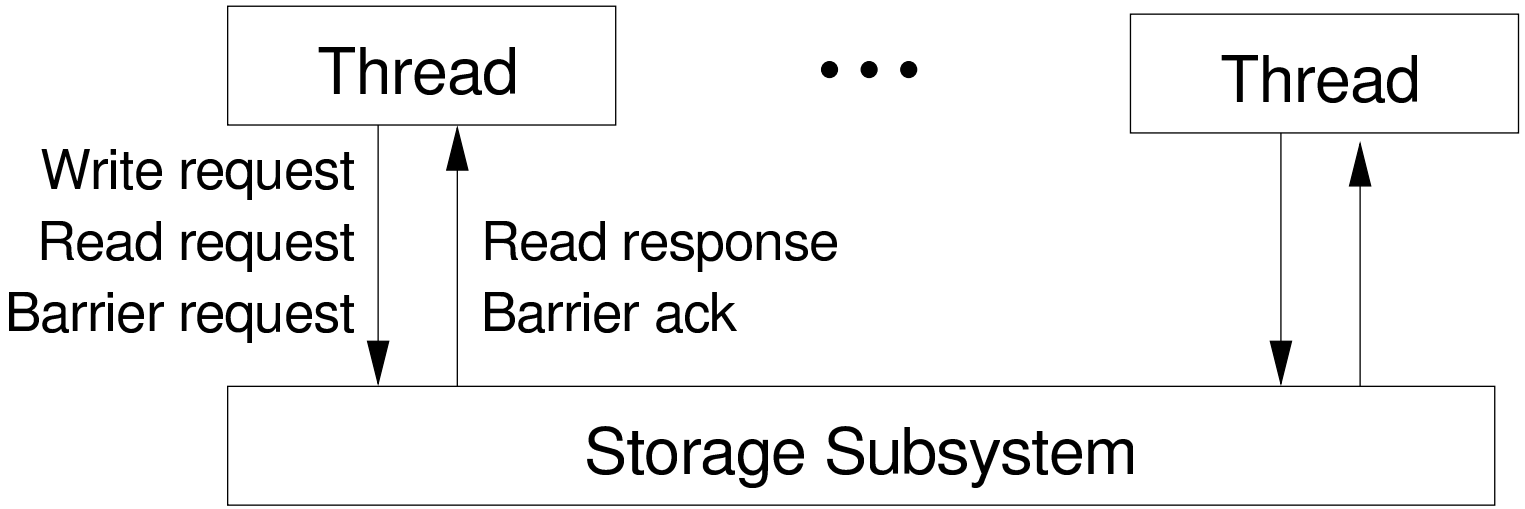
...by a long process of

- generating test cases
- experimental testing of hardware
- talking with IBM and ARM architects
- checking candidate models

Contrasts with Classic Concurrency Theory

Operational abstract-machine model:

- thread-local LTS *(speculation)*
- storage subsystem LTS *(propagation)*
- top-level LTS parallel composition of those



SOS? rules? interleaving? observational? obs.cong? mechanised? tools?

Sample Transition Rule

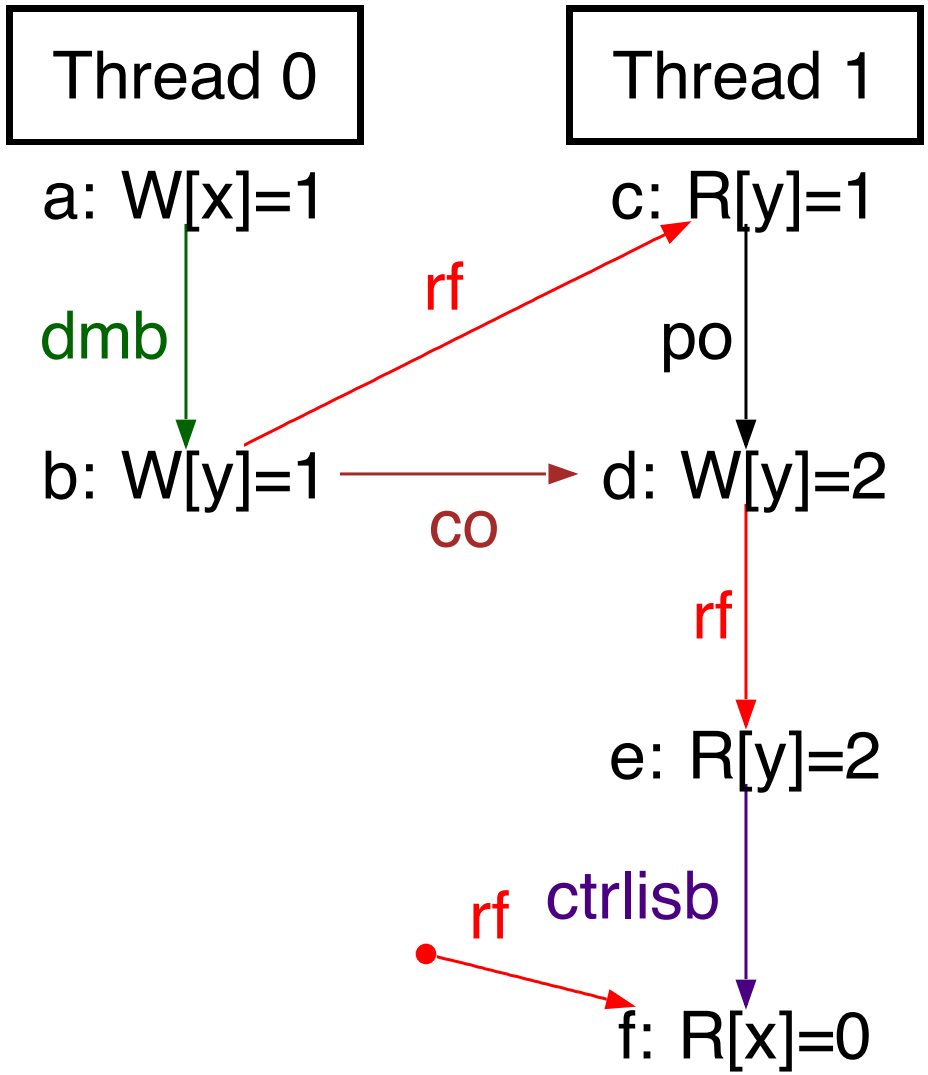
Propagate write to another thread (a τ transition)
The storage subsystem can propagate a write w (by thread tid) that it has seen to another thread tid', if:

- 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 has w in its "Group A" can be propagated to tid'.

MP+dmb+fri-rfi-ctrlisb?



Test MP+dmb+fri-rfi-ctrlisb

Message passing in C/C++11, relaxed

```
int x = 0;     atomic_bool y = false;
```

// sender thread	*// receiver thread*
`x = 1;`	`while (y.load(relaxed)==`
`y.store(true,relaxed);`	`false) {};`
	`print(x);`

Bad behaviour: Can print 0

Message passing fix, C/C++11, rel/acq

```
int x = 0;     atomic_bool y = false;
```

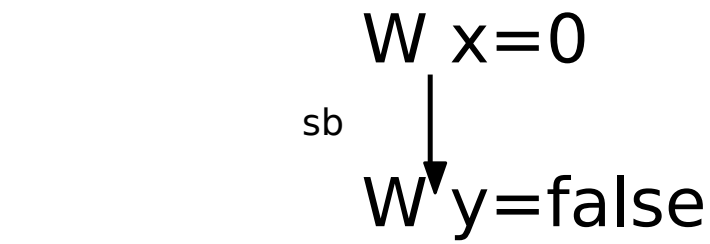
// sender thread	*// receiver thread*
`x = 1;`	`while (y.load(acquire)==`
`y.store(true,release);`	`false) {};`
	`print(x);`

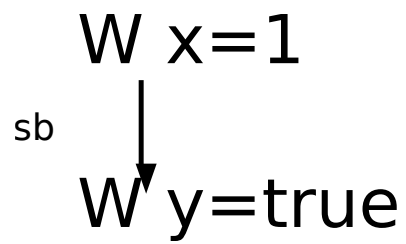
Good behaviour: Always prints 1

Message passing fix, C/C++11, rel/acq

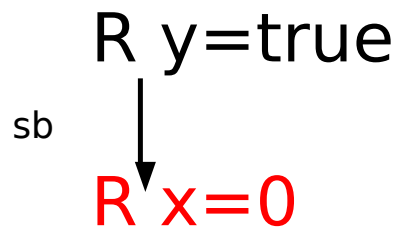
```
int x = 0;     atomic_bool y = false;
```

// sender thread	// receiver thread
`x = 1;`	`while (y.load(acquire)==`
`y.store(true,release);`	`false) {};`
	`print(x);`





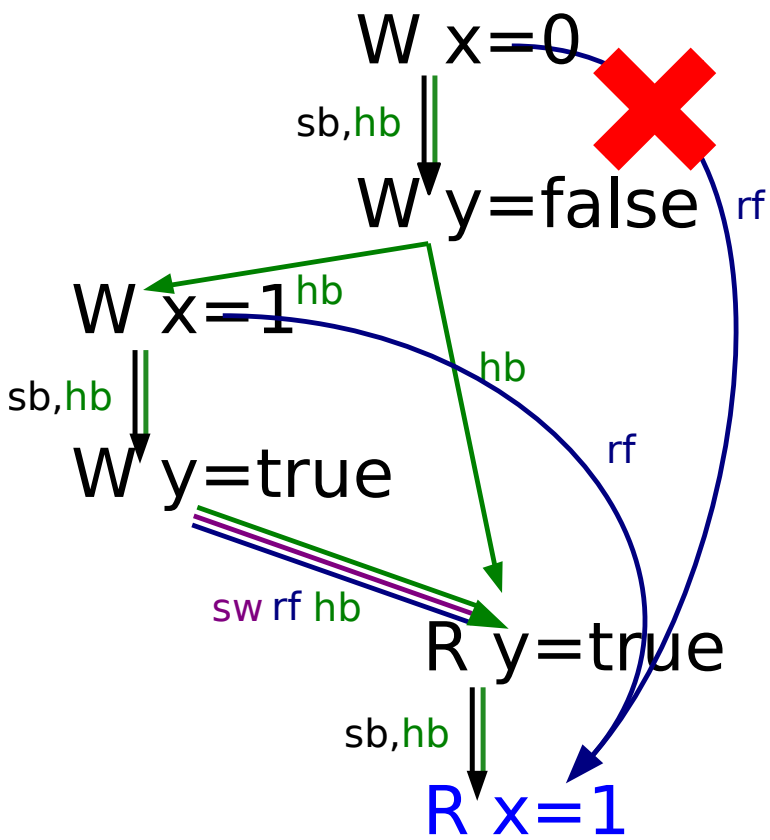
sb= sequenced-before



Message passing fix, C/C++11, rel/acq

```
int x = 0;     atomic_bool y = false;
```

// sender thread	// receiver thread
`x = 1;`	`while (y.load(acquire)==`
`y.store(true,release);`	`false) {};`
	`print(x);`



sb= sequenced-before

rf= reads-from

hb= happens-before

Basic Question

What *is* the concurrency semantics of C and C++?

[Mark Batty, Scott Owens, Susmit Sarkar, Tjark Weber, Sewell]

We've built a model...

...by a long process of

- formalising draft C++11 standard concurrency
- generating and checking test cases
- proving some facts
- talking with the C/C++ standards committees
- fixing the ISO C11 and C++11 standards
  (their text and our math now correspond)

Contrasts with Classic Concurrency and PL Semantics

no operational semantics. Instead, three-phase:

- simple (recursive-on-AST) calculation of sets of candidate executions;
- filter by axiomatic memory model;
- check for races

no interleaving. no memory. no global time.

hb reminiscent of true-concurrency causality relations? (but... hb not transitive, other machinery, per single conflict-free candidate)

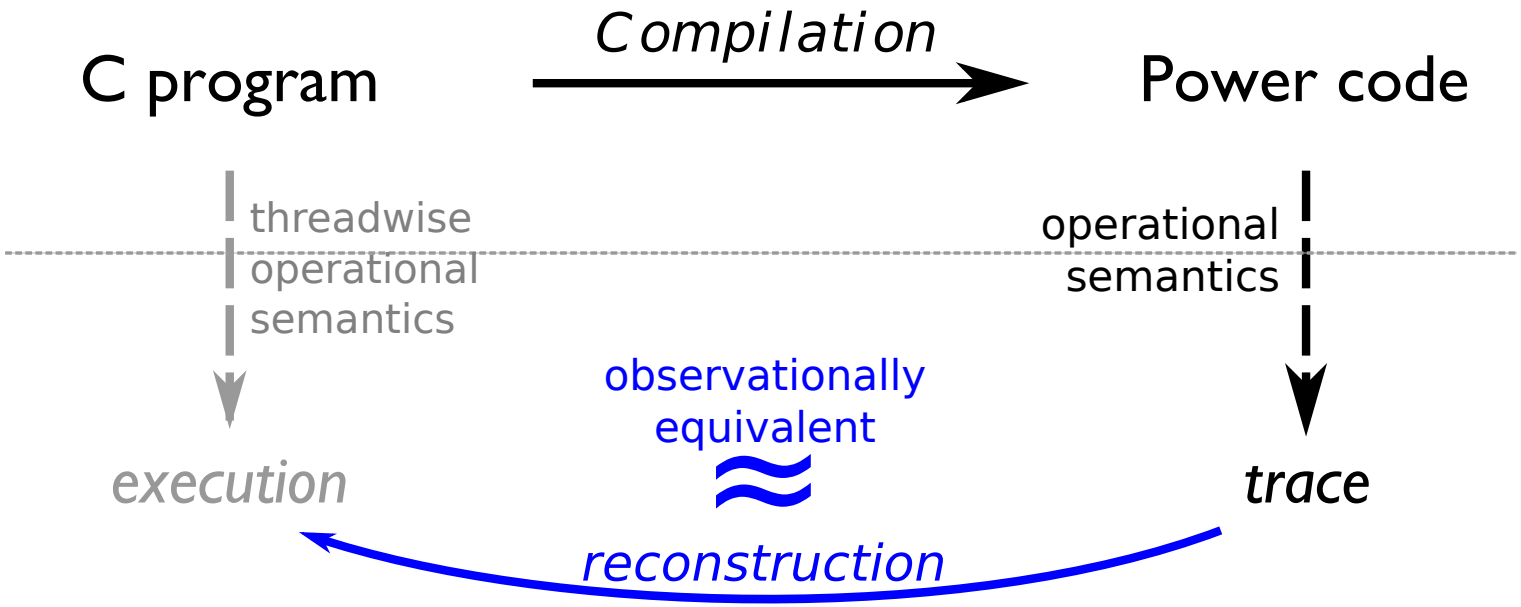
Compilation scheme

C11/C++11 operation	Power implementation
Non-atomic load	`ld`
Load relaxed	`ld`
Load consume	`ld` (with dependency preservation)
Load acquire	`ld; cmp; bc; isync`
Load seq. cst.	`sync; ld; cmp; bc; isync`
Non-atomic store	`st`
Store relaxed	`st`
Store release	`lwsync; st`
Store seq. cst.	`sync; st`

by Paul McKenney & Raul Silvera

actually, several schemes — and previous one was unsound, in some sense...

Theorem: that scheme is sound



For any Power abstract-machine trace of the compiled program there is a C/C++11 execution which is:

- equivalent to the trace (isomorphic reads-from)
- accepted by the original C/C++11 program

[Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, Sewell]

3

Proving Compiler Correctness in a Mechanized Logic

R. Milner and R. Weyhrauch

Computer Science Department
Stanford University

Abstract

We discuss the task of machine-checking the proof of a simple compiling algorithm. The proof-checking program is LCF, an implementation of a logic for computable functions due to Dana Scott, in which the abstract syntax and extensional semantics of programming languages can be naturally expressed. The source language in our example is a simple ALGOL-like language with assignments, conditionals, whiles and compound statements. The target language is an assembly language for a machine with a pushdown store. Algebraic methods are used to give structure to the proof, which is

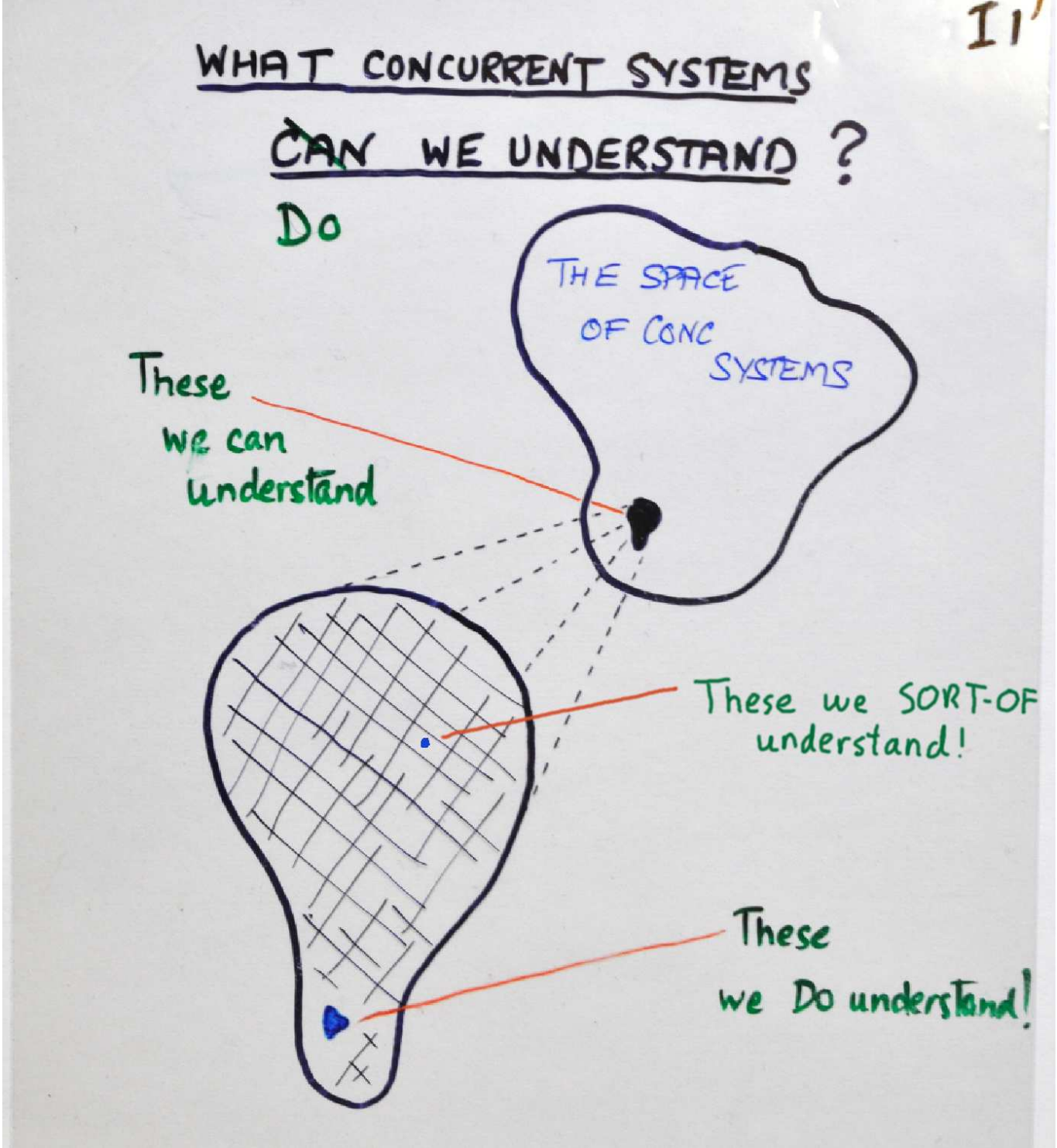
What about all the *rest* of a compiler?

CompCertTSO: a verified (in Coq) compiler targetting realistic x86 concurrency

building on Leroy's CompCert compiler for sequential code

[Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, Sewell]

Again, using the tools from Robin's workshop: labelled transition systems, top-level parallel composition, several kinds of simulation argument,... and a mechanised prover


WHAT CONCURRENT SYSTEMS
CAN WE UNDERSTAND ?
Do
THE SPACE
OF CONC
SYSTEMS
These
we can
understand
These we SORT-OF
understand!
These
we Do understand!

A MAJOR CONCERN OF COMPUTATION THEORY

Not so much

"How efficient is a system?"

But

"Does it work at all?"

ALSO:

- Is there a unifying mathematical framework for Distributed Computing?
- What does the mathematics MEAN? What is a BEHAVIOUR?
  (cf LOGIC, MODEL THEORY)
- Can engineers use the notations, as well as the theorists?

The *Foundations* of Computer Science?

The End

P.S. We're looking for postdocs (advert on the web). As Robin wrote in a different context, in 1998:

> *"A combined practico-theoretical group has formed around this work in Cambridge, and the work is ongoing"*.