Synchronising C/C++ and POWER

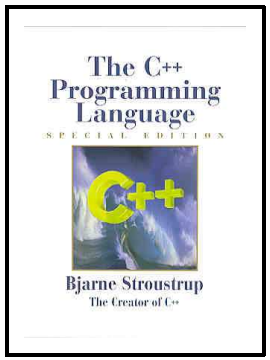
Susmit Sarkar[1] Kayvan Memarian[1] Scott Owens[1] Mark Batty[1]
Peter Sewell[1] Luc Maranget[2] Jade Alglave[3,4] Derek Williams[5]

[1]University of Cambridge [2]INRIA [3]Oxford University
[4]Queen Mary London [5]IBM Austin

June 2012

Relaxed Memory Concurrency

- Concurrency on modern hardware/compilers:
  Relaxed Memory, not Sequential Consistency (SC)



- Semantics of concurrent programming languages
- ISO C/C++: introduces a new concurrency model

- Hardware: very different concurrency models
  - Different between x86, Power, ARM
  - Different from C/C++

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
- 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) Load (non-atomic)	`st` `ld`

(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`

(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`

(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:`
...	...

(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)

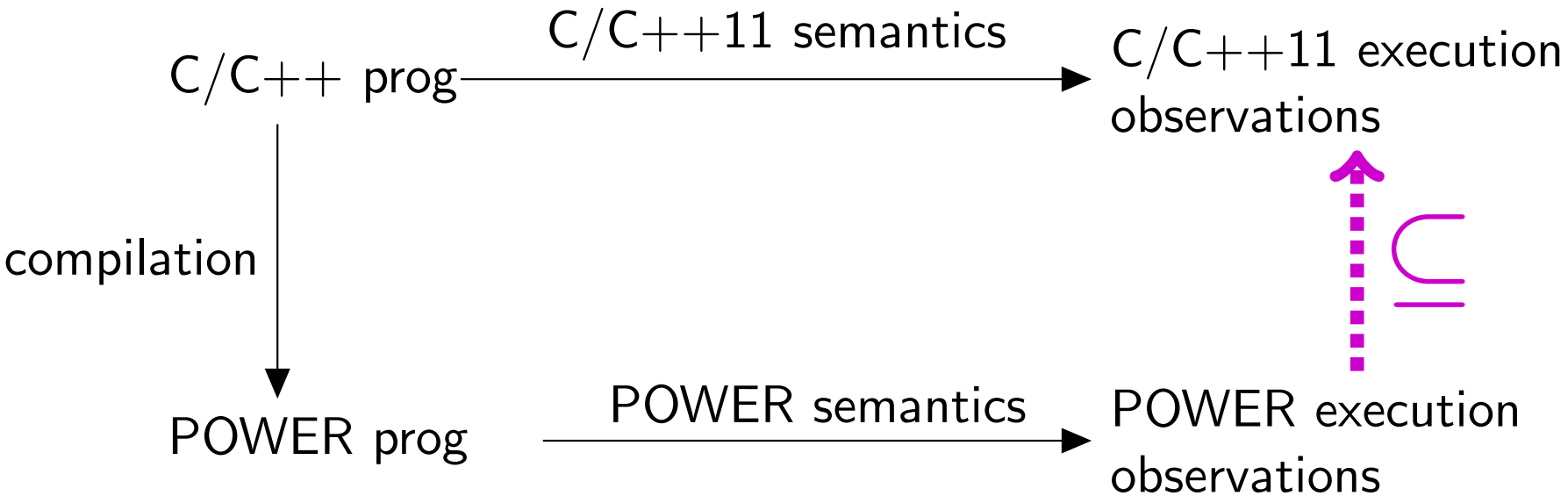
Implementing C/C++11 on POWER: Pointwise Mapping

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 ??

Context of This Paper

Before [POPL'12]: just loads and stores

- Power concurrency model (of loads and stores) [PLDI'11]
- C++11 concurrency model [POPL'11]
- Proof:
  - some concepts correspond (e.g. coherence $\rightarrow$ modification order)
  - others depend on key properties of abstract machine

This paper: also with synchronisation constructs

- Power: `load-reserve` and `store-conditional`
- C++11: locks, read-modify-writes, fences
- Proof:
  - extends smoothly (new cases to be checked)
  - points out interesting features of the models

Outline

Example: Message Passing

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

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

Finally: r = 0 ??	

- Forbidden on SC

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 ??	

- Forbidden on SC
- In C/C++11, this has undefined semantics
- Data race on d and f variables

Example (contd.): mark atomics

Mark atomic variables (accesses have memory order parameter)

Initially: 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

Mark atomic variables (accesses have memory order parameter)

Initially: 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

Example (contd.): release-acquire synchronization

Mark release stores and acquire loads

Initially: 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

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

Outline

What about Synchronising (Atomic) Operations?

- Synchronization operations, e.g. “atomic add”, “CAS”, . . .
- RISC-friendly alternative: Load-reserve/Store-conditional

What about Synchronising (Atomic) Operations?

- Synchronization operations, e.g. "atomic add", "CAS", ...
- RISC-friendly alternative: Load-reserve/Store-conditional
- Can be used to implement CAS, spinlocks, ...
- Universal (like CAS) [Herlihy'93], but no ABA problem

Atomic Addition

```
loop: lwarx r, d;
      add r,v,r;
      stwcx r, d;
      bne loop;
```

- Informally, stwcx succeeds only if no other write to the same address since last lwarx

What *is* no write since . . . ?

- In machine time?
  - Neither necessary, nor sufficient

What *is* no write since . . . ?

- In machine time?
  - Neither necessary, nor sufficient
- Microarchitecturally (simplified): if cache-line ownership not lost since last `lwarx`

Modeling "not lost since"

- Abstractly: ownership chain modeled by building up coherence order
- Coherence: order relating stores to the same location (eventually linear)

- A `stwcx` succeeds only if it is (becomes) coherence-next-to the write read from by `lwarx`
- . . . and no other write can later come in between

Modeling "not lost since"

- Abstractly: ownership chain modeled by building up coherence order
- Coherence: order relating stores to the same location (eventually linear)

- A `stwcx` succeeds only if it is (becomes) coherence-next-to the write read from by `lwarx`
- . . . and no other write can later come in between

- Isolate key concept: write reaching coherence point —
  - coherence is linear below this write, and no new edges will be added below

Load-reserve/store-conditional and ordering

- Same-thread load-reserve/store-conditionals ordered by program order

  If all memory accesses are atomic sequences

  Then: only SC behaviour

- But . . . normal loads/stores (to different addresses) not ordered

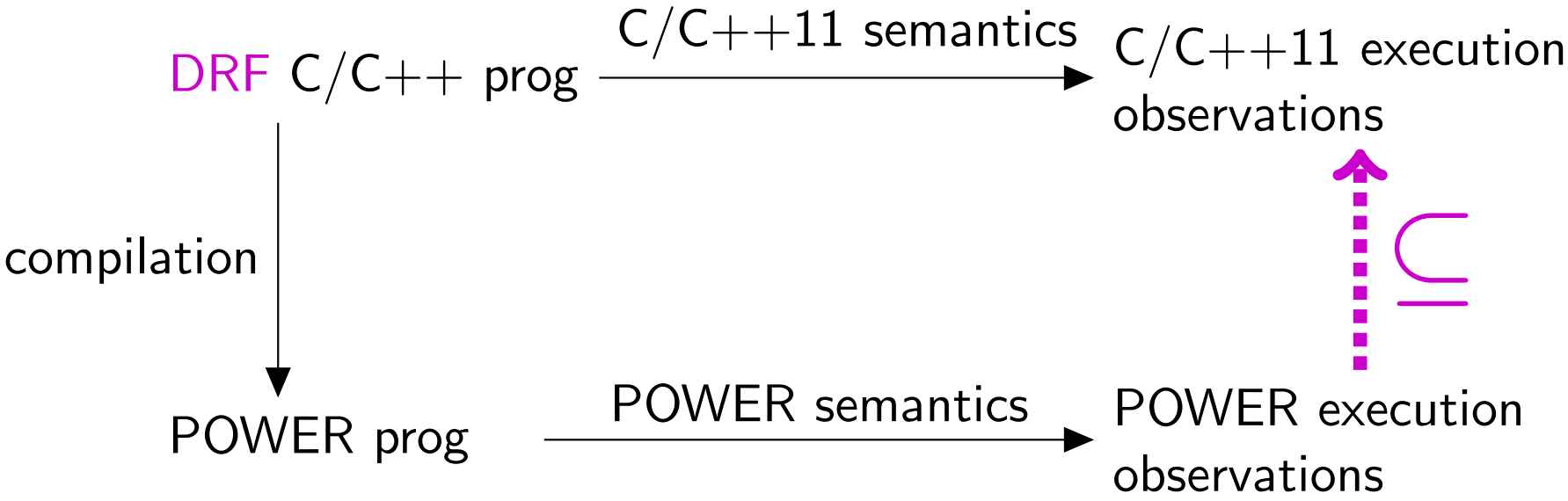
  Confusion here led to Linux bug

  . . . bad barrier placement in atomic-add-return

Outline

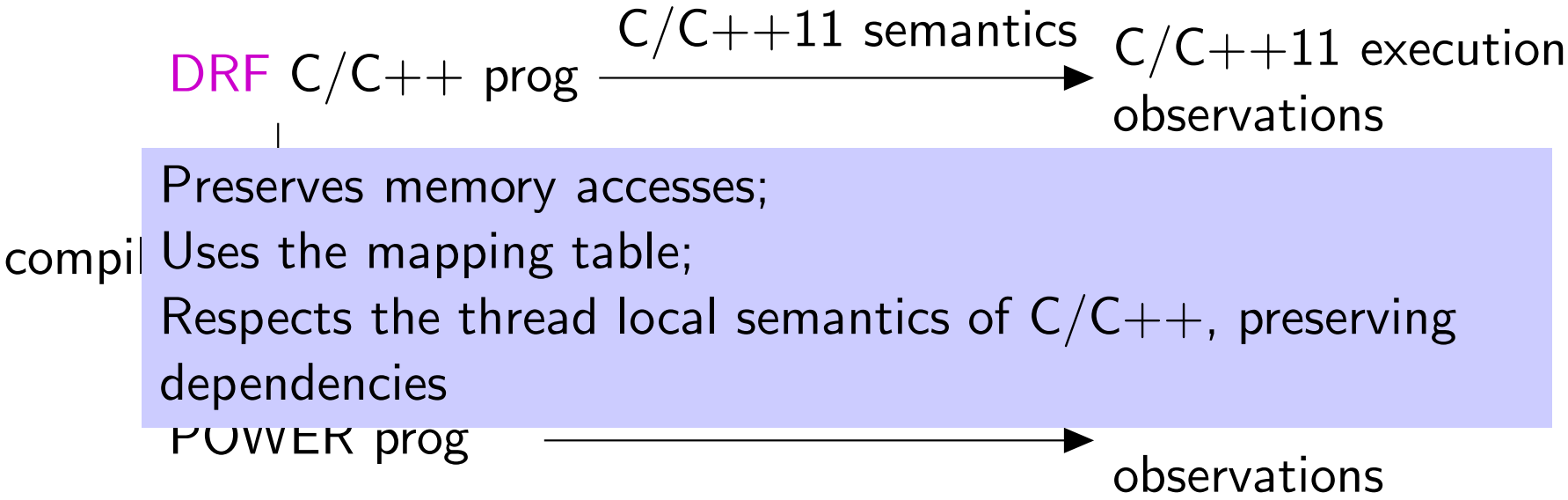
Proof outline

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



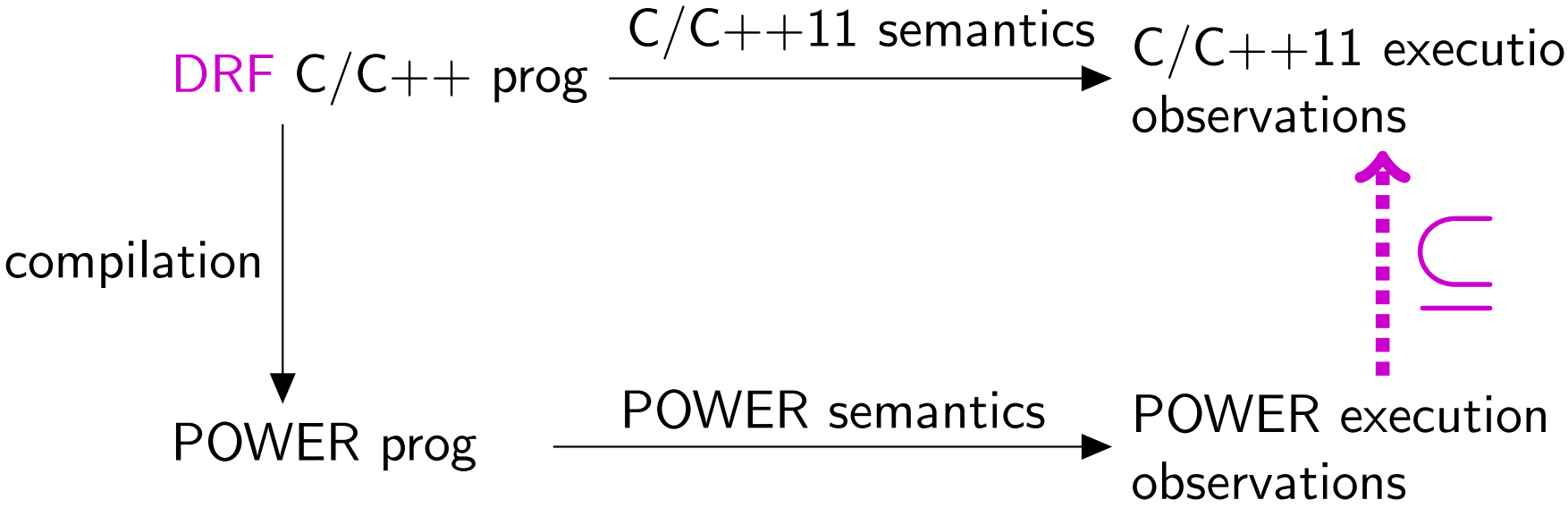
Proof outline

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

Also in the paper

- A formal model of load-reserve/store-conditional (in Lem)
- An executable model with exploration tool (ppcmem)
- Simplifications to the C/C++11 lock model
- Models "tight" against each other: relaxing the Power model would make C/C++11 unimplementable

Conclusion

Reasoning about industrial-strength concurrency

- Correct compilation of C/C++ concurrency primitives on Power
  - Formal relaxed-memory semantics of load-reserve/store-conditional
  - Allow proof of SC via atomic RMW sequences
  - Technical simplifications to the C/C++ lock model
- Confidence in both models
- Compiler implementation relevance
- Reasoning about machine code at C/C++ level

Thank You!

More details at:

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

Store-conditional speculation?

- Power allows stores to forward value to same thread speculatively
- Can (and should) stwcx be allowed to be speculated (even before the lwarx) ?

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

`d = 1; # d.store(1,rlx)`
`lwsync; # f.store(1,rel)`
`f = 1;`
	`loop: lwarx f, rl; # CAS (f,1,2)`
	`cmp rl 1; bne exit;`
	`stwcx f 2; bne loop;exit:`
	`ld` r_1 `f; # `r_1` = f.load(con)`
	`xor` r_2`,` r_1`,`r_1`; # `$r_2 = r_1 \oplus r_1$
	`ld [d +` r_2`] r; # r = d[`r_2`]`

Finally: $r = 0$??	

Store-conditional speculation?

- Can (and should) stwcx be allowed to be speculated (even before the lwarx) ?

Initially: $d = 0 \quad f = 0;$	
Thread 0	Thread 1

`d = 1; # d.store(1,rlx)`
`lwsync; # f.store(1,rel)`
`f = 1;`
	`loop: lwarx f, rl; # CAS (f,1,2)`
	`cmp rl 1; bne exit;`
	`stwcx f 2; bne loop;exit:`
	`ld` r_1 `f; # `r_1` = f.load(con)`
	`xor` r_2`,` r_1`,`r_1`; # `$r_2 = r_1 \oplus r_1$
	`ld [d +` r_2`] r; # r = d[`r_2`]`

Finally: $r = 0$??	

- C/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

60 second pitch

Hi, I am Susmit Sarkar, and I am going to be speaking about shared-memory concurrency not as we would like it to be, but as it actually is in the real world, on mainstream hardware such as PowerPC or ARM and on software such as the new C and C++ concurrency model. These two models are quite strange, and quite different from each other so it is a real question whether you can even compile from one to the other. Yes you can, and we prove this. This explains how these very different models really work. Come to Room B, just after lunch