Load-reserve / Store-conditional on POWER and ARM

Peter Sewell (slides from Susmit Sarkar)

[1]University of Cambridge

June 2012

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	`sync; 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	`sync; ld; cmp; bc; isync`
Fence acquire	`lwsync`
Fence release	`lwsync`
Fence seq-cst	`sync`

(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	`sync; ld; cmp; bc; isync`
Fence acquire	`lwsync`
Fence release	`lwsync`
Fence seq-cst	`sync`
CAS relaxed	`_loop: lwarx; cmp; bc _exit;` `stwcx.; bc _loop; _exit:`
CAS seq-cst	`sync; _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	`sync; ld; cmp; bc; isync`
Fence acquire	`lwsync`
Fence release	`lwsync`
Fence seq-cst	`sync`
CAS relaxed	`_loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; _exit:`
CAS seq-cst	`sync; _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; sync; st`
Load relaxed	`ld`
Load consume	`ld` (and preserve dependency)
Load acquire	`ld; cmp; bc; isync`
Load seq-cst	`sync; ld; cmp; bc; isync`
Fence acquire	`lwsync`
Fence release	`lwsync`
Fence seq-cst	`sync`
CAS relaxed	Answer: No!
CAS seq-cst	`sync; _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	`sync; st`
Load relaxed	`ld`
Load consume	`ld (and preserve dependency)`
Load acquire	
Load seq-cst	
Fence acquire	`lwsync`
Fence release	`lwsync`
Fence seq-cst	`sync`
CAS relaxed	
CAS seq-cst	`sync; _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	`sync; st`
Load relaxed	`ld`
Load consume	`ld (and preserve dependency)`
L	
L	
Fence acquire	`lwsync`
Fence release	`lwsync`
Fence seq-cst	`sync`
CAS relaxed	
CAS seq-cst	`sync; _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	`sync; st`	`sync; st; sync;`
Load relaxed	`ld`	
Load consume	`ld` (and preserve dependency)	
Load acquire	`ld; cmp; bc; isync`	
Load seq-cst	`sync; ld; cmp; bc; isync`	`ld; sync`
Fence acquire	`lwsync`	
Fence release	`lwsync`	
Fence seq-cst	`sync`	
CAS relaxed	`_loop: lwarx; cmp; bc _exit;` `stwcx.; bc _loop; _exit:`	
CAS seq-cst	`sync; _loop: lwarx; cmp; bc _exit;` `stwcx.; bc _loop; isync; _exit:`	
...	...	

All compilers must agree for separate compilation

Machine Synchronisation Operations

- x86: atomic synchronization operations, e.g. “atomic add”, “CAS”, . . .
- RISC-friendly alternative: Load-reserve/Store-conditional (aka LL/SC, larx/stcx and lwarx/stwcx, LDREX/STREX)

Machine Synchronisation Operations

- x86: atomic synchronization operations, e.g. “atomic add”, “CAS”, . . .
- RISC-friendly alternative: Load-reserve/Store-conditional (aka LL/SC, larx/stcx and lwarx/stwcx, LDREX/STREX)
- Can be used to implement CAS, atomic add, 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, setting a flag iff it succeeds

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`

(but we don't want to model the microarchitecture...)

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 (or at least, if it can become) 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 (or at least, if it can become) coherence-next-to the write read from by `lwarx`
- . . . and no other write can later come in between

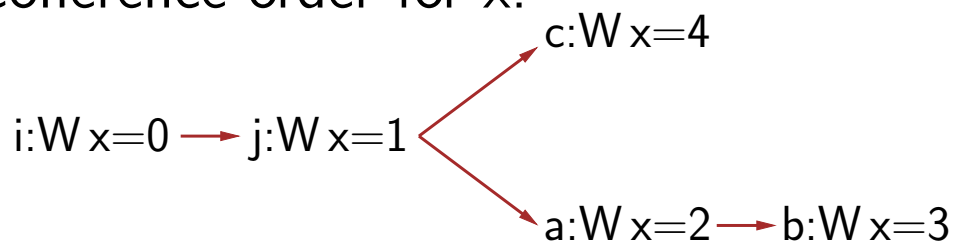
- Isolate key concept: write reaching coherence point —
  - coherence is linear below this write, and no new edges will be added below

Coherence points and a successful stwcx

Atomic Addition

`loop: lwarx r, x;`
`add r,3,r;`
`stwcx r, x;`
`bne loop;`

Coherence order for x:



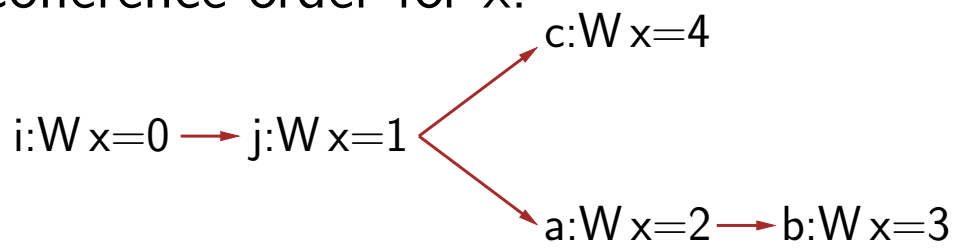
Suppose `lwarx` reads from the "a:W x:2"

Coherence points and a successful stwcx

Atomic Addition

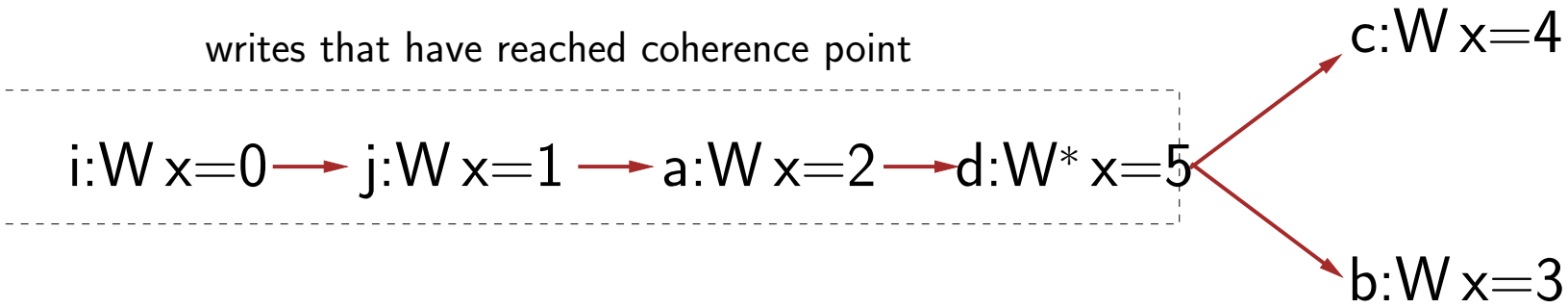
`loop: lwarx r, x;`
`add r,3,r;`
`stwcx r, x;`
`bne loop;`

Coherence order for x:



Suppose `lwarx` reads from the "a:W x:2"

`stwcx` can succeed if this becomes possible:



Warning: `stwcx` can fail spuriously

Load-reserve/store-conditional and ordering

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

  If all memory accesses are l-r/s-c sequences

  Then: only SC behaviour

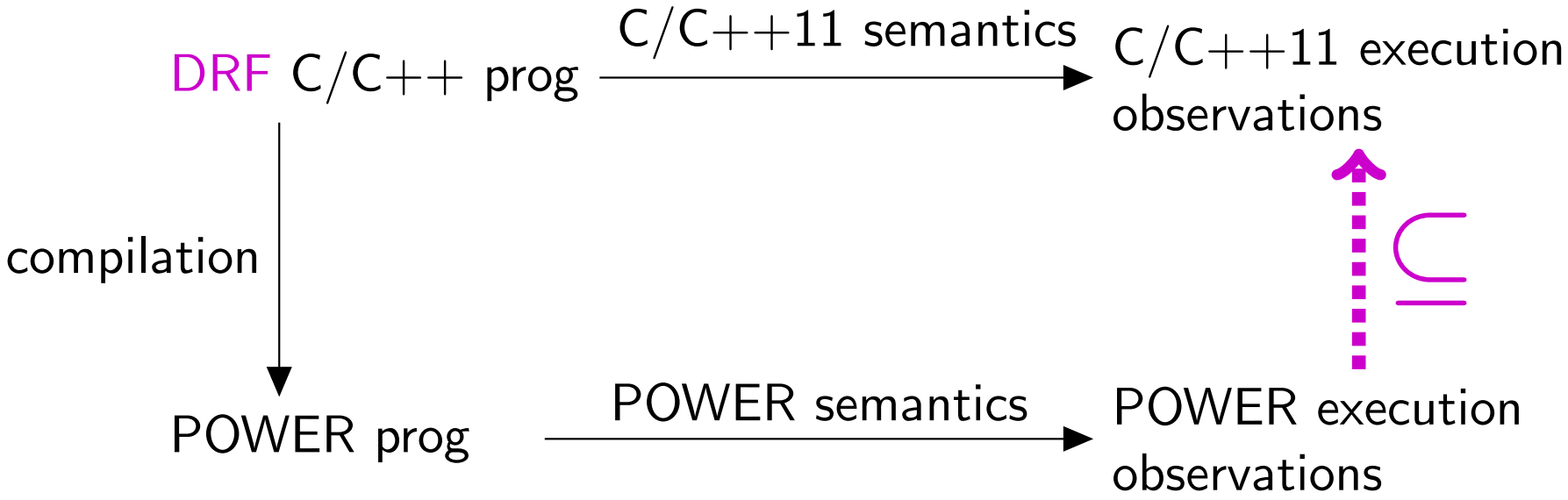
- But . . . normal loads/stores (to different addresses) not ordered; the l-r/s-c do not act as a barrier

  Confusion here led to Linux bug

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

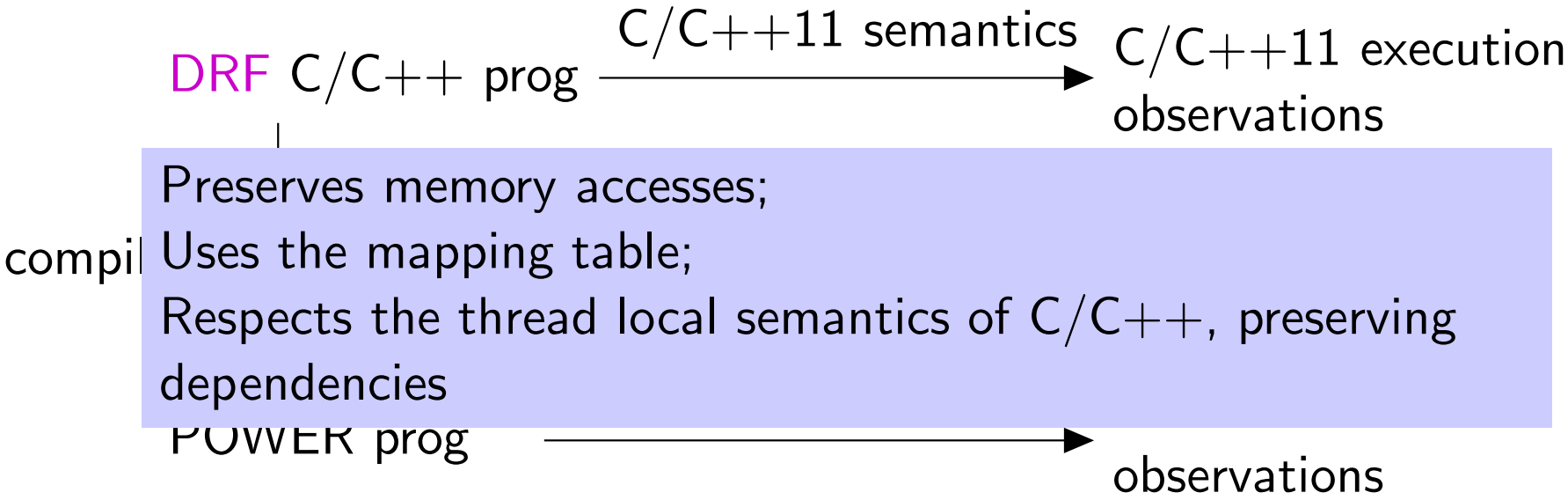
Correctness of the Mapping

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



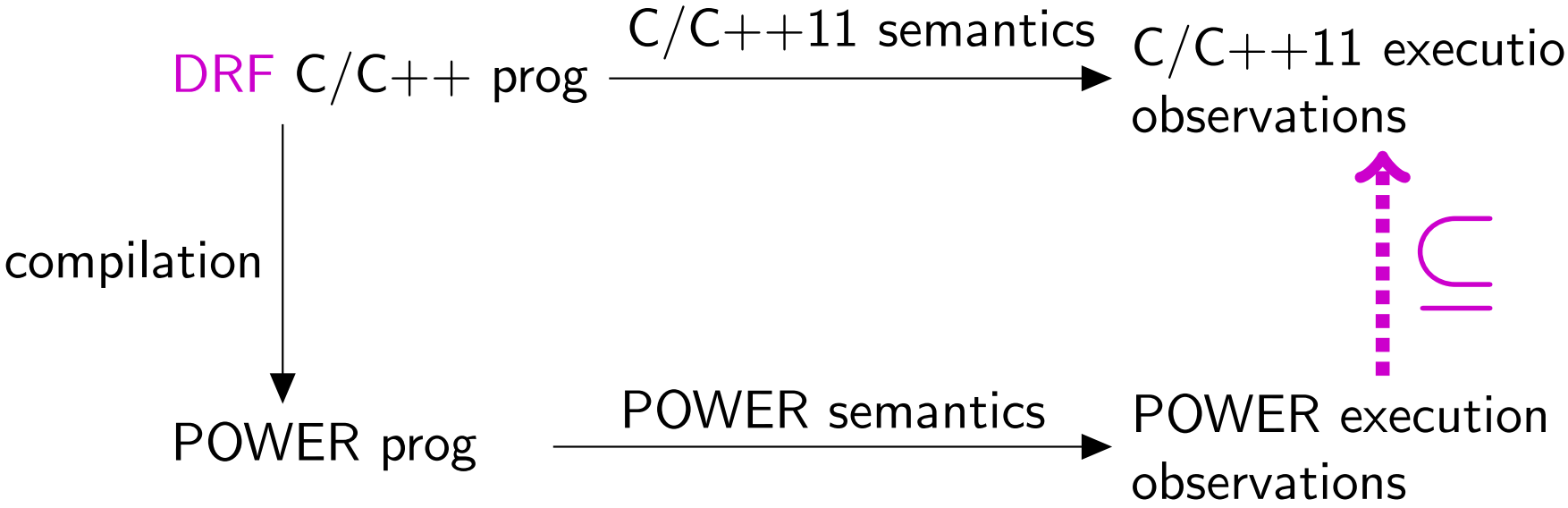
Correctness of the Mapping

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



Correctness of the Mapping

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 details...

see *Synchronising C/C++ and POWER*, Sarkar et al., PLDI 2012

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

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