Inventing Abstractions

An Academic Perspective on Industrial Memory Models

Susmit Sarkar

with: Scott Owens, Kayvan Memarian, Mark Batty, Peter Sewell,
Magnus Myreen, Jade Alglave, Luc Maranget, Francesco Zappa Nardelli,
Derek Williams, Sela Mador-Haim, Rajeev Alur, Milo Martin

REORDER, July 2012

Once Upon a Time . . .

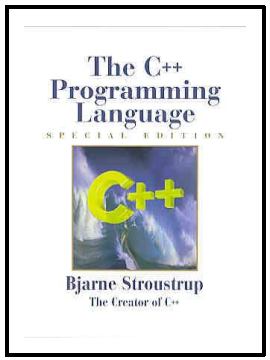
BURROUGHS D825, 1962

‘‘Outstanding features include truly modular hardware with parallel processing throughout’’

‘‘FUTURE PLANS
The complement of compiling languages is to be expanded.’’

Today: Relaxed Memory Concurrency

- Concurrency on modern (since IBM370, $\sim$ 1972) 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++

Example: Message Passing

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

`data = 1;`
`flag = 1;`
	`while (flag == 0)`
	`  {};`
	`r = data;`

Finally: r = 0 ??	

- Forbidden on SC

Example: Message Passing

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

`data = 1;`
`flag = 1;`
	`while (flag == 0)`
	`  {};`
	`r = data;`

Finally: r = 0 ??	

- Not observed (and explicitly forbidden) on x86
- Observed on POWER ($\sim$ 1e6 in 2e9 on a POWER7) and ARM ($\sim$ 4e6 in 3e9 on a Tegra2)

Message Passing: What's going on?

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

`data = 1;`
`flag = 1;`
	`while (flag == 0)`
	`  {};`
	`r = data;`

Finally: r = 0 ??	

Hardware optimizations:

- Writes propagated out of order
- Reads can be done out of order/speculatively

Programming Message Passing

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

`data = 1;`
`lwsync;`
`flag = 1;`
	`while (flag == 0)`
	`{}; isync;`
	`r = data;`

Finally: r = 0 ??	

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

(Other programming methods possible)

Message Passing in high-level languages

- Have to run on hardware
- But compiler can do optimizations as well

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

`data = 1;`
`flag = 1;`
	r_0 `= data;`
	`while (flag == 0)`
	`{};`
	`r = data;`

Finally: r = 0 ??	

- Forbidden on SC (regardless of other reads)

Message Passing in high-level languages

- Have to run on hardware
- But compiler can do optimizations as well

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

`data = 1;`
`flag = 1;`
	r_0 `= data;`
	`while (flag == 0)`
	`{};`
	`r =` r_0`;`

Finally: r = 0 ??	

- Forbidden on SC (regardless of other reads)
- Suppose compiler does Common Subexpression Elimination
- Programmer has to mark operations specially to compiler

Message Passing in C/C++11: release-acquire

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

Programming: the general case

Questions, questions, . . .

- Can we remove a barrier in the spinlock implementation? [Linux, 1999]
- Can we implement C/C++11 correctly? [ISO C/C++ committee, 2011]
- Can we regain SC easily? [C, C++, Java]
- Is an optimization legal?

Programmer model: How do we find out?

Answer I: Read the fantastic manuals!

- Big books, in prose
  - Intel64 and IA-32 Architectures Software Developer's Manual: 5 vol, about 3000 pages
  - ARM Architecture Reference Manual v7: about 2100 pages
  - ISO/IEC 14882:2011 C++ standard: about 1400 pages

- Necessarily imprecise, Leaves things out, and sometimes, Just Wrong!

  "all that horrible horribly incomprehensible and confusing [...] text that no-one can parse or reason with — not even the people who wrote it"

  — Anonymous Processor Architect, 2011

Programmer model: How do we find out?

Answer II: Test actual implementations

- Short litmus tests
- Run lots of times, with randomisation (some results occur once in 1e9!)
- Effective in finding corner cases
- Essential: automated oracle (formal modelling tools)
- Found bugs in deployed and pre-silicon hardware
- Industrial uptake of our tools

Programmer model: How do we find out?

Answer III: Talk to designers

POWER architect

Lead Architect

C++/C standards committee, concurrency group

Concurrent Programmers

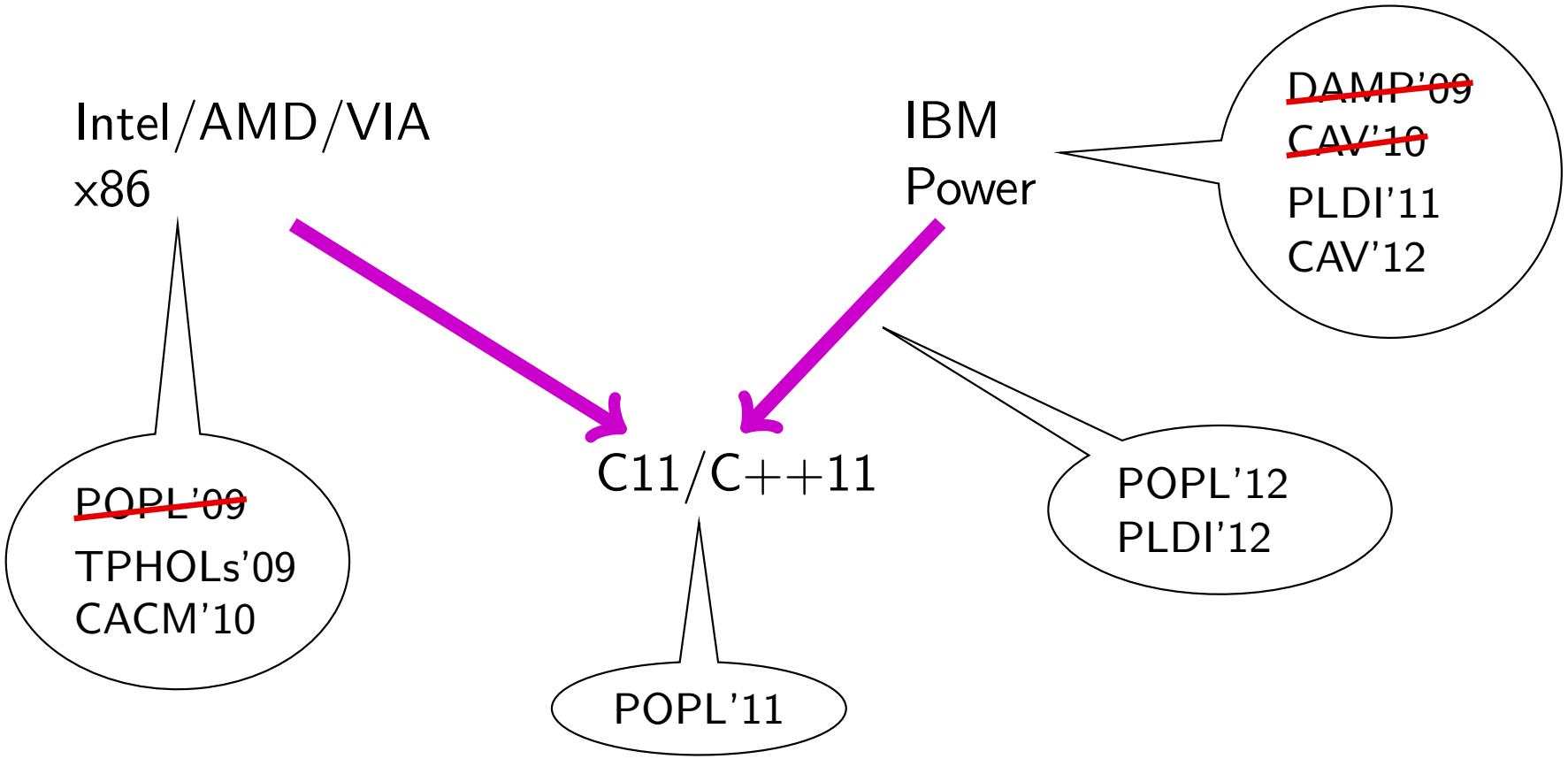
Focus on programmer-observable behaviour

~~Discovering~~ Inventing the programmer model

In reality, do all three

- Invent abstractions in collaboration
  - Have to be loose specifications
- Develop formal model, test its consequences, iterate!
- Machine assistance critical (proof assistants, interactive theorem provers, axiom system explorers, SMT solvers)

Relaxed Memory Models



A few years ago . . . (late 2008)

Q: What is the POWER model, anyway?
A1: Let's just read the manuals [DAMP'09]

- An axiomatic model
- Took great care with parallelizable instruction semantics
- Axioms relating "view orders" of every thread
- Choices about barrier axioms

> *We are developing a tool for exploring the consequences of our semantics. [. . .] It is work in progress: of the tests in the previous section, currently [2] can be executed. [. . .] Further engineering is required to support the other tests.*

Broken for many examples

Some time later. . .

Q: What is the POWER model, anyway?
A2: Let's read the manuals (more seriously). . .

> *A load by a processor (P1) is performed with respect to any processor (P2) when the value to be returned by the load can no longer be changed by a store by P2.*

Used to define the semantics of dependencies and barriers.

This style of definition goes back to the work of Dubois et al. (1986).

Some time later. . .

Q: What is the POWER model, anyway?
A2: Let's read the manuals (more seriously). . .

> *A load by a processor (P1) is performed with respect to any processor (P2) when the value to be returned by the load can no longer be changed by a store by P2.*

Used to define the semantics of dependencies and barriers.

This style of definition goes back to the work of Dubois et al. (1986).
But it's *subjunctive*: it refers to a hypothetical store by P2.

Formalizing the manuals

- Make several candidate formalizations of "performed"
- Email a bunch of people who Might Know ...
- ... long silence

Test the machines

Q: What is the POWER model, anyway?
A3: Let's run a few litmus tests...

- Found some surprising results
- Got the attention of Derek Williams (IBM)
- Memory Models: Industry knows this is complex to get right

Testing-based model generation

- CAV'10: An axiomatic model of POWER
- Matches test results for many tests
- Simple axiomatic model (global-happens-before or global-time), but non-multi-copy-atomic
- Sound and precise for tests with dependencies, sync, isync
- Question: How to incorporate lwsync?

From Architecture to Microarchitecture (and back again)

- Axiomatic Models: Hard to see how they work
- . . . or predict the effect of changing the axioms

- Microarchitecture: Lots of detail
- Easier (but still hard) to predict the consequences of changing it

- PLDI'11: Abstract microarchitectural model (and test it extensively)
- Space for Interactive Model Checking (!)

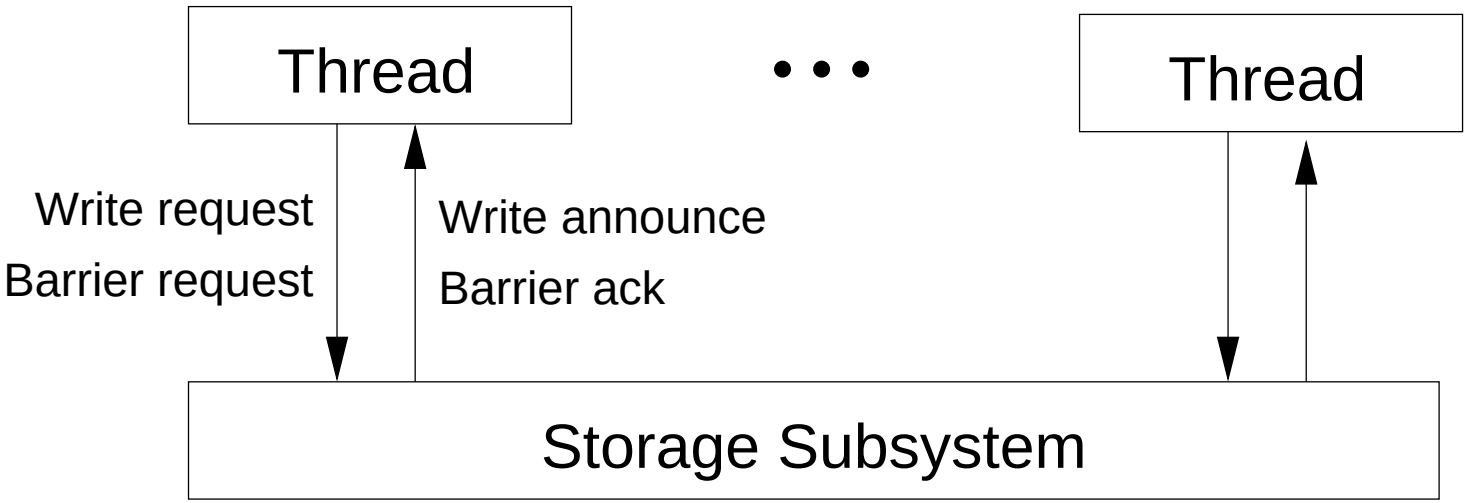
- CAV'12: Proven equivalent axiomatic model

Architectural Models

- Serves as a basis for communication
  - Now mostly communicate with Derek Williams using abstract operational model
- Must describe a range of implementations (incl. future)
- Must not (even seemingly) overspecify hardware
- Must make programming, and reasoning about programs, possible

The model structure

Overall structure:



- Some aspects are thread-only, some storage-only, some both
- Threads and Storage Subsystem: Abstract state machines

  Speculative execution in Threads;
  Topology-independent Storage Subsystem
- Formally: transitions, guarded by preconditions, change state, and synchronize with each other

Reasoning about C/C++11 implementation on POWER

C/C++11 Recap:

- Axiomatic model defining which executions legal
- "Happens-before" relation constrains which writes legal to read-from
- Complex definition: not transitive, but parts transitive
- Consistency required with a "modification order", a "SC order"

C/C++11 on Power: PropBefore Lemma

- Key step in reasoning: What can we say about "Happens-Before" in an abstract machine characterization?
- Propagates-before Lemma: correspondence to facts about propagation of underlying writes to different threads

C/C++11 on Power: PropBefore Lemma

- Key step in reasoning: What can we say about "Happens-Before" in an abstract machine characterization?
- Propagates-before Lemma: correspondence to facts about propagation of underlying writes to different threads

- Delicate balance between C/C++ and POWER models
- Base Case: release-acquire $\Longrightarrow$ lwsync and control-isync
- Transitive reasoning $\Longrightarrow$ cumulative barriers
- CAS in release sequences $\Longrightarrow$ restrict stwcx forwarding

Conclusion

Reasoning about industrial-strength concurrency

- A precise, formal, abstract model of x86-TSO
- A precise, formal, abstract model of Power
- Correct compilation of C/C++ concurrency primitives on Power
- Invent abstractions by testing and discussion
- Metatheory builds confidence in the models
- Relevance to compilers, architectural design
- Enables reasoning about machine code, by itself and at C/C++ level

Thank You!

More details; formal models, papers, the ppcmem tool, examples, etc. at:

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