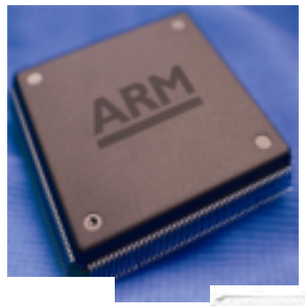
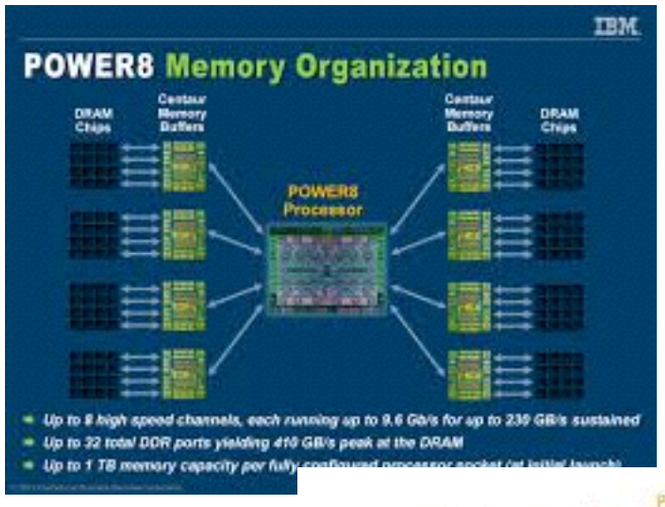
Rigorous Architectural Modelling for Production Multiprocessors

Shaked Flur, Kathryn E. Gray, Gabriel Kerneis[*], Luc Maranget [**],
Dominic Mulligan, Christopher Pulte, Susmit Sarkar [***], Peter Sewell

University of Cambridge
[*] Google, [**] Inria, [***] University of St Andrews

Multicore Machines





But everyone uses locks?

Lock-free data structures
Remove unnecessary locks speeds performance

So must determine when a lock is needed

And locks themselves must be implemented properly

Not Sequentially Consistent

X |-> 0 Y |-> 0

Thread1

```
MOV R0, #1
STR R0, X
DMB
MOV R1, #1
STR R1, y
```

Thread2

```
LDR R0, Y
LDR R1, X
```

Not Sequentially Consistent

X |-> 0 Y |-> 0

Thread1

```
MOV R0, #1
STR R0, X
DMB
MOV R1, #1
STR R1, y
```

Thread2

```
LDR R0, Y
LDR R1, X
```

expect: R0 = 1 & R1 = 1
R0 = 0 & R1 = 0
R0 = 0 & R1 = 1

Not Sequentially Consistent

X |-> 0 Y |-> 0

Thread1

MOV R0, #1
STR R0, X
DMB
MOV R1, #1
STR R1, y

Thread2

LDR R0, Y
LDR R1, X

expect: R0 = 1 & R1 = 1
R0 = 0 & R1 = 0
R0 = 0 & R1 = 1

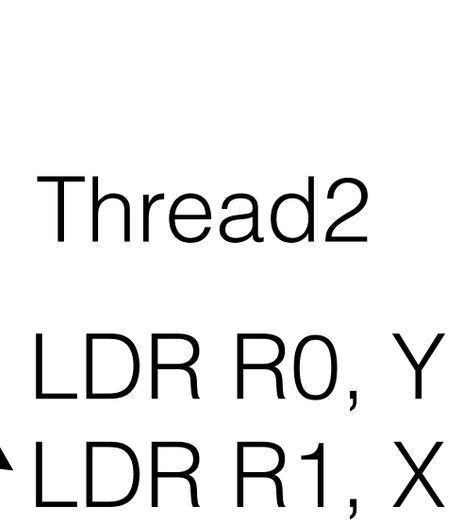
Not Sequentially Consistent

X |-> 0 Y |-> 0

Thread1

MOV R0, #1
STR R0, X
DMB
MOV R1, #1
STR R1, y

write X |-> 1



Thread2

LDR R0, Y
LDR R1, X

expect: R0 = 1 & R1 = 1
R0 = 0 & R1 = 0
R0 = 0 & R1 = 1

Not Sequentially Consistent

X |-> 0 Y |-> 0

Thread1

MOV R0, #1
STR R0, X
DMB
MOV R1, #1
STR R1, y

write X |-> 1

write Y |-> 1

Thread2

LDR R0, Y
LDR R1, X

expect: R0 = 1 & R1 = 1
R0 = 0 & R1 = 0
R0 = 0 & R1 = 1

Not Sequentially Consistent

X |-> 0 Y |-> 0

Thread1

MOV R0, #1
STR R0, X
DMB
MOV R1, #1
STR R1, y

write X |-> 1

write Y |-> 1

Thread2

LDR R0, Y
LDR R1, X

expect: R0 = 1 & R1 = 1
R0 = 0 & R1 = 0
R0 = 0 & R1 = 1

Not Sequentially Consistent

X |-> 0 Y |-> 0

Thread1

MOV R0, #1
STR R0, X
DMB
MOV R1, #1
STR R1, y

write X |-> 1

write Y |-> 1

Thread2

LDR R0, Y
LDR R1, X

possible: R0 = 1 & R1 = 0

Rigorous model vs Common Practice Concurrency

- Precise
- Comprehensible
- Testable
- Suitable for verification
- Encoding intuition and knowledge from industry engineers

- In imprecise prose
- Contradictory
- In engineer's heads
- Explained by litmus tests

Rigorous model vs Common Practice Instruction set architecture (ISA)

- Precise
- Comprehensible
- Testable
- Connectable to concurrency models

- Prose (X86)
- Pseudo code and prose (Power, ARM)
- Spaghetti specifications

Bringing rigour to Multiprocessors

- Executable formal model of concurrent behaviour
- Executable formal model of instruction set architecture
- Litmus tests for concurrent interactions
  - Exhaustively searchable as well as executable
- Single instruction tests for confidence building

ISA + Concurrency Challenges

- No single program point
- No per-thread register state
- Register self-reads
- Dependencies on sub-register granularity
- Reading from uncommitted instructions
- Non-atomic intra-instruction semantics on register reads
- Undefined values

No Per-thread register state

MP+sync+rs POWER

Thread 0		Thread 1	
stw r7,0(r1)	# x=1	lwz r5,0(r2)	# r5=y
sync	# sync	mr r6,r5	# r6=r5
stw r8,0(r2)	# y=1	lwz r5,0(r1)	# r5=x
Initial state: 0:r1=x, 0:r2=y, 0:r7=1, 0:r8=1, 1:r1=x, 1:r2=y, x=0			
Allowed: 1:r6=1, 1:r5=0			

Sub-register dependencies

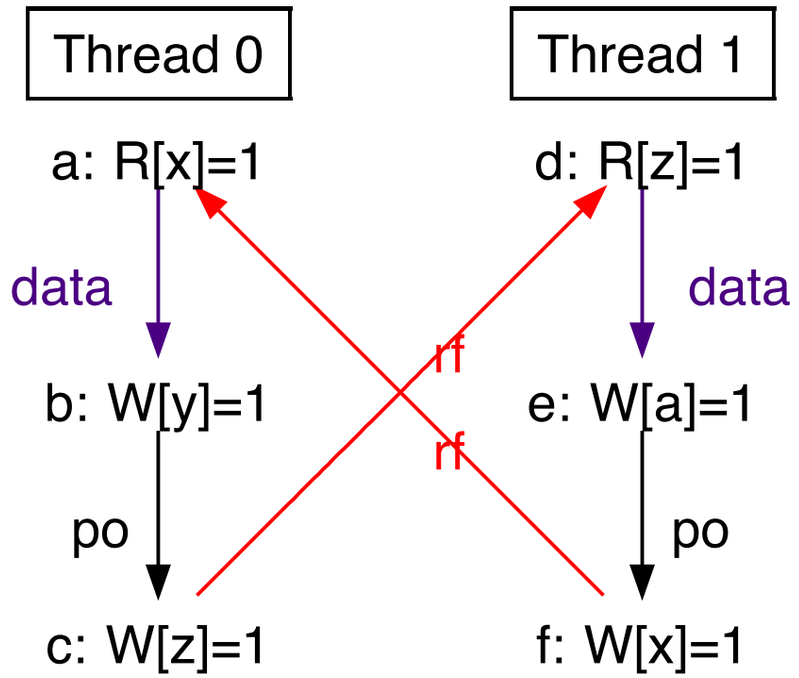
MP+sync+addr-cr POWER

Thread 0		Thread 1	
stw r7,0(r1)	# x=1	lwz r5,0(r2)	# r5 = y
sync	# sync	mtocrf cr3,r5	# cr3 = 4 bits of r5
stw r8,0(r2)	# y=1	mtocrf r6,cr4	# set 4 bits of r6 = cr4
		xor r7,r6,r6	# r7 = r6 xor r6
		lwzx r8,r1,r7	# r8 = *(&x + r7)
Initial state: 0:r1=x, 0:r2=y, 0:r7=1, 0:r8=1, 1:r1=x, 1:r2=y, x=0			
Allowed: 1:r6=1, 1:r5=0			

Non-atomic intra-instruction semantics on register reads

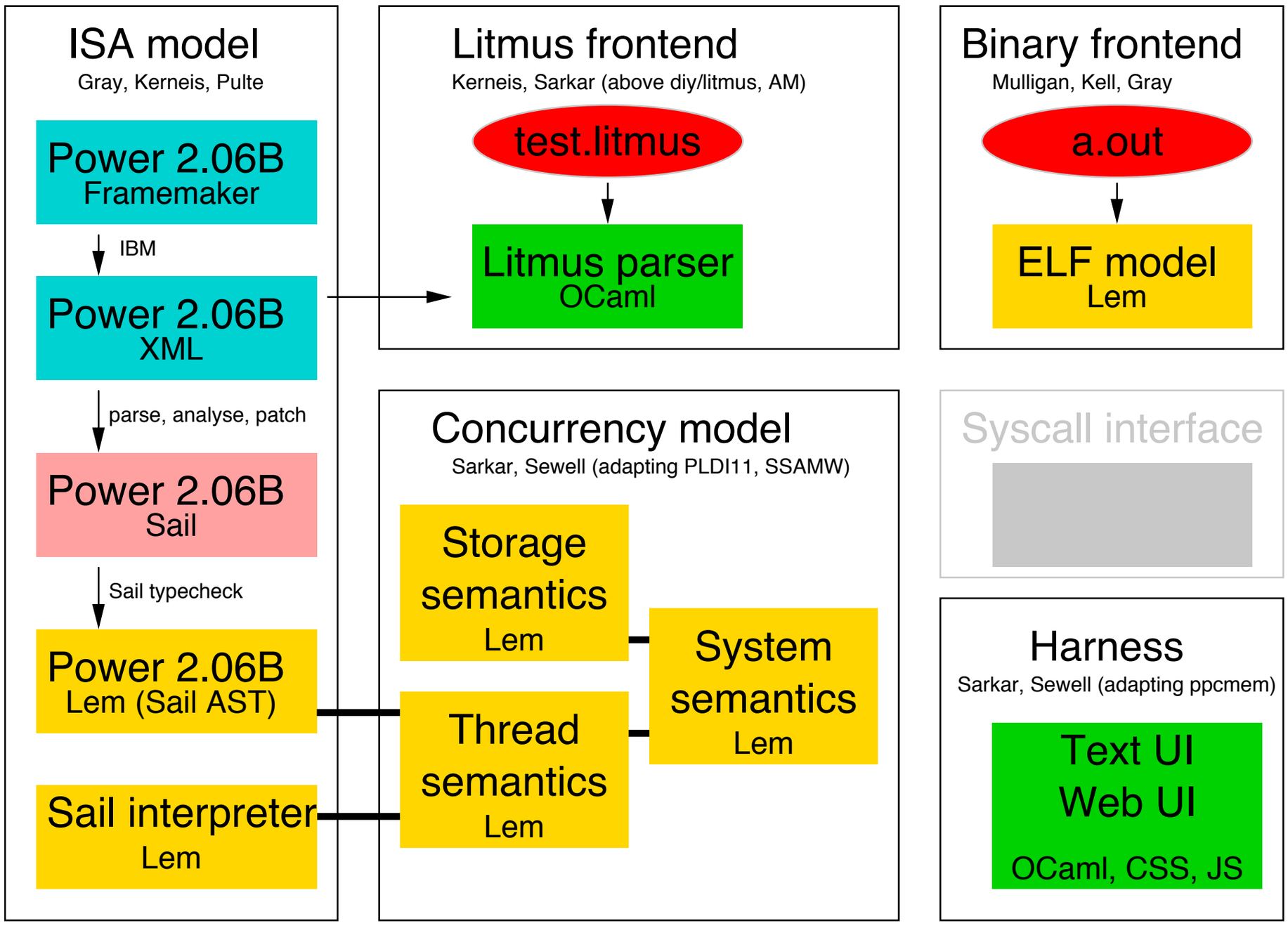
LB+datas+WW

Thread 0	Thread 1
a: r1=x	d: r2=z
b: y=r1	e: a=r2
c: z=1	f: x=1
Initial state: x=0 ∧ z=0	
Allowed: r1=1 ∧ r2=1	



Test LB+datas+WW: Allowed

PPCMEM2: a case study



Sail: for specifying concurrent ISAs

- C-like/ISA Pseudo-code like imperative language with
  - Built-in understanding of registers and memory operation
  - Pattern matching
- Polymorphic type system with
  - Inference
  - Dependent interval types
  - Effect tracking

Simple Instruction

Store Word with Update ***D-form***

stwu RS,D(RA)

37	RS	RA	D
0	6	11	16 31

```
EA ← (RA) + EXTS(D)
MEM(EA, 4) ← (RS)32:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Simple Instruction in Sail

Simple Instruction in Sail

```
union ast member (bit[5](*RS*), bit[5](*RA*), bit[16](*D*))
StoreWordwithUpdate
```

Simple Instruction in Sail

```
union ast member (bit[5](*RS*), bit[5](*RA*), bit[16](*D*))
StoreWordwithUpdate

function clause decode ((0b100101 : _) as instr) =
  StoreWordwithUpdate(instr[6..10],instr[11..15],instr[16..31])
```

Simple Instruction in Sail

```
union ast member (bit[5](*RS*), bit[5](*RA*), bit[16](*D*))
StoreWordwithUpdate

function clause decode ((0b100101 : _) as instr) =
  StoreWordwithUpdate(instr[6..10],instr[11..15],instr[16..31])

function clause execute (StoreWordwithUpdate(RS, RA, D)) =
{(bit[64]) EA := 0;
  EA := GPR[RA] + EXTS(D);
  GPR[RA] := EA;
  MEMw(EA, 4) :=  (GPR[RS])[32..63];
}
```

Simple Instruction in Sail

```
union ast member (bit[5](*RS*), bit[5](*RA*), bit[16](*D*))
StoreWordwithUpdate

function clause decode ((0b100101 : _) as instr) =
  StoreWordwithUpdate(instr[6..10],instr[11..15],instr[16..31])

function clause execute (StoreWordwithUpdate(RS, RA, D)) =
{(bit[64]) EA := 0;
  EA := GPR[RA] + EXTS(D);
  GPR[RA] := EA;
  MEMw(EA, 4) :=  (GPR[RS])[32..63];
}

val extern forall Nat 'n.
    (bit[64], [|'n|], bit[8*'n]) -> unit effect { wmem } MEMw
```

Simple Instruction in Sail

```
union ast member (bit[5](*RS*), bit[5](*RA*), bit[16](*D*))
StoreWordwithUpdate

function clause decode ((0b100101 : _) as instr) =
  StoreWordwithUpdate(instr[6..10],instr[11..15],instr[16..31])

function clause execute (StoreWordwithUpdate(RS, RA, D)) =
{(bit[64]) EA := 0;
  EA := GPR[RA] + EXTS(D);
  GPR[RA] := EA;
  MEMw(EA, 4) :=  (GPR[RS])[32..63];
}

val extern forall Nat 'n.
    (bit[64], [|'n|], bit[8*'n]) -> unit effect { wmem } MEMw
```

Instruction <-> Thread/Storage Communication

```
type slice = nat * nat

type reg =
| Reg of string
| Reg_slice of string * slice

type outcome
| Read_mem of read_kind * address * size *
              (memory_value -> instruction_state)
| Write_mem of write_kind * address * size * memory_value *
              (bool -> instruction_state)
| Barrier of barrier_kind * instruction_state
| Read_reg of reg * (register_value -> instruction_state)
| Write_reg of reg * register_value * instruction_state
```

Modelling Programs

- Exhaustively, symbolically interpret each instruction
  - Tracking all possible register reads/writes
    - and all possible memory reads/writes
- Thread and/or Storage subsystems calculate dependencies between instructions
- Dependencies tracking portions of registers

Demo

Coverage of POWER ISA Version 2.06 Revision B

Book 1: Power ISA User Instruction Set Architecture

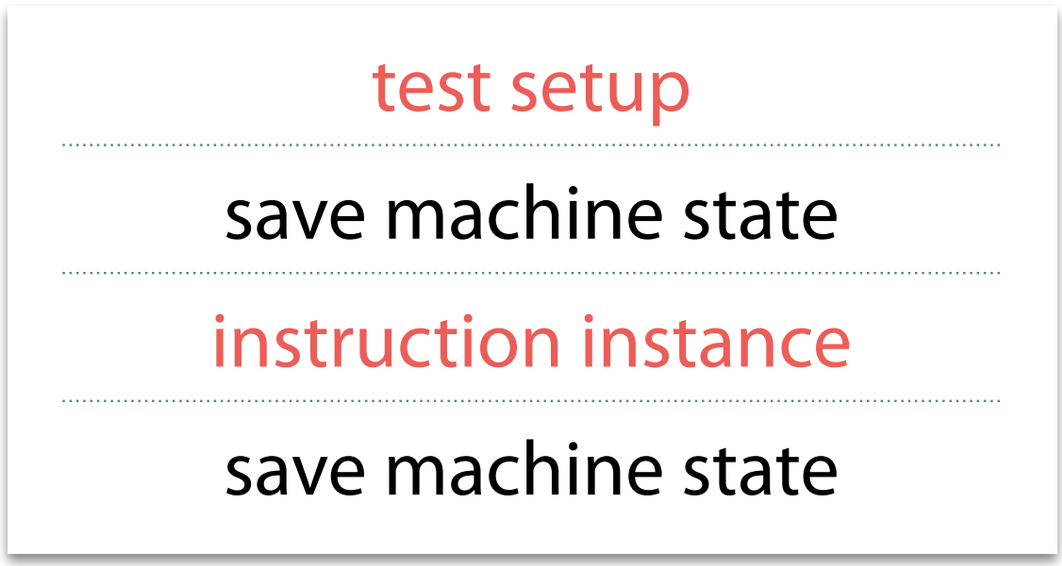
Chapter	Instructions	Auto-Extracted	Tested	Ok
2: Branch Facility	15	13	13	13
3: Fixed-Point Facility	149	146	140	139
4: Floating-Point Facility	78	23	23	19
5: Decimal Floating-Point	50 (no code)	0		
6: Vector Facility	156	112	112	36
7: Vector-Scalar FP	136	0		
8: Signal Processing				
9: Embedded FP				
10: Legacy Move Assist	1	0		
11: Legacy Int Mult-Accum	24	0		

Book 2: Power ISA Virtual Environment Architecture

Chapter	Instructions	Extracted	Tested	Ok
4: Storage Control	21	4	3	3

Single instruction testing

- write tests as assembly programs
- ppcmem is connected to ELF model
- run same* gcc-produced binary tests in model and on machine



* except for logging mechanism

Generating tests

pick *(instruction instance, setup)* based on type information and ISA information

Move From One Condition Register Field XFX-form

mfocrf RT,FXM

31	RT	1	FXM	/	19	/
0	6	11	12	20	21	31

RT (6:10)
Field used to specify a GPR to be used as a target.

FXM (12:19)
Field mask used to identify the CR fields that are to be written by the ***mtcrf*** and ***mtocrf*** instructions, or read by the ***mfocrf*** instruction.

```
| ("RT", Bitvector n) ->
 [let rt1 = random_gpr () in ( rt1, set_GPR rt1 (random_uint64 ()) );
  let rt2 = random_gpr () in ( rt2, set_GPR rt2 (random_uint64 ()) )]
...
| ("FXM", Bitvector n) ->
 [( 2^(Random.int n), set_CR (random_uint32 ()) );
  ( 2^(Random.int n), set_CR (random_uint32 ()) )]
```

Running Litmus Tests

- Exhaustively enumerate possible behaviours
- Run many iterations of test on real hardware
- Tests drawn from manuals, discussions with engineers, exploratory questions

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

Bugs

- Litmus testing found bugs in
  - memory model
  - and hardware
- Single instruction testing found bugs in
  - ISA model
  - Model framework
  - ISA specification (typos and under specification)

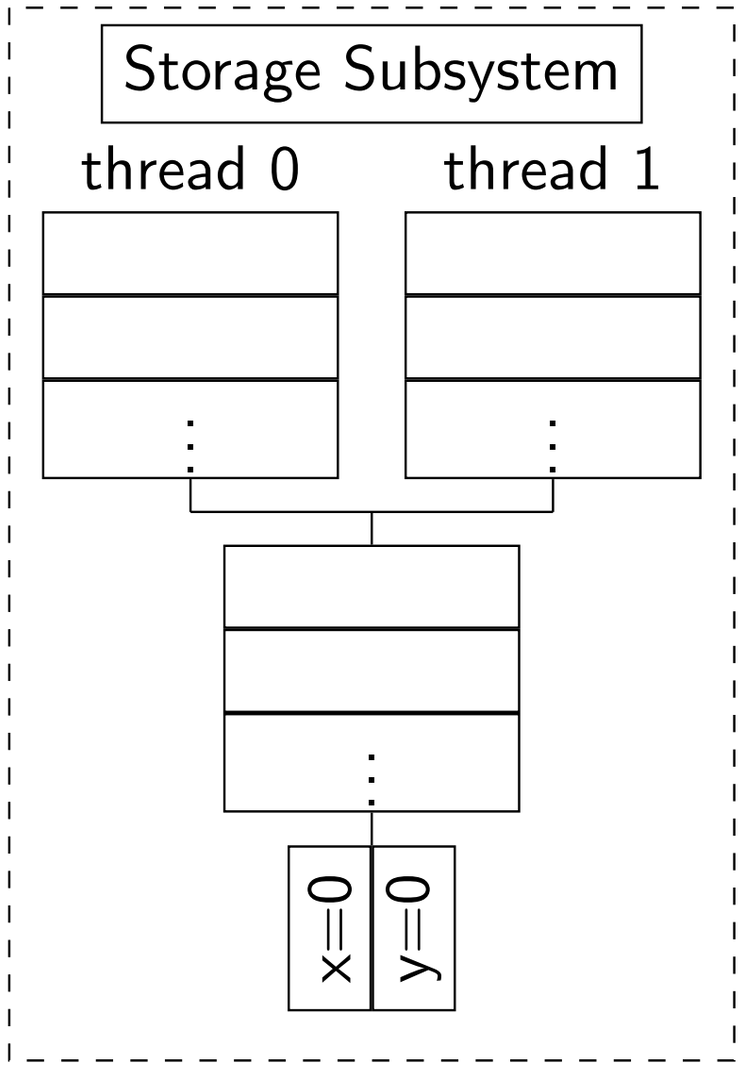
And again for ARM

ARM vs Power

- Memory models similar but with differences
  - Designers have different abstractions
  - Support different instructions (DMB LD/ST, load-acquire, store-release)
  - Different observable behaviour
- ISA contains more pseudocode than average
  - Our spec fully hand-written

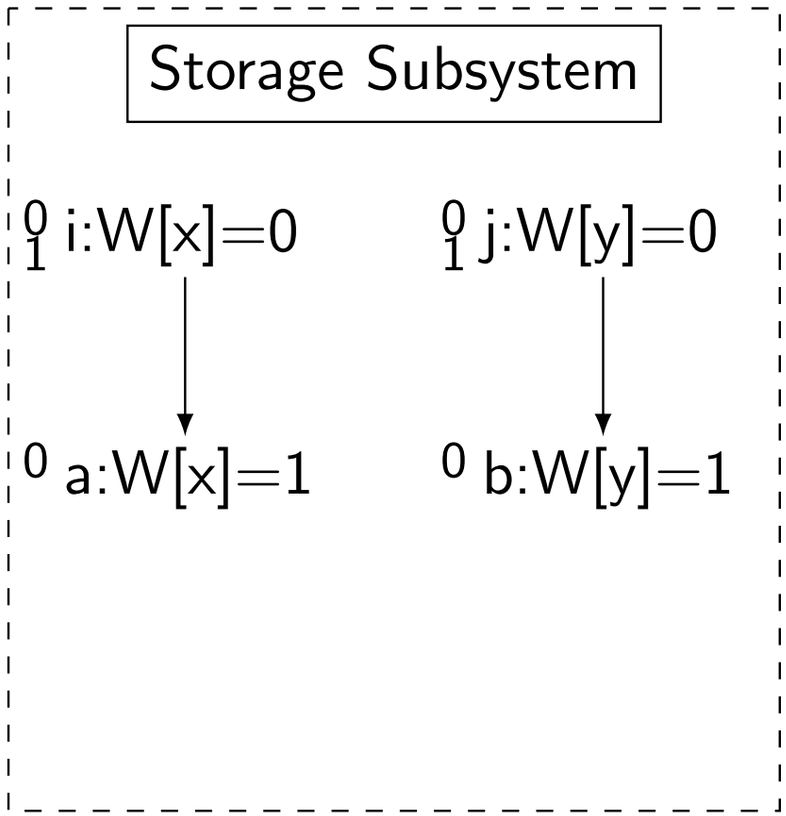
ARM Flowing model

- Operational
- Inspired by designer's intuition
- Storage: Tree topology of queues
  - leaf queue for each thread
  - events flow down



POP memory model
(partial order propagation)

- Eliminates topology
- Sound abstraction of flowing
- Uses a partial order between events
  - An event propagates to a thread if it is sufficiently ordered with other events



What we have

- Executable, explorable model of Power
  - able to run real programs (if small)
    - basis for design of language-level memory
      - including C/C++11
- Similar for ARM (although still in development)

Future paths

Permit program exploration and validation
performance improvements in progress

Use framework to explore other memory models

Use as a platform for analysing programs,
for trusted compilers

Use to prove connection between language models and actual hardware