Applying Concurrency Cookbook Recipes to SPEC JBB

Jade Alglave – Architecture and Technology
Monica Beckwith – Infrastructure LOB; Advanced Server Team

About Us

Dr. Jade Alglave

- Memory model architect at Arm
- Co-developer and maintainer of the herd+diy toolsuite with Luc Maranget (INRIA, France)
- Co-developer and maintainer of the Linux kernel memory model

Monica Beckwith

Managed runtime performance architect at Arm

- Experience with OpenJDK HotSpot JIT, GC
- Experience with JMM and with strong and weakly ordered architecture such as x86-64, SPARC, Arm64 and (very briefly) PPC64

What We Will Cover Today

Introduction to –

Memory Models (Java 💙 Relaxed)

Performance Methodology using Litmus Tests and Tools

Performance Analysis and Measurement using Java Micro-Benchmark Harness (JMH)

Performance Study on Scaling CPU Cores and Simultaneous Multithreading (SMT)

What We Will NOT Cover Today

Details of –

Java 💙 Relaxed memory model

JMH benchmarking

SPEC JBB 2015 benchmarking

CPU cores and simultaneous multithreading (SMT)

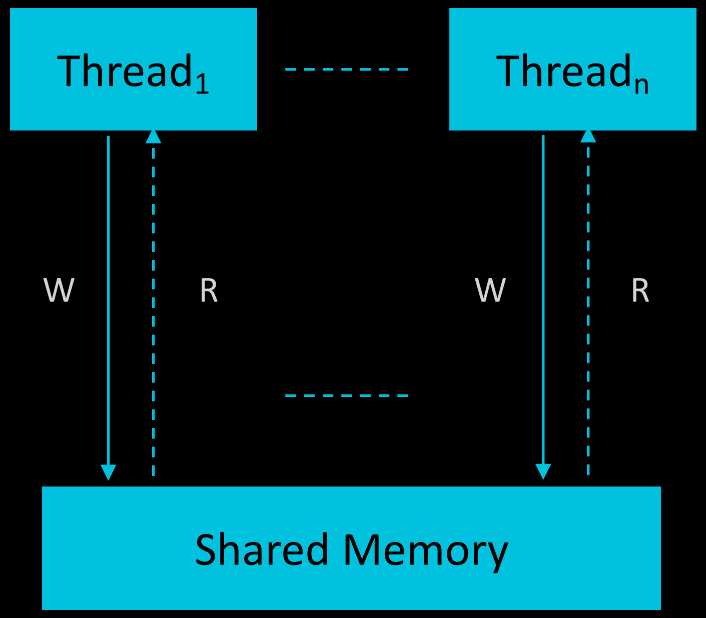
Memory Models

- What value can a load read?

The Ideal Concurrent World of Hardware and Software

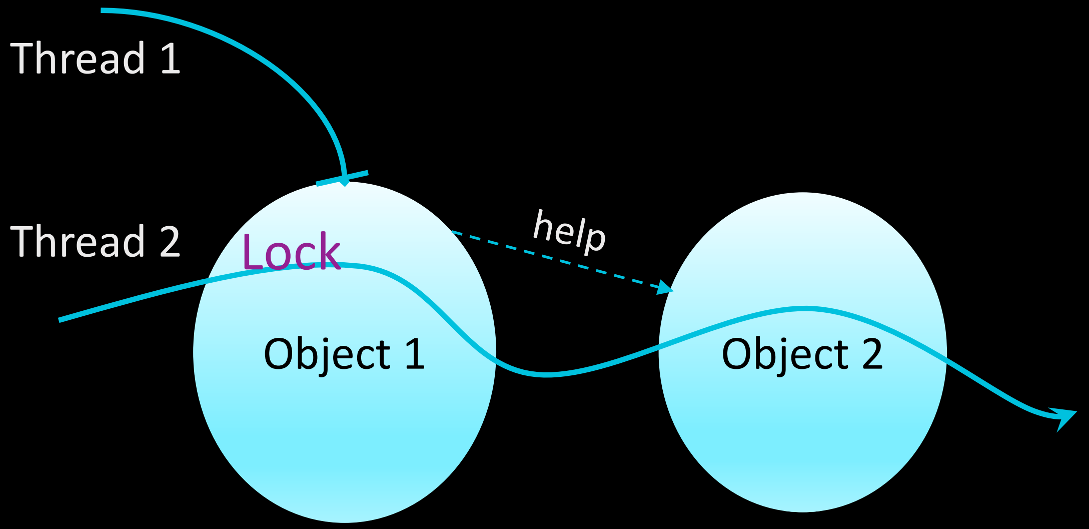
Processor Threads and Software Threads …

Multi-threaded hardware with shared memory structure



* This drawing is heavily inspired by ‘A Tutorial Introduction to the ARM and POWER Relaxed Memory Models’ by Sewell et. al.

A multi-threaded, concurrency-aware program

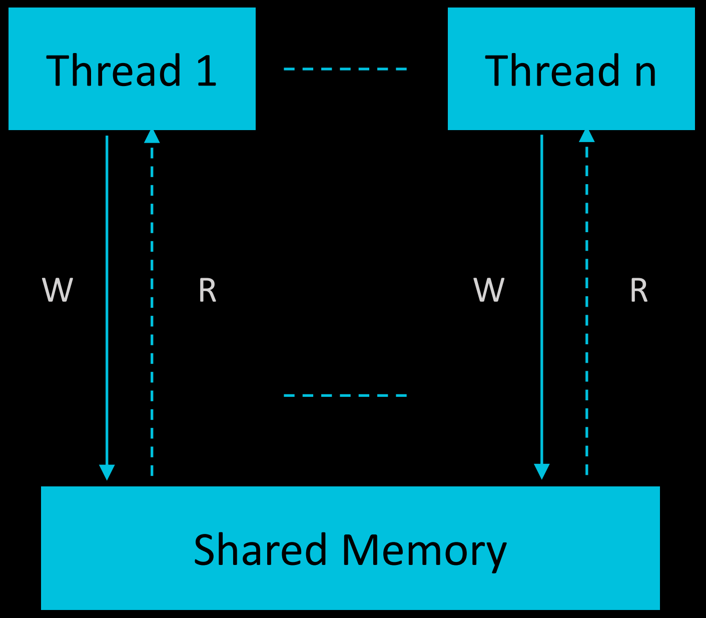


* This drawing is heavily inspired by “timethreads“ concept in Doug Lea’s ‘Concurrent Programming in Java: Design Principles and Patterns, Second Edition ‘

Sequentially Consistent Shared Memory

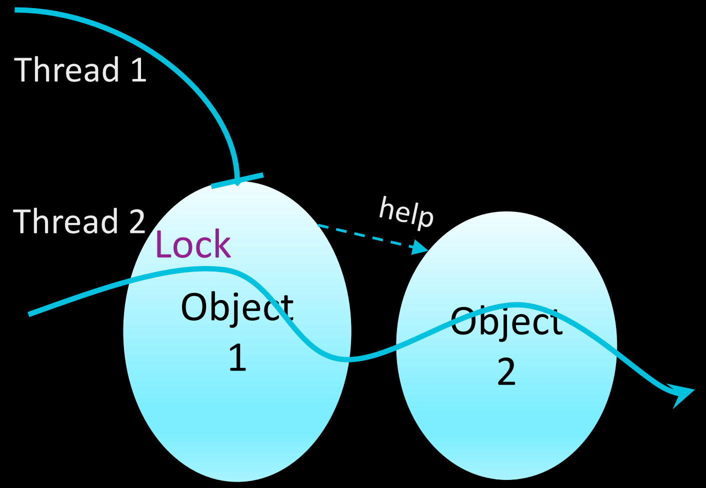
Execution Order == Program Order == Sequential Order

Timeline (Single Global Execution Order)



+

Timeline (Program Order)



=

A Sequentially Consistent Machine

- No local reordering
- Writes become visible simultaneously to all threads

Sequential Consistency in Practice

Store Buffering Example

Initially, `X` and `Y` are `0` in memory; `foo` and `bar` are local (register) variables:

```
     p0                         p1
a: X = 1;                  c: Y = 1;
b: foo = Y;                d: bar = X;
```

What are the permissible values for `foo` and `bar`?

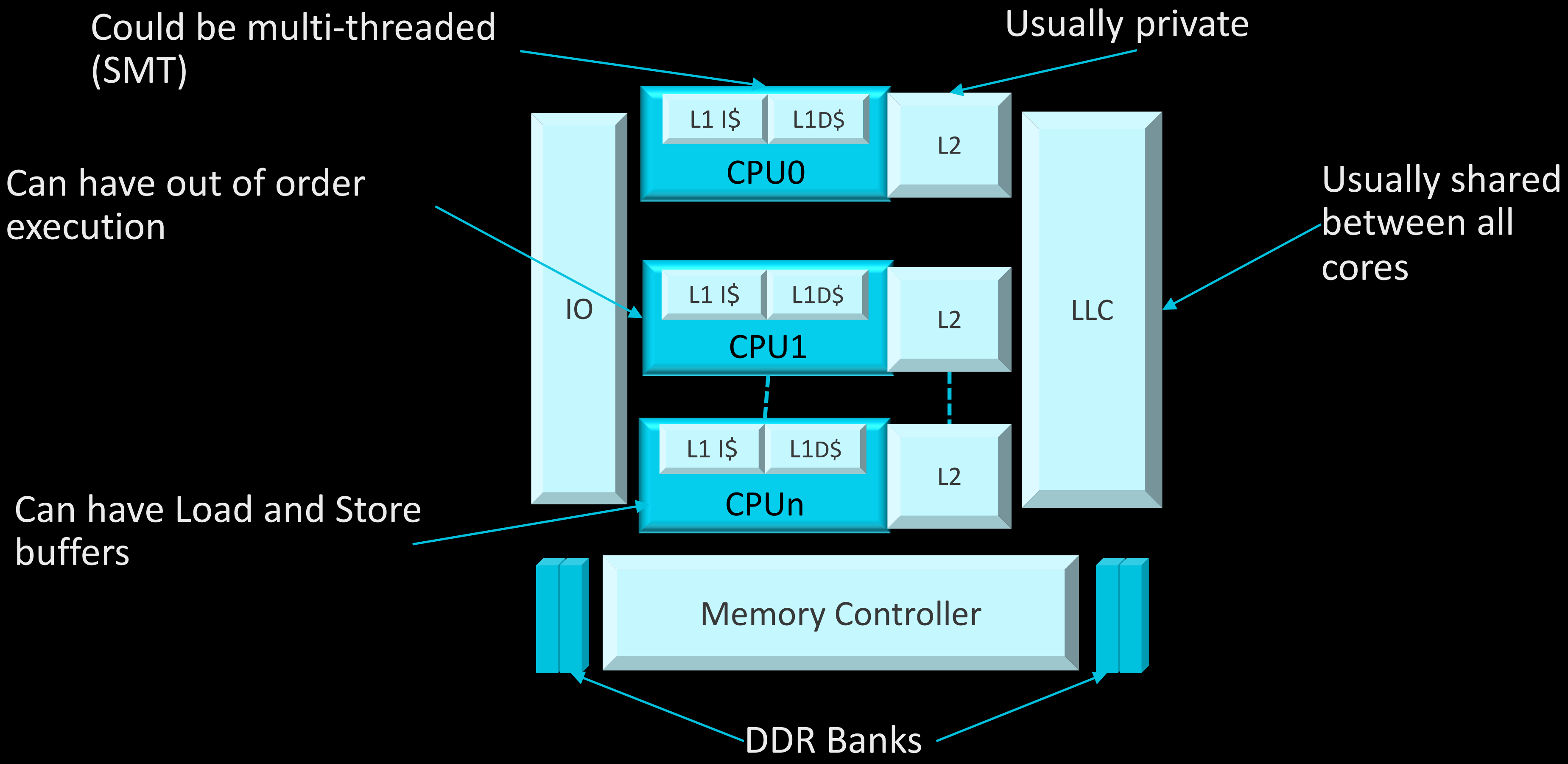
On Sequential Consistency, they are the values reachable by interleavings:

{a,b,c,d} {c,d,a,b} {a,c,b,d}

Therefore we cannot have `foo` and `bar` both equal to `0`.

The Real Concurrent World of Hardware

Multi Processor Threads/Cores with Tiered Memory Structure


Could be multi-threaded (SMT)
Usually private
Can have out of order execution
Usually shared between all cores
Can have Load and Store buffers
IO
L1 I$
L1D$
CPU0
L2
L1 I$
L1D$
CPU1
L2
L1 I$
L1D$
CPUn
L2
LLC
Memory Controller
DDR Banks

Life In The Real World Without Sequential Consistency

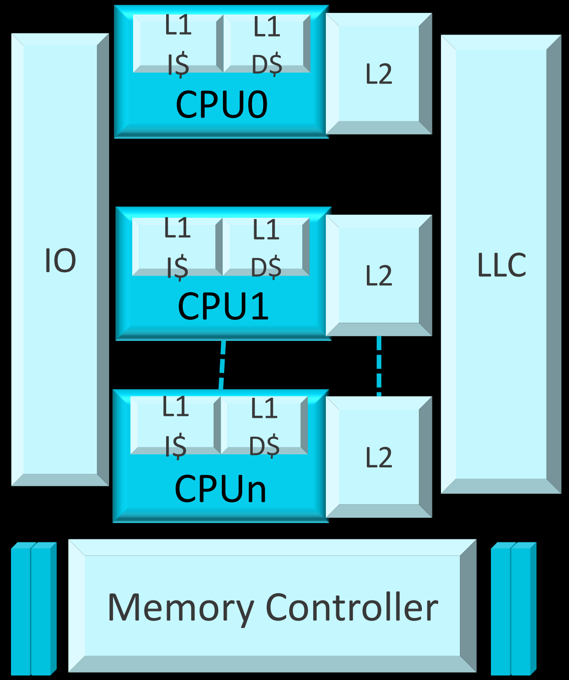
Relaxed vs Strong Memory Model

Strong Memory Models

Strong Models based on Total Store Ordering (TSO)

X86, SPARC

- A thread can see it’s own write before other threads
- All other threads see the write simultaneously: Multiple Copy Atomic Model



Weaker Memory Models

POWER, Arm v7

- Local reordering is allowed
- All threads are not guaranteed to see the write simultaneously: Not Multiple Copy Atomic Model

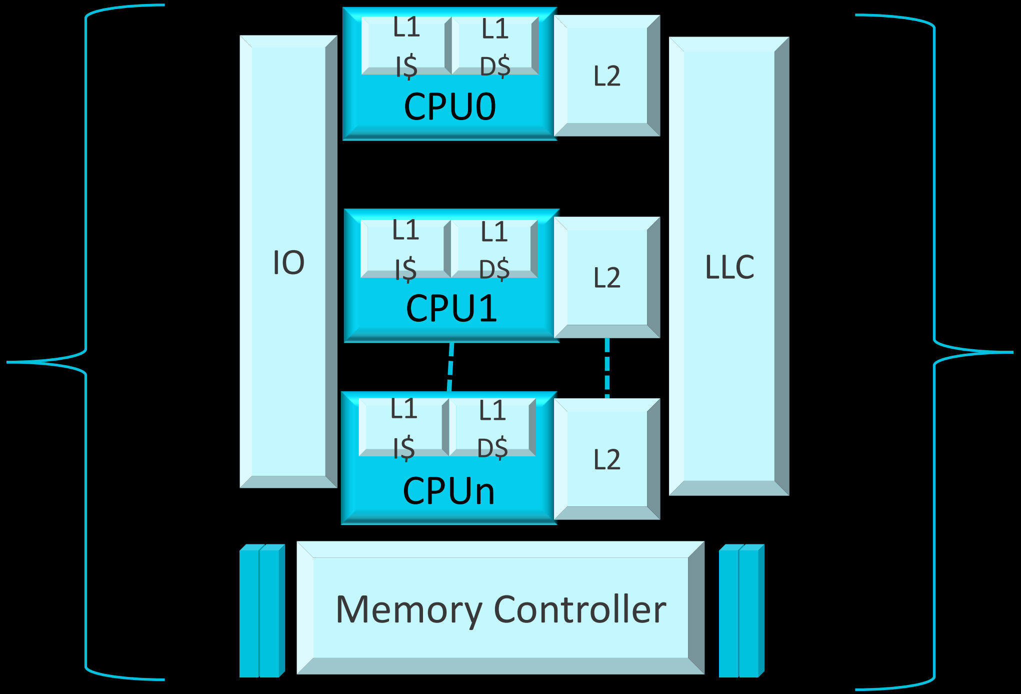
Life In The Real World Without Sequential Consistency

Relaxed vs Strong Memory Model

Strong Models based on Total Store Ordering (TSO)

X86, SPARC

- A thread can see it's own write before other threads
- All other threads see the write simultaneously: Multiple Copy Atomic Model



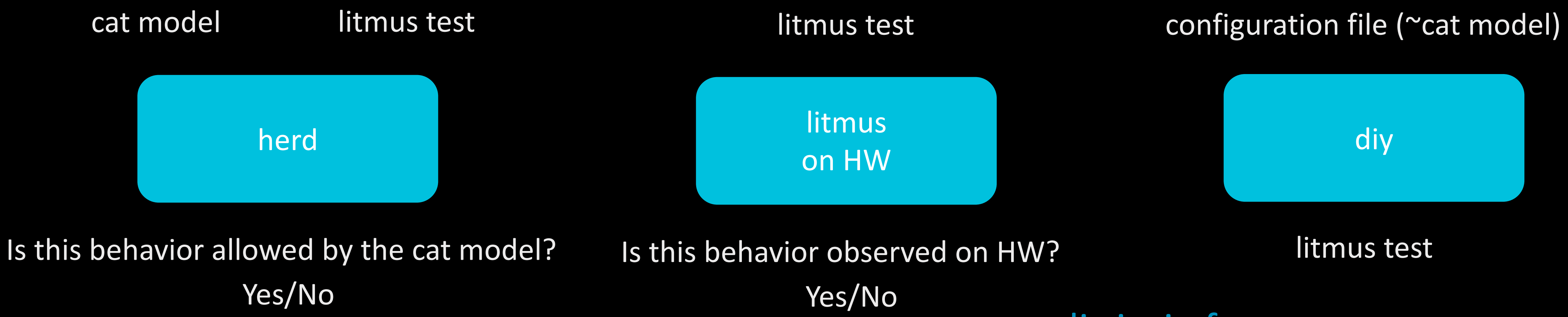
Weaker Memory Models

ARM v8

- Local reordering is allowed
- All threads are guaranteed to see the write simultaneously: Multiple Copy Atomic Model

Going Back To Our Store Buffer Example

- Can we reason about our concurrent programs following Sequential Consistency?
- Probably if we had a formal, preferably executable, memory models to ensure that we understand the guarantees given by architectures and programming languages.
- Here's where Jade would come in talking about her cool tools that allow programmers to explore the consequences of a given memory model or generate vast families of **litmus tests** to run against hardware.



cat model

litmus test

herd

Is this behavior allowed by the cat model?

Yes/No

litmus test

litmus on HW

Is this behavior observed on HW?

Yes/No

configuration file (~cat model)

diy

litmus test

diy.inria.fr

Store Buffer Litmus Test on a TSO Hardware

```
X86 SB
{x=0; y=0;}
 P0          | P1          ;
 MOV [x],$1  | MOV [y],$1  ;
 MOV EAX,[y] | MOV EAX,[x] ;
exists (0:EAX=0 /\ 1:EAX=0)
```

Hardware architecture and test name

Initial state (x and y are shared memory location)

Thread names

Sequence of instructions displayed as columns

Question: can we observe this final state of given that x=0; y=0?

Armed With Knowledge

On TSO hardware, can we observe the final state of `foo=0 and bar=0;` given that `X=0; Y=0?`

`...`

[`...`]

Yes! All production architectures allow the outcome where both foo and bar equal 0.

So, what do we do? …

[Use mfence as needed].

Performance Methodology

- Using Litmus Tests and Tools To Avoid Barriers Where-ever Possible

What & The Why Of Barriers / Fences?

Barriers ensure ordering properties

Barriers enforce strong order

Barriers (when inserted correctly) restore sequential consistency

Barriers can be potentially expensive

Data Memory Barriers on Arm:

- DMB SY (full system)
- DMB ST (wait for store to complete)
- DMB LD (wait for only loads to complete)

Normal Load-Stores

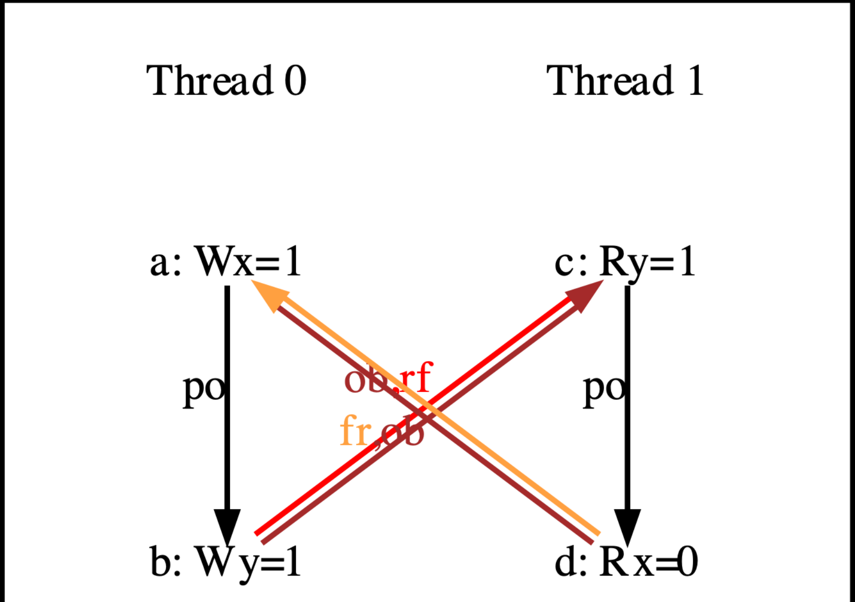
No Barriers

Litmus test

```
AArch64 MP
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0              | P1         ;
 MOV W0,#1       | LDR W0,[X1] ;
 STR W0,[X1]     | LDR W2,[X3] ;
 MOV W2,#1       |            ;
 STR W2,[X3]     |            ;
exists
(1:X0=1 /\ 1:X2=0)
```

Check for any reorder

Check if X0 = 1 and X2 = 0 can exist on P1.



Test MP, Generic(Proposal AArch64)

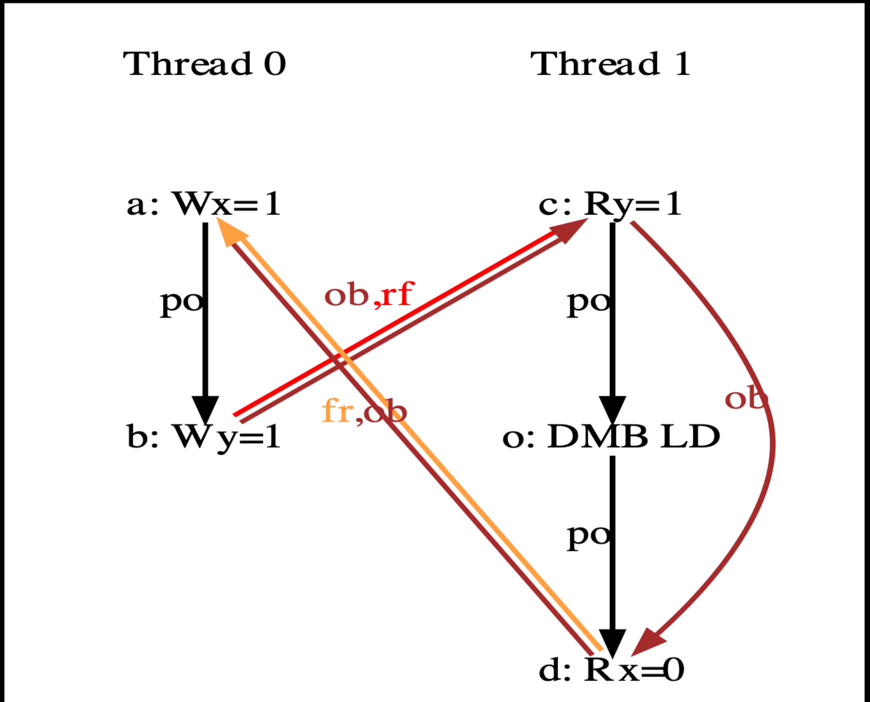
Normal Stores

Load Barrier

Litmus test

```
AArch64 MP+DMB.LD
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0                    | P1        ;
 MOV W0,#1             | LDR W0,[X1] ;
 STR W0,[X1]           | DMB LD;
 MOV W2,#1             | LDR W2,[X3]         ;
 STR W2,[X3]           |          ;
exists
(1:X0=1 /\ 1:X2=0)
```

Check for Store reorder



Test MP, Generic(Proposal AArch64)

Load & Store Barriers

Litmus test

```
AArch64 MP+DMB.LD+DMB.ST
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0                 | P1        ;
 MOV W0,#1          | LDR W0,[X1] ;
 STR W0,[X1]        | DMB LD;
DMB ST              | LDR W2, [X3];
MOV W2,#1           |           ;
 STR W2,[X3]        |           ;
exists
(1:X0=1 /\ 1:X2=0)
```

```
Generated assembler
#START _litmus_P1
ldr w4,[x1]
dmb ld
ldr w5,[x2]
#START _litmus_P0
mov w7,#1
str w7,[x0]
dmb st
mov w6,#1
str w6,[x2]

Test MP+DMB.ST+DMB.LD Allowed
Histogram (3 states)
499999:>1:X0=0; 1:X2=0;
20    :>1:X0=0; 1:X2=1;
499981:>1:X0=1; 1:X2=1;
No

Witnesses
Positive: 0, Negative: 1000000
Condition exists (1:X0=1 /\ 1:X2=0) is NOT vali
Hash=4d15dccdb1da0ce51fac17dea068d047
Observation MP+DMB.ST+DMB.LD Never 0 1000000
```

But Aren’t Barriers Expensive?

Thinking about lock-free?

Acquire – Release (implicit barrier) semantic – One way barriers

LDAR - All loads and stores that are after an LDAR in program order, … must be observed after the LDAR

STLR - All loads and stores preceding an STLR …, must be observed before the STLR

Performance Analysis and Measurement

- Using Java Micro-Benchmarking Harness (JMH)

JMM Rule 1 for Volatile Stores

Order 1	Normal/Volatile Load Normal/Volatile Store	Can’t Reorder
Order 2	Volatile Store	

Any load/store (normal or volatile) followed by a ‘volatile store’ can’t be reordered.

Volatile Stores - Barriers With DMBs

JMM rule: Any load/store followed by Volatile Store can’t be re-ordered

Litmus test

```
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0                | P1        ;
 MOV W0,#1         | LDR W0,[X1] ;
 STR W0,[X1]       | DMB LD ;
 LDR W2,[X1]       | LDR W2,[X3]       ;
 DMB SY            |           ;
 STR W2,[X3]       |           ;
exists
(1:X0=1 /\ 1:X2=0)
```

Results

```
Test MP Allowed
States 3
1:X0=0; 1:X2=0;
1:X0=0; 1:X2=1;
1:X0=1; 1:X2=1;
No
Witnesses
Positive: 0 Negative: 3
Condition exists (1:X0=1 /\ 1:X2=0)
Observation MP Never 0 3
```

Volatile Stores - Can Barriers Be Replaced By STLR?

JMM rule: Any load/store followed by Volatile Store can't be re-ordered

Litmus test

```
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0               | P1      ;
 MOV W0,#1        | LDR W0,[X1] ;
 STR W0,[X1]      | DMB LD ;
 MOV W2,#1        | LDR W2,[X3]      ;
STLR W2,[X3]      |         ;
exists
(1:X0=1 /\ 1:X2=0)
```

Results

```
Test MP Allowed
States 3
1:X0=0; 1:X2=0;
1:X0=0; 1:X2=1;
1:X0=1; 1:X2=1;
No
Witnesses
Positive: 0 Negative: 3
Condition exists (1:X0=1 /\ 1:X2=0)
Observation MP Never 0 3
```

JMM Rule 2 for Volatile Stores

Order 1	Can Reorder	Volatile Store		Can't Reorder
Order 2		Normal Load Normal Store	Volatile Load Volatile Store	

- A ‘volatile store’ followed by any normal load/store CAN be reordered.
- A ‘volatile store’ followed by any volatile load/store CANNOT be reordered.

Volatile Stores – Can Barriers Be Replaced By STLR?

STLR doesn’t guarantee that a subsequent volatile load/store will not be reordered

JMH Test Code

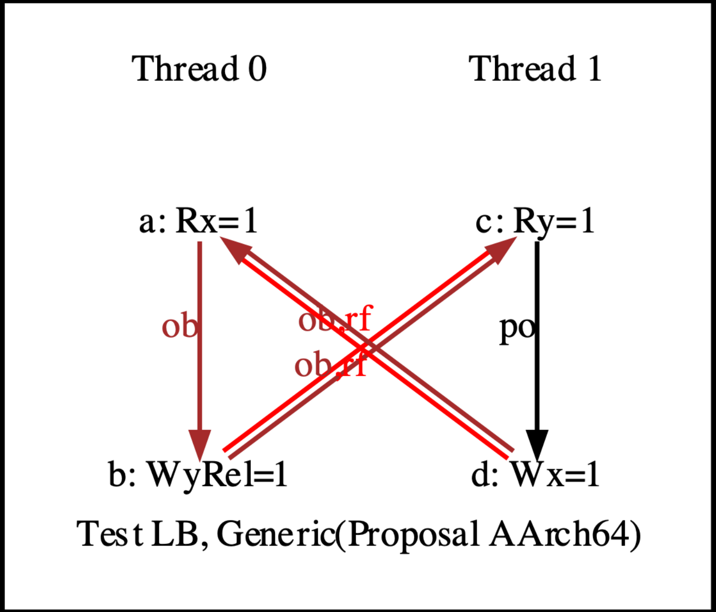
```
   static class
TestNormalLoadPostVolatileStores {
      volatile int intField1;
      int intNorm1;


       public
TestNormalLoadPostVolatileStores() {
            intField1 = 32;
            intNorm1 = intField1;
       }
   }
```

Litmus Test

```
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0                | P1        ;
 LDR W0,[X1]       | LDR W0,[X1] ;
 MOV W2,#1         | MOV W2,#1   ;
 STLR W2,[X3]      | STR W2,[X3] ;
exists
(0:X0=1 /\ 1:X0=1)
```

Positive Event Structure



Volatile Stores – Can Barriers Be Replaced By STLR?

Success! STLR + LDAR of volatiles provide that guarantee

Litmus test

```
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0                | P1       ;
 LDR W0,[X1]       | LDAR W0,[X1] ;
 MOV W2,#1         | MOV W2,#1   ;
 STLR W2,[X3]      | STR W2,[X3] ;
exists
(0:X0=1 /\ 1:X0=1)
```

Results

```
Test LB Allowed
States 3
0:X0=0; 1:X0=0;
0:X0=0; 1:X0=1;
0:X0=1; 1:X0=0;
No
Witnesses
Positive: 0 Negative: 3
Condition exists (0:X0=1 /\ 1:X0=1)
Observation LB Never 0 3
```

Volatile Stores – JMH Profiles

Success! STLR + LDAR of volatiles provide that guarantee

Load Acquire – Store Release Pair

```
stlr  w11, [x10]      ;*putfield intField1 ;

add   x10, x2, #0xc

ldar  w11, [x10]      ;*getfield intField1
```

36% faster on max SMT count!!

Data Memory Barrier (inner share-ability domain

```
 str     w10, [x2,#12]

dmb     ish            ;*putfield intField1

ldr     w11, [x2,#12]

dmb     ishld          ;*getfield intField1
```

JMM Rule 1 for Volatile Loads

A ‘volatile load’ followed by any load/store (normal or volatile) can’t be reordered.

Order 1	Volatile Load	Can’t Reorder
Order 2	Normal/Volatile Load Normal/Volatile Store	

Volatile Load - Barriers With DMBs

JMM rule: A Volatile Load followed by any load/store can't be re-ordered

Litmus test

```
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0              | P1       ;
 LDR W0,[X1]     | LDR W0,[X1] ;
 DMB SY          | MOV W2,#1   ;
 MOV W2,#1       | DMB SY ;
 STR W2,[X3]     | STR W2,[X3] ;
exists
(0:X0=1 /\ 1:X0=1)
```

Results

```
Test LB Allowed
States 3
0:X0=0; 1:X0=0;
0:X0=0; 1:X0=1;
0:X0=1; 1:X0=0;
No
Witnesses
Positive: 0 Negative: 3
Condition exists (0:X0=1 /\ 1:X0=1)
Observation LB Never 0 3
```

Volatile Stores – Can Barriers Be Replaced By LDAR?

Success! LDAR provides the right guarantee

Litmus test

```
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0              | P1         ;
 LDAR W0,[X1]    | LDR W0,[X1] ;
                 | MOV W2,#1   ;
 MOV W2,#1       | DMB SY ;
 STR W2,[X3]     | STR W2,[X3] ;
exists
(0:X0=1 /\ 1:X0=1)
```

Results

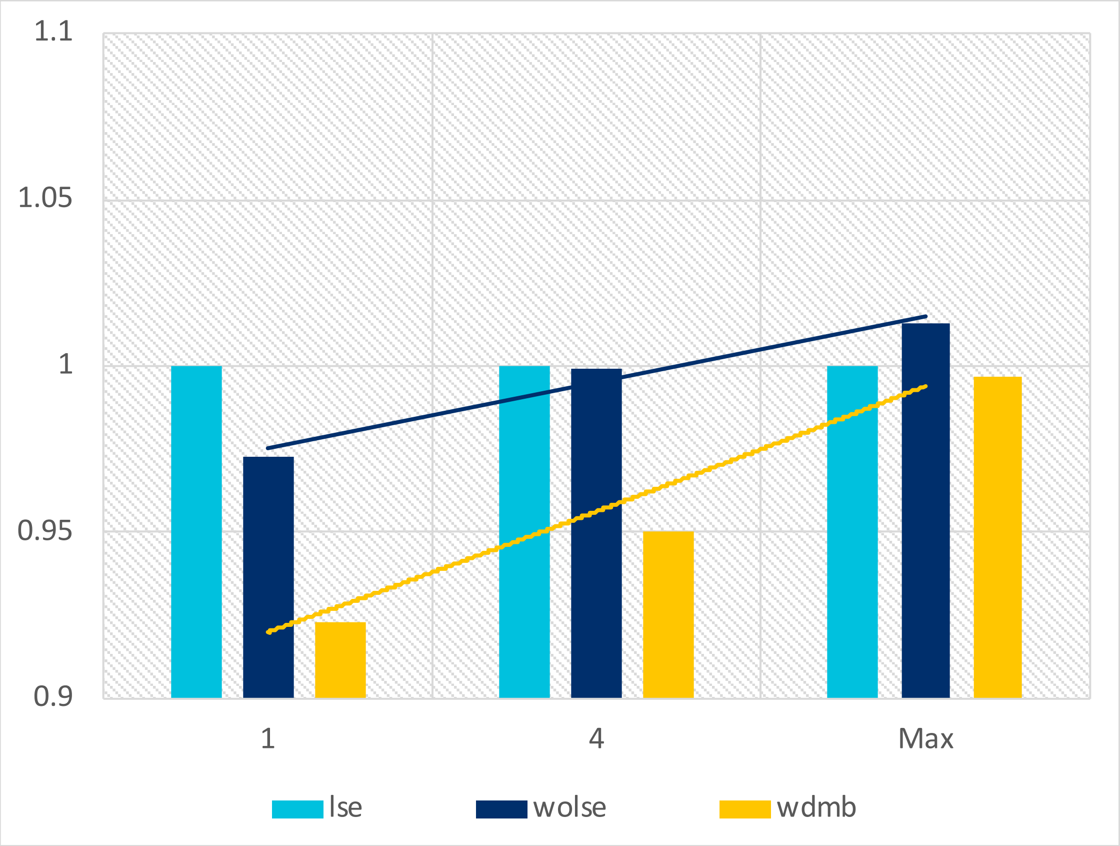
```
Test LB Allowed
States 3
0:X0=0; 1:X0=0;
0:X0=0; 1:X0=1;
0:X0=1; 1:X0=0;
No
Witnesses
Positive: 0 Negative: 3
Condition exists (0:X0=1 /\ 1:X0=1)
Observation LB Never 0 3
```

Performance Study

- Scaling CPU Cores / Simultaneous Multithreading (SMT)

Applying Cook Book Recipes to SPECJBB

Bigger is Better



Core or Thread Count	With LSE; With LDAR (baseline)	Without LSE; With LDAR	With LSE; With DMB
1	1.00	0.97	0.92
4	1.00	1.00	0.95
Max	1.00	1.01	1.00

- Fences/Barriers(e.g. DMB ST, DMB LD, DMB SY)
- Atomics/LSE (e.g. LDREX/STREX or CAS)

Single Core Performance

The Quest and Guarantee of Sequential Consistency

Hardware improvements measured on Java micro-benchmarks (OpenJDK JDK11):

- Object/memory allocations up to ***2.4x faster***
- Object/array initializations up to ***5x faster***
  - Smart issuing and cost reduction of SW barriers (i.e. DMB) required by Arm's relaxed memory model
- Copy chars up to ***1.6x faster***
- New atomic instructions improve locking throughput and contention latency by up to ***2x***

Cortex-A72	Code	Neoverse N1
0.21%	`dmb ishst ;*new`	(0 cycles) 0.00%
7.73% (~3.5 cycles)	`ldr x11, [sp,#8]`	
1.75%	`ldr w17, [x11,#12];*getfield`	0.06%
	`mov x2, x0`	
0.51%	`ldp w0, w18,[x11,#16];*getfield`	0.11%
0.42%	`ldp w3, w1, [x11,#24];*getfield`	

org.openjdk.bench.vm.compiler.generated.StoreAfterStore_testAllocAndZeroStore_jmhTest::testAllocAndZeroStore_avgt_jmhStub

Ares Single Core Performance

Hardware improvements measured on SPECJBB (OpenJDK JDK11):

- Neoverse N1 CPU improves performance from Cortex-A72 by ***1.7x***

Software improvements measured on SPECJBB:

- JDK11 improves performance vs JDK8 on Arm by up to ***14%***

Resources

http://g.oswego.edu/dl/jmm/cookbook.html
https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf

http://hg.openjdk.java.net/code-tools/jmh-jdk-microbenchmarks/file/92c55597888e/README.md

http://infocenter.arm.com/help/topic/com.arm.doc.genc007826/Barrier_Litmus_Tests_and_Cookbook_A08.pdf

Appendix

The herd+diy Toolsuite

The tool suite supports and provides a formal consistency model for all of the following:

- Arm, IBM Power, Intel x86
- Nvidia GPUs
- C/C++
- Linux C

This means that a user can experiment with the concurrency implemented at all these levels and generate systematic families of tests to probe implementations.

diy.inria.fr

A Store Barrier Litmus Test

PodWW Rfe PodRR Fre

Fre PodWR Fre PodWR

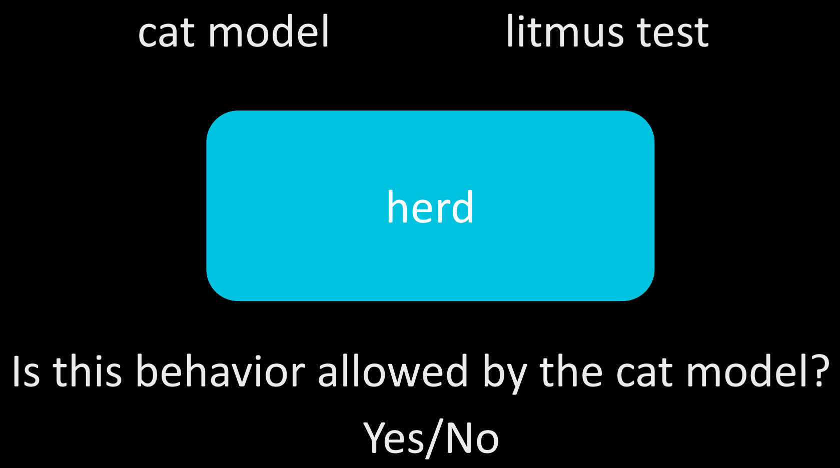
A litmus test source has three main sections:

The initial state defines the initial values of registers and memory locations. Initialisation to zero may be omitted.

The code section defines the code to be run concurrently — above there are two threads. Yes we know, our X86 assembler syntax is a mistake.

The final condition applies to the final values of registers and memory locations.

Executing the model: herd



cat model

litmus test

herd

Is this behavior allowed by the cat model?

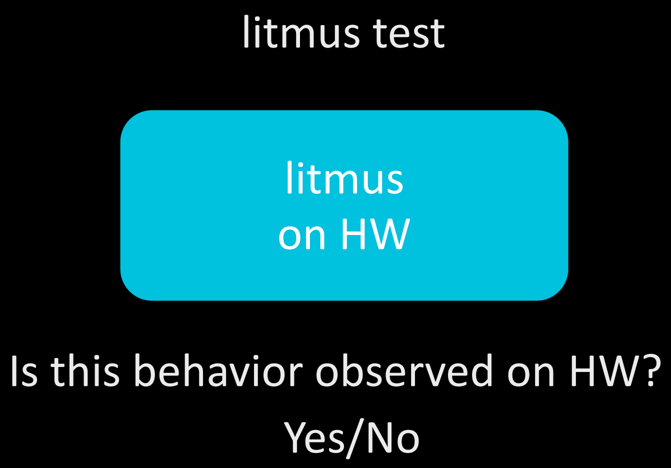
Yes/No

The herd tool allows a user to execute a formal model, written in the cat language.

Given a litmus test and a cat model, herd runs the litmus test against the cat model:

herd tries to determine whether the model allows the final state given in the test can be reached.

Running tests on hardware: litmus



litmus test

litmus on HW

Is this behavior observed on HW?

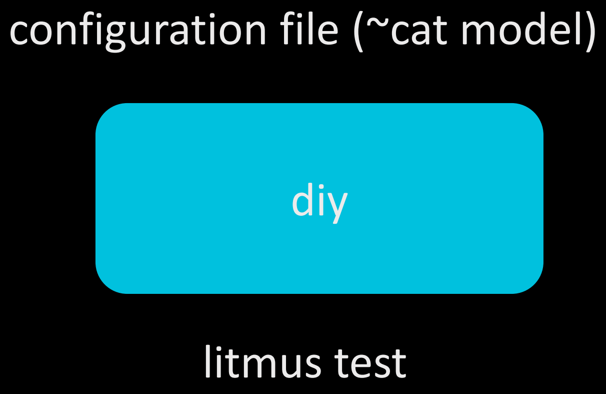
Yes/No

The litmus tool allows a user to run a litmus test against hardware.

The tool gathers all the final states that were observed on hardware during multiple runs of the test.

We can then compare the output of herd and litmus, to check whether they are in accord.

Generating tests: diy



configuration file (~cat model)

diy

litmus test

The diy tool allows a user to generate interesting families of litmus tests.

It takes as input a configuration file, where a user should list the features of interest to them.

We can use families of diy-generated tests to run validation campaigns,
comparing the cat model and prototypes.

arm NEOVERSE

The Cloud to Edge Infrastructure Foundation
for a World of 1T Intelligent Devices

Thank You!