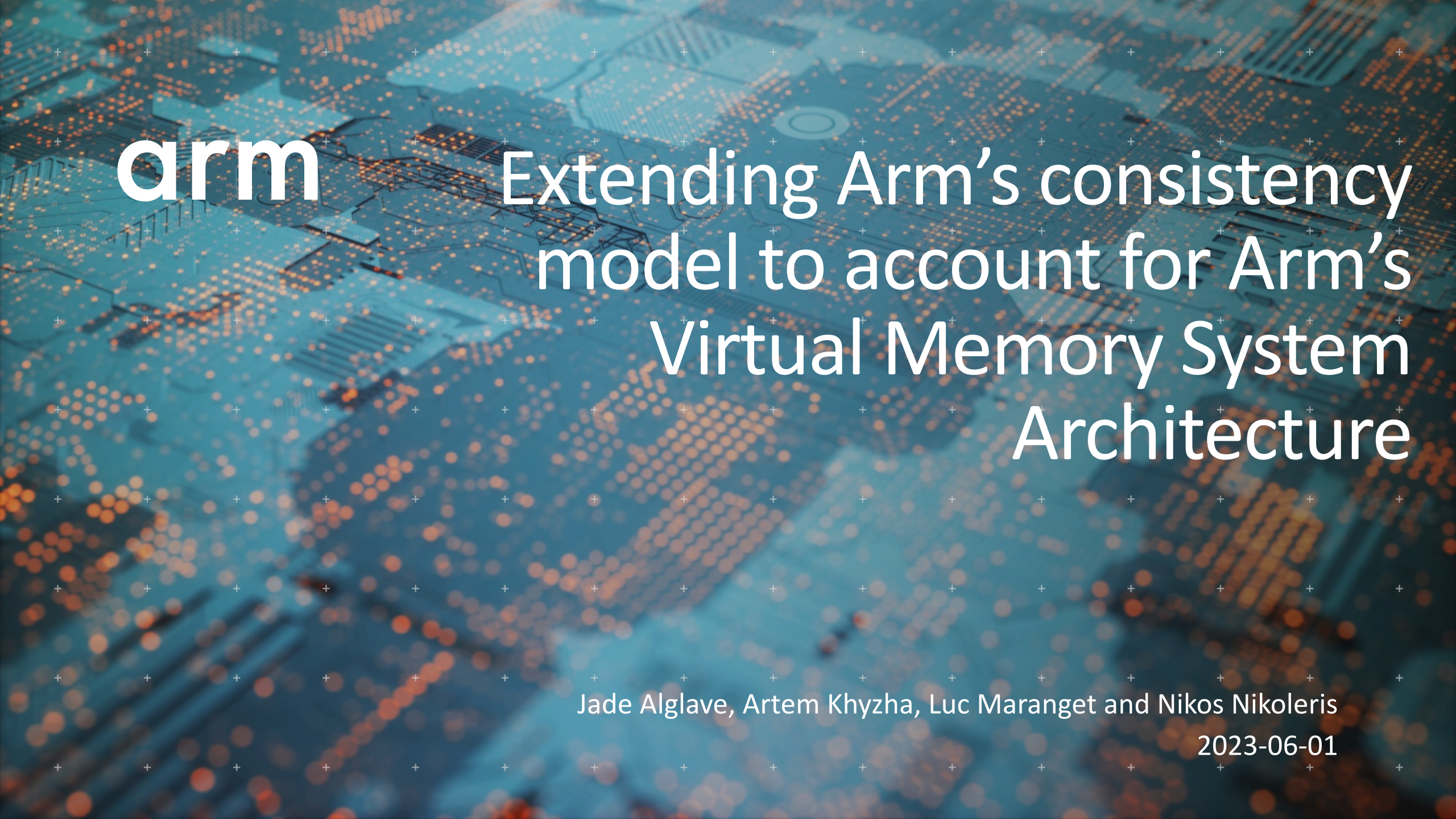


Extending Arm's consistency model to account for Arm's Virtual Memory System Architecture

Jade Alglave, Artem Khyzha, Luc Maranget and Nikos Nikoleris

2023-06-01

Arm Architecture

- Arm develops *architecture specifications.*
- Architecture describes the envelope of allowed and forbidden behaviours that implementations must adhere to – in that sense the Arm architecture presents a unifying view of the variety of hardware Arm implementations, which are in phones, tablets, laptops and servers.
- Or the other side of the same coin: architecture describes what software developers can rely on coming from hardware.
- For example the following message passing behaviour (MP) is allowed by the Arm architecture:

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

- Within the Arm architecture group, I am in charge of the memory model.

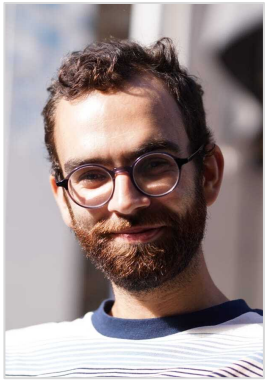
arm

Our team

Nikos Nikoleris

Artem Khyzha

Hugo O'Keeffe



Hadrien Renaud

Luc Maranget

Arm's memory model

- Since 2016, Arm's memory model has been the subject of a formalization effort.
- We write our memory model in a language called cat, which has a formally defined semantics; hence all models written in cat have a precise mathematical meaning.
- The cat language is also used to describe the Linux kernel memory model, but also C++, RISCV…
- This language drives a collection of tools (see diy.inria.fr), amongst which:
  - The herd7 tool takes as input a cat model and a litmus test, and returns the architecturally permissible behaviors of that test under that model;
  - The diy7 tool generates systematic families of litmus tests;
  - The litmus7 tool runs those tests on hardware.
- Therefore a cat model is a formal, machine-readable and executable artefact.
- We upstream extensions to the model and the companion tools regularly:
  https://developer.arm.com/architectures/cpu-architecture/a-profile/memory-model-tool
- We have written a number of tutorials on how to use the tools.

Our tools are available

The releases and documentation can be found at:

https://diy.inria.fr/

The development version is at:

https://github.com/herd/herdtools7

The web interface to the Armv8 cat model, including the VMSA extension, is at:

https://diy.inria.fr/www

Tutorial blogs are available

How to use the herd7 simulator:
https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/how-to-use-the-memory-model-tool

How to generate tests with the diy7 test generator (including a configuration file to feed to diy to generate interesting tests):
https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/generate-litmus-tests-automatically-diy7-tool

How to run tests on silicon with the litmus7 tool:
https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/running-litmus-tests-on-hardware-litmus7

How we have extended our testing and modelling to page table related concerns:
https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/expanding-memory-model-tools-system-level-architecture

How we have extended our testing and modelling to Memory Tagging and Morello:
https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/memory-model-tool-morello-and-some-memory-tagging

arm

The herd7 simulator

https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/how-to-use-the-memory-model-tool

Executing the model using the herd7 tool

Demo in web interface: https://diy.inria.fr/www

MP is Allowed

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

MP+DMB.ST+DMB.LD is Forbidden

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


arm

Armv8 memory model at a high level

- The Armv8 memory model gives ordering requirements over memory accesses, intuitively stating which local orderings must be respected by hardware – those are called hardware requirements.
- Those constraints forbid certain shapes in program executions.
- For example the External visibility requirement forbids cycles in the Ordered-before relation, which consists of local orderings (Hardware-required-ordered-before relation) and interactions between threads (Observed-by relation).

Correspondence between Arm cat model and Architecture Reference Manual

Section B2.3

In cat

```
let rec ob = obs
           | hw-reqs
           | ob; ob

irreflexive ob as external
```

In English

Ordered-before

An Effect E1 is Ordered-before an Effect E2 if and only if one of the following applies:

- E1 is a Read or Write Memory Effect RW1, E2 is a Read or Write Memory Effect RW2 and RW1 is Observed-by RW2
- E1 is hardware-required-ordered-before E2
- E1 is Ordered-before an Effect E3 and E3 is Ordered-before E2.

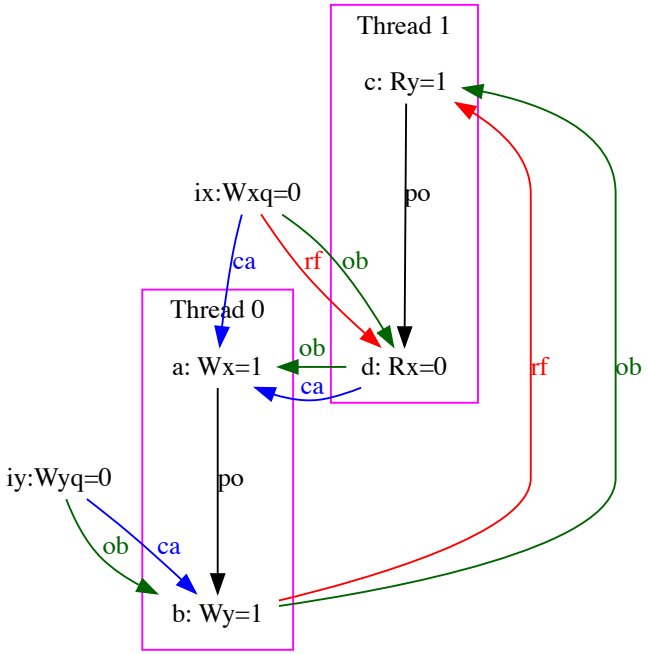
External visibility

An architecturally well-formed execution must not exhibit a cycle in the ordered-before relation.

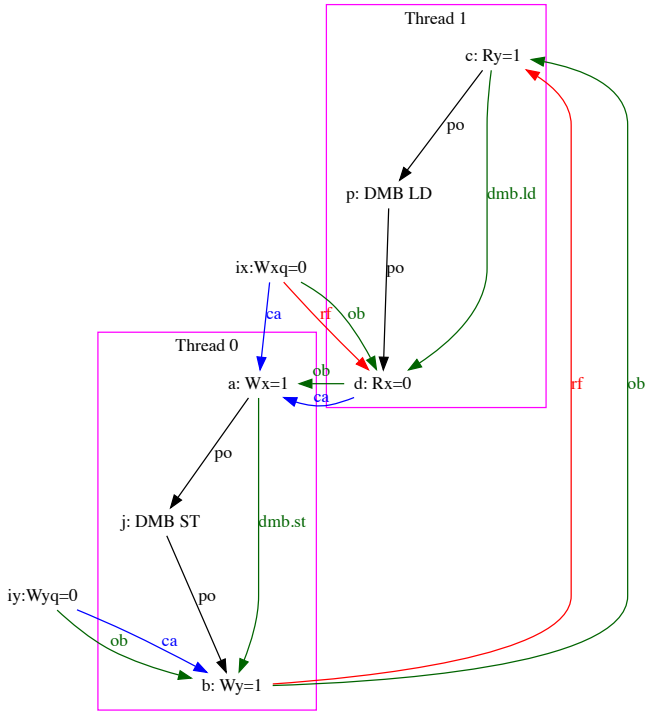
Executing the model using the herd tool

Demo in web interface: https://diy.inria.fr/www

MP is Allowed



MP+DMB.ST+DMB.LD is Forbidden



arm

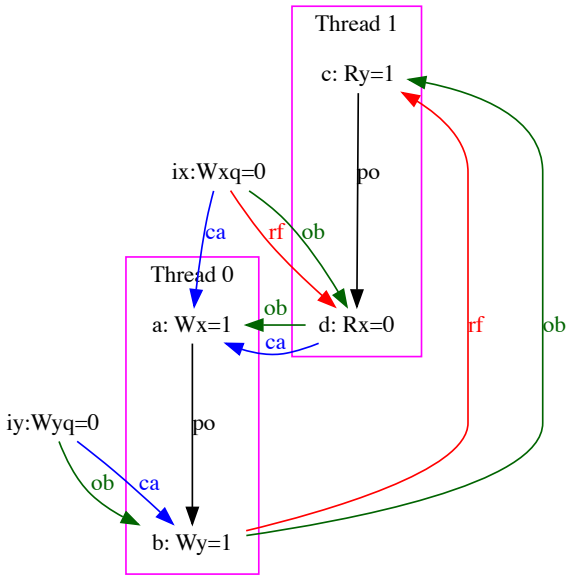
The diy7 test generator

https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/generate-litmus-tests-automatically-diy7-tool

Generating one test with the diyone7 tool

Demo on the command line

Execution of the MP test



The cycle at play: "PodWW Rfe PodRR Fre"

- Program order on Thread 0 between a write of location x (event a) and a write of location y (event b) -> PodWW
- Read-from from the write of y on Thread 0 to the read of y on Thread 1 -> Rfe
- Program order on Thread 1 between a read of y (event c) and a read of x (event d) -> PodRR
- From-read (Coherence-after) from the read of x on Thread 1 to the write of x on Thread 0 -> Fre

Using diyone7 to generate MP

On the command line:

```
diyone7 -arch AArch64 "PodWW Rfe PodRR Fre"
```

gives:

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


arm

arm

The litmus7 test harness

https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/running-litmus-tests-on-hardware-litmus7

Running tests on hardware with the litmus7 tool

Demo on the command line

- We can run a test on hardware, as follows:

```
litmus7 mp.litmus
```

- The tool then reports which final states have been observed during the runs.
- It is possible to run a set of tests at once.
- It is also possible to prepare a set of Arm tests on an x86 machine (for example) using cross-compilation, then to copy those compiled tests on an Arm machine and run them there.
- The litmus7 tool can be useful for post-silicon validation.

arm

Virtual Memory

Virtual and physical memory

- In the previous examples I showed, we assumed that ‘x’ and ‘y’ were *physical memory locations.*
- But usually one interacts with *virtual memory addresses* rather than physical memory directly.
- Virtual memory gives an abstract view of the physical memory resources, which amongst other benefits:
  - frees software developers from having to manage a shared memory space, and
  - affords them more memory than might be physically available.
- The mapping from virtual addresses to physical locations resides in a collection of *page tables*.

Page tables

- Page tables are used to translate the virtual addresses seen by software into physical locations used by the hardware
- Each *page table entry* (PTE) holds a flag indicating whether the entry is *valid or not*:
  - If it is valid, reading the PTE gives the physical location corresponding to a virtual address;
  - If it is not valid, accessing a virtual address via that PTE raises a *fault*.

Translation Lookaside Buffers

- A translation lookaside buffer (TLB) is a special cache that stores the recent translations of virtual memory to physical memory.
- Arm provides a special instruction called TLBI to ensure that this special cache is kept up to date.
- Part of the complexity of modelling VMSA is to understand exactly what TLBI does.

arm

Extending litmus tests and tools

Extended litmus test format

PTE values and faults

A variant of MP: MP-pte

```
AArch64 MP-pte
{
pte_x=(valid:0,oa:phy_x);
0:X2=pte_x;
0:X1=(valid:1,oa:phy_y);
1:X3=x;
y=1;
0:X8=z; 1:X8=z;
}
P0           | P1;
STR X1,[X2] | LDR W7,[X8]   ;
MOV W7,#1    | L0:LDR W4,[X3];
STR W7,[X8] |                ;
exists(1:X7=1/\fault(P1:L0,x))
```

MP-pte in words

- Starting from an initial state where the page table entry `pte_x` (which by default points to the physical address `phy_x`) is invalid (because the `valid` bit is set to 0), and where `y` holds the value 1
- A thread P0 updates the page table entry `pte_x`, making it not only valid but also pointing to a different physical address `phy_y`, and sets up a flag `z` to 1
- A thread P1 reads the flag `z`, and reads `x`

New syntax

- The initial state can now refer to a new type of value, which corresponds to page table entries. For example in the initial state for MP-pte, we have: `pte_x=(valid:0,oa:phy_x)`. In words, this says that the page table entry for `x` is initialized so that its `valid` bit is 0, and so that it points to a physical address `phy_x`
- In the final clause, we can now specify whether an access is expected to fault. We can do this using labels in the body of the test (see label `L0` on thread P1). For example, in the final clause for MP-pte, we have: `fault(P1:L0,x)`. In words, this asks whether the access at line `L0` on thread P1 can provoke a fault relative to address `x`.

arm

Extending the herd tool

- To model page table related concerns, we added several features to the herd7 tool, amongst which:
  - A new notion of PTE accesses;
  - A new Fault effect and the ability to ask whether a litmus test can fault.

Translation-intrinsically-before

In cat

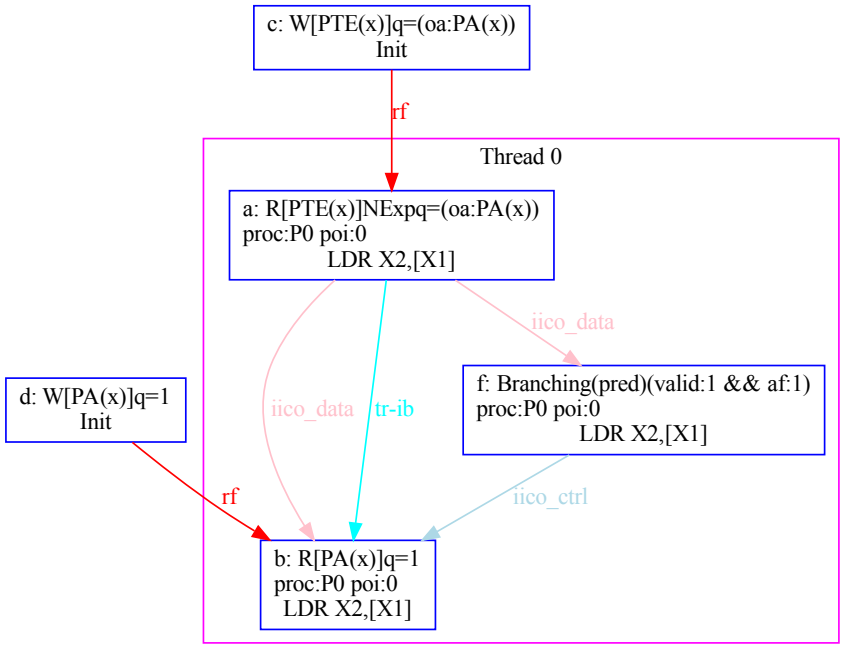
```
let tr-ib =
[R & PTE & Imp]; iico_data; [B]; iico_ctrl; [M & Exp | FAULT]
```

In English

An Implicit PTE Read Memory Effect R1 is Translation-intrinsically-before an Explicit Read or Write Memory or a Fault Effect E2 if and only if all of the following applies:

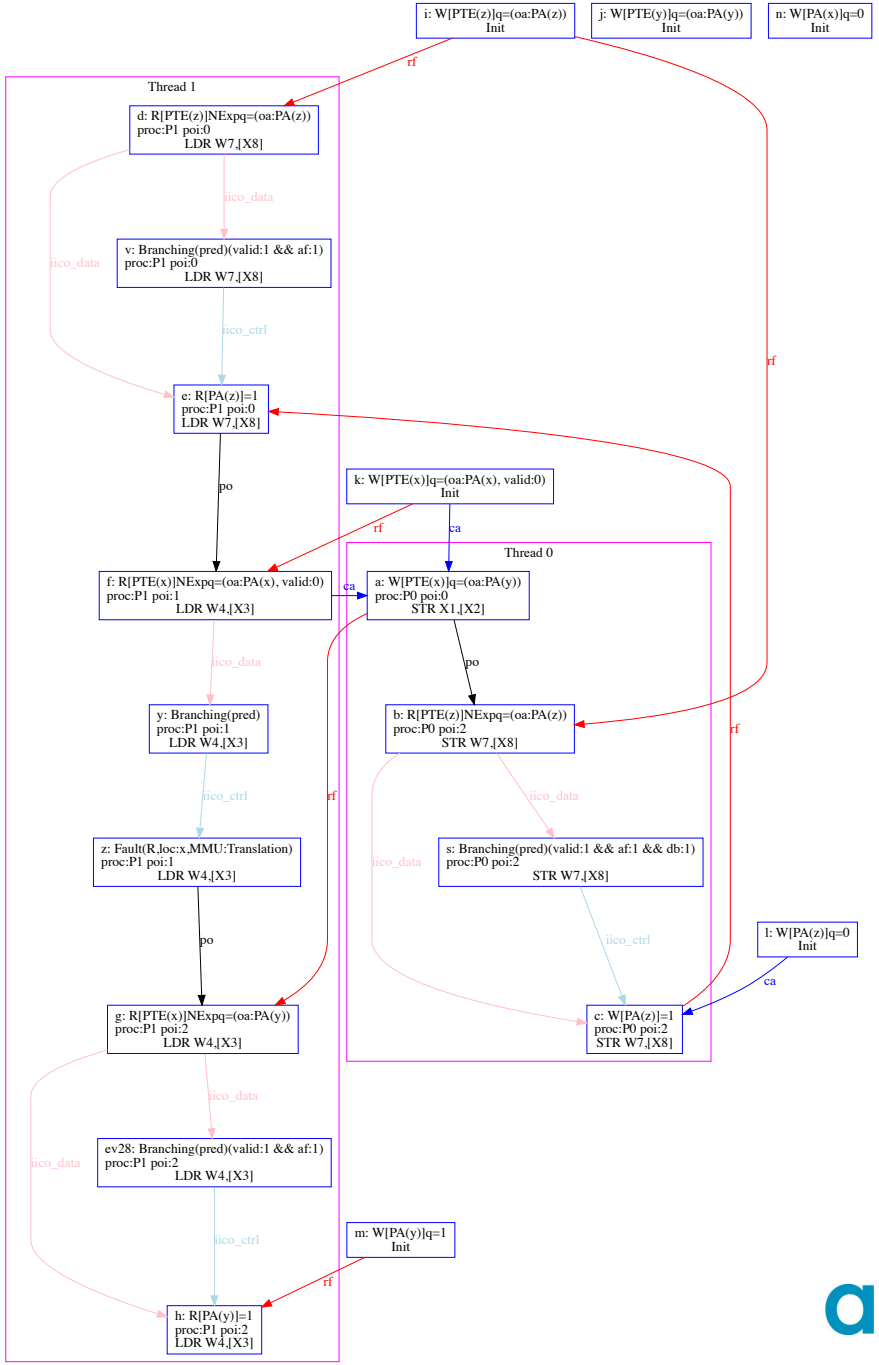
- R1 is before a Branching Effect B3 in the Intrinsic Data dependency order, and
- B3 is before E2 in the Intrinsic Control dependency order.

Illustration: semantics of LDR



Execution of MP-pte

```
AArch64 MP-pte
{
pte_x=(valid:0,oa:phy_x);
0:X2=pte_x;
0:X1=(valid:1,oa:phy_y);
1:X3=x;
y=1;
0:X8=z; 1:X8=z;
}
P0           | P1;
STR X1,[X2] | LDR W7,[X8]   ;
MOV W7,#1   | L0:LDR W4,[X3];
STR W7,[X8] |                ;
exists(1:X7=1/\fault(P1:L0,x))
```



Extending the litmus tool

- To run those page table tests, we have extended our litmus7 tool
  `sh MP-pte/run.sh`
- The litmus7 tool can now generate the sources for a binary that runs as a Virtual Machine (VM).
- To run litmus tests in a VM, litmus7 builds on kvm-unit-tests. The kvm-unit-tests project was originally intended to test the Kernel-based Virtual Machine (KVM) features, and litmus7 uses its functionality to generate litmus tests that can run in a VM.
- By doing that, we can control and make changes to functionality that typically would be provided by an operating system running in EL1. For example, the test can set up and control translation tables and the Memory Management Unit (MMU), or devices available to the VM.
- We also have the ability to run those tests bare-metal.

arm

A Copy-on-Write example from Linux

A user-level example: coRR

```
AArch64 coRR
{
int x=1; 0:X2=x; 2:X2=x;
}
P0            | P2            ;
LDR W0,[X2]   | MOV W0,2      ;
LDR W1,[X2]   | STR W0,[X2]   ;
~exists 0:X0=2 /\ 0:X1=1
```

Initially we have a VA x with value 1.

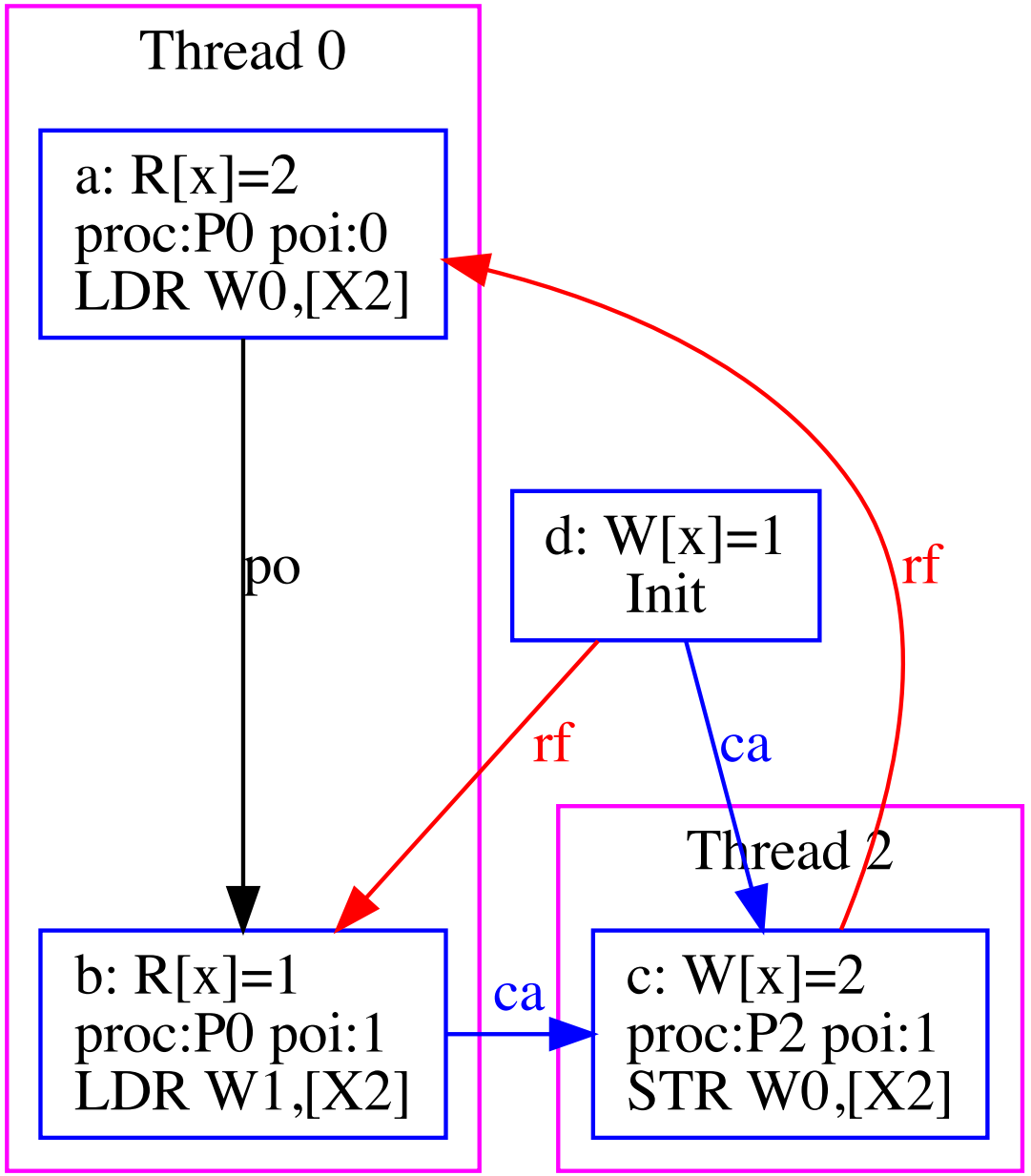
Thread P2 updates x to 2.

Thread P0 reads x twice.

It must not be the case that Thread P0 first sees the updated value 2 then the stale value 1.

A user-level example: coRR

```
AArch64 coRR
{
int x=1; 0:X2=x; 2:X2=x;
}
P0            | P2             ;
LDR W0,[X2]   | MOV W0,2       ;
LDR W1,[X2]   | STR W0,[X2]    ;
~exists 0:X0=2 /\ 0:X1=1
```



arm

coRR within the context of the OS

```
AArch64 coRR-OS
{
int x=1; 0:X2=x; 2:X2=x;
}
P0            |                | P2            ;
              |                |               ;
              |                |               ;
              |                |               ;
              |                |               ;
              |       OS       |               ;
              |                |               ;
              |                |               ;
              |                |               ;
              |                |               ;
LDR W0,[X2]   |                | MOV W0,2      ;
LDR W1,[X2]   |                | STR W0,[X2]   ;
~exists 0:X0=2 /\ 0:X1=1
```

- System software should not work in a way that breaks the user-level model.
- So even if the OS is doing some work in the background, coRR must still be forbidden.

A Copy On Write example from Linux – courtesy of Marc Zyngier

```
AArch64 CopyOnWrite
{
int x =  1; int y =  0; int z =  0;
pte_z = (oa:phy_x); 1:X7=(oa:phy_x,valid:0);
1:X6=pte_x; 1:X8=pte_y;
0:X2=x; 1:X2=x; 1:X4=y; 1:X10=z; 2:X2=x;
}
P0          | P1               | P2          ;
(* 'Break' ie invalidate 'x' VA  *)
            | STR X7,[X6]      |             ;
            | LSR X3,X2,12     |             ;
            | DSB ISH          |             ;
            | TLBI VAAE1IS,X3  |             ;
            | DSB ISH          |             ;
(* memcopy from old (through 'z')  to new *)
            | LDR W5,[X10]     |             ;
            | STR W5,[X4]      |             ;
            | DMB ISH          |             ;
(* update 'x' pte *)
LDR W0,[X2] | LDR X1,[X8]      | MOV W0,2    ;
LDR W1,[X2] | STR X1,[X6]      | STR W0,[X2] ;
~exists 0:X0=2 /\ 0:X1=1
```

Here is the sort of work that the OS (thread P1 here) could be doing in the background:

- disable all accesses to the VA of x
- copy the value of the physical location of x (via the VA z) to y
- make x an alias of y

P1's work should be transparent to P0 and P2, meaning the memory accesses of P0 and P2 should be unaffected by the remapping and copying by P1.

Breaking this down a little

```
AArch64 CopyOnWrite
{
int x =  1; int y =  0; int z =  0;
pte_z = (oa:phy_x); 1:X7=(oa:phy_x,valid:0);
1:X6=pte_x; 1:X8=pte_y;
0:X2=x; 1:X2=x; 1:X4=y; 1:X10=z; 2:X2=x;
}
P0          | P1                | P2          ;
(* 'Break' ie invalidate 'x' VA  *)
            | STR X7,[X6]       |             ;
            | LSR X3,X2,12      |             ;
            | DSB ISH           |             ;
            | TLBI VAAE1IS,X3   |             ;
            | DSB ISH           |             ;
(* memcopy from old (through 'z')  to new *)
            | LDR W5,[X10]      |             ;
            | STR W5,[X4]       |             ;
            | DMB ISH           |             ;
(* update 'x' pte *)
LDR W0,[X2] | LDR X1,[X8]       | MOV W0,2    ;
LDR W1,[X2] | STR X1,[X6]       | STR W0,[X2] ;
~exists 0:X0=2 /\ 0:X1=1
```

Initially we have a VA x with value 1, a VA y with value 0 and a VA z with value 0. The VA z is an alias of x, in other words, the PTE of z points to the same PA phy_x as the PTE of x does.

Thread P1:

- invalidates the VA x by setting the valid bit of the PTE of x to 0.
- makes a copy of the value of the PA phy_x (accessed via the VA z) into y.
- overwrites the PTE of x with the PTE of y.

Thread P2 updates x to 2.

Thread P0 reads x twice.

It must not be the case that Thread P0 first sees the updated value 2 then the stale value 1.

Explaining the synchronization in this example

```
AArch64 CopyOnWrite
{
int x =  1; int y =  0; int z =  0;
pte_z = (oa:phy_x); 1:X7=(oa:phy_x,valid:0);
1:X6=pte_x; 1:X8=pte_y;
0:X2=x; 1:X2=x; 1:X4=y; 1:X10=z; 2:X2=x;
}
P0          | P1              | P2          ;
(* 'Break' ie invalidate 'x' VA  *)
            | STR X7,[X6]     |             ;
            | LSR X3,X2,12    |             ;
            | DSB ISH         |             ;
            | TLBI VAAE1IS,X3 |             ;
            | DSB ISH         |             ;
(* memcopy from old (through 'z')  to new *)
            | LDR W5,[X10]    |             ;
            | STR W5,[X4]     |             ;
            | DMB ISH         |             ;
(* update 'x' pte *)
LDR W0,[X2] | LDR X1,[X8]     | MOV W0,2    ;
LDR W1,[X2] | STR X1,[X6]     | STR W0,[X2] ;
~exists 0:X0=2 /\ 0:X1=1
```

Initially we have a VA x with value 1, a VA y with value 0 and a VA z with value 0. The VA z is an alias of x, in other words, the PTE of z points to the same PA phy_x as the PTE of x does.

Thread P1:

- invalidates the VA x by setting the valid bit of the PTE of x to 0.
- uses a TLB synchronization sequence
- makes a copy of the value of the PA phy_x (accessed via the VA z) into y.
- uses a DMB barrier
- overwrites the PTE of x with the PTE of y.

Thread P2 updates x to 2.

Thread P0 reads x twice.

It must not be the case that Thread P0 first sees the updated value 2 then the stale value 1.

Explaining the DMB barrier

```
AArch64 CopyOnWrite
{
int x =  1; int y =  0; int z =  0;
pte_z = (oa:phy_x); 1:X7=(oa:phy_x,valid:0);
1:X6=pte_x; 1:X8=pte_y;
0:X2=x; 1:X2=x; 1:X4=y; 1:X10=z; 2:X2=x;
}
P0          | P1              | P2          ;
(* 'Break' ie invalidate 'x' VA  *)
            | STR X7,[X6]     |             ;
            | LSR X3,X2,12    |             ;
            | DSB ISH         |             ;
            | TLBI VAAE1IS,X3 |             ;
            | DSB ISH         |             ;
(* memcopy from old (through 'z')  to new *)
            | LDR W5,[X10]    |             ;
            | STR W5,[X4]     |             ;
            | DMB ISH         |             ;
(* update 'x' pte *)
LDR W0,[X2] | LDR X1,[X8]     | MOV W0,2    ;
LDR W1,[X2] | STR X1,[X6]     | STR W0,[X2] ;
~exists 0:X0=2 /\ 0:X1=1
```

The DMB barrier on P1 ensures that the STR W5,[X4] and the LDR X1,[X8] are not reordered. In other words, this ensures that when P1:

- makes a copy of the value of the PA phy_x (accessed via the VA z) into y.
- uses a DMB barrier
- overwrites the PTE of x with the PTE of y.

The copy is done ahead of the overwrite.

Definition of the DMB barrier

In cat

```
(* Barrier-ordered-before *)
let bob =
po; [dmb.full]; po
[…]
```

In the Arm ARM (see page B2-177 in J.a)

Barrier-ordered-before

A Read or Write Memory effect RW1 is Barrier-ordered-before a Read Memory or write effect RW2 from the same Observer if RW1 appears in program order before RW2 and any of the following cases apply:

RW1 appears in program order before a DMB FULL that appears in program order before RW2.

[…]

Let’s try and run this test – No TLBI sequence

```
sh CoW/run.sh -r 20k -s 500 | tee CoWNoTLBISequence.result
```

```
AArch64 CopyOnWriteNoTLBISequence
{
int x =  1; int y =  0; int z =  0;
pte_z = (oa:phy_x); 1:X7=(oa:phy_x,valid:0);
1:X6=pte_x; 1:X8=pte_y;
0:X2=x; 1:X2=x; 1:X4=y; 1:X10=z; 2:X2=x;
}
P0          | P1              | P2          ;
(* 'Break' ie invalidate 'x' VA  *)
            | STR X7,[X6]     |             ;
(* memcopy from old (through 'z')  to new *)
            | LDR W5,[X10]    |             ;
            | STR W5,[X4]     |             ;
            | DMB ISH         |             ;
(* update 'x' pte *)
LDR W0,[X2] | LDR X1,[X8]     | MOV W0,2    ;
LDR W1,[X2] | STR X1,[X6]     | STR W0,[X2] ;
exists 0:X0=2 /\ 0:X1=1
```

Explaining the TLB synchronization sequence

```
AArch64 CopyOnWrite
{
int x =  1; int y =  0; int z =  0;
pte_z = (oa:phy_x); 1:X7=(oa:phy_x,valid:0);
1:X6=pte_x; 1:X8=pte_y;
0:X2=x; 1:X2=x; 1:X4=y; 1:X10=z; 2:X2=x;
}
P0             | P1                    | P2          ;
(* 'Break' ie invalidate 'x' VA  *)
               | e: STR X7,[X6]        |             ;
               | LSR X3,X2,12           |             ;
               | ev29: DSB ISH          |             ;
               | ev31: TLBI VAAE1IS,X3  |             ;
               | ev32: DSB ISH          |             ;
(* memcopy from old (through 'z')  to new *)
               | LDR W5,[X10]           |             ;
               | STR W5,[X4]             |             ;
               | DMB ISH                 |             ;
(* update 'x' pte *)
a: LDR W0,[X2] | LDR X1,[X8]           | MOV W0,2    ;
c: LDR W1,[X2] | k: STR X1,[X6]        | STR W0,[X2] ;
~exists 0:X0=2 /\ 0:X1=1
```

For example, let's examine an execution where:

k –rf-> a

This is just one possibility, there are many more.

Explaining the TLB synchronization sequence

```
AArch64 CopyOnWrite
{
int x =  1; int y =  0; int z =  0;
pte_z = (oa:phy_x); 1:X7=(oa:phy_x,valid:0);
1:X6=pte_x; 1:X8=pte_y;
0:X2=x; 1:X2=x; 1:X4=y; 1:X10=z; 2:X2=x;
}
P0              | P1                    | P2          ;
(* 'Break' ie invalidate 'x' VA  *)
                | e: STR X7,[X6]        |             ;
                | LSR X3,X2,12           |             ;
                | ev29: DSB ISH          |             ;
                | ev31: TLBI VAAE1IS,X3  |             ;
                | ev32: DSB ISH          |             ;
(* memcopy from old (through 'z')  to new *)
                | LDR W5,[X10]           |             ;
                | STR W5,[X4]            |             ;
                | DMB ISH                |             ;
(* update 'x' pte *)
a: LDR W0,[X2]  | LDR X1,[X8]           | MOV W0,2    ;
c: LDR W1,[X2]  | k: STR X1,[X6]        | STR W0,[X2] ;
~exists 0:X0=2 /\ 0:X1=1
```

We are examining an execution where:

k –rf-> a

Suppose c takes it value from the initial state as per the final state. In this case the PTE that c uses is overwritten by k:

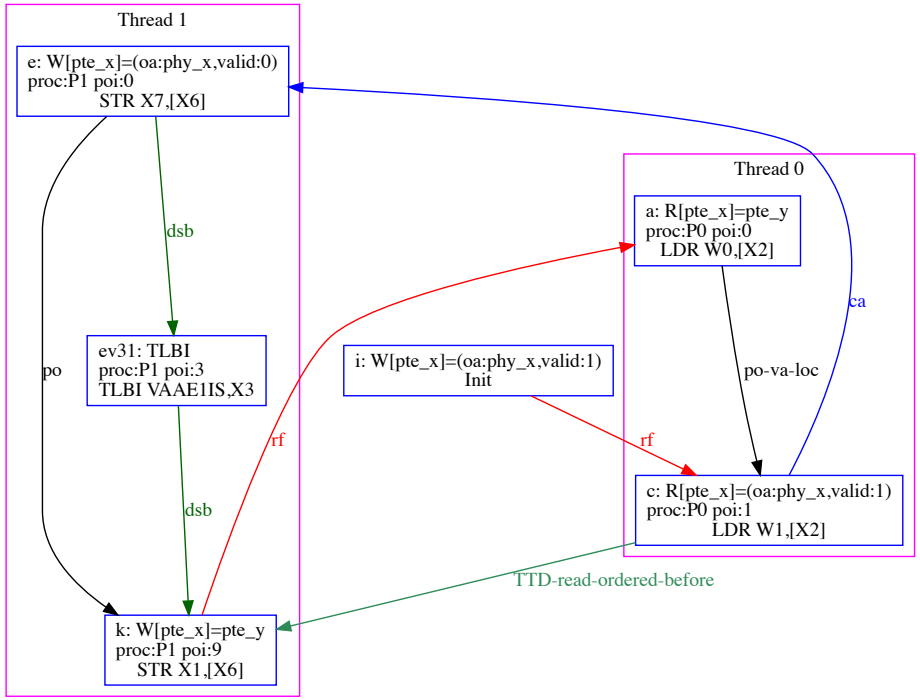
c –TTD-read-ordered-before-> k

Finally, a and c use the same VA:

a -po-va-loc-> c

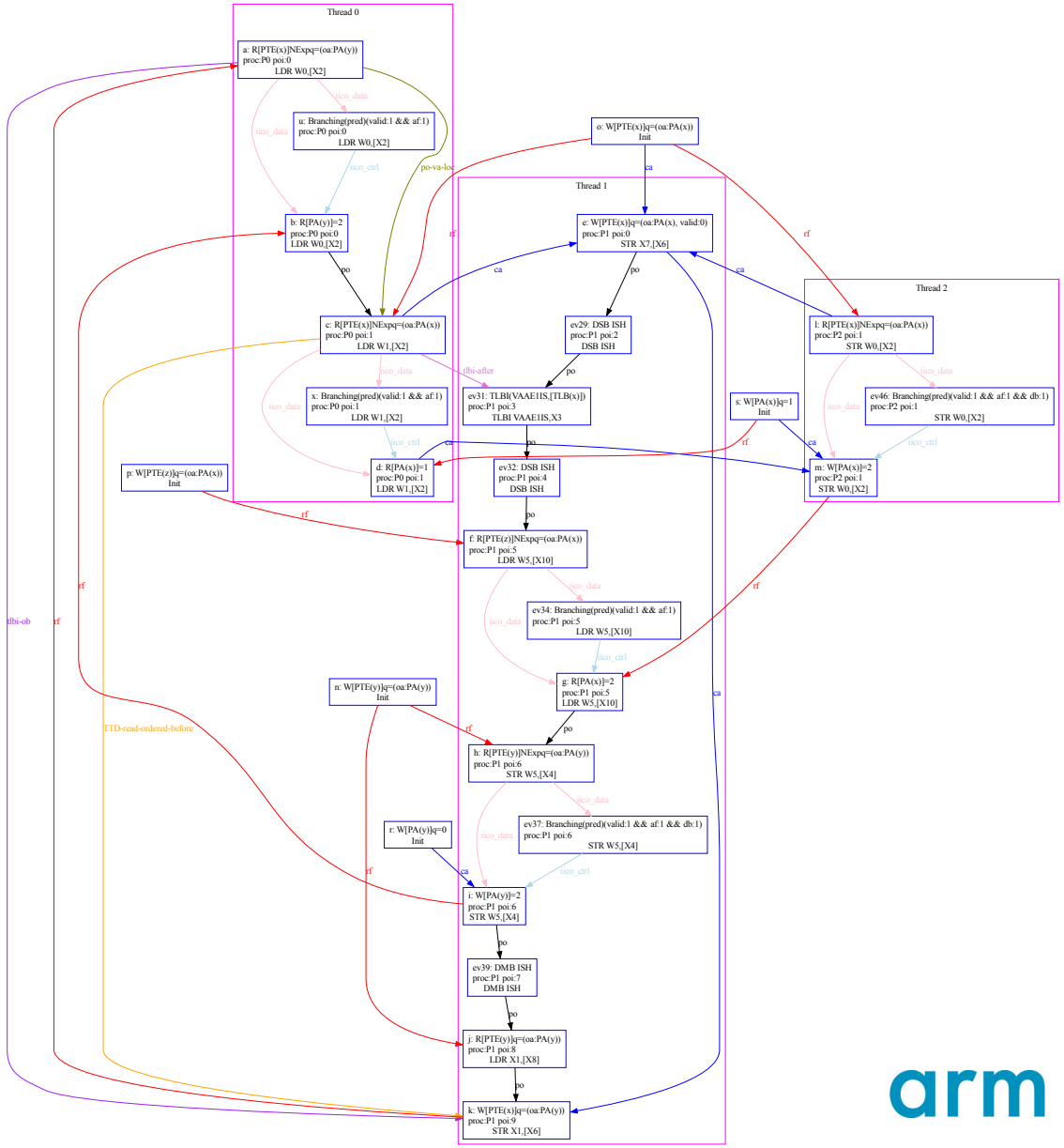
Explaining the TLB synchronization sequence

```
AArch64 CopyOnWrite
{
int x =  1; int y =  0; int z =  0;
pte_z = (oa:phy_x); 1:X7=(oa:phy_x,valid:0);
1:X6=pte_x; 1:X8=pte_y;
0:X2=x; 1:X2=x; 1:X4=y; 1:X10=z; 2:X2=x;
}
P0             | P1                    | P2              ;
(* 'Break' ie invalidate 'x' VA  *)
               | e: STR X7,[X6]        |                 ;
               | LSR X3,X2,12           |                 ;
               | ev29: DSB ISH          |                 ;
               | ev31: TLBI VAAE1IS,X3  |                 ;
               | ev32: DSB ISH          |                 ;
(* memcopy from old (through 'z')  to new *)
               | LDR W5,[X10]           |                 ;
               | STR W5,[X4]            |                 ;
               | DMB ISH                |                 ;
(* update 'x' pte *)
a: LDR W0,[X2] | LDR X1,[X8]           | MOV W0,2        ;
c: LDR W1,[X2] | k: STR X1,[X6]        | l: STR W0,[X2]  ;
~exists 0:X0=2 /\ 0:X1=1
```



Explaining the TLB synchronization sequence

```
AArch64 CopyOnWrite
{
int x =  1; int y =  0; int z =  0;
pte_z = (oa:phy_x); 1:X7=(oa:phy_x,valid:0);
1:X6=pte_x; 1:X8=pte_y;
0:X2=x; 1:X2=x; 1:X4=y; 1:X10=z; 2:X2=x;
}
P0              | P1                        | P2          ;
(* 'Break' ie invalidate 'x' VA  *)
                | e: STR X7,[X6]            |             ;
                | LSR X3,X2,12               |             ;
                | ev29: DSB ISH              |             ;
                | ev31: TLBI VAAE1IS,X3      |             ;
                | ev32: DSB ISH              |             ;
(* memcopy from old (through 'z')  to new *)
                | LDR W5,[X10]               |             ;
                | STR W5,[X4]                |             ;
                | DMB ISH                    |             ;
(* update 'x' pte *)
a: LDR W0,[X2]  | LDR X1,[X8]               | MOV W0,2    ;
c: LDR W1,[X2]  | k: STR X1,[X6]            | STR W0,[X2] ;
~exists 0:X0=2 /\ 0:X1=1
```



Explaining the TLB synchronization sequence

Our test

```
AArch64 CopyOnWrite
{
int x =  1; int y =  0; int z =  0;
pte_z = (oa:phy_x); 1:X7=(oa:phy_x,valid:0);
1:X6=pte_x; 1:X8=pte_y;
0:X2=x; 1:X2=x; 1:X4=y; 1:X10=z; 2:X2=x;
}
P0          | P1              | P2          ;
(* 'Break' ie invalidate 'x' VA  *)
            | STR X7,[X6]     |             ;
            | LSR X3,X2,12    |             ;
            | DSB ISH         |             ;
            | TLBI VAAE1IS,X3 |             ;
            | DSB ISH         |             ;
(* memcopy from old (through 'z')  to new *)
            | LDR W5,[X10]    |             ;
            | STR W5,[X4]     |             ;
            | DMB ISH         |             ;
(* update 'x' pte *)
LDR W0,[X2] | LDR X1,[X8]     | MOV W0,2    ;
LDR W1,[X2] | STR X1,[X6]     | STR W0,[X2] ;
~exists 0:X0=2 /\ 0:X1=1
```

In English (B2.3 in ARM)

TLBI-before, TLBI-after

For two effects E1 and E2 such that E1 is a completed TLBI effect and E2 is a TTD Read Memory effect RW2, one and only one of the following applies:

- E1 is TLBI-before E2.
- E2 is TLBI-before E1.

For two effects E1 and E2, E2 is TLBI-after E1 if and only if E1 is TLBI-before E2.

The TLBI-after relation enumerates all possible pairs (E1,E2) where E1 is a TLBI effect and E2 is a TTD Read Memory effect such that E2 is in the Invalidation Scope of E1, and the TLBI-after relation is asymmetric.

TTD-read-ordered-before

An effect E1 is TTD-read-ordered-before another effect E2 if […] all of the following apply:

- A TLBI effect E3 is TLBI-after E1.
- A DSB.FULL effect E4 is program order after E3.
- E2 is program order after E3.
- E2 is any effect other than an Implicit Memory effect.

[…]

TLBI-ordered-before

E1 is TLBI-ordered-before another effect E2 if any of the following apply:

- E1 is TTD-read-ordered-before E2.
- All of the following apply:
  - An Implicit TTD Read Memory effect E3 is Translation-intrinsically-before E1.
  - E3 is TTD-read-ordered-before E2.
  - E3 and E2 are from different observers.
- All of the following apply:
  - E1 is program-ordered-before E3, and E1 and E3 access the same VA.
  - E3 is TTD-read-ordered-before E2.
  - E3 and E2 are from different observers.

In cat

```
(* TLBI-after *)
include "tlbi-after.cat"
with tlbi-after from (pte-tlbi-pairs hw-reqs)

(* TTD-read-ordered-before *)
let TTD-read-ordered-before =
  tlbi-after; [TLBI]; po; [dsb.full]; po; [~(M&Imp)]
  | […]

(* TLBI-Ordered-Before *)
let tlbi-ob =
     TTD-read-ordered-before
   | tr-ib^-1; TTD-read-ordered-before & ext
   | po-va-loc; TTD-read-ordered-before & ext
```

arm

Conclusion

Correspondence between Arm cat model and Architecture Reference Manual

Section B2.3

This formalization is now in the Arm Architecture Reference Manual - Issue J.a

Nomenclature – a record of intent

In English

- “Intrinsic” relations, e.g. Intrinsic data/control dependencies and Translation-intrinsically-before: those are hardware requirements which stem from the instruction semantics
- “After” relations, e.g. Coherence-after or TLBI-after: those are relations which happen to be oriented that way in a specific execution, but could be oriented the other way in a different execution
- “Observation” relations, e.g. Observed-by or TLBI-Observed-by: build on “After” relations to describe an execution
- “Ordered” relations, e.g. Ordered-before or TLBI-Ordered-before: architectural requirements which must be respected by hardware in all executions

In cat

- Those are the “iico” relations and combinations thereof. We build the iico relations in the instruction semantics module of herd.
- Those typically require the use of the “with” cat construct to choose amongst all the possible enumerations of orderings.
- Those are built by herd from the post-condition, which determines which observations must be made in the execution of interest.
- Those are built in cat from the preceding building blocks, defined using “let” or “let rec”.

New hardware requirements

Add the following to hw-reqs

In cat

```
| tr-ib
| [M & Exp]; lob; [M & Exp | FAULT & MMU]
| DSB-ob
| CSE-ob; [TLBI]
| [CSE]; po
| [R & TTD & Imp]; po-loc; [W & TTD]
| [R & TTD & Imp]; rmw; [HU]
| (if "ETS2" then [M & Exp]; po; [TLBUncacheableFault]; tr-ib^-1; [R & Imp & TTD] else 0)
| [M & Exp]; po-loc; [TLBUncacheableFault]
```

In English

- E1 is Translation-intrinsically-before E2
- E1 is an Explicit Memory Effect and E2 is an Explicit Memory Effect or an MMU Fault Effect and E1 is Locally-ordered-before E2
- E1 is DSB-ordered-before E2
- E2 is a TLBI Effect and E1 is CSE-ordered-before E2
- E1 is a CSE Effect and E1 is program-ordered-before E2
- E1 is an Implicit TTD Memory Read R1, E2 is a TTD Memory Write W2, R1 and W2 are to the same Location and R1 is program-ordered-before W2
- E1 is an Implicit TTD Memory Read R1, E2 a Hardware Update Write of the Access Flag or the Dirty Bit W2, and R1 and W2 form a Read- Modify-Write
- ETS2 is implemented, E1 is an Explicit Memory Effect, E2 is an Implicit Read of a TTD and all of the following apply:
  - E1 is program-ordered-before a TLBUncacheable Fault Effect E3
  - E2 is Translation-intrinsically-before E3
- E1 is an Explicit Memory Effect and E2 is an TLBUncacheable Fault Effect, E1 and E2 are to the same Location and E1 is program-ordered-before E2.

New observation requirements

Add the following to ob

In cat

```
| rf & int & (_ * R & TTD & Imp)
| rf & ext & (_ * R & TTD & Imp)
| tlbi-ca
| TLBUncacheable-pred
| HU-pred
| [HU]; co | co; [HU]
| tlbi-ob
```

In English

- E2 is an Implicit TTD Read Effect, E1 and E2 are from the same Observer and E2 Reads-from E1
- E2 is an Implicit TTD Read Effect, E1 and E2 are from different Observers and E2 Reads-from E1
- E2 is TLBI-Coherence-after E1
- E1 is a TLBUncacheable-Predecessor of E2
- E1 is a Hardware-Update-Predecessor of E2
- E1 is Coherence-before E2, and either E1 or E2 is a Hardware Update
- E1 is TLBI-Ordered-before E2

There’s plenty more to talk about

- Other bits such as AccessFlag, DirtyBit
- Hardware management of those bits
- TLBUncacheable
- ETS2
- …

arm

Explaining the TLB synchronization sequence

```
AArch64 CopyOnWrite
{
int x =  1; int y =  0; int z =  0;
pte_z = (oa:phy_x); 1:X7=(oa:phy_x,valid:0);
1:X6=pte_x; 1:X8=pte_y;
0:X2=x; 1:X2=x; 1:X4=y; 1:X10=z; 2:X2=x;
}
P0             | P1                    | P2          ;
(* 'Break' ie invalidate 'x' VA  *)
               | e: STR X7,[X6]        |             ;
               | LSR X3,X2,12           |             ;
               | ev29: DSB ISH          |             ;
               | ev31: TLBI VAAE1IS,X3  |             ;
               | ev32: DSB ISH          |             ;
(* memcopy from old (through 'z')  to new *)
               | LDR W5,[X10]           |             ;
               | STR W5,[X4]             |             ;
               | DMB ISH                 |             ;
(* update 'x' pte *)
a: LDR W0,[X2] | LDR X1,[X8]           | MOV W0,2    ;
c: LDR W1,[X2] | k: STR X1,[X6]        | STR W0,[X2] ;
~exists 0:X0=2 /\ 0:X1=1
```

We are examining an execution where:

k –rf-> a

This is just one possibility, there are many more.

In this case, the forbidding cycle is:

a -tlbi-ob-> k -> rf -> a

Where a is tlbi-ob k because

a -po-va-loc-> c -TTD-read-ordered-before-> k

And c is TTD-read-ordered-before k because:

c -tlbi-after-> ev31 -po-> ev32 -po-> k

And c -tlbi-after-> ev31 because:

- tlbi-after is total over TLBI effects and TTD Read Memory effect
- if it was the other way around, we would have a cycle:

  ev31 -tlbi-ca-> e -DSB-ob-> ev31

  (i.e. ev31 -tlbi-after-> c -ca-> e -po-> ev29 -po-> ev31)

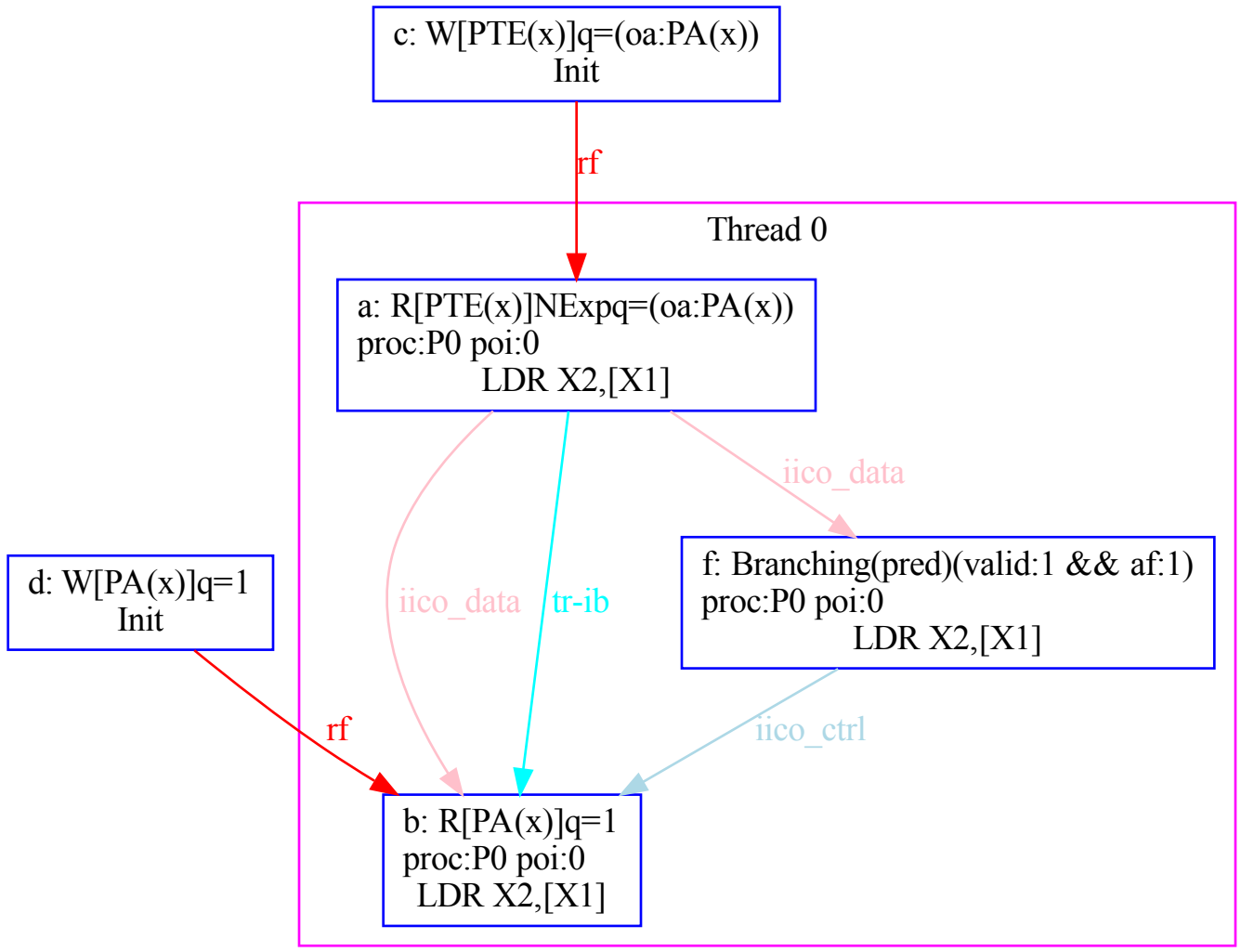
arm

Semantics of instructions

Semantics of LDR

With page table considerations

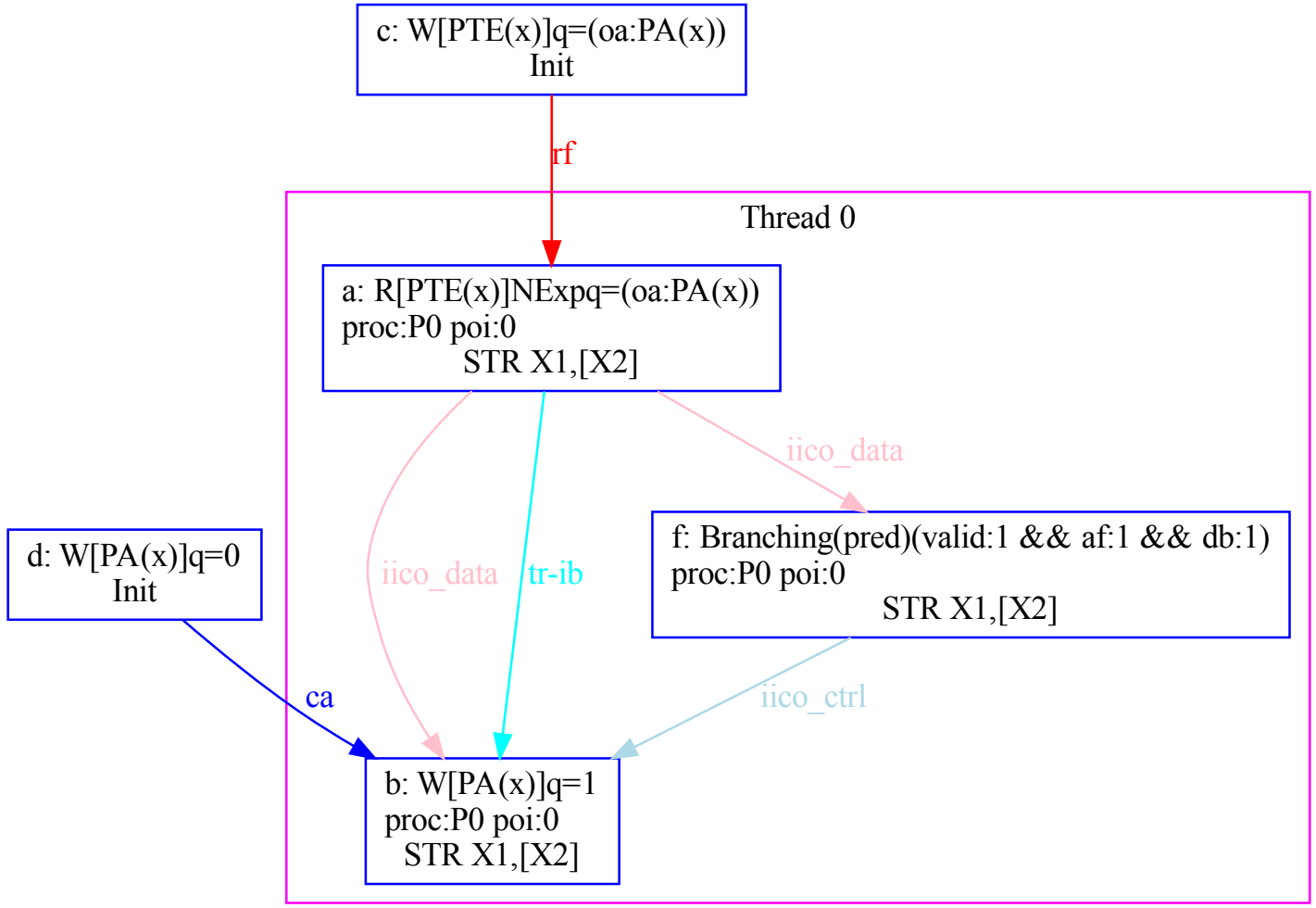
```
AArch64 LDR
{
0:X2=x;
0:X1=0;
x=1;
}
P0          ;
LDR X1,[X2];
exists(0:X1=1)
```



Semantics of STR

With page table considerations

```
AArch64 STR
{
0:X1=0;
0:X2=x;
}
P0;
MOV X1,#1;
STR X1,[X2];
exists(x=1)
```



PTE values

A page table entry value such as `PTE(x)` is a tuple (`oa`, `valid`, `af`, `db`, `dbm`), where:

- `oa` is the “output address”
- `valid` is the valid bit
- `af` is the Access Flag
- `db` is the Dirty Bit
- `dbm` is the Dirty Bit Management bit

Extended litmus test format

PTE values and faults

New syntax

- The initial state can now refer to a new type of value, which corresponds to page table entries. For example in the initial state for MP-pte, we have: `pte_x = (valid:0,oa:phy_x)`. In words, this says that the page table entry for `x` is initialized so that its `valid` bit is 0, and so that it points to a physical address `phy_x`
- In the final clause, we can now specify whether an access is expected to fault. We can do this using labels in the body of the test (see label `L0` on thread P1). For example, in the final clause for MP-pte, we have: `fault(P1:L0,x)`. In words, this asks whether the access at line `L0` on thread P1 can provoke a fault relative to address `x`.

An example litmus test

```
AArch64 LDRv0
{
 PTE(x)=(valid:0);
 0:X2=x;
}
P0                ;
 L0:LDR W1,[X2] ;
forall(fault(P0:L0,x))
```

LDRv0 in words

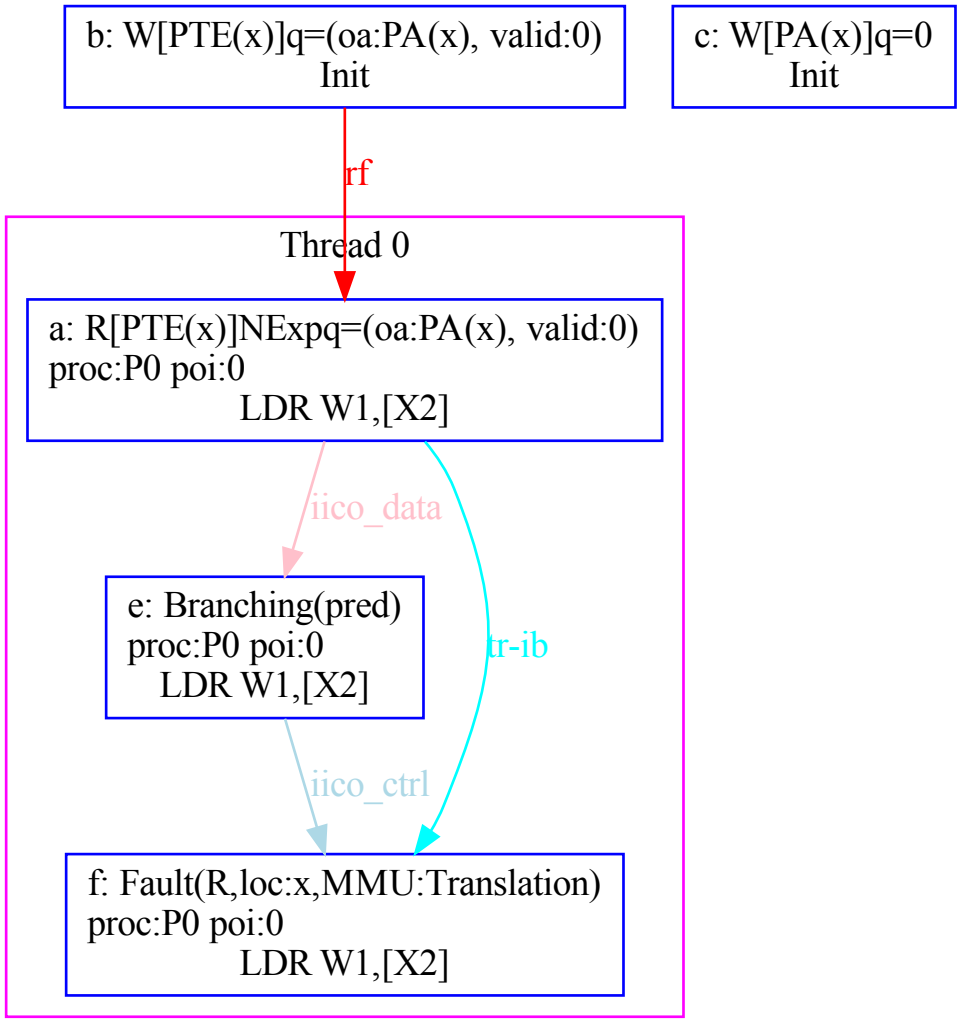
- The initial value for `x` is 1, and initially the page table entry for `x` has the valid bit `valid` to 0. Since the other values are omitted, this entails that `pte_x` is invalid and points to `phy_x`.
- At line `L0` the thread P0 does a load of `x`.
- The final clause states that it must always be the case (see keyword “`forall`”) that: there is a fault at line `L0` relative to address `x`.

arm

Semantics of LDR

When the valid bit is 0

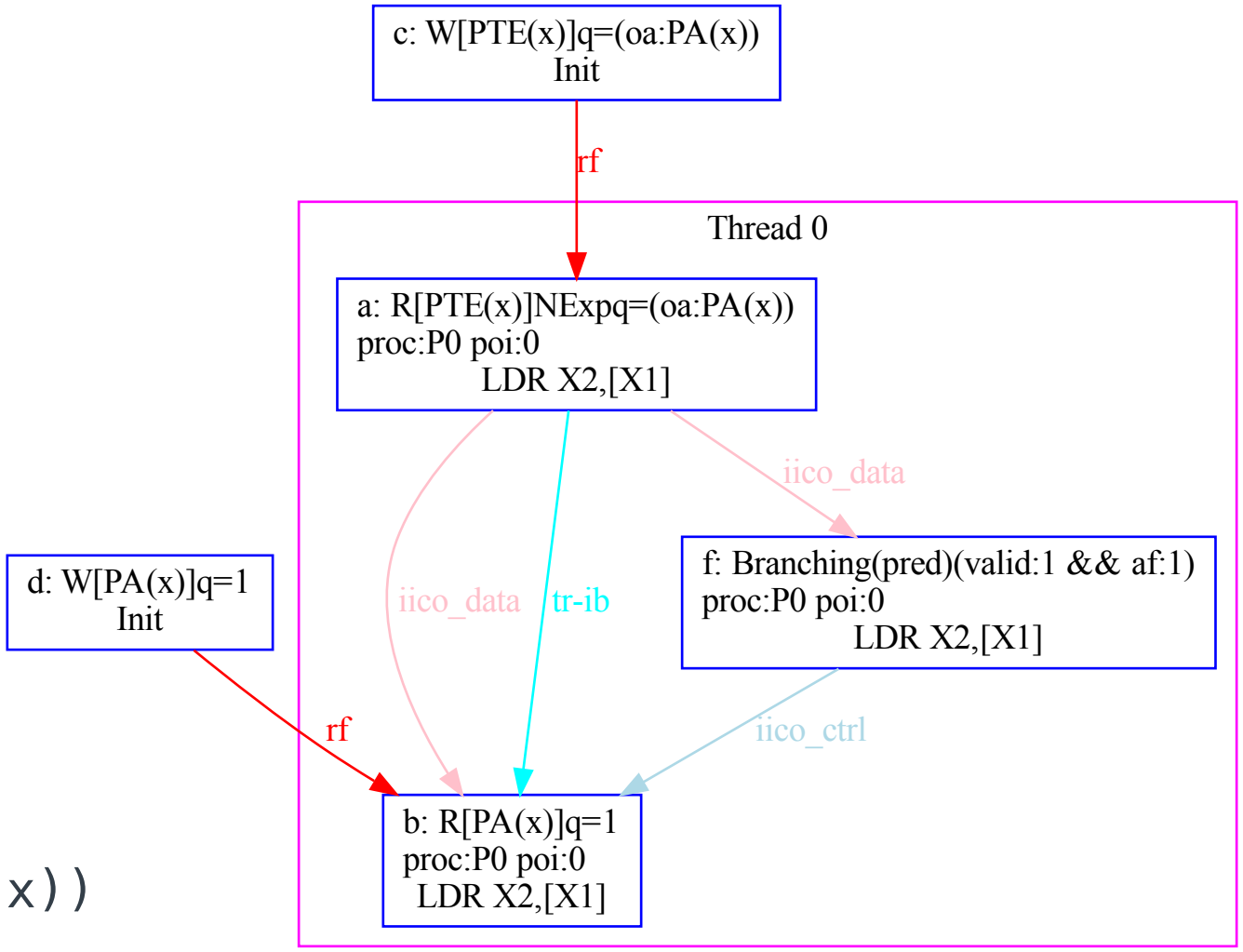
```
AArch64 LDRv0
{
 PTE(x)=(valid:0);
 0:X2=x;
}
P0                ;
 L0:LDR W1,[X2] ;
forall(fault(P0:L0,x))
```



Semantics of LDR

When the valid bit is 1

```
AArch64 LDRv1
{
 PTE(x)=(valid:1);
 x=1;
 0:X1=x;
}
 P0           ;
L0:           ;
 LDR X2,[X1] ;
forall(0:X2=1 /\ ~fault(P0:L0,x))
```



Extended litmus test format

Access flag, dirty bit and hardware management

New syntax

- A litmus test can now specify whether ARMv8.1-TTHM (aka FEAT_HAFDBS) is implemented, and whether TCR_ELx.{HA, HD} are equal to 1 via metadata at the top of the test. By default, none of those are equal to 1.
- The default value for pte_x is (oa:phy_x, valid:1, af:1, db:1, dbm:0).
- If certain fields are omitted (for example in the initial state or the exists clause), they are assumed to have their default values.

An example litmus test

```
AArch64 LDRaf0-H
TTHM=P0:HA
{
 [PTE(x)]=(oa:PA(x),af:0);
 x=1;
 0:X2=x;
}
 P0           ;
L0:           ;
 LDR W1,[X2]  ;
forall(0:X1=1 /\
~fault(P0:L0,x))
```

LDRaf0-HA in words

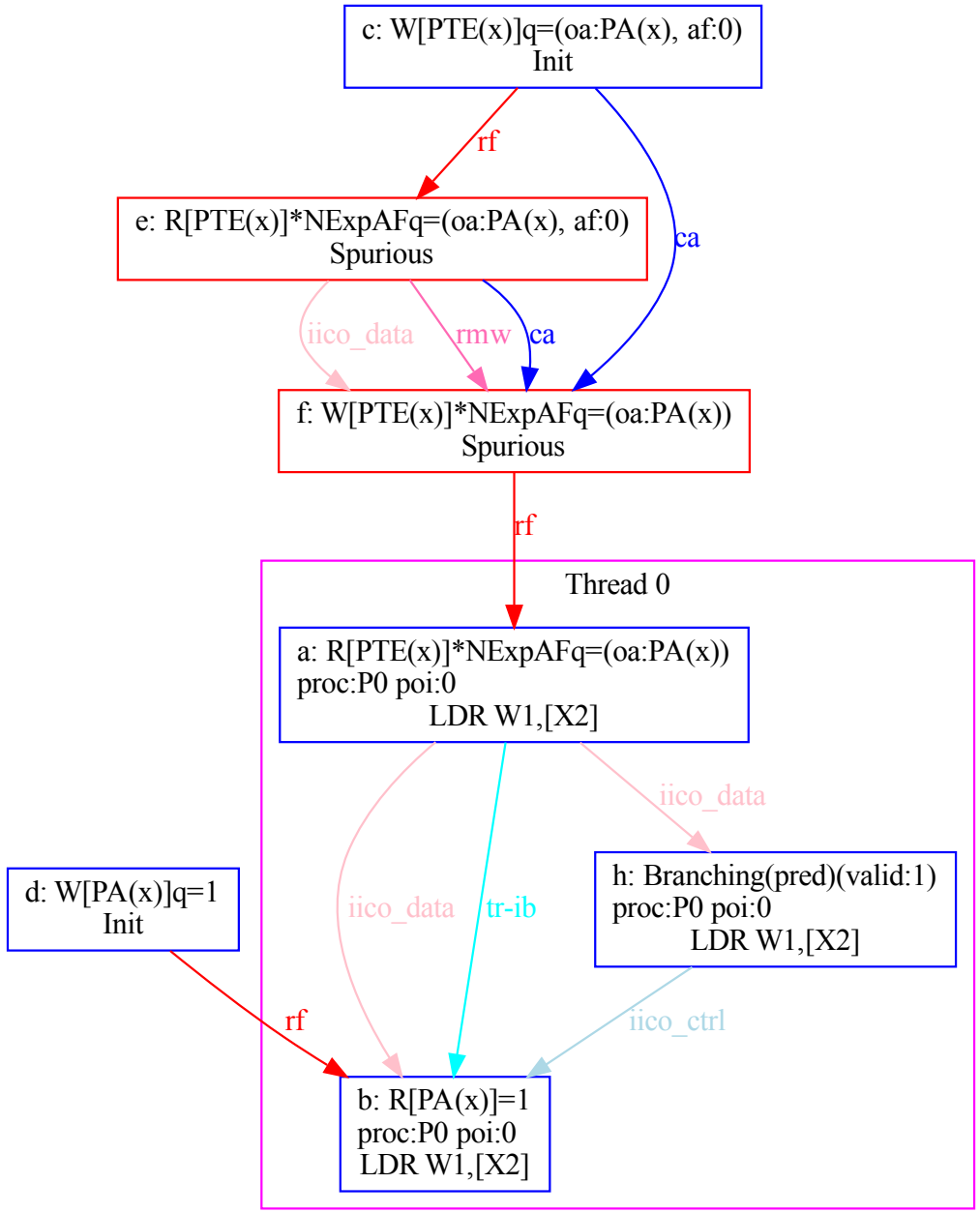
- The test assumes that ARMv8.1-TTHM is implemented, and states that TCR_ELx.HA is equal to 1.
- The initial value for x is 1, and initially the page table entry for x has the Access Flag af set to 0. Since the other values are omitted, this entails that pte_x is valid and points to phy_x.
- At line L0 the thread P0 does a load of x.
- The final clause states that it must always be the case (see keyword "forall") that:
  - The register X2 holds the value 1
  - The page table entry for x now has the Access Flag af set to 1
  - There was no fault at line L0 relative to address x.

arm

Semantics of LDR

When af:0 and TTHM HA is on (SPURIOUS case)

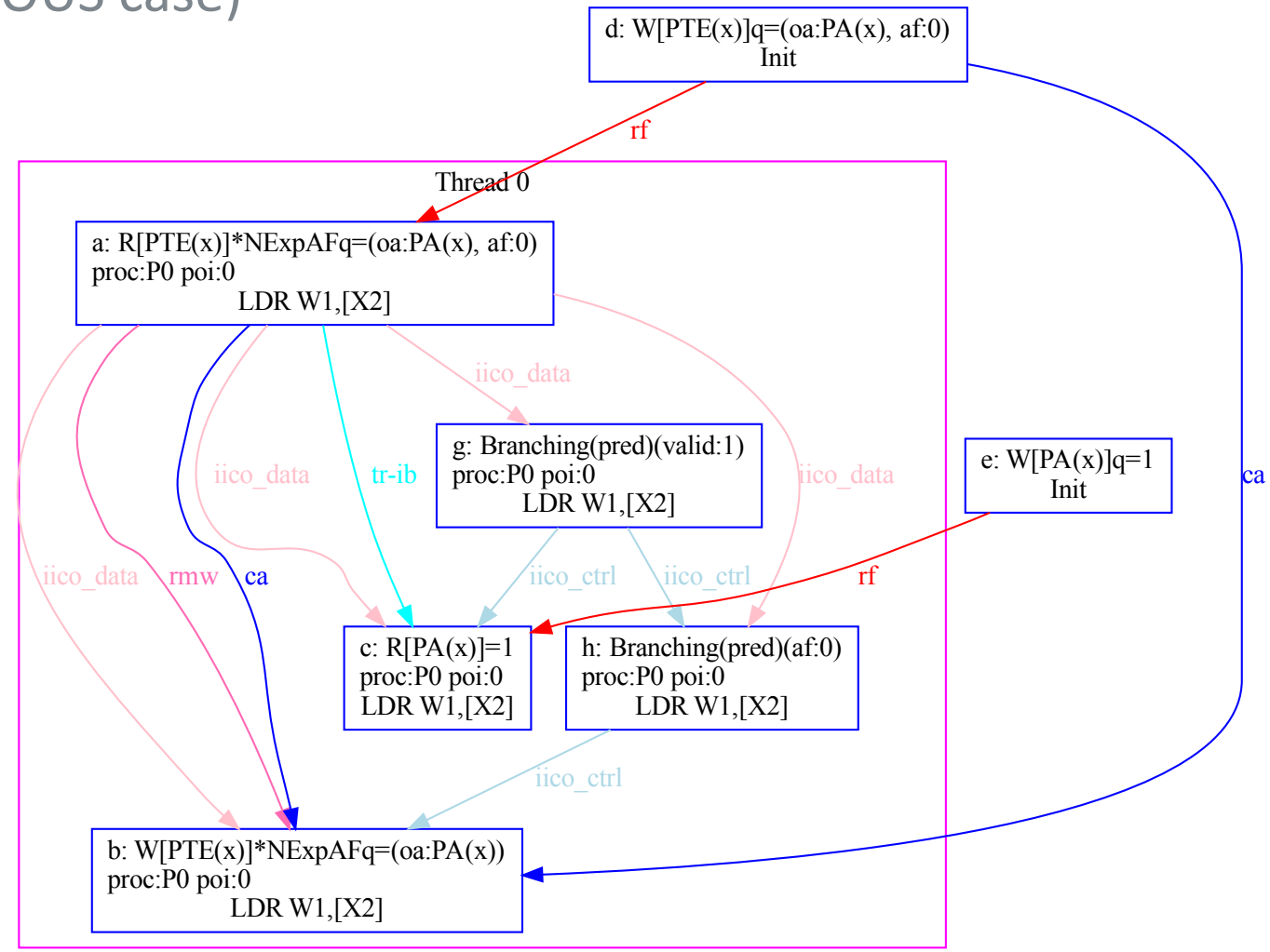
```
AArch64 LDRaf0-HA
TTHM=P0:HA
{
 [PTE(x)]=(oa:PA(x),af:0);
 x=1;
 0:X2=x;
}
 P0            ;
L0:            ;
 LDR W1,[X2]   ;
forall(0:X1=1 /\ ~fault(P0:L0,x))
```



Semantics of LDR

When af:0 and TTHM HA is on (non SPURIOUS case)

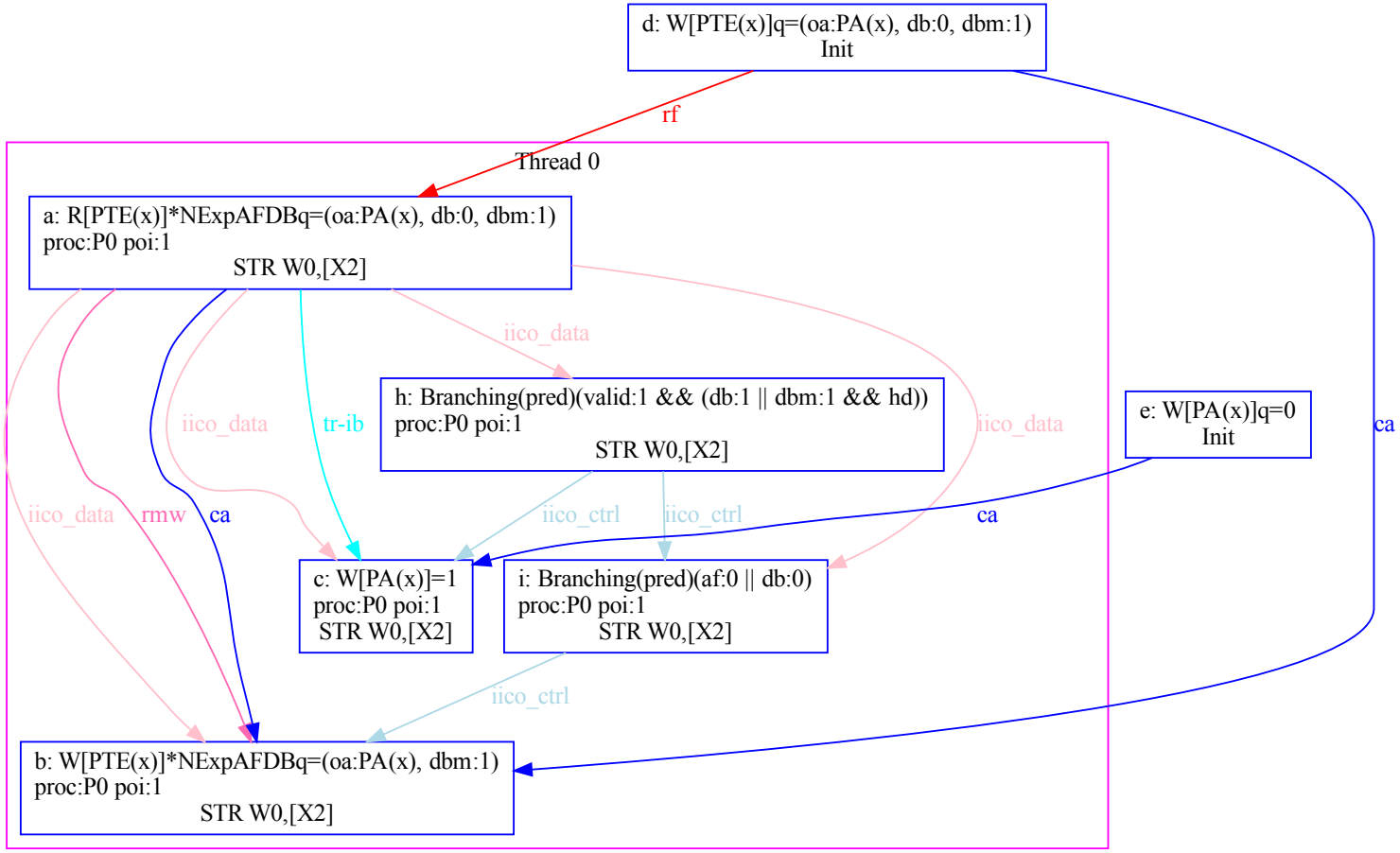
```
AArch64 LDRaf0-HA
TTHM=P0:HA
{
 [PTE(x)]=(oa:PA(x),af:0);
 x=1;
 0:X2=x;
}
 P0            ;
L0:            ;
 LDR W1,[X2]   ;
forall(0:X1=1 /\ ~fault(P0:L0,x))
```



Semantics of STR

When af:1, db:0, dbm:1 and HD is on

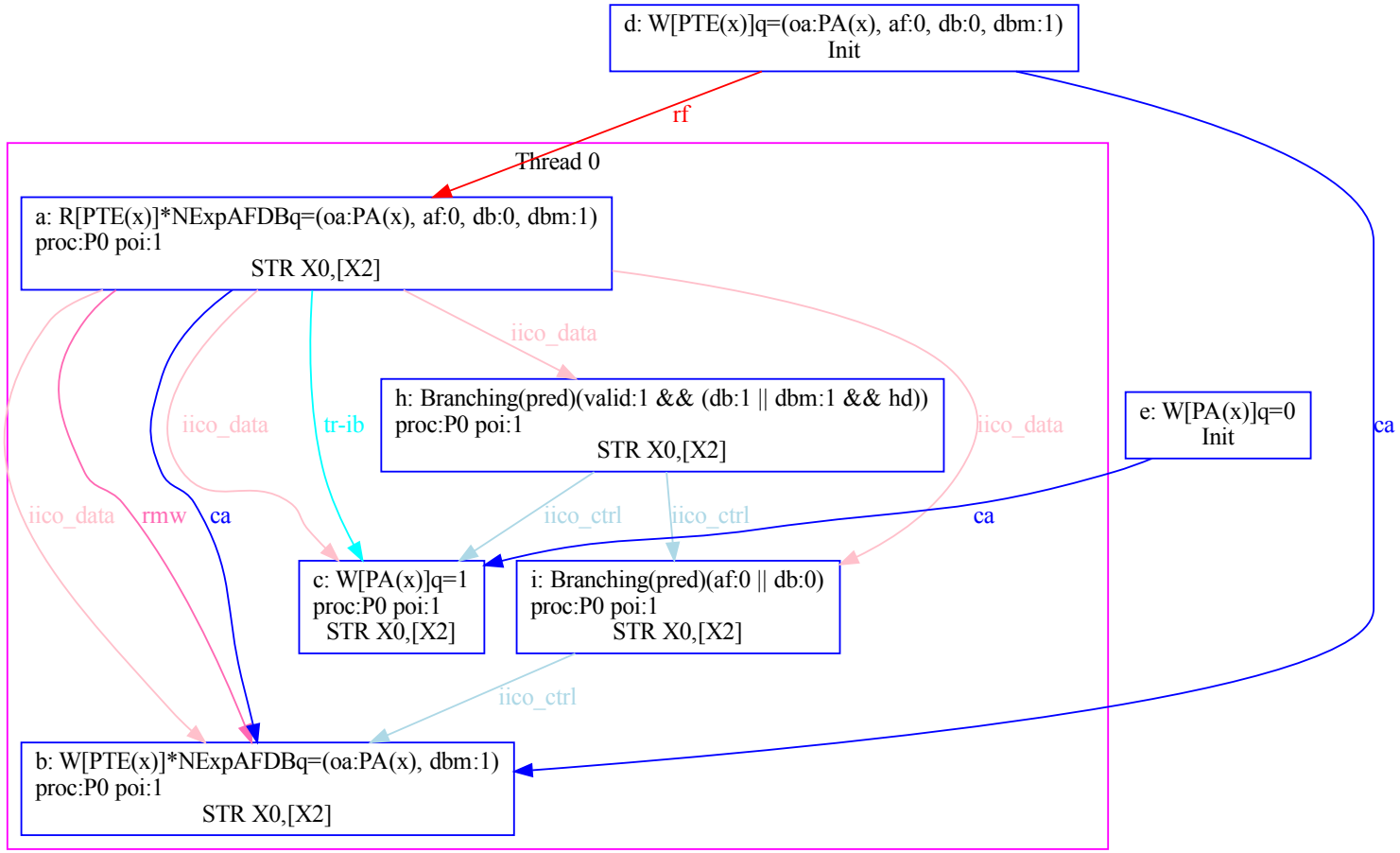
```
AArch64 STR-db0+dbm1-HD
TTHM=HA HD
{
 PTE(x)=(db:0,dbm:1);
 0:X2=x;
}
 P0           ;
 MOV W0,#1    ;
 STR W0,[X2] ;
forall(x=1 /\ ~fault(P0,x))
```



Semantics of STR

When af:0 and HA is on, db:0, dbm:1 and HD is on (non SPURIOUS case)

```
AArch64 STR-af0db0+dbm1+HA+HD
TTHM=HA HD
{
 [PTE(x)]=(af:0,db:0,dbm:1);
 0:X2=x;
}
 P0           ;
 MOV X0,#1    ;
 STR X0,[X2] ;
forall(x=1 /\ ~fault(P0,x))
```



arm

Ordering requirements

Nomenclature – a tentative record of intent

In English

- “Intrinsic” relations, e.g. Intrinsic data/control dependencies and Translation-intrinsically-before: those are hardware requirements which stem from the instruction semantics
- “After” relations, e.g. Coherence-after or TLBI-after: those are relations which happen to be oriented that way in a specific execution, but could be oriented the other way in a different execution
- “Observation” relations, e.g. Observed-by or TLBI-Observed-by: build on “After” relations to describe an execution
- “Ordered” relations, e.g. Ordered-before or TLBI-Ordered-before: architectural requirements which must be respected by hardware in all executions

In cat

- Those are the “iico” relations and combinations thereof. We build the iico relations in the instruction semantics module of herd.
- Those typically require the use of the “with” cat construct to choose amongst all the possible enumerations of orderings.
- Those are built by herd from the post-condition, which determines which observations must be made in the execution of interest.
- Those are built in cat from the preceding building blocks, defined using “let” or “let rec”.

Translation-intrinsically-before

In cat

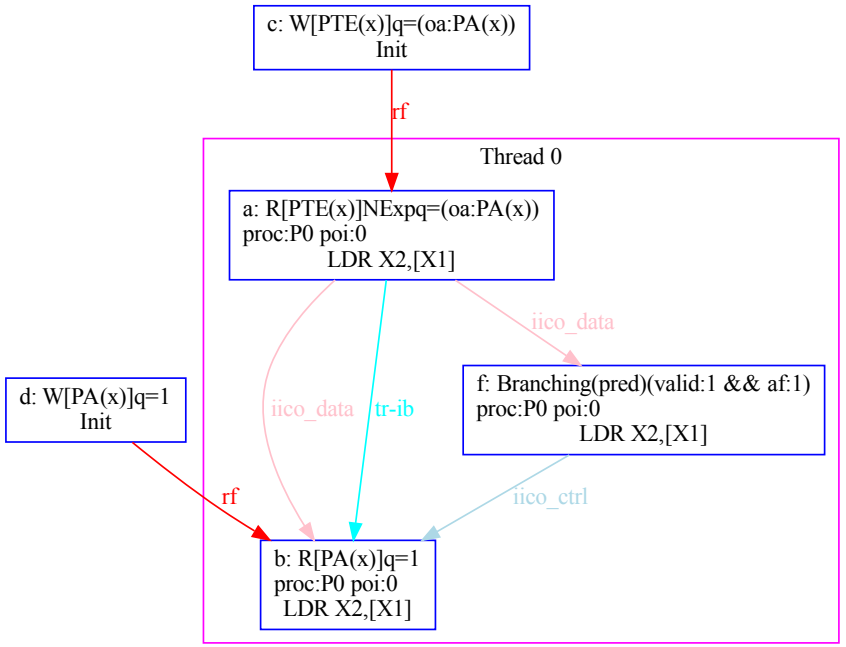
```
let tr-ib =
[R & PTE & Imp]; iico_data; [B]; iico_ctrl; [M & Exp | FAULT]
```

In English

An Implicit PTE Read Memory Effect R1 is Translation-intrinsically-before an Explicit Read or Write Memory or a Fault Effect E2 if and only if all of the following applies:

- R1 is before a Branching Effect B3 in the Intrinsic Data dependency order, and
- B3 is before E2 in the Intrinsic Control dependency order.

Illustration: semantics of LDR



arm

New hardware requirements

New hardware requirements

Add the following to hw-reqs

In cat

```
| tr-ib
| [M & Exp]; lob; [M & Exp | FAULT & MMU]
| DSB-ob
| CSE-ob; [TLBI]
| [CSE]; po
| [R & TTD & Imp]; po-loc; [W & TTD]
| [R & TTD & Imp]; rmw; [HU]
| (if "ETS2" then [M & Exp]; po; [TLBUncacheableFault]; tr-ib^-1; [R & Imp & TTD] else 0)
| [M & Exp]; po-loc; [TLBUncacheableFault]
```

In English

- E1 is Translation-intrinsically-before E2
- E1 is an Explicit Memory Effect and E2 is an Explicit Memory Effect or an MMU Fault Effect and E1 is Locally-ordered-before E2
- E1 is DSB-ordered-before E2
- E2 is a TLBI Effect and E1 is CSE-ordered-before E2
- E1 is a CSE Effect and E1 is program-ordered-before E2
- E1 is an Implicit TTD Memory Read R1, E2 is a TTD Memory Write W2, R1 and W2 are to the same Location and R1 is program-ordered-before W2
- E1 is an Implicit TTD Memory Read R1, E2 a Hardware Update Write of the Access Flag or the Dirty Bit W2, and R1 and W2 form a Read- Modify-Write
- ETS2 is implemented, E1 is an Explicit Memory Effect, E2 is an Implicit Read of a TTD and all of the following apply:
  - E1 is program-ordered-before a TLBUncacheable Fault Effect E3
  - E2 is Translation-intrinsically-before E3
- E1 is an Explicit Memory Effect and E2 is an TLBUncacheable Fault Effect, E1 and E2 are to the same Location and E1 is program-ordered-before E2.

New hardware requirements

Systematic review

```
| tr-ib
| [M & Exp]; (lob | pick-lob); [M & Exp | FAULT & MMU]
| DSB-ob
| CSE-ob; [TLBI]
| [CSE]; po
| [R & TTD & Imp]; po-loc; [W & TTD]
| [R & TTD & Imp]; rmw; [HU]
| (if "ETS2" then [M & Exp]; po; [TLBUncacheableFault]; tr-ib^-1; [R & Imp & TTD] else 0)
| [M & Exp]; po-loc; [TLBUncacheableFault]
```

Translation-intrinsically-before in Ordered-before

Locally-ordered-before to MMU Faults

DSB semantics

CSE semantics I

CSE semantics II

No implicit read from the future

Sanity check on Hardware Updates

ETS2

Partial parity with ETS2 for TLBI cases

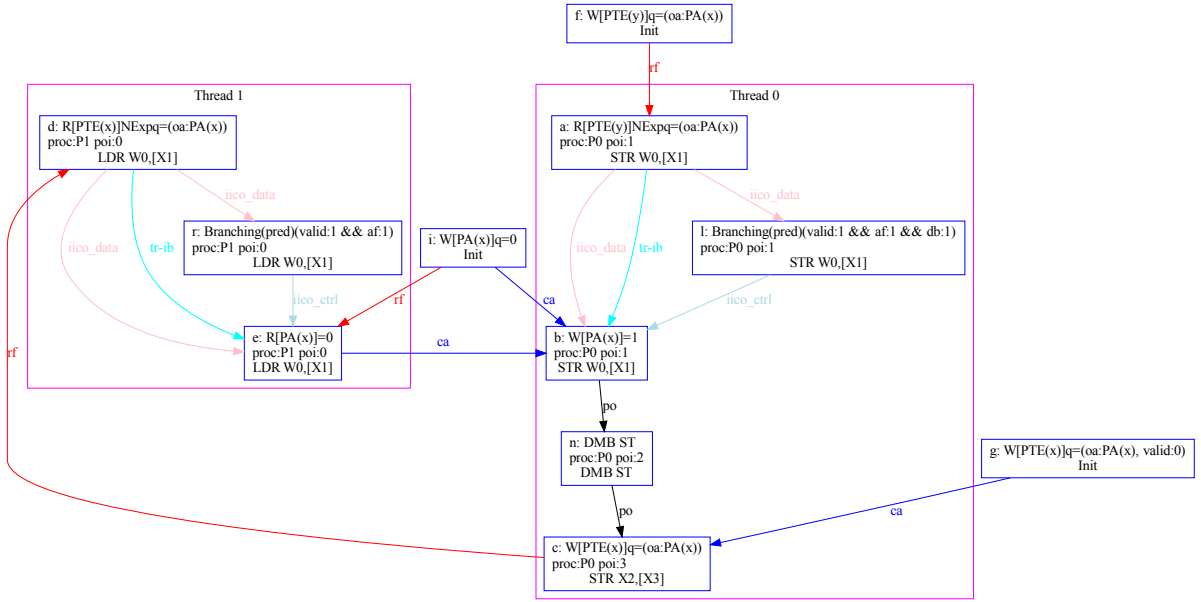
Translation-intrinsically-before in Ordered-before

tr-ib; [M & Exp]

Test (Forbidden)

```
AArch64 MP-x-ptex+dmb.st
{
 [PTE(y)]=(oa:PA(x),valid:1);
 [PTE(x)]=(oa:PA(x),valid:0);
 0:X1=y;
 0:X2=(oa:PA(x),valid:1); 0:X3=PTE(x);
 1:X1=x;
}
P0            | P1            ;
 MOV W0,#1    |L0:            ;
 STR W0,[X1]  |  LDR W0,[X1]  ;
 DMB ST       |               ;
 STR X2,[X3]  |               ;
exists(~fault(P1:L0,x) /\ 1:X0=0)
```

Forbidden execution



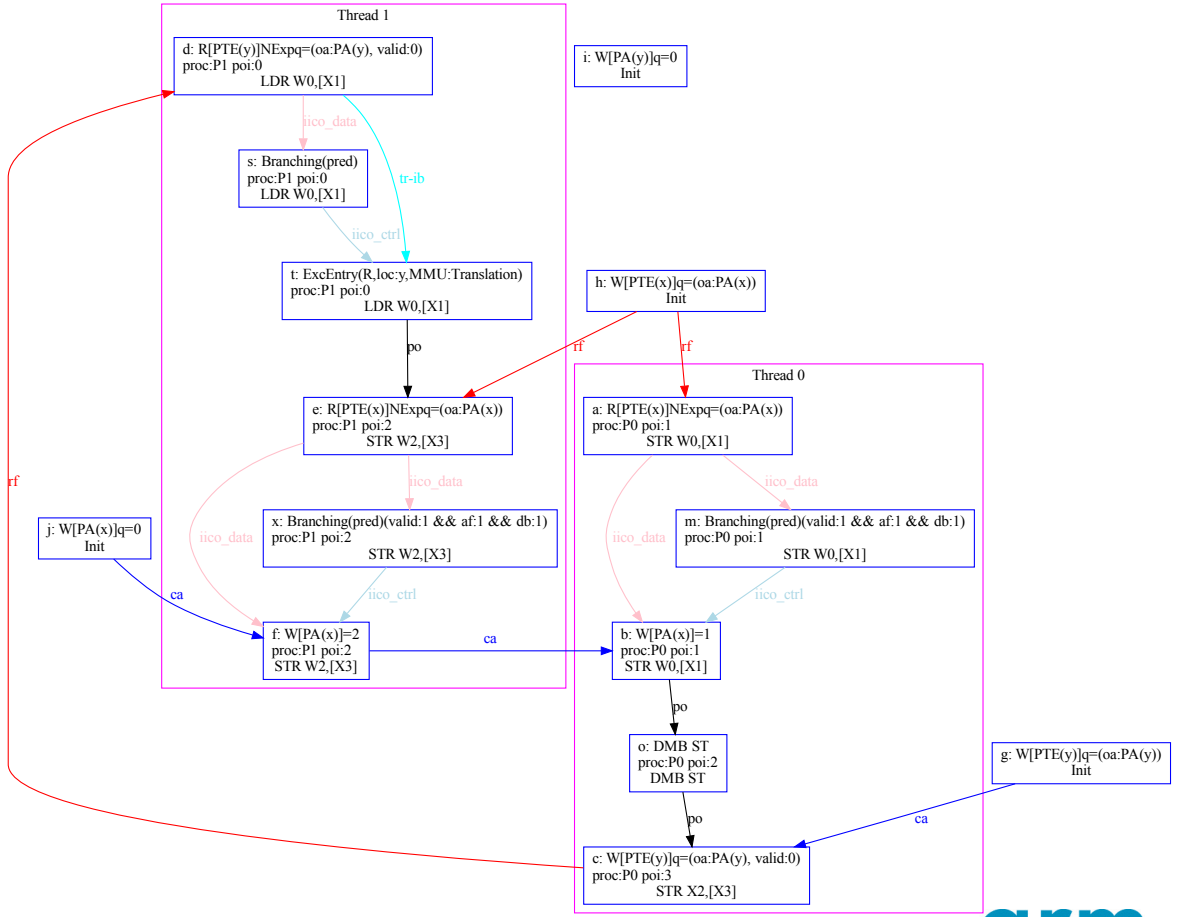
Translation-intrinsically-before in Ordered-before

tr-ib; [FAULT & MMU]

Test (Forbidden)

```
AArch64 S-pte+dmb.st+fault
{
0:X1=x;
0:X2=(oa:PA(y),valid:0); 0:X3=PTE(y);
1:X1=y; 1:X3=x;
}
 P0           | P1              | P1.F           ;
 MOV W0,#1    |L0:              | MOV W2,#2      ;
 STR W0,[X1]  | LDR W0,[X1]     | STR W2,[X3]    ;
 DMB ST       |                 |                ;
 STR X2,[X3]  |                 |                ;
 exists(1:X2=2 /\ x=1)
```

Forbidden execution



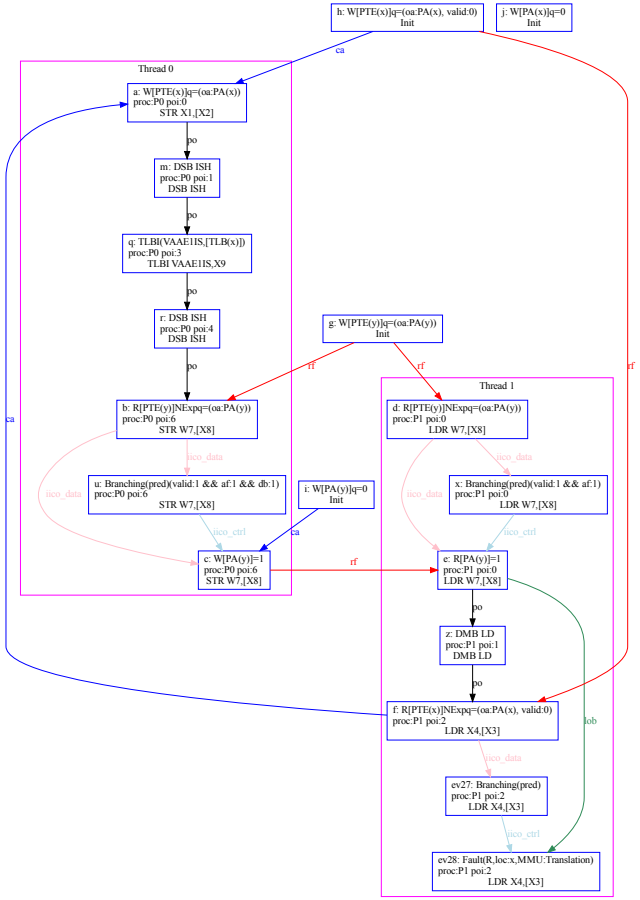
Locally-ordered-before to MMU Faults

[M & Exp]; lob; [FAULT & MMU]

Test (Forbidden)

```
AArch64 D15347-M-load-shoot+DMB.LD
{
 [PTE(x)]=(valid:0);
 0:X1=(oa:PA(x)); 0:X2=PTE(x);
 0:X3=x; 0:X8=y;
 1:X3=x; 1:X8=y;
}
 P0                | P1;
 STR X1,[X2]       | LDR W7,[X8]    ;
 DSB ISH           | DMB LD         ;
 LSR X9,X3,#12     |L0:             ;
 TLBI VAAE1IS,X9   | LDR X4,[X3]    ;
 DSB ISH           |                ;
 MOV W7,#1         |                ;
 STR W7,[X8]       |                ;
exists(1:X7=1 /\ fault(P1:L0,x))
```

Forbidden execution



DSB-Ordered-before

In cat

```
let DSB-ob =
 [M]; po; [dsb.full]; po; [~(M & Imp)]
 | [R & Exp]; po; [dsb.ld]; po; [~(M & Imp)]
 | [W & Exp]; po; [dsb.st]; po; [~(M & Imp)]
```

In English

An Effect E1 is DSB-ordered-before another Effect E2 if one of the following applies:

- E1 is a Memory Effect, E2 is a Memory Effect except an Implicit Effect, E1 is program-order-before an Effect E3 generated by a DSB.FULL instruction and E3 is program-order-before E2
- E1 is an Explicit Read Memory Effect R1, E2 is a Memory Effect except an Implicit Effect, R1 is program-order-before an Effect E3 generated by a DSB.LD instruction and E3 is program-order-before E2
- E1 is an Explicit Write Memory Effect W1, E2 is a Memory Effect except an Implicit Effect, W1 is program-order-before an Effect E3 generated by a DSB.ST instruction and E3 is program-order-before E2

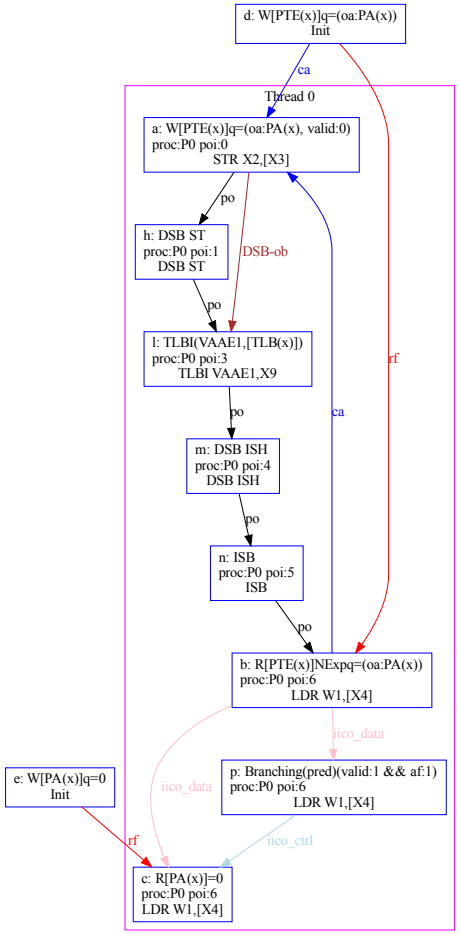
DSB semantics

DSB-ob; [TLBI]

Test (Forbidden - could use a DSB ISH instead)

```
AArch64 V2I-W-DSB.ST-TLBI-DSB.ISH-ISB-R
{
 [PTE(x)]=(oa:PA(x),valid:1);
 0:X2=(oa:PA(x),valid:0); 0:X3=PTE(x);
 0:X4=x;
}
 P0              ;
 STR X2,[X3]     ;
 DSB ST          ;
 LSR X9,X4,#12   ;
 TLBI VAAE1,X9   ;
 DSB ISH         ;
 ISB             ;
L0:              ;
 LDR W1,[X4]     ;
exists(~fault(P0:L0,x))
```

Forbidden execution



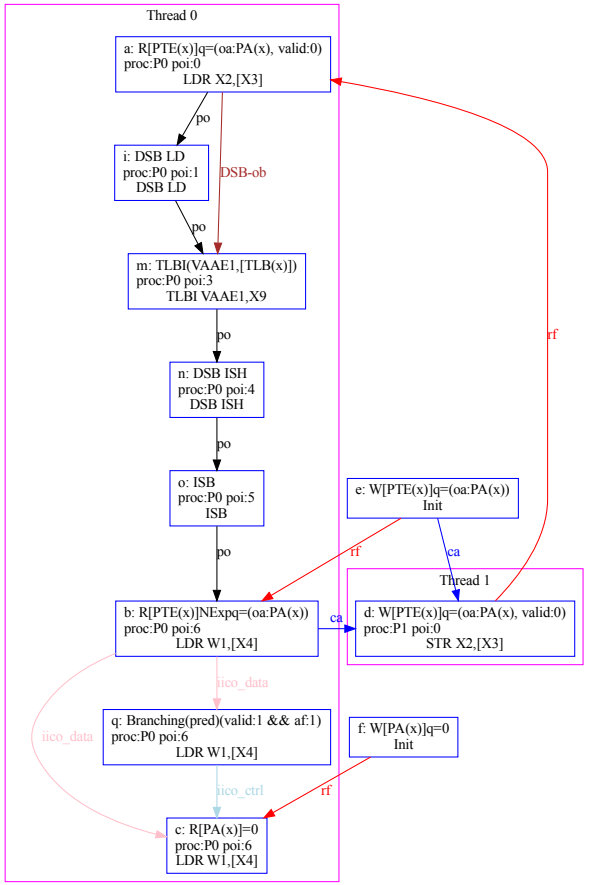
DSB semantics

DSB-ob; [TLBI]

Test (Forbidden - could use a DSB ISH instead)

```
AArch64 V2I-R-DSB.LD-TLBI-DSB.ISH-ISB-R+W
{
 PTE(x)=(oa:PA(x),valid:1);
 0:X3=PTE(x);
 0:X4=x;
 1:X2=(oa:PA(x),valid:0); 1:X3=PTE(x);
}
 P0              | P1          ;
 LDR X2,[X3]     | STR X2,[X3] ;
 DSB LD          |             ;
 LSR X9,X4,#12   |             ;
 TLBI VAAE1,X9   |             ;
 DSB ISH         |             ;
 ISB             |             ;
L0:              |             ;
 LDR W1,[X4]     |             ;
exists(0:X2=(oa:PA(x),valid:0) /\ ~fault(P0:L0,x))
```

Forbidden execution



CSE-Ordered-before

In cat

```
let CSE = ISB | EXC-ENTRY | EXC-RET
let CSE-ob = [R & Exp]; ctrl; [CSE]; po
```

In English

A Context Synchronisation Event generates a Context Synchronisation effect (CSE effect). For example, an ISB instruction generates an ISB Effect, which is a CSE effect. Exception Entry and Exception Return Effects also are CSE effects.

An Effect E1 is CSE-ordered-before another Effect E2 if E1 is an Explicit Read Effect and there exists a CSE E3 such that:

- E1 has a control dependency to E3
- E3 is program-order-before E2

arm

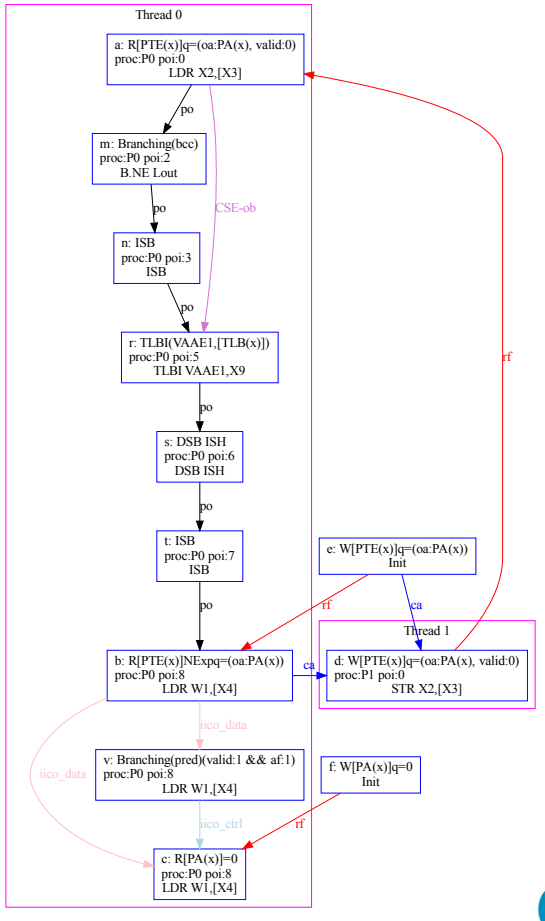
CSE semantics I

CSE-ob; [TLBI] (* Instead of using a DMB before the TLBI, we can use a branch+ISB *)

Test (Forbidden)

```
AArch64 V2I-R-CTRL-ISB-TLBI-DSB.ISH-ISB-R+W
{
 [PTE(x)]=(oa:PA(x),valid:1);
 0:X3=PTE(x); 0:X0=(oa:PA(x),valid:0);
 0:X4=x;
 1:X2=(oa:PA(x),valid:0); 1:X3=PTE(x);
}
 P0             | P1          ;
 LDR X2,[X3]    | STR X2,[X3] ;
 CMP X0,X2      |             ;
 B.NE Lout      |             ;
 ISB            |             ;
 LSR X9,X4,#12  |             ;
 TLBI VAAE1,X9  |             ;
 DSB ISH        |             ;
 ISB            |             ;
L0:             |             ;
 LDR W1,[X4]    |             ;
Lout:           |             ;
exists(0:X2=(oa:phy_x,valid:0) /\ ~fault(P0:L0,x))
```

Forbidden execution



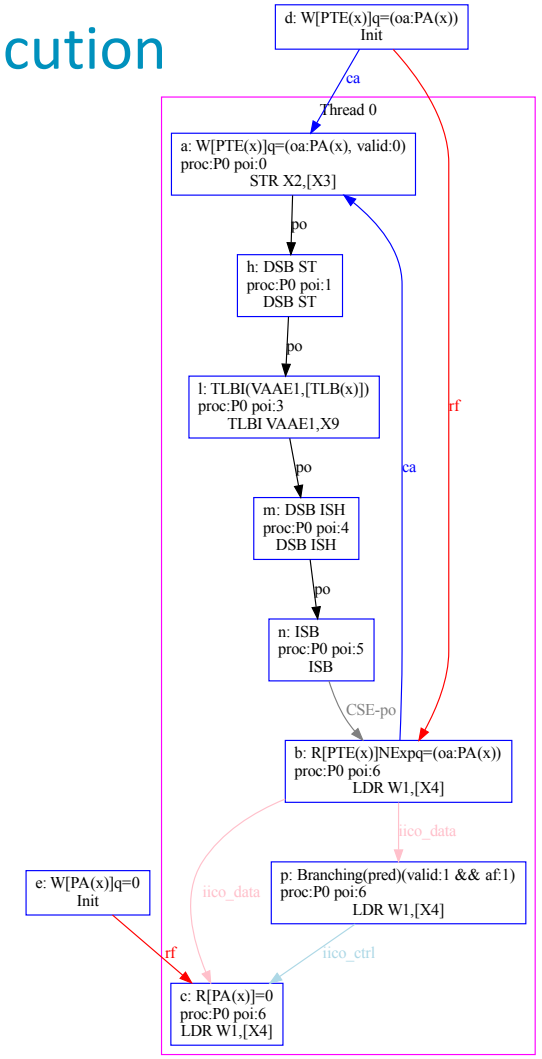
CSE semantics II

[ISB]; po (* ISB orders Implicit Effects after it in program order – this is why it is necessary to use after the TLBI; DSB in the uniprocessor case *)

Test (Forbidden)

```
AArch64 V2I-W-DSB.ST-TLBI-DSB.ISH-ISB-R
{
 [PTE(x)]=(oa:PA(x),valid:1);
 0:X2=(oa:PA(x),valid:0); 0:X3=PTE(x);
 0:X4=x;
}
 P0              ;
 STR X2,[X3]     ;
 DSB ST          ;
 LSR X9,X4,#12   ;
 TLBI VAAE1,X9   ;
 DSB ISH         ;
 ISB             ;
L0:              ;
 LDR W1,[X4]     ;
exists(~fault(P0:L0,x))
```

Forbidden execution



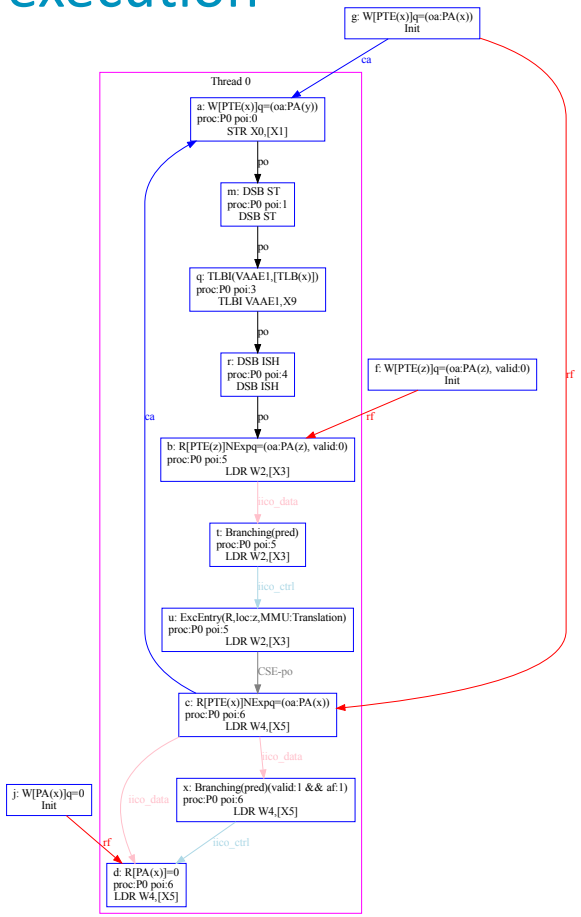
CSE semantics II

[EXC-ENTRY]; po

Test (Forbidden - for systems with FEAT_ExS when EIS is not enabled this becomes Allowed)

```
AArch64 coWR-pte+dsb.st-tlbi-dsb.ish-fault
{
 [PTE(x)]=(oa:PA(x));
 [PTE(z)]=(oa:PA(z),valid:0);
 y=1;
 0:X0=(oa:PA(y)); 0:X1=PTE(x);
 0:X3=z; 0:X5=x;
}
 P0             | P0.F            ;
 STR X0,[X1]    | LDR W4,[X5]     ;
 DSB ST         |                 ;
 LSR X9,X5,#12  |                 ;
 TLBI VAAE1,X9  |                 ;
 DSB ISH        |                 ;
 LDR W2,[X3]    |                 ;
exists(0:X4=0)
```

Forbidden execution



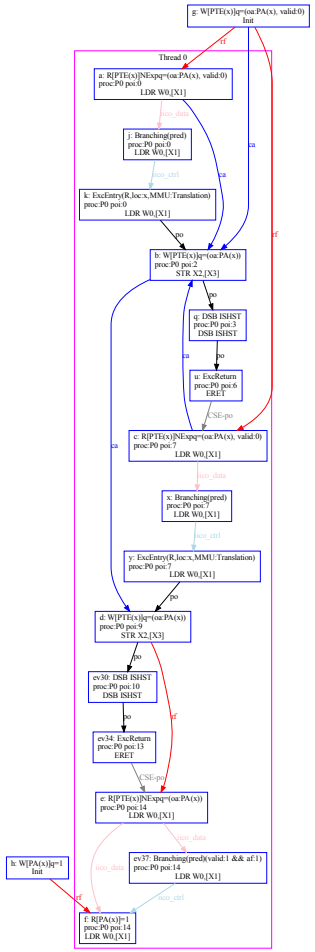
CSE semantics II

[EXC-RET]; po

Test (Forbidden - for systems with FEAT_ExS when EOS is not enabled this becomes Allowed)

```
AArch64 LDRv0+I2V-dsb.ishst-once
{
 [PTE(x)]=(oa:PA(x),valid:0);
 x=1;
 0:X1=x;
 0:X2=(oa:PA(x),valid:1); 0:X3=PTE(x);
}
 P0           | P0.F           ;
L0:           | ADD X8,X8,#1   ;
 LDR W0,[X1]  | STR X2,[X3]    ;
              | DSB ISHST      ;
              | ADR X9,L0      ;
              | MSR ELR_EL1,X9 ;
              | ERET          ;
exists(0:X8!=1)
```

Forbidden execution



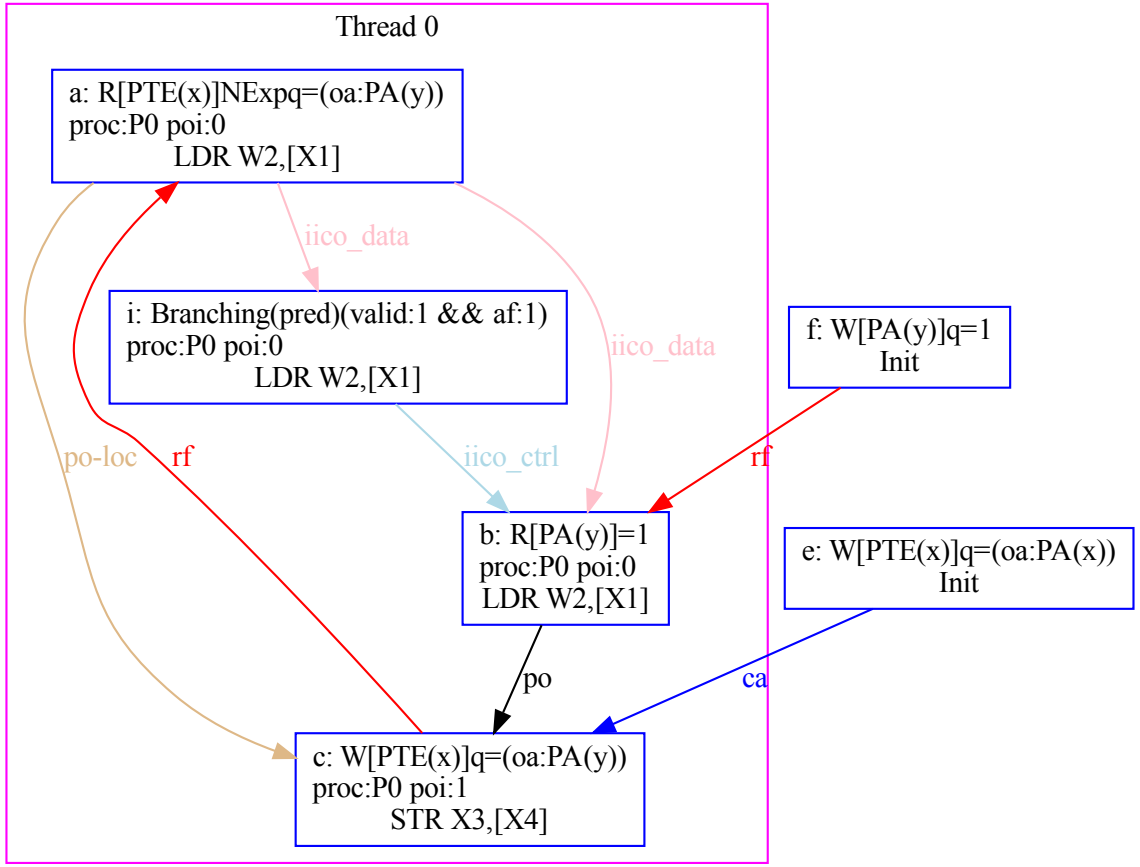
No implicit read from the future

[R & PTE & Imp]; po-loc; [W & PTE & Exp]

Test (Forbidden)

```
AArch64 VIS01-load
{
 [PTE(x)]=(oa:PA(x));
 y=1;
 0:X1=x;
 0:X3=(oa:PA(y)); 0:X4=PTE(x);
}
 P0            ;
 LDR W2,[X1]   ;
 STR X3,[X4]   ;
exists(0:X2=1)
```

Forbidden execution



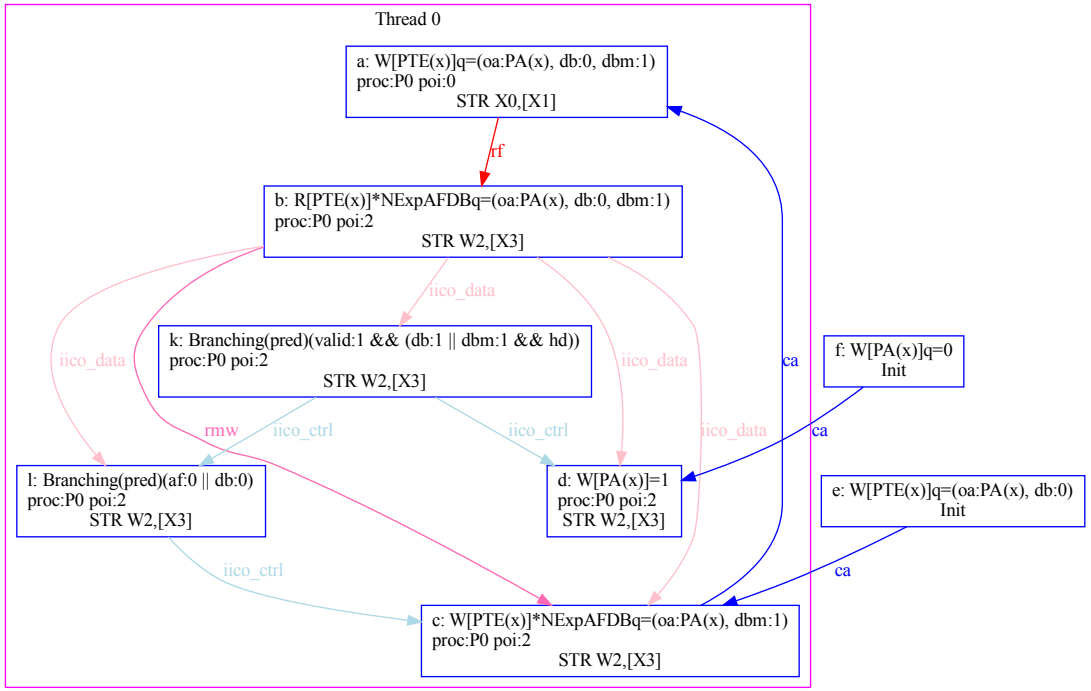
Sanity check on Hardware Updates

[R & PTE & Imp]; rmw; [HU]

Test (Forbidden)

```
AArch64 coWW-HD
TTHM=HA HD
{
 [PTE(x)]=(oa:PA(x),db:0,dbm:0);
 0:X0=(oa:PA(x),db:0,dbm:1); 0:X1=PTE(x);
 0:X3=x;
}
 P0           ;
 STR X0,[X1] ;
 MOV W2,#1    ;
L0:           ;
 STR W2,[X3] ;
exists(~fault(P0:L0,x) /\
        [PTE(x)]=(oa:PA(x),db:0,dbm:1))
```

Forbidden execution



TLBUncacheable Faults

In cat

```
let TLBUncacheableFault =
  FAULT & MMU & (Translation | AccessFlag)
```

In English

In this presentation, an MMU Fault Effect is TLBUncacheable if it is one of:

- A Translation Fault Effect
- An Access Flag Fault Effect

Note: a Permission Fault Effect is not TLBUncacheable – it is permitted to be cached in a TLB.

Intuitively, if an entry could legitimately not fault for some access, then that entry can be cached in a TLB, even if it would fault for other accesses.

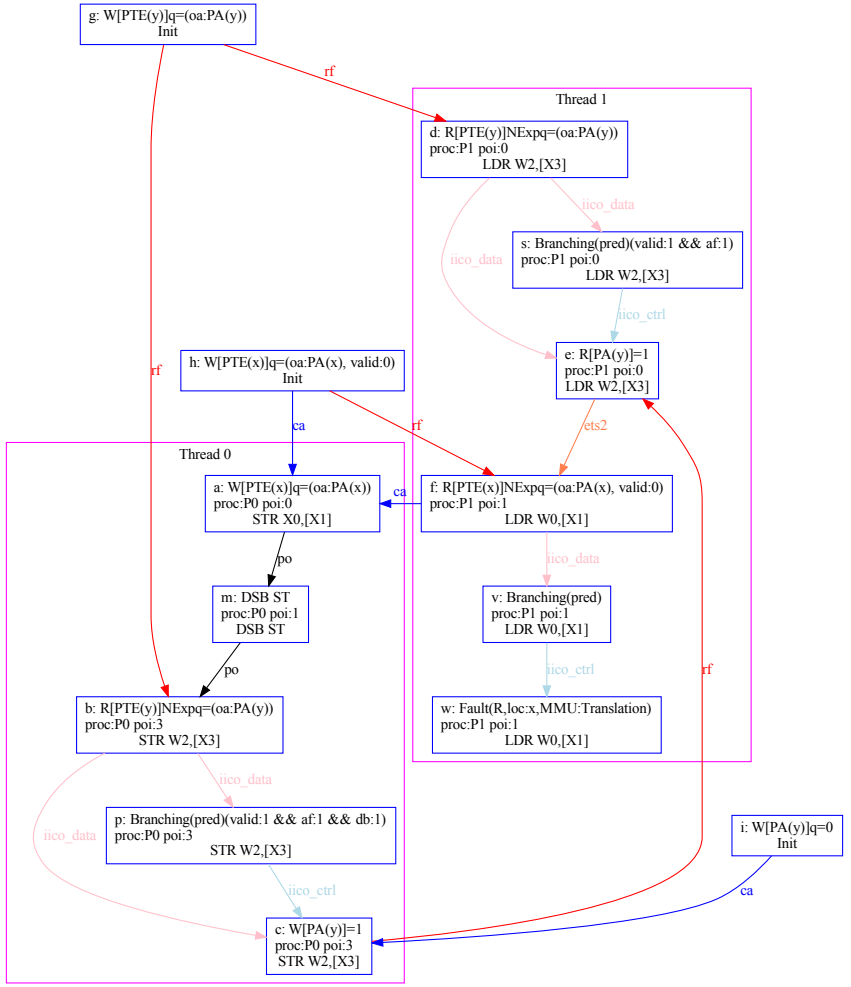
ETS2

[M & Exp]; po; [TLBUncacheableFault]; tr-ib^-1; [R & Imp & PTE]

Test (**with ETS2**, Forbidden)

```
AArch64 I2V-MP+dsb.st+po
{
 [PTE(x)]=(oa:PA(x),valid:0);
 0:X0=(oa:PA(x),valid:1); 0:X1=PTE(x);
 0:X3=y;
 1:X1=x; 1:X3=y;
}
 P0              | P1              ;
 STR X0,[X1]     | LDR W2,[X3]     ;
 DSB ST          |L0:              ;
 MOV W2,#1       | LDR W0,[X1]     ;
 STR W2,[X3]     |                 ;
exists(1:X2=1 /\ fault(P1:L0,x))
```

Forbidden execution



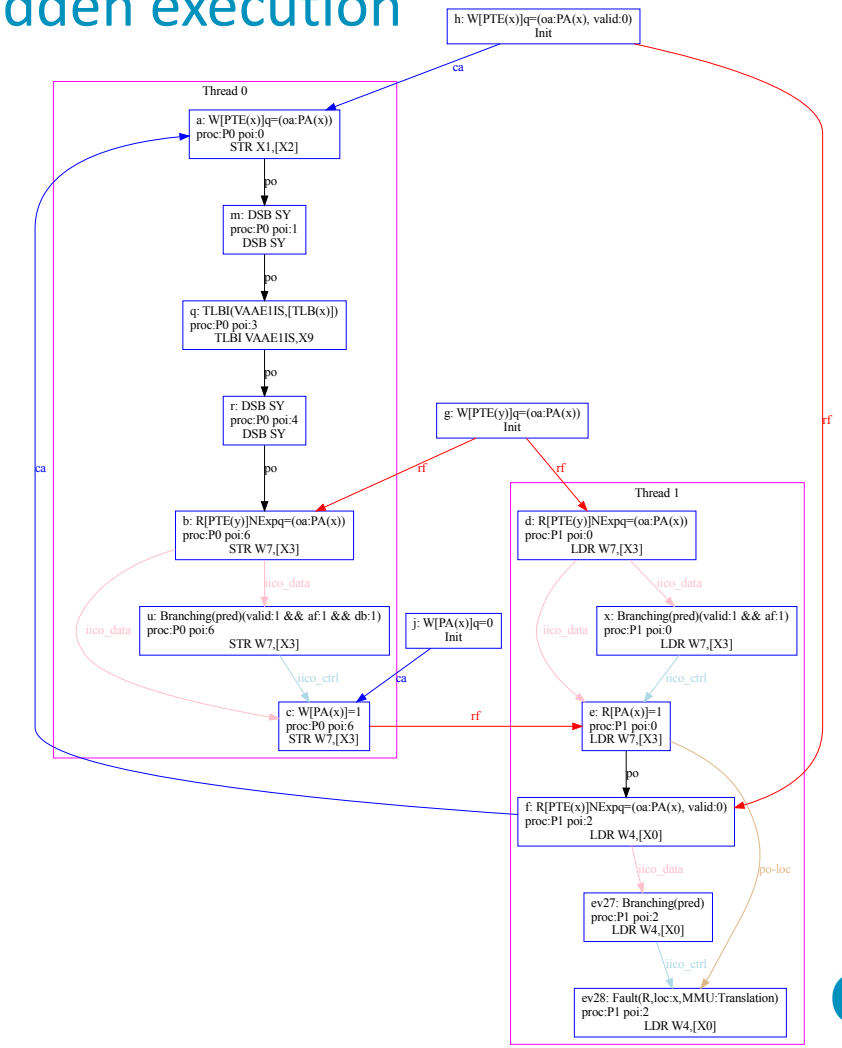
Partial parity with ETS2 for TLBI cases

[M & Exp]; po-loc; [TLBUncacheableFault]

Test (Forbidden)

```
AArch64 Ja4-TLBIDSB
{
  [PTE(y)]=(oa:PA(x),valid:1);
  [PTE(x)]=(oa:PA(x),valid:0);
  0:X1=(oa:PA(x),valid:1); 0:X2=PTE(x);
  0:X3=y; 0:X0=x;
  1:X3=y; 1:X0=x;
}
P0               |  P1          ;
 STR X1,[X2]     | LDR W7,[X3] ;
 DSB SY          | MOV W4, #2  ;
 LSR X9,X0,#12   |L0:          ;
 TLBI VAAE1IS,X9 | LDR W4,[X0] ;
 DSB SY          |             ;
 MOV W7, #1      |             ;
 STR W7,[X3]     |             ;
exists(1:X7=1 /\ fault(P1:L0,x))
```

Forbidden execution



arm

New observation requirements

New observation requirements

Add the following to ob

In cat

```
| rf & int & (_ * R & TTD & Imp)
| rf & ext & (_ * R & TTD & Imp)
| tlbi-ca
| TLBUncacheable-pred
| HU-pred
| [HU]; co | co; [HU]
| tlbi-ob
```

In English

- E2 is an Implicit TTD Read Effect, E1 and E2 are from the same Observer and E2 Reads-from E1
- E2 is an Implicit TTD Read Effect, E1 and E2 are from different Observers and E2 Reads-from E1
- E2 is TLBI-Coherence-after E1
- E1 is a TLBUncacheable-Predecessor of E2
- E1 is a Hardware-Update-Predecessor of E2
- E1 is Coherence-before E2, and either E1 or E2 is a Hardware Update
- E1 is TLBI-Ordered-before E2

New observation requirements

Systematic review

```
| rf & int & (_ * R & TTD &Imp)
| rf & ext & (_ * R & TTD & Imp)
| tlbi-ca
| TLBUncacheable-pred
| HU-pred
| [HU]; co | co; [HU]
| tlbi-ob
(*including:
  | tr-ib^-1; TTD-read-ordered-before & ext
  | po-va-loc; TTD-read-ordered-before & ext
*)
```

We extend ob with new observation requirements so that we have the following property:

Implicit PTE Reads observe PTE Writes consistently with the hardware requirements.

- Separate observer: Internal reads-from maintained
- External reads-from maintained
- Coherence restored/maintained 1
- Coherence restored/maintained 2
- Coherence restored/maintained 3
- Coherence restored/maintained 4
- TLBI semantics
- Broadcasting
- Hazarding

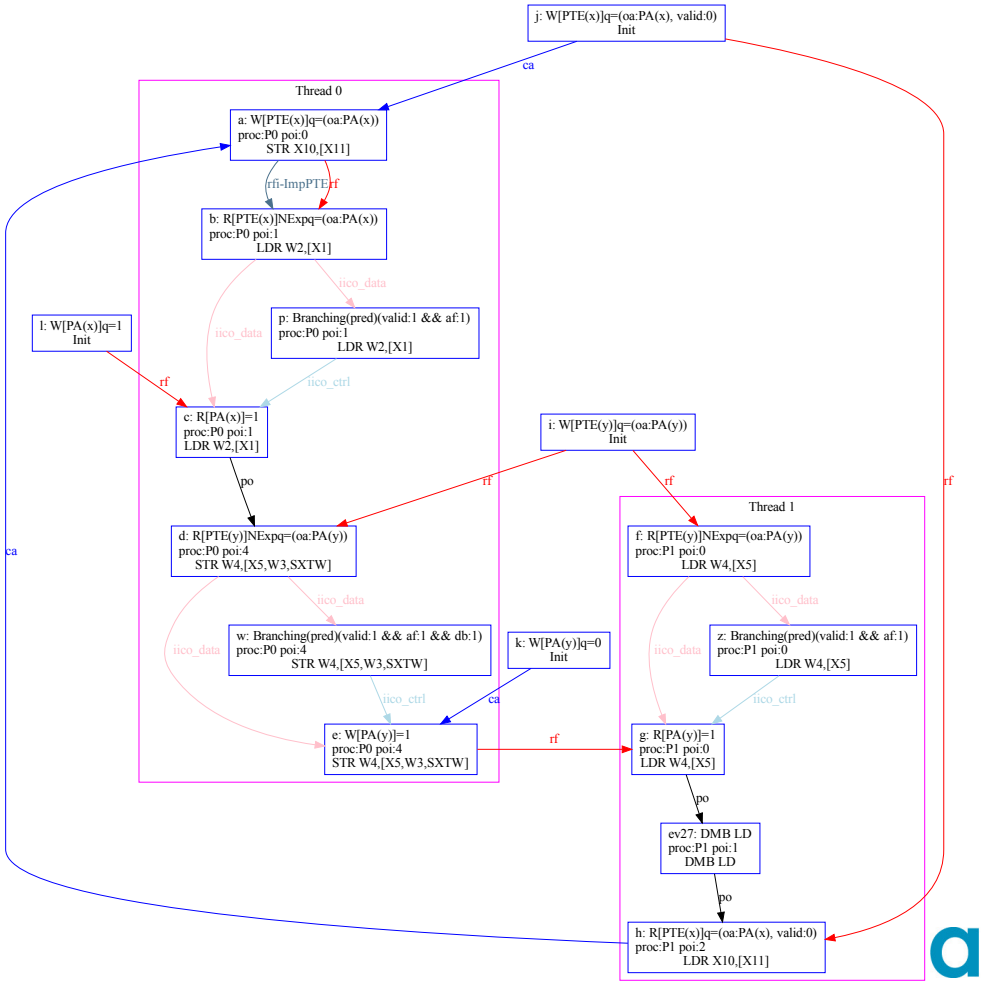
Separate observer: Internal reads-from maintained

rfi & (_ * R & TTD & Imp)

Test (Forbidden)

```
AArch64 MP+pterfi-addr+dmb.ld
{
  x=1;
  [PTE(x)]=(oa:PA(x),valid:0);
  0:X10=(oa:PA(x), valid:1); 0:X11=PTE(x);
  0:X1=x; 0:X5=y;
  1:X5=y;
  1:X11=PTE(x);
}
 P0                  | P1            ;
 STR X10,[X11]       | LDR W4,[X5]   ;
 LDR W2,[X1]         | DMB LD        ;
 EOR W3,W2,W2        | LDR X10,[X11] ;
 MOV W4,#1           |               ;
 STR W4,[X5,W3,SXTW] |               ;
exists(1:X4=1 /\ 1:X10=(oa:PA(x),valid:0))
```

Forbidden execution



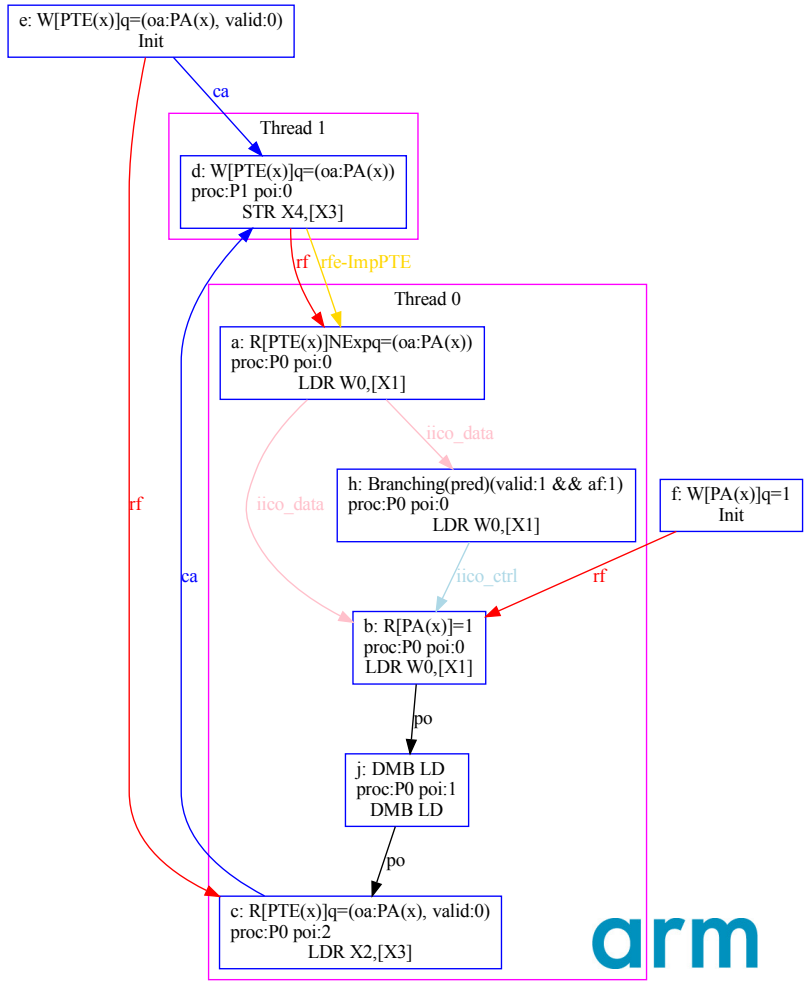
External reads-from maintained

rfe & (_ * R & TTD & Imp)

Test (Forbidden)

```
AArch64 coRR+Imp-dmb.ld-Exp
{
 x=1;
 [PTE(x)]=(oa:PA(x),valid:0);
 0:X1=x;
 0:X3=PTE(x);
 1:X4=(oa:PA(x),valid:1); 1:X3=PTE(x);
}
 P0           | P1           ;
L0:           | STR X4, [X3] ;
 LDR W0,[X1]  |              ;
 DMB LD       |              ;
 LDR X2,[X3]  |              ;
exists(~fault(P0:L0,x) /\ 0:X2=(oa:PA(x),valid:0))
```

Forbidden execution



TLB Invalidation scope and TLBI-after

In cat

```
cat primitive inv-scope

let CTLBI = TLBI & domain(po;[dsb.full])

(* given a cat function add_both_choices which enumerates
pairs one way or the other: *)
let ttd-tlbi-pairs =
   [R & TTD]; inv-scope; [CTLBI]
with tlbi-after from add_both_choices(ttd-tlbi-pairs)
```

In English

A TLBI instruction defines an Invalidation scope, which is a set of TTDs. A TTD Memory Read or Write Effect is in the Invalidation Scope of a TLBI instruction if its address is in the Invalidation Scope of the TLBI instruction.

A TLBI Effect is a Completed TLBI Effect if and only if it is program-ordered-before an Effect generated by a DSB FULL.

For two Effects E1 and E2 such that E1 is a completed TLBI Effect and E2 is a TTD Read Memory Effect RW2 and RW2 is in the Invalidation Scope of E1, one and only one of the following applies:

- E1 is TLBI-after E2 (equivalently E2 is TLBI-before E1), or
- E2 is TLBI-after E1 (equivalently E1 is TLBI-before E2)

Note: The TLBI-after relation enumerates all possible pairs (E1,E2) where E1 is a TTD Read Memory Effect and E2 is a TLBI Effect such that E1 is in the Invalidation Scope of E2, and the relation TLBI-after is asymmetric.

TLBI-Coherence-{before,after}

In cat

```
let tlbi-ca =
   [TLBI]; tlbi-after; [R & TTD & Imp]; ca; [W & TTD]
```

In English

An Effect E1 is TLBI-Coherence-before an Effect E2 (equivalently, E2 is TLBI-Coherence-after E1) if all of the following applies:

- E1 is a TLBI Effect
- E2 is a TTD Write Effect
- E2 is an Implicit TTD Read Effect such that all of the following applies:
  - E1 is TLBI-before E3
  - E3 is Coherence-before E2

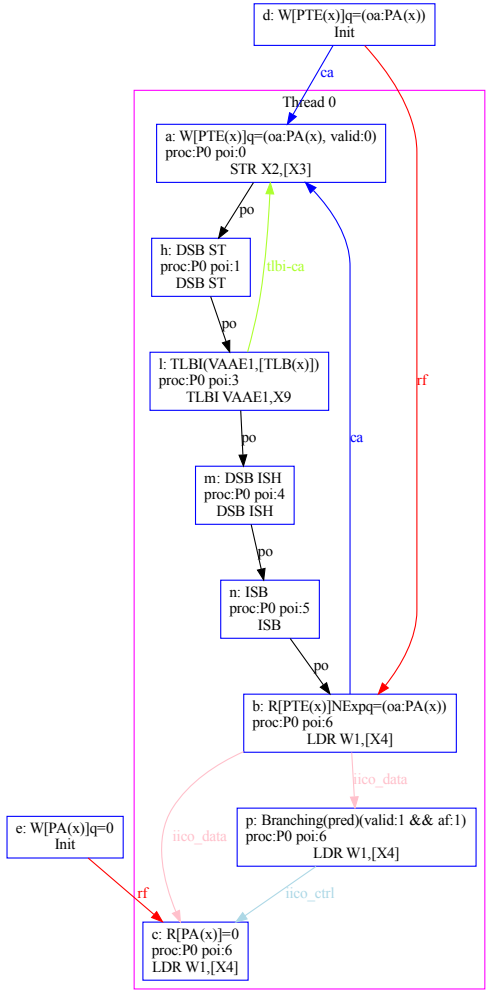
Coherence restored/maintained 1

tlbi-ca

Test (Forbidden)

```
AArch64 V2I-W-DSB.ST-TLBI-DSB.ISH-ISB-R
{
 [PTE(x)]=(oa:PA(x),valid:1);
 0:X2=(oa:PA(x),valid:0); 0:X3=PTE(x);
 0:X4=x;
}
 P0              ;
 STR X2,[X3]     ;
 DSB ST          ;
 LSR X9,X4,#12   ;
 TLBI VAAE1,X9   ;
 DSB ISH         ;
 ISB             ;
L0:              ;
 LDR W1,[X4]     ;
exists(~fault(P0:L0,x))
```

Forbidden execution



TLBUncacheable-Write-{Predecessor, Successor}

In cat

```
let TLBuncacheable-pred =
 [TLBUncacheable];ca\intervening-write(ca);[W & TTD & Exp]
```

In English

A Read or Write Memory Effect RW1 is a TLBUncacheable-Write-Predecessor of an Explicit TTD Write Memory Effect W2 if and only if all of the following applies:

- RW1 reads a TLBUncacheable TTD or writes a TLBUncacheable value to a TTD
- RW1 is Coherence-Before W2,
- there is not a Write Memory Effect W3 such that RW1 is Coherence-Before W3 and W3 is Coherence-Before W2.

E2 is a TLBUncacheable-Write-Successor of E1 if and only if E1 is a TLBUncacheable-Write-Predecessor of E2.

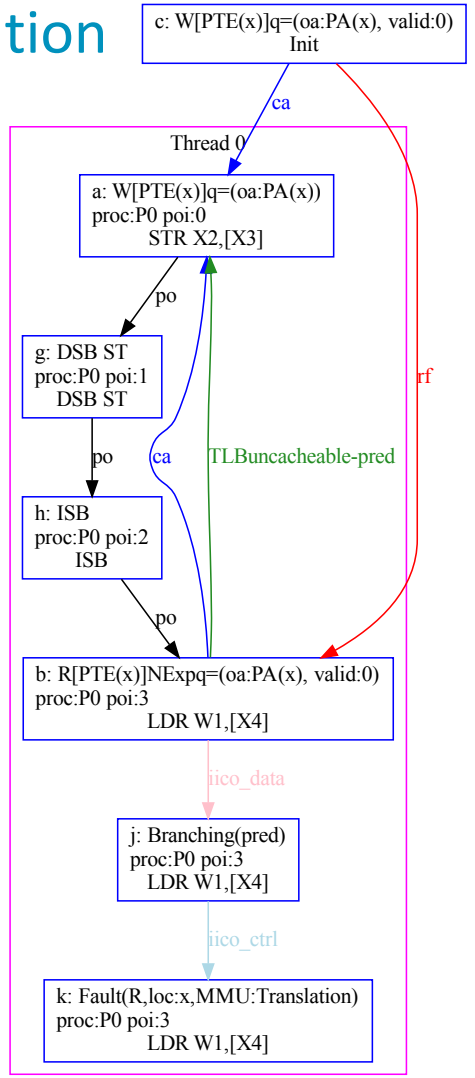
Coherence restored/maintained 2

TLBUncacheable-pred

Test (Forbidden)

```
AArch64 I2V-W-DSB.ST-ISB-R
{
 [PTE(x)]=(oa:PA(x),valid:0);
 0:X2=(oa:PA(x),valid:1); 0:X3=PTE(x);
 0:X4=x;
}
 P0              ;
 STR X2,[X3]     ;
 DSB ST          ;
 ISB             ;
L0:              ;
 LDR W1,[X4]     ;
exists(fault(P0:L0,x))
```

Forbidden execution



Hardware-Update-{Predecessor, Successor}

In cat

```
let HU-predecessor =
  [TTD]; ca \ intervening-write(ca); [HU]
```

In English

A TTD Effect E1 is a Hardware-Update-Predecessor of another Effect E2 if and only if all of the following applies:

- E2 is a Hardware Update Effect,
- E1 is Coherence-Before E2 and
- there is not a Write Memory Effect W3 such that E1 is Coherence-Before W3 and W3 is Coherence-Before E2.

E2 is a Hardware-Update-Successor of E1 if and only if E1 is a Hardware-Update-Predecessor of E2.

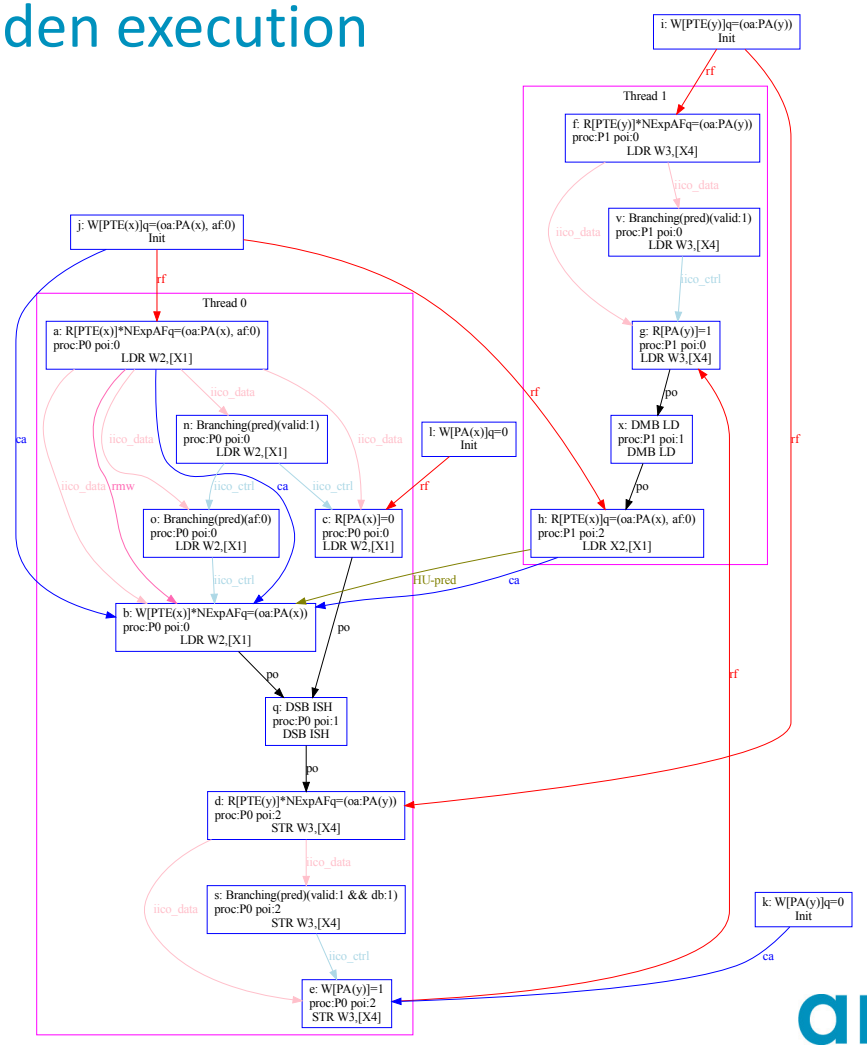
Coherence restored/maintained 3

HU-pred

Test (Forbidden)

```
AArch64 MP+HA-DSB.ISH
TTHM=HA
{
 PTE(x)=(oa:PA(x),af:0);
 0:X1=x; 0:X4=y;
 1:X1=PTE(x); 1:X4=y;
}
P0           | P1           ;
LDR W2,[X1]  | LDR W3,[X4]  ;
DSB ISH      | DMB LD       ;
MOV W3,#1    | LDR X2,[X1]  ;
STR W3,[X4]  |              ;
exists(1:X3=1 /\ 1:X2=(oa:PA(x),af:0))
```

Forbidden execution



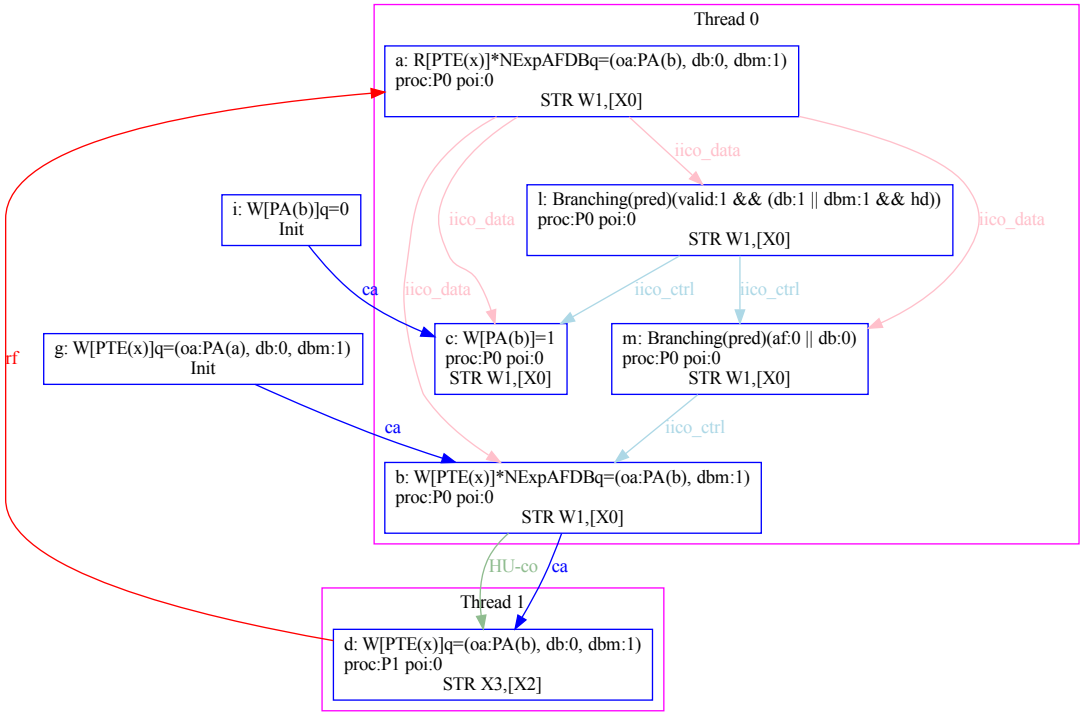
Coherence restored/maintained 4

[HU]; co | co; [HU]

Test (Forbidden)

```
AArch64 STRva-STRpte2
TTHM=HA HD
{
 int a; int b;
 [PTE(x)]=(oa:PA(a),db:0,dbm:1);
 0:X1=1; 0:X0=x;
 1:X3=(oa:PA(b),db:0,dbm:1); 1:X2=PTE(x);
}
 P0            | P1;
 STR W1, [X0] | STR X3, [X2];
exists(b=1 /\ PTE(x)=(oa:PA(b),db:0, dbm:1))
```

Forbidden execution



TTD-read-ordered-before

In cat

```
let TTD-read-ordered-before =
    [R & TTD & Imp]; tlbi-after; [TLBI]; po;
[dsb.full]; po; [~(M&Imp)]
```

In English

If there exist Effects E3, E4 and E2 such that all of the following applies:

- E3 is a TLBI Effect (i.e. generated by a TLBI instruction)
- E4 is generated by a DSB FULL
- E3 is program-order-before E4
- E4 is program-order-before E2
- E2 is any Effect other than an Implicit Memory Effect

Any Implicit TTD Read Memory Effect E1 that is TLBI-before E3 is TTD-read-ordered-before E2

arm

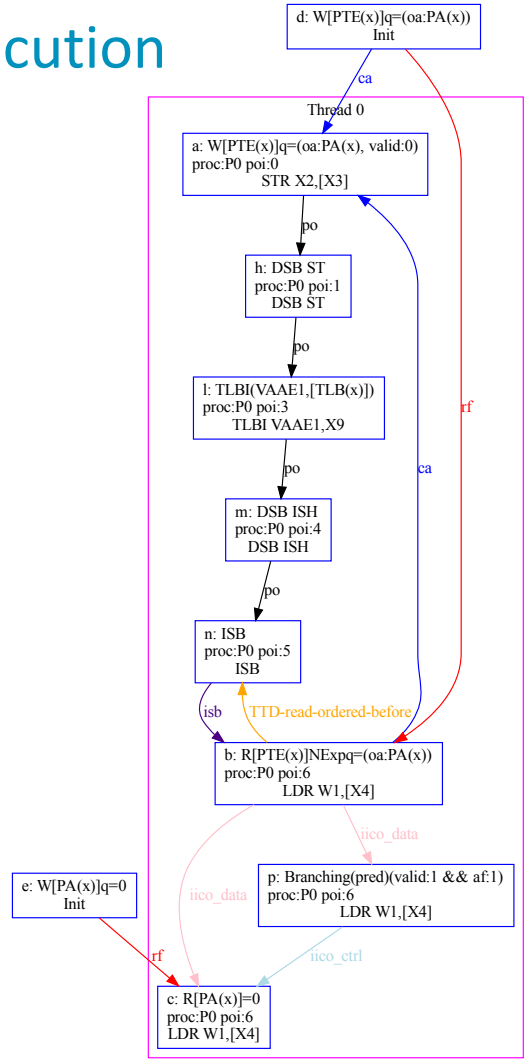
Basic TLBI semantics

TTD-read-ordered-before

Test (Forbidden)

```
AArch64 V2I-W-DSB.ST-TLBI-DSB.ISH-ISB-R
{
 [PTE(x)]=(oa:PA(x),valid:1);
 0:X2=(oa:PA(x),valid:0); 0:X3=PTE(x);
 0:X4=x;
}
 P0              ;
 STR X2,[X3]     ;
 DSB ST          ;
 LSR X9,X4,#12   ;
 TLBI VAAE1,X9   ;
 DSB ISH         ;
 ISB             ; // could be any CSE
L0:              ;
 LDR W1,[X4]     ;
exists(~fault(P0:L0,x))
```

Forbidden execution



Broadcasting

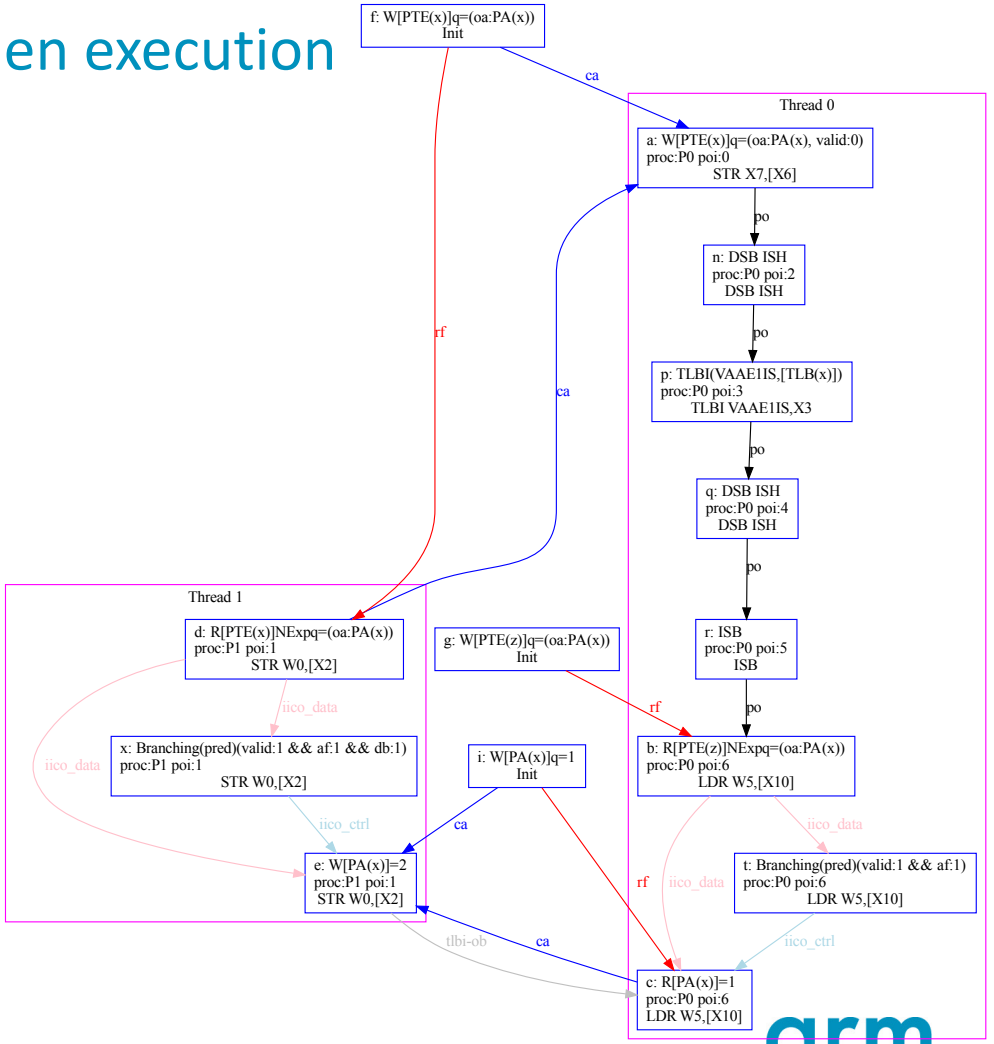
tr-ib^-1; TTD-read-ordered-before & ext

Test (Forbidden)

```
AArch64 miniMarc03
{
 x=1;
 PTE(z) = (oa:PA(x));
 0:X7=(oa:PA(x),valid:0); (* invalid entry *)
 0:X6=PTE(x);
 0:X2=x; 0:X10=z;
 1:X2=x;
}

 P0              | P1           ;
 STR X7,[X6]     | MOV W0,2     ;
 LSR X3,X2,12    | STR W0,[X2]  ;
 DSB ISH         |              ;
 TLBI VAAE1IS,X3 |              ;
 DSB ISH         |              ;
 ISB             |              ;
 LDR W5,[X10]    |              ;
exists(x=2 /\ 0:X5=1)
```

Forbidden execution



Hazarding

po-va-loc; TTD-read-ordered-before & ext

Test (Forbidden)

```
AArch64 coRR-pte9
{
 y=1;
 0:X1=x;
 1:X0=(oa:PA(x),valid:0);
 1:X4=(oa:PA(y));
 1:X3=PTE(x);
 1:X1=x;
}
 P0           | P1                ;
L1:           | STR X0,[X3]       ;
 LDR W2,[X1]  | DSB ISH           ;
L0:           | LSR X9,X1,#12     ;
 LDR W4,[X1]  | TLBI VAAE1IS,X9;
              | DSB ISH           ;
              | STR X4,[X3]       ;
exists(0:X2=1/\ 0:X4=0 /\ ~fault(P0:L0,x))
```

Forbidden execution