Precise exceptions in relaxed architectures

Simner et al., 2025

18-742 Presentation

Ania Krzyżańska, Abnash Bassi, Soren Dupont

Outline

- Context
- Motivation
- Background
- Model
- Limitations & Conclusion
- Open Discussion

Who are the authors?

University of Cambridge:

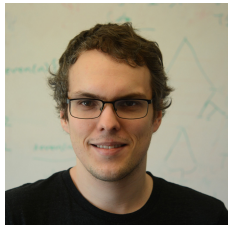
- Ben Simner
- Alasdair Armstrong
- Thomas Bauereiss
- Peter Sewell (Fellow of the Royal Society)

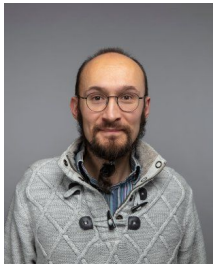
University of Edinburgh:

- Brian Campbell
- Ohad Kammar

Aarhus University:

- Jean Pichon-Pharabod

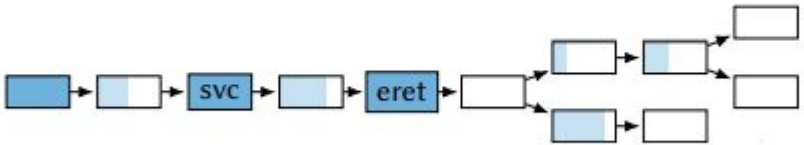

ACM SIGARCH/IEEE-CS TCCA ISCA Best Paper Award

Context

- Exceptions are supposed to appear as happening between instructions
- however this is not what happens
- new model is required
- ARM
  - from cambridge
  - preferred ISA for embedded processors
- Interrupts and Exceptions in embedded systems
  - usually triggered in response to sensors, requiring some swift action
  - less exceptional than in other kinds of computing
  - much more important to handle quickly and seamlessly in embedded systems

Exceptions in ARM

- Synchronous exceptions
  - supervisor calls, traps, page faults, etc.
  - ‘precise’
- Interrupts
  - interrupt requests, fast interrupt requests, system errors
  - all are ‘precise’, except external system aborts may not be
- Reminders in the paper:
  - ‘Instructions’ are actually fetch-decode-execute instances
  - Speculative execution



Context Synchronization

- “Updates to the context (ex: writes to system registers - ESR, FAR, ELR, EL) need synchronisation to be guaranteed to have an effect”
- Context Synchronization event
  - guarantee that no instruction after the event is observed occurring (fetched, decoded, or executed) until the context-synchronising event has happened
  - semantically equivalent to flushing the pipeline
- Software may implement context synchronisation by issuing Instruction Synchronisation Barriers (ISB)

Relaxed Behaviors

- Examples from the paper showing behaviors that are allowed and not allowed
- Different exception kinds may have different requirements/behaviors
- Mostly overviewed examples of system calls using SVC, which unconditionally generate an exception at a specific point in the program
- Asynchronous exceptions cannot be taken speculatively; they must be completed before proceeding in the program
- Authors spoke with ARM architects to verify the intentions behind the architecture

Out of order w.r.t. exception barriers: allowed on ARM processors

S+dmb.sy+svc AArch64

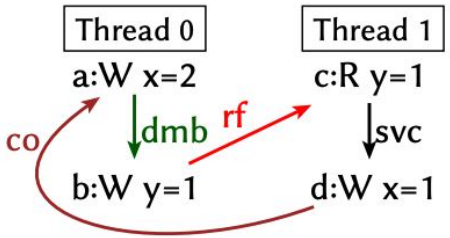
Initial state: *x=0, *y=0;

0:X1=x, 0:X3=y; 1:X1=y, 1:X3=x

Thread 0	Thread 1	T1 Handler
MOV X0,#2 **STR** X0,[X1] **DMB** SY **MOV** X2,#1 **STR** X2,[X3]	**LDR** X0,[X1] **SVC** #0	**MOV** X2,#1 **STR** X2,[X3]

Allowed: 1:X0=1, *x=2

ordering: dabc



Out of order w.r.t. exception barriers: allowed on ARM processors

SB+dmb.sy+eret AArch64

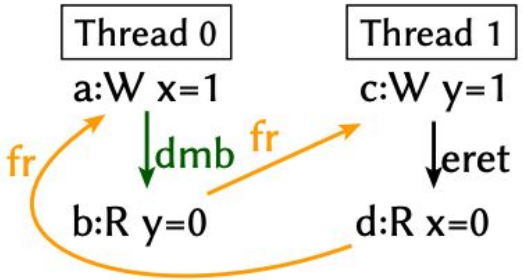
Initial state: *x=0, *y=0;

0:X1=x, 0:X3=y; 1:X1=y, 1:X3=x

Thread 0	Thread 1	T1 Handler
MOV X0,#1 **STR** X0,[X1] **DMB** SY **LDR** X2,[X3]	**SVC** #0 **LDR** X2,[X3]	**MOV** X0,#1 **STR** X0,[X1] **ERET**

Allowed: 0:X2=0, 1:X2=0

ordering: dabc



Out of order w.r.t. exception barriers: allowed on ARM processors

MP+svceret+addr AArch64

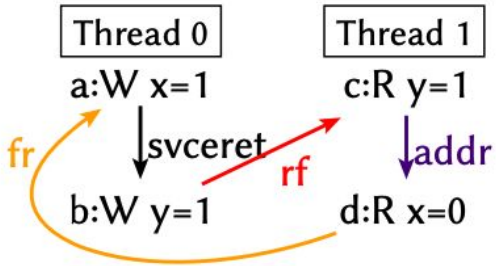
Initial state: *x=0, *y=0;

0:X1=x, 0:X3=y; 1:X1=y, 1:X3=x

Thread 0	T0 Handler	Thread 1
MOV X0,#1 **STR** X0,[X1] **SVC** #0 **MOV** X2,#1 **STR** X2,[X3]	**ERET**	**LDR** X0,[X1] **EOR** X4,X0,X0 **LDR** X2,[X3,X4]

Allowed: 1:X0=1, 1:X2=0

ordering: bcda



Speculative instruction of exceptions: forbidden

MP+dmb.sy+ctrlsvc AArch64

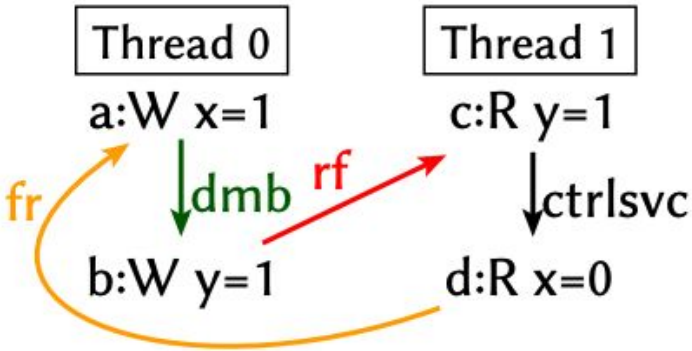
Initial state: *x=0, *y=0;

0:X1=x, 0:X3=y; 1:X1=y, 1:X3=x

Thread 0	Thread 1	T1 Handler
MOV X0,#1 **STR** X0,[X1] **DMB** SY **MOV** X2,#1 **STR** X2,[X3]	**LDR** X0,[X1] **CBNZ** X0,LC00 LC00: **SVC** #0	**LDR** X2,[X3]

Forbidden: 1:X0=1, 1:X2=0

ordering: dabc



Forwarding stores: allowed

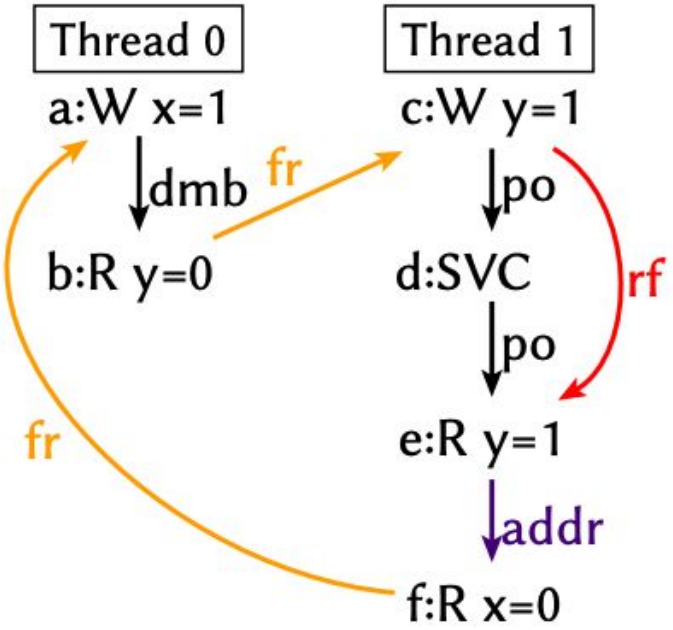
SB+dmb.sy+rfisvc-addr AArch64

Initial state: *x=0, *y=0;

0:X1=x, 0:X3=y; 1:X1=y, 1:X3=y, 1:X5=x

Thread 0	Thread 1	T1 Handler
MOV X0,#1 **STR** X0,[X1] **DMB** SY **LDR** X2,[X3]	**MOV** X0,#1 **STR** X0,[X1] **SVC** #0	**LDR** X2,[X3] **EOR** X6,X2,X2 **LDR** X4,[X5,X6]

Allowed: 1:X0=1, 1:X2=0



ordering: cdefab

System registers and context synchronisation

MP.EL1+dmb.sy+dataesrsvc AArch64

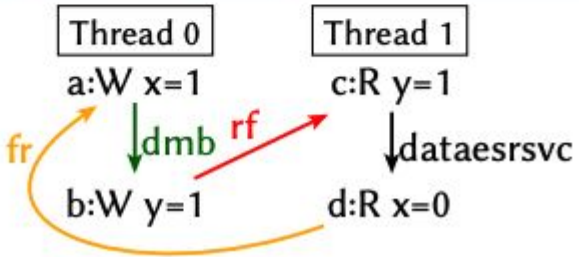
Initial state: *x=0, *y=0;

0:X1=x, 0:X3=y; 1:PSTATE.EL=0b1, 1:X1=y,

1:X3=x

Thread 0	Thread 1	T1 Handler
MOV X0,#1 **STR** X0,[X1] **DMB** SY **MOV** X2,#1 **STR** X2,[X3]	**LDR** X0,[X1] **MRS** X4,ESR_EL1 **EOR** X5,X0,X0 **ADD** X5,X4,X5 **MSR** ESR_EL1,X5 **SVC** #0	**LDR** X2,[X3]

Forbidden: 1:X0=1, 1:X2=0



ordering: dabc

System registers and context synchronisation

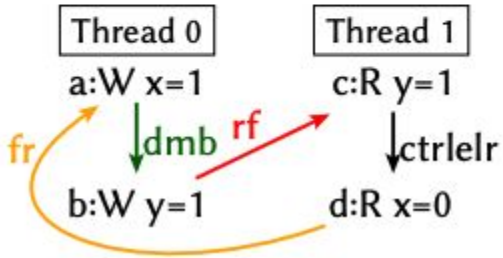
MP+dmb.sy+ctrlelr AArch64

Initial state: *x=0, *y=0;

0:X1=x, 0:X3=y; 1:X1=y, 1:X3=x

Thread 0	Thread 1	T1 Handler
MOV X0,#1 **STR** X0,[X1] **DMB** SY **MOV** X2,#1 **STR** X2,[X3]	**SVC** #0 **LDR** X2,[X3]	**LDR** X0,[X1] **MRS** X4,ELR_EL1 **EOR** X5,X0,X0 **ADD** X5,X4,X5 **MSR** ELR_EL1,X4 **ERET**

Forbidden: 1:X0=1, 1:X2=0



ordering: dabc

Different demands for different exceptions

MP+dmb.sy+fault AArch64

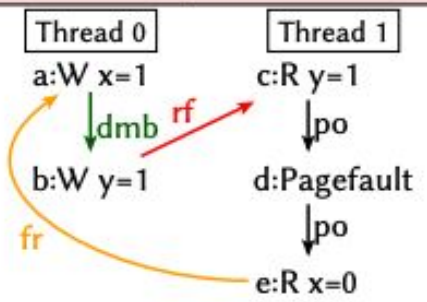
Initial state: *x=0, *y=0;

0:X1=x, 0:X3=y; 1:X1=y, 1:X3=x

Thread 0	Thread 1	T1 Handler

MOV X0,#1
STR X0,[X1]
DMB SY
MOV X2,#1
STR X2,[X3]
	LDR X0,[X1]
	MOV X5,#0
	/ segfault
	LDR X4,[X5]
		LDR X2,[X3]

Forbidden: 1:X0=1, 1:X2=0



MP+dmb.sy+int AArch64

Initial state: *x=0, *y=0;

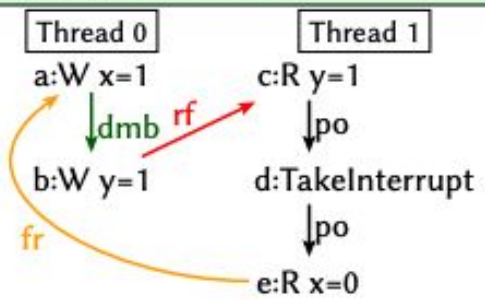
0:X1=x, 0:X3=y; 1:X1=y, 1:X3=x

interrupt at=L

Thread 0	Thread 1	T1 Handler

MOV X0,#1
STR X0,[X1]
DMB SY
MOV X2,#1
STR X2,[X3]
	LDR X0,[X1]
	L:
	NOP
		LDR X2,[X3]

Allowed: 1:X0=1, 1:X2=0



ordering: eabc

Exception special case: intra-instruction exception

STR Xt, [Xn], #8

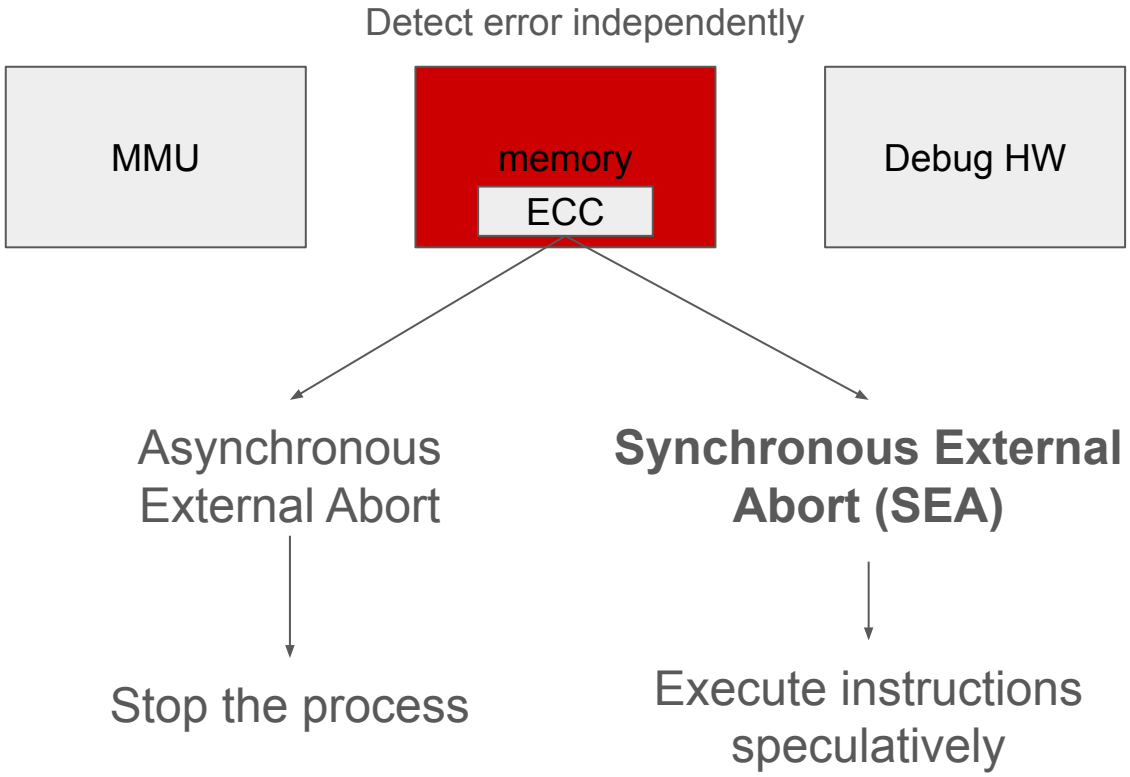
- In this instruction, Xt is stored into the address at location Xn, and Xn is incremented
- If the memory operation triggers an exception, the increment should not happen

Experimental results from different processors

Name	m6g	m7g	m8g	odroid	m2	pi3	pi4	pi5
MP+dmb+ctrl-svc	0/16M	0/24M	0/12M	0/329M	0/360M	0/10M	0/230M	0/136M
MP+dmb+ctrlelr	0/16M	0/24M	0/12M	0/329M	0/360M	0/30M	0/318M	0/130M
MP+svc-eret+addr	U0/16M	U0/24M	U0/12M	149K/328M	U0/360M	376/9M	U0/228M	12/136M
MP.EL1+dmb+dataesrsvc	0/16M	0/24M	0/12M	0/16M	0/0	0/4M	0/14M	0/27M
S+dmb+svc	U0/16M	U0/24M	U0/12M	U0/328M	U0/360M	U0/41M	U0/222M	U0/101M
SB+dmb+eret	60/16M	120/24M	213/12M	262/328M	12K/360M	203K/41M	946K/222M	4K/100M
SB+dmb+rfisvc-addr	4/16M	235/24M	1K/12M	305K/328M	12/360M	1M/30M	7K/316M	197K/128M
MP+dmb+fault	0/16M	0/24M	0/12M	0/74M	0/0	0/2M	0/46M	0/80M

Figure 9: Experimental results.

Synchronous external aborts (§4)



RAS Extension helps handle HW errors (§4)

Reliability, Availability, and Service ARM ISA Extension, introducing:

1. Error Synchronization Barrier instructions
2. RAS Extension Registers
3. Poison: lets HW flag that data has been corrupted

Architecture implementations should use this extension

Left as future work for concurrency models

Behaviour resulting from synchronous external aborts (§4.1)

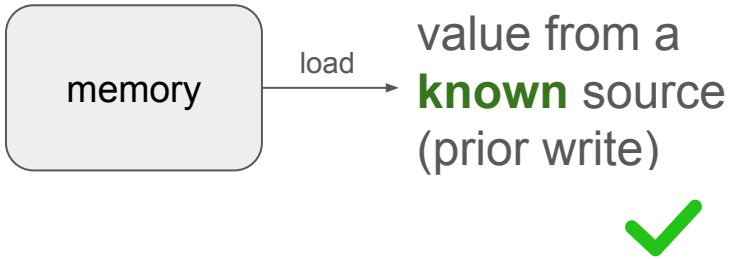
If SEA caused by store

Later instances are speculative until store reaches memory
Cannot re-order later stores
Can re-order later loads

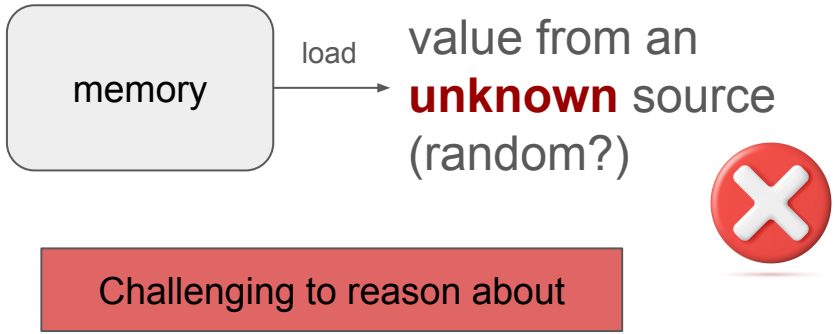
Else if SEA caused by load

Later instances are speculative until load is completely done and cannot be restarted
Cannot speculatively execute later stores, let alone re-order
Can re-order later loads (?)

Concurrency models do not capture how load buffering can lead to the out-of-thin-air-problem (§4.2)



Due to re-ordering before prior writes…



Solution:

Do not buffer loads, so loads are not re-ordered before stores to the same address

Ensures that data loaded was from a prior store, and not a value “out-of-thin-air”

How can we reason about HW/SW behaviour?

An axiomatic model of exceptions (§5)!

Describes how concurrent precise exceptions behave on Arm-A architecture

Parameterized:

1. `FEAT_ExS`: if “exception entry and exception return are Context Synchronization events” [Armv8.5]; not all HW supports
2. `SEA_R` and `SEA_S`: if reads & writes can generate SEAs

(Cont). How can we reason about HW/SW behaviour?

```
(* might-be speculatively
     executed *)
let speculative =
    ctrl
  | addr; po
  | if "SEA_R" then [R]; po
    else 0
  | if "SEA_W" then [W]; po
    else 0
```

`po`: program order

`ctrl`: control dependence

`ISB`: instr. synch. barrier

```
(* context-sync-events *)
let CSE =
    ISB
  | if "FEAT_ExS" & ~"EIS"
    then 0 else TE
  | if "FEAT_ExS" & ~"EOS"
    then 0 else ERET
```

EIS: Entry to exception is context synchronizing

EOS: Exit from exception is context synchronizing

TE: Take exception

ERET: Entering/returning from an exception

Their model:

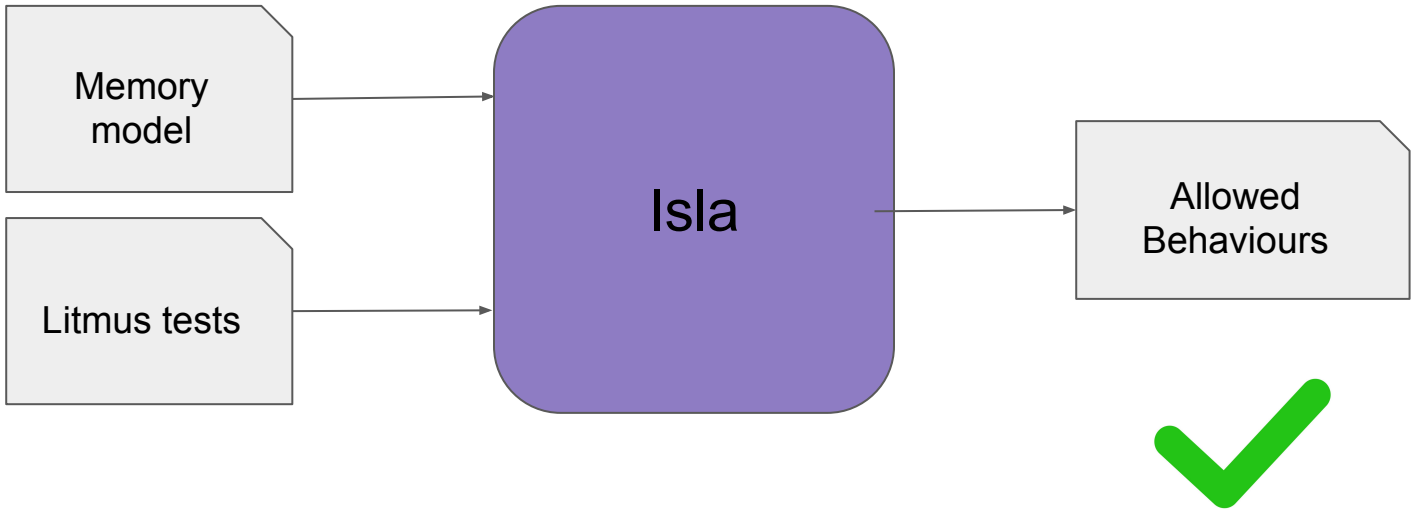
Supports most precise asynch exceptions

Does *not* support interprocessor interrupts & the generic interrupt controller

Async-ordered-before (`asyncob`): asynch. events cannot be executed speculatively

More model axioms can be found in the paper

Executable-as-a-test-oracle implementation (§5.1)



Challenges in defining precision (§6)

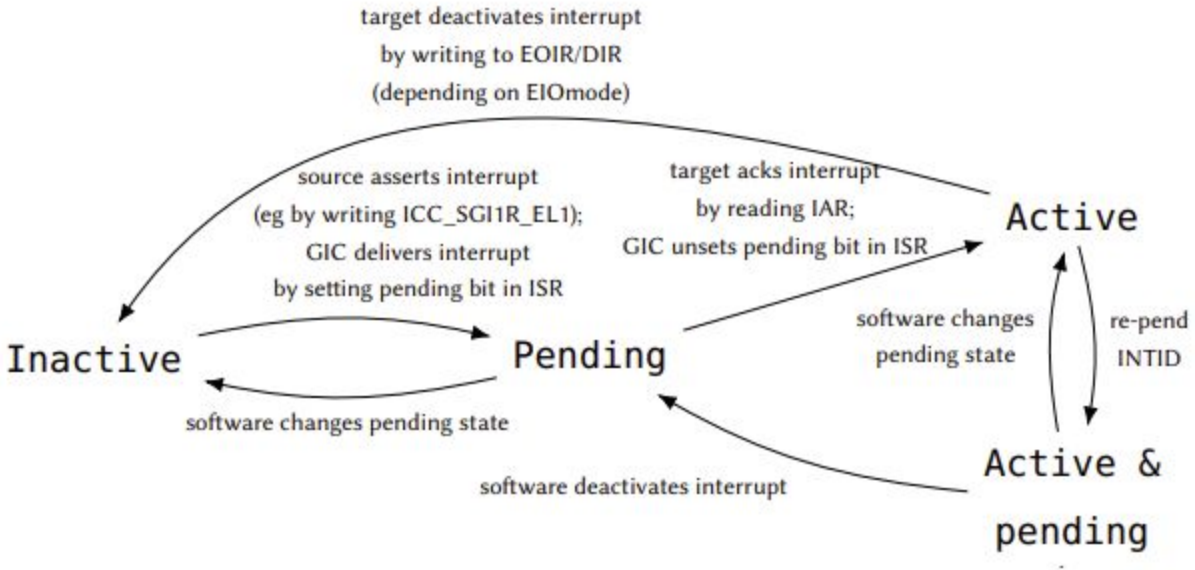
- Precise exception: HW thread (PE) state and mem sys state appear as though instructions up to exception in the instruction stream took place
  - Registers and memory values can be UNKNOWN
    - These side effects are visible (handling unknowns is not codified)
  - Intent of precision is to resume execution after exception

- All exceptions in ARM are precise except for asynchronously reported external memory errors

SW generated interrupts (SGIs) (§7)

- aka Inter-processor interrupts (IPIs)
- Interrupts are generated from a source (eg HW thread) for an event (eg SGI)
  - Sent to interrupt controller
    - Distributor - routes interrupts to cores
    - Redistributor - (per thread) maintain thread local state for each interrupt
  - Identified by INTID
  - Each HW thread has Interrupt Status Register
- Note: the authors suggests additions to the axiomatic model to account for these things

A generic concept: Generic Interrupt Controller



- Interrupts can be delivered at any time, with multiple pending delivered in any order

System-wide Memory Barrier

- The HW thread that issues the interrupt notifies the other HW threads and waits for their response
  - Kernel RCU (synchronize_rcu)
    - an interrupt is taken before or after the critical read section, but not during it
      - Interrupts are masked when a HW thread is in this critical section

Scope & Limitations

- Small test suite
- Don’t provide semantics for imprecise exceptions
- Don’t precisely model relaxed behavior of system registers
- Don’t provide a detailed model of the Generic Interrupt Controller

- This paper is meant to lay the foundation for future work regarding exceptions and interrupts

Conclusion

- General Goal: Clarifying the definition of precision on relaxed architectures
- Contributions:
  - Extends Arm-A model to cover exceptions
  - Implements axiomatic model to delve into effects of synchronization
  - Discusses model for SGIs

Time to Discuss

1. Why do you think this won best paper at ISCA 2025?
2. Why is this an important topic? How strict should our models be when designing processors?
3. What can go wrong if we don’t have/follow an adequate model for exceptions?
4. Are there any additional limitations you can think of?
5. What is the connection between this and Cohmeleon?
6. What did you find most interesting?
7. Would you read this paper?
8. Recall question: what is an exception? What is a precise exception?