Precise exceptions in relaxed architectures

Ben Simner
University of Cambridge
Cambridge, United Kingdom
Ben.Simner@cl.cam.ac.uk

Alasdair Armstrong
University of Cambridge
Cambridge, United Kingdom
Alasdair.Armstrong@cl.cam.ac.uk

Thomas Bauereiss
University of Cambridge
Cambridge, United Kingdom
Thomas.Bauereiss@cl.cam.ac.uk

Brian Campbell
University of Edinburgh
Edinburgh, United Kingdom
Brian.Campbell@ed.ac.uk

Ohad Kammar
University of Edinburgh
Edinburgh, United Kingdom
ohad.kammar@ed.ac.uk

Jean Pichon-Pharabod
Aarhus University
Aarhus, Denmark
Jean.Pichon@cs.au.dk

Peter Sewell
University of Cambridge
Cambridge, United Kingdom
Peter.Sewell@cl.cam.ac.uk

Abstract

To manage exceptions, software relies on a key architectural guarantee, *precision*: that exceptions appear to execute between instructions. However, this definition, dating back over 60 years, fundamentally assumes a sequential programmers model. Modern architectures such as Arm-A with programmer-observable relaxed behaviour make such a naive definition inadequate, and it is unclear exactly what guarantees programmers have on exception entry and exit.

In this paper, we clarify the concepts needed to discuss exceptions in the relaxed-memory setting – a key aspect of precisely specifying the architectural interface between hardware and software. We explore the basic relaxed behaviour across exception boundaries, and the semantics of external aborts, using Arm-A as a representative modern architecture. We identify an important problem, present yet unexplored for decades: pinning down what it means for exceptions to be precise in a relaxed setting. We describe key phenomena that any definition should account for. We develop an axiomatic model for Arm-A precise exceptions, tooling for axiomatic model execution, and a library of tests. Finally we explore the relaxed semantics of software-generated interrupts, as used in sophisticated programming patterns, and sketch how they too could be modelled.

CCS Concepts

• **Computer systems organization** → **Architectures**; **Multicore architectures**; • **Theory of computation** → *Axiomatic semantics.*

Keywords

Computer architecture, relaxed memory, exceptions and interrupts, exception handling, semantics

ACM Reference Format:
Ben Simner, Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Ohad Kammar, Jean Pichon-Pharabod, and Peter Sewell. 2025. Precise exceptions in relaxed architectures. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3695053.3731102

1 Introduction

Hardware exceptions (and their many variants: interrupts, traps, faults, aborts, etc.) provide support for many exceptional situations that systems software has to manage. This includes explicit privilege transitions via system calls, implicit privilege transitions from trappable instructions, inter-processor software-generated interrupts, external interrupts from timers or devices, recoverable faults like address translation faults, and non-recoverable faults like memory error correction faults.

To confidently write concurrent systems code that handles exceptions, e.g. mapping on demand at page faults, programmers need a well-defined and well-understood semantics. The definition given in modern architectures (e.g. in the current Arm-A documentation) is basically unchanged since the IBM System/360, roughly as Hennessy and Patterson [34] state: *"An exception is imprecise if the processor state when an exception is raised does not look exactly as if the instructions were executed sequentially in strict program order".* However, on pipelined, out-of-order processors with observable speculative execution, exceptions have subtle interactions with relaxed memory behaviour which have not previously been investigated.

1.1 Contributions

In this paper, we investigate the relaxed concurrency semantics of exceptions on modern high-performance architectures. We focus on the Arm-A application-profile architecture as a representative example, although we expect that the challenges we describe also appear in other, similarly relaxed, architectures. This work involved detailed discussions with Arm senior staff, including the Arm Chief Architect and an Arm Generic Interrupt Controller (GIC) expert. Our contributions are:

- We clarify the concepts and terminology needed to discuss exceptions in relaxed-memory executions (§2).
- We explore the relaxed behaviour of exceptions: out-of-order and speculative execution, and forwarding across exception entry/exit boundaries (§3). This is based on discussions with Arm and testing of several processor implementations, using a test harness for hardware testing of exceptions, and a library of hand-written litmus tests.
- We explore the semantics of memory errors (§4). In Arm-A, these can generate *external aborts*. Some implementations, including server designs, may exhibit *synchronous* external aborts. Such implementations rule out load-buffering (LB) relaxed behaviour, which substantially curtails how relaxed observable behaviour is.
- We develop an axiomatic model for Arm-A precise exceptions (§5). We extend Isla [13] to support both ISA and relaxed-memory concurrency aspects of exceptions, and we use it to evaluate the axiomatic model on tests.
- We identify and discuss the substantial open problem of what it means for exceptions to be precise in relaxed setting (§6). We characterise key properties that a definition should respect, and highlight the challenge of giving a proper definition of precision when relaxed behaviour is allowed across exception boundaries.
- Finally, we explore a significant use-case of exceptions that benefits from the clarification of their interaction with relaxed memory: the relaxed semantics of software-generated interrupts as used for sophisticated low-cost synchronisation, e.g. in Linux's RCU [52] and Verona [14] (§7). We sketch this in an axiomatic model.

This is an essential part of the necessary foundation for confidently programming systems code, building on previous work that has clarified 'user' relaxed concurrency [1–3, 7–9, 13, 21, 28–32, 37, 57, 58, 60–62, 64, 68] and complementing recent work on the systems aspects of instruction fetch [67] and virtual memory [4, 66]. It helps put processor architecture specifications such as Arm-A on an unambiguous footing, where the allowed behaviour of systems-code idioms can be computed from a precise and executable-as-test-oracle definition of the architecture.

1.2 Scope and limitations

Our models cover important use cases of exceptions, but there remain several questions to be addressed by future work. Our testing suite is relatively small, and a much larger corpus would give higher confidence, and ideally could be auto-generated [5, 9, 35]. We do not give semantics to imprecise exceptions, and it is unclear how to do so at an architectural level. For our specific modelling of Arm: we do not define the behaviour of 'constrained unpredictable', and merely flag when it is triggered. Clarifying it will require substantial extensive discussions with Arm architects, likely affecting the wording in the architectural specifications, beyond the scope of this paper. We do not try to precisely model the relaxed behaviour of system registers, but merely sufficient conditions for conservative use cases in the context of exceptions (§3.1). We do not model switching between Arm FEAT_ExS modes (§3.5): they are supported architecturally, but are not commonly implemented. We rely on a specific configuration to illustrate the use of interrupts for synchronisation (§7), without detailed modelling of the Arm Generic Interrupt Controller (GIC), or other system-on-chip (SoC) aspects. The GIC is a complex hardware component, with a 950-page specification [11, H.b], and modelling it in full would be a major project in itself. This work is validated by substantial discussion and hardware testing, but more extensive testing on more devices is always desirable; we hope that our work will spur such additional testing on devices not available to us. Finally, while we believe our models correctly capture the Arm architectural intent, and that it gives a solid basis for programmers, this paper is not an authoritative definition of the architecture, which is in any case subject to change.

2 Arm-A architectural concepts for exceptions

We start by recalling, and then refining the architectural concepts for exceptions in Arm.

2.1 Exception taxonomy

Arm-A defines multiple kinds of exception [10, D1.3.1, p6060]: *Synchronous exceptions* (supervisor/hypervisor calls, traps, data/instruction, page faults, etc.) and *interrupts* (IRQ/FIQ from processors/peripherals/timers and system errors).

The preferred return address of synchronous exceptions has an architecturally defined relationship with the instruction that caused them. Such exceptions are *precise*. This means, roughly, that they are observed at particular points in the instruction stream, and so can use the preferred return address to resume executing it after handling the exception. All interrupts are precise apart from external system aborts (SError), for which it is implementation-defined (per-kind) whether they are precise. Such errors may or may not be recoverable in practice. For example, an unrecoverable imprecise error may be generated by late detection of an uncorrectable memory error correction error. In §3, we discuss how the choice of mechanism used to report external aborts affects the relaxed behaviour.

2.2 Architectural exception machinery

In Arm-A, when an exception is taken, execution jumps to the exception vector, an offset from the appropriate vector base address register (VBAR) value depending on the kind of exception. The appropriate exception syndrome register (ESR), fault address register (FAR), and exception link register (ELR) are written with information about the cause and the preferred return address. In some cases, the exception level (EL) register value, ranging in increasing privilege from 0 to 3, is also changed. Exception handlers typically use ERET to return from an exception, which restores some processor state and branches to the address in the appropriate ELR. Most of these system registers (VBAR, ESR, etc.) are banked.

2.3 Instructions and instruction streams

One often thinks of processors as executing *instructions* in some *instruction sequence*, and common terminology is based on those two concepts. For example, the Arm manual has around 60 instances of *instruction stream* or *execution stream*. However, to account for relaxed behaviours and exceptions, we must refine these concepts.

2.3.1 From instructions to fetch-decode-execute instances. Exceptions can arise at multiple points within the fetch-decode-execute cycle, including during the fetch and decode, when there is no 'instruction'. For Armv9.4-A, much of this is captured in an Arm top-level function written in the Arm Architecture Specification Language (ASL).

We have then integrated this into Sail-based tooling to obtain an executable-as-test-oracle semantics of the sequential ISA aspects of Armv9.4-A with exceptions (§5.1). A highly simplified outline of a single-instruction slice of the (400k line) instruction semantics is:

```
function __TopLevel() =
  // in TakePendingInterrupts:
  if IRQ then AArch64_TakePhysicalIRQException()
  if SE then AArch64_TakePhysicalSErrorException(...)
  // in AArch64_CheckPCAlignment:
  if pc[1..0] != 0b00 then AArch64_PCAlignmentFault()
  // in __FetchInstr:
  opcode = AArch64_MemSingle_read(pc, 4) // read memory
  // in __DecodeA64:
  match opcode
    [1,_,1,1,1,0,0,1,0,1,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,
     _,_,_,_] =
      // the semantics for one family of instructions,
      // including loads LDR Xt,[Xn]
      // execute_aarch64_instrs_memory_single_general_
      // immediate_signed_post_idx(n,t,...)
      let address = X_read(n, 64) // read register n
      let data : bits('datasize) = // read memory
        Mem_read(address, DIV(datasize,8))
      // write register t
      X_set(t, regsize) = ZeroExtend(data, regsize)
```

Executing this semantics may lead to one or more kinds of exception, calling the ASL/Sail function `AArch64_TakeException()`. This function writes the appropriate values to registers, e.g. computing the next PC, exception level, etc. and terminates this `__TopLevel()` execution. So instead of 'instruction' instances, we refer to *fetch-decode-execute* (FDX) instances, each of which is a single execution of `__TopLevel()`.

2.3.2 Fetch-decode-execute trees and streams. One must relate the out-of-order speculative execution of hardware implementations and the architectural definition of the allowed behaviours. We will use the following concepts, well-understood when modelling relaxed memory without exceptions. At any instant, each hardware thread may be processing, out-of-order and speculatively, many instructions (each corresponding to an architectural FDX instance). Partially executed instances are restarted or discarded if they would violate the intended semantics (e.g. on mispredicted branches).

One can visualise the state of a single core abstractly as a tree of partially and completely executed instances, as in Fig. 1 (top). Abstract-microarchitectural operational models use this abstraction [28, 29, 32, 58, 60, 61]. We depict the retired (committed) FDX instances as solid dark green, and partially/tentatively executed in-flight instances as light green. The arrows depict program order. Committed instances can be program-order after in-flight instances, and non-committed instances may need to be restarted. Eventually

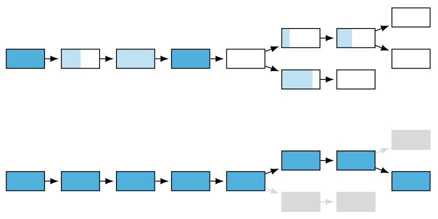


Figure 1: Top. The tree of (partially) executed FDX instances at one time, in hardware or operational model execution. Bottom. The sequence of architecturally executed FDX instances in a completed execution.

all FDX instances for this hardware thread will be either committed or discarded, e.g. as in Fig. 1 (bottom). These are the *architecturally executed* FDX instances. The architecture definition, and any formal semantics thereof, have to define which such sequences are allowed for each thread. This definition includes the register content; memory read values; and their relationships with other threads, as determined by the relaxed concurrency model. Architectural axiomatic concurrency models, e.g. [1–3, 7–9, 13, 21, 30, 31, 35, 37, 62, 64, 68], use candidate executions containing the events just from these architecturally executed instances. Note that the events comprising architectural FDX instances are abstract: they do not represent any individual sequence of microarchitectural operations as one would need for side-channel analysis [23, 24, 53] or to reason about individual microarchitectures [48]; we give no bound on the extent of non-architecturally-executed instances except in that they cover what one needs to capture the architectural bound.

The Arm prose specification in Fig. 2 (top) previously attempted to capture the relationship between implementation execution (out of order and speculative) and the architectural definition of allowed behaviour in terms of a notion of "simple sequential execution". As the prose says, simple sequential execution does not hold for the intended relaxed-memory architecture. We propose a more correct rephrasing that allows for exceptions and other systems phenomena in Fig. 2 (bottom).

Fig. 3 depicts a tree of instances involving exception entry (`svc`) and return (`eret`). Arm-A allows implementations to observe the exception handling instances as executing before program-order previous instances have been retired, and similarly exception return. Exception entry and return may never be observed as starting to execute speculatively, however, and so the three speculative branches may not observe exception entry or return instances. Precision must account for these allowed and prohibited relaxed behaviours.

3 Relaxed behaviour of precise exceptions

Exceptions change the control flow and processor context, that is, the collection of system and special registers which control the execution of the machine, such as the current exception level (`PSTATE.EL`), masking of interrupts (`PSTATE.{D,A,I,F}`), processor flags, etc. However, changes to the context may not take effect immediately, and so, to ensure that program-order-later instructions see

Architecturally executed An instruction is architecturally executed only if it would be executed in a simple sequential execution of the program. [...]
Simple sequential execution The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and Arm does not expect this model to correspond to a realistic implementation of the architecture.

Architecturally executed A candidate execution can be architecturally executed if it is composed of a sequence of FDX instances for each thread that together satisfy the Arm concurrency model [extended to cover exceptions, as described here, and other systems phenomena], starting from the machine initial state.

Figure 2: Arm prose specification [10, Glossary, p14749] (top) and our suggested rephrasing (bottom).

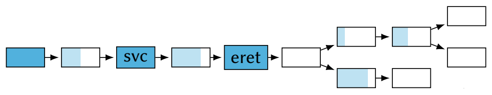


Figure 3: The tree of partially and completely executed FDX instances with exceptions, in hardware or operational model execution. Instructions may execute out-of-order across exception boundaries, requiring a modern definition for precision.

such changes, exceptions usually come with context synchronisation. It is this context synchronisation which imposes ordering, and we show how, without such context synchronisation, we observe reordering across exception boundaries. For this reason, exceptions are usually context-synchronising on Arm.

There are many things that can trigger exceptions. While exceptions like interrupts and page faults are likely the most common, they may come with extra synchronisation and/or non-determinism. The simplest way to explore the relaxed behaviours is therefore to use 'exception-generating instructions', such as system calls (using the Arm `SVC` instruction), which unconditionally generate an exception at a particular program point. These provide a baseline for precision, and therefore we use them in our exploration of the behaviour of exceptions in the remainder of this section; we return to discuss other exceptions later on.

In this section, we explain relaxed behaviour of precise exceptions through litmus tests, the usual standard for succinctly cataloguing the relaxed behaviours allowed by an architecture [8, 9, 13]. Litmus tests are small programs capturing specific software patterns or hardware mechanisms, whose outcome depends on some kind of out-of-order execution. Precise exceptions do not change the memory model between exception boundaries, and so the interesting questions concern out-of-order execution across exception boundaries.

We will talk about context synchronisation in detail (§3.1), explore the baseline out-of-order execution across exception boundaries (§3.2), then the stronger behaviour of specific types of exceptions (§3.3), touch on how the instruction semantics needs to be adapted (§3.4), and finally discuss a corner case disabling context synchronisation (§3.5).

3.1 Context-synchronisation

Updates to the context, such as writes to system registers, need synchronisation to be guaranteed to have an effect. We do not model the behaviour of such context-changing operations when such synchronisation is not performed. Instead, we merely identify when and how exceptions are context-synchronising, and note that this has a knock-on effect on memory accesses.

Architecturally, a context synchronisation event guarantees that no instruction program-order-after the event is observably fetched, decoded, or executed until the context-synchronising event has happened. A simple microarchitectural implementation for context synchronisation is to flush the pipeline: restarting all program-order-later instances once the context-synchronising effect occurs. More complex implementations may be more clever, as long as they preserve the semantics. Software can explicitly generate context synchronisation events by issuing an Instruction Synchronisation Barrier (`ISB`). Context synchronisation can also happen implicitly, for example on exception entry and exit. This is the case in Arm, except in a rare use case we return to in §3.5.

The effect of context synchronisation events in exception boundaries is that any instance after the boundary has an `ISB`-equivalent dependency on the instances before the boundary. This mechanism implies the following fundamental invariant: *context synchronising exceptions are never taken speculatively*, and it limits speculation to the same well-understood extent as `ISB` limits speculation. This invariant has interesting interactions with external aborts, which we discuss in §4.

3.2 Relaxed behaviours

In this section, we explore the relaxed behaviour of exceptions, with a selection of litmus tests from our larger suite of 61 hand-written tests. For each test, we include whether the behaviour is allowed in our understanding of the architectural intent; and a candidate execution graph. We mark behaviours as allowed/disallowed based on discussions with Arm architects.

3.2.1 Out-of-order execution across exception boundaries. Exception boundaries do not act as memory barriers, so loads and stores may be executed out-of-order over an exception entry or an exception exit or the composition of both.

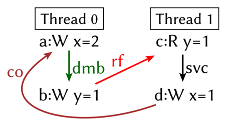
Figure 4 contains an illustrative sample of three such shapes. Each test contains the code listing with the pertinent (relaxed) final state and architectural intent, the graph of architecturally-executed FDX instances comprising the candidate execution.

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

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

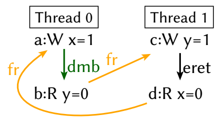


SB+dmb.sy+eret AArch64

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

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



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

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

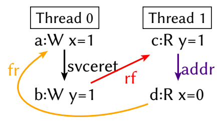


Figure 4: Reads and writes may be executed out-of-order across exception entry, exit, or even both.

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

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

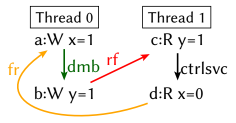


Figure 5: Context synchronising exceptions are not executed speculatively.

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

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

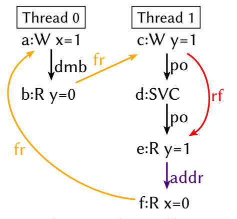


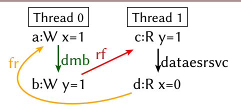
Figure 6: Forwarding into a non-speculative handler.

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

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



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

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

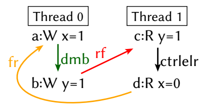


Figure 7: System registers and context synchronisation

3.2.2 Speculative exception entry or return. The invariant 'context synchronising exceptions cannot be taken speculatively' imposes the same kind of barrier as a `ctrlisb` dependency would impose between program-order-previous instances and the instances in the handler. The control dependency is due to the branching to the handling code, and the `ISB` dependency is due to context synchronisation. As a consequence, the two behaviours in Figure 5 are forbidden. On architectures that allow the `FEAT_ExS` extension, they would be allowed when the exception entry/exit is not context synchronising, i.e., when the corresponding `EIS`/`EOS` bit is cleared. This mechanism also explains why we do not observe load-load reordering on the Raspberry Pi devices, but we do observe them on the ODROID-N2+ (exhibited by the test `MP+dmb+svc` which can be found in the extended version [65]). These machines exhibit the same behaviour as they would for the corresponding `MP+dmb+isb` behaviour from previous work.

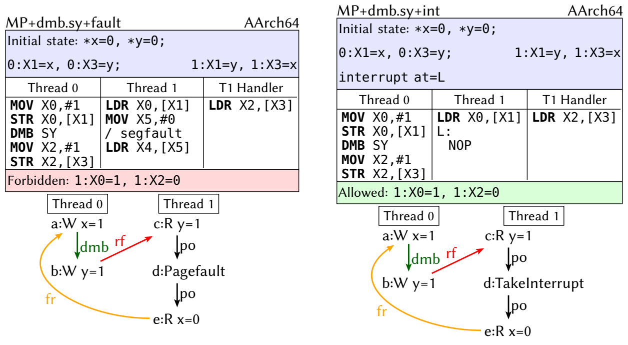
3.2.3 Privilege level. The privilege level (`PSTATE.EL`) has little to no additional effect on the behaviours we present: their allowed/forbidden status remains the same whether the privilege goes up/down in entry/exit or remains the same. The one exception to this principle is the effect a privilege change has on non-faulting translation table walks. A non-faulting translation walk for an instance program-order-before a privilege-changing exception entry from `ELn` may be reordered with the entry, but would then also be reordered with every subsequent exception boundary until the privilege level returns to `ELn`. Explaining this case in full detail would require substantial details of Arm's virtual memory architecture [66], and we leave it to future work.

3.2.4 Forwarding writes. It is permitted for writes to be forwarded from a store to a read across exception entry and return (`SB+dmb+rfisvc-addr` in Figure 6).

3.2.5 Dependency through system registers. Where exceptions are taken to and returned to are part of the context, and must be read by exception taking and returning, and so they can be involved in register dependency chains. Here, we do not characterise the general effect of such dependencies, but focus on the effect exceptions have on them. Dependencies on system register accesses compose with ordering from context synchronisation events to program-order-later instructions. Test `MP.EL1+dmb+dataesrsvc` in Fig. 7 demonstrates that a write to the system register `ESR` that depends on a read forbids reordering this read across the boundary, even though resolving the dependency does not affect the exception.

The `ELR` register is a special-purpose register, and is therefore 'self-synchronising'. Therefore, writes into the `ELR` do not need context synchronisation to guarantee that they are seen by program-order-later instructions, and this means that dependencies into the `ELR` are preserved (see Fig. 7).

This has two related subtleties, and is currently under investigation by Arm. The Software Thread ID Register (`TPIDR`) is a system register in which the operating system can store thread identifying information, but has no relevant indirect effects. Further testing



```
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      | LDR X0,[X1]     | LDR X2,[X3]
STR X0,[X1]    | MOV X5,#0       |
DMB SY         | / segfault      |
MOV X2,#1      | LDR X4,[X5]     |
STR 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      | LDR X0,[X1]     | LDR X2,[X3]
STR X0,[X1]    | L:              |
DMB SY         |   NOP           |
MOV X2,#1      |                 |
STR X2,[X3]    |                 |
Allowed: 1:X0=1, 1:X2=0
```

Figure 8: Different exception kinds can have different behaviour.

and discussions may clarify whether it forbids reordering. While dependencies through special-purpose registers are preserved, context synchronisation does not necessarily need to wait for those writes, and so these dependencies do not necessarily pass to instructions after context synchronisation (in contrast to system register writes).

3.2.6 Ordering from asynchronous exceptions. Asynchronous exceptions cannot be taken speculatively. Therefore, all instructions program-order-after an asynchronous exception happen after that exception.

3.3 Exception-specific mechanisms

Not all exception kinds are equal. For example, when an implementation supports the Enhanced Translation Synchronisation feature (`FEAT_ETS2`), the translation-table-walks which generate translation faults (pagefaults) gain additional ordering from program-order-previous instances. Figure 8 compares the MP test involving a page-fault (`MP+dmb.sy+fault`, forbidden under ETS) and the same shape involving an asynchronous interrupt (`MP+dmb.sy+int`, allowed). As such, combining the exceptions model here with any of the existing models (for virtual memory, cache maintenance, memory tagging, transactional memory, etc.) would require clarification around the domain-specific exceptions associated with those features. We are aware that the specification of additional mechanisms per exception-kind is an active area of interest for Arm.

3.4 Intra-instruction exceptions

Wherever possible, we want to interpret the intra-instruction ASL ordering as preserved, both for conceptual simplicity, memory-model tool execution, and reasoning. This has previously been possible except in a few specific cases that are inherently concurrent, e.g. instructions that do multiple accesses. Exceptions introduce a new interesting case for instructions that do a register writeback concurrently with a memory access. For example, `STR` (immediate) has a "Post-index" and a "Pre-index" versions [10, C6.2.365, p2442]. The post-index `STR Xt, [Xn], #8`, for example, stores the value in `Xt` to the address initially in register `Xn` and adds 8 to `Xn`. The Arm ARM ASL for `STR` puts that register write at the end, after the memory access has completed. The architectural intent is that program-order-later instances that depend on `Xn` can go ahead early, e.g. before the data in register `Xt` is available to be written to memory, and this has been observed in practice [36].

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.

Previous work captured this allowed by having the register writeback before the memory access in the instruction semantics. However, exceptions require more care: when the memory access generates an exception, the writeback register should appear unchanged to instances after the exception boundary.

3.5 Disabling context synchronisation

On Arm, the optional `FEAT_ExS` feature provides two new fields in the system control register to disable context synchronisation on exception entry or return, respectively: `EIS` and `EOS`. While the semantics is clear for these systems, the programming model is unpredictable and hard to program correctly, and so this configuration is rarely encountered in practice.

3.6 Hardware results

We extend the testing harness of Simner et al. [66] to collect preliminary results from hardware, on the following implementations: AWS M6/7/8G instances (with Neoverse N1/V1/V2), an ODROID N2+ (on the big Arm Cortex-A73 cores), an Apple M2, and Raspberry Pi 3B+/4B/5 (with Arm Cortex-A53/A72/A76). Results can be found in Fig. 9, given as observations over the total number of runs. Results marked with U are allowed, but not observed on that device. For the complete hardware results, see the extended version [65].

4 Synchronous external aborts

The memory system may detect errors such as data corruption independently of the MMU or Debug hardware, e.g., using parity bits or error correcting code. In those cases, it will signal the error by a class of exceptions called *external aborts*. The architecture does not define at which granularity implementations may report such aborts synchronously, which we refer to as *synchronous external aborts* (SEAs). Instances program-order-after a potential cause for synchronous external aborts are considered speculative until this external abort can be ruled out, resulting in stronger behaviour (§4.1). In an implementation that always reports external aborts asynchronously, the later instances become non-speculative earlier, allowing them to exhibit weaker behaviours. When external aborts are reported asynchronously, the simplest recovery is to wind down the aborting process. To allow programmers more reliable recovery, implementations can support the Reliability, Availability, and Serviceability (RAS) extension. This extension is a substantial component of the architecture, far beyond the scope of this work. Here, we are merely taking the first steps, describing a baseline of behaviours in a very constrained setting, that further work may be able to extend to account for the RAS.

Whether any external abort could be reported synchronously is implementation-defined, with no architected way of identifying the choice. Nevertheless, the choice impacts the permissible relaxed behaviours.

4.1 Behaviour resulting from synchronous external aborts

There is an asymmetry between reads and writes with respect to speculation: writes cannot be propagated speculatively, whereas reads can be satisfied speculatively. We will therefore consider the store and load cases separately.

If a store may generate an SEA, then program-order-later instances are speculative until the store has (at least) propagated to memory. In that case, write-write re-ordering (MP+po+addr) is forbidden. Reads program-order-after writes are permitted to execute speculatively anyway, and so the presence of these SEAs does not restrict their ability to execute early.

More interestingly, if a load may generate an SEA, then program-order-later instances are speculative until the load has completed all its reads, and is non-restartable. This forbids interesting tests which would otherwise be allowed, namely context-synchronisation after reads (e.g. MP+dmb.sy+isb), which must wait for that control flow to be resolved [63]; and writes program-order-after reads (e.g. LB+pos), since writes must not be propagated speculatively [63].

4.2 Load buffering and the out-of-thin-air problem

This has an important and hitherto not well-understood impact on programming-language concurrency models. Ruling out LB enables substantially simpler design of programming language concurrency models: they can execute instructions in-order and merely keep a history of the writes seen so far, e.g. [46], and thereby avoid the notorious out-of-thin-air problem [15]. These simpler semantics support a line of model checkers for C/C++ and LLVM [42–44]. In contrast, the presence of LB seems to require significant sophistication [3, 15, 16, 19, 38, 39, 55, 56].

5 An axiomatic model of exceptions

We now give a formal semantics that describes the concurrent behaviour of precise exceptions on Arm-A. We give it as an extension of the previous model of Pulte et al. [58], a predecessor of the current Arm model [25], in the standard 'cat' format [9, 13], in Figure 10.

While the model captures the architectural intent as we understand it, the architecture remains the sole responsibility of Arm; the intent may change over time and the model presented here is not officially endorsed by Arm.

The model is parameterised along two axes:

- FEAT_ExS corresponds to the feature of the same name being implemented; we do not support runtime changes of the related SCTLR_ELx.{EIS,EOS} fields, and so fix them as variants.
- SEA_R and SEA_W correspond to the IMPLEMENTATIONDEFINED choice of whether loads or stores may generate synchronous external aborts.

```
"Arm-A exceptions"

include "cos.cat"
include "arm-common.cat"

(* 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

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

let ASYNC =
  TakeInterrupt

(* observed by *)
let obs = rfe | fr | co

(* dependency-ordered-
     before *)
let dob =
    addr | data
  | speculative ; [W]
  | speculative ; [ISB]
  | (addr | data); rfi

(* atomic-ordered-before *)
let aob =
   rmw
 | [range(rmw)]; rfi; [A|Q]

(* barrier-ordered-before
     *)
let bob =
    [R] ; po ; [dmbld]
  | [W] ; po ; [dmbst]
  | [dmbst]; po; [W]
  | [dmbld]; po; [R|W]
  | [L]; po; [A]
  | [A | Q]; po; [R | W]
  | [R | W]; po; [L]
  | [dsb]; po

(* contextually-ordered-
     before *)
let ctxob =
    speculative; [MSR|CSE]
  | [MSR]; po; [CSE]
  | [CSE]; po

(* async-ordered-before *)
let asyncob =
    speculative; [ASYNC]
  | [ASYNC]; po

(* Ordered-before *)
let ob = (obs | dob | aob |
  bob | ctxob | asyncob)+

(* Internal visibility
     requirement *)
acyclic po-loc | fr | co |
     rf as internal

(* External visibility
     requirement *)
irreflexive ob as external

(* Atomic: Basic LDXR/STXR
     constraint to forbid
     intervening writes. *)
empty rmw & (fre; coe) as
     atomic
```

Figure 10: Arm-A exceptional model (greyed out parts are unchanged from the original model).

Most current hardware does not support FEAT_ExS, and moreover, we expect that most software would not use it. However, its semantics is relatively straight-forward as we understand it, and so we include it in our model, although without the hardware validation we have for the non-ExS fragment.

We add new events to the candidate execution: TE (take exception) and ERET, which correspond to the synchronisation points (whether they *are* synchronising) of taking or returning from an exception; and MRS and MSR events, for reading and writing system registers, corresponding to the Arm MRS and MSR instructions which change the context.

Exceptions and program order. We include all the new events in program-order. This includes the events from instructions directly before and after taking or returning from an exception.

Interrupts. While this cat model does not support inter-processor interrupts and the generic interrupt controller (see §7 for a draft extension to support them), it does support other precise asynchronous exceptions (e.g. timers).

Ordered-before. We expand ordered-before:

- Wherever `ctrl|(addr;po)` was used before, we also include instructions program-order-after reads or writes when in the relevant SEA variant. With those variants, the instructions program-order-after those events are speculative up until the memory access has completed.
- The previous model's use of ISB was purely for its context synchronisation effect. Accordingly, wherever `[ISB]` was used before, we include exception entry (TE) and exit (ERET), unless we are in the variant where context synchronisation on those events is disabled.
- We extend barrier-ordered-before with the DSB barriers. The barrier event classes are upwards-closed, so that DSB.SY is included in all the dmb events.
- We add a context-ordered-before (ctxob) sub-clause to the ordered-before relation, which captures the ordering of context-changing operations and context-synchronisation: namely, that context-changes and context-synchronisation cannot happen speculatively; that all context-changes are ordered before any context-synchronisation; and that no instruction program-order-after context-synchronisation can be executed until the synchronisation is complete.
- We add an async-ordered-before (asyncob) clause to ordered-before, capturing that asynchronous events (such as interrupts) cannot be done speculatively, and instructions program-order-after them may not happen before the asynchronous event which precipitated them.

5.1 Executable-as-a-test-oracle implementation

We implement the model in Isla [13], an SMT-based executable oracle for axiomatic concurrency models (and ISA semantics). Isla takes as input a memory model in herdtools-like cat format, and a litmus tests. To support tests with asynchronous exceptions, we added a construct to specify a label where the exception will occur, so that Isla then pends an interrupt at that program point.

The instruction semantics we use is a translation into the Sail language of the Armv9.4-A ASL specification, including the top-level function provided by Arm [17]. The translation process [12] is mostly automatic, requiring select manual interventions mostly due to differences in the type systems of ASL and Sail. We also added patches to support the integration with Isla, in particular adding hooks to expose information about exceptions being taken in a form that can be readily consumed by Isla. In doing so, we encountered and fixed some bugs in the ASL model related to uses of uninitialised fields in data structures, as well as missing checks for implemented processor features that led to spurious system register accesses.

For all the (non-IPI) tests, Isla, the architectural intent as we understand it, and the results of hardware testing from §3.2 are consistent.

6 Challenges in defining precision

The phenomena we describe in §3 highlight that the historical, naive definition of precision does not account for relaxed memory. The open problem is then *how to adequately define precision in a relaxed-memory setting*. This challenge is hinted at in the way the Arm reference manual [10, D1.3.1.4, p6060] defines precision as:

> An exception is *precise* if on taking the exception, the hardware thread (aka processing element, PE) state and the memory system state is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where the exception was taken from, and none afterwards. [except that in certain specific cases some registers and memory values may be UNKNOWN]

This definition explicitly allows various side effects of an instruction executing when an exception is taken to be visible. The details are intricate, but in outline: registers that would be written by the instruction but which are not used by it (to compute memory access addresses) can become UNKNOWN, and for instructions that involve multiple single-copy-atomic memory writes (e.g. misaligned writes and store-pair instructions), where each write might generate an exception (e.g. a translation fault), the memory locations of the writes that do not generate exceptions become UNKNOWN. These side effects could be observed by the exception handler, and the memory write side effects could be observed by other threads doing racy reads. Hardware updates to page-table access flags and dirty bits, and to performance counters, could also be observable. This means that the abstraction of a stream of instructions executed up to a given point does not account for the relaxed-memory behaviour.

Arm *classify* particular kinds of exceptions as precise or not, but all the above makes it hard to *define* in general what it means for an exception to be precise in a relaxed setting.

The ultimate architectural intent of precision is that it is sufficient to meaningfully resume execution after the exception. For example, for software that does mapping on demand, when an instruction causes a fault by accessing an address which is not currently mapped, the exception handler will map that address and return. This means that re-executing the original instruction will overwrite these UNKNOWNs, and will have ordering properties much like the original instruction would have had if the mapping had already been in place.

Our models are complete enough to reason about such cases in concrete examples. However, a general definition of precision, and the accompanying reasoning principle, would have to capture assumptions about the exception handler and its concurrent context to ensure that they do not observe the above side effects.

More straightforwardly, the above definition of what becomes UNKNOWN would have to be codified, as that is not currently in the ASL architectural pseudocode. Without a clear definition of precision architectures must independently enumerate the possible relaxations across exception boundaries (as we do in §3 for Arm).

Exceptions may also be *imprecise*, in which case the behaviour is very loosely constrained. The current architectural intent does not give well-defined guarantees in the presence of imprecise exceptions, and models that account for imprecision likely need to expose more of the microarchitectural state than we capture here [33]. All exceptions in Arm are precise except for those external memory errors which are not reported synchronously (§4), which we do not cover.

7 Software-generated interrupts

Inter-processor interrupts (IPIs), known as software-generated interrupts (SGIs) on Arm, are an important synchronisation mechanism available to software. They are used throughout systems software to signal other threads, including within the Linux kernel (in its RCU synchronisation mechanism), in software (via Linux's `sys_membarrier`), e.g. in JITs [67], and in programming language runtimes (e.g. in Microsoft'sVerona [14, 20]). Such use of SGIs critically depends on a detailed understanding of the interaction of exceptions with relaxed-memory behaviour.

To manage the sending, routing, prioritisation, and delivery of interrupts, Arm define an optional *generic interrupt controller* (GIC). The GIC provides a uniform API for sending and routing interrupts from peripherals to threads, and comes in several versions. We focus on GICv3 and its CPU interface, but expect the behaviour we describe should apply to GICv4.

There are many interesting questions about SGIs. We cover just a simple baseline: enough to reason about the synchronisation used by software, but ignoring much of the complexity of the GIC. We fix a relatively simple configuration, and focus on the relaxed-memory aspects of the interaction between SGIs and the rest of the memory and processor state.

7.1 The Generic Interrupt Controller – basic machinery

We begin by introducing the context of the basic Arm GIC machinery, before addressing its relaxed ordering in later subsections. An interrupt is *generated* on its *source* (a hardware thread or some peripheral) for a particular *event* (e.g. an SGI). This interrupt is then sent to the interrupt controller, which is split into a distributor, the global machinery in charge of routing interrupts to cores, and the per-thread redistributors, each of which maintains a thread-local state for each interrupt (which we describe in more detail later). Interrupts are identified in the GIC by its 'interrupt ID number' (INTID). Each instance of an interrupt sent to the interrupt controller is associated with an INTID, either by software or a peripheral, and is provided to the receiving core in a register it can read (via acknowledgement, described later).

Each hardware thread (PE) has an interrupt status register (the `ISR`), which has a single pending status bit for each interrupt class (IRQ, FIQ, SError, etc). For each fetch-decode-execute cycle of the top-level loop (see §2.3.1), the processor checks these status bits to determine whether an interrupt is pending; if an interrupt is pending and is not masked on that PE, the PE takes that interrupt. It is the interrupt controller's responsibility to set and clear the pending bit in that register, notifying the thread of a pending interrupt. To determine when to deliver (set the bit in the interrupt status register) interrupts to the core, the redistributor maintains three key pieces of state (this is for an 'edge-triggered' interrupt, such as for SGIs; we do not discuss 'level-sensitive' interrupts):

- A priority to assign to each interrupt source, and the current 'working' priority of the interrupt(s) being handled.
- A priority mask, which prevents interrupts with too low a priority from being delivered to the core.
- A per-INTID state, which is one of:
  - Inactive: there is no current interrupt;
  - Pending: the GIC has received an interrupt, and maybe delivered it, but the core has not begun handling it; or
  - Active: the core has signalled it is handling the interrupt, but not yet signalled it is done.

Lifecycle of an interrupt. Interrupts start out Inactive. When an interrupt is asserted by the source, the GIC sets the state for this interrupt's INTID to Pending. Within some unspecified, finite amount of time, the GIC will set the pending bit in the interrupt status register for the core, enabling the core to take an exception on the next fetch-decode-execute loop.

The core should then *acknowledge* the interrupt, by reading the appropriate interrupt-acknowledge-register (IAR); this returns the INTID for use by the core, and sends a request to the redistributor to mark the INTID as Active. Transitioning to the active state sets the working priority to the priority of that INTID's source, preventing lower-priority interrupts from pre-empting the core, and clears the pending bit in the interrupt-status-register on the core. If another interrupt with the same INTID is asserted while the interrupt is active, that instance will be buffered (only a single extra instance may be buffered) and taken later, and the INTID is said to be 'Active and Pending'. While the interrupt is active, it will not be re-delivered to the core, so even if the interrupt service routine performs an `ERET`, it will not re-take the exception.

At some later time, the core may finish handling the interrupt and be ready to receive further instances of that INTID. There are two ways to do this, depending on whether one wants to separate *priority drop* from *deactivation*, which is controlled by the `EOImode`. With `EOImode=0`, by writing the INTID to the end-of-interrupt register (EOIR), the interrupt is deactivated simultaneously with the the priority drop. With `EOImode=1`, writes to the EOIR only perform priority drop, requiring separate deactivation through a write to the deactivate-interrupt-register (DIR). Additionally, the GIC interface provides registers which can manually set the current priority, or mask, or explicitly set the state of an interrupt. Figure 11 shows the typical transitions between states.

Intended software usage. Typically, software use of interrupts falls into one of two categories:

- Nested interrupt servicing, where software readily uses priorities and handles the interrupt directly in the interrupt service routine, as it typical in real-time OSs.

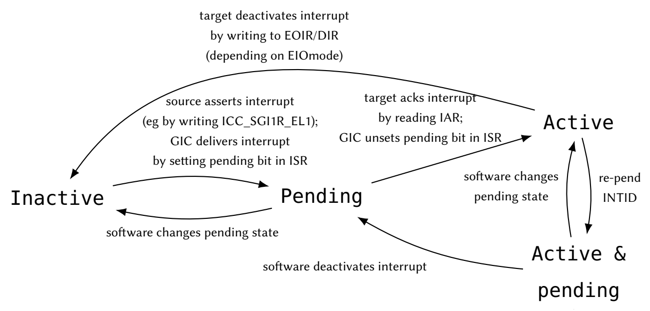


Figure 11: GIC automaton, for each PE and each INTID, based on Figure 4-3 "Interrupt handling state machine" from Arm [11, §4.1.2], specialised to edge-triggered behaviour.

- Deferred interrupt handling, where software acknowledges the interrupt directly, but handles it later.

Linux falls into the second category, utilising only a single interrupt priority. This 'split' approach to handling interrupts, where the interrupt service routine merely acknowledges, and the actual handling of the interrupt comes later, leads Linux to adopt `EOImode=1`. When the interrupt is taken by the core, it is acknowledged, the INTID is checked against special cases, priority is quickly dropped, and interrupts are unmasked. The actual interrupt may then be handled, concurrently with new interrupts being signalled to the core, although duplicates of the incident INTID will still be masked as it is not yet deactivated. Eventually, the core completes the work for that interrupt and then deactivates it, advancing the state machine.

7.2 Ordering of the propagation of SGIs

An SGI is generated by a write to the appropriate register (e.g. `ICC_SGI1R_EL1`), and is received on one or several thread(s). This gives rise to questions of three kinds:

- What is required to order the generation of the SGI after earlier accesses?
- Does routing of the SGI imply ordering? e.g. is the interrupt controller an observer wrt. multi-copy-atomicity?
- What is required to ensure that the sequence of acknowledgement and deactivation happens correctly?

There are few guarantees about the order of propagation of SGIs, or interrupts generally. Interrupts may be delivered to the core at any time, and multiple pending interrupts may be delivered in any order (priorities allowing). There are no guarantees analogous to the coherence or atomicity of memory, and generated interrupts may be re-ordered, or delivered to different cores in different orders. However, as discussed earlier, interrupts may not be speculated, and so the interrupt cannot be delivered to the target PE before it is generated.

SGI litmus testing. We extract the fundamental Message-Pass-via-SGI shape underlying Linux's implementation of RCU on Armv8 as a litmus test, `MPviaSGIEIOmode1sequence`, in Figure 12. Passing a message through an SGI requires some synchronisation between the write of the data and the generation of the SGI (here a `DSB ST` on Thread 0), and requires observation of the data in the exception

```
MPviaSGIEIOmode1sequence                                      AArch64
Initial state: *x=0;
0:PSTATE.EL=1;                                             1:EIOMode=1
0:X1=x;                                     1:X0=0, 1:X1=0, 1:X2=x
         Thread 0          | Thread 1 |        T1 Handler
MOV X0,#1                  | NOP      | MRS X3, IAR // ack interrupt
STR X0,[X1] // write data  |          | AND X3, X3, #0xFFFFFF
DSB ST                     |          | DSB SY
MOV X2, #1, LSL #40        |          | MSR EOIR, X3 // drop priority
//generate SGI             |          | ISB
MSR ICC_SGI1R_EL1, X2      |          | MOV X0, #1
ISB                        |          | LDR X1,[X2] // read data
                           |          | DSB SY
                           |          | MSR DIR, X3 // deactivate
                           |          | ERET
Forbidden: 1:X0=1, 1:X1=0
```

Figure 12: MPviaSGIEIOmode1sequence: Synchronisation-via-SGI with the full acknowledge-drop-deactivate sequence appropriate for `EOImode=1`.

```
MPviaSGI                                        AArch64
Initial state: *x=0;
0:PSTATE.EL = 1
0:X1=x;                       1:X0=0, 1:X1=0, 1:X2=x
        Thread 0         | Thread 1 | T1 Handler
MOV X0, #1               | NOP      | MOV X0, #1
STR X0, [X1]             |          | LDR X1, [X2]
MOV X2, #1, LSL #40      |          | ERET
MSR ICC_SGI1R_EL1, X2    |          |
Allowed: 1:X0=1, 1:X1=0
```

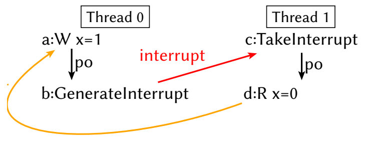


Figure 13: `MPviaSGI`: message passing via SGI, illustrating two potential phenomena: (1) On the writer side: a po-earlier write gets reordered with a po-later GenerateInterrupt. (2) On the reader side: a po-earlier TakeInterrupt gets reordered with a po-later read (from the interrupt handler).

handler; the SGI also needs to be is properly acknowledged and deactivated, with the appropriate barriers.

This test is composed of two interacting parts: the part that imposes the ordering between the write and the read of the data, and the part that interacts with the GIC to manage the interrupt. Figure 13 asks the most basic question of this shape: if we try pass a message via an SGI, without any further synchronisation, can we still read an old value? The answer is yes, because the generation and subsequent delivery of the SGI could happen before the propagation of the store. On the other hand, the extensive synchronisation on the receiving thread imposed by GIC management is accidental for the read, which is already strongly ordered after the taking of the exception.

7.3 Software usage of SGIs

Synchronisation mechanisms like those discussed above rely on this link between memory accesses and interrupts to achieve low-overhead synchronisation. More specifically, they push the cost away from normal memory accesses and onto a "system-wide memory barrier" implemented using interrupts. This is a fork-join barrier, not a fence. Interestingly, RCU and the Verona asymmetric lock rely on two different aspects of this system-wide memory barrier: RCU relies on masking of interrupts to implement cheap read critical sections, whereas the Verona asymmetric lock relies on precision of interrupts (§6).

System-wide memory barrier. This system-wide memory barrier is a two-way barrier: the issuing PE notifies all other PEs, and waits for a reply from all of them. The notification is implemented using interrupts, relying on the ordering described above, which is guaranteed by Arm-A. In Kernel RCU (where this barrier forms the core of `synchronize_rcu`, exposed to userland as the `sys_membarrier` syscall), the wait for a reply is implemented using memory operations, namely a lock-protected counter that threads increment to acknowledge receipt of the interrupt. We simplify this (to a write to a flag) in our litmus tests to reduce complexity.

RCU. The key concept of RCU is that of a grace period [51][50, §9], as captured by Alglave et al. [6] in the `RCU-MP` litmus test (Figure 14).

We focus on the use of interrupts in Kernel RCU. For performance, RCU also relies on address dependencies to implement cheap ordering in read sections, but that is already explained in the 'user' model of Arm-A [28, 58] by `MP+dmbst+addr`.

At the level of Arm assembly, the `synchronize_rcu` system-wide memory barrier is decomposed into a `DSB ST` followed by an `MSR` to `SGI1R`, and a wait for the acknowledgement (in our cut-down tests, a read acquire of the ack flag); entering the read critical section via `rcu_read_lock` and leaving it via `rcu_read_unlock` decompose to writes to the `DAIF` (pseudo)register that mask and unmask interrupts.

The crux of this litmus test is that interrupts are masked between the two reads, and that the handler is therefore either before both reads, or after both reads, but not in between (as in, no event of the handler is in between the two reads in program order). At the Linux C level, this masking ensures that the interrupt generated by the `synchronize_rcu` system-wide memory barrier is taken either before or after the read section, but not during, providing the basis for mutual exclusion. In the litmus tests, this is captured by the fact that if the read of the flag `y` sees the flag, the read of the data `x` sees the new data.

Verona asymmetric lock. We capture the key scenario of the asymmetric lock of Verona [54] (and of 'biased locking' and 'asymmetric Dekker synchronisation' [18, 22, 26, 27, 40, 41, 49, 59] as used in the JVM). It occurs when an 'internal acquire' from the (unique) owner thread contends with an 'external acquire' from another thread. The internal acquire is meant to be cheap, and only involves writing to an 'external' flag to express interest, and then, in program order, reading from an 'internal' flag to ensure that other threads have not expressed interest (falling onto the slow path if they have). Crucially, in C++, there is a `Barrier::compiler()` that prevents reordering of two instructions by the compiler, but does not appear in the generated assembly. The external acquire does the symmetric thing, writing on the 'internal' flag to express interest, and then

RCU-MP AArch64

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

`0:X1=x, 0:X4=y, 0:X6=z;`

`1:X0=0, 1:X1=y, 1:X2=0, 1:X3=x, 0:X5=z`

Thread 0	Thread 1	T1 Handler

`//*x = 1;`
`MOV X0, #1`
`STR X0, [X1]`
`//sync_rcu();`
`MOV X2, #1, LSL #40`
`MSR ICC_SGI1R_EL1, X2`
`LDAR X5, [X6]`
`//*y = 1;`
`MOV X3, #1`
`STR X3, [X4]`
	`//read_lock();`
	`MSR DAIFSet,`
	` #0xf`
	`//... = *y;`
	`LDR X0, [X1]`
	`//... = *x;`
	`LDR X2, [X3]`
	`//read_unlock();`
	`MSR DAIFClr, #0xf`
		`ACK-DEACT(X6)`
		`MOV X2, #1`
		`STLR X2,[X5]`
		`ERET`

Allowed: `0:X5=1, 1:X0=1, 1:X2=0`

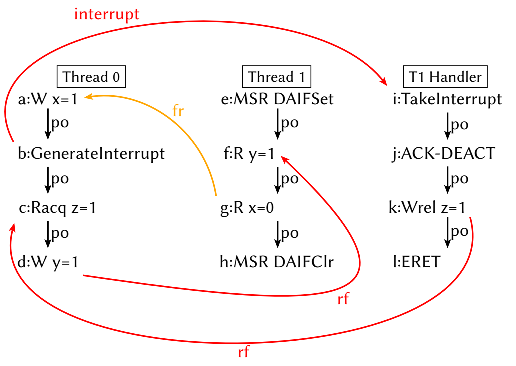


Figure 14: `RCU-MP`: the key test of RCU: are two writes separated by the generation of an SGI ordered with respect to a read critical section implemented via interrupts masking? With a `DSB ST` between `a` and `b`, this is forbidden.

reading from the 'external' flag to ensure that the owner has not expressed interest. To order this, it uses a `Barrier::memory()`, which involves a `FlushProcessWriteBuffers()`, which on Linux is implemented using a `sys_membarrier`, which essentially boils down to a `synchronize_rcu`.

The key guarantee that is relied in the 'cheap' thread is that the interrupt must be taken precisely, and that it is therefore taken, in program order, either entirely before the read of the internal flag, entirely between the read of the internal flag and the write to the external flag, or entirely after the write to the external flag. In all three cases, the system-wide memory barrier ensures that at least one of the two threads must see that the other thread has expressed interest (must read the recent write), and therefore backs off, ensuring mutual exclusion.

7.4 Ordering of GIC register writes

The Arm GIC Architecture Specification text (IHI 0069H.b) is reasonably clear about the relaxed ordering of GIC events induced by accesses to GIC registers with program-order later events (12.1.6 "Observability of the effects of accesses to the GIC registers"), though there are still subtle requirements for barriers. A `DSB.SY` enforces ordering of GIC events (generate, acknowledge, drop priority, and deactivate) induced by accesses to GIC registers (SGI1R, IAR, EOIR, DIR) with program-order-later events, as they are such effects. DSBs are not needed to merely order the register accesses themselves.

An ISB ensures that any pending interrupts are taken before executing the program-order later instructions.

If there was an interrupt in the Active and Pending state at deactivation, then it is immediately re-pended on the PE (and so delivery can immediately happen again). But, if there is no DSB between the write of the deactivation and the context synchronisation, it might be that the assertion and delivery did not yet occur, causing the interrupt to be taken later.

7.5 A draft axiomatic extension

We give a draft extension to the previous axiomatic model to support inter-processor interrupts, noting the challenges.

GIC candidates. Unlike with most of the instruction semantics, there is very little public ASL from Arm which describes the priority and INTID state machine system. While much of the GIC's machinery, routing, virtualisation and so on, is not required to discuss the usage of interrupts here, a large quantity of the base GIC architecture would need to be turned into ASL and incorporated into the machinery. The rest of this extension assumes one has such machinery in place.

First, we must extend the thread semantics: reads and writes of the registers of the CPU interface to the GIC, and interrupt status register, must be treated differently than other registers, lifting them to the memory model with a relation constraining the values they could read, analogous to 'reads-from'. This allows us to tie the thread's events interacting with the GIC, with those events coming from the GIC ASL.

We add the following new events, grouped as `GICEvents`:

- `GenerateInterrupt`, for the GIC action from writing the `SGI1R` register, which sends an IPI to other cores. It is associated with a *target* set of CPUs.
- `Acknowledge`, for the relevant effect in the GIC, i.e. the state machine change and related updates to registers. Here, we assume the GIC update is atomic, which ought to be true for simple physical SGIs.
- `DropPriority` and `Deactivate`, for the relevant effects on the GIC state machine and priority masking.

These new events are placed `iio`-*after* (intra-instruction-ordered) the respective register events. Such events could instead be inserted into `po`, with suitable modification of the previous relations, although for simplicity here we do not.

Interrupt witness. We add a new existentially-quantified relation to the witness: `interrupt`. This associates the `TakeInterrupt` with the `GenerateInterrupt` which caused it, constraining any program-order-later `Acknowledge` and corresponding `MRS` event INTID values. This effectively assigns the INTID at the point the interrupt is taken, and makes `interrupt` behave like `rf` for INTIDs; if the INTID is never read, one must consider all possible interrupt sources.

Update to relations and axioms. The update to the relations is then fairly straightforward: insert `interrupt` into `ob`, and make `DSB` instructions order GIC events in program-order. We do not put `GICEvents` in program order to express that they may execute out-of-order with respect to other events in the same thread, including context-synchronisation, unless explicitly ordered (e.g. by DSBs).

8 Conclusion

We identify an open problem in giving a definition of precision on relaxed architectures, and describe the challenge in doing so. We characterise some basic guarantees of precision, which should make it possible to apply some of the abstraction techniques used to reason about nesting of interrupts [45, 47].

We extend the Arm-A memory model to cover exceptions, an important aspect of defining the architectural interface, clarifying the behaviour at that interface, and giving an executable-as-a-test-oracle implementation of an axiomatic model usable as an exploration tool to investigate the effect of synchronisation on hardware exceptions and interrupts. We describe the interaction of hardware exceptions with memory errors, and the consequences on the user model.

We begin building a model for software-generated interrupts and the required parts of the interrupt machinery relied upon by the common computing base, giving the key shapes and litmus tests, some baseline behaviours of the Arm GIC, and a draft extension that covers key use cases.

Although there is much work still to do on exceptions, interrupts, and their interaction with other features, this work creates a robust foundation that future work can build on.

Acknowledgments

We thank Richard Grisenthwaite (Arm EVP, Chief Architect, and Fellow), Martin Weidmann (Director of Product Management, Arm Architecture and Technology Group), and Will Deacon (Google) for detailed discussions about the Arm architecture. We thank Ben Laurie and Sarah de Haas (Google) for their support. We thank Jonathan Woodruff and others at the CL for their insightful discussions.

This work was funded in part by Google. This work was funded in part by Arm. This work was funded in part by an AUFF starter grant (Pichon-Pharabod). This work was funded in part by two Amazon Research Awards (Pichon-Pharabod; Sewell and Simner). This work was funded in part by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee for ERC-AdG-2022, EP/Y035976/1 SAFER. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 789108, ERC-AdG-2017 ELVER). This work is supported by ERC-2024-POC grant ELVER-CHECK, 101189371. Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694. The authors would like to thank the Isaac Newton Institute for Mathematical Sciences, Cambridge, for support and hospitality during the programme Big Specification, where work on this paper was undertaken. This work was supported by EPSRC grant EP/Z000580/1. This work was funded in part by a Royal Society University Research Fellowship. One of the authors has received funding from the UK Advanced Research and Innovation Agency (ARIA) as part of the project Qbs4Safety: Core Representation Underlying Safeguarded AI.

References

[1] A. Adir, H. Attiya, and G. Shurek. 2003. Information-Flow Models for Shared Memory with an Application to the PowerPC Architecture. *IEEE Trans. Parallel Distrib. Syst.* 14, 5 (2003), 502–515. doi:10.1109/TPDS.2003.1199067

[2] Jade Alglave. 2010. *A Shared Memory Poetics.* Ph. D. Dissertation. Université Paris 7 – Denis Diderot.

[3] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 8:1–8:54. doi:10.1145/3458926

[4] Jade Alglave, Richard Grisenthwaite, Artem Khyzha, Luc Maranget, and Nikos Nikoleris. 2024. Puss In Boots: on formalising Arm's Virtual Memory System Architecture (extended version). (May 2024). https://inria.hal.science/hal-04567296 working paper or preprint.

[5] Jade Alglave and Luc Maranget. [n. d.]. The herdtools7 tool suite. diy.inria.fr, https://github.com/herd/herdtools7/. Accessed 2023-08-30.

[6] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar (Eds.). ACM, 405–418. doi:10.1145/3173162.3177156

[7] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 258–272. doi:10.1007/978-3-642-14295-6_25

[8] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. 2011. Litmus: Running Tests Against Hardware. In *Proc. TACAS.* doi:10.1007/978-3-642-19835-9_5

[9] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. doi:10.1145/2627752

[10] Arm. 2024. Arm Architecture Reference Manual: for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest. Accessed 2024-05-11. Issue K.a. 14777 pages..

[11] Arm. 2024. *Arm Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4.* Technical Report. Arm. IHI 0069H.b (ID041224).

[12] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages.* doi:10.1145/3290384 Proc. ACM Program. Lang. 3, POPL, Article 71.

[13] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. 2021. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *Proc. 33rd International Conference on Computer-Aided Verification (Lecture Notes in Computer Science, Vol. 12759).* Springer, 303–316. doi:10.1007/978-3-030-81685-8_14

[14] Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference Capabilities for Flexible Memory Management. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1363–1393. doi:10.1145/3622846

[15] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 283–307. doi:10.1007/978-3-662-46669-8_12

[16] Mark John Batty. 2015. *The C11 and C++11 concurrency model.* Ph. D. Dissertation. University of Cambridge, UK. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.708458

[17] Thomas Bauereiss, Brian Campbell, Alasdair Armstrong, Alastair Reid, Kathryn E. Gray, Anthony Fox, Peter Sewell, and Arm Limited. 2024. Sail Armv9.4-A instruction-set architecture (ISA) model. https://github.com/rems-project/sail-arm. Accessed 2024-05-11..

[18] Mike Burrows. 2004. *How to Implement Unnecessary Mutexes.* Springer New York, New York, NY, 51–57. doi:10.1007/0-387-21821-1_7

[19] Soham Chakraborty. 2019. *Correct Compilation of Relaxed Memory Concurrency.* Ph. D. Dissertation. Kaiserslautern University of Technology, Germany. https://kluedo.ub.rptu.de/frontdoor/index/index/docId/5697

[20] Luke Cheeseman, Matthew J. Parkinson, Sylvan Clebsch, Marios Kogias, Sophia Drossopoulou, David Chisnall, Tobias Wrigstad, and Paul Liétar. 2023. When Concurrency Matters: Behaviour-Oriented Concurrency. *Proc. ACM Program. Lang.* 7, OOPSLA2 (October 2023). https://www.microsoft.com/en-us/research/publication/when-concurrency-matters-behaviour-oriented-concurrency/

[21] William W. Collier. 1992. *Reasoning about parallel architectures.* Prentice Hall.

[22] Mingyao Yang Dave Dice, Hui Huang. 2001. Asymmetric Dekker Synchronization. http://web.archive.org/web/20070214114205/http://blogs.sun.com/dave/resource/Asymmetric-Dekker-Synchronization.txt

[23] Hernán Ponce de León and Johannes Kinder. 2021. Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks. *CoRR* abs/2108.13818 (2021). arXiv:2108.13818 https://arxiv.org/abs/2108.13818

[24] Hernán Ponce de León and Johannes Kinder. 2022. Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022.* IEEE, 235–248. doi:10.1109/SP46214.2022.9833774

[25] Will Deacon, Jade Alglave, Nikos Nikoleris, and Artem Khyzha. 2023. The ARMv8 Application Level Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat (accessed 2019-07-01). Accessed 2024-11-19.

[26] Dave Dice. 2006. Biased Locking in Hotspot. Oracle Blog, Wayback Machine. http://web.archive.org/web/20150320095550/https://blogs.oracle.com/dave/entry/biased_locking_in_hotspot

[27] David Dice, Mark S. Moir, and William N. Scherer III. 2010. United States Patent US 7814488B1 Quickly Reacquirable Locks. United Statess Patent Office.

[28] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (St. Petersburg, FL, USA).* 608–621. doi:10.1145/2837614.2837615

[29] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 429–442. doi:10.1145/3009837.3009839

[30] Kourosh Gharachorloo. 1995. *Memory Consistency Models for Shared-Memory Multiprocessors.* Ph. D. Dissertation. Stanford University.

[31] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, June 1990*, Jean-Loup Baer, Larry Snyder, and James R. Goodman (Eds.). ACM, 15–26. doi:10.1145/325164.325102

[32] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. 2015. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture (Waikiki).* 635–646. doi:10.1145/2830772.2830775

[33] Siddharth Gupta, Yuanlong Li, Qingxuan Kang, Abhishek Bhattacharjee, Babak Falsafi, Yunho Oh, and Mathias Payer. 2023. Imprecise Store Exceptions. In *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*, Yan Solihin and Mark A. Heinrich (Eds.). ACM, 52:1–52:15. doi:10.1145/3579371.3589087

[34] John L. Hennessy and David A. Patterson. 2012. *Computer Architecture: A Quantitative Approach* (5 ed.). Morgan Kaufmann, Amsterdam.

[35] Naorin Hossain, Caroline Trippel, and Margaret Martonosi. 2020. TransForm: Formally Specifying Transistency Models and Synthesizing Enhanced Litmus Tests. *CoRR* abs/2008.03578 (2020). arXiv:2008.03578 https://arxiv.org/abs/2008.03578

[36] Luc Maranget. 2024. Personal communication.

[37] Intel. 2002. A Formal Specification of Intel Itanium Processor Family Memory Ordering. developer.intel.com/design/itanium/downloads/251429.htm.

[38] Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 194:1–194:30. doi:10.1145/3428262

[39] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17).* Association for Computing Machinery, New York, NY, USA, 175–189. doi:10.1145/3009837.3009850

[40] Kiyokuni Kawachiya. 2005. *Java Locks: Analysis and Acceleration.* Ph. D. Dissertation. Keio University.

[41] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. 2002. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Seattle, Washington, USA) *(OOPSLA '02).* Association for Computing Machinery, New York, NY, USA, 130–141. doi:10.1145/582419.582433

[42] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (dec 2017), 32 pages. doi:10.1145/3158105

[43] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019).* Association for Computing Machinery, New York, NY, USA,

96–110. doi:10.1145/3314221.3314609

[44] Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A Model Checker for Weak Memory Models. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 427–440. doi:10.1007/978-3-030-81685-8_20

[45] Daniel Kroening, Lihao Liang, Tom Melham, Peter Schrammel, and Michael Tautschnig. 2015. Effective Verification of Low-Level Software with Nested Interrupts. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, Wolfgang Nebel and David Atienza (Eds.). EDA Consortium, 229–234. http://www.cs.ox.ac.uk/tom.melham/pub/Kroening-2015-EVL.pdf

[46] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 618–632. doi:10.1145/3062341.3062352

[47] Lihao Liang, Tom Melham, Daniel Kroening, Peter Schrammel, and Michael Tautschnig. 2017. Effective Verification for Low-Level Software with Competing Interrupts. *ACM Transactions on Embedded Computing Systems* 17, 2 (December 2017), 36:1–36:26. doi:10.1145/3147432

[48] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. 2016. COATCheck: Verifying Memory Ordering at the Hardware-OS Interface. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, Tom Conte and Yuanyuan Zhou (Eds.). ACM, 233–247. doi:10.1145/2872362.2872399

[49] Patricio Chilano Mateo. 2021. JEP 374: Deprecate and Disable Biased Locking. JDK Enhancement Proposal. https://openjdk.org/jeps/374

[50] Paul E. McKenney. 2023. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html

[51] Paul E. McKenney. 2024. RCU Concepts. https://www.kernel.org/doc/Documentation/RCU/rcu.txt Accessed 2024-11-19.

[52] Paul E McKenney and John D Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, Vol. 509518. 509–518.

[53] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. 2022. Axiomatic hardware-software contracts for security. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 72–86. doi:10.1145/3470496.3527412

[54] Matthew J. Parkinson. 2024. Some things I wish I hadn't seen. presented at The Future of Weak Memory 2024.

[55] Jean Pichon-Pharabod and Peter Sewell. 2016. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 622–633. doi:10.1145/2837614.2837616

[56] William W. Pugh. 1999. Fixing the Java Memory Model. In *Proceedings of the ACM 1999 Conference on Java Grande, JAVA '99, San Francisco, CA, USA, June 12-14, 1999*, Geoffrey C. Fox, Klaus E. Schauser, and Marc Snir (Eds.). ACM, 89–98. doi:10.1145/304065.304106

[57] Christopher Pulte. 2018. *The Semantics of Multicopy Atomic ARMv8 and RISC-V.* Ph. D. Dissertation. University of Cambridge. https://www.repository.cam.ac.uk/handle/1810/292229.

[58] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29. doi:10.1145/3158107

[59] Kenneth Russell and David Detlefs. 2006. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 263–272. doi:10.1145/1167473.1167496

[60] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 311–322. doi:10.1145/2254064.2254102

[61] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 175–186. doi:10.1145/1993498.1993520

[62] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. 2009. The semantics of x86-CC multiprocessor machine code. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 379–391. doi:10.1145/1480881.1480929

[63] Peter Sewell, Christopher Pulte, Shaked Flur, Mark Batty, Luc Maranget, and Alasdair Armstrong. 2022. Multicore Semantics: Making Sense of Relaxed Memory (MPhil slides). https://www.cl.cam.ac.uk/~pes20/slides-acs-2022.pdf

[64] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. doi:10.1145/1785414.1785443

[65] Ben Simner, Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Ohad Kammar, Jean Pichon-Pharabod, and Peter Sewell. 2024. Relaxed exception semantics for Arm-A (extended version). *CoRR* abs/2412.15140 (2024). doi:10.48550/ARXIV.2412.15140 arXiv:2412.15140

[66] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. 2022. Relaxed virtual memory in Armv8-A. In *Proceedings of the 31st European Symposium on Programming (Lecture Notes in Computer Science, Vol. 13240)*. Springer, 143–173. doi:10.1007/978-3-030-99336-8_6

[67] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. 2020. ARMv8-A System Semantics: Instruction Fetch in Relaxed Architectures. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 626–655. doi:10.1007/978-3-030-44914-8_23

[68] P. S. Sindhu, J.-M. Frailong, and M. Cekleov. 1991. Formal Specification of Memory Models. In *Scalable Shared Memory Multiprocessors.* Kluwer, 25–42. doi:10.1007/978-1-4615-3604-8_2