ARMv8-A system semantics: instruction fetch in relaxed architectures(extended version)*

Ben Simner[1], Shaked Flur[1*], Christopher Pulte[1*], Alasdair Armstrong[1],
Jean Pichon-Pharabod[1], Luc Maranget[2], and Peter Sewell[1]

[1] University of Cambridge, UK
[2] INRIA Paris, France
[*] These authors contributed equally

Abstract. Computing relies on *architecture specifications* to decouple hardware and software development. Historically these have been prose documents, with all the problems that entails, but research over the last ten years has developed rigorous and executable-as-test-oracle specifications of mainstream architecture instruction sets and "user-mode" concurrency, clarifying architectures and bringing them into the scope of programming-language semantics and verification. However, the *system semantics*, of instruction-fetch and cache maintenance, exceptions and interrupts, and address translation, remains obscure, leaving us without a solid foundation for verification of security-critical systems software.

In this paper we establish a robust model for one aspect of system semantics: instruction fetch and cache maintenance for ARMv8-A. Systems code relies on executing instructions that were written by data writes, e.g. in program loading, dynamic linking, JIT compilation, debugging, and OS configuration, but hardware implementations are often highly optimised, e.g. with instruction caches, linefill buffers, out-of-order fetching, branch prediction, and instruction prefetching, which can affect programmer-observable behaviour. It is essential, both for programming and verification, to abstract from such microarchitectural details as much as possible, but no more. We explore the key architecture design questions with a series of examples, discussed in detail with senior Arm staff; capture the architectural intent in operational and axiomatic semantic models, extending previous work on "user-mode" concurrency; make these models executable as test oracles for small examples; and experimentally validate them against hardware behaviour (finding a bug in one hardware device).

[*] This is an extended version of a paper in ESOP 2020, with extra details in the appendix.

We thereby bring these subtle issues into the mathematical domain, clarifying the architecture and enabling future work on system software verification.

1 Introduction

Computing relies on the *architectural abstraction*: the specification of an envelope of allowed hardware behaviour that hardware implementations should lie within, and that software should assume. These interfaces, defined by hardware vendors and relatively stable over time, notionally decouple hardware and software development; they are also, in principle, the foundation for software verification. In practice, however, industrial architectures have accumulated great complexity and subtlety: the ARMv8-A and Intel architecture reference manuals are now 7476 and 4922 pages [9,26], and hardware optimisations, including out-of-order and speculative execution, result in surprising and poorly-understood programmer-observable behaviour. Architecture specifications have historically also been entirely informal, describing these complex envelopes of allowed behaviour solely in prose and pseudocode. This is problematic in many ways: do not serve as clear documentation, with the inevitable ambiguity and incompleteness of informal prose leaving major questions unanswered; without a specification that is executable as a test oracle (that can decide whether some observed behaviour is allowed or not), hardware validation relies on test suites that must be manually curated; without an architecturally-complete emulator (that can exhibit all allowed behaviour), it is very hard for software developers to "program to the specification" – they rely on test-and-debug development, and can only test above the hardware implementation(s) they have; and without a mathematically rigorous semantics, formal verification of hardware or software is impossible.

Over the last 10 years, much has been done to put architecture specifications on a more rigorous footing, so that a single specification can serve all those purposes. There are three main problems, two of which are now largely solved.

The first is the instruction-set architecture (ISA): the specification of the sequential behaviour of individual instructions. This is chiefly a problem of scale: modern industrial architectures such as Arm or x86 have large instruction sets, and each instruction involves many details, including its behaviour at different privilege levels, virtual-to-physical address translation, and so on – a single Arm instruction might involve hundreds of auxiliary functions. Recent work by Reid et al. within Arm [40,41,42] transitioned their internal ISA description into a mechanised form, used both for documentation and testing, and with him we automatically translated this into publicly available Sail definitions and thence into theorem-prover definitions [11,10]. Other related work is in §7.

The second is the relaxed-memory concurrent behaviour of "user-mode" operations: memory writes and reads, and the mechanisms that architectures provide to enforce ordering and atomicity (dependencies, memory barriers, load-linked/store-conditional operations, etc.). In 2008, for ARMv7, IBM POWER, and x86, this was poorly understood, and the architects regarded even their own prose specifications as inscrutable. Now, following extensive work by many people [36,37,19,18,22,8,31,45,7,46,48,35,6,2,47,13,1], ARMv8-A has a well-defined and simplified model as part of its specification [9, B2.3], including a prose transcription of a mathematical model [15], and an equivalence proof between operational and axiomatic presentations [36,37]; RISC-V has adopted a similar model [52]; and IBM POWER and x86 have well-established de-facto-standard models. All of these are experimentally validated against hardware, and supported by tools for exhaustively running tests [17,4]. The combination of these models and the ISA semantics above is enough to let one reason about or model-check concurrent algorithms.

That leaves the third part of the problem: the "system" semantics, of instruction-fetch and cache maintenance, exceptions and interrupts, and address translation and TLB (translation lookaside buffer) maintenance. Just as for "user-mode" relaxed memory, these are all areas where microarchitectural optimisations can have surprising programmer-visible effects, especially in the concurrent context. The mechanisms are relied on by all code, but they are explicitly managed only by systems code, in just-in-time (JIT) compilers, dynamic loaders, operating-system (OS) kernels, and hypervisors. This is, of course, exactly the security-critical computing base, currently trusted but not trustworthy, that is especially in need of verification – which requires a precise and well-validated definition of the architectural abstraction. Previous work has scarcely touched on this: none of seL4 [27], CertiKOS [24,23], Komodo [16], or [25,12], address realistic architecture concurrency, and they use (at best) idealised models of the sequential systems architecture. The CakeML [51,28] and CompCert [29] verified compilers target only sequential user-mode ISA fragments.

In this paper we focus on one aspect of system semantics: instruction fetch and cache maintenance, for ARMv8-A. The ability to execute code that has previously been written to data memory is fundamental to computing: fine-grained self-modifying code is now rare, and (rightly) deprecated, but program loading, dynamic linking, JIT compilation, debugging, and OS configuration all rely on executing code from data writes. However, because these are relatively infrequent operations, hardware designers have been able to optimise by partially separating the instruction and data paths, e.g. with distinct instruction caching, which by default may not be coherent with data accesses. This can introduce programmer-visible behaviour analogous to that of user-mode relaxed-memory concurrency, and require specific additional synchronisation to correctly pick up code modifications. Exactly what these are is not entirely clear in the current ARMv8-A architecture text, just as pre-2018 user-mode concurrency was not.

Our main contribution is to clarify this situation, developing precise abstractions that bring the instruction-fetch part of ARMv8-A system behaviour into the domain of rigorous semantics. Arm have stated [private communication] that they intend to incorporate a version of this into their architecture. We aim thereby to enable future work on system software verification using the techniques of programming languages research: program analysis, model-checking, program logics, etc. We begin (§2) by recalling the informal architectural guarantees that Arm provide, and the ways in which real-world software systems such as Linux, JavaScript, and WebAssembly change instruction memory. Then:

(1) **We explore the fundamental phenomena and architecture design questions with a series of examples** (§3). We explore the interactions between instruction fetching, cache maintenance and the 'usual' relaxed memory stores and loads, showing that instruction fetches are more relaxed, and how even fundamental coherence guarantees for data memory do not apply to instruction fetches. Most of these questions arose during the development of our models, in detailed ongoing discussion with the Arm Chief Architect and other Arm staff. They include questions of several different kinds. Six are clear from the Arm prose specification. Of the others: two are not implied by the prose but are natural choices; five involved substantive new choices by Arm that had not previously been considered and/or documented; for two, either choice could be reasonable, and Arm chose the simpler (and weaker) option; and for one, Arm were independently already strengthening the architecture to accommodate existing software.

(2) **We give an operational semantics for Arm instruction fetch and icache maintenance** (§4). This is in an abstract-microarchitectural style that supports an operational intuition for how hardware actually works, while abstracting from the mass of detail and the microarchitectural variation of actual hardware implementations. We do so by extending the Flat model [37] with simple abstractions of instruction caches and the coherent data cache network, in a way that captures the architectural intent, defining the entire envelope of behaviours that implementations should be allowed to exhibit.

(3) **We give a more concise presentation of the model in an axiomatic style** (§5), extending the "user-mode" axiomatic model from previous work [37,36,15,9], and intended to be functionally equivalent. We discuss how this too matches the architectural intent.

(4) **We validate all this** in two ways: by the extensive discussion with Arm staff mentioned above, and by experimental testing of hardware behaviour, on a selection of ARMv8-A cores designed by multiple vendors (§6).

We run tests on hardware with a mild extension of the Litmus tool [5,7]. We make the operational model executable as a test oracle by integrating it into the RMEM tool and its web interface [17], introducing optimisations that make it possible to exhaustively execute the examples. We make the axiomatic model executable as a test oracle with a new tool that takes litmus tests and uses a Sail [11] definition of a fragment of the ARMv8-A ISA to generate SMT problems for the model. We then compare hardware and the two models for the handwritten tests (modulo two tests not supported by the axiomatic checker), compare hardware and the operational model on a suite of 1456 tests, automatically generated with an extension of the diy tool [3], and check the operational and axiomatic models against sets of previous non-ifetch tests. In all this data our models are equivalent to each other and consistent with hardware observations, except for one case where our testing uncovered a hardware bug on a Qualcomm device.

Finally, we discuss other related work (§7) and conclude (§8). We do all this for ARMv8-A, but other relaxed architectures, e.g. IBM POWER and RISC-V, face similar issues; our tests and tooling should enable corresponding work there.

The models are too large to include or explain in full here, so we focus on explaining the motivating examples, the main intuition and style of the operational model, in a prose rendering of its executable mathematics, and the definition of the axiomatic model. Appendices provide additional examples, a complete prose description of the operational model, and additional explanation of the axiomatic model. The complete executable mathematics version, the web-interface tool for running it, and our test results are at `https://www.cl.cam.ac.uk/~pes20/iflat/`.

Caveats and Limitations Our executable models are integrated with a substantial fragment of the Sail ARMv8-A ISA (similar to that used for CakeML), but not yet with the full ISA model [11,40,41,42]; this is just a matter of additional engineering. We only handle the 64-bit AArch64 part of ARMv8-A, not AArch32. We do not handle the interaction between instruction fetch and mixed-size accesses, or other variants of the cache maintenance instructions, e.g. those used for interaction with DMA engines, and variants by set or way instead of by virtual address. Finally, the equivalence between our operational and axiomatic models is validated experimentally. A proof of this equivalence is essential in the long term, but would be a major work in itself: the complexity makes mechanisation essential, but the operational model (in all its scale and complexity) has not yet been subject to mechanised proof. Without instruction fetch, a non-mechanised proof was the main result of an entire PhD thesis [36], and we expect the addition of instruction fetch to require global changes to the argument.

2 Industry Practice and the Existing ARMv8-A Prose

Computer architecture relies on a host of sophisticated techniques, including buffering, caching, prediction, and pipelining, for performance. For the normal memory reads and writes of “user-mode” concurrency, the programmer-visible relaxed-memory effects largely arise from store buffering and from out-of-order and speculative pipeline behaviour, not from the cache hierarchy (though some IBM POWER phenomena do arise from the interconnect, and from late processing of cache invalidates). All major architectures provide a strong per-location guarantee of *coherence*: for each memory location, different threads cannot observe the writes to that location in different orders. This is implemented in hardware by coherent cache protocols, ensuring (roughly) that each cache line is writable by at most one hardware thread at a time, and by additional machinery restricting store buffer and pipeline behaviour. Then each architecture provides additional synchronisation mechanisms to let the programmer enforce ordering properties involving multiple locations.

At first sight, one might expect instruction fetches to act like other memory reads but, because writes to instruction memory are relatively rare, hardware designers have adopted different caching mechanisms. The Arm architecture carefully does not mandate exactly what these must be, to allow a wide range of possible hardware implementations, but, for example, a high-performance Arm processor might have per-core separate L1 instruction and data caches, above a unified per-core L2 cache and an L3 cache shared between cores. There may also be additional structures, e.g. per-core fetch queues, and caching of decoded micro-operations. This instruction caching is not necessarily coherent with data memory accesses: *"the architecture does not require the hardware to ensure coherency between instruction caches and memory"* [9, B2.4.4 (B2-114)]; instead, programmers must use explicit cache maintenance instructions. The documentation gives a particular sequence of these: *"If software requires coherency between instruction execution and memory, it must manage this coherency using Context synchronization events and cache maintenance instructions. The following code sequence can be used to allow a processing element (PE) to execute code that the same PE has written."*

```
; Coherency example for data and instruction accesses [...]
; Enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
STR Wt, [Xn]; Store new instruction
DC CVAU, Xn ; Clean data cache by virtual address (VA) to PoU
DSB ISH     ; Ensure visibility of the data cleaned from cache
IC IVAU, Xn ; Invalidate instruction cache by VA to PoU
DSB ISH     ; Ensure completion of the invalidations
ISB         ; Synchronize the fetched instruction stream
```

At first sight, this may be entirely mysterious. The remainder of the paper establishes precise semantics for each instruction, explaining why each is required, but as a rough intuition:

1. The `DC CVAU,Xn` cleans this core's data cache for address `Xn`, pushing the new write far enough down the hierarchy for an instruction fetch that misses in the instruction cache to be guaranteed to see the new value. This point is the *Point of Unification* (PoU) and is usually the point where the instruction and data caches become unified (L2 for most modern devices).
2. The `DSB ISH` waits for the clean to have happened before letting the later instructions execute (without this, the sequence itself can execute out-of-order, and the clean might not have pushed the write down far enough

before the instruction cache is updated). The ISH makes this specific to the *Inner Shareable Domain*: the processor itself, not the system-on-chip. We do not model shareability domains in this paper, so this is equivalent to a DSB SY.
3. The IC IVAU,Xn invalidates any entry for that address in the instruction caches for all cores, forcing any future fetch to miss in the instruction cache, and instead read the new value from the data memory hierarchy; it also touches some fetch queue machinery.
4. The second DSB ISH ensures the invalidation completes.
5. The final ISB flushes this core's pipeline, forcing a re-fetch of all program-order-later instructions.

Some hardware implementations provide extra guarantees, rendering the DC or IC instructions unnecessary. Arm allow software to discover this in an architectural way, by reading the CTR_EL0 register's DIC and IDC bits. Our modelling handles this, but for brevity we only discuss the weakest case, with CTR_EL0.DIC=CTR_EL0.IDC=0, that requires full cache maintenance.

Arm make clear that instructions can be prefetched (perhaps speculatively): *"How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory, the PE might have fetched the instructions from memory at any time since the last Context synchronization event on that PE."*

Concurrent modification and instruction fetch require the same sequence, with an ISB on each thread that executes the new instructions, and the rest of the sequence on the modifying thread [9, B2.2.5 (B2-94)]. Concurrent modification without synchronisation is restricted to particular instructions (B (branch), BL (branch-and-link), BRK (break), SMC, HVC, SVC (secure monitor, hypervisor, and supervisor calls), ISB, and NOP), otherwise there could be *constrained unpredictable behaviour*: *"any behavior that can be achieved by executing any sequence of instructions that can be executed from the same*

Exception level". Concurrent modification of conditional branches is allowed but can result in the old condition with the new target address or vice versa.

All this gives some guidance for programmers, but it leaves the exact semantics of instruction fetch and those cache maintenance instructions unclear, and in practice software typically does not use the above sequence verbatim. For example, it may synchronise a range of addresses at once, looping the `DC` and `IC` parts, or the final `ISB` may be subsumed by instruction synchronisation from exception entry or return. Linux has many places where it modifies code at runtime: in boot-time patching of *alternatives*, modifying kernel code to specialise it to the particular hardware being run on; when the kernel loads code (e.g. when the user calls `dl_open`); and in the `ptrace` system call, used e.g. by the GDB debugger to patch arbitrary instructions with breakpoints at runtime. In Google's *Chrome* web browser, its WebAssembly and JavaScript just-in-time (JIT) compilers are required to both write new code during execution and modify existing code at runtime. In JavaScript, this modification happens inside a single thread and so is quite straightforward. The WebAssembly case is more complex, as one thread is modifying the code of another. A software thread can also be moved (by the OS or hypervisor) from one hardware thread to another, perhaps while it is in the middle of some instruction cache maintenance. Moreover, for security reasoning, we have to be able to bound the possible behaviour of arbitrary code.

All this means that we cannot treat the above sequence as a whole, as an opaque black box. Instead, we need a precise semantics for each individual instruction, but the existing prose documentation does not provide that.

The problem we face is to give such a semantics, that correctly defines behaviour in arbitrary concurrent contexts, that captures the Arm architectural intent, that is strong enough for software, and that abstracts from the variety of hardware implementations (e.g. with differing cache structures) that the architecture intends to allow – but which programmers should not have to think about.

3 Instruction Fetch Phenomena and Examples

We now describe the main instruction-fetch phenomena and architecture design questions for ARMv8-A, illustrated by handwritten litmus tests, to guide the following model design.

3.1 Instruction-Fetch Atomicity

The first point, as mentioned in §2, is that concurrent modification and fetch is only permitted if the original and modified instructions are in a particular set: various branches, supervisor/hypervisor/secure-monitor calls, the `ISB` instruction synchronisation barrier, and `NOP`. Otherwise, the architecture permits *constrained unpredictable* behaviour, meaning that the resulting machine state could be anything that would be reachable by arbitrary instructions at the same exception level. The following W+F test illustrates this.

W+F AArch64

Initial state: `0:W0="SUB X0,X0,#1", 0:X1=l`	
Thread 0	**Thread 1**
`STR W0,[X1] // modify Thread 1 at l`	`l: ADD X0,X0,#1 // initial code`
Allowed: constrained-unpredictable final state	

In this test Thread 0 performs a memory store (with the `STR` instruction) to the code that Thread 1 is executing; overwriting the `ADD X0,X0,#1` instruction with the 32-bit encoding of the `SUB X0,X0,#1` instruction. If the fetch were atomic, the outcome of this test would be the result of executing either the `ADD` or the `SUB` instruction, but, since at least one of those is not in the set of the 8 atomically-fetchable instructions given previously, Thread 1 has constrained-unpredictable behaviour and the final state is very loosely constrained. Note, however, that this is nonetheless much stronger than the C/C++ whole-program undefined behaviour in the presence of a data race: unlike C/C++, a hardware architecture has to define a useful envelope of behaviour for arbitrary code, to provide guarantees for the rest of the system when one user thread has a race.

Conditional Branches For conditional branches, the Arm architecture provides a specific non-single-copy-atomic fetch guarantee: the execution will be consistent with either the old or new target, and either the old or new condition. For example, this W+F+branches test can overwrite a `B.EQ g` with a `B.NE h`, and end up executing `B.NE g` or `B.EQ h` instead of one of those. Our future examples will only modify `NOP`s and unconditional branch instructions.

W+F+branches AArch64

Initial state: `0:W0="B.NE h", 0:X1=l`	
Thread 0	**Thread 1**
`STR W0,[X1]`	`l: B.EQ g`
Allowed: execute `"B.NE g"`	

3.2 Coherence

Data writes and reads are coherent, in Arm and in other major architectures: in any execution, for each address, the reads of each hardware thread must see a subsequence of the total *coherence order* of all writes to that address. The plain-data CoRR test [46] illustrates one case of this: it is forbidden for a thread to read a new write of `x` and then the initial state for `x`. However, instruction fetches are not necessarily coherent: one instruction fetch may be inconsistent with a program-order-previous fetch, and the data and instruction streams can become out-of-sync with each other. We explore three kinds of coherence:

- Instruction-to-Instruction Coherence: whether fetches of the same location must observe writes to the same location coherently.
- Data-to-Instruction Coherence: whether fetches and then reads to the same location must observe writes to the same location coherently.
- Instruction-to-Data Coherence: whether reads and then fetches of the same location must observe writes to the same location coherently.

Instruction-to-Instruction Coherence Arm explicitly do not guarantee any consistency between fetches of the same location: fetching an instruction does not mean that a later fetch of that location will not see an older instruction [9, B2.4.4]. This is illustrated by CoFF, like CoRR but with fetches instead of reads.

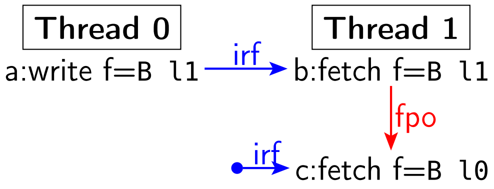
CoFF AArch64

Initial state: `0:W0="B l1", 0:X1=f`

Thread 0	Thread 1	Common

`STR W0,[X1] //a`	`BL f`
	`MOV X0,X10`
	`BL f`
	`MOV X1,X10`
		`f:  B l0`
		`l1: MOV X10,#2`
		`    RET`
		`l0: MOV X10,#1`
		`    RET`

Allowed: `1:X0=2, 1:X1=1`



Here Thread 1 makes two calls to address `f` (`BL` is branch-and-link), while Thread 0 overwrites the instruction at that address. The interesting potential execution is that in which the first call to `f` fetches and executes the newly-written `B l1`, but the second call fetches and executes the original `B l0`. We can view such executions as graphs, similar to previous axiomatic-model candidate executions but with new fetch events, one per instruction, and new edges. As usual, we use `po` and `rf` edges for the program-order and reads-from relations, together with:

- `fe` (fetch-to-execute), which relates the fetch event of an instruction to all the execution events (memory writes, reads or barriers) of the instruction;
- `irf` (instruction-read-from), relating a write to all fetches that read from it (analogous to reads-from, `rf`); and
- `fpo` (fetch-program-order), relating fetches of instructions that are in program order (analogous to program order, `po`).

Edges from the initial state are drawn from a small circle. Since we do not modify the code of most locations, we usually omit the fetch events for those instructions, showing only a subgraph of the interesting events, e.g. as on the right above. For Arm, this execution is both architecturally allowed and experimentally observed.

Here, and in future tests, we assume some common code consisting of a function at address `f` which always has the same shape: a branch that might be overwritten, which selects a block that writes a value to register `X10` before returning. This is sometimes duplicated at different addresses (`f1`, `f2`, ...) or extended to `g`, with three cases. We sometimes elide the common code.

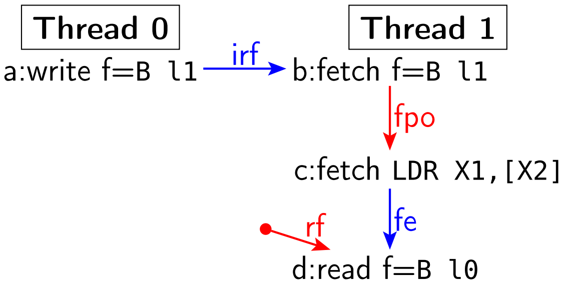
Data-to-Instruction Coherence Fetching from a particular write does imply that program-order-later reads from the same address will see that write (or a coherence successor thereof). This is a *data-to-instruction* coherence property, illustrated by CoFR below. Here Thread 1 fetches the newly-written `B l1` at `f` and then, when reading from `f` with its `LDR` load instruction, cannot read the original `B l0` instruction (it can only read the new `B l1`).

CoFR AArch64

Initial state: `0:W0="B l1", 0:X1=f, 1:X2=f`		
Thread 0	**Thread 1**	**Common**

`STR W0,[X1]`	`BL f`
	`MOV X0,X10`
	`LDR X1,[X2]`
		`f: B l0`
		`l1: MOV X10,#2`
		`RET`
		`l0: MOV X10,#1`
		`RET`

Forbidden: `1:X0=2, 1:X1="B l0"`		



This is not clear in the existing prose specification, but the architectural intent that emerged during discussion with Arm is that the given execution should be forbidden, reflecting microarchitectural choices that (1) instructions decode in order, so the fetch `b` must occur before the read `d`, and (2) fetches that miss in the instruction cache must read from data storage, so the instruction cache cannot be ahead of the available data. This ensures that fetching from a write means that all threads are now guaranteed to read from that write (or another coherence-after it).

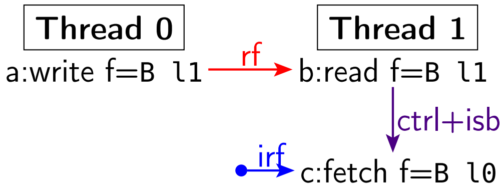
Instruction-to-Data Coherence In the other direction, reading from a particular write to some location does *not* imply that later fetches of that location will see that write (or a coherence successor), as in the following CoRF+ctrl-isb.

CoRF+ctrl-isb AArch64

Initial state: `0:W0="B l1", 0:X1=f, 1:X2=f`		
Thread 0	**Thread 1**	**Common**

`STR W0,[X1]`	`LDR X0,[X2]`
	`CBNZ X0,l`
	`l: ISB`
	`BL f`
	`MOV X1,X10`
		`f: B l0`
		`l1: MOV X10,#2`
		`RET`
		`l0: MOV X10,#1`
		`RET`

Allowed: `1:X0="B l1", 1:X1=1`		



Here Thread 1 has a control dependency and an instruction synchronisation barrier (the `CBNZ` conditional branch, dependent on the value read by its `LDR` load, and `ISB`), abbreviated to ctrl+isb, between its load and the fetch from `f`. If the latter were a data load, this would ensure the two loads are satisfied in order. This is not explicit in the existing prose, but it is what one would expect, and it is observed in practice. Microarchitecturally, it is easily explained by an out-of-date entry for `f` in the instruction cache of Thread 1: if Thread 1 had previously fetched `f` (perhaps speculatively), and that instruction cache entry has not been evicted or explicitly invalidated since, then this fetch of `f` will simply read the old value from the instruction cache without going out to data memory. The `ISB` ensures that `f` is freshly fetched, but does not ensure that Thread 1's instruction cache is up-to-date with respect to data memory.

3.3 Instruction Synchronisation

Instruction fetches satisfy few guarantees, so explicit synchronisation must be performed when modifying the instruction stream.

Same-Thread Synchronisation Test SM below shows the simplest self-modifying code case: without additional synchronisation, a write to program memory can be ignored by a program-order-later fetch.

In this execution, the fetch `b`, fetching the instruction at `f`, fetches a value from a write coherence-before `a`, even though `b` is the fetch of an instruction program-order after `a`. We illustrate this with an *instruction from-reads* (`ifr`)

SM AArch64

Initial state: 0:W0="B l1", 0:X1=f

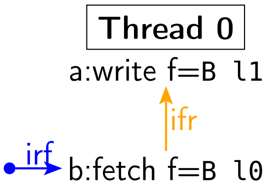
Thread 0

```
STR W0,[X1] // a
BL f
MOV X0,X10
```

Common

```
f:  B l0
l1: MOV X10,#2
    RET
l0: MOV X10,#1
    RET
```

Allowed: 1:X0=1



edge. This is a derived relation, analogous to the usual *from-reads* (fr) relation, that relates each fetch to all writes that are coherence-after the write it read from; it is defined as $\texttt{ifr} = \texttt{irf}^{-1};\texttt{co}$. If the fetch were a data read, this would be a forbidden coherence shape (COWR). As it is, it is architecturally allowed, as described explicitly by Arm [9, B2.4.4], and it is experimentally observed on all devices we have tested. Microarchitecturally, this too is simply due to fetches from old instruction cache entries.

Cache Maintenance As we saw in §2, the Arm architecture provides cache maintenance instructions to synchronise the instruction and data streams: the DC data-cache clean and IC instruction-cache invalidate instructions. To forbid the relaxed outcome of SM, by forcing a fetch of the modified code, the specified sequence of cache maintenance instructions must be inserted, with an ISB.

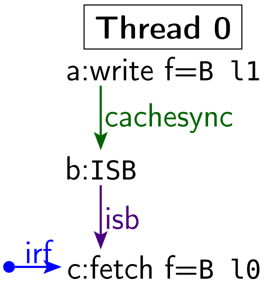
SM+cachesync-isb AArch64

Initial state: 0:W0="B l1", 0:X1=f

Thread 0

```
STR W0,[X1] //overwrite f with branch
DC CVAU,X1  //clean data cache
DSB ISH
IC IVAU,X1  //invalidate instruction cache
DSB ISH
ISB         //flush pipeline
BL f
MOV X0,X10
```

Forbidden: 1:X0=1



Now the outcome is forbidden. The cache synchronisation sequence `DC CVAU; DSB ISH; IC IVAU; DSB ISH` (which we abbreviate to a single `cachesync` edge) ensures that by the time the `ISB` executes, the instruction and data memory have been made coherent with each other for `f`. The `ISB` then ensures the final fetch of `f` is ordered after this sequence. The microarchitectural intuition for this was in §2; our §4 operational model will describe the semantics of each instruction.

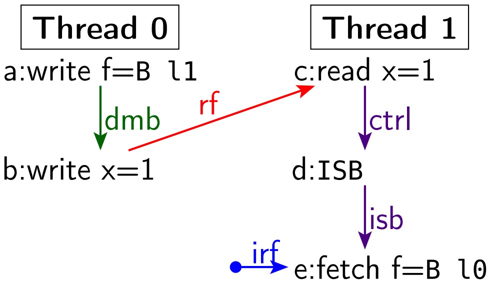
Cross-Thread Synchronisation We now consider modifying code that can be fetched by other threads, using variants of the standard message-passing shape MP. That checks whether two writes (to different locations) on one thread can be seen out-of-order by two reads on another thread; here we replace one or both of those reads by fetches, and ask what synchronisation is required to ensure that the relaxed outcome is forbidden. Consider first an MP variant where the first write is of a new instruction, and the second is just a simple data memory flag:

MP.RF+dmb+ctrl-isb AArch64

Initial state: `0:W0="B l1", 0:X1=f, 0:X2=1, 0:X3=x, 1:X2=x, [x]=0`	
Thread 0	**Thread 1**

`STR W0,[X1]`
`DMB ISH`
`STR X2,[X3]`
	`    LDR X0,[X2]`
	`    CBNZ X0,l`
	`l:  ISB`
	`    BL f`
	`    MOV X1,X10`

Allowed: `1:X0=1, 1:X1=1`	



This test includes sufficient synchronisation on each thread to enforce thread-local ordering of data accesses: the `DMB` in Thread 0 ensures the writes `a` and `b` propagate to memory in program order, and the control-dependency into an `ISB` on Thread 1 ensures the read `c` and the fetch `e` happen in program order. However, as we saw in §2, this is not enough to synchronise concurrent modification and execution of code in ARMv8-A. Thread 0 needs the entire cache synchronization sequence (giving test MP.RF+cachesync+ctrl-isb, not shown), not just a **DMB**, to forbid this outcome.

Another variant of this MP-shape test where the message passing itself is done using modification of code gives a much stronger guarantee, as can be seen from the following MP.FR+dmb+fpo-fe test. This is not clear from the architecture manual, but this outcome is already forbidden with only the **DMB**. This is for similar reasons to the above CoFR test: since Thread 1 fetched the updated value for `f`, we know that value must have reached at least the data caches (since that is where the instruction cache reads from) and therefore multi-copy atomicity guarantees that a normal load instruction will observe it.

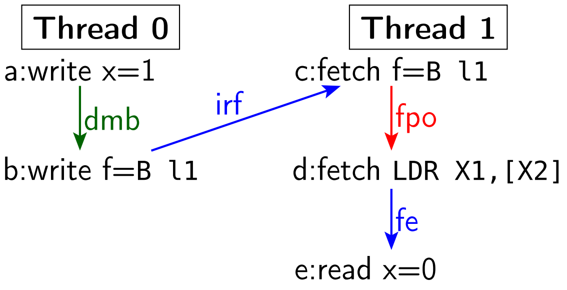
MP.FR+dmb+fpo-fe AArch64

Initial state: `0:X0=1, 0:X1=x, 1:X2=x, [x]=0, 0:W2="B l1", 0:X3=f`

Thread 0	Thread 1

`STR X0,[X1]`
`DMB ISH`
`STR W2,[X3]`
	`BL f`
	`MOV X0,X10`
	`LDR X1,[X2]`

Forbidden: `1:X0=2, 1:X1=0`



The final variant of these MP-shaped tests has both Thread 0 writes be of new instructions. This idiom is very common in practice; it is currently how Chrome's WebAssembly JIT synchronises the modified thread with the new code.

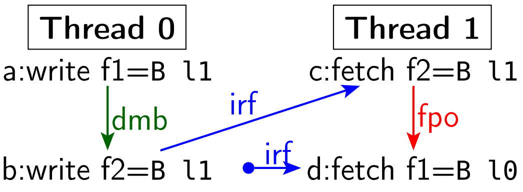
MP.FF+dmb+fpo AArch64

Initial state: `0:W0="B l1", 0:X1=f1, 0:W2="B l1", 0:X3=f2`

Thread 0	Thread 1

`STR W0,[X1]`
`DMB ISH`
`STR W2,[X3]`
	`BL f2`
	`MOV X0,X10`
	`BL f1`
	`MOV X1,X10`

Allowed: `1:X0=2, 1:X1=1`



Without the full cachesync sequence on Thread 0, this is an allowed outcome. Interestingly, adding the cachesync sequence to Thread 0 (Test MP.FF+cachesync+fpo, not shown) is sufficient to make the outcome forbidden, without an `ISB` in Thread 1, as the cachesync sequence is intended to make it appear that fetches occur in program order. Microarchitecturally, that could be ensured in two ways: either by actually fetching in-order, or by making the `IC` instruction not only invalidate all the instruction caches (for this address) but also clean any core's pre-fetch buffer stale entries (for this address). Architecturally, this is not clear in the current prose, but, concurrent with this work, Arm were independently strengthening their definition to make it so.

Incremental Synchronisation The cache synchronisation sequence need not be contiguous, or even all in the same thread. So long as the sequence in its entirety has been performed by the time the fetch happens, then the instruction stream will have been made consistent with the data stream for that address.

This is demonstrated by the following test, where Thread 0 performs a write to `f` and then only a `DC` before synchronizing with Thread 1, which performs the `IC`, while Thread 2 observes the modified code. This can happen in practice when a software thread is migrated between hardware threads at runtime, by a hypervisor or OS. Thread 0 and Thread 1 may just represent the runtime scheduling of a single process, beginning execution on hardware Thread 0 but migrated to hardware Thread 1 between the `DC` and `IC` instructions. In the graph, the dcsync and icsync represent the `DC;DSB ISH` and `DSB ISH;IC;DSB ISH` combinations. The `DC` does not need a preceding `DSB ISH` because it is ordered w.r.t. the preceding store to the same cache line.

Here the `IC` gets broadcast to all threads [9, B2.2.5p3], and so the fact that it happens on a different thread to the `DC` does not affect the outcome. Similarly, if the `DC` were to happen on another thread first (to get the test

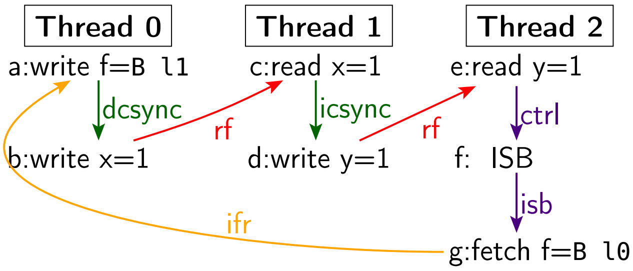
ISA2.F+dc+ic+ctrl-isb AArch64

Initial state: `0:W0="B l1", 0:X1=f, 0:X2=1, 0:X3=x, [x]=0, 1:X4=f, 1:X1=x, 1:X2=1, 1:X3=y, [y]=0, 2:X2=y`

Thread 0	Thread 1	Thread 2

`STR W0,[X1]`
`DC` `CVAU, X1`
`DSB ISH`
`STR X2,[X3]`
	`LDR X0,[X1]`
	`DSB ISH`
	`IC` `IVAU, X4`
	`DSB ISH`
	`STR X2,[X3]`
		`LDR X0,[X2]`
		`CBZ X0,l`
		`l:`**`ISB`**
		`BL f`
		`MOV X1,X10`

Forbidden: `1:X0=1, 1:X1=1`



MP.RF+[dc]-ic+ctrl-isb, not shown), then it would have the effect of ensuring consistency globally, for all threads.

3.4 Multi-Copy Atomicity

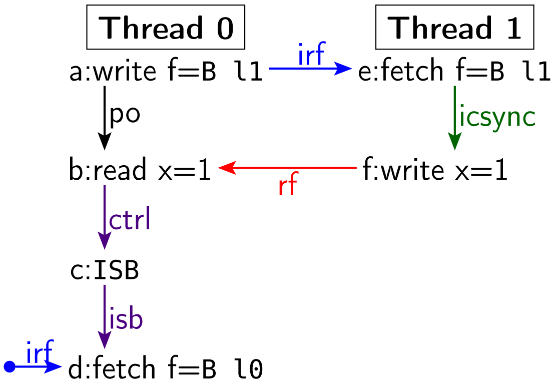
For data accesses, the question of whether they are *multi-copy atomic* is a crucial one for relaxed architectures. IBM POWER, ARMv7, and pre-2018 ARMv8-A are/were non-multi-copy atomic: two writes to different addresses could become visible to distinct other threads in different orders. Post-2018 ARMv8-A and RISC-V are multi-copy atomic (or “other multi-copy-atomic” in Arm terminology) [37,36,9]: the programmer can assume there is a single shared memory, with all relaxed-memory effects due to thread-local out-of-order execution.

However, for fetches, due to the lack of any fetch atomicity guarantee for most instructions (§3.1), and the lack of coherent fetches for the others (§3.2), the question of multi-copy atomicity is not particularly interesting. Tests are either trivially forbidden (by data-to-instruction coherence) or are allowed but only the full cache synchronisation sequence provides enough guarantees to forbid it, and (§3.3) this ensures all cores will share the same consistent view of memory.

3.5 Strength of the IC Instruction

Multiple Points of Unification Cleaning the data cache, using the DC instruction, makes a write visible to instruction memory. It does this by pushing the write past the Point of Unification. However, there may be multiple Points of Unification: one for each core, where its own instruction and data memory become unified, and one for the entire system (or shareability domain) where all the caches unify. Fetching from a write implies that it has reached the closest PoU, but does not imply it has reached any others, even if the write originated from a distant core. Consider: Here Thread 0 modifies f, Thread 1 fetches the new value and performs just an IC and DSB, before signalling Thread 0 which also fetches f. That IC is not strong enough to ensure that the write is pulled into the instruction cache of Thread 0.

```
SM.F+ic                                     AArch64
Initial state: 0:W0="B l1", 0:X4=f,
0:X3=x, [x]=0, 1:X4=f, 1:X2=1, 1:X3=x
      Thread 0          Thread 1
   STR W0,[X4]      BL f
   LDR X2,[X3]      MOV X0,X10
   CBZ X2,l         IC IVAU, X4
l: ISB              DSB ISH
   BL f             STR X2,[X3]
   MOV X1,X10
Allowed: 1:X0=2, 0:X2=1, 0:X1=1
```



This is not clear in the existing prose, but the architectural intent is that it be allowed (i.e., that IC is weak in this respect). We have not so far observed it in practice. The write may have passed the Point of Unification for Thread 1, but not the shared Point of Unification for both threads. In other words, the write might reach Thread 1's instruction cache without being pushed down from Thread 0's data cache. Microarchitecturally this can be explained by *direct data intervention* (DDI), an optimisation allowing cache lines to be migrated directly from one thread's (data) cache to another. The line could be migrated from Thread 0 to Thread 1, then pushed past Thread 1's Point of Unification, making it visible to Thread 1's instruction memory without ever making it visible to Thread 0's own instruction memory. The lack of coherence between instruction and data caches would make this observable, even in multi-copy atomic machines.

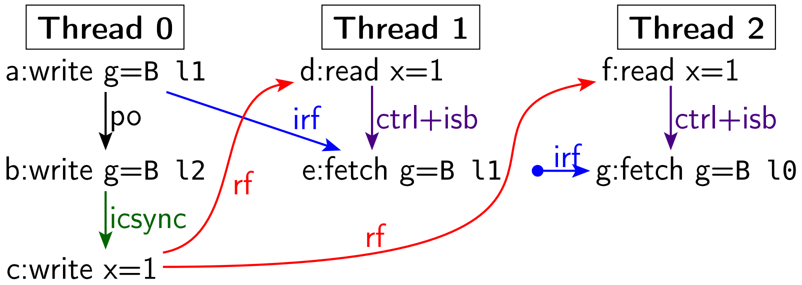
Stale Fetches So far, we have only talked about fetching from two distinct writes. But theoretically there is no limit to how far back we can fetch from, with insufficient synchronization. The MP.RF+dmb+ctrl-isb test (§3.3) required the full `cachesync` sequence to forbid the given behaviour. Below we give a test, FOW, similar to that MP-shaped test but allowing many consumer threads to independently and simultaneously see different values in their instruction memory, even after invalidating their caches.

FOW AArch64

Initial state: `0:W0="B l1", 0:X2=g, 0:W1="B l2", 0:X3=1, 0:X4=x, [x]=0, 1:X4=x, 2:X4=x`			
Thread 0	**Thread 1**	**Thread 2**	**Common**

`STR W0,[X2]`
`STR W1,[X2]`
`DSB ISH`
`IC` `IVAU, X2`
`DSB ISH`
`STR X3,[X4]`
	`    LDR X0, [X4]`
	`    CBNZ X0, la`
	`la: `**`ISB`**
	`    BL g`
	`    MOV X1,X10`
		`    LDR X0, [X4]`
		`    CBNZ X0, lb`
		`lb: `**`ISB`**
		`    BL g`
		`    MOV X1,X10`
			`g:  B l0`
			`l2: MOV X10, #3`
			`    RET`
			`l1: MOV X10, #2`
			`    RET`
			`l0: MOV X10, #1`
			`    RET`

Allowed: `1:X0=1, 1:X1=2, 2:X0=1, 2:X1=1`			



This is not clear in the existing architecture text. It is a case where the architecture design is not very constrained. On the one hand, it has not been observed, and it is thought unlikely that hardware will ever exhibit this behaviour: it would require keeping multiple writes in the coherent part of the data caches, rather than a single dirty line, which would require more complex cache coherence protocols. On the other hand, there does not seem to be any benefit to software from forbidding it. Arm therefore prefer the choice that gives a simpler and weaker model (here the two happen to coincide), to make it easier to understand and to provide more flexibility for future microarchitectural optimisations. We therefore design our models to allow the above behaviour.

3.6 Strength of the `DC` Instruction

Instruction Cache depth Test CoFF (§3.2) showed that fetches can see "old" writes. In principle, there is no limit to the depth of the instruction-cache hierarchy: there could be many values for a single location cached in the instruction memory for each core, even if the data cache has been cleaned. The test below illustrates this, with Thread 1 able to see all three values for `g`.

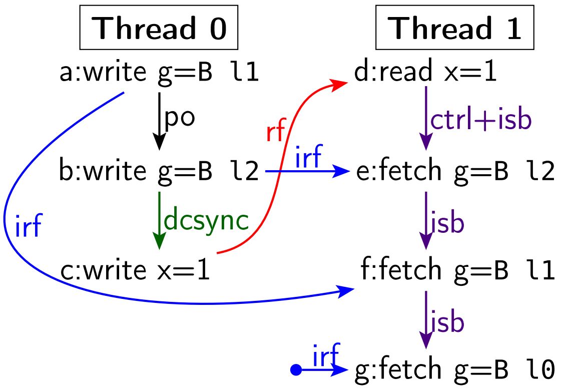
MP.RF+dc+ctrl-isb-isb AArch64

Initial state: `0:W0="B l1", 0:X2=g, 0:W1="B l2", 0:X3=1, 0:X4=x, [x]=0, 1:X4=x`

Thread 0	Thread 1	Common

`STR W0,[X2]`
`STR W1,[X2]`
`DSB ISH`
`DC` `CVAU,X2`
`DSB ISH`
`STR X3,[X4]`
	`LDR X0, [X4]`
	`CBNZ X0, l`
	`l:`**`ISB`**
	`BL g`
	`MOV X1,X10`
	`ISB`
	`BL g`
	`MOV X2,X10`
	`ISB`
	`BL g`
	`MOV X3,X10`
		`g: B l0`
		`l2:MOV X10,#3`
		`RET`
		`l1:MOV X10,#2`
		`RET`
		`l0:MOV X10,#1`
		`RET`

Allowed: `1:X0=1, 1:X1=3, 1:X2=2, 1:X3=1`

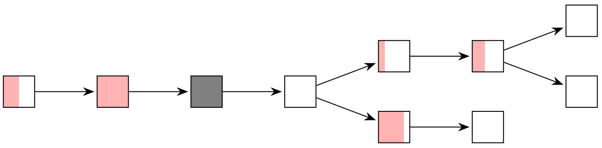


This is similar to the preceding FOW case: it is thought unlikely that hardware will exhibit this in practice, but the desire for the simpler and weaker option means the architectural intent is to allow it, and we follow that in our models.

4 An Operational Semantics for Instruction Fetch

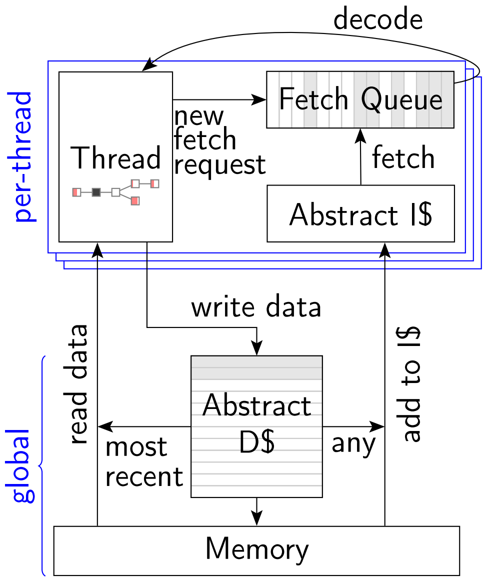
Previous work on operational models for IBM POWER and Arm "user-mode" concurrency [46,45,22,18,19,37] has shown, surprisingly, that as far as programmer-visible behaviour is concerned, one can abstract from almost all hardware implementation details of data memory (store queues, the cache hierarchy, the cache protocol, etc.). For ARMv8-A, following their 2018 shift to a multicopy-atomic architecture, one can do so completely: the *Flat* model of [37] has a shared flat memory, with a per-thread out-of-order thread subsystem, modelling pipeline effects, responsible for all observable relaxed behaviour. For instruction-fetch, it is no longer possible to abstract completely from the data and instruction cache hierarchy, but we can still abstract from much of it.

The Flat Model is a small-step operational semantics for multi-copy atomic ARMv8-A, including the relaxed behaviours of loads and stores [37]. Its states are abstract machine states consisting of a tree of instructions for each thread, and a flat memory subsystem shared by all threads. Each instruction in each thread corresponds to a sequence of transitions, with some guards and a potential effect on the shared memory state. The Flat model is made executable in our RMEM tool, which can exhaustively interleave transitions to enumerate all the possible behaviours. The tree of instructions for each thread models out-of-order and speculative execution explicitly. Below we show an example for a thread that is executing 10 instruction instances. Some (grey) are finished, no longer subject to restart; others (pink) have run some but perhaps not all of their instruction semantics; instructions are not necessarily atomic. Those with multiple children are branch instructions with multiple potential successors speculated simultaneously.



For each state, the model defines the set of allowed transitions, each of which steps to a new machine state. Transitions correspond to steps of single instructions, and individual instructions may give rise to many. Example transitions include Register Write, Propagate Write to Memory, etc.

iFlat Extension Originally, Flat had a fixed instruction memory, with a single transition that can speculate the address of any program-order successor of any instruction in flight, fetch it from the fixed instruction memory, and decode it. We now remove that fixed instruction memory, so that instructions can be fetched from data writes, and add the additional structures as shown on the right. These are all of unbounded size, as is appropriate for an architecture definition.



Fetch Queues (per-thread) These are ordered buffers of pre-fetched entries, waiting to be decoded and begin execution. Entries are either a fetched 32-bit opcode, or an unfetched request. The fetch queues allow the model to speculate and pre-fetch many instructions ahead of where the thread is currently executing. The model's fetch queues abstract from multiple real-hardware structures: instruction queues, line-fill buffers, loop buffers, and slots objects. We keep a close relation to this underlying microarchitecture by allowing out-of-order fetches, but we believe this is not experimentally observable on real hardware.

Abstract Instruction Cches (per-thread) These are just sets of writes. When the fetch queue requests a new entry, it gets satisfied from the instruction cache, either immediately (a *hit*) or at some later point in time (a *miss*). The instruction cache can contain many possible writes for each location (§3.6), and it can be spontaneously updated with new writes in the system at any time ([9, B2.4.4]). To manage `IC` instructions, each thread keeps a list of addresses yet to be invalidated by in-flight `IC`s.

Data Cache (global) Above the single shared flat memory for the entire system, which sufficed for the multi-copy-atomic ARMv8-A data memory, we insert a shared buffer which is just a list of writes; abstracting from the many possible coherent data cache hierarchies. Data reads must be coherent, reading from the most recent write to the same address in the buffer, but instruction fetches are allowed to read from any such write in the buffer (§3.2).

Transitions To accommodate instruction fetch and cache maintenance, we introduce new transitions: Fetch Request, Fetch Instruction, Fetch Instruction (Unpredictable), Fetch Instruction (B.cond), Decode Instruction, Begin IC, Propagate IC to Thread, Complete IC, Perform DC, and Update Instruction Cache. We also have to modify some Flat transitions: Commit ISB, Wait for DSB, Commit DSB, Propagate Memory Write, and Satisfy Read from Memory. These transitions define the lifecycle of each instruction: a request gets issued for the fetch, then at some later point the fetch gets satisfied from the instruction cache, the instruction is then decoded (in program-order) and then handed to the existing semantics to be executed. To give a flavour, we show just one, the *Propagate IC to Thread* transition, which is responsible for invalidation of the abstract instruction caches. This is a prose rendering of the rule in our executable mathematical model, which is expressed in the typed functional subset of Lem [32].

Propagate IC to Thread An instruction *i* (with ID *iiid*) in state WAIT_IC*(address, state_cont)* can do the relevant invalidate for any thread *tid'*, modifying that thread's instruction cache and fetch queue, if there exists a pending entry *(iiid, address)* in that thread's *ic_writes*. Action:

1. for any entry in the fetch queue for thread *tid*, whose *program_loc* is in the same minimum-size instruction cache line as *address*, and is in FETCHED(_) state, set it to the UNFETCHED state;
2. for the instruction cache of thread *tid*, remove any write-slices which are in the same instruction cache line of minimum size as *address*.

This rule can be found under the same name in the full prose description, and in the `handle_ic_ivau` and `flat_propagate_cache_maintenance` functions in `machineDefThreadSubsystem.lem` and `machineDefFlatStorageSubsystem.lem` in the executable mathematics. Cache maintenance operations work over entire cache lines, not individual addresses. Each address is associated with at least one cache line for the data (and unified) caches, and one for the instruction caches. The cache line of minimum size is the (architected) smallest possible cache line for each of these.

Example This model correctly explains all the behaviours of §3. We illustrate this by revisiting the cache synchronization explanation of §2, which can now be re-interpreted w.r.t. our precise model, and using this to explain the thread migration case of §3.3. Given `DC Xn; DSB; IC Xn; DSB` we can use this model to give meaning to it (omitting uninteresting transitions): First the `DC CVAU` causes a **Perform DC** transition. This pushes any write that might have been in the abstract data cache into memory. Now the first `DSB`'s **Commit DSB** can be taken, allowing **Begin IC** to happen. This creates entries for each thread, which are discharged by each **Propagate IC to Thread** (see above). Once all entries are invalidated, a **Complete IC** can happen. Now, if any thread decodes an instruction for that address, it must have been fetched from the write the `DC` pushed, or something coherence-after it. If the software thread performing this sequence is interrupted and migrated (by the OS) to a different hardware thread, then, so long as the OS includes the `DSB` to maintain the thread-local DC ordering, the `DC` will push the write in an identical way, since it only affects the global abstract data cache. The `IC` transitions can all be taken, and the sequence continues as before, just on a new hardware thread. So when the second `DSB` finishes, and the final **Commit DSB** transitions is taken, the effect of the full sequence will be seen system-wide even if the thread was migrated.

5 An Axiomatic Semantics for Instruction Fetch

Based on the operational model, we develop an axiomatic semantics, as an extension of the ARMv8 axiomatic reference model [15,37]. Since that does not have mixed-size support, we do not model the concurrent modification of conditional branches (§3.1), as this would require mixed-size machinery. The existing axiomatic model is a predicate on *candidate executions*, hypothetical complete executions of the given program that satisfy some basic well-formedness conditions, defining the set of *valid* executions to be those satisfying its axioms. Each candidate execution abstractly captures a particular concrete execution of the program in terms of events and relations over them. This model is expressed in the herd language [8,6,4]. The events of these executions are memory reads (the set `R`), memory writes (`W`), and memory barrier/fence events (`F`). The relations are: *program order* (`po`), capturing the sequencing of events by the same thread in the execution's control-flow unfolding; *reads-from* (`rf`), relating a write event w with any read event r that reads from it; the *coherence order* (`co`), recording the execution's sequencing of same-address writes in memory; and *read-modify-write* (`rmw`), capturing which load/store exclusive instructions form a successful exclusive *pair* in the execution. The derived relation *from-reads* $\mathtt{fr} = \mathtt{rf}^{-1}\mathtt{;co}$ relates a read r with a write w' if r reads from a write w coherence before w'. In addition, candidate executions also have relations capturing dependencies between events: address (addr), data (data), and control dependencies (ctrl). The relation loc relates any two read/write events that are to the same memory address. The model also has relations suffixed "i" and "e": rfi/rfe, coi/coe, fri/fre. These are the restrictions of the relations rf, co, and fr, to same-thread/"internal" event pairs or different-thread/"external" event pairs. The model is defined in relational algebra. In herd, R;S stands for sequential composition of relations R and S, R^{-1} for the inverse of relation R, R|S and R&S for the union and intersection of R and S, and [A];R;[B] for the restriction of R to the domain A and range B.

Handling instruction fetch requires extending the notion of candidate execution. We add new events: an *instruction-fetch* (IF) event for each executed instruction; a DC event for each DC CVAU instruction; an IC event for each IC IVAU and IC IALLU instruction. We replace po with *fetch-program-order* (fpo) which orders the IF event of an instruction before any program-order later IF events. We add a relation *same-cache-line* (scl), relating reads, writes, fetches, DC and IC events to addresses in the same cache line. We add an acyclic transitively closed relation wco, which extends co with orderings for cache maintenance (DC or IC) events: it includes an ordering (e, e') or (e', e) for any cache maintenance event e and same-cache-line event e' if e' is a write or another cache maintenance event; where co = ([W];wco;[W]) & loc. The loc, addr, and ctrl are all extended to include DC and IC events. We add a *fetch-to-execute* relation (fe), relating an IF event to any event generated by the execution of that instruction; and an *instruction-read-from* relation (irf), which relates a write to any IF event that fetches from it. Finally, we add a boolean *constrained-unpredictable* (CU) to detect badly behaved programs. Now we derive the following relations: the standard po relation, as po = $\mathtt{fe}^{-1}$;fpo;fe (two events e and e' are po-related if their fetch-events are fpo-related); and *instruction-from-reads* (ifr), the analogue of fr for instruction fetches, relating a fetch to all writes coherence-after the one it fetched from: ifr = $\mathtt{irf}^{-1}$;co.

We then make two semantics-preserving rewrites of the existing model to make adding instruction fetches easier (described in the appendix); and make the following changes and additions to the model. The full model is shown in Figure 2, with comments pointing to the relevant locations in the model definition. For lack of space we only describe the main addition, the `iseq` relation, in detail (including its correspondence with the operational model of §4); for the others we give an overview and refer to the appendix for the full description.

```
let iseq = [W];(wco&scl);[DC];  (*1*)
                (wco&scl);[IC]

(* Observed-by *)
let obs = rfe | fr | wco        (*2*)
        | irf | (ifr;iseq)      (*3, 4*)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]       (*5*)
  | [IF]; fe                    (*6*)
  | [ISB]; fe^-1; fpo           (*7*)

(* Dependency-ordered-before *)
let dob = addr | data
  | ctrl; [W]
  | (ctrl | (addr; po)); [ISB]
(*| [ISB]; po; [R] *)           (*8*)
  | addr; po; [W]
  | (addr | data); rfi

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

(* Barrier-ordered-before *)
let bob = [R|W]; po; [dmb.sy]
  | [dmb.sy]; po; [R|W]
  | [L]; po; [A]
  | [R]; po; [dmb.ld]
  | [dmb.ld]; po; [R|W]
  | [A|Q]; po; [R|W]
  | [W]; po; [dmb.st]
  | [dmb.st]; po; [W]
  | [R|W]; po; [L]
  | [R|W|F|DC|IC]; po; [dsb.ish]   (*9*)
  | [dsb.ish]; po; [R|W|F|DC|IC]   (*10*)
  | [dmb.sy]; po; [DC]             (*11*)

(* Cache-op-ordered-before *)
let cob = [R|W]; (po&scl); [DC]    (*12*)
  | [DC]; (po&scl); [DC]           (*13*)

(* Ordered-before *)
let ob = (obs|fob|dob|aob|bob|cob)+

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

(* External visibility requirement *)
irreflexive ob as external

(* Atomic *)
empty rmw & (fre; coe) as atomic

(* Constrained unpredictable *)
let cff = ([W];loc;[IF]) \         (*14*)
            ob^-1 \ (co;iseq;ob)
cff_bad cff ≡ CU                   (*15*)
```

Fig. 1. Axiomatic model

We define the relation `iseq`, relating some write w to address x to an `IC` event completing a cache synchronisation sequence (not necessarily on a single thread): w is followed by a same-cache line `DC` event, which is in turn followed by a same-cache line `IC` event. In operational model terms, this captures traces that propagated w to memory, subsequently performed a same-cache-line `DC`, and then began an `IC` (and eagerly propagated the `IC` to all threads). In any state after this sequence it is guaranteed that w, or a coherence-newer same-address write, is in the instruction cache of all threads: performing the `DC` has cleared the abstract data cache of writes to x, and the subsequent `IC` has removed old instructions for location x from the instruction caches, so that any subsequent updates to the instruction caches have been with w, or `co`-newer writes. Adding `ifr;iseq` to the *observed-by* relation (`obs`) `(4)` relates an instruction fetch i to location x to an `IC` *ic* if: i fetched from a write w to x, some write w' to x is coherence-after w, and *ic* completes a cache synchronisation sequence (`iseq`) starting from w'. Then the `irreflexive ob` axiom requires that i must be ordered-before *ic* (because it would otherwise have fetched w').We now briefly overview other changes made to the axiomatic model and their intuition. We include `irf` in `obs (3)`: for an instruction to be fetched from a write, the write has to have been done before. We add a relation *fetch-ordered-before* (`fob`) `(5-7)`, which is included in *ordered-before*. The relation `fob` includes `fpo` and `fe`; including `fpo (5)` requires fetches to be ordered according to their position in the control-flow unfolding of the execution. and including the `fe` (*fetch-to-execute*) relation `(6)` captures the idea that an instruction must be fetched before it can execute; fetches program-order-after an ISB happen after the ISB (or else are restarted) `(7)`. For `DSB ISH` instructions the edge `[R|W|F|DC|IC];po;[dsb.ish]` is included in `ob (9)`: `DSB ISH`s are ordered with all program-order-preceding non-fetch events. Symmetrically, all non-`IF` events are ordered after program-order-preceding `dsb.ish` events `(10)`. `DC`s wait for preceding `dmb.sy` events `(11)`. We include the relation *cache-op-ordered-before* (`cob`) in `ob`. This relation orders `DC` instructions with program-order previous reads/writes and other `DC`s to the same cache line `(12,13)`.

Finally, *could-fetch-from* (`cff`) `(14)` captures, for each fetch i, the writes it could have fetched from (including the one it did fetch from), which we use to define the *constrained unpredictable* axiom `cff_bad` (not given) `(15)`.

6 Validation

To gain confidence in the presented models we validated the models against the Arm architectural intent, against each other, and against real hardware.

Validation against the Architecture To ensure our models correctly captured the architectural intent we engaged in detailed discussions with Arm, including the Arm chief architect. These involved inventing litmus tests (including, those described in §3 and many others) and discussing what the architecture should allow in each case.

Validating against hardware To run instruction-fetch tests on hardware, we extended the litmus tool [7]. The most significant extension consists in handling code that can be modified, and thus has to be restored between experiments. To that end, code copies are executed, those copies reside in mmap'd memory with (execute permission granted. Copies are made from "master" copies, in effect C functions whose contents basically consist of gcc extended inline assembly. Of course, such code has to be position independent, and explicit code addresses in test initialisation sections (such as in `0:X1=l` in the test of §3.1) are specific to each copy. All the cache handling instructions used in our experiments are all allowed to execute at exception level 0 (user-mode), and therefore no additional privilege is needed to run the tests.

To automatically generate families of interesting instruction-fetch tests, we extended the diy test generation tool [3] to support instruction-fetch reads-from (`irf`) and instruction-fetch from-reads (`ifr`) edges, in both internal (same-thread) and external (inter-thread) forms, and the `cachesync` edge. We used this to generate 1456 tests involving those edges together with `po`, `rf`, `fr`, `addr`, `ctrl`, `ctrlisb`, and `dmb.sy`. diy does not currently support bare `DC` or `IC` instructions, locations which are both fetched and read from, or repeated fetches from the same location.

We then ran the diy-generated test suite on a range of hardware implementations, to collect a substantial sample of actual hardware behaviour.

Correspondence between the models We experimentally test the equivalence of the operational and axiomatic models on the above hand-written and diy-generated tests, checking that the models give the same sets of allowed final states, and that these are consistent with the hardware observations.

Making the models executable as a test oracle To make the operational model executable as a test oracle, capable of computing the set of all allowed executions of a litmus test, we must be able to *exhaustively enumerate* all possible traces. For the model as presented, doing this naively is infeasible: for each instruction it is theoretically possible to speculate any of the 2^{64} addresses as potential next address, and the interleaving of the new fetch transitions with others leads to an additional combinatorial explosion.

We address these with two new optimisations. First, we extend the fixed-point optimisation in RMEM (incrementally computing the set of possible branch targets) [37] to keep track not only of indirect branches but also the successors of every program location, and only allow speculating from this set of successors. Additionally, we track during a test which locations were both fetched and modified during the test, and eagerly take fetch and decode transitions for all other locations. As before, the search then runs until the set of branch targets *and* the set of modified program-locations reaches a fixed point. We also take some of the transitions eagerly to reduce the search space, in cases where this cannot remove behaviour: **Wait for IC**, **Complete IC**, **Fetch Request**, and **Update Instruction Cache**.

Making the axiomatic model executable as a test oracle The axiomatic model is expressed in a herd-like form, but the herd tool does not support instruction fetch and cache maintenance instructions. To make the model executable as a test oracle, we built a new tool that takes litmus tests and uses a Sail [11] definition of a fragment of the ARMv8-A ISA to generate SMT problems for the model. Using the Sail instruction semantics, we generate a Sail program that corresponds to each thread within a litmus test. The tool then partially evaluates these programs using the concrete values for addresses and registers specified in the litmus file, while allowing memory values and arbitrary addresses to remain symbolic. Using a Sail to SMT-LIB backend, these are translated into SMT definitions that include all possible behaviours of each thread as satisfiable solutions. The rules for the axiomatic model are then applied as assertions restricting the possible behaviours to just those allowed by the axiomatic model. The tool also derives the addr and data relations, using the syntactic dependencies within the instruction semantics to derive the syntactic dependencies between instructions.

For litmus tests, where we can know up-front which instructions may be modified, we would like to avoid generating IF events for instructions that cannot be modified. If we naively removed certain IF events, however, we would break the correspondence between po and $\texttt{fe}^{-1}$;fpo;fe. This can be worked around by ensuring that every modifiable instruction generates an event which appears in po, allowing fpo between the modifiable instructions to instead be derived as fe;po;$\texttt{fe}^{-1}$. Branches emit a special branch address announce event for this purpose, which is also used to derive the ctrl relation. The fpo relation can then be modified, replacing [ISB];$\texttt{fe}^{-1}$;fpo with [ISB];po;$\texttt{fe}^{-1}$ and adding [ISB];po. The second change ensures that all the transitive edges generated by [ISB];$\texttt{fe}^{-1}$;fpo followed by [IF];fe remain with fob and hence ob.

A limitation of this approach is it cannot support cases where two threads both attempt to execute the same possibly-modified instruction, as in the SM.F+ic and FOW tests.

Validation results First, to check for regressions, we ran the operational model on all the 8950 non-mixed-size tests used for developing the original Flat model (without instruction fetch or cache maintenance). The results are identical, except for 23 tests which did not terminate within two hours. We used a 160 hardware-thread POWER9 server to run the tests.

We have also run the axiomatic model on the 90 basic two-thread tests that do not use Arm release/acquire instructions (not supported by the ISA semantics used for this); the results are all as they should be. This takes around 30 minutes on 8 cores of a Xeon Gold 6140.

Then, for the key handwritten tests mentioned in this paper, together with some others (that have also been discussed with Arm), we ran them on various hardware implementations and in the operational and axiomatic models. The models' results are identical to the Arm architectural intent in all cases, except for two tests which are not currently supported by the axiomatic checker.

Test	Arm intent	op. model	ax. model	hardware obs.
CoFF	allow	=	=	42.6k/13G
CoFR	forbid	=	=	0/13G
CoRF+ctrl-isb	allow	=	=	3.02G/13G
SM	allow	=	=	25.8G/25.9G
SM+cachesync-isb	forbid	=	=	0/25.9G
MP.RF+dmb+ctrl-isb	allow	=	=	480M/6.36G
MP.RF+cachesync+ctrl-isb	forbid	=	=	0/13G
MP.FR+dmb+fpo-fe	forbid	=	=	0/13G
MP.FF+dmb+fpo	allow	=	=	447M/13G
MP.FF+cachesync+fpo	forbid	=	=	$^{\mathrm{F}}$2.3k/13G
ISA2.F+dc+ic+ctrl-isb	forbid	=	=	0/6.98G
SM.F+ic	allow	=	unsupported	$^{\mathrm{U}}$0/12.9G
FOW	allow	=	unsupported	$^{\mathrm{U}}$0/7G
MP.RF+dc+ctrl-isb-isb	allow	=	=	$^{\mathrm{U}}$0/12.94G
MP.R.RF+addr-cachesync+dmb+ctrl-isb	forbid	=	=	0/6.97G
MP.RF+dmb+addr-cachesync	allow	=	=	$^{\mathrm{U}}$0/6.34G

[The hardware observations are the sum of testing seven devices: a Snapdragon 810 (4x Arm A53 + 4x Arm A57 cores), Tegra K1 (2x NVIDIA Denver cores), Snapdragon 820 (4x Qualcomm Kryo cores), Exynos 8895 (4x Arm A53 + 4x Samsung Mongoose 2 cores), Snapdragon 425 (4x Arm A53), Amlogic 905 (4x Arm A53 cores), and Amlogic 922X (4x Arm A73 + 2x Arm A53 cores). U: allowed but unobserved. F: forbidden but observed.]

Our testing revealed a hardware bug in a Snapdragon 820 (4 Qualcomm Kryo cores). A version of the first cross-thread synchronisation test of §3.3 but with the full cache synchronisation (MP.RF+cachesync+ctrl-isb) exhibited an illegal outcome in 84/1.1G runs (not shown in the table), which we have reported. We have also seen an anomaly for MP.FF+cachesync+fpo, currently under investigation by Arm. Apart from these, the hardware observations are all allowed by the models. As usual, specific hardware implementations are sometimes stronger.

Finally, we ran the 1456 new instruction-fetch diy tests on a variety of hardware, for around 10M iterations each, and in the operational model. The model is sound with respect to the observed hardware behaviour except for that same Snapdragon 820 device.

7 Related Work

To the best of our knowledge, no previous work establishes well-validated rigorous semantics for any systems aspects, of any current production architecture, in a realistic concurrent setting.

The closest is Raad et al.'s work on non-volatile memory, which models the required cache maintenance for persistent storage in ARMv8-A [39], as an extension to the ARMv8-A axiomatic model, and for Intel x86 [38] as an operational model, but neither are validated against hardware. In the sequential case, Myreen's JIT compiler verification [33] models x86 icache behaviour with an abstract cache that can be arbitrarily updated, cleared on a `jmp`. For address translation, the authoritative Arm-internal ASL model [40,41,42], and Sail model derived from it [11] cover this, and other features sufficient to boot an OS (Linux), as do the handwritten Sail models for RISC-V (Linux and FreeBSD) and MIPS/CHERI-MIPS (FreeBSD, CheriBSD), but without any cache effects. Goel et al. [21,20] describe an ACL2 model for much of x86 that covers address translation; and the Forvis [34] and RISCV-PLV [14] Haskell RISC-V ISA models are also complete enough to boot Linux. Syeda and Klein [49,50] provide an somewhat idealised model for ARMv7 address translation and TLB maintenance. Komodo [16] uses a handwritten model for a small part of ARMv7, as do Guanciale et al. [25,12]. Romanescu et al. [44,43] do discuss address translation in the concurrent setting, but with respect to idealised models. Lustig et al. [30] describe a concurrent model for address translation based on the Intel Sandy Bridge microarchitecture, combined with a synopsis of some of the relevant Linux code, but not an architectural semantics for machine-code programs.

8 Conclusion

The mainstream architectures are the most important programming languages used in practice, and their systems aspects are fundamental to the security (or lack thereof) of our computing infrastructure. We have established a robust semantics for one of those systems aspects, soundly abstracting the hardware complexities to a manageable model that captures the architectural intent. This enables future work on reasoning, model-checking, and verification for real systems code.

Acknowledgements This work would not have been possible without generous technical assistance from Arm. We thank Richard Grisenthwaite, Will Deacon, Ian Caulfield, and Dave Martin for this. We also thank Hans Boehm, Stephen Kell, Jaroslav Ševčík, Ben Titzer, and Andrew Turner, for discussions of how instruction cache maintenance is used in practice, and Alastair Reid for comments on a draft. This work was partially supported by EPSRC grant EP/K008528/1 (REMS), ERC Advanced Grant 789108 (ELVER), an ARM iCASE award, and ARM donation funding. This work is part of the CIFV project sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8650-18-C-7809. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

References

1. Adir, A., Attiya, H., Shurek, G.: Information-flow models for shared memory with an application to the PowerPC architecture. IEEE Trans. Parallel Distrib. Syst. **14**(5),

502–515 (2003). https://doi.org/10.1109/TPDS.2003.1199067

2. Alglave, J., Fox, A., Ishtiaq, S., Myreen, M.O., Sarkar, S., Sewell, P., Zappa Nardelli, F.: The semantics of Power and ARM multiprocessor machine code. In: Proc. DAMP 2009 (Jan 2009)
3. Alglave, J., Maranget, L.: The `diy7` tool. `http://diy.inria.fr/` (2019), accessed 2019-07-08
4. Alglave, J., Maranget, L.: The `herd7` tool. `http://diy.inria.fr/doc/herd.html/` (2019), accessed 2019-07-08
5. Alglave, J., Maranget, L., Deplaix, K., Didier, K., Sarkar, S.: The `litmus7` tool. `http://diy.inria.fr/doc/litmus.html/` (2019), accessed 2019-07-08
6. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in weak memory models. In: Proc. CAV (2010)
7. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Litmus: running tests against hardware. In: Proceedings of TACAS 2011: the 17th international conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 41–44. Springer-Verlag, Berlin, Heidelberg (2011), `http://dl.acm.org/citation.cfm?id=1987389.1987395`
8. Alglave, J., Maranget, L., Tautschnig, M.: Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. ACM TOPLAS **36**(2), 7:1–7:74 (Jul 2014). https://doi.org/10.1145/2627752
9. ARM Limited: ARM architecture reference manual. ARMv8, for ARMv8-A architecture profile (Oct 2018), v8.4. ARM DDI 0487D.a (ID103018)
10. Armstrong, A., Bauereiss, T., Campbell, B., Gray, S.F.J.F.K.E., Kerneis, G., Krishnaswami, N., Mundkur, P., Norton-Wright, R., Pulte, C., Reid, A., Sewell, P., Stark, I., Wassell, M.: Sail. `https://www.cl.cam.ac.uk/~pes20/sail/` (2019)
11. Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K.E., Norton, R.M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami, N., Sewell, P.: ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In: Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (Jan 2019). https://doi.org/10.1145/3290384, proc. ACM Program. Lang. 3, POPL, Article 71
12. Baumann, C., Schwarz, O., Dam, M.: Compositional verification of security properties for embedded execution platforms. In: PROOFS@CHES 2017, 6th International

Workshop on Security Proofs for Embedded Systems, Taipei, Taiwan, Friday September 29th, 2017. pp. 1–16 (2017), http://www.easychair.org/publications/paper/wkpS

13. Chong, N., Ishtiaq, S.: Reasoning about the ARM weakly consistent memory model. In: MSPC (2008)

14. Clester, I.J., Bourgeat, T., Wright, A., Gruetter, S., Chlipala, A.: riscv-plv risc-v isa formal specification. https://github.com/mit-plv/riscv-semantics (2019), accessed 2019-07-01

15. Deacon, W.: The ARMv8 application level memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat (accessed 2019-07-01) (2016)

16. Ferraiuolo, A., Baumann, A., Hawblitzel, C., Parno, B.: Komodo: Using verification to disentangle secure-enclave hardware from software. In: Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017. pp. 287–305 (2017). https://doi.org/10.1145/3132747.3132782

17. Flur, S., French, J., Gray, K., Pulte, C., Sarkar, S., Sewell, P.: rmem. www.cl.cam.ac.uk/~pes20/rmem/ (2017)

18. Flur, S., Gray, K.E., Pulte, C., Sarkar, S., Sezgin, A., Maranget, L., Deacon, W., Sewell, P.: Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In: Proceedings of POPL: the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2016)

19. Flur, S., Sarkar, S., Pulte, C., Nienhuis, K., Maranget, L., Gray, K.E., Sezgin, A., Batty, M., Sewell, P.: Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In: The 44st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France. pp. 429–442 (Jan 2017). https://doi.org/10.1145/3009837.3009839

20. Goel, S.: The x86isa books: Features, usage, and future plans. In: Proceedings 14th International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, May 22-23, 2017. pp. 1–17 (2017). https://doi.org/10.4204/EPTCS.249.1, arXiv version: https://arxiv.org/abs/1705.01225

21. Goel, S., Hunt, W.A., Kaufmann, M., Ghosh, S.: Simulation and formal verification of x86 machine-code programs that make system calls. In: Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design. pp. 18:91–18:98. FMCAD

’14, FMCAD Inc, Austin, TX (2014), http://dl.acm.org/citation.cfm?id=2682923.2682944

22. Gray, K.E., Kerneis, G., Mulligan, D., Pulte, C., Sarkar, S., Sewell, P.: An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In: Proc. MICRO-48, the 48th Annual IEEE/ACM International Symposium on Microarchitecture (Dec 2015)

23. Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: An extensible architecture for building certified concurrent OS kernels. In: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. pp. 653–669 (2016), https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

24. Gu, R., Shao, Z., Kim, J., Wu, X.N., Koenig, J., Sjöberg, V., Chen, H., Costanzo, D., Ramananandro, T.: Certified concurrent abstraction layers. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 646–661 (2018). https://doi.org/10.1145/3192366.3192381

25. Guanciale, R., Nemati, H., Dam, M., Baumann, C.: Provably secure memory isolation for linux on ARM. Journal of Computer Security **24**(6), 793–837 (2016). https://doi.org/10.3233/JCS-160558

26. Intel Corporation: Intel 64 and ia-32 architectures software developer’s manual combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d and 4. https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4, accessed 2019-06-30 (May 2019), 325462-070US

27. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems **32**(1), 2:1–2:70 (Feb 2014). https://doi.org/10.1145/2560537

28. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14, San Diego, CA, USA, January 20-21, 2014. pp. 179–192 (2014). https://doi.org/10.1145/2535838.2535841

29. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning **43**(4), 363–446 (2009). https://doi.org/10.1007/s10817-009-9155-4

30. Lustig, D., Sethi, G., Martonosi, M., Bhattacharjee, A.: COATCheck: Verifying memory ordering at the hardware-OS interface. SIGOPS Oper. Syst. Rev. **50**(2), 233–247 (Mar 2016). https://doi.org/10.1145/2954680.2872399

31. Maranget, L., Sarkar, S., Sewell, P.: A tutorial introduction to the ARM and POWER relaxed memory models. Draft available from `http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf` (2012)

32. Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.: Lem: reusable engineering of real-world semantics. In: Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 175–188 (2014). https://doi.org/10.1145/2628136.2628143

33. Myreen, M.O.: Verified just-in-time compiler on x86. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 107–118. POPL '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1706299.1706313

34. Nikhil, R.S., Sharma, N.N.: Forvis: A formal RISC-V ISA specification. `https://github.com/rsnikhil/Forvis_RISCV-ISA-Spec` (2019), accessed 2019-07-01

35. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Proceedings of TPHOLs 2009: Theorem Proving in Higher Order Logics, LNCS 5674. pp. 391–407 (2009)

36. Pulte, C.: The Semantics of Multicopy Atomic ARMv8 and RISC-V. Ph.D. thesis, University of Cambridge (2019), `https://doi.org/10.17863/CAM.39379`

37. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In: Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (Jan 2018). https://doi.org/10.1145/3158107

38. Raad, A., Wickerson, J., Neiger, G., Vafeiadis, V.: Persistency semantics of the Intel-x86 architecture. PACMPL **4**(POPL), 11:1–11:31 (2020). https://doi.org/10.1145/3371079

39. Raad, A., Wickerson, J., Vafeiadis, V.: Weak persistency semantics from the ground up: Formalising the persistency semantics of ARMv8 and transactional models. Proc. ACM Program. Lang. **3**(OOPSLA), 135:1–135:27 (Oct 2019). https://doi.org/10.1145/3360561

40. Reid, A.: Trustworthy specifications of ARM v8-A and v8-M system level architecture. In: FMCAD 2016. pp. 161–168 (October 2016), https://alastairreid.github.io/papers/fmcad2016-trustworthy.pdf

41. Reid, A.: ARM releases machine readable architecture specification. https://alastairreid.github.io/ARM-v8a-xml-release/ (Apr 2017)

42. Reid, A., Chen, R., Deligiannis, A., Gilday, D., Hoyes, D., Keen, W., Pathirane, A., Shepherd, O., Vrabel, P., Zaidi, A.: End-to-end verification of processors with ISA-Formal. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9780, pp. 42–58. Springer (2016)

43. Romanescu, B., Lebeck, A., Sorin, D.J.: Address translation aware memory consistency. IEEE Micro **31**(1), 109–118 (Jan 2011). https://doi.org/10.1109/MM.2010.99

44. Romanescu, B.F., Lebeck, A.R., Sorin, D.J.: Specifying and dynamically verifying address translation-aware memory consistency. In: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems. pp. 323–334. ASPLOS XV, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1736020.1736057

45. Sarkar, S., Memarian, K., Owens, S., Batty, M., Sewell, P., Maranget, L., Alglave, J., Williams, D.: Synchronising C/C++ and POWER. In: Proceedings of PLDI 2012, the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (Beijing). pp. 311–322 (2012). https://doi.org/10.1145/2254064.2254102

46. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: Proceedings of PLDI 2011: the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation. pp. 175–186 (2011). https://doi.org/10.1145/1993498.1993520

47. Sarkar, S., Sewell, P., Zappa Nardelli, F., Owens, S., Ridge, T., Braibant, T., Myreen, M., Alglave, J.: The semantics of x86-CC multiprocessor machine code. In: Proceedings of POPL 2009: the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. pp. 379–391 (Jan 2009). https://doi.org/10.1145/1594834.1480929

48. Sewell, P., Sarkar, S., Owens, S., Zappa Nardelli, F., Myreen, M.O.: x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Communications of the ACM **53**(7), 89–97 (Jul 2010), (Research Highlights)

49. Syeda, H., Klein, G.: Reasoning about translation lookaside buffers. In: LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017. pp. 490–508 (2017), http://www.easychair.org/publications/paper/340347

50. Syeda, H.T., Klein, G.: Program verification in the presence of cached address translation. In: Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings. pp. 542–559 (2018). https://doi.org/10.1007/978-3-319-94821-8_32

51. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A.C.J., Owens, S., Norrish, M.: The verified CakeML compiler backend. J. Funct. Program. **29**, e2 (2019). https://doi.org/10.1017/S0956796818000229

52. Waterman, A., Asanović, K. (eds.): The RISC-V Instruction Set Manual Volume I: Unprivileged ISA (Dec 2018), document Version 20181221-Public-Review-draft. Contributors: Arvind, Krste Asanović, Rimas Avižienis, Jacob Bachmeyer, Christopher F. Batten, Allen J. Baum, Alex Bradbury, Scott Beamer, Preston Briggs, Christopher Celio, Chuanhua Chang, David Chisnall, Paul Clayton, Palmer Dabbelt, Roger Espasa, Shaked Flur, Stefan Freudenberger, Jan Gray, Michael Hamburg, John Hauser, David Horner, Bruce Hoult, Alexandre Joannou, Olof Johansson, Ben Keller, Yunsup Lee, Paul Loewenstein, Daniel Lustig, Yatin Manerkar, Luc Maranget, Margaret Martonosi, Joseph Myers, Vijayanand Nagarajan, Rishiyur Nikhil, Jonas Oberhauser, Stefan O'Rear, Albert Ou, John Ousterhout, David Patterson, Christopher Pulte, Jose Renau, Colin Schmidt, Peter Sewell, Susmit Sarkar, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn, Caroline Trippel, Ray VanDeWalker, Muralidaran Vijayaraghavan, Megan Wachs, Andrew Waterman, Robert Watson, Derek Williams, Andrew Wright, Reinoud Zandijk, and Sizhuo Zhang

Appendices

We include three appendices for the interested reader:

A Further semantics design questions, illustrated with litmus tests.
B A full prose description of the iFlat operational model. The formal and executable Lem semantics is available online at the URL in the Introduction, as is the RMEM web interface tool for running it.
C Additional explanation of the iFlat axiomatic model.

A Additional Instruction Fetch Phenomena and Examples

Multi-Copy Atomicity Data-to-instruction coherence requires that local memory accesses ordered after a fetch are coherent with the instruction stream. In ARMv8's multi-copy atomic architecture, this also enforces that *all other observers* also observe that same coherent view.

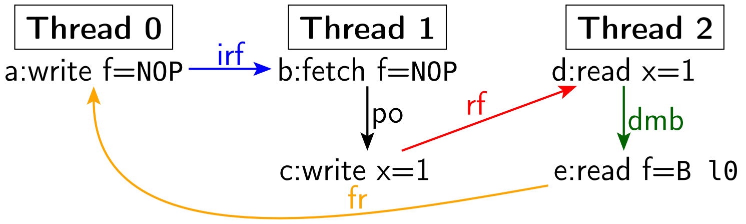
This is demonstrated by the following WRC.F.RR+po+dmb test, where the fetch b in Thread 1 requires the subsequent read in Thread 2 to see the updated value.

WRC.F.RR+po+dmb AArch64

Initial state: `0:W0="NOP", 0:X1=f,` `1:X1=1, 1:X2=x, [x]=0, 2:X1=x, 2:X3=f`		
Thread 0	**Thread 1**	**Thread 2**

`STR W0,[X1]`	`BL f`
	`MOV X0,X10`
	`STR X1, [X2]`
		`LDR X0,[X1]`
		`DMB SY`
		`LDR X2,[X3]`

Forbidden: `1:X0=1, 2:X0=2, 2:X2="B l0"`		



This is true even if the observed read is not the location that was fetched:

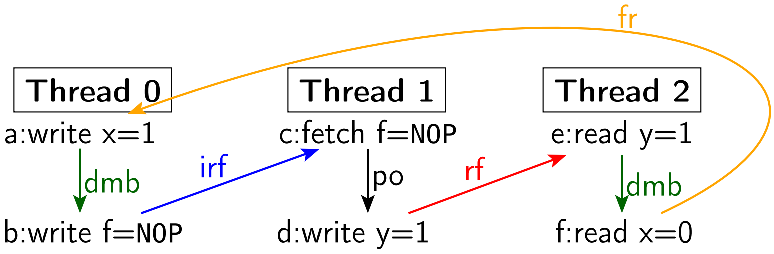
ISA2.F+dmb+po+dmb AArch64

Initial state:
`0:X0=1, 0:X1=x, [x]=0,0:W2="NOP", 0:X3=f`
`1:X1=1, 1:X2=y, [y]=0,2:X1=y, 2:X3=x`

Thread 0	Thread 1	Thread 2

`STR X0,[X1]`
`DMB SY`
`STR W2,[X3]`
	`BL f`
	`MOV X0,X10`
	`STR X1, [X2]`
		`LDR X0,[X1]`
		`DMB SY`
		`LDR X2,[X3]`

Forbidden: `1:X0=1, 2:X0=2, 2:X2="B l0"`



However, instruction memory itself is not multi-copy atomic.

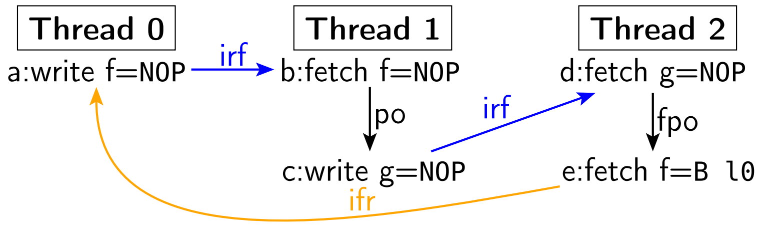
WRC.F.FF+pos AArch64

Initial state:
`0:W0="NOP", 0:X1=f,1:W1="NOP", 1:X2=g`

Thread 0	Thread 1	Thread 2

`STR W0,[X1]`	`BL f`
	`MOV X0,X10`
	`STR W1, [X2]`
		`BL f`
		`MOV X0,X10`
		`BL g`
		`MOV X1,X10`

Allowed: `1:X0=2, 2:X0=2, 2:X1=1`



`DC` and `IC` Address Speculation Normal data load and store instructions (in ARMv8-A and in other relaxed architectures) respect *address dependencies*: reads cannot be satisfied, and writes cannot be forwarded from or committed, until their addresses are resolved from previous register writes (though those can still be out-of-order or speculative). In other words, the architecture forbids programmer-visible value speculation of such addresses.

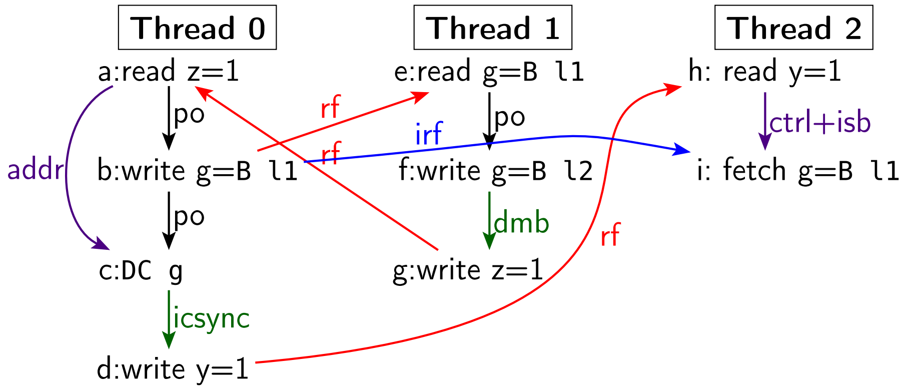
The same question arises here for `DC CVAU` and `IC IVAU`, which are loosely analogous to loads from the specified addresses: they respect address dependencies but not control dependencies. The following test illustrates the former for `DC`. Thread 0 writes to `g` and performs the full cache synchronization sequence. However, the `DC`'s address depends on a detour through Thread 1 which writes an even newer instruction to `g`. Since the address of the `DC` cannot be speculated, this address dependency must be preserved and so the final fetch of `g` after the cache synchronization must observe the branch Thread 1 wrote.

MP.R.RF+addr-cachesync+dmb+ctrl-isb AArch64

Initial state: `0:X1=z, 0:W2="B l1", 0:X3=g, 0:X5=1, 0:X6=y, 1:W1="B l2", 1:X2=g, 1:X3=1, 1:X4=z, 2:X2=y, [x]=0, [y]=0`

Thread 0	Thread 1	Thread 2	Common
`LDR X0,[X1]`	`LDR W0,[X2]`	`LDR X0,[X2]`	`g: B l0`
`STR W2,[X3]`	`STR W1,[X2]`	`CBNZ X0,l`	`l2: MOV X10, #3`
`EOR X4,X0,X0`	`DMB SY`	`l:` **`ISB`**	`RET`
`ADD X4,X4,X3`	`STR X3,[X4]`	`BL g`	`l1: MOV X10, #2`
`DC` `CVAU,X4`		`MOV X1,X10`	`RET`
`DSB ISH`			`l0: MOV X10, #1`
`IC` `IVAU,X4`			`RET`
`DSB ISH`			
`STR X5,[X6]`			

Forbidden: `0:X0=1, 1:W0="B l1", 2:X0=1, 2:X1=1`



This is unclear in the current prose, but the architectural intent is that it should be forbidden: addresses of cache maintenance instructions should not be visibly value-speculated, and these instructions must respect their address dependencies.

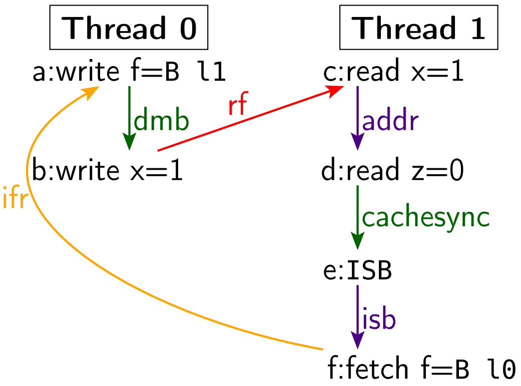
DC Might Be To Same Address Data loads and stores can be ordered by the fact that they might access the same address [31, §12.5]. Arm [9, D4.4.8] *Ordering and completion of data and instruction cache instructions* makes clear that `DC` is ordered with respect to loads and stores with addresses in the same cache line, while `IC` is not. We therefore have to ask whether `DC` is subject to a might-access-same-address restriction in the same way as data loads and stores. The test below illustrates this, with a case in which program-order previous load/store addresses may not be determined when the `DC` executes. The architectural intent (not clear from the current text) is that `DC` should be like loads in this respect too, with the following test allowed. Microarchitecturally, the `DC` is not required to wait for those addresses to be determined before executing, but if they end up being to the same address, the `DC` must be re-issued. Because the read `d` was not to the same location, the `DC` need not be re-issued and so may have happened before the write `a` to `f`.

MP.RF+dmb+addr-cachesync AArch64

Initial state: `0:W0="B l1", 0:X1=f, 0:X2=1, 0:X3=x, [x]=0, 1:X1=x, 1:X4=z, [z]=0, 1:X5=f`

Thread 0	Thread 1
STR W0,[X1]	LDR X0,[X1]
DMB SY	EOR X2,X0,X0
STR X2,[X3]	LDR X3,[X4,X2]
	DC CVAU,X5
	DSB ISH
	IC IVAU,X5
	DSB ISH
	ISB
	BL f
	MOV X6,X10

Allowed: `1:X0=1, 1:X6=1`



`DC` Ordering with po-previous/successor Memory Accesses? We know that the `DC` instruction is ordered with po-previous stores to the same address. But does this imply any other ordering between the **DC** and po-related instructions?

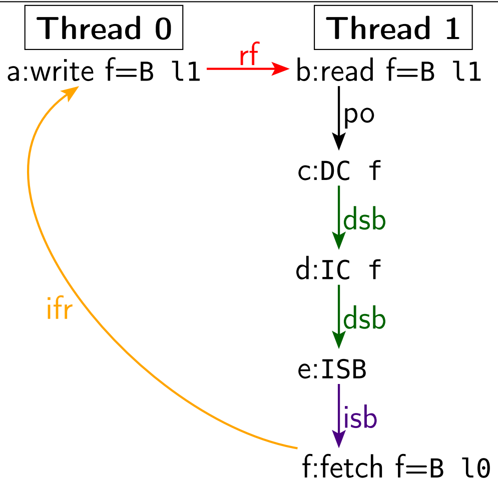
po-previous loads If the **DC** is ordered with respect to po-previous loads, then the following CoRF+cachesync-isb test is forbidden.

CoRF+cachesync-isb AArch64

Initial state: `0:W0="B l1", 0:X1=f` `1:X1=f`		
Thread 0	**Thread 1**	**f**

`STR W0,[X1] // a`	`LDR W0,[X1] // b`
	`DC` `CVAU, [X1] // c`
	`DSB ISH`
	`IC` `IVAU, [X1] // d`
	`DSB ISH`
	`ISB` `// d`
	`BL f`
		`f:  B l0  // e`
		`l1: MOV X2,#2`
		`    RET`
		`l0: MOV X",#1`
		`    RET`

Final state: Forbidden?		



***DC** before po-successor loads and stores*

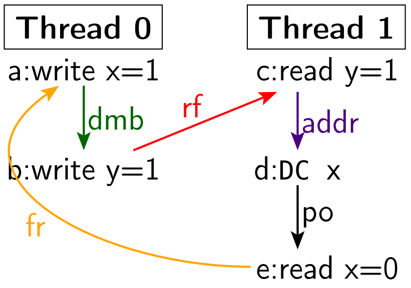
MP+dmb+addr-dc AArch64

Initial state: `0:X0=1, 0:X1=x`
`0:X2=1, 0:X3=y`
`1:X1=y, 1:X3=x`

Thread 0	Thread 1

`STR X0,[X1] // a`
`DSB SY`
`STR X2,[X3] // b`
	`LDR X0,[X1] // c`
	`EOR X5,X5,X0`
	`ADD X5,X5,X3`
	`DC` `CVAU,X5 // d`
	`LDR X2,[X3] // e`

Final state: Allowed?



Same-Cache-Line Ordering ARMv8 has an architected *cache line of minimum size*. Accessible as the `DMinLine` and `IMinLine` bitfields of the `CTR_EL0` register, encoding the smallest cache-line size for the data and instruction caches, respectively.

Given `X` is an array of $2^{\text{DMinLine}-1}$ 64-bit integers, and `X` is aligned on a cache boundary. Therefore, all `X[i]` are in the same (data) cache line of minimum size.

We assume a cache line size of at least 16 bytes (4 words) for these tests.

Is there any preserved ordering between loads and stores to different addresses the same cache line of minimum size? We believe the answer to be No, and our models currently allow the following SB+scls test, however, we have not fully explored these semantics with the architects.

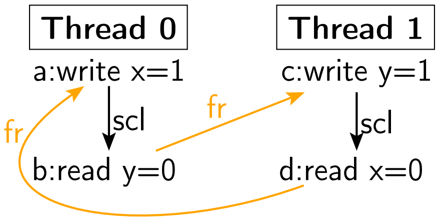
SB+scls AArch64

Initial state:
`0:X0=1, 0:X1=X[0], 0:X3=X[1]`
`1:X0=1, 1:X1=X[1], 1:X3=X[0]`

Thread 0	Thread 1

`STR X0,[X1] // a`
`LDR X2,[X3] // b`
	`STR X0,[X1] // c`
	`LDR X2,[X3] // d`

Final state: `0:X2=0, 1:X2=0`



DC to same cache line Given two locations `f` and `g` are in the same cache line of minimum size, then performing the cache clearing sequence for one will also clear the other.

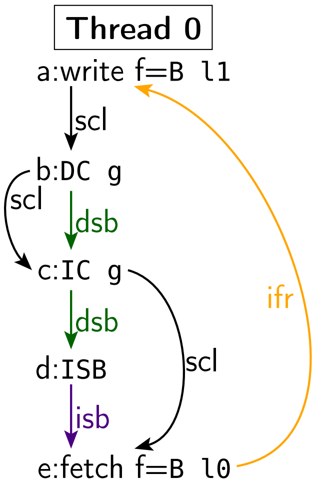
SM+sclcachesync-isb AArch64

Initial state:
`0:W0="B l1", 0:X1=f, 0:X2=g`

Thread 0	f

`STR W0,[X1] //a`
`DC`` CVAU, [X2] //b`
`DSB ISH`
`IC`` IVAU, [X2] //c`
`DSB ISH`
`ISB`` // d`
`BL f // e`
	`f:  B l0`
	`l1: MOV X0,#2`
	`    RET`
	`l0: MOV X0,#1`
	`    RET`

Forbidden: `0:X0=1`



Mixed-Size Instruction Fetching In the tests so far we have always replaced a single instruction, with another whole instruction, with a single write. But it is easy to imagine code that replaces an instruction byte-by-byte, or perhaps even only replacing a single field in the instruction encoding! But are these valid architecturally?

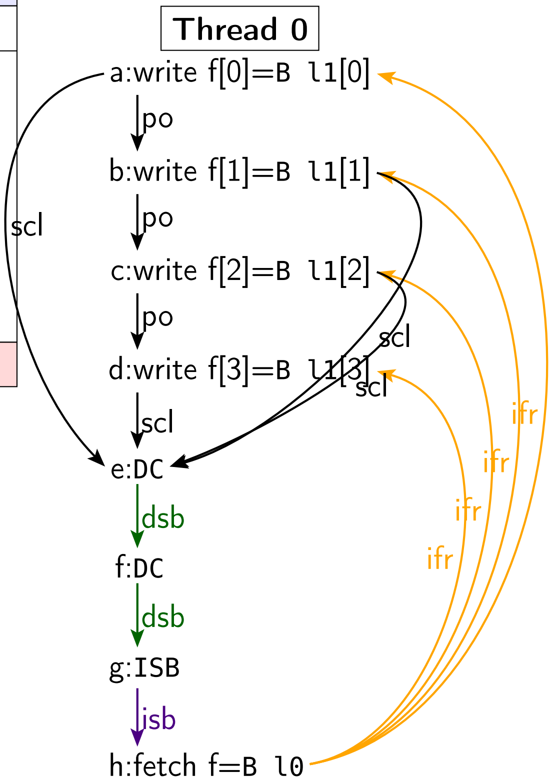
It seems clear that performing individual per-byte writes and then performing the cache synchronization should give the desired result without 'bad' (unpredictable) behaviour.

SM+sclcachesync-isb AArch64

Initial state: `0:<W0,W1,W2,W3>="B l1"` `0:X1=f, 0:X2=g`	
Thread 0	**f**

`STRB W0,[X4,#0] // a`
`STRB W1,[X4,#1] // b`
`STRB W2,[X4,#2] // c`
`STRB W3,[X4,#3] // d`
`**DC** CVAU, [4] // e`
`DSB ISH`
`**IC** IVAU, [X4] // f`
`DSB ISH`
`**ISB** // g`
`BL f`
	`f:  B l0  // h`
	`l1: MOV X0,#2`
	`    RET`
	`l0: MOV X0,#1`
	`    RET`

Forbidden: `0:X0=1`	



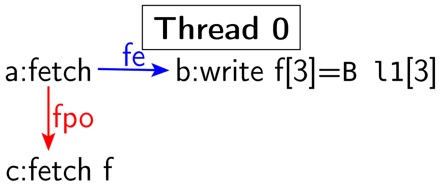
It is less clear what happens if one were to *concurrently* modifiy part of an instruction. This is not discussed in detail, and the questions are left open for the architects.

SM+mixed AArch64

Initial state: `0:W0="B l1"[3], 0:X1=f`	
Thread 0	**f**

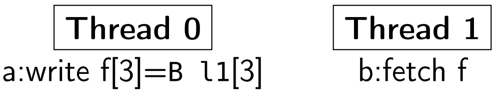
`STRB W0,[X1,#3] // a, b`
`BL f`
	`f:  B l0   // c`
	`l1: MOV X0,#2`
	`    RET`
	`l0: MOV X0,#1`
	`    RET`

Final state: Unpredictable?	



W+F+mixed AArch64

Initial state: `0:W0="B l1"[3], 0:X1=f`	
Thread 0	**Thread 1**
`STRB W0,[X1,#3] // a`	`BL f // b`
Final state: Unpredictable?	



B Full Operational Semantics

This appendix gives a prose description of the iFlat operational model, as formally defined in its Lem definition.

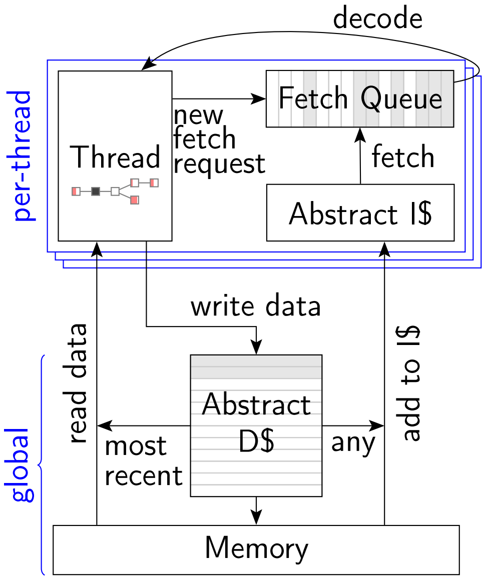
B.1 Introduction

To help reading this document we have colour-coded some text as follows:

- [release/acquire] Release/Acquire instructions
- [exclusive] Exclusive instructions
- [`dmb ld`/`dmb st`] `dmb ld` and `dmb st` instructions
- [ifetch] Instruction fetch and cache maintenance instructions

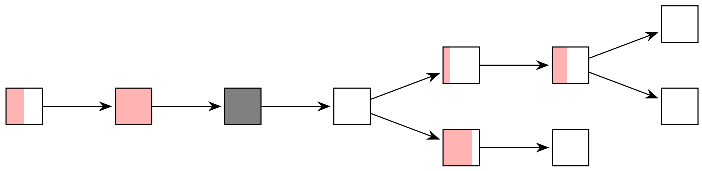
The operational model is expressed as a state machine, with states that are an abstract representation of hardware machine states. We first introduce the model states and transitions informally.

Model states A model state consists just of a shared memory and a tuple of thread model states:



The shared memory state effectively just records the most recent write to each location. To handle load/store-exclusives, the memory is extended with a map (the exclusives map) from read requests to sets of write slices, that associates a read request of a load-exclusive with the write slices it read from (excluding writes that have been forwarded to the read and have not reached memory yet). To handle instruction fetching, the shared memory is extended with a data cache buffer of all the writes still visible to instruction fetches. Each thread is extended with an instruction cache that can be fetched from and fetch queue of buffered pre-fetched instructions.

Each thread model state consists principally of a list or tree of instruction instances, some of which have been finished, and some of which have not. Below we show an example for a thread that is executing 10 instruction instances. Some (grey) are finished; others (pink) have run some but perhaps not all of their instruction semantics; instructions are not necessarily atomic. Those with multiple children are branch instructions with multiple potential speculative successors being explored simultaneously.



Non-finished instruction instances can be subject to restart, e.g. if they depend on an out-of-order or speculative read that turns out to be unsound. The finished instances are not necessarily contiguous: in the example, the third is finished even though its predecessors are not, which can only happen if they are sufficiently independent. Instruction instances with multiple children are conditional branches for which the thread has fetched multiple possible successors. When a conditional branch is finished, any un-taken alternative paths are discarded, and instruction instances that follow (in program order) a non-finished conditional branch cannot be finished until that conditional branch is. One can choose whether or not to allow simultaneous exploration of multiple successors of a conditional branch (as shown above); this does not affect the set of allowed outcomes.

The intra-instruction behaviour of a single instruction can largely be treated as sequential (but not atomic) execution of its ASL/Sail pseudocode. Each instruction instance state includes a pseudocode execution state, which one can think of as a representation of the pseudocode control state, pseudocode call stack, and local variable values. An instruction instance state also includes information, detailed below, about the instruction instance's memory and register footprints, its register and memory reads and writes, whether it is finished, etc.

Model transitions For any state, the model defines the set of allowed transitions, each of which is a single atomic step to a new abstract machine state. Each transition arises from the next step of a single instruction instance; it will change the state of that instance, and it may depend on or change the rest of its thread state and/or the shared memory state. Instructions cannot be treated as atomic units: complete execution of a single instruction instance may involve many transitions, which can be interleaved with those of other instances in the same or other threads, and some of this is programmer-visible. The transitions are introduced below and defined in §B.5, with a precondition and a construction of the post-transition model state for each. The transitions labelled ◦ can always be taken eagerly, as soon as they are enabled, without excluding other behaviour; the – cannot.

Transitions for all instructions:

◦ [ifetch] Fetch request: This transition speculates the next address as a po-successor of a previously speculated instruction.
– [ifetch] Fetch instruction: Satisfy the fetch from instruction memory.
◦ [ifetch] Decode instruction: Decode the instruction.
◦ Register read: This is a read of a register value from the most recent program-order predecessor instruction instance that writes to that register.
◦ Register write

- Pseudocode internal step: this covers ASL/Sail internal computation, function calls, etc.
- Finish instruction: At this point the instruction pseudocode is done, the instruction cannot be restarted or discarded, and all memory effects have taken place. For a conditional branch, any non-taken po-successor branches are discarded.

Load instructions:

- Initiate memory reads of load instruction: At this point the memory footprint of the load is provisionally known and its individual reads can start being satisfied.
- Satisfy memory read by forwarding from writes: This partially or entirely satisfies a single read by forwarding from its po-previous writes.
- Satisfy memory read from memory: This entirely satisfies the outstanding slices of a single read, from memory.
- Complete load instruction (when all its reads are entirely satisfied): At this point all the reads of the load have been entirely satisfied and the instruction pseudocode can continue execution. A load instruction can be subject to being restarted until the Finish instruction transition. In some cases it is possible to tell that a load instruction will not be restarted or discarded before that, e.g. when all the instructions po-before the load instruction are finished. The Restart condition over-approximates the set of instructions that might be restarted.

Store instructions:

- Initiate memory writes of store instruction, with their footprints: At this point the memory footprint of the store is provisionally known.
- Instantiate memory write values of store instruction: At this point the writes have their values and program-order-subsequent reads can be satisfied by forwarding from them.
- Commit store instruction: At this point the store is guaranteed to happen (it cannot be restarted or discarded), and the writes can start being propagated to memory.

– Propagate memory write: This propagates a single write to memory.

○ Complete store instruction (when its writes are all propagated): At this point all writes have been propagated to memory, and the instruction pseudocode can continue execution.

Store-exclusive instructions:

– Guarantee the success of store-exclusive: This guarantees the success of the store-exclusive.

– Make a store-exclusive fail: This makes the store-exclusive fail.

Barrier instructions:

○ Commit barrier

Cache maintenance instructions:

– [ifetch] Begin IC: Initiate instruction cache maintenance.

○ [ifetch] Propagate IC to thread: Do instruction cache maintenance for a specific thread.

– [ifetch] Perform DC: Clean the abstract data cache for a specific cache line.

Instruction cache updates:

○ [ifetch] Add to instruction cache for thread: Update instruction cache for thread with write.

B.2 Intra-instruction Pseudocode Execution

To link the model transitions introduced above to the execution of the instructions an interface is needed between Sail and the rest of the concurrency model. For each instruction instance this intra-instruction semantics is expressed as a state machine, essentially running the instruction pseudocode, where each pseudocode execution state is a request of one of the following forms:

READ_MEM(read_kind, address, size, read_continuation)	Read request
EXCL_RES(res_continuation)	Store-exclusive result
PERFORM_IC(address, res_continuation)	Propagate an `ic ivau`
WAIT_IC(address, res_continuation)	Wait for an `ic ivau` to complete
PERFORM_DC(address, res_continuation)	Propagate a `dc cvau`
WRITE_EA(write_kind, address, size, next_state)	Write effective address
WRITE_MEMV(memory_value, write_continuation)	Write value
BARRIER(barrier_kind, next_state)	Barrier
READ_REG(reg_name, read_continuation)	Register read request
WRITE_REG(reg_name, register_value, next_state)	Write register
INTERNAL(next_state)	Pseudocode internal step
DONE	End of pseudocode

Each of these states is a suspended computation with a request for an action or input from the concurrency model and, except in the case of DONE, a continuation for the remaining execution.

Here memory values are lists of bytes, addresses are 64-bit numbers, read and write kinds identify whether they are regular, exclusive, and/or release/acquire operations, register names identify a register and slice thereof (start and end bit indices), and the continuations describe how the instruction instance will continue for any value that might be provided by the surrounding memory model. This largely follows [22, §2.2], except that memory writes are split into two steps, WRITE_EA and WRITE_MEMV. We ensure these are paired in the pseudocode, but there may be other steps between them: it is observable that the WRITE_EA can occur before the value to be written is determined, because the potential memory footprint of the instruction becomes provisionally known then.

We ensure that each instruction has at most one memory read, memory write, or barrier step, by rewriting the pseudocode to coalesce multiple reads or writes, which are then split apart into the architecturally atomic units by the thread semantics; this gives a single commit point for all memory writes of an instruction.

Each bit of a register read should be satisfied from a register write by the most recent (in program order) instruction instance that can write that bit, or from the thread's initial register state if there is no such. That instance may not have executed its register write yet, in which case the register read should block. The semantics therefore has to know the register write footprint of each instruction instance, which it calculates when the instruction instance is created. We ensure in the pseudocode that each instruction does exactly one register write to each bit of its register footprint, and also that instructions do not do register reads from their own register writes. In some cases, but not in the fragment of ARM that we cover at present, register write footprints need to be dynamically recalculated, when the actual footprint only becomes known during pseudocode execution.

Data-flow dependencies in the model emerge from the fact that a register read has to wait for the appropriate register write to be executed (as described above). This has to be carefully handled in order not to create unintentional strength. First, for some instructions we need to ensure that the pseudocode is in the maximally liberal order, e.g. to allow early computed-address register writebacks before the corresponding memory write. Leaving load-pair aside (which we do not cover), and the treatment of the multiple reads or writes that can be associated with a single load or store instruction (which we do), we have not so far needed other intra-instruction concurrency. Second, the model has to be able to know when a register read value can no longer change (i.e. due to instruction restart). We approximate that by recording, for each register write, the set of register and memory reads the instruction instance has performed at the point of executing the write. This information is then used as follows to determine whether a register read value is final: if the instruction instance that performed the register write from which the register reads from is finished, the value is final; otherwise check that the recorded reads for the register write do not include memory reads, and continue recursively with the recorded register reads. For the instructions we cover this approximation is exact.

We express the pseudocode execution semantics in two ways: a definitional interpreter for Sail [22], with an exhaustive symbolic mode to (re)calculate an instruction's memory and register footprints, and as a shallow embedding, translating Sail into directly executable code, with separate hand-written definitions of the footprint functions. The two are essentially equivalent: the first lets one small-step through the pseudocode interactively, while the second is more efficient and should be more convenient for proof.

B.3 Instruction Instance States

Each instruction instance *i* has a state comprising:

- *program_loc*, the memory address from which the instruction was fetched;
- *instruction_kind*, identifying whether this is a load, store, or barrier instruction, each with the associated kind; or a conditional branch; or a 'simple' instruction.
- *regs_in*, the set of input *reg_name*s, as statically determined;
- *regs_out*, the output *reg_name*s, as statically determined;
- *pseudocode_state* (or sometimes just 'state' for short), one of
  - PLAIN *next_state*, ready to make a pseudocode transition;
  - PENDING_MEM_READS *read_cont*, performing the read(s) from memory of a load; or
  - PENDING_MEM_WRITES *write_cont*, performing the write(s) to memory of a store;
- *reg_reads*, the accumulated register reads, including their sources and values, of this instance's execution so far;
- *reg_writes*, the accumulated register writes, including dependency information to identify the register reads and memory reads (by this instruction) that might have affected each;
- *mem_reads*, a set of memory read requests. Each request includes a memory footprint (an address and size) and, if the request has already been satisfied, the set of write slices (each consisting of a write and a set of its byte indices) that satisfied it.
- *mem_writes*, a set of memory write requests. Each request includes a memory footprint and, when available, the memory value to be written. In addition, each write has a flag that indicates whether the write has been propagated (passed to the memory) or not.

- [exclusive] *successful_exclusive*, for store-exclusives, indicates whether it was previously guaranteed to succeed or made to fail.
- information recording whether the instance is committed, finished, etc.

Read requests include their read kind and their memory footprint (their address and size), the as-yet-unsatisfied slices (the byte indices that have not been satisfied), and, for the satisfied slices, information about the write(s) that they were satisfied from. Write requests include their write kind, their memory footprint, and their value. When we refer to a write or read request without mentioning the kind of request we mean the request can be of any kind. A load instruction which has initiated (so its read request list *mem_reads* is not empty) and for which all its read requests are satisfied (i.e. there are no unsatisfied slices) is said to be *entirely satisfied*. A load-exclusive is called *successful* if the first po-following store-exclusive has been guaranteed to succeed (as opposed to does not exist or has not been guaranteed to succeed or made to fail). The successful load-exclusive and the successful store-exclusive are said to be *paired*. If a successful load-exclusive has a read request that is mapped, in the exclusives map, to a write slice *ws*, we say the load-exclusive has an outstanding lock on *ws*.

B.4 Thread States

The model state of a single hardware thread includes:

- *thread_id*, a unique identifier of the thread;
- *register_data*, the name, bit width, and start bit index for each register;
- *initial_register_state*, the initial register value for each register;
- *initial_fetch_address*, the initial fetch address for this thread;
- *instruction_tree*, a tree or list of the instruction instances that have been fetched (and not discarded), in program order.

B.5 Model Transitions

Fetch request For some instruction i, any possible next fetch address *loc* can be requested, adding it to the fetch queue, if:

1. it has not already been requested, i.e., none of the immediate successors of i in the thread's *instruction_tree* are from *loc*; and
2. either i is not decoded, or, if it has been, *loc* is a possible next fetch address for i:
   (a) for a non-branch/jump instruction, the successor instruction address (*i.program_loc+4*);
   (b) for a conditional branch, either the successor address or the branch target address[3]; or
   (c) for a jump to an address which is not yet determined, any address (this is approximated in our tool implementation, necessarily).

Note that this allows speculation past conditional branches and calculated jumps. Action: add an unfetched entry for *loc* to the fetch queue for i's thread.

Fetch instruction For any fetch-queue entry in the UNFETCHED state, its fetch can be satisfied from the memory and abstract data cache, from write-slices *ws*, if:

1. the write-slices (parts of writes) *ws* have the 4-byte footprint of the entry and can be constructed by composing some combination of the flat memory and a set of writes from the abstract data cache.

   Action: change the fetch-queue entry's state to FETCHED*(ws)*.

Fetch instruction (unpredictable) For any fetch-queue entry in the UNFETCHED state, its fetch can be satisfied from the memory and abstract data cache in a constrained-unpredictable way, if:

[3] In AArch64, all the conditional branch instructions have statically determined addresses.

1. there exists a set of sets of write-slices, each of which can be constructed in the same way as above;
2. that set contains multiple values, and at least one of those values corresponds to an instruction that is not `B.cond` or one of {`B`, `BL`, `BRK`, `HVC`, `SMC`, `SVC`, `ISB`, `NOP`}, and they are not all `B.cond` instructions.

Action: record the fetch-queue entry as CONSTRAINED_UNPREDICTABLE. When this has reached decode and the corresponding point in the instruction tree becomes non-speculative, the entire thread state will become CONSTRAINED_UNPREDICTABLE.

Fetch instruction (B.cond) For any fetch-queue entry in the UNFETCHED state, its fetch can be satisfied from the memory and abstract data cache, from write-slices ws and ws', with value ws'', if:

1. there exists write-slices ws and ws', each of which can be constructed in the same way as above;
2. ws and ws' correspond to the encoding of two conditional branch instructions b and b';
3. the write-slices ws'' can be constructed as the combination of ws and ws' such that ws'' is the encoding of the branch instruction with b's condition and b''s target.

Action: record the fetch-queue entry as FETCHED(ws'').

Decode instruction If the last entry in the fetch queue is in FETCHED(ws) state, it can be removed from the queue, decoded, and begin execution, if all po-previous `ISB` instructions in the instruction tree have finished. Action:

1. Construct a new instruction instance i with the correct instruction kind and state, for i's program location, and add it to the instruction tree.
2. Discard all speculative entries in the instruction tree that are successors of i that are now known to be incorrect speculations.

Initiate memory reads of load instruction An instruction instance i with next state

Read_mem*(read_kind, address, size, read_cont)* can initiate the corresponding memory reads. Action:

1. Construct the appropriate read requests *rrs*:
   - if *address* is aligned to *size* then *rrs* is a single read request of *size* bytes from *address*;
   - otherwise, *rrs* is a set of *size* read requests, each of one byte, from the addresses *address*...*address+size-1*.
2. set *i.mem_reads* to *rrs*; and
3. update the state of i to Pending_mem_reads *read_cont*.

Satisfy memory read by forwarding from writes For a load instruction instance i in state Pending_mem_reads *read_cont*, and a read request, r in *i.mem_reads* that has unsatisfied slices, the read request can be partially or entirely satisfied by forwarding from unpropagated writes by store instruction instances that are po-before i, if the *read-request-condition* predicate holds. This is if:

1. [ifetch] all po-previous `dsb sy` instructions are finished;
2. all po-previous `dmb sy` and `isb` instructions are finished;
3. [dmb ld/ dmb st] all po-previous `dmb ld` instructions are finished;
4. [release/ acquire] if i is a load-acquire, all po-previous store-releases are finished; and
5. [release/ acquire] all non-finished po-previous load-acquire instructions are entirely satisfied.

Let *wss* be the maximal set of unpropagated write slices from store instruction instances po-before i (if i is a load-acquire, exclude store-exclusive writes), that overlap with the unsatisfied slices of r, and which are not superseded by intervening stores that are either propagated or read from by this thread. That last condition requires, for each write slice *ws* in *wss* from instruction i':

- that there is no store instruction po-between i and i' with a write overlapping *ws*, and

– that there is no load instruction po-between i and i' that was satisfied from an overlapping write slice from a different thread.

Action:

1. update r to indicate that it was satisfied by wss; and
2. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' that is a po-successor of i, and every read request r' of i' that was satisfied from wss', if there exists a write slice ws' in wss', and an overlapping write slice from a different write in wss, and ws' is not from an instruction that is a po-successor of i, or if i' was a data-cache maintenance by virtual address to a cache line that overlaps with any of the write slices in wss', restart i' and its data-flow dependents (including po-successors of load-acquire instructions).

Note that store-release writes cannot be forwarded to load-acquires: a load-acquire instruction cannot be satisfied before all po-previous store-release instructions are finished, and wss does not include writes from finished stores (as those must be propagated).

Satisfy memory read from memory For a load instruction instance i in state PENDING_MEM_READS *read_cont*, and a read request r in *i.mem_reads*, that has unsatisfied slices, the read request can be satisfied from memory if i is not a successful load-exclusive or no other successful load-exclusive from a different thread has an outstanding lock on the writes r is trying to read from.

If: the read-request-condition holds (see previous transition).

Action: let wss be the write slices from memory or the data cache network, whichever is newer, covering the unsatisfied slices of r, and apply the action of Satisfy memory read by forwarding from writes. In addition, if i is a successful load-exclusive, union wss with the set of write slices r is mapped to in the exclusives map.

Note that Satisfy memory read by forwarding from writes might leave some slices of the read request unsatisfied. Satisfy memory read from memory, on the other hand, will always satisfy all the unsatisfied slices of the read request.

Complete load instruction (when all its reads are entirely satisfied) A load instruction instance i in state PENDING_MEM_READS *read_cont* can be completed (not to be confused with finished) if all the read requests $i.mem\_reads$ are entirely satisfied (i.e., there are no unsatisfied slices).

Action: update the state of i to PLAIN *(read_cont (memory_value))*, where *memory_value* is assembled from all the write slices that satisfied $i.mem\_reads$.

Guarantee the success of store-exclusive A store-exclusive instruction instance i with next state EXCL_RES*(res_cont)* can be guaranteed to succeed if:

1. the store-exclusive has not been made to fail (as recorded in $i.successful\_exclusive$);
2. assuming i is successful, it can be paired with a load-exclusive i' (see §B.3); and
3. if i' has already been satisfied (not necessarily entirely), let *wss* be the set of propagated write slices i' has read from, then, no slice in *wss* has been overwritten (in memory) by a write from another thread, and no other successful load-exclusive from a different thread has an outstanding lock on a write slice from *wss*.

Action:

1. record in $i.successful\_exclusive$ that the store-exclusive will be successful;
2. if i' has already been satisfied, union *wss* with the set of write slices the read request of i' is mapped to in the exclusives map, where *wss* is as above; and
3. update the state of i to PLAIN *(res_cont (true))*.

Make a store-exclusive fail A store-exclusive instruction instance i with next state EXCL_RES*(res_continuation)* can be made to fail if the store-exclusive has not been guaranteed to succeed (as recorded in *i.successful_ exclusive*) Action:

1. record in *i.successful_ exclusive* that the store-exclusive was made to fail; and
2. update the state of i to PLAIN *(res_ cont (false))*.

Note the promise-success transition is enabled before the store-exclusive commits, and we do not require it to have a fully-determined address or to be non-restartable. As a result, a store-exclusive that has already promised its success might be restarted. Since other instructions may rely on its promise, the restart will not affect the value of *i.successful_ exclusive*. Instead, when the store-exclusive is restarted it will take the same promise/failure transition as before its restart — based on the value of *i.successful_ exclusive*.

Initiate memory writes of store instruction, with their footprints An instruction instance i with next state WRITE_ EA*(write_ kind, address, size, next_ state′)* can announce its pending write footprint. Action:

1. construct the appropriate write requests:
   - if *address* is aligned to *size* then *ws* is a single write request of *size* bytes to *address*;
   - otherwise *ws* is a set of *size* write requests, each of one byte size, to the addresses *address*. . .*address+size-1*.
2. set *i.mem_ writes* to *ws*; and
3. update the state of i to PLAIN *next_ state′*.

Note that at this point the write requests do not yet have their values. This state allows non-overlapping po-following writes to propagate.

Instantiate memory write values of store instruction An instruction instance i with next state WRITE_ MEMV*(memory_ value, write_ cont)* can initiate the corresponding memory writes. Action:

1. split *memory_ value* between the write requests *i.mem_ writes*; and
2. update the state of i to PENDING_ MEM_ WRITES *write_ cont*.

Commit store instruction For an uncommitted store instruction i in state PENDING_MEM_WRITES *write_cont*, i can commit if:

1. i has fully determined data (i.e., the register reads cannot change, see §B.6);
2. all po-previous conditional branch instructions are finished;
3. all po-previous `dmb sy` and `isb` instructions are finished;
4. [ifetch] all po-previous `dsb sy` instructions are finished;
5. [dmb ld/ dmb st] all po-previous `dmb ld` instructions are finished;
6. [release/ acquire] all po-previous load-acquire instructions are finished;
7. all po-previous store instructions, except for store-exclusives that failed, have initiated and so have non-empty *mem_writes*;
8. [release/ acquire] if i is a store-release, all po-previous memory access instructions are finished;
9. [dmb ld/ dmb st] all po-previous `dmb st` instructions are finished;
10. all po-previous memory access instructions have a fully determined memory footprint; and
11. all po-previous load instructions have initiated and so have non-empty *mem_reads*.

Action: record i as committed.

Propagate memory write For an instruction i in state PENDING_MEM_WRITES *write_cont*, and an unpropagated write, w in *i.mem_writes*, the write can be propagated if:

1. all memory writes of po-previous store instructions that overlap w have already propagated
2. all read requests of po-previous load instructions that overlap with w have already been satisfied, and the load instruction is non-restartable (see §B.6);
3. all read requests satisfied by forwarding w are entirely satisfied; and

4. [exclusive] no successful load-exclusive from a different thread has an outstanding lock on a write slice that overlaps with w.

Action:

1. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' po-after i and every read request r' of i' that was satisfied from wss', if there exists a write slice ws' in wss' that overlaps with w and is not from w, and ws' is not from a po-successor of i, or if i' is a data-cache maintenance instruction to a cache line whose footprint overlaps with w, restart i' and its data-flow dependents;
2. record w as propagated;
3. add w as a complete slice to the data cache network.

Complete store instruction (when its writes are all propagated) A store instruction i in state PENDING_MEM_WRITES *write_cont*, for which all the memory writes in *i.mem_writes* have been propagated, can be completed. Action: update the state of i to PLAIN*(write_cont(true))*.

Commit barrier A barrier instruction i in state PLAIN *next_state* where *next_state* is
BARRIER*(barrier_kind, next_state')* can be committed if:

1. all po-previous conditional branch instructions are finished;
2. [dmb ld/ dmb st] if i is a `dmb ld` instruction, all po-previous load instructions are finished;
3. [dmb ld/ dmb st] if i is a `dmb st` instruction, all po-previous store instructions are finished;
4. all po-previous `dmb sy` barriers are finished;
5. [ifetch] all po-previous `dsb sy` barriers are finished;
6. if i is an `isb` instruction, all po-previous memory access instructions have fully determined memory footprints; and
7. if i is a `dmb sy` instruction, all po-previous memory access instructions and barriers are finished;

8. [ifetch] if *i* is a `dsb sy` instruction, all po-previous memory access instructions, barriers and cache maintenance instructions have finished.

Note that this differs from the previous Flowing and POP models: there, barriers committed in program-order and potentially re-ordered in the storage subsystem. Here the thread subsystem is weakened to subsume the reordering of Flowing's (and POP's) storage subsystem.

Action:

1. update the state of *i* to Plain *next_state′*;
2. [ifetch] if *i* is an `isb` instruction, for all threads instruction tree's, for any instruction instance *i* in the Fetched state, set it to the Unfetched state.

Begin IC An instruction *i* (with unique instruction instance ID *iiid*) in state Perform_IC*(address, state_cont)* can begin performing the `IC` behaviour if all po-previous `DSB ISH` instructions have finished. Action:

1. For each thread *tid'* (including this one), add *(iiid, address)* to that thread's *ic_writes*;
2. Set the state of *i* to Propagate_IC*(address, state_cont)*.

Propagate IC to thread An instruction *i* (with ID *iiid*) in state Wait_IC*(address, state_cont)* can do the relevant invalidate for any thread *tid'*, modifying that thread's instruction cache and fetch queue, if there exists a pending entry *(iiid, address)* in that thread's *ic_writes*. Action:

1. for any entry in the fetch queue for thread *tid*, whose *program_loc* is in the same minimum-size instruction cache line as *address*, and is in Fetched(_) state, set it to the Unfetched state;
2. for the instruction cache of thread *tid*, remove any write-slices which are in the same instruction cache line of minimum size as *address*.

Complete IC An instruction i (with ID *iiid*) in the state WAIT_IC*(address, state_cont)* can complete if there exists no entry for *iiid* in any thread's *ic_writes*. Action: set the state of i to PLAIN*(state_cont)*.

Perform DC An instruction i in the state PERFORM_DC*(address, state_cont)* can complete if all po-previous `DMB ISH` and `DSB ISH` instructions have finished. Action:

1. For the most recent write slices *wss* which are in the same data cache line of minimum size in the abstract data cache as *address*, update the memory with *wss*;
2. Remove all those writes from the abstract data cache.
3. Set the state of i to PLAIN*(state_cont)*.

Add to instruction cache for thread A thread *tid*'s instruction cache can become spontaneously updated with a write w from the storage subsystem, if this write (as a complete slice) does not already exist in the instruction cache. Action: Add this write (as a complete slice) to thread *tid*'s instruction cache.

Register read An instruction instance i with next state READ_REG*(reg_name, read_cont)* can do a register read if every instruction instance that it needs to read from has already performed the expected register write.

Let *read_sources* include, for each bit of *reg_name*, the write to that bit by the most recent (in program order) instruction instance that can write to that bit, if any. If there is no such instruction, the source is the initial register value from *initial_register_state*. Let *register_value* be the assembled value from *read_sources*. Action:

1. add *reg_name* to *i.reg_reads* with *read_sources* and *register_value*; and
2. update the state of i to PLAIN *(read_cont(register_value))*.

Register write An instruction instance i with next state WRITE_REG(*reg_name*, *register_value*, *next_state′*) can do the register write. Action:

1. add *reg_name* to *i.reg_writes* with *write_deps* and *register_value*; and
2. update the state of i to PLAIN *next_state′*.

where *write_deps* is the set of all *read_sources* from *i.reg_reads* and a flag that is set to true if i is a load instruction that has already been entirely satisfied.

Pseudocode internal step An instruction instance i with next state INTERNAL(*next_state′*) can do that pseudocode-internal step. Action: update the state of i to PLAIN *next_state′*.

Finish instruction A non-finished instruction i with next state DONE can be finished if:

1. if i is a load instruction:
   (a) all po-previous `dmb sy` and `isb` instructions are finished;
   (b) [`dmb ld/` `dmb st`] all po-previous `dmb ld` instructions are finished;
   (c) [release/ acquire] all po-previous load-acquire instructions are finished;
   (d) it is guaranteed that the values read by the read requests of i will not cause coherence violations, i.e., for any po-previous instruction instance i', let *cfp* be the combined footprint of propagated writes from store instructions po-between i and i' and fixed writes that were forwarded to i from store instructions po-between i and i' including i', and let *cfp′* be the complement of *cfp* in the memory footprint of i. If *cfp′* is not empty:
      i. i' has a fully determined memory footprint;
      ii. i' has no unpropagated memory write that overlaps with *cfp′*; and
      iii. If i' is a load with a memory footprint that overlaps with *cfp′*, then all the read requests of i' that overlap with *cfp′* are satisfied and i' can not be restarted (see §B.6).

   Here a memory write is called fixed if it is the write of a store instruction that has fully determined data.

   (e) [release/acquire] if *i* is a load-acquire, all po-previous store-release instructions are finished;

2. *i* has fully determined data; and
3. all po-previous conditional branches are finished.

Action:

1. if *i* is a branch instruction, discard any untaken path of execution, i.e., remove any (non-finished) instructions that are not reachable by the branch taken in *instruction_tree*; and
2. record the instruction as finished, i.e., set *finished* to *true*.

B.6 Auxiliary Definitions

Fully determined An instruction is said to have fully determined footprint if the memory reads feeding into its footprint are finished: A register write *w*, of instruction *i*, with the associated *write_deps* from *i.reg_writes* is said to be *fully determined* if one of the following conditions hold:

1. *i* is finished; or
2. the load flag in *write_deps* is *false* and every register write in *write_deps* is fully determined.

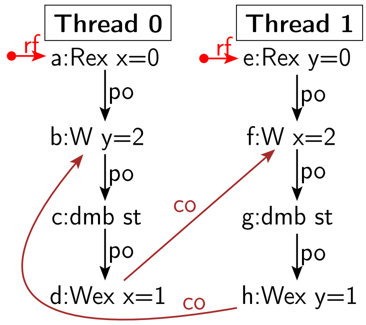
An instruction *i* is said to have *fully determined data* if all the register writes of *read_sources* in *i.reg_reads* are fully determined. An instruction *i* is said to have a *fully determined memory footprint* if all the register writes of *read_sources* in *i.reg_reads* that are associated with registers that feed into *i*'s memory access footprint are fully determined.

Restart condition To determine if instruction *i* might be restarted we use the following recursive condition: *i* is a non-finished instruction and at least one of the following holds,

1. there exists an unpropagated write w such that applying the action of the Propagate memory write transition to s will result in the restart of i;
2. there exists a non-finished load instruction l such that applying the action of the Satisfy memory read from memory transition to l will result in the restart of i (even if l is already entirely satisfied); or
3. there exists a non-finished instruction i' that might be restarted and i is in its data-flow dependents (including po-successors of load-acquire instructions).

Cache Line of Minimum Size Cache maintenance operations work over entire cache lines, not individual addresses. Each address is associated with at least one cache line for the data (and unified) caches, and one for the instruction caches. The cache line of minimum size is the smallest possible cache line for each of these. The `CTR_EL0.{DMinLine, IMinLine}` values describe the cache lines of minimum size for the data and instruction caches as $\log_2$ of the number of words in the cache line.

Remarks about load/store exclusive instructions The MCA ARMv8 architecture intends that the success bit of store exclusives does not introduce dependencies, to allow (e.g.) hardware optimisations that dynamically replace load/store exclusive pairs by atomic read-modify-write operations that can execute in the memory subsystem and therefore be guaranteed to succeed. The ARMv8-axiomatic definition assumes all address/data/control dependencies to be from reads, not writes. In the operational model, matching this weakness has proved to be difficult: it means the operational model must be able to promise the success or failure of a store-exclusive instruction even before any of its registers reads/writes have been done, so before the store-exclusive's address and data are available. The early success promises are the source of deadlocks in the operational model. To illustrate this consider, for example, the following litmus test and a state where both `a` and `e` are satisfied and finished, and where `b` and `f` are not propagated. Then `d` can promise its success, locking memory location `x`, and `h` can promise its success, locking location `y`. But now there is a deadlock:



- For d to propagate c has to be committed and hence b propagated. But b cannot propagate since y is locked.
- For h to propagate g has to be committed and hence f propagated. But f cannot propagate since x is locked.

Similar situations arise from cases where there are other barriers or release/acquire instructions in-between the load and the store exclusive, or if the store exclusive has additional dependencies that the load exclusive does not have. These are cases that are not really intended to be supported by the architecture.

The model can also currently deadlock if a load and a store-exclusive are paired successfully but later turn out to have different addresses: if the store-exclusive promises its success before its address is known it locks the matched load-exclusive's memory location; when they later turns out to be to a different addresses it never unlocks it. This issue can be fixed, but it is currently still being clarified what exactly the architecturally allowed behaviour should be.

C Additional explanation of the iFlat axiomatic model

We make two semantics-preserving rewrites of the existing model before we add instruction fetches:

- The existing axiomatic model expresses the semantics of various barriers with composite edges: e.g. `[R|W];po;[dmb.sy];po;[R|W]` ("a `DMB SY` barrier creates order from program-order-preceding read or write events to program-order-succeeding read or write events"). We replace such edges with more "fine grained" edges, e.g. splitting the previous edge into the two edges `[R|W];po;[dmb.sy]` and `[dmb.sy];po;[R|W]`, which informally require a barrier to "wait" for preceding reads and writes, and succeeding reads and writes to wait for the barrier. Ignoring instruction fetches, this rewrite does not change the semantics. In the extended model, however, it allows capturing the interaction between the barrier ordering and instruction fetches: e.g. assume some edge (e, i) imposing an ordering constraint on the fetch i of a `dmb.sy` event b, and some read event r program-order-after the `DMB SY`. Then we want r to also be ordered after e. Splitting the edges as above means there is an edge (b, r), and we can specify the rules so (e, i) composes with (b, r).
- The existing ARMv8-A model includes the relations `fre` and `coe` into the definition of the *observed-by* relation, but for presentation not the relation `fri` and `coi`. Including these is equivalent and allows simplifying the definitions: it allows deleting the edges such as "`(ctrl|data);coi`" which are now subsumed by the combination of existing edges with `coi` or `fri` in `ob`.

We now make the following changes and additions to the model. The full model is shown in Figure 2 (exactly as in the main body), with comments referring to the items in the following explanation.

1. We define the relation `iseq`, relating some write w to x to an `IC` completing a cache synchronisation sequence (not necessarily on a single thread): w is `wco`-followed by a same-cache-line `DC`, in turn `wco`-followed by the

```
let iseq = [W];(wco&scl);[DC]; (*1*)
                (wco&scl);[IC]
(* Observed-by *)
let obs = rfe | fr | wco       (*2*)
        | irf | (ifr;iseq)     (*3,4*)
(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]      (*5*)
  | [IF]; fe                   (*6*)
  | [ISB]; fe^{-1}; fpo        (*7*)
(* Dependency-ordered-before *)
let dob = addr | data
  | ctrl; [W]
  | (ctrl | (addr; po)); [ISB]
(*| [ISB]; po; [R] *)          (*8*)
  | addr; po; [W]
  | (addr | data); rfi
(* Atomic-ordered-before *)
let aob = rmw
  | [range(rmw)]; rfi; [A|Q]
(* Barrier-ordered-before *)
let bob = [R|W]; po; [dmb.sy]
  | [dmb.sy]; po; [R|W]
  | [L]; po; [A]
  | [R]; po; [dmb.ld]
  | [dmb.ld]; po; [R|W]
  | [A|Q]; po; [R|W]
  | [W]; po; [dmb.st]
  | [dmb.st]; po; [W]
  | [R|W]; po; [L]
  | [R|W|F|DC|IC]; po; [dsb.ish]  (*9*)
  | [dsb.ish]; po; [R|W|F|DC|IC]  (*10*)
  | [dmb.sy]; po; [DC]            (*11*)
(* Cache-op-ordered-before *)
let cob = [R|W]; (po&scl); [DC]   (*12*)
  | [DC]; (po&scl); [DC]          (*13*)
(* Ordered-before *)
let ob = (obs|fob|dob|aob|bob|cob)+
(* Internal visibility requirement *)
acyclic (po-loc|fr|co|rf) as internal
(* External visibility requirement *)
irreflexive ob as external
(* Atomic *)
empty rmw & (fre; coe) as atomic
(* Constrained unpredictable *)
let cff = ([W];loc;[IF]) \        (*14*)
          ob^{-1} \ (co;iseq;ob)
cff_bad cff ≡ CU                  (*15*)
```

Fig. 2. Axiomatic model

same-cache-line `IC`. In operational model terms, this captures traces that propagated w to memory, subsequently performed a same-cache-line `DC`, and then a began an `IC` (and eagerly propagated the `IC` to all threads). In any state after this sequence it is guaranteed that w, or a coherence-newer same-address write, is in the instruction cache of all threads: performing the `DC` has cleared the abstract data cache of writes to x, and the subsequent `IC` has removed old instructions for location x from the instruction caches, so that any subsequent updates to the instruction caches have been with w, or `co`-newer writes.

2. After the aforementioned two rewrites, the model includes `co` in `obs`; we instead include the relation `wco`. Including `wco` in *ordered-before* corresponds to the intuition that `wco` records the ordering of the Write Propagate, Perform data cache maintenance, and Perform instruction cache maintenance transitions in a matching trace.
3. We also include `irf` in `obs`: informally, for an instruction to be fetched from a write, the write has to have been done before. Correspondingly, in the operational model, a write has to have been propagated before it can satisfy fetches in the storage subsystem.
4. We add to the *observed-by* relation the edge `ifr;iseq`, relating an instruction fetch i to x to an `IC` ic if: i fetched from a write w to x, some write w' to x is coherence-after w, and ic completes a cache synchronisation sequence (`iseq`) starting from w'. Then i must be ordered-before ic, because if it happened "after" ic, the cache synchronisation sequence would force i to read from w' or a coherence-newer write.

   This corresponds to the operational model in the following way: assume a trace where w was propagated, before w' was propagated, and before the model took a sequence of transitions containing a cache-synchronisation sequence on x's cache-line. If the fetch transition i were to satisfy its fetch in a subsequent state, it would be guaranteed that w' (or a coherence-newer write) would be in the instruction cache, and i would not be able to fetch from w. Hence, i must have happened before the `IC` completing the cache synchronisation sequence.

5. We add a relation *fetch-ordered-before* (`fob`), which is included in *ordered-before*. The relation `fob` includes `fpo`, informally requiring fetches to be ordered according to their order in the control-flow unfolding of the execution. Or correspondingly in the operational model: fetch requests for instructions within the same thread appear to be satisfied in program order.
6. `fob` also includes the `fe` *fetch-to-execute* relation, capturing the idea that an instruction must be fetched before it can execute. In the operational model, a read can only satisfy/a write can only propagate/a barrier can only commit/etc. after its instruction's fetch is satisfied.
7. More interestingly, `fob` includes the edge `[ISB];fe`$^{-1}$`;fpo`, ordering the fetch of any instruction program-order-succeeding an `ISB` instruction after the `ISB` event. In the operational model, a decoded `ISB` instruction prevents any program-order-later instructions from being removed from the fetch queue and decoded, and when an `ISB` is executed, it returns all entries in this thread's fetch queue (so any program-order-later instructions) to "un-fetched state".
8. The rule `[ISB];po;[R]` in `dob` is no longer needed as the combination of rules `[ISB];fe`$^{-1}$`;fpo`, and `[IF];fe` subsume it.
9. For `DSB ISH` instructions the edge `[R|W|F|DC|IC];po;[dsb.ish]` is included: `DSB ISH`s are ordered with all program-order-preceding non-fetch events. Correspondingly, in the operational model a `DSB ISH` instruction can only commit once all preceding loads/stores/barriers/`DC` or `IC` instructions are finished.
10. Symmetrically, all non-`IF` events are ordered after program-order-preceding `dsb.ish` events. In the operational model, instructions can only execute after all preceding `DSB ISH` instructions are finished.
11. `bob` also includes the edge `[dmb.sy];po;[DC]`, ordering `DC` events after program-order-preceding `DMB SY`s. Correspondingly, in the operational model, a `DC` can only be performed when all preceding `DMB SY` are finished.

12. We include the relation *cache-op-ordered-before* (`cob`) in `ob`. This relation contains the edge `[R|W];(po&scl);[DC]`, ordering `DC` events after program-order-preceding same-cache-line read and write events.
    Operationally, a `DC` will be restarted by a program-order-preceding same-cache-line load if it was performed before the load was satisfied, and by a program-order-preceding same-cache-line store if it was performed before the store propagated its write.
13. Moreover, `cob` contains the edge `[DC];(po&scl);[DC]`, ordering two same-cache-line, same-thread `DC` events in program-order. In the operational model, a `DC` can only be performed when program-order-preceding same-cache-line `DC` instructions have been performed.
14. We define the relation *could-fetch-from* (`cff`), capturing, for each fetch i, the writes it could have fetched from (including the one it did fetch from), as the set of same-address writes that are not ordered-after i, and which are not overwritten by coherence-newer writes that were followed by a cachesync sequence ordered-before i. Operationally, this captures writes that could have been in the instruction cache of i's thread: writes that did not happen *after* i in the trace, and excluding writes cleared by earlier cache synchronisation sequences.
15. Then finally, the *constrained unpredictable* axiom requires that the candidate execution's `CU` boolean is 'true' if-and-only-if the predicate `cff_bad` indicates "a bad execution": an execution corresponding to a trace in which some instruction fetch could have fetched from multiple writes, at least one of which writes an instruction that architecturally is not "concurrently modifiable". This predicate is defined more naturally in first-order logic than herd's cat language: $\texttt{cff\_bad}\ \texttt{cff} = \exists i \in \texttt{IF}.\ |\{w|(w,i) \in \texttt{cff}\}| > 1 \wedge \exists w.(w,i) \in \texttt{cff} \wedge \neg\mathsf{concurrently\text{-}modifiable}\ (\mathsf{val}\ w)$.