Armed Cats: formal concurrency modelling at Arm

Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, Luc Maranget

Armed cats: formal concurrency modelling at Arm

JADE ALGLAVE, Arm Ltd and University College London
WILL DEACON*, Arm Ltd
RICHARD GRISENTHWAITE, Arm Ltd
ANTOINE HACQUARD, EPITA Research and Development Laboratory
LUC MARANGET, INRIA

We report on the process for formal concurrency modelling at Arm. An initial formal consistency model of the Arm achitecture, written in the cat language, was published and upstreamed to the herd+diy tool suite in 2017. Since then, we have extended the original model with extra features, for example mixed-size accesses, and produced two provably equivalent alternative formulations.

In this paper, we present a comprehensive review of work done at Arm on the consistency model. Along the way, we also show that our principle for handling mixed-size accesses applies to x86: we confirm this via vast experimental campaigns. We also show that our alternative formulations are applicable to any model phrased in a style similar to the one chosen by Arm.

ACM Reference Format:
Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed cats: formal concurrency modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 8 (July 2021), 57 pages. https://doi.org/10.1145/3458926

1 INTRODUCTION

Arm has invested in formal modelling of the concurrency aspects of its architecture, which led to the publication of a formal model in 2017 [18]. This model has since been maintained and enhanced in various ways, for example by adding features such as mixed-size accesses [3]. The importance and necessity of that mixed-size extension can be seen in the fact that without it, Linux's lockref structure cannot be used soundly on Arm machines, as we detail in this paper.

Arm has also developed and released two alternative formulations [30]. All three models are written in the cat language [14], a domain-specific language for writing formal consistency models in a concise and executable manner. The original model [18] is written in idiomatic cat, as a set of constraints over relations, which essentially state the existence of a partial order amongst memory accesses. That model's formulation takes a whole-system view of program executions. This is not an ideal view when designing a single core, because a hardware designer or verification engineer would have to work out what those whole-system constraints mean per thread.

This is where the alternative definitions come into play: they allow hardware designers and verification engineers to reason about the validity of an execution from the point of view of one thread, not the whole system. This allows for designs which are more easily built, and checked for soundness.

*Now at Google

Authors' addresses: Jade Alglave, Arm Ltd and University College London, jade.alglave@arm.com, j.alglave@ucl.ac.uk; Will Deacon, Arm Ltd, will@kernel.org; Richard Grisenthwaite, Arm Ltd, richard.grisenthwaite@arm.com; Antoine Hacquard, EPITA Research and Development Laboratory, ahacquard@lrde.epita.fr; Luc Maranget, INRIA, luc.maranget@inria.fr.

Additionally, those alternative formulations carry more hardware intuition: their cornerstone is the notion of store forwarding, or in other words, when a local read can take a value early from a local write, before the rest of the system is aware of the value written by that local write. On the other hand, those alternative formulations are less easily related to the specification style used by Linux [10] or C++ [27].

This need for alternative formulations with different purposes and appeals echoes the following remark made by Hoare and Lauer [22]:

> *a single formal definition is unlikely to be equally acceptable to both implementor and user, and [...] at least two definitions are required, a constructive one [...] for the implementor, and an implicit one for the user.*

We go slightly further in that we propose three alternative formulations: the original one appears better suited to programmers, i.e. users; the second one seems better suited to hardware designers, i.e. implementors, and the third one seems better suited to current verification practices.

We have done equivalence proofs between all three formulations; Arm intends to maintain those equivalences when its concurrency model evolves with new features.

1.1 Outline of the paper

The purpose of this paper is three-fold:

(1) to present this new technical material, viz, the two alternative formulations of the original model, and the mixed-size extensions for all three models, by giving the rationale and details of the models, their mixed-size extensions, and accompanying equivalence proofs;
(2) to give the overarching principles applied by Arm to the design of its formal concurrency model, by listing the criteria that Arm tries to follow and ensure its formal concurrency model respects;
(3) to contribute to the growing chorus of formally minded industrialists, by publicising the fact that Arm is investing in formal methods, and as a consequence is distributing its formal concurrency models and accompanying equivalence proofs.

The outline of this paper is as follows. We review related works in Section 3. In Section 4, we summarise the ideas behind the cat language and the herd tool, using the original Armv8 model [18] as a running example. In Section 5, we detail how we extend the model to handle mixed-size accesses. More precisely, we propose a principle to extend certain existing concurrency models so that they apply to mixed-size accesses, and confirm that our principle applies to x86 and Arm, in ways that we detail in that section. In Section 6, we detail the alternative formulations of Armv8's model, including mixed-size accesses. Adding mixed-size accesses to those formulations was challenging due to the design principles we needed to follow as they were established practice at Arm; we detail those challenges in that section. In Section 7, we sketch the equivalence proofs between those models.

But first we give the list of the material that we distribute, and where to find it. We then give a preamble on our design principles and the rationale behind Arm's two alternative formulations of its consistency model.

1.2 Additional material and where to find it

The work presented in this paper resulted in the development of the following material, which we distribute online:

- the official extension of the Armv8 model to handle mixed-size accesses, which can be found at `https://github.com/herd/herdtools7/blob/95785c747750be4a3b64adfab9d5f5ee0ead8240/herd/libdir/aarch64.cat`;

- two alternative formulations of the Armv8 original model, which are also maintained by Arm; they can be found at `https://github.com/herd/herdtools7/blob/master/herd/libdir/arm-models`;
- an extension of the x86 model to handle mixed-size accesses, which we validated experimentally; it can be found at `https://github.com/herd/herdtools7/blob/master/herd/libdir/x86tso-mixed.cat`;
- the extension to all the tools in the herd+diy toolsuite to handle mixed-size accesses, which can be found at `http://diy.inria.fr` and `https://github.com/herd/herdtools7`;
- our experimental results for mixed-size tests on both x86 and Arm hardware, which can be found at `http://diy.inria.fr/mixed`;
- our experimental results for non-mixed-sized Arm tests, which can be found at `http://diy.inria.fr/aarch64`.

2 DESIGN PRINCIPLES AND RATIONALE

Modelling the concurrency aspects of the Armv8 architecture entails developing a consistency model for Armv8. Consistency models determine what values a read can take; weak consistency models such as the ones of Arm [18, 29], IBM [37, 38], Intel [39, 40], Nvidia [5, 32], RISC-V [36], C++ [15, 27], Linux [10] and others allow more behaviours than Sequential Consistency (SC) [28].

The original Armv8 consistency model [18] was developed following a number of design principles, which we list here. We followed those same principles whenever we have extended the model, and some arose during this extension work; we also list them below. We intend to follow them in future developments.

The Armv8 consistency model is written in the cat language [14]. The cat language is a domain-specific language dedicated to describing consistency models in concise, formal and executable ways. The language drives a collection of open-source tools [9]:

- the herd7 tool [14], given a cat model and a small program called a litmus test, outputs all possible executions (up to a certain bound on loop unrolling if necessary) of that program under that model;
- the diy7 tool [11, 13] generates systematic families of litmus tests which are designed to highlight the discrepancies between weak consistency models and Sequential Consistency;
- the litmus7 tool [12] runs such litmus tests on native hardware and collects the observed outcomes.

2.1 Open, formal and executable semantics

The remark about these tools being open-source is not accidental: we aim for the semantics of the cat language, and hence of the various models written in cat, to be open, and for contributors to be able to investigate the code implementing them. Thus the Armv8 model is distributed within the herd+diy tool suite [9]. The fact that the Armv8 model is written in cat makes it:

- formal, since the cat language has a formal semantics [7];
- machine-readable and more importantly executable, thanks to the herd7 tool [14].

If extensions to the cat language are required, the semantics of those extensions should be added to the existing ones [7], and an executable version of those semantics implemented in the herd7 tool.

2.2 Soundness with respect to hardware

The Armv8 model and the extensions that we present in this paper have been extensively tested thanks to the diy7 [11, 13] and litmus7 tools [12]. Soundness with respect to hardware is mandatory, modulo bugs that have been acknowledged as such in the hardware design.

2.3 Incrementality of definitions

The model is built from definitions (detailed in Section 4.4.2) which can be read independently, and combine to form a notion called Locally-ordered-before, which contributes to one of the main axioms of the model. Intuitively this notion captures all possible instructions pairs which must not be reordered according to the architecture. Put another way, two instructions in program order which are not included in Locally-ordered-before can be reordered: this is where there is opportunity for hardware optimisations, and conversely where programmers need to place synchronisation if those reorderings are undesirable.

Of course building definitions incrementally is useful to explain the model step-by-step and avoid mutual recursion for example.

2.4 Per-thread reasoning

But the fact that we *can* build the axioms of the model in such a way, by combining independent definitions, is not an accident: it stems from a design choice to make the model *multi-copy-atomic* (often abbreviated as "MCA", and called "Other-multi-copy-atomic" in the Arm documentation). Other-muti-copy-atomicity is defined as follows in the Arm documentation—note that the Arm documentation uses the term "Observer" to designate what we call processor or thread in this paper: :

> *In an Other-multi-copy atomic system, it is required that a write from an Observer, if observed by a different Observer, is then observed by all other Observers that access the Location coherently. It is, however, permitted for an Observer to observe its own writes prior to making them visible to other observers in the system.*

This term essentially means that one can reason locally about which synchronisation to use, instead of having to analyse the entirety of a multi-threaded program, including the interaction between its threads. To make this principle less abstract, consider the two tests given in Figure 1.

The first test MP is a message passing example: thread P0 updates the data x and sets up flag y to signal that the new data is ready to thread P1. The question asked by the test is whether there is a scenario under which the reading thread P1 can see the new flag but not the new data. The second test WRC is a distributed version of the same test, extended over three threads: now P0 only updates the data x, which is observed by P1, and P1 is in charge of updating the flag to signal the readiness of the new data to the reading thread P2.

In the scenario described by the exists clause of the MP litmus test, the read c of flag y by P1 takes its value from the write b of flag y by P0. The read d of the data x by P1 takes its value from the initial state. Finally, the write a of the data x on P0 overwrites the value read by the read d.

In the execution of WRC, the scenario is very similar; the only difference being that the write a of the data x is not on the same thread as the write d of the flag y. Instead a first thread (header "Thread 0") updates x to 1, and this update is then read by Thread 1. The rest of the execution is as in the MP scenario.

Our locality principle ensures that from a programming point of view, one can both synchronise MP and WRC by reasoning about synchronisation on each thread and not across the whole program. This is not the case on all architectures, as for example WRC requires an extra notion called "cumulativity" to be synchronised on IBM Power [37, 38].

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

AArch64 WRC
{0:X1=x; 1:X1=x; 1:X3=y; 2:X1=y; 2:X3=x;}
 P0          | P1          | P2          ;
 MOV W0,#1   | LDR W0,[X1] | LDR W0,[X1] ;
 STR W0,[X1] | MOV W2,#1   | LDR W2,[X3] ;
             | STR W2,[X3] |             ;
exists (1:X0=1 /\ 2:X0=1 /\ 2:X2=0)
```

Fig. 1. Two litmus tests to illustrate locality

The reason for this is as follows: due to MCA, in the WRC example, when Thread 1 sees the write of x, that write is also made visible to Thread 2. Therefore the only synchronisation required to forbid the behaviour of WRC is to preserve the order in which instructions are written on each thread. This can be achieved using *dependencies* on Thread 1 and Thread 2, for example address dependencies. Without MCA, forbidding WRC requires to ensure visibility of the write of x by Thread 0 to both Thread 1 and Thread 2. Dependencies will not provide cumulativity, therefore a way to forbid WRC's behaviour in a non-MCA system would be to use a cumulative fence, for example IBM Power's `lwsync`.

From a hardware implementation point of view, MCA requires more constrained mechanisms to ensure that `WRC` does not exhibit additional behaviours. Whilst this might be an extra burden on the hardware, this has the benefit of enabling a somewhat easier consistency model.

As a corollary of this principle, the alternative formulations too must stay local. Formally, those alternative formulations differ from the original one in that the constraints are now stated over a total order over memory events, as opposed to a partial order in the original formulation. By "staying local" in the context of the alternative formulations, we mean that the total order prescribed by those models must remain a linearisation of the Locally-ordered-before relation, or an extension thereof, as long as that extension can be determined by analysing a single thread.

2.5 Higher levels of the stack must be upheld

The model was developed in the light of ordering guarantees necessary to languages such as C++ or operating systems such as Linux. Overall the architectural intent behind the Armv8 model is to be as weak as possible, whilst upholding the guarantees needed by higher levels of the stack.

An example of application of this principle is the introduction of load Acquire (LDAR) and store Release (STLR) in the Arm ISA, to enable direct compilation of the sequentially consistent C++ atomic types. Another example is the introduction of the special load instruction (LDAPR), which in tandem with store Release instructions, enables efficient emulation of x86-TSO. Efficient emulation of x86-TSO might come into consideration when transitioning from x86 hardware to Arm hardware for example, if there is a need to support legacy x86 code running on top of the newly deployed Arm hardware.

Finally, we present in Section 5.2 the case of Linux's lockref primitive. This primitive requires certain ordering properties over mixed-size accesses, which the Arm architecture did not provide: we retrospectively strengthened the consistency model of the Arm architecture as we detail in this paper, to ensure that the guarantees needed by Linux were in place.

2.6 Equivalence of formulations

Arm intends to maintain all three formulations of the model, and the equivalence between them. Therefore, extensions of one of the formulations of the model must be reflected, in a provably equivalent way, in the other two formulations.

A putative extension might be rejected if we cannot integrate it in the alternative formulations, or if we can only integrate it in a way that betrays the per-thread reasoning principle. An example of this situation happened when developing our mixed-size extension, which involves augmenting the Observed-by (ob) relation.

The Observed-by relation describes interactions between threads. Therefore modifying it intuitively endangers the per-thread reasoning principle of Section 2.4. The per-thread reasoning principle is instrumental in ensuring that the three formulations are equivalent. To tackle this challenge, of reconciling our mixed-size extension with the per-thread principle, thereby ensuring the equivalence of our formulations, we found a formal trick which we detail in Section 6.5, which consists in ordering not just sole memory events, but equivalence classes of events in the two alternative formulations.

3 RELATED WORKS

This work is a descendant and sibling of several previous works, amongst which the original definition of the cat language [14] and works on attempting to define a consistency model for various aspects of the Arm architecture. We give a timeline of those models, starting in 2008, in Table 1.

Those previous works have contributed to our community's understanding and knowledge of the Arm consistency model. However even those that have been written in the context of a relationship with Arm have not been integrated into the Arm memory model. The exception to this is the cat model written by Deacon [18], presented as part of a paper by Pulte et al [33]. That paper introduces the original Armv8 model of [18], but the focus of the paper is on an equivalent operational formulation.

On the contrary in this paper we try to give a detailed explanation and motivation of the original cat model of [18]. Moreover our work on extending the original cat model to mixed-size accesses, as well as the introduction of new instructions into the ARM ISA, led us to make a few changes to the original model, which we detail in Section 4.

Two existing works are concerned with mixed-size accesses in the context of Arm's consistency model [20, 43].

The work of Flur et al [20] predates the original Arm model, hence cannot extend the cat model to handle mixed-size, unlike the approach in this paper. Instead the work of Flur et al [20] extends an obsolete operational formulation (given in a previous paper by Flur et al [19] and based upon a previous desire from Arm to specify a more relaxed, non-multi-copy-atomic architecture), which is an ancestor of the one presented by Pulte et al in [33]. Our experiments (detailed online at [24]) demonstrate that the mixed-size model of [20] is unsound with respect to the architecture. For example, the test given in Figure 21 is allowed by the architecture and observed on hardware, but forbidden by the model of [20].

The work of Watt et al [43] does not predate the original Arm model, nor the upstreaming of the Armv8 mixed-size model introduced in the present paper [1]. It does however set out to build "*a novel mixed-size ARMv8 axiomatic model, as a generalisation of ARM's axiomatic reference model, and validate it with respect to [...] [20, 33], a well-tested mixed-size operational model for ARMv8*". Therefore the model proposed by Watt et al [43] exposes itself to being unsound, much like the one of Flur et al [20]. Interestingly, the choice made by Watt et al in [43] was to use a much weaker model than the one in Flur et al [20], and in fact seems to coincide with the original guarantees made by the Arm specification, which we formalise in Figure 18.

Therefore, the results of Watt et al [43] are likely sound with respect to the architecture; however given that a much weaker model is used, those results may be quite a bit stricter than necessary. The reason for this is that a weaker model will not account for all the orderings that are occurring naturally, and therefore will require superfluous synchronisation. Quoting from Watt et al [43]:

> *We invest significant effort into defining and validating a mixed-size relaxed memory model for ARMv8. We benefit from the extensive body of existing work on the ARMv8 (and the related Power) memory model. To investigate compilation to other architectures, more work is needed to define their mixed-size behaviours. Most glaringly, we lack a formal model of mixed-size x86, one of the most common target platforms for JavaScript. Moreover, our ARMv8 model sidesteps some outstanding questions about the architecture's mixed-size behaviour, by, in doubt, choosing a reasonable weak option.*

The model of Watt et al is in fact not a "reasonable [...] option" since it is not strong enough to uphold guarantees necessary to Linux running soundly as we demonstrate in this paper.

Additionally, the present paper should spare the need to "invest significant effort" in defining putative mixed-size extensions of the Armv8 model, and risking for them to be either too weak or too strong. We would like to take this lost opportunity as an example and a plea for the academic community to reach out to Arm if needing official extensions to the Arm memory model, rather than building theorems, compiler mappings, or verified software stacks, on top of models which are either too weak or too strong. Additionally, the present paper also provides a mixed-size model for x86, which Watt et al [43] explicitly ask for in the quote above.

The work of Pulte et al [34] presents another operational model of Armv8, alongside an operational model of RISC-V, this time using the "Promising Semantics" approach of Kang et al [26]. The model presented in that paper does not handle mixed-size accesses. Indeed the authors write:

> *We do not yet model mixed-size accesses [. . .] since their architecturally intended semantics is still being clarified for ARM [. . .].*

Therefore the work we present in this paper should enable further development of this Promising Semantics-based model.

The work of Simner et al [41] tackles the semantics of instruction fetch in the context of the Arm architecture. It presents an extension of the operational model of Pulte et al [33], and interestingly,

a draft extension of the Arm cat model. It does not, however, give a formal semantics to the new cat constructs needed (for example, `wco`, `scl` or `CU`), nor does it provide an implementation thereof within the herd tool.

The works presented by Chong et al [17] and Raad et al [35] propose putative extensions of the Armv8 model to handle transactional and persistent memory respectively. Those formalisations have been expressed in Alloy [23], and they are not implemented within the herd tool. Those models have not been tested against hardware.

The work presented by Jagadeesan et al [25] presents an alternative formulation of the Armv8 model in terms of pomsets. Interestingly, it appears related in style with the alternative formulations that we present in this paper.

Year	Ref.	Comments
2008	[16]	An account of the pre-v8 Arm consistency model, mostly discussing litmus tests. The architecture was not multi-copy-atomic then.
2009	[8]	A paper focussing mostly on IBM Power—Arm was thought to be similar, in that neither architecture was multi-copy-atomic.
2014	[14]	A cat model of the pre-v8 Arm consistency model, based on extensive experiments.
2016	[19]	Two operational models, which are unsound w.r.t. hardware and have been obsoleted by the model of [18].
2017	[18]	The original Armv8 cat model.
2017	[20]	An extension of the models of [19] to mixed-size accesses.
2018	[33]	A presentation the original model of [18], focussing on an equivalent operational model, itself a descendant of the models of [19].
2018	[17]	Paper presenting a putative extension of the Armv8 model to handle transactional memory—the formalisation is done in Alloy, and not implemented within the herd tool.The model has not been tested due to lack of hardware.
2019	[35]	Paper presenting a putative extension of the Armv8 model to handle persistent memory—the formalisation is done in Alloy, and not implemented within the herd tool.The model has not been tested.
2019	[34]	Another operational model of Armv8, presented together with an operational model of RISC-V, using the "Promising Semantics" approach of [26].
2020	[41]	A putative extension of the Armv8 model to handle instruction fetches—proposes an extension of the operational model of [33], as well as a draft extension of the cat model. No semantics nor implementation is given to those new cat constructs.
2020	[43]	A mixed-size extension of the Armv8 model to handle mixed-size accesses, in Alloy and Coq. Appears to coincide with the original Arm guarantees given in Figure 18, which are too weak to program with—therefore the results in [43] might be quite a bit stricter than necessary.
2020	[25]	captures Arm-style MCA execution at the language level (including compiler optimizations).

Table 1. Timeline of Arm models. Only [18] has been adopted by the architecture.

4 THE CAT LANGUAGE AND THE ORIGINAL ARMV8 MODEL

Concurrent programs may communicate via shared locations (e.g., symbolic addresses x, y, z), use private locations (e.g., registers X1, X2) for logic or arithmetic, and control their execution flow with conditionals and loops. Use of shared accesses may result in weak behaviours. Figure 3 shows a concurrent program in A64 assembly code where two threads (called "observers" in Arm terminology) communicate via shared locations x and y, initialised to 0. P0 updates x, executes a DMB ST barrier instruction, and sets y to 1. P1 reads y, executes a DMB LD instruction, and reads x. This is a message passing idiom: with enough synchronisation, after P1 sees that the flag y is set, it must see the updated data. Here DMB ST and DMB LD are enough.

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

Fig. 2. The MP+DMB.ST+DMB.LD litmus test

For a given program, a consistency model determines which values can be returned from shared memory by load instructions. An axiomatic model—the style chosen by Arm—does so by determining whether candidate executions of a program are allowed. Candidate executions are graphs:

- nodes are events (or "effects", in Arm terminology) modeling the effect of instructions. For example, a memory read is an effect of a load instruction.
- edges form relations over events. For example, the program order relation (po) represents the order in which a Von Neuman computer would execute the instructions of a thread. The read-from relation (rf) specifies where a read takes its value from.

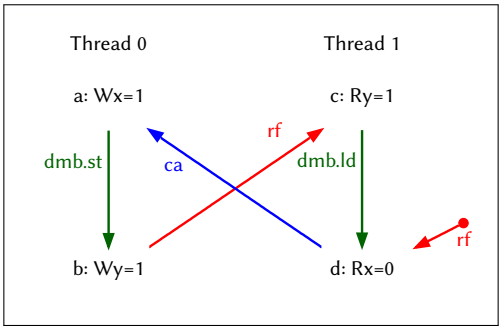


Fig. 3. A candidate execution of the MP+DMB.ST+DMB.LD litmus test

Figure 3 shows a candidate execution of the test in Figure 2. The nodes aligned vertically correspond to the semantics of a thread: P0 on the left and P1 on the right. The writer P0 writes the

value 1 into memory location x (event *a* in Figure 3), and sets the flag y to 1 (event *b* in Figure 3). The two corresponding instructions in Figure 2 are separated by a fence DMB ST, which gives rise in the candidate execution of Figure 3 to an arrow (dmb.st) between events *a* and *b*, with a label eponymous to the fence. The reader P1 reads the updated flag y (event *c* in Figure 3) and reads the data x (event *d*). The corresponding instructions are separated by a fence DMB LD in Figure 3, which gives rise to an eponymous arrow between events *c* and *d* (arrow dmb.ld).

In the specific execution we are interested in, the read *c* of variable y on P1 takes its value from the write of y on P0. The read-from (rf) arrow between *b* and *c* represents this. The read of x on P1 reads from the initial value of x; we depict this with a read-from arrow (rf) with a circle at its source. The initial write of x is by convention written to memory before any update of x in the program; we therefore say that the update *a* is coherence-after the read *d*, which takes its value from the initial write. We depict this with a coherence-after (ca) arrow between *d* and *a*.

4.1 Notions fundamental to cat models, and basic Arm definitions

4.1.1 Events. Events model the effects of instructions. Reads (R) take the value from a shared location, writes (W) to a shared location update said location with a given value, and fences (F) may prevent undesirable behaviours. Read-modify-write instructions generate a read and a write for the same shared location. Events bear annotations reflecting the corresponding instructions: plain, Acq or AcqPC (for reads); plain or Rel (for writes); and isb, dmb {sy, st, ld} or dsb {sy, st, ld} (for fences). For example, executing a load acquire instruction generates a read event annotated Acq, a plain store generates a write with no annotation, and dmb st generates a fence annotated dmb st. Table 2 lists the events for each instruction, omitting locations for brevity. Note that fence events are often omitted from drawings, and instead we use arrows labelled with the fence name.

Instruction	Events	cat set or relation
LDR	R	R
LDAR	R_{Acq}	A
LDAPR	R_{AcqPc}	Q
STR	W	W
STLR	W_{Rel}	L
DMB Type	$F_{DMB\ Type}$	dmb.type
DSB Type	$F_{DSB\ Type}$	dsb.type
ISB	F_{ISB}	isb
LDXR	R	domain(lxsx)
STXR	W	range(lxsx)
CAS	R,W	amo

Table 2. AArch64 instructions and their corresponding events and cat sets

4.1.2 Candidate executions. Candidate executions consist of abstract executions, representing the semantics of each thread, and execution witnesses, representing communications between threads. More precisely, independent symbolic execution of program threads first results in a set of *candidate executions* (detailed hereafter), each of which extends to a set of *witnesses*, which supplement the candidates with communication relations between threads.

Abstract executions (E, po, addr, data, ctrl, lxsx, amo) contain:

- E, the set of events;

- po, the program order, specifies instruction order in a thread after evaluating conditionals and unrolling loops;
- addr, data, and ctrl are the address, data, and control dependency relations—they are included in po, and always start from a read.
- lxsx links the read of a load exclusive to the corresponding write of a successful store exclusive. Note that a store exclusive may not be successful if its location has been tampered with by another thread, between the corresponding load exclusive and that store exclusive. In this case there is no lxsx relation, because there is no write event to link to.
- amo links the read of an atomic operation to its write.

Execution witnesses (rf, co) contain:

- the reads-from relation rf, which determines where reads take their value from. For each read r there is a unique write w to the same location s.t. r takes its value from w.
- the coherence order relation co, representing the history of writes to each location. It is a total order over writes to the same location, starting with the initialising write.

The notions constitutive of an execution witness are reflected in the Arm documentation [31], and we reproduce them verbatim in Figure 39.

4.1.3 The cat language. The cat language [7, 14] formalises consistency models as sets of constraints over candidate executions. In other words, a cat model states constraints over relations over events.

Sets and relations over events. The language provides the user with predefined sets of events (W contains all write events, R all reads, M all writes and reads and _ all events) and the relations forming candidate executions (po, addr, data, ctrl, lxsx, amo, rf, and co), as well as the identity relation id, the loc relation, which contains all pairs of events that access the same location, and the int relation, which contains all pairs of events that belong to the same thread. We also distinguish the initial writes IW (those which have no predecessor in coherence order) and the final writes FW (those which have no successor in coherence order).

Users can build new relations (declared with let or let rec for recursive definitions) via several operations: union (|), intersection (&), difference (\), complement (~), inverse (r^-1), reflexive closure (r?), transitive closure (r+), reflexive transitive closure (r*), sequence (r1 ; r2, defined as $\{(x, z) \mid \exists y \mid [(x, y) \in \texttt{r1} \wedge (y, z) \in \texttt{r2}]\}$) and Cartesian product of sets of events (S1 * S2). One can thus build the following relations, which often appear in cat models (and in particular in the Arm model):

- the program order relation restricted to accesses of the same location: po-loc = po & loc;
- the from-read relation makes one step of reads-from backwards, then one step of coherence: fr = rf^-1 ; co. In the Arm model, fr is presented as part of the wider coherence-after ca notion, which comprises both fr and co;
- the external relation ext, containing pairs of events that belong to different threads: ext = ~ int;
- the external reads-from, coherence and from-reads: rfe = rf & ext, coe = co & ext, and fre = fr & ext.

Finally, the cat language allows a user to define functions and procedures to be used later. One such example is the function intervening-write, which builds the subset of a transitive relation r which exhibits intervening writes:

```
let intervening-write(r) = r; [W]; r
```

More precisely, the cat expression above defines, for a given relation r a new relation as follows: take one step of r which lands on a write, then another step of r. The notation r; [W] is syntactic

```
(* Local read successor *)
let lrs = [W]; (po-loc \ intervening-write(po-loc)); [R]

(* Local write successor *)
let lws = po-loc; [W]

(* Coherence-after *)
let ca = fr | co

(* Observed-by *)
let obs = rfe | fre | coe
```

Fig. 4. Formal definitions of basic Arm terminology

sugar for the restriction of the relation `r` to its range (its right extremity) being in the set `W`, viz, being a write.

Building on those notions, the Arm model defines a handful of basic notions: we give their cat code in Figure 4, and the verbatim copy of their English transliteration as it appears in the Arm documentation [31] in Figure 40. Those notions are building blocks used in the statements of the constraints which constitute the Arm model:

- The Local read successor r of a write w is the first read to the same location as w occurring later in program order; thus we build it as the program order between events to the same location `po-loc`, between a write and a read, such that there is no intervening write in between those events.
- The Local write successors of a write w are all the writes to the same location after w in program order; thus we build those as the restriction of program order between writes to the same location.
- The Coherence-after relation (`ca`) gathers the from-read relation (`fr`) and the coherence order (`co`); intuitively a write w is coherence-after another event e (whether read or write) if w overwrites the value of e.
- The Observed-by relation gathers the external read-from (`rfe`) and external from-read (`fre`) relations and the external coherence order (`coe`); intuitively this represents the interactions between distinct threads over shared-memory.

Constraints over candidate executions. A cat model can constrain a relation `r` to be irreflexive, acyclic, or empty. Consider for instance the candidate execution depicted in Figure 3: read c takes its value from write b, hence the reads-from (`rf`) arrow between them. Read d takes the initial value, which is overwritten by write a, hence the coherence-after (`ca`) arrow between them.

This candidate execution is forbidden by the Arm model: the synchronisation enforced by the `DMB ST` instruction ensures that the updated data `x` is visible to `P1` when `P1` reads the flag `y`. This is because this execution exhibits a certain cycle, and such a cycle is forbidden by the Arm model. More precisely, the Arm model states three constraints over relations, which we will examine in the following sections:

- the Internal Visibility requirement in Section 4.2;
- the Atomicity requirement in Section 4.3;
- the External Visibility requirement in Section 4.4.

```
(* Internal visibility requirement *)
acyclic po-loc | ca | rf as internal
```

Fig. 5. Internal visibility requirement

Interestingly, the x86-TSO model can be stated following the same outline. One can formulate the x86-TSO model as a conjunction of three constraints: the first two being identical as the Arm ones (viz, Internal visibility in Section 4.2 and Atomicity in Section 4.3), and the third (External visibility in Section 4.4 being the same in spirit. More precisely the External visibility constraint has the exact same shape for both Arm and x86-TSO: both architectures require the acyclicity of the union of an intra-thread, local, ordering relation, and the communications between threads. The architecture only differs in the definition of the intra-thread local ordering relation. We detail this in Sections 4.4.2 and 5.4.3.

4.2 Internal Visibility Requirement

The Internal Visibility requirement can be understood as a sanity check on the communications, or interferences, between threads. It helps prevent unsettling behaviours commonly described as "lack of coherence", "reading from the future" or "dropping certain writes".

This requirement has been called "uniproc" [2], "SC per location" [14], or "SC per variable" [6]. Formally it states that there cannot be a cycle in the union of the following three relations:

- the program order restricted to events with the same location (`po-loc`);
- the Coherence-after relation (`ca`), defined in Figure 40; and
- the Read-from relation (`rf`).

We give the corresponding formalisation in cat in Figure 5.

The transliteration of this requirement into English in the Arm documentation [31, B2.3] is interesting. Rather than going for the succinct, literal transliteration of the requirement as it is given above, Arm has chosen to use an alternative, provably equivalent, phrasing. It has been proved before that the Internal visibility requirement forbids exactly the five patterns of Figure 6, as shown in [2, A.3 p. 184]).

In Figure 6, recall that read-from arrow (`rf`) with a circle at their source indicate that the target read takes its value from the initial state. The pattern `coWW` forces two writes to the same memory location `x` in program order to be in the same order in the coherence-after relation `ca`. The pattern `coRW1` forbids a read from `x` to read from a `po`-subsequent write. The pattern `coRW2` forbids the read a to read from a write c which is coherence-after a write b, if b is after a in program order. The pattern `coWR` forbids a read b to read from a write c which is coherence-before a previous write a in program order. The pattern `coRR` imposes that if a read a reads from a write c, all subsequent reads in program order from the same location (e.g. the read b) read from c or a coherence-successor write.

A last equivalent phrasing is given in [2]; it differs stylistically from the other two in that it is phrased positively. It says that two events e_1 and e_2 which are related by `po-loc` (i.e. which are related by the program order `po` and refer to the same location) are also related by one of the following relations: `co`, `rf`, `co;rf` (viz, `co` followed by `rf`), `fr;rf` (viz, `fr` followed by `rf`) or `rf^-1;rf` (viz, one step of `rf` backwards followed by one step of `rf` forwards; intuitively this relation links reads which read from the same write).

Thus the English transliteration of the Internal visibility requirement enumerates those cases: in Figure 7 the first bullet corresponds to e_1 and e_2 being in `co`; the second bullet corresponds to them being in `rf` or `co;rf`; and the third bullet corresponds to them being in `fr;rf`.

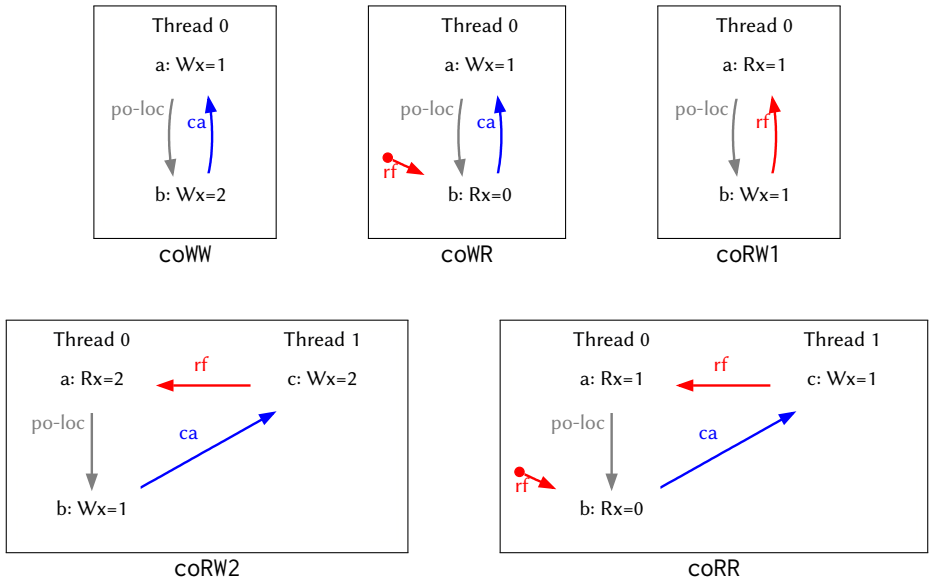


Fig. 6. The five patterns forbidden by the Internal Visibility Requirement

Internal visibility requirement

For a read or a write RW1 that appears in program order before a read or a write RW2 to the same location, the internal visibility requirement requires that exactly one of the following statements is true:

- RW2 is a write W2 that is Coherence-after RW1
- RW1 is a write W1 and RW2 is a read R2 such that either:
  - R2 Reads-from W1, or
  - R2 Reads-from a write that is Coherence-after W1
- RW1 and RW2 are both reads R1 and R2, such that R1 reads-from a write W3 and either:
  - R2 Reads-from W3, or
  - R2 Reads-from a write that is Coherence-after W3

Fig. 7. English transliteration of the Internal Visibility Requirement

4.3 Atomicity Requirement

Next up, we focus on the Atomicity requirement. Arm (as well as other architectures such as IBM Power for example) provides special instructions Load Exclusive (LDXR) and Store Exclusive (STXR): paired together, they allow a user to build an atomic access. More precisely, the LDXR instruction loads the location it has been given as an argument, but also *reserves* that location. Essentially it indicates that this location is not to be touched until that reservation has expired. The STXR instruction checks whether the location is still reserved: if so it succeeds and stores to it, otherwise it fails. However, the reservation mechanism is not forbidding: if another thread than the one executing the LDXR/STXR pair wants to write to the reserved location, it may—but then the STXR will fail. We give an illustration of this situation in Figure 8 where the * on the read and the write

indicate that they have been generated from an LDXR and an STXR respectively, as opposed to a plain LDR or STR.

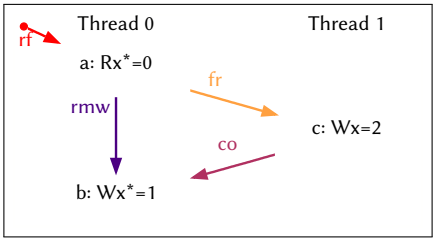


Fig. 8. An illustration of an intervening write making an exclusive pair non-successful

The cat primitive lxsx (for "load-exclusive-store-exclusive") describes a relation which only exists between a LDXR instruction and the next successful STXR. If the LDXR was paired with a non-successful STXR then there is no lxsx relation between them, because a failed STXR does not generate any write.

As per the architectural description above, an STXR instruction can only be successful if there has not been an intervening write by another thread, which tampered with the reservation between the LDXR and the STXR. Therefore there cannot be an external write to the same location in between the LDXR and the next successful STXR. In other words, there cannot be a write from a different thread which is coherence-after the read of the LDXR and coherence-before the write of the next successful STXR.

The same constraint holds for *atomic operations*, such as SWP or CAS. Such instructions generate a pair read-write to a given location, which also must not be interrupted by an intervening write. The cat primitive amo describes the relation which exists between the read of an atomic operation and the write of that operation when that write exists, viz, when the atomic operation is successful.

The cat relation rmw (for "read-modify-write") gathers those two relations: the lxsx relation between the read of a load exclusive and the corresponding write of a successful store exclusive, and the amo relation between the read of an atomic operation and the write of that operation. In cat, we have: `let rmw = lxsx | amo`.

Thus the Atomicity requirement is phrased as given in Figure 9: the intersection of the rmw relation with the set of read-write pairs which have a write in between them in the Coherence-after relation, is empty.

4.4 External Visibility Requirement

Lastly, we move on to the External Visibility requirement. In practice, this requirement states which synchronisation to use when one wants to forbid undesirably weak behaviours.

4.4.1 External Visibility Requirement. The Arm model does so by defining and using two building blocks:

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

Fig. 9. Atomicity requirement

```
(* Ordered-before *)
let rec ob = obs
           | lob
           | ob; ob

(* External visibility requirement *)
irreflexive ob as external
```

Fig. 10. External visibility requirement

Ordered-before

An arbitrary pair of Memory effects is ordered if it can be linked by a chain of ordered accesses consistent with external observation. A read or a write RW1 is Ordered-before a read or a write RW2 if and only if any of the following cases apply:

- RW1 is Observed-by a read or write RW2;
- RW1 is Locally-ordered-before RW2;
- RW1 is Ordered-before a read or write that is Ordered-before RW2.

External visibility requirement

For a read or a write RW1 from an Observer that is Ordered-before a read or a write RW2 from a different Observer, the external visibility constraint requires that RW2 is not Observed-by RW1. This means that an Architecturally well-formed execution must not exhibit a cycle in the Ordered-before relation.

Fig. 11. English transliteration of the External Visibility Requirement

- the Locally-ordered-before relation `lob` gathers all the possible ways that one can synchronise two instructions on the same thread. Examples include dependencies, fences, exclusive pairs and atomic accesses.
- the Observed-by relation `ob` gathers all the possible ways two different threads can interact: combinations of Coherence-after `ca` and Reads-from `rf`.

Then the External Visibility requirement, given in cat in Figure 10 and in English in Figure 11, states that those two relations cannot contradict each other: formally, there cannot be a cycle in their union, which is called Ordered-before `(ob)`.

The relationship between the cat formulation and the English transliteration of the External Visibility requirement is interesting. To state the acyclicity of the union of Locally-ordered-before (`lob`) and Observed-by (`obs`), we could have defined Ordered-before (`ob`) as the union of those two relations, and required Ordered-before to be acyclic. Another way would have been to define the Ordered-before relation (`ob`) as the transitive closure of the union of those two relations (making use of the cat postfix operator `r+`), then requiring Ordered-before to be, equivalently, irreflexive or acyclic.

However, Arm has chosen a different, provably equivalent, approach: instead we define Ordered-before (`ob`) recursively (see the use of `let rec` in Figure 10), thereby avoiding the use of the transitive closure operator. This allows the English transliteration to be much closer to the cat code: the three prose bullets in Figure 11 correspond exactly to the three cat clauses of Figure 10.

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

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

(* Barrier-ordered-before *)
let bob = po; [dmb.full]; po
        | po; ([A];amo;[L]); po
        | [L]; po; [A]
        | [R]; po; [dmb.ld]; po
        | [A | Q]; po
        | [W]; po; [dmb.st]; po; [W]
        | po; [L]

(* Locally-ordered-before *)
let rec lob = lws
            | dob
            | aob
            | bob
            | lob; lob
```

Fig. 12. Formal definitions for Arm Locally-ordered-before relation

But of course for the External Visibility requirement to fully make sense, we need to dive into the definition of the Locally-ordered-before relation `lob`, which we do next.

4.4.2 Arm Locally-ordered-before `lob`. The Locally-ordered-before relation `lob` is another case where the cat clauses (given in Figure 12) and the English bullets (given in Figure 41) correspond quite closely. Hence we refrain from giving a detailed paraphrasing of those definitions in the body of the text.

At a high-level the Locally-ordered-before relation `lob` is made of four distinct components:

- Local write successor `lws`, as defined in Figures 4 and 40;
- Dependency-ordered-before `dob`, which gathers all possible chains of dependencies which provide order;
- Atomic-ordered-before `aob`, which states how to make use of exclusive pairs (`LDXR/STXR`) and atomic operations to provide order; and
- Barrier-ordered-before `bob`, which gathers all possible ways to use fences to provide order.

Finally, Locally-ordered-before (`lob`) is transitive: as with the definition of Ordered-before (`ob`) in Figure 10, Arm has chosen to phrase this as a recursive definition (see the use of `let rec` in Figure 12), to enable a more direct English transliteration (see Figure 41).

4.4.3 Differences with the original Arm cat model of [18]. Our work on the cat model for Arm has led us to make a few changes, which are now upstream in the herd+diy distribution [9, 30], and which we detail here:

- the notion of internal coherence order (`coi` in cat) has been subsumed by the notion of Local-write-successor (`lws` in cat). The new notion can be determined solely from the program order—this is an instance of applying our "per-thread reasoning" principle (stated in Section 2.4).
- the notion of internal reads-from (`rfi` in cat) has been replaced by the notion of Local-read-successor (`lrs` in cat). The new notion can be determined solely from the program order—this is an instance of applying our "per-thread reasoning" principle (stated in Section 2.4).

The relation `lws` is equal to the union of `coi` and `fri`: this is because `co` is by definition total and both `coi` and `fri` must respect program order as per the Internal visibility requirement (see Section 4.2). Note that one can prove hat `lob; fri` is included in `lob` by case disjunction over the components of `lob`. Therefore replacing `coi` by `lws` does not add any extra constraint to `lob`, hence to the overall model.

The relation `lrs` is larger than `rfi` since having $(w, r) \in$ `lrs` allows for r reading from a write w' which is itself `co`-after w. If w' is on a different thread from w and r, then we have $(w, w') \in$ `coe` and $(w', r) \in$ `rfe`, in which case $(w, r) \in$ `ob` and hence we have not added a constraint to the model. If w' is on the same thread as w and r, then one can prove that `lob; fri` is included in `lob` by case disjunction over the components of `lob`. Therefore replacing `rfi` by `lrs` does not add any extra constraint to `lob`, hence to the overall model.

Two further changes are as follows:

- the use of `coi` in the original model was distributed along the clauses of Dependency-ordered-before. We observed that it was equivalent to factor it out and now list `lws` as an independent clause of Locally-ordered-before. This refactoring was beneficial when adding the mixed-size extensions, as it allowed us to add mixed-size considerations into the definition of Locally-ordered-before very locally (viz, only in the Local-write-successor clause) instead of every time that `coi` was used.
- The original model did not account for atomic operations such as `CAS` or `SWP`, which did not exist then. We have added the semantics of those instructions to the cat model, via the new cat construct `amo`. We have also added the primitive `lxsx` to be able to distinguish between read-modify-writes made of load-store exclusive pairs, and the ones made of atomic operations. Previously there was only one `rmw` notion. We need to distinguish between both types of read-modify-write (`rmw`) because atomic operations (`amo`) have an extra property which Load-Exclusive/Store-Exclusive pairs (`lxsx`) do not. This can be seen in the relation called Barrier-ordered-before (`bob`): if an atomic operation is so that its read is an Acquire and its write is a Release, then that atomic operation acts as a full barrier.

We have validated those changes experimentally as well, using 8970 litmus tests which can be found online at `http://diy.inria.fr/aarch64`.

5 A PRINCIPLE TO HANDLE MIXED-SIZE ACCESSES IN MULTI-COPY-ATOMIC MODELS

Modelling mixed-size concurrency means modelling the interactions of shared memory accesses of various sizes. For example, what happens if, following the litmus test given in Figure 13, two threads `P0` and `P1` communicate via a 16-bit variable `x`, in the following way: `P0` performs an 8-bit store to `x`, writing 1 into the least significant byte of `x`, and `P1` performs a 16-bit store to `x`, writing 2 in each 8-bit half of `x`. If `P0` performs a 16-bit load of `x`, what value is the load to see?

```
AArch64 WbRh+Wh
{
uint16_t x;

0:X1=x;
1:X1=x;
}
 P0            | P1            ;
 MOV W0,0x1    | MOV W0,0x202 ;
 STRB W0,[X1] | STRH W0,[X1] ;
 LDRH W2,[X1] |               ;
exists (x=0x202 /\ 0:X2=0x201)
```

Fig. 13. A first AArch64 mixed-size litmus test

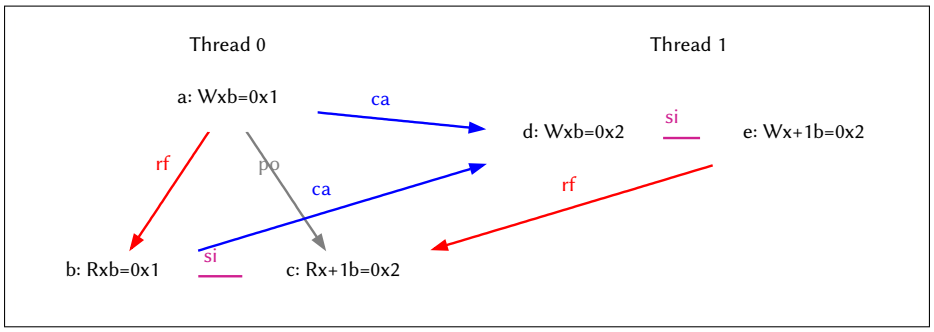


Fig. 14. Forbidden candidate execution of the test of Figure 13

We address such questions in the context of naturally aligned accesses only. We define a new relation `si` ("same instruction"), which relates events generated by the same execution of the same instruction. We extend the notion of abstract execution (see Section 4.1.2) to contain the relation `si` as well. For example, we give in Figure 14 one of the candidate executions of the test given in Figure 13. Note that the display of mixed-size accesses coming from the same instruction is different from the non-mixed-size case: for example, in Figure 14, the two events b and c come from the same instruction (the `LDRH` on `P0`, and are both in program order after the write a (generated by the `STRB` on `P0`)). Observe that the two write events d and e of the 16-bit store on `P1` are related by `si`, as are the two read events b and c of the 16-bit load on `P0`.

Interestingly, the example of Figure 13 is one of the only idioms which was originally forbidden by the Arm documentation, as we examine in Section 5.1. Those guarantees were fairly minimal, and did not make any provision about the interaction of mixed-size accesses and synchronisation such as barriers or atomics for example. This means that it would not have been possible to program with mixed-size accesses at a higher level, because the Arm architecture would not have made any guarantees.

Yet this still allows programmers to design and implement algorithms making use of this paradigm. In Section 5.2, we review such an example which comes from the Linux kernel, as it is one of the motivating examples which gave us a design guideline for extending the Arm model as we did.

Properties of single-copy atomic accesses

A memory access instruction that is single-copy atomic has the following properties:
1. For a pair of overlapping single-copy atomic store instructions, all of the overlapping writes generated by one of the stores are Coherence-after the corresponding overlapping writes generated by the other store.
2. For a single-copy atomic load instruction L1 that overlaps a single-copy atomic store instruction S2, if one of the overlapping reads generated by L1 Reads-from one of the overlapping writes generated by S2, then none of the overlapping writes generated by S2 are Coherence-after the corresponding overlapping reads generated by L1.

Fig. 15. Verbatim of B2.2.2 in ARM DDI 0487D.a ID103018

We then present in Section 5.3 a principle for modelling mixed-size concurrency in the context of multi-copy-atomic models. More precisely, we propose a way to extend existing concurrency models so that they apply to mixed-size accesses. For both x86 and Arm, we extend the cat model distributed within the herd+diy tool suite in consequence, as detailed in Sections 5.4.1 and 5.4.3.

We give detailed results of our experimental campaigns checking the soundness of our models against existing Arm and x86 hardware online [24]. To do so we have extended the test generator diy7 and the testing backend litmus7 to handle mixed-size accesses for both x86 and Arm. The extended tools are now distributed within the herd+diy toolsuite [9].

For Arm, we can go even further: the cat model distributed within the herd+diy tool suite is the official definition of the concurrency model guaranteed by the Arm architecture. Therefore we have the opportunity to review and extend the authoritative Arm definitions, in a way that is ratified by Arm. We give the verbatim copy of the extensions as they appear in the Arm Architecture Manual in Figures 43 and 44.

5.1 A note on what the original Arm documentation guaranteed

We review here the guarantees made about mixed-size accesses in the original Arm documentation. We give their verbatim copy in Figure 15. Note that in the Arm documentation, the term "overlapping" is applied to instructions which have one or more of their corresponding events accessing the same memory location.

Intuitively, those guarantees ensure that two stores do not mix their values. Here are two specific instances:

(1) two stores of the same size cannot result in a final value in memory being made of one half of the first store and one half of the second store. This corresponds to the first clause of Figure 15. This is illustrated by the litmus test in Figure 16, and the forbidden candidate execution given in Figure 17.
(2) two stores of different sizes cannot result in a subsequent load reading half of one store and the value from the second store. This corresponds to the second clause of Figure 15. This is illustrated by the litmus test in Figure 13 and the forbidden candidate execution given in Figure 13. Note that this holds regardless of if the smaller store comes first or second in the coherence order, since the coherence order will relate write events of the same size, hence will relate the sub-events of the bigger store with respect to the write event of the smaller store.

```
AArch64 SCA-1
{
0:X0=0x1010101; 0:X1=x;
1:X0=0x2020202; 1:X1=x;
}
 P0          | P1          ;
 STR W0,[X1] | STR W0,[X1] ;
exists x=0x1010202
```

Fig. 16. AArch64 litmus test forbidden by the first clause of Figure 15

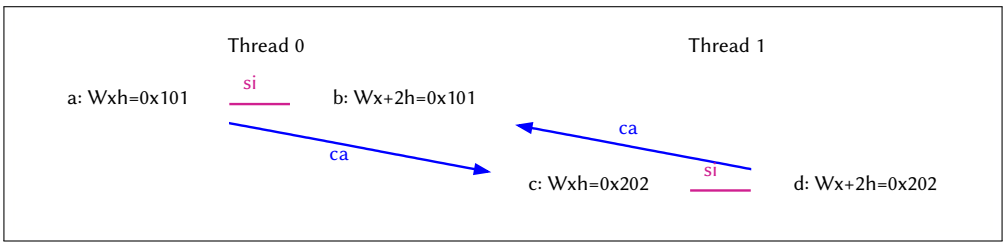


Fig. 17. Forbidden candidate execution of the AArch64 litmus test of Figure 16

```
(* Original mixed-size guarantees as per B2.2.2 in ARM DDI 0487D.a ID103018 *)
irreflexive co;si;co;si as sca1
irreflexive fr;si;rf;si as sca2
```

Fig. 18. Formal definitions of the original Arm mixed-size guarantees

For the sake of completeness, we give our cat formalisation of those original mixed-size guarantees in Figure 18.

5.2 A motivating example from Linux

The Virtual File System layer [21] inside the Linux kernel is responsible for handling system calls relating to POSIX filesystem operations in a manner that is agnostic to the low-level filesystem and on-disk format. This is primarily achieved using a set of abstract data structures for representing files and their metadata. Of particular significance are the `struct inode` and `struct dentry` objects, which we present briefly below.

A `struct inode` holds metadata about a filesystem object such as access permissions, file size and modification time. Locating an inode requires an iterative traversal of the path name components, a process known as a *path walk*. To improve performance of path name resolution, the Linux kernel uses a hash table to resolve a path name to a `struct dentry` pointer as quickly as possible. This hash table is known as the "Directory Entry Cache" (dcache) and is indexed by a hash function of the component name and parent dentry pointer. The Linux kernel goes to considerable lengths to ensure that the dcache is not a bottleneck when performing filesystem operations on even the largest of systems.

5.2.1 Scalability and `struct lockref`. To improve scalability of dcache lookups, the lockref structure is used to ensure that dcache hits returning a shared dentry structure can increment the

reference count without having to serialise on the spinlock when it is otherwise unheld. This is achieved by embedding the 32-bit dentry spinlock and reference count fields into `struct lockref`, which ensures that they are adjacent to each other in memory and aligned on a 64-bit boundary:

```
struct lockref {
    union {
        aligned_u64 lock_count;
        struct {
            spinlock_t lock;
            int count;
        };
    };
};
```

This structure allows the reference count to be incremented using a 64-bit `cmpxchg()` operation to ensure that the lock remains unheld while the count is incremented, and avoiding the update altogether if either the lock is taken or if the count indicates that the dentry is dead. For the sake of exposition simplicity, here we assume that the spinlock is a simple test-and-set implementation, using an atomic `xchg()` operation with acquire semantics in a busy-loop when taking the lock and an atomic store of zero with Release semantics when dropping the lock.

5.2.2 Mixed-size concurrency concerns. Overlaying a 32-bit lock word and a 32-bit reference count onto a combined 64-bit field, as is the case with the lockref structure, introduces mixed-size concurrency if one thread attempts a locked operation on a dentry while another attempts a 64-bit `cmpxchg()` operation to obtain a reference to the same dentry. In this situation, it is critical that mutual exclusion is ensured; either the `cmpxchg()` or the lock acquisition must fail.

As an example of this concurrency in action, consider a scenario where a thread has finished with a dentry and therefore drops its reference using a 64-bit `cmpxchg()`, indicating that the reference count is now zero and that the dentry is unused. Concurrently, a second thread locates the same dentry in the dcache as part of a path name lookup and increments the reference count from zero to one, again using a 64-bit `cmpxchg()`. The first thread then attempts to transition the dentry to "dead" and takes the spinlock using a 32-bit atomic operation. With the spinlock held, the first thread rechecks the refcount with a 32-bit load to ensure that it is still zero and can therefore be transitioned to the "dead" state before clearing its inode pointer to `NULL`.

For this protocol to work, it must be the case that either the first thread observes the second thread's increment to the refcount, or that the second thread observes that the dentry lock is held and fails to increment the refcount. In other words, the first thread must not clear the inode to `NULL` if the second thread succeeded in taking a reference on the dentry. This constraint can be expressed as the litmus test given in Figure 19.

We constructed this litmus test manually in order to reduce its complexity and to avoid confronting the reader with hundreds of unrelated instructions present in the vmlinux disassembly due to function inlining, uninteresting conditional code, stack management etc. Although this required some significant changes, for example replacing Linux's queued spinlock implementation with a test-and-set lock as noted in the paper, we manually correlated the result with the compiler output. This correlation was relatively straightforward, since Linux exclusively uses inline assembly for its atomic instructions (as opposed to C11 atomics or compiler builtins) and therefore we were able to match some of the instructions in the litmus test directly to sequences in the source code.

In Figure 20, we give the execution witness which corresponds to the mutual exclusion violation exposed above as being undesirable.

```
AArch64 Lockref
{
uint64_t lock_count;
0:X0=lock_count; 0:X2=0x1; 1:X0=lock_count; 1:X2=0x1; }
P0                                  | P1                                ;
(* Load the lock_count field *)     | (* Try to acquire the lock *)     ;
LDR X1, [X0]                        | SWPA W2, W1, [X0]                 ;
(* Check if the lock is held *)     | (* Check if we got the lock *)    ;
CBNZ W1, out0                       | CBNZ W1, out1                     ;
(* Check if the dentry is dead *)   | (* Load the reference count *)     ;
TBNZ X1, #63, out0                  | LDR W1, [X0, #4]                  ;
(* Increment the reference count *) |                                   ;
ADD X2, X1, X2, LSL #32             |                                   ;
(* Attempt the cmpxchg() *)         |                                   ;
CAS X3, X2, [X0]                    |                                   ;
(* Set X1 to 0 iff CAS succeeded *) |                                   ;
SUB X1, X1, X3                      |                                   ;
out0:                               | out1:                             ;
(* If X1 is 0, access inode *)      | (* If X1 is 0, clear inode *)     ;
exists(0:X1 = 0 /\ 1:X1 = 0)
```

Fig. 19. A Linux lockref example

Prior to our work, the undesirable outcome of this litmus test was not prevented by the Arm architecture, which in turn means that Linux's `lockref` could not be soundly used on Arm. Our work addresses this as follows.

5.3 Our principle

In this section we detail how we model mixed-size accesses in x86 and Arm. More precisely, we propose a modelling principle to extend certain models to mixed-size accesses. Our models are distributed within the herd+diy toolsuite [1, 4].

We place ourselves in the restricting context of models which are structured like the Arm or the TSO ones, viz, following the three constraints Internal visibility (see Section 4.2), Atomicity (see Section 4.3) and External visibility (see Section 4.4).

Another way to understand those models is as follows. Essentially those models create a taxonomy of relations:

- the local orderings preserved by the architecture, i.e. Locally-ordered-before (`lob`): see Section 4.4.2 for Arm, and Section 5.4.3 for x86-TSO;
- the orderings relative to the values taken by one given memory location (sometimes called communications, or interferences) which are deemed "global" by the architecture, i.e. Local-write-successor (`lws`) and Observed-by (`obs`) in Arm and all communications except for the internal reads-from relation in TSO.

We could find two obvious pathways to extending existing models to mixed-size accesses:

- when two events are locally ordered, then some of the sub-events generated by the same instructions are too;
- when two events are ordered by a communication relation, then some of the sub-events generated by the same instructions are too.

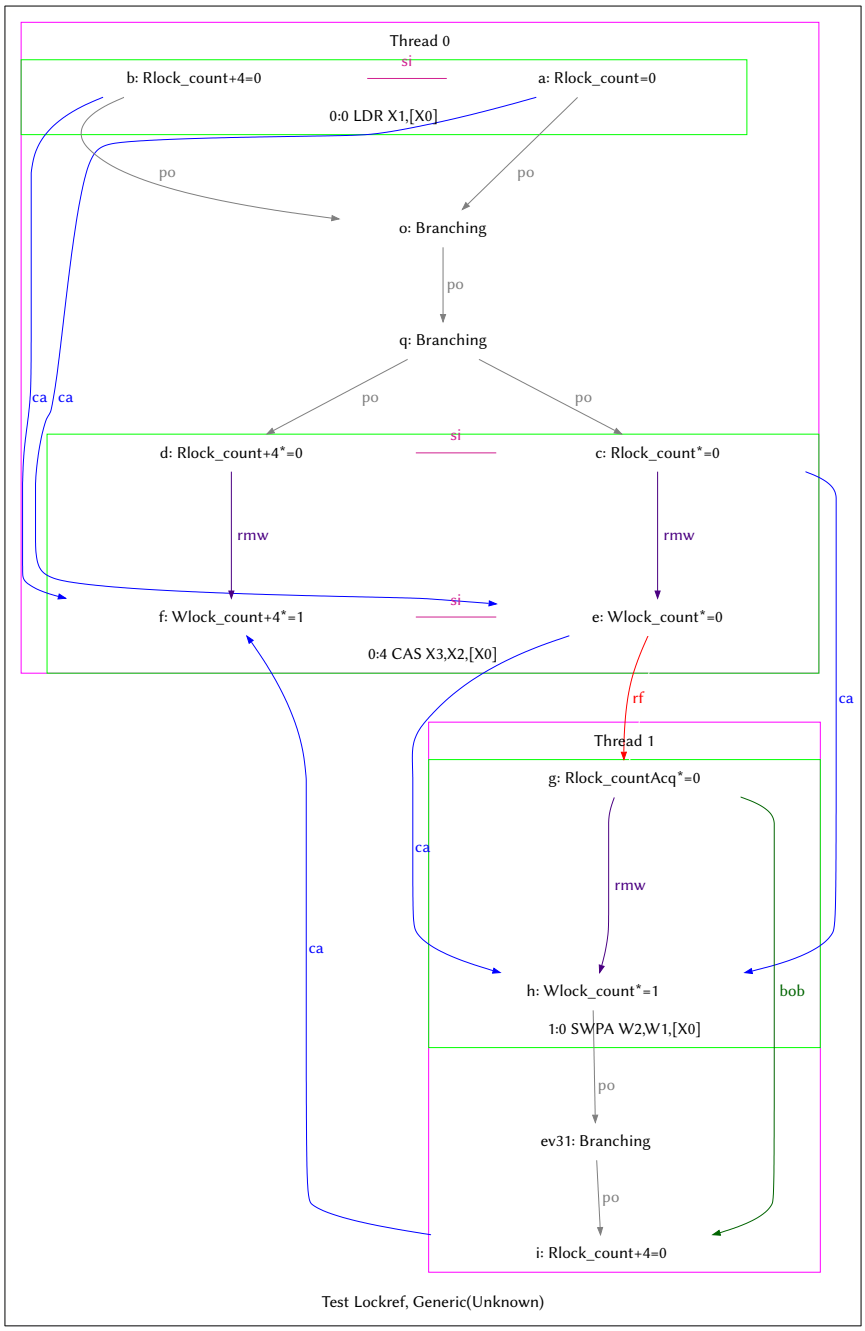


Fig. 20. Execution witness corresponding to the test of Figure 19

```
AArch64 MP+dmb+addr-rfi+MIX+OK
{
uint16_t y; uint16_t x; uint16_t 1:X5; uint16_t 1:X0;
0:X0=0x11; 0:X1=x; 0:X2=0x1111; 0:X3=y;
1:X1=y; 1:X4=x; 1:X9=0x22
}
 P0           | P1                    ;
 STRB W0,[X1] | LDRH W0,[X1]          ;
 DMB SY       | AND W2,W0,#1          ;
 STRH W2,[X3] | STRB W9,[X4,W2,SXTW] ;
              | LDRH W5,[X4]          ;
exists(x=0x2211 /\ 1:X5=0x2200 /\ 1:X0=0x1111)
```

Fig. 21. An AArch64 test observed on hardware

Consider the test given in Figure 21. This test asks the following question: does the Locally-ordered-before link between the instructions on P1 extend its reach to all the sub-events of each instruction? To understand this question more precisely, consider the candidate execution given in Figure 22.

The sequence of instructions on P1 is made of a step of address dependency (addr) between the first LDRH and the STRB (due to the AND instruction) followed by an internal read-from (rf) between the STRB and the second LDRH. This read-from however is between the whole of the write event f generated by the STRB and one of the two read events g and h of the second LDRH, more precisely, the read h. We know that in the original model, the sequence of address dependency and internal read-from (viz, addr;rfi) is in Locally-ordered-before. Therefore we know that (d, h) and (e, h) are in Locally-ordered-before The question that the test of Figure 21 is asking is whether this ordering extends to the other half of the same LDRH instruction as well, i.e. to the read g.

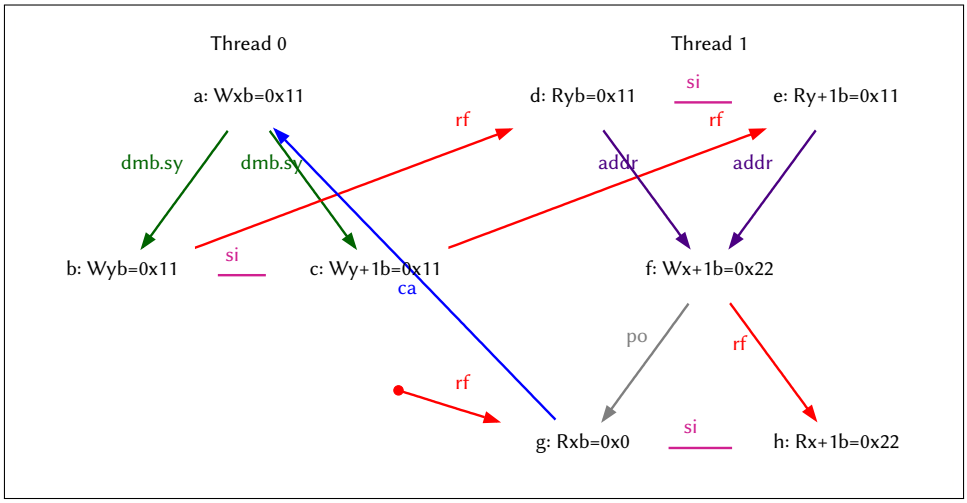


Fig. 22. Execution witness corresponding to the test of Figure 21

The answer is no: this test is observed on hardware, and has been deemed architecturally allowed by Arm. Thus we cannot simply extend the Arm model by extending Locally-ordered-before to apply to all sub-events of an instruction.

Rather we take the other approach: we consider that mixed-size accesses inherit the orderings due to communication relations, which seems a natural extension of the minimal guarantees that already appeared in the Arm documentation, given in Figure 15. Thus our extension principle is as follows: *if an event e_1 is before another event e_2 with respect to a global communication relation, then e_1 is also before all the sub-events generated by the same instruction as e_2.*

The test of Figure 21 also illustrates the necessity for the extension principle to be slightly asymmetric. Indeed it might be tempting to augment the extension principle by requiring that if an event e_1 is before another event e_2 with respect to a global communication relation, then all sub-events generated the same instruction as e_1 are before e_2.

However doing so would make the test of Figure 21 forbidden, when it is observed on hardware and allowed by the Arm architecture. The reason for the test becoming forbidden if we were to extend our principle in a symmetric manner can be seen in its execution witness, given in Figure 22: the write a of x on P0 is Ordered-before (due to the `DMB SY`) the write c of y+1. The write c of y+1 on P0 is Ordered-before the read e of y+1 on P1 (because e reads-from c and c and e come from different threads). The read e of y+1 on P1 is Locally-ordered-before, hence Ordered-before the read h of x+1. That is because an address dependency followed by an internal read-from is Locally-ordered-before. Then if we made our extension principle symmetric, the read h of x+1 would be Ordered-before the write a of x on P0, thereby creating a forbidding cycle where there should not be one.

5.4 Validation

5.4.1 Arm model extended to mixed-size accesses. Consequently, the Arm model is extended as given in Figures 23 and 24—there are two places in the Arm model where global communications appear:

- in the Local-write-successor clause (`lws`) of the definition of Locally-ordered-before (`lob`) given in Figures 12 and 41: RW1 is a write W1 and RW2 is a write W2 such that W2 is a Local write successor of W1;
- in the Observed-by clause (`obs`) of the Ordered-before definition (`ob`) given in Figures 4 and 40: RW1 is Observed-by a read or write RW2.

Applying our principle to those two relations, we get the following two new clauses:

- change the Local write successor clause of the Locally-ordered-before definition to: RW1 is a write W1 and RW2 is a write W2 that is equal to or generated by the same instruction as a Local write successor of RW1. We give the corresponding change to the cat file in Figure 23.
- change the Observed-by clause of the Ordered-before definition to: RW1 is Observed-by a read or write RW3 that is equal to or generated by the same instruction as RW2. We give the corresponding change to the cat file in Figure 24.

For the sake of completeness, we also give the verbatim copy of the Arm documentation translitterating our formal cat definitions in Figures 43 and 44.

Note that the Arm documentation now does not include the original Properties of single-copy atomic accesses given in Figure 15. Indeed the new definitions of Figures 23 and 24 include the original properties given in Figure 18:

- For the clause sca1: this is an immediate consequence of `lws;si` and `obs;si` (hence `coe;si`) being in `ob`, therefore being irreflexive.
- For the clause sca2, let us distinguish the cases where the read-from `rf` are internal or external:

```
(* Arm Locally-ordered-before modified to handle mixed-size accesses *)
let rec lob = lws; si
            | dob
            | aob
            | bob
            | lob; lob
```

Fig. 23. Arm Locally-ordered-before relation modified to handle mixed-size accesses

```
(* Arm Ordered-before modified to handle mixed-size accesses *)
let obs = rfe | fre | coe

let rec ob = obs; si
           | lob
           | ob; ob
```

Fig. 24. Arm Ordered-before relation modified to handle mixed-size accesses

- when rf is internal, then the fr link also is. As a consequence, the sequence fr;si;rf;si cannot be reflexive as it would be a contradiction of the Internal visibility requirement.
- when rf is external, then the fr link also is. Hence the sequence fr;si;rf;si cannot be reflexive as it would be a contradiction of the External visibility requirement (since both rfe and fre are included in obs).

5.4.2 Revisiting our Linux lockref example. Let us turn back to our Linux lockref example, given in Figure 19, and its undesirable execution, given in Figure 20. Our new model does forbid this undesirable execution, as follows:

- the event g from the SWPA on P1 is Ordered-before the event i from the LDR on P1, because the SWPA has acquire semantics, and therefore memory events which come from instructions in program order after SWPA must remain after the events of the SWPA. This is captured in the Barrier-ordered-before relation (clause [A]; po).
- the event i from the LDR on P1 is Ordered-before the event f from the CAS on P0, because the read i reads from a write (the initial write to location lock_count+4) which is overwritten by the write f of the CAS. Therefore the write f is Coherence-after the read i, and because they come from different processors, this entails that i is Ordered-before f.
- the read g from the SWPA on P1 reads from the write e from the CAS on P0, and because those events come from different processors, this entails that e is Ordered-before g.
- Finally, and this is where our mixed-size extension comes into play: because the events e and f from the CAS are generated from the same instruction, they are related by the "same-instruction" (si) relation.

Therefore we get the following cycle: (e, g, i, f, e), which is forbidden by the extended External visibility requirement.

5.4.3 An x86 model extended to mixed-size accesses. Following the principle of Section 5.3, we extend the TSO model of the herd+diy distribution as given in Figure 26. The TSO cat file is structured much like the Arm one; it has the same three axioms although they may have been called differently in the literature: Internal visibility, Atomicity and External visibility.

The main difference is in the phrasing of the External visibility requirement. To make our exposition easier, we have done some provably equivalent reorganisation of the TSO-equivalent of the External visibility requirement, so that it is now structured like the Arm one.

More precisely, we now have a notion of Locally-ordered-before for TSO, which is given in Figure 26. The definitions of Observed-by and Ordered-before for TSO are the same as the ones for Arm.

x86-TSO Locally-ordered-before. The Locally-ordered-before relation for TSO, given in Figure 25, is stronger than Arm's, which means that more behaviours will be naturally forbidden on TSO.

Essentially, TSO maintains all pairs in program order except for write-read pairs. Therefore the first component of TSO's Locally-ordered-before relation, given in Figure 25 is `po \ ([W];po;[R])` reads as "the whole program order relation, minus the pairs which start with a write and end with a read".

Those write-read pairs can be maintained if one uses an `MFENCE` instruction in between them (as formalised by the second component of the `lob` relation given in Figure 25), or if at least one of the two accesses results from the execution of an atomic instruction—in the x86 sense, for example using x86's `lock` prefix, which allows a user to transform a simple addition into an atomic addition. This is formalised by the third and fourth components of `lob` in Figure 25, where `X` symbolises accesses generated by an atomic instruction. The last component of `lob` ensures that it is a transitive relation, viz, one can chain the first fourth components to create a `lob` link.

```
let rec lob = po \ ([W]; po; [R])
        | [W]; po; [MFENCE]; po; [R]
        | [W]; po; [R & X]
        | [W & X]; po; [R]
        | lob; lob
```

Fig. 25. Formal definition for x86-TSO Locally-ordered-before relation

x86-TSO extended to mixed-size following our principle. Thus to handle mixed-size accesses in TSO, much like in the Arm case, we simply extend the model as follows:

- extend the Locally-ordered-before clause relative to internal coherence (Local write successor) and from-read to accesses generated by the same instruction
- extend the Observed-by clause in the Ordered-before definition to accesses generated by the same instruction.

Note however that the Locally-ordered-before relation in the TSO case is wider, and encompasses much of the program order, except for the write-read pairs. Hence much like in the non-mixed case, the TSO model is stronger than the Arm model: for example the litmus test of Figure 21 is forbidden under our TSO mixed model. The corresponding cat is given in Figure 26.

We now move on to presenting the two alternative formulations of the Arm memory model. Those were prompted by consumers of the specification such as hardware designers or verification engineers, who felt alternative formulations could be more intuitive, or more easily communicated and taught.

5.5 Experimental results

We give here an overview of our experimental campaigns and results. Detailed tables can be found at `http://diy.inria.fr/mixed`.

```
let rec lob = lws; si
            | po \ ([W]; po; [R])
            | [W]; po; [MFENCE]; po; [R]
            | [W]; po; [R & X]
            | [W & X]; po; [R]
            | lob; lob

let rec ob = obs; si
           | lob
           | ob; ob

irreflexive ob as external
```

Fig. 26. TSO External visibility requirement extended to handle mixed-size accesses

5.5.1 Experiments on x86. Our x86 test base consists of 14833 mixed-size tests, most of which generated by the `diy7` tool. Our pool of machines consists of:

- a dual-core (Intel Core i5-53005U, 2.30 GHz) HP EliteBook laptop,
- a quad-core (Intel Core i7-4770HQ, 2.20 GHz) Apple MacBook pro,
- an octo-core (quad-core Intel Xeon E5620, 2.40 GHz, x2) HP Z800 desktop,
- a 12-core (hexa-core Intel Xeon E5-2620, 2.40 GHz, x2) Dell Precision 7810 desktop,
- a 40-core (10-core Intel Xeon E7-4870, 2.40 GHz, x4) Dell PowerEdge R910 Server.

All tested machines have 2-way hyper-threading enabled.

We ran each test 18G times in total, spread across those machines. We have not observed any contradiction of our proposed model, which demonstrates its experimental soundness. For completeness, we observe that 69 tests which are allowed by our model have not exhibited themselves—a small number as compared to the total number of tests run.

5.5.2 Experiments on Arm. Our Arm test base consists of 2669 mixed-size tests—a combination of tests automatically generated by `diy7` and hand-written tests. Our pool of machines consists of:

- S905, a quad-core Arm Cortex-A53-based SoC
- S922X, a quad-core Arm Cortex-A73 and dual-Core Arm Cortex-A53-based SoC
- a Qualcomm Snapdragon810
- a Qualcomm Snapdragon820
- a Qualcomm Snapdragon425
- an Apple A10XFusion
- a Samsung Exynos9
- BCM2711, the Broadcom chip used in the Raspberry Pi 4 Model B
- a Mediatek HelioG25, composed of 8 Arm Cortex-A53 CPUs

We ran each test between 15G and 96G, with an average of 62G, spread across those machines. We have observed contradictions of our proposed model, but those were deemed by Arm to be hardware anomalies. For example, we observed the test `SCA-02` (test `WbRh+Wh` of Figure 13), 10 times over 37G, on Snapdragon810, Snapdragon 425 and HelioG25. An analysis of those anomalies is available online at `http://diy.inria.fr/mixed/classify-mixed-aarch64/`.

In more detail, we have categorised the hardware anomalies we observed according to which cycle they exhibit. The model is phrased in terms of forbidding cycles, therefore an anomaly can be characterised by which cycle it exhibits. The anomalies we observed were of three kinds:

- 245 invalid executions that contain a cycle "cae;si;rfe;si", for example the test of Figure 13 and its execution show in Figure 14;
- 1 invalid execution that contains a cycle "cae;dmb.sy;rfe;si", i.e. the test of Figure 27;
- 87 invalid executions that contain a cycle "cai;rfe;si;cae;si", for example the test of Figure 28.

```
AArch64 WW+R+dmb.sysw4w0+q0+BIS
{
uint64_t x; uint64_t 1:X0;
0:X0=0x1010101; 0:X1=x; 0:X2=0x2020202; 1:X2=0x3030303;
1:X1=x;
}
 P0              | P1              ;
 STR W0,[X1,#4] | STR W2,[X1,#4] ;
 DMB SY          | LDR X0,[X1]     ;
 STR W2,[X1]     |                 ;
exists (x=0x101010102020202 /\ 1:X0=0x303030302020202)
```

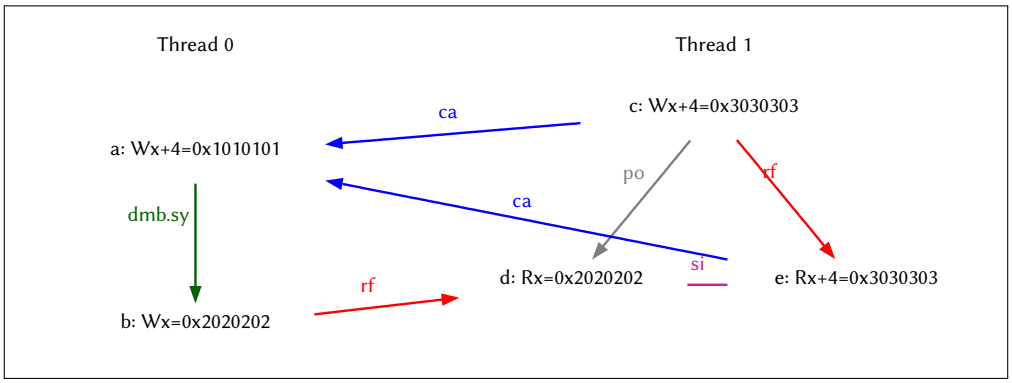


Fig. 27. Anomaly WW+R+dmbsysw4w0+q0+BIS

In the anomaly given in Figure 27, the write a of x+4 is Ordered-before the write b of x, thanks to the DMB SY fence between them. The read d of x and the read e of x+4 are both generated by the LDR X0,[X1] instruction on P1, hence related by si. The anomalous behaviour that we observed on hardware is as depicted in Figure 27: the read d reads from the write b, yet the read e is Ordered-before the write b. This is because the read e is Coherence-before the write a, itself Barrier-ordered-before the write b, altogether a contradiction of the External Visibility requirement.

In the anomaly given in Figure 28, the write a of x and the write b of x+4 are both generated by the STR X0,[X1] instruction on P0. The write a of x is Coherence-before the write c of x. The read e of x and the read f of x+4 are both generated by the LDR X0,[X1] instruction on P1. The write c gives its value to the read e of x, whilst the read f of x+4 is Coherence-before the write b of x+4. This is another contradiction of the External Visibility requirement.

```
AArch64 SCA-04
{
uint64_t x; uint64_t 1:X0;

0:X0=0x101010101010101; 0:X1=x; 0:X2=0x2020202;
1:X1=x; 1:X2=0x3030303;
}
 P0          | P1             ;
 STR X0,[X1] | STR W2,[X1,#4] ;
 STR W2,[X1] | LDR X0,[X1]    ;
exists(x=0x101010102020202 /\ 1:X0=0x303030302020202)
```

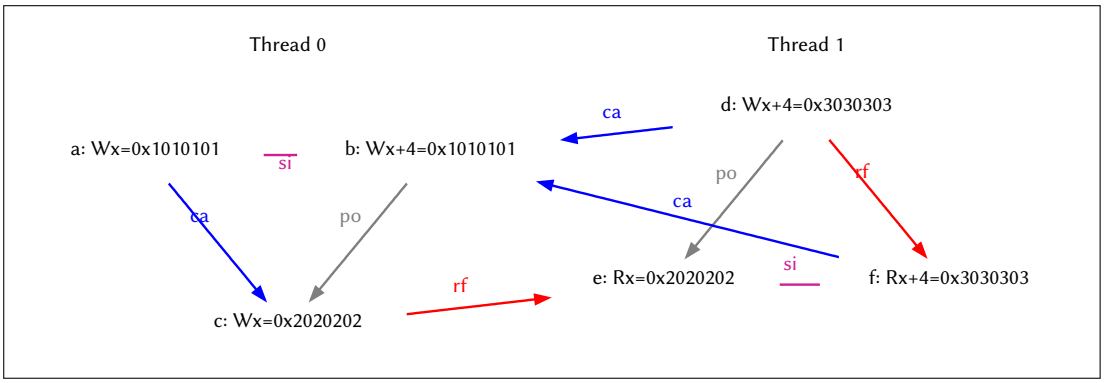


Fig. 28. Anomaly SCA-04

6 TWO ALTERNATIVE FORMULATIONS OF THE EXTERNAL VISIBILITY REQUIREMENT

6.1 A worked example

To provide an intuitive overview of the alternative formulations, let us examine the test given in Figure 29. This test is another message-passing shape between two threads P0 and P1 communicating via shared-memory locations x and y. This time the interesting thread is the writer P0, since the reader P1 is properly synchronised with a DMB LD between its two loads. On P0, the store of the data x with value 1 is immediately followed in program order by a load of x. The store of the flag y has an address dependency on this load of x, due to the manipulation of registers via EOR.

The question that the test is asking is as follows: if the load of the data x on P0 reads from the store of x immediately before it in program order, and if the load of the flag y on P1 does receive the updated flag from P0, can the load of the data x on P1 see the stale value of x still, or must it see the updated value by P0?

Naturally the answer is that the Arm architecture permits the load on P1 to see the stale value—since the thread P0 is not properly synchronised: the sequence on P0 does not contribute to the Locally-ordered-before relation of Figure 12.

Therefore all three formulations will answer the same: that the final state given in Figure 29 is allowed. However, they will proceed differently to each other in determining that answer.

Before we dive into how each formulation builds its execution witness for the test of Figure 29, let us present briefly the differences and motivations for each formulation:

```
AArch64 MP+rfi-addr+dmb.ld
{0:X1=x; 0:X5=y; 1:X1=x; 1:X5=y;}
P0                  | P1         ;
MOV W0,#1           | LDR W0,[X5];
STR W0,[X1]         | DMB LD     ;
LDR W2,[X1]         | LDR W2,[X1];
EOR W3,W2,W2        |            ;
MOV W4,#1           |            ;
STR W4,[X5,W3,SXTW] |            ;
exists (0:X2=1 /\ 1:X0=1 /\ 1:X2=0)
```

Fig. 29. A test to illustrate the alternative formulations of the Arm model

- the original model is written in a quite classic axiomatic way: stating constraints over relations, which essentially ensure that those relations are partial orders. This style is quite close to higher-level models such as C++ and Linux, and therefore lends itself quite well to be compared to those models.
- the second formulation prefers a formulation where a total order is built, but otherwise follows quite closely the original model, in that it simply linearises the partial orders constrained by the original model. It appears that total order formulations are more intuitive to certain audiences (certain hardware designer in particular). This style is also much closer to operational models (although this second formulation is not itself an operational model, but rather could be seen as an abstraction of an operational model), and therefore lends itself quite well to be compared to those models.
- finally, the third formulation also chooses to build a total order, but this time the construction of that total order does not simply follow the definitions of the original model. In particular, it departs from the Locally-ordered-before relation, in order to pick read events to place in the total order, such that those reads are valid a priori. By contrast, the second formulation requires an a posteriori check to ensure that the reads placed in the total order are valid. Ensuring that reads are valid by design, a priori, ensures a lesser load for verification methods checking that executions are correct with respect to the consistency model.

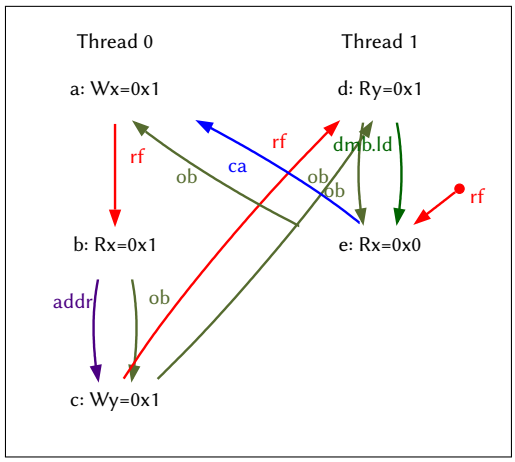
6.2 Building the execution witness of the test of Figure 29 using the original model

In the original model, the External visibility requirement given in Figure 10 is at play. In this point of view, to determine whether an execution is valid or not, we proceed as follows:

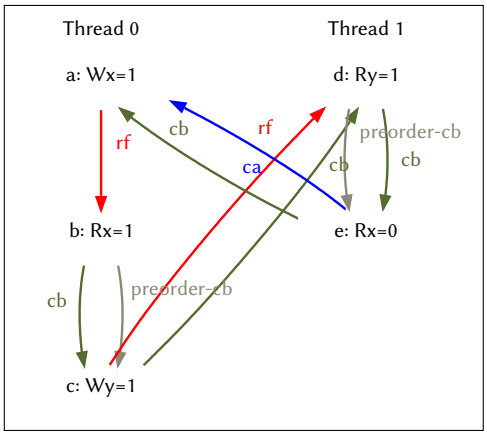
- per thread, we build the Locally-ordered-before relation, by looking for dependencies, fences, atomics and exclusives (see definition of `lob` in Figure 12);
- between threads, we build the Observed-by relation (see definition of `obs` in Figure 4);
- we check whether those relations form a cycle (see definition of `external` in Figure 10).

This leads to the execution witness given in Figure 30-(a). On each thread we build the Locally-ordered-before relation:

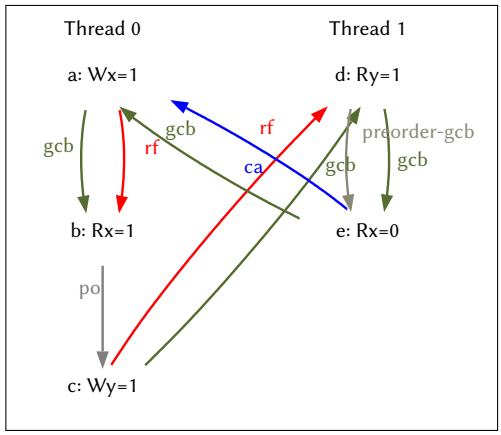
- on `P0`, the `EOR` between the load and the second store creates an address dependency; this is depicted by an addr arrow between the corresponding events: the read b of `x` and the write c of `y`.



(a) External visibility formulation.



(b) External completions formulation.



(c) External global completions formulation.

Fig. 30. Execution witnesses of the test of Figure 29, as produced three models

- on P1, the DMB LD between the two loads creates a barrier-ordered-before relation; this is depicted by a dmb.ld arrow between the corresponding events: the read *d* of y and the read *e* of x.

Between threads, we build the Observed-by relation:

- the read *d* of y on P1 reads from the write *c* on P0, which is depicted by the read-from arrow between those events;
- the read *e* of x on P1 reads from the initial state. The initial state for x is overwritten by the write *a* on P0. Therefore the read *e* is coherence-before the write *a*; we depict this with a coherence-after (ca) arrow from *e* to *a*.

Now we need to establish whether the Locally-ordered-before relation and the Observed-by relation that we just built form a cycle. To help with this, we have depicted the corresponding Ordered-before (ob) arrows. Observed that the read-from arrow on P0 (which corresponds to the

load of x reading from the first store on P0) does not qualify as an Ordered-before arrow. Therefore there is no cycle in the candidate execution of Figure 29-(a), which makes it a valid candidate execution as per the External visibility requirement.

We now move on to presenting the two alternative views.

6.3 External completion requirement

In the first alternative view, a new requirement replaces the External visibility requirement: it is called the External completion requirement. This alternative view differs from the original view in that it seems better suited to hardware designers, because it does not require to reason about a given program by consider the whole program. We give the formal details of the External completion requirement in Figure 31—here we simply give an illustration of the requirement on the test of Figure 29. In the External completion point of view, to determine whether an execution is valid or not, we proceed as follows:

- we build a relation called IM0, in which the initial writes are before any other access to the same location, and all accesses to a given location are before the final write to that location;
- per thread, we build the Locally-ordered-before relation, by looking for dependencies, fences, atomics and exclusives (see definition of lob in Figure 12);
- we then try to build a total order called "Completes-before" (cb) over all accesses, which respects both IM0 and the Locally-ordered-before relation built as above.

This leads to the execution witness given in Figure 29-(b). We build the Locally-ordered-before relation as before: the address dependency on P0 and the barrier-ordered-before relation due to the DMB LD on P1 both contribute to this. We then try to build a total order which embeds those Locally-ordered-before relations: the order b, c, d, e, a satisfies those constraints.

If such a Completes-before order exists, we then need to justify the value taken by each read access r, as follows:

- either it is a case of store forwarding: there exists a write w to the same location as r on the same thread as r, which is the last write to the same location before r in program order. If in the total order built as above, r appears before w, then we record r as reading from w. In Figure 29-(b), this corresponds to $w = a$ and $r = b$.
- or it is not a forwarding case: then r reads from its closest preceding write to the same location in the Completes-before order. For example in Figure 29-(b), this corresponds to $w = c$ and $r = d$. If no such write exists, then r reads from the initial value of the memory location, as is the case in Figure 29-(b) for $r = e$.

The cat formulation of the External completion requirement, given in Figure 31, is interesting: it makes use of the cat constructs with and linearisations. Thus the way to read this cat file is as follows: we build all the total orders which respect Locally-ordered-before using linearisations(M,preorder-cb). We then pick one, which we call cb, using the cat construct with. We check that cb is not empty, which means that there exists such a total order. We then check that the reads-from and the coherence order are valid as presented above.

For completeness we give the English transliteration of the External completion requirement in Figure 42.

6.4 External global completion requirement

In the second alternative view, a new requirement replaces the External visibility requirement: it is called the External global completion requirement. We give the formal details of it in Figure 32—here we simply give an illustration of the requirement on the test of Figure 29.

```
(* External completion requirement *)
let IM0 = loc & ((IW * (M\IW)) | ((W\FW) * FW))
let preorder-cb = IM0 | lob
with cb from linearisations(M, preorder-cb)
~empty cb

let rf-fwd = (W * R) & po-loc & cb^-1 \ intervening-write(po-loc)
let rf-nfwd = (W * R) & loc & cb \ intervening-write(cb & loc)
let rf-cb = rf-fwd | rf-nfwd
let co-cb = (W * W) & loc & cb

call equal(rf, rf-cb)
call equal(co, co-cb)
```

Fig. 31. External completion requirement

The External global completion requirement has the same overall principle as the External completion requirement given above, viz, building a total order called Globally-completes-before over all accesses which respects the Locally-ordered-before relation. However, instead of justifying who reads from where a posteriori like in the External completion view, the External global completion view chooses to embed the constraints on the validity of reads within the building of the Globally-completes-before order. This has benefits for verification engineers in charge of validating core designs: in a given total order trace, the reads given are valid by construction—there is no need for an extra algorithmic step to check their validity.

Intuitively, the Globally-completes-before order can be seen as the order in which events are recorded by a global checker which collects events after they have exited the memory hierarchy fully.

In the External global completion point of view, to determine whether an execution is valid or not, we proceed as follows:

- we build a relation called `IM0`, in which the initial writes are before any other access to the same location, and all accesses to a given location are before the final write to that location;
- per thread, we build the Locally-ordered-before relation, by looking for dependencies, fences, atomics and exclusives (see definition of `lob` in Figure 12);
- we then restrict it as follows:
  - we keep all pairs of accesses in `lob` where the first access is a write;
  - if the first access is a read r, we keep the pair when:
    - either r reads from another thread,
    - or r reads from a write w on the same thread, which is also locally-ordered-before the second access of the pair.
- we then try to build a total order called Globally-completes-before (`gcb`) over all accesses, which respects both `IM0` and this restricted Locally-ordered-before relation built as above.

The construction of which Locally-ordered-before pairs to keep ordered appears considerably more involved in the case of pairs headed by a read. However, this cost is paid up-front, and removes the need for an a posteriori check on the validity of reads, as would be the case in the second formulation. This cost a priori translates into a requirement to build traces where the reads which are picked are valid by design, hence the verification of those traces is less involved. The trade-off

```
(* External global completion requirement *)
let IM0 = loc & ((IW * (M\IW)) | ((W\FW) * FW))
let gc-req = (W * _) | ((R * _) & ((range(rfe) * _) | (rfi^-1; lob)))
let preorder-gcb = IM0 | lob & gc-req

with gcb from linearisations(M, preorder-gcb)
~empty gcb

let rf-gcb = (W * R) & loc & gcb \ intervening-write(gcb & loc)
let co-gcb = (W * W) & loc & gcb

call equal(rf, rf-gcb)
call equal(co, co-gcb)
```

Fig. 32. External global completion requirement

is really a matter of what the model is used for: for verification engineers, this third model permits easier verification algorithms.

This leads to the execution witness given in Figure 29-(c). We build the Locally-ordered-before relation as above, and restrict it as follows:

- there are no Locally-ordered-before pairs which start with a write;
- the pair (b, c) is in Locally-ordered-before. However, the write a from which b reads is not Locally-ordered-before c; hence we do not keep the pair (b, c).
- the pair (d, e) is in Locally-ordered-before, and the read d reads from another thread. Therefore we keep the pair (d, e)

We then try to build a total order which embeds this restricted Locally-ordered-before relation. The order c, d, e, a, b satisfies those constraints.

If such a Globally-completes-before order exists, we then justify the value taken by each read access r, as follows:

- r reads from its closest preceding write to the same location in the Globally-completes-before order. For example in Figure 29-(c), this corresponds to $w = a$ and $r = b$, as well as $w = c$ and $r = d$.
- if no such write exists, then r reads from the initial value of the memory location, as is the case in Figure 29-(c) for $r = e$.

We also need to ensure that the coherence order can be obtained from the Globally-completes-before order: as above, the coherence order is the projection of the Globally-completes-before order onto writes to the same location.

The cat formulation of the External global completion requirement also makes use of the cat constructs `with` and `linearisations`, much like the External completion requirement.

For completeness we give the English transliteration of the External global completion requirement in Figure 45.

6.5 Extending the alternative formulations to mixed-size accesses

Adding mixed-size accesses to those alternative formulations was challenging because we wanted to respect the design principles outlined in our introduction. Specifically, the principle given in Section 2.4 requires us to enable per-thread reasoning, or more precisely to keep the Completes-before and Globally-complete-before orders linearisations of `lob` or extensions thereof.

```
let ER = range(rfe)
let IR = range(rfi)
let rfisw = rfi^-1;si;rfi

let erln = (si & (W*W) |
            si & (ER * ER) |
            si & (R*R) & rfisw)+

let MC = classes(erln)
let scaob = si & (ER*IR)
```

Fig. 33. Mixed-size equivalence classes and Single-copy-atomic-ordered-before relation

Now, observe that our extension of the External visibility requirement for mixed-size accesses augments the Observed-by relation with the `si` relation. In other words, this is precisely not a local extension: rather the way we handle mixed-size accesses is by altering the interactions over a same variable, in particular the interactions between threads as defined by Observed-before.

To tackle this challenge, we proceeded as follows: instead of ordering sole events, we order equivalence classes of events.

6.5.1 Certain events are equivalent w.r.t. mixed-size orderings. Intuitively, the Arm mixed-sized formulation of the External visibility requirement considers that certain orderings towards a given event extend to the other events generated by the same instruction. For example writes generated by the same instruction are equivalent: if two writes w_1 and w_2 are such that w_2 is coherence-after w_1, all the events generated by the same instruction as w_2 are coherence-after w_1. This is because:

- the sequence `lws; si` is included in `lob` (see Figure 23)
- the sequence `obs; si` (and therefore `coe; si`) is included in `ob` (see Figure 24).

Therefore we build our equivalence classes as restrictions over events generated by the same instruction, viz, as restrictions over the relation `si`, and order those classes of events following a total order which linearises `lob` or an extension therefore: this formal trick allowed us to satisfy our design principles.

We now detail those formal constructions, given in Figure 33. The cat language already provides the notion of equivalence classes, via the keyword `classes`. Thus we build the set `MC` ("memory classes") of equivalence classes from an equivalence relation `erln`. Defining `ER` to be the set of reads reading from external writes and `IR` to be the set of reads reading from internal writes, we build the equivalence relation `erln`, as the union of pairs of the following events:

- writes generated by the same instruction, viz, `si & (W*W)`
- reads generated by the same instruction which both read from external writes, viz, `si & (ER*ER)`
- reads generated by the same instruction which both read from internal writes generated by the same instruction, viz, `si & (R*R) & rfisw`.

Note that we do not include pairs of reads where both reads read internally, but from writes which are not generated by the same instruction. To see why, consider the test given in Figure 34, its execution being given in Figure 35.

In this test, we are asking the following question: if the load of variable `y` on `P1` observes the update made by `P0`, can the load of variable `x` at the bottom of `P1` observe the writes of `x` and `x4+`

```
AArch64 MP+dmb.syw4w0+dataw0w0-rfiw0q0+RFI00
{
uint64_t y; uint64_t x; uint64_t 1:X5; uint64_t 1:X0;
0:X0=0x2020202; 0:X1=x; 0:X2=0x1010101; 0:X3=y;
1:X1=y; 1:X2=0x1010101; 1:X4=x; 1:X9=0x3030303;
}
P0             | P1             ;
               | STR W9,[X4,#4] ;
               | DMB SY         ;
STR W0,[X1,#4] | LDR W0,[X1]    ;
DMB SY         | EOR X3,X0,X0   ;
STR W2,[X3]    | ADD W3,W3,W2   ;
               | STR W3,[X4]    ;
               | LDR X5,[X4]    ;
exists(x=0x202020201010101 /\ 1:X5=0x303030301010101 /\ 1:X0=0x1010101)
```

Fig. 34. A test which justifies why internal reads reading from different writes are not equivalent

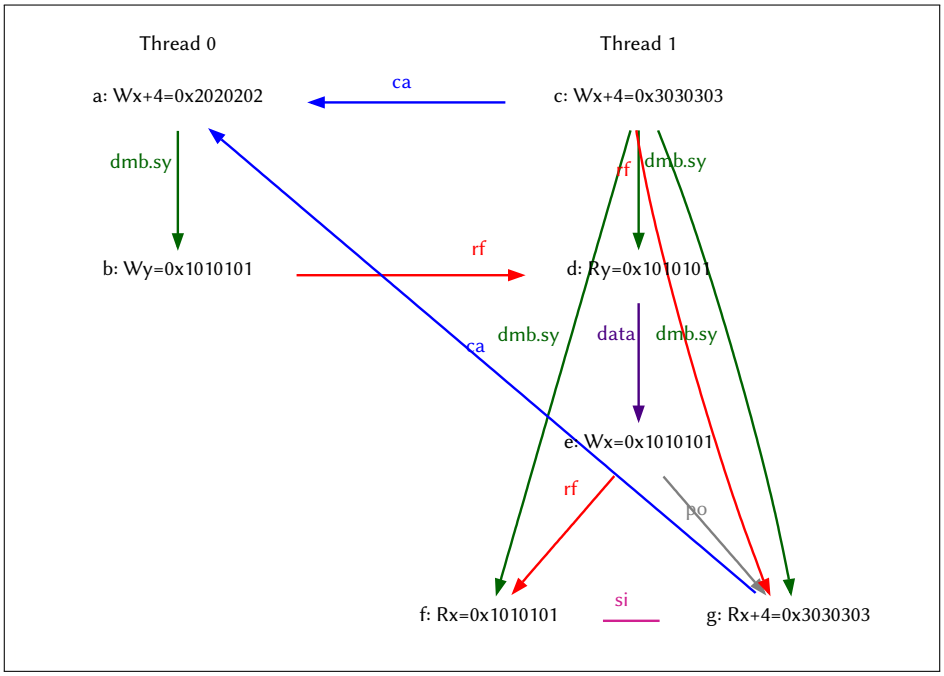


Fig. 35. Execution witness for the candidate of Figure 34

made by P1 in each half respectively? In other words, are both halves of the load of x by P1 ordered either with respect to one another, or both together with respect to the rest of the events?

This test is observed on hardware, which confirms that those two reads cannot be considered equivalent from an ordering point of view.

Single-copy-atomic-ordered-before

A read (R1) is single-copy-atomic-order-before another read (R2) if and only if all the following statements are true:

- R1 and R2 are reads generated by the same instruction,
- R1 is not a local-read-successor of a write,
- R2 is a local-read-successor of a write.

Fig. 36. English transliteration of the definition of the Single-copy-atomic-ordered-before relation

6.5.2 Single-copy-atomic-ordered-before relation. The cat language did not provide the ability to build linearisations over classes; we extended it to do so. Now, our first attempt in designing a mixed-size extension to our alternative formulations was simply to linearise the same relation (i.e. `lob` for the External completion requirement and `lob & gc-req` for the External global completion requirement), but over the classes `MC` instead of the set of events as in the non-mixed-size case.

This does not quite work. To see why, consider the test given in Figure 13. In this case, the `lob` relation is empty, hence there is nothing to order if we linearise only following `lob`. However, we can learn from how the External visibility requirement forbids the outcome of the test of Figure 13.

The mixed-size extension of the External visibility requirement (see Figures 23 and 24) ensures that the read b is ordered-before the write d because b is coherence-before d, but also that b is ordered-before the write e because e is generated by the same instruction as d. We handle this by treating the two writes d and e to be equivalent (see definition of the `erln` relation in Figure 33).

The mixed-size extension of the External visibility requirement ensures that the write e is ordered-before the read c, but also before the read b which is generated by the same instruction as c. Therefore the way the External visibility requirement rules this behaviour out is by ordering the read c before the read b, so that the following cycle is formed: d, e, c, b.

We generalise this analysis to the definition of the Single-copy-atomic-ordered-before relation `scaob`: we give its cat definition in Figure 33 and its English transliteration in Figure 36.

6.5.3 From events to equivalence classes of events and vice-versa. We also added two primitives to the cat language which allow a user to navigate from events to their classes and back. We therefore introduced two new primitives:

- `lift`: given a relation r over events and a set C of equivalence classes, `lift` returns the relation lifted to the classes. Formally, `lift(C,r)` returns $\{(C_1, C_2) \mid C_1 \in C \wedge C_2 \in C \wedge \exists e_1 \in C_1 \wedge \exists e_2 \in C_2 \wedge (e_1, e_2) \in r\}$.
- `delift`: given a relation over classes of events and a set of events, `delift` returns the corresponding relation over the set of events. Formally, `delift(s,R)` returns $\{(e_1, e_2) \mid e_1 \in s \wedge e_2 \in s \wedge \exists C_1, e_1 \in C_1 \wedge \exists C_2, e_2 \in C_2 \wedge (C_1, C_2) \in R\}$.

We use those primitives in the mixed-size formulations of the External completion and External global completion requirements, to ensure that the reads-from relation and the coherence order are valid. We now move on to exposing the details of those definitions.

6.5.4 Mixed-size external completion. The cat definition of the mixed-size External completion requirement is given in Figure 37, and its English transliteration as appears in the Arm documentation is given in Figure 46.

Much like in the non-mixed case, we define a relation called `preorder-cb`, which contains `lob`. To handle the mixed-size case (see discussion of the test of Figure 13), we also add the relation `scaob` to

```
(* External completion requirement, with mixed-size *)
let IM0 = loc & ((IW * (M\IW)) | ((W\FW) * FW))
let preorder-cb = IM0 | lob | scaob
let preorder-cb-lift = lift(MC,preorder-cb)

with cb from linearisations(MC,preorder-cb-lift)
~empty cb

let dcb = delift(M,cb) & loc
let rf-fwd = (W*R) & po-loc & dcb^-1 \ intervening-write(po-loc)
let rf-nfwd = (W*R) & dcb \ intervening-write(po-loc | dcb)
let rf-cb = rf-fwd | rf-nfwd
let co-cb = dcb & (W*W)

call equal(rf, rf-cb) as rfeq
call equal(co, co-cb) as cbeq
```

Fig. 37. Mixed-size external completion requirement

the preorder-cb relation. Then we lift the preorder-cb to the level of the equivalence classes MC (as defined in Figure refcat:classes), resulting in a relation called preorder-cb-lift. We then linearise this preorder-cb-lift relation, which gives us a total order which we call cb.

To check the validity of the reads-from relation and the coherence order, we project the total order cb from classes down to the level of events, and restrict the resulting relation to events relative to the same location using the relation loc. We then check that the reads-from relation and coherence order are valid as before.

6.5.5 Mixed-size global external completion. The cat definition of the mixed-size Global external completion requirement is given in Figure 38, and its English transliteration as appears in the Arm documentation is given in Figure 47.

Much like in the non-mixed case, we define a relation called preorder-gcb, which contains lob & gc-req. To handle the mixed-size case (see discussion of the test of Figure 13), we also add the relation scaob to the preorder-gcb relation. Then we lift the preorder-gcb to the level of the equivalence classes MC (as defined in Figure refcat:classes), resulting in a relation called preorder-gcb-lift. We then linearise this preorder-gcb-lift relation, which gives us a total order which we call gcb.

To check the validity of the reads-from relation and the coherence order, we project the total order gcb from classes down to the level of events, and restrict the resulting relation to events relative to the same location using the relation loc. We then check that the reads-from relation and coherence order are valid as before.

7 ALL THREE FORMULATIONS OF THE MODEL ARE EQUIVALENT

In this section we give the formal statements of the equivalence theorems, and a high-level overview of the main arguments for each proof. The formal proofs have been done by Viktor Vafeiadis using the Coq proof assistant and can be found online [42].

Here we give those proof sketches for the non-mixed-size models. We give the mixed-size proofs in Appendix P.

```
(* External global completion requirement, with mixed-size *)
let IM0 = loc & ((IW * (M\IW)) | ((W\FW) * FW))
let gc-req = (W * _) | (R * _) & ((range(rfe) * _) | (rfi^-1; lob))
let preorder-gcb = IM0 | lob & gc-req | scaob
let preorder-gcb-lift = lift(MC,preorder-gcb)

with gcb from linearisations(MC, preorder-gcb-lift)
~empty gcb

let dgcb = delift(M,gcb) & loc
let rf-gcb = (W * R) & dgcb \ intervening-write(dgcb)
let co-gcb = (W * W) & dgcb

call equal(rf, rf-gcb)
call equal(co, co-gcb)
```

Fig. 38. Mixed-size external global completion requirement

To start with, observe that all three models follow the same structure: Internal visibility, Atomicity, and a third axiom. Therefore proving their equivalence amounts to proving that their third axioms are equivalent.

7.1 External visibility and External global completion equivalence

Theorem 7.1 (External visibility and External global completion equivalence). *For all execution witness, such that:*

- *its relations* `rf`, `co`, `si` *and* `lob` *are well-formed*
- *the Internal Visibility requirement is satisfied*

then:

- *if the External Visibility requirement is satisfied, then there exists a global-completes-before order compatible with that execution witness which satisfies the External global completion requirement;*
- *if there exists a global-completes-before order compatible with that execution witness which satisfies the External global completion requirement, then that execution witness satisfies the External Visibility requirement.*

7.1.1 From External visibility to External global completion. The structure of the proof is as follows. We build a relation called `pre_egc` which consists of the transitive closure of the union of the following relations: `rf`, `ca` and `preorder-gcb`. We linearise that relation (which requires showing that `pre_egc` is a partial order). We show that the resulting total order is a qualifying globally-completes-before order.

The relation pre_egc is a partial order. The relation `pre_egc` is evidently transitive as it is defined to be a transitive closure. The fact that it is irreflexive is less obvious, and we prove it by contradiction.

Observe that the relation `pre_egc` can be decomposed into the following relations:

- the reflexive-transitive closure of the union of `obs+` and `preorder-gcb`, all this sequenced with `rfi`
- the transitive closure of `obs`

- the reflexive transitive closure of the sequence of obs* with preorder-gcb, itself sequenced with obs+ again.

Now, suppose that pre_egc is not irreflexive: then there would exists an event x such that (x, x) is in pre_egc. This means that one of the three relations listed above is reflexive. The second two relations cannot be reflexive as this would be a direct contradiction of the External Visibility requirement. Suppose the first one is reflexive: there exists x such that (x, x) is either in (obs+ | preorder-gcb)+, a contradiction of External Visibility, or in rfi, a contradiction because the extremities of that relation are a write at the source and a read at the target.

A linearisation of pre_egc is a qualifying globally-completes-before order. The fact that a linearisation of the relation pre_egc is a total order is a triviality because a linearisation is a total order by design. Ensuring that the reads-from and coherence order built from that total order are valid relies on the fact that the original reads-from relation and coherence order are well-formed.

7.1.2 From External global completion to External Visibility. We reason again by contradiction. Given the existence of a total globally-completes-before order, suppose that the execution witness does not satisfy the External Visibility requirement. This means that there exists x such that (x, x) is in ob. Much like in the previous case, we find a way to decompose ob in various subrelations:

- obs+
- the transitive closure of the sequence of obs+ and preorder-gcb, itself sequenced with obs+
- the reflexive closure of that same relation, preceded by lob \ gc-req
- lob \ gc-req followed by the reflexive-transitive closure of obs.

The first relation cannot be reflexive, as it would contradict the Internal Visibility requirement. The second relation cannot be reflexive, as it would contradict the External Visibility requirement. If either of the last two relations was reflexive, then either so would be lob gc-req, a contradiction since lob is included in po hence cannot be reflexive, or so would be the transitive closure of the sequence of obs* and preorder-gcb, itself sequenced with obs+. We know that last relation cannot be reflexive as this would contradict External Visibility.

7.2 External visibility and External completion equivalence

Theorem 7.2 (External visibility and External completion equivalence). *For all execution witness, such that:*

- *its relations* rf, co, *and* lob *are well-formed*
- *the Internal Visibility requirement is satisfied*

then:

- *if the External Visibility requirement is satisfied, then there exists a completes-before order compatible with that execution witness which satisfies the External completion requirement;*
- *if there exists a completes-before order compatible with that execution witness which satisfies the External completion requirement, then that execution witness satisfies the External Visibility requirement.*

7.2.1 From External visibility to External completion. The structure of the proof is as follows. We build a relation called pre_ec which consists of the transitive closure of the union of the following relations: rfe, ca and lob. We linearise that relation (which requires showing that pre_ec is a partial order). We show that the resulting total order is a qualifying completes-before order.

The relation pre_ec is a partial order. The relation pre_ec is evidently transitive as it is defined to be a transitive closure. The fact that it is irreflexive is less obvious, and we prove it by contradiction.

Observe that the relation pre_ec can be decomposed into:

- the transitive closure of obs
- the transitive closure of the sequence of obs* with lob, itself sequenced with obs* again.

Now, suppose that pre_ec is not irreflexive: then there would exists an event x such that (x, x) is in pre_ec. This means that one of the two relations above is reflexive.

The first relation being reflexive is a contradiction of the Internal Visibility requirement. The second relation being reflexive contradicts the External visibility requirement.

A linearisation of the lift of pre_ec is a qualifying globally-completes-before order. The fact that a linearisation of the relation pre_ec is a total order is a triviality because a linearisation is a total order by design. Ensuring that the reads-from and coherence order built from that total order are valid relies on the fact that the original reads-from relation and coherence order are well-formed.

7.2.2 From External completion to External visibility. We reason again by contradiction. Given the existence of a total completes-before order, suppose that the execution witness does not satisfy the External Visibility requirement. This means that there exists x such that (x, x) is in ob.

Much like in the previous case, we find a way to decompose ob in various subrelations:

- obs+
- the transitive closure of the sequence of obs* and lob; obs*

The first relation cannot be reflexive, as it would contradict the Internal Visibility requirement. The second relation cannot be reflexive, as it would be a contradiction of the External Visibility requirement.

7.3 Application to x86

Interestingly, the fact that our equivalence proofs are parametric in lob means that those alternative formulations apply to x86 as well. In other words this means that our work also offers two new ways of looking at modelling and verifying x86 designs, including mixed-size accesses.

We distribute those alternative formulations for x86 within the herd+diy distribution [9].

8 CONCLUSION: A WORK IN PROGRESS

We have given a review of the process for formalising the concurrency aspects of the Arm architecture, and given details about the design principles and rationale behind the Arm consistency model.

Needless to say, the design, maintenance and extension of such a model is work in progress. At the time of writing, several extensions are in the works internally at Arm and waiting to be upstreamed.

We do hope however that this paper and its accompanying material gives the community the indication that this investment in formal modelling is in fact a commitment. We very much hope for those formal models to be helpful foundations or stepping stones towards achieving greater safety and security goals, as highlighted in the introduction.

Acknowledgements. We thank Azalea Raad and James Riely for comments on a draft.

A COMMON DEFINITIONS (ARMV8-COMMON.CAT)

```
let IM0 = loc & ((IW * (M\IW)) | ((W\FW) * FW))

(* Coherence-after *)
let ca = fr | co

(* Local read successor *)
let lrs = [W]; po-loc \ intervening-write(po-loc); [R]

(* Local write successor *)
let lws = po-loc; [W]

(* Observed-by *)
let obs = rfe | fre | coe

(* Read-modify-write *)
let rmw = lxsx | amo

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

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

(* Barrier-ordered-before *)
let bob = po; [dmb.full]; po
| po; ([A];amo;[L]); po
| [L]; po; [A]
| [R]; po; [dmb.ld]; po
| [A | Q]; po
| [W]; po; [dmb.st]; po; [W]
| po; [L]

(* Tag-ordered-before *)
let tob = [R & T]; intrinsic; [M \ T]

(* Locally-ordered-before *)
let rec lob = if "Mixed" then lws; si else lws
| dob
| aob
| bob
| tob
```

```
| lob; lob

(* Single-copy-atomic-ordered-before *)
let ER = range(rfe)
let IR = range(rfi)
let rfisw = rfi^-1;si;rfi
let erln = (si & (W*W) | si & (ER * ER) | si & (R*R) & rfisw)+
let MC = classes(erln)
let scaob = si & (ER*IR)

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

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

B ORIGINAL ARMV8 MODEL WITHOUT MIXED-SIZE

```
(* Ordered-before *)
let rec ob = obs
        | lob
        | ob; ob

(* External visibility requirement *)
irreflexive ob as external
```

C MIXED-SIZE EXTENSION OF THE ORIGINAL ARMV8 MODEL

```
(* Ordered-before *)
let rec ob = obs; si
        | lob
        | ob; ob

(* External visibility requirement *)
irreflexive ob as external
```

D EXTERNAL COMPLETION MODEL WITHOUT MIXED-SIZE

```
let preorder-cb = IM0 | lob
with cb from linearisations(M,preorder-cb)
~empty cb

let rf-fwd = (W*R) & po-loc & (cb & loc)^-1 \ intervening-write(po-loc)
let rf-nfwd = (W*R) & (cb & loc) \ intervening-write(cb & loc | po-loc)
let rf-cb = rf-fwd | rf-nfwd
let co-cb = (cb & loc) & (W*W)

call equal(rf, rf-cb) as rfeq
call equal(co, co-cb) as cbeq
```

E MIXED-SIZE EXTENSION OF THE EXTERNAL COMPLETION MODEL

```
let preorder-cb = IM0 | lob | scaob
let preorder-cb-lift = lift(MC,preorder-cb)
with cb from linearisations(MC,preorder-cb-lift)
~empty cb

let rf-fwd = (W*R) & po-loc & (delift(cb) & loc)^-1 \ intervening-write(po-loc)
let rf-nfwd = (W*R) & (delift(cb) & loc) \ intervening-write(delift(cb) & loc | po-loc)
let rf-cb = rf-fwd | rf-nfwd
let co-cb = (delift(cb) & loc) & (W*W)

call equal(rf, rf-cb) as rfeq
call equal(co, co-cb) as cbeq
```

F EXTERNAL GLOBAL COMPLETION MODEL WITHOUT MIXED-SIZE

```
let gc-req = (W * _) | (R * _) & ((range(rfe) * _) | (rfi^-1; lob))
let preorder-gcb = IM0 | lob & gc-req
with gcb from linearisations(M, preorder-gcb)
~empty gcb

let rf-gcb = (W * R) & (gcb & loc) \ intervening-write(gcb & loc)
let co-gcb = (W * W) & (gcb & loc)

call equal(rf, rf-gcb)
call equal(co, co-gcb)
```

G MIXED-SIZE EXTENSION OF THE EXTERNAL GLOBAL COMPLETION MODEL

```
let gc-req = (W * _) | (R * _) & ((range(rfe) * _) | (rfi^-1; lob))
let preorder-gcb = IM0 | lob & gc-req | scaob
let preorder-gcb-lift = lift(MC,preorder-gcb)
with gcb from linearisations(MC, preorder-gcb-lift)
~empty gcb

let rf-gcb = (W * R) & (delift(gcb) & loc) \ intervening-write(delift(gcb) & loc)
let co-gcb = (W * W) & (delift(gcb) & loc)

call equal(rf, rf-gcb)
call equal(co, co-gcb)
```

H ENGLISH TRANSLITERATION OF ARM BASIC TERMINOLOGY

Memory effects on a Location are related by the following relations:

Reads-from

A Reads-from relation that couples reads and writes to the same Location such that each read is paired with a single write in the program. A read R2 of a Location Reads-from a write W1 to the same Location if and only if R2 takes its data from W1.

Coherence order

A Coherence order relation for each Location in the program that provides a total order on all writes from all coherent Observers to that Location, starting with a notional write of the initial value.

Fig. 39. English transliteration of notions fundamental to cat models

Local read successor

A read R2 of a Location is the Local read successor of a write W1 from the same Observer to the same Location if and only if W1 appears in program order before R2 and there is not a write W3 from the same Observer to the same Location appearing in program order between W1 and R2.

Local write successor

A write W2 of a Location is a Local write successor of a write W1 from the same Observer to the same Location if and only if W1 appears in program order before W2.

Coherence-after

A write W2 to a Location is Coherence-after another write W1 to the same Location if and only if W2 is sequenced after W1 in the Coherence order of the Location.
A write W2 to a Location is Coherence-after a read R1 of the same location if and only if R1 Reads-from a write W3 to the same Location and W2 is Coherence-after W3.

Observed-by

A read or a write RW1 from an Observer is Observed-by a write W2 from a different Observer if and only if W2 is coherence-after RW1.
A write W1 from an Observer is Observed-by a read R2 from a different Observer if and only if R2 Reads-from W1.

Fig. 40. English transliteration of basic Arm terminology

I ENGLISH TRANSLITERATION OF ARM LOCALLY-ORDERED-BEFORE RELATION - NON MIXED-SIZE

Dependency-ordered-before

A dependency creates externally-visible order between a read and another Memory effect generated by the same observer. A read R1 is Dependency-ordered-before a read or a write RW2 from the same observer if and only if R1 appears in program order before RW2 and any of the following cases apply:

- There is an Address or a Data dependency from R1 to RW2
- RW2 is a write W2 and there is a Control dependency from R1 to W2
- RW2 is a read R2 appearing in program order after a Context synchronization event CSE3, and there is a Control dependency from R1 to CSE3
- RW2 is a write W2 appearing in program order after a read or a write RW3 and there is an Address dependency from R1 to RW3
- RW2 is a read R2 that is a Local read successor of a write W3 and there is an Address or a Data dependency from R1 to W3.

Atomic-ordered-before

Load-Exclusive and Store-Exclusive instructions provide some ordering guarantees, even in the absence of dependencies. A read or a write RW1 is Atomic-ordered-before a read or a write RW2 from the same Observer if and only if RW1 appears in program order before RW2 and either of the following cases apply:

- RW1 is a read R1 and RW2 is a write W2 such that R1 and W2 are generated by an atomic instruction or a successful Load-Exclusive/Store-Exclusive instruction pair to the same Location.
- RW1 is a write W1 generated by an atomic instruction or a successful Store-Exclusive instruction and RW2 is a read R2 generated by an instruction with Acquire or AcquirePC semantics such that R2 is a Local read successor of W1.

Barrier-ordered-before

Barrier instructions order prior Memory effects before subsequent Memory effects generated by the same Observer. A read or a write RW1 is Barrier-ordered-before a read or a write RW2 from the same Observer if and only if RW1 appears in program order before RW2 and any of the following cases apply:

- RW1 appears in program order before a DMB FULL that appears in program order before RW2.
- At least one of RW1 and RW2 is generated by an atomic instruction with both Acquire and Release semantics.
- RW1 is a write W1 generated by an instruction with Release semantics and RW2 is a read R2 generated by an instruction with Acquire semantics
- RW1 is a read R1 and either:
  - R1 appears in program order before a DMB LD that appears in program order before RW2, or
  - R1 is generated by an instruction with Acquire or AcquirePC semantics
- RW2 is a write W2 and either:
  - RW1 is a write W1 appearing in program order before a DMB ST that appears in program order before W2,
  - or W2 is generated by an instruction with Release semantics.

Locally-ordered-before

Dependencies, Local write successor, Load/Store-Exclusive, atomic and barrier instructions can be composed within an observer to create externally-visible order. A read or a write RW1 is locally-ordered-before a read or a write RW2 from the same observer if and only if any of the following cases apply:

J ENGLISH TRANSLITERATION OF THE EXTERNAL COMPLETION REQUIREMENT - NON MIXED-SIZE

The Completes-before order is a total order that corresponds to the order in which memory effects complete within the system.

Deriving Reads-from from the Completes-before order

The Completes-before order can be used to resolve the Reads-from relation for every memory access in the system as follows. For a read R1 of a memory location by an observer, then:

- If there is a write W2 to the same Location from the same Observer and all of the following are true:
  - W2 appears in program order before R1
  - R1 Completes-before W2
  - There are no writes to the Location appearing in program order between W2 and R1 then R1 reads-from W2.
- Otherwise, R1 reads-from its closest preceding write to the same location in the Completes-before order. If no such write exists, then R1 reads-from the initial value of the memory location.

Deriving Coherence order from the Completes-before order

The Completes-before order can be used to resolve the Coherence order relation for every memory access in the system as follows:
The Coherence order of writes to a memory location is the order in which those writes appear in the Completes-before order. The final value of each memory location is therefore determined by the final write to each Location in the Completes-before order. If no such write exists for a given location, the final value is the initial value of that Location.

External completion requirement

The Completes-before order is a total order over memory effects, such that: for a read or a write RW1 that is Locally-ordered-before a read or a write RW2, the external completion requirement requires that RW1 Completes-before RW2.

Fig. 42. English transliteration of the External completion requirement

K ENGLISH TRANSLITERATION OF ARM LOCALLY-ORDERED-BEFORE - MIXED-SIZE

Locally-ordered-before

Dependencies, Local write successor, Load/Store-Exclusive, atomic and barrier instructions can be composed within an Observer to create externally-visible order. A read or write RW1 is Locally-ordered-before a read or write RW2 from the same Observer if and only if any of the following apply:

- RW1 is a write W1 and RW2 is a write W2 that is equal to or generated by the same instruction as a Local write successor of RW1.
- RW1 is Dependency-ordered-before RW2.
- RW1 is Atomic-ordered-before RW2.
- RW1 is Barrier-ordered-before RW2.
- RW1 is Locally-ordered-before a read or a write that is Locally-ordered-before RW2.

Fig. 43. English transliteration of the new Arm Locally-ordered-before relation

L ENGLISH TRANSLITERATION OF ARM ORDERED-BEFORE - MIXED-SIZE

Ordered-before

An arbitrary pair of Memory effects is ordered if it can be linked by a chain of ordered accesses consistent with external observation. A read or a write RW1 is Ordered-before a read or a write RW2 if and only if any of the following cases apply:

- RW1 is Observed-by a read or write RW3 that is equal to or generated by the same instruction as RW2;
- RW1 is Locally-ordered-before RW2;
- RW1 is Ordered-before a read or write that is Ordered-before RW2.

Fig. 44. English transliteration of the new Arm Ordered-before relation

M ENGLISH TRANSLITERATION OF THE EXTERNAL GLOBAL COMPLETION REQUIREMENT - NON MIXED-SIZE

The Globally-completes-before order is a total order that corresponds to the order in which memory effects globally-complete within the system.

Deriving Reads-from and Coherence order from the Globally-completes-before order

The Globally-completes-before order can be used to resolve the Reads-from and Coherence order relations for every memory access in the system as follows:

- A read R1 of a memory location by an Observer Reads-from its closest preceding write to the same Location in the Globally-completes-before order. If no such write exists, then R1 Reads-from the initial value of the memory location.
- The Coherence order of writes to a memory location is the order in which those writes appear in the Globally-completes-before order. The final value of each memory location is therefore determined by the final write to each Location in the Globally-completes-before order. If no such write exists for a given Location, the final value is the initial value of that Location.

External global completion requirement

For a read or a write RW1 that is Locally-ordered-before a read or a write RW2, the external global completion requirement requires that RW1 Globally-completes-before RW2 if and only if any of the following statements are true:

- RW1 is a write
- RW1 is a read R1 and either:
  - R1 does not Locally-reads-from a write, or
  - R1 Locally-reads-from a write that is Locally-ordered-before RW2

Fig. 45. English transliteration of the External global completion requirement

N ENGLISH TRANSLITERATION OF THE EXTERNAL COMPLETION REQUIREMENT - MIXED-SIZE

The Completes-before order is a total order that corresponds to the order in which memory effects complete within the system. The following effects constitute a single entry in the Completes-before order:

- writes from the same instruction
- reads from the same instruction which read from external writes
- reads from the same instruction which read from the same internal write.

All other reads constitute distinct entries in the Completes-before order.

External completion requirement

A read or a write RW1 Completes-before a read or a write RW2 if and only if any of the following statements are true:

- RW1 is Locally-ordered-before RW2.
- RW1 is a read R1 and RW2 is a read R2 and R1 is Single-copy-atomic-ordered-before R2.

Fig. 46. English transliteration of the mixed-size External completion requirement

O ENGLISH TRANSLITERATION OF THE EXTERNAL GLOBAL COMPLETION REQUIREMENT - MIXED-SIZE

The Globally-completes-before order is a total order that corresponds to the order in which memory effects globally-complete within the system. The following effects constitute a single entry in the Globally-completes-before order:

- writes from the same instruction
- reads from the same instruction which read from external writes
- reads from the same instruction which read from the same internal write.

All other reads constitute distinct entries in the Globally-completes-before order.

External global completion requirement

For a read or a write RW1 that is Locally-ordered-before a read or a write RW2, the external global completion requirement requires that RW1 Globally-completes-before RW2 if and only if any of the following statements are true:

- RW1 is Locally-ordered-before RW2 and either:
  - RW1 is a write.
  - RW1 is a read R1 and either:
    - R1 is not a Local read successor of a write.
    - R1 is a Local read successor of a write that is Locally-ordered-before RW2.
- RW1 is a read R1 and RW2 is a read R2 and R1 is Single-copy-atomic-ordered-before R2.

Fig. 47. English transliteration of the mixed-size External global completion requirement

P ALL THREE FORMULATIONS OF THE MIXED-SIZE MODEL ARE EQUIVALENT

In this section we give the formal statements of the equivalence theorems, and a high-level overview of the main arguments for each proof. The formal proofs have been done by Viktor Vafeiadis using the Coq proof assistant and can be found online [42].

To start with, observe that all three models follow the same structure: Internal visibility, Atomicity, and a third axiom. Therefore proving their equivalence amounts to proving that their third axioms are equivalent.

P.1 External visibility and External global completion equivalence

Theorem P.1 (External visibility and External global completion equivalence). *For all execution witness, such that:*

- *its relations* `rf`, `co`, `si` *and* `lob` *are well-formed*
- *the Internal Visibility requirement is satisfied*

then:

- *if the External Visibility requirement is satisfied, then there exists a global-completes-before order compatible with that execution witness which satisfies the External global completion requirement;*
- *if there exists a global-completes-before order compatible with that execution witness which satisfies the External global completion requirement, then that execution witness satisfies the External Visibility requirement.*

P.1.1 From External visibility to External global completion. The structure of the proof is as follows. We build a relation called pre_egc which consists of the transitive closure of the union of the following relations: rf;si, ca;si and preorder-gcb. We lift that relation to the level of classes and then linearise the result (which requires showing that pre_egc is a partial order). We show that the resulting total order is a qualifying globally-completes-before order.

The relation pre_egc is a partial order. The relation pre_egc is evidently transitive as it is defined to be a transitive closure. The fact that it is irreflexive is less obvious, and we prove it by contradiction.

Observe that the relation pre_egc can be decomposed into the following relations:

- the reflexive-transitive closure of the union of erln;obs+;erln and erln; preorder-gcb; erln, all this sequenced with rfi; erln
- the transitive closure of erln; obs+; erln
- the reflexive transitive closure of the sequence of (erln; obs+; erln)* with erln;preorder-gcb; erln, itself sequenced with erln; obs+; erln again.

Now, suppose that pre_egc is not irreflexive: then there would exists an event x such that (x, x) is in pre_egc. This means that one of the three relations listed above is reflexive. The second two relations cannot be reflexive as this would be a direct contradiction of the External Visibility requirement. Suppose the first one is reflexive: there exists x such that (x, x) is in ((erln; obs+; erln ∪ erln; preorder-gcb; erln))*; (rfi; erln). Therefore there exists y such that (y, y) is in (rfi;erln); ((erln; obs+; erln ∪ erln; preorder-gcb; erln))*, which in turn means that (y, y) is either in (erln;obs+;erln | erln; preorder-gcb; erln)*, a contradiction of External Visibility, or in rfi; erln, a contradiction because the extremities of that relation are a write at the source and a read at the target.

A linearisation of the lift of pre_egc is a qualifying globally-completes-before order. The fact that a linearisation of the lift of the relation pre_egc is a total order is a triviality because a linearisation is a total order by design. Ensuring that the reads-from and coherence order built from that total order are valid relies on following the definition of delift and using the fact that the original reads-from relation and coherence order are well-formed.

P.1.2 From External global completion to External Visibility. We reason again by contradiction. Given the existence of a total globally-completes-before order, suppose that the execution witness does not satisfy the External Visibility requirement. This means that there exists x such that (x, x) is in ob.

Much like in the previous case, we find a way to decompose ob in various subrelations:

- (obs+; si)*
- the transitive closure of the sequence of (obs+; si)* and preorder-gcb, itself sequenced with obs+; si
- the reflexive closure of that same relation, preceded by lob\ gc-req
- lob\ gc-req followed by the reflexive-transitive closure of obs+; si.

The first relation cannot be reflexive, as it would contradict the Internal Visibility requirement. The second relation cannot be reflexive, as one can prove that if two events are related by that relation, then their respective classes are ordered in the globally-completes-before order under scrutiny. Therefore we would have a cycle (via the class of x) in our order, a contradiction.

If either of the last two relations was reflexive, then either so would be lob\ gc-req, a contradiction since lob is included in po hence cannot be reflexive, or so would be the transitive closure of the sequence of (obs+; si)* and preorder-gcb, itself sequenced with obs+; si. We know that last relation cannot be reflexive by the reasoning given in the previous paragraph.

P.2 External visibility and External completion equivalence

Theorem P.2 (External visibility and External completion equivalence). *For all execution witness, such that:*

- *its relations* rf, co, si *and* lob *are well-formed*
- *the Internal Visibility requirement is satisfied*

then:

- *if the External Visibility requirement is satisfied, then there exists a completes-before order compatible with that execution witness which satisfies the External completion requirement;*
- *if there exists a completes-before order compatible with that execution witness which satisfies the External completion requirement, then that execution witness satisfies the External Visibility requirement.*

P.2.1 From External visibility to External completion. The structure of the proof is as follows. We build a relation called pre_ec which consists of the transitive closure of the union of the following relations: rfe; erln, ca; erln and preorder-cb. We lift that relation to the level of classes and then linearise the result (which requires showing that pre_ec is a partial order). We show that the resulting total order is a qualifying completes-before order.

The relation pre_ec is a partial order. The relation pre_ec is evidently transitive as it is defined to be a transitive closure. The fact that it is irreflexive is less obvious, and we prove it by contradiction.

Observe that the relation pre_ec can be decomposed into:

- the transitive closure of the union of erln; obs+; erln
- the sequence of (erln; obs+; erln) with erln; preorder-cb; erln, itself sequenced with (erln; obs+; erln) again.

Now, suppose that pre_ec is not irreflexive: then there would exists an event x such that (x, x) is in pre_ec. This means that one of the two relations above is reflexive.

The first relation being reflexive is a contradiction of the Internal Visibility requirement. The second relation being reflexive contradicts the External visibility requirement.

A linearisation of the lift of pre_ec is a qualifying globally-completes-before order. The fact that a linearisation of the lift of the relation pre_ec is a total order is a triviality because a linearisation is a total order by design. Ensuring that the reads-from and coherence order built from that total order are valid relies on following the definition of delift and using the fact that the original reads-from relation and coherence order are well-formed.

P.2.2 From External completion to External visibility. Let us reason by contradiction once again: assume a cycle in ob. Then there also is a cycle in either:

- obs+; si, or
- ((obs+; si); preorder-cb; (obs+; si))+

The first one is a contradiction of the Internal Visibility requirement. The second one cannot be reflexive, as one can prove that if two events are related by that relation, then their respective classes are ordered in the completes-before order under scrutiny.

REFERENCES

[1] 2019.
[2] Jade Alglave. 2010. A Shared Memory Poetics. In *PhD thesis*.

[3] Jade Alglave. 2019. Adding mixed-size accesses to Arm formal memory model. https://github.com/herd/herdtools7/commit/95785c747750be4a3b64adfab9d5f5ee0ead8240#diff-cc249d0a9116e1ab890d8b52586bc702

[4] Jade Alglave. 2020. https://github.com/herd/herdtools7/blob/master/herd/libdir/x86tso-mixed.cat

[5] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 577–591.

[6] Jade Alglave and Patrick Cousot. 2017. Ogre and Pythia: an invariance proof method for weak consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 3–18.

[7] Jade Alglave, Patrick Cousot, and Luc Maranget. 2016. Syntax and semantics of the weak consistency model specification language cat. *CoRR* abs/1608.07531 (2016). http://arxiv.org/abs/1608.07531

[8] Jade Alglave, Anthony C. J. Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2009. The semantics of power and ARM multiprocessor machine code. In *Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009*, Leaf Petersen and Manuel M. T. Chakravarty (Eds.). ACM, 13–24.

[9] Jade Alglave and Luc Maranget. 2010-present. The herd+diy toolsuite. http://diy.inria.fr

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

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

[12] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests against Hardware. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science)*, Parosh Aziz Abdulla and K. Rustan M. Leino (Eds.), Vol. 6605. Springer, 41–44.

[13] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. Fences in weak memory models (extended version). *Formal Methods Syst. Des.* 40, 2 (2012), 170–205.

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

[15] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 55–66.

[16] Nathan Chong and Samin Ishtiaq. 2008. Reasoning About the ARM Weakly Consistent Memory Model. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (acm sigplan workshop on memory systems performance and correctness ed.).

[17] Nathan Chong, Tyler Sorensen, and John Wickerson. 2018. The semantics of transactions and weak memory in x86, Power, ARM, and C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 211–225.

[18] Will Deacon. 2017. Formal memory model for Armv8.0 application level. https://github.com/herd/herdtools7/commit/daa126680b6ecba97ba47b3e05bbaa51a89f27b7#diff-0461c726950c4454a08bd97fbfd49252

[19] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 608–621.

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

[21] Richard Gooch and Pekka Enberg. 2005. Linux Kernel Virtual File System (VFS). https://www.kernel.org/doc/html/latest/filesystems/vfs.html

[22] C. A. R. Hoare and Peter E. Lauer. 1974. Consistent and Complementary Formal Theories of the Semantics of Programming Languages. *Acta Inf.* 3 (1974), 135–153.
[23] Daniel Jackson. 2019. Alloy: a language and tool for exploring software designs. *Commun. ACM* 62, 9 (2019), 66–76.
[24] Antoine Hacquard Jade Alglave and Luc Maranget. [n.d.]. Mixed-size experiments on AArch64 and x86. ([n. d.]). http://diy.inria.fr/mixed/
[25] Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.* 4, OOPSLA.
[26] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189.
[27] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 618–632.
[28] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691.
[29] Arm Ltd. 2020. Arm Memory Model. https://developer.arm.com/architectures/cpu-architecture/a-profile/memory-model-tool
[30] Arm Ltd. 2020. Arm released cat models. https://github.com/herd/herdtools7/tree/master/herd/libdir/arm-models
[31] Arm Ltd. 2020. Armv8, for Armv8-A architecture profile. In *Arm Architecture Reference Manual*, Vol. ARM DDI 0487F.c. https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile
[32] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 257–270.
[33] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL* 2, POPL, 19:1–19:29.
[34] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1–15.
[35] Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak persistency semantics from the ground up: formalising the persistency semantics of ARMv8 and transactional models. *PACMPL* 3, OOPSLA, 135:1–135:27.
[36] RISC-V. 2019. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA. https://content.riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf
[37] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 311–322.
[38] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 175–186.
[39] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. 2009. The semantics of x86-CC multiprocessor machine code. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 379–391.
[40] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97.
[41] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. 2020. ARMv8-A system semantics: instruction fetch in relaxed architectures (extended version). In *29th European Symposium on Programming (ESOP 2020), April 2020.*
[42] Viktor Vafeiadis. 2021. https://github.com/vafeiadis/arm-model
[43] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. 2020. Repairing and mechanising the JavaScript relaxed memory model. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020,*

London, UK, June 15-20, 2020, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 346–361.