Isla: Integrating Full-Scale ISA Semantics and Axiomatic Concurrency Models

Alasdair Armstrong[1(✉)], Brian Campbell[2], Ben Simner[1], Christopher Pulte[1], and Peter Sewell[1]

[1] University of Cambridge, Cambridge, UK
alasdair.armstrong@cl.cam.ac.uk
[2] University of Edinburgh, Edinburgh, UK

Abstract. Architecture specifications such as Armv8-A and RISC-V are the ultimate foundation for software verification and the correctness criteria for hardware verification. They should define the allowed sequential and relaxed-memory concurrency behaviour of programs, but hitherto there has been no integration of full-scale instruction-set architecture (ISA) semantics with axiomatic concurrency models, either in mathematics or in tools. These ISA semantics can be surprisingly large and intricate, e.g. 100k+ lines for Armv8-A.

In this paper we present a tool, Isla, for computing the allowed behaviours of concurrent litmus tests with respect to full-scale ISA definitions, in Sail, and arbitrary axiomatic relaxed-memory concurrency models, in the Cat language. It is based on a generic symbolic engine for Sail ISA specifications, which should be valuable also for other verification tasks. We equip the tool with a web interface to make it widely accessible, and illustrate and evaluate it for Armv8-A and RISC-V.

By using full-scale and authoritative ISA semantics, this lets one evaluate litmus tests using arbitrary user instructions with high confidence. Moreover, because these ISA specifications give detailed and validated definitions of the sequential aspects of *systems* functionality, as used by hypervisors and operating systems, e.g. instruction fetch, exceptions, and address translation, our tool provides a basis for developing concurrency semantics for these. We demonstrate this for the Armv8-A instruction-fetch model and self-modifying code examples of Simner et al.

1 Introduction

A processor architecture should define, for any initial machine state, the set of all architecturally allowed observable executions—thus specifying the basic assumptions for programming and for software verification, and the correctness criterion for hardware verification. Architecture specifications have two main parts: the sequential and relaxed-memory concurrent aspects of instruction behaviour, each of which have been studied in previous work. For Armv8-A and RISC-V, Armstrong et al. have established full-scale sequential models in Sail [10,15], a domain-specific language for instruction-set architecture

A. Silva and K. R. M. Leino (Eds.) CAV 2021, LNCS 12759, pp. 303–316, 2021.
https://doi.org/10.1007/978-3-030-81685-8_14

(ISA) specification, that are complete enough to boot real-world operating systems such as Linux. For Armv8-A this model is automatically derived from the authoritative Arm-internal specification [24], while for RISC-V it has been hand-written and adopted by RISC-V International. On the concurrency side, relaxed-memory semantics can be specified in two main styles: either as *abstract-microarchitectural operational* models, characterising observable behaviour with explicit out-of-order execution and buffering, or as *axiomatic* models, expressed as a predicate over complete candidate executions represented as graphs of memory events. For Armv8-A and RISC-V "user" concurrency, both exist [1,7,8,22], along with a "Promising ARM" variant [23]. For Armv8-A they have been proved equivalent [21,22]; the authoritative vendor definition is the axiomatic one.

However, while an architecture *should* define the set of allowed executions for arbitrary programs, hitherto there has been no integration of full-scale ISA definitions with axiomatic concurrency models, either in mathematics or in tools (for operational models, this has only been done for RISC-V; other operational models have used small ISA fragments). Research and industry practice for relaxed memory semantics rely on making the semantics *executable as a test oracle*: not just a paper definition (in prose or mathematics), but tool-supported definitions that for small litmus-test examples can *compute* the set of all allowed executions, that can then be compared against experimental data. Many tools have been developed for operational and axiomatic architectural concurrency models [4,6,8,12,14,17–20,25,26,28–32], with axiomatic tools notably including the Herd tool of Alglave and Maranget [4,6,8], that can evaluate litmus tests w.r.t. axiomatic memory models specified in a relational-algebra style in the Cat language [2]. However, all of these previous tools for axiomatic models have (at best) used hard-coded ISA semantics that cover only small fragments of the complete architecture. For example, Zhang et al. [32] use a SMT solver based approach for SoC verification, with a user-specified memory model (TSO or SC), however the instruction level abstractions (ILAs) they use are much more abstract than the ISA semantics we consider.

In this paper we describe a tool, Isla, that integrates full-scale ISA specifications, in Sail, with arbitrary axiomatic models, in the Cat language. We first build a generic symbolic execution library for Sail specifications—which should also be valuable for other verification tasks. We use this to construct a tool for symbolically running binary litmus tests for any Sail ISA under any (non-recursive) Cat axiomatic memory model, using an SMT solver. We equip it with a web interface to make it widely accessible, and illustrate and evaluate all this for Armv8-A and RISC-V. Isla is available at https://isla-axiomatic.cl.cam.ac.uk and https://github.com/rems-project/isla. An extended version of the paper [11], available at https://www.cl.cam.ac.uk/~pes20/isla/, includes appendices showing the main parts of the full Sail/ASL semantics of a sample Armv8-A instruction (`add x4, x3, #1`); the Armv8-A axiomatic concurrency model (combining the official Arm specification for user concurrency [9,13] with the additions for instruction fetch semantics by Simner et al. [27]); and examples of the latter.

Our approach has several key advantages, which all follow from the fact that mainstream industry ISAs are surprisingly large and intricate. The Armv8-A ISA specification is around 100k lines. It defines the sequential behaviour of the full instruction set in all its detail, including e.g. instruction decoding, behaviour at each exception level, register banking, floating-point, vector instructions, system registers, exceptions, address translation, virtualisation, security extensions, and a host of optional architectural features. Simple litmus tests developed to investigate user concurrency have historically used only very few instructions and very little of this, and hand-written ISA models have sufficed, but even a 'simple' `ADD` instruction can, in reality, involve surprisingly much of the specification. If one wants to examine arbitrary compiler-generated code one needs many more instructions; and to develop systems concurrency semantics, e.g. covering the concurrency behaviour of instruction fetch, exceptions, or address translation, one might need any of the specification—and it would be exceedingly laborious and error-prone to reproduce it by hand in a hard-coded semantics. By handling the full authoritative Armv8-A ISA, we automatically support litmus tests that use arbitrary instructions, and we enable research on systems concurrency, with high confidence that the ISA follows the vendor specification. We demonstrate this by applying our tool to the model and examples for self-modifying code by Simner et al. [27], and our integration has also identified several places where the ISA specification needs modifications to correctly give the intended behaviour in a concurrent setting, e.g. to remove or enforce additional ordering. Because this is based on authoritative Arm and RISC-V ISA specifications, the work should enable relaxed-memory behaviour to be included in the standard test-edit-debug cycle used in the development of such large and critical specifications.

2 Implementation

Axiomatic relaxed-memory concurrency models, being expressed as logical constraints over candidate execution graphs, lend themselves to solver-based tool implementations. For the instruction-semantics part of such a tool, the most direct approach would be to translate the ISA semantics (for the instructions that occur in a litmus test) directly into SMT and combine that with the axiomatic-model constraints, roughly along the lines of Alglave et al. [3]. That approach was followed by Simner et al. [27], who compiled Sail directly into SMT to test an axiomatic model for instruction-fetch tests, but using a small handwritten Arm fragment, rather than the full Sail model derived from the Arm-internal model. The problem with this direct approach is one of scale: as one covers more of the Arm semantics, the resulting SMT problem simply becomes too large to be practicable. For example, for a load instruction, the virtual address must be translated into a physical address, which is a complex process with a great deal of configurability—there may be zero, one, or two stages of address translation, the page size may vary, the number of levels used in the page table may differ, etc. This approach also required the top level fetch-execute-decode loop to be handled specially, as one cannot translate such an unbounded loop directly into SMT, which imposes significant constraints on the shape of allowable tests.

In contrast, here we build and use a generic symbolic evaluation for Sail definitions using the Z3 SMT solver, which lets us compute the possible symbolic thread-local traces of each instruction, and hence of each thread (treating memory read values as unknowns, left to the concurrency model constraints). It also lets us use the same fetch-decode-execute loop that is used for emulation and co-simulation (which embodies various architecture-specific subtleties).

2.1 Symbolic Execution for Sail

Sail is attractive for symbolic execution for several reasons. First, it is an intentionally simple language, lacking many of the features found in general-purpose languages. Second, it has to support very few programs, just the specifications of major ISAs, so (unlike tools for conventional programming languages) we can tune the execution to them. Third, almost all of the loops in these programs are bounded. Our starting point is the translation of Sail to C, for emulation, by Armstrong et al. [10]. This goes via a simple goto-language intermediate representation which is already well-suited for this task.

Static Function Linearisation. Our symbolic execution always creates a new task when we hit a branch, and we do not ever merge these tasks at join points. This is a good strategy for instruction semantics, as it simplifies the symbolic execution engine significantly, but it does mean some code can cause unnecessary branching. To avoid this we have a static rewrite that can take a function with if statements and rewrite it into a 'linear' form, e.g. as below:

```
var x = 2;
if undefined {
  x = x + 1
} else {
  x = x + 2
};
return x
```

$\Longrightarrow$

```
let x0 = 2;
let b = undefined;
let x1 = x0 + 1;
let x2 = x0 + 2;
let x3 = ite(b, x1, x2);
return x3
```

This works by translating the body of the function into SSA form, then replacing the ϕ-functions with if-then-else (ite) functions that translate into the SMT `ite`. This results in a more complex SMT expression, but less branching in the symbolic execution, so it is a trade-off, but often worthwhile.

Per-Thread Candidate Executions. For each litmus-test thread this symbolic execution will produce a number of *candidate executions*, each of which is a sequence of memory events (memory reads and writes, fences, register accesses, and so on) with the symbolic values of these events potentially being constrained by some SMT formula for the overall execution. For example, consider the Armv8-A instruction `add x4, x3, #1`. For this instruction, our symbolic evaluator generates an execution:

```
(declare-const input (_ BitVec 64))
(read-reg |R3| nil input)
(define-const output (bvadd input #x0000000000000001))
(write-reg |R4| nil output)
```

where the SMTLIB formula is defined by the `declare-const` and `define-const` statements, with `read-reg` and `write-reg` effects indicating which variables in the SMT formula correspond to the values read and written to registers (which are otherwise just global variables) by the instruction. We simplify here for brevity, omitting the negative, zero, carry and overflow flags that the model computes. For more complex instructions, there are additional effects for memory accesses, cache maintenance events, barriers, and so on.

2.2 Checking a Litmus Test

Figure 1 shows the overall process of checking a litmus test. Tests can be supplied either in the `.litmus` format of previous axiomatic and operational tools [4,5, 14], reusing the parser from [4], or as a TOML file (a standard configuration file format, with libraries available for most languages). We first assemble the test with a conventional assembler into an ELF binary and load it into the representation of memory that will be used, before initialising the model with the program counter set to the entry point for each thread, then we symbolically execute the instructions in each thread separately, using the Sail semantics for each instruction, plus the same fetch-execute-decode loop in Sail we would use for emulation, to produce sets of per-thread traces as above. Treating litmus tests essentially as binaries, rather than the more-or-less ad hoc fragments of assembly abstract syntax used by earlier tools, accommodates the fact that the Armv8-A model does not define an abstract syntax, and reduces the gap between what the tool evaluates and what is run in experimental testing. Note that the Arm assembly in Fig. 1, as well as subsequent assembly snippets in this paper, use the standard Arm convention that `x0` and `w0` refer to the same register, where `w0` refers to the lower 32-bits of the register, and `x0` refers to the full 64-bit width.

We then generate an SMT problem for every combination of the candidate executions of each thread. This problem consists of the per-thread SMT formulae concatenated together (renaming variables as necessary to avoid name-clashes), combined with the axiomatic memory model (described in more detail below).

Finally, we need to generate some 'glue' SMT that connects the per-thread semantics with the memory model. For every effect in the per-thread SMT semantics we generate an enumeration of *events*, e.g. for an execution with two reads and two writes:

```
(declare-datatypes ((Event 0)) (((R1) (R2) (W1) (W2) (IW))))
```

The event `IW` is a special write event that represents the initial state. We generate relations such as `value-of` that relate events to their values as determined by the effects in the per-thread semantics, so if the second read event `R2` read the value `#xABCD`, `(value-of R2 #xABCD)` would be true. We generate *syntactic dependency*

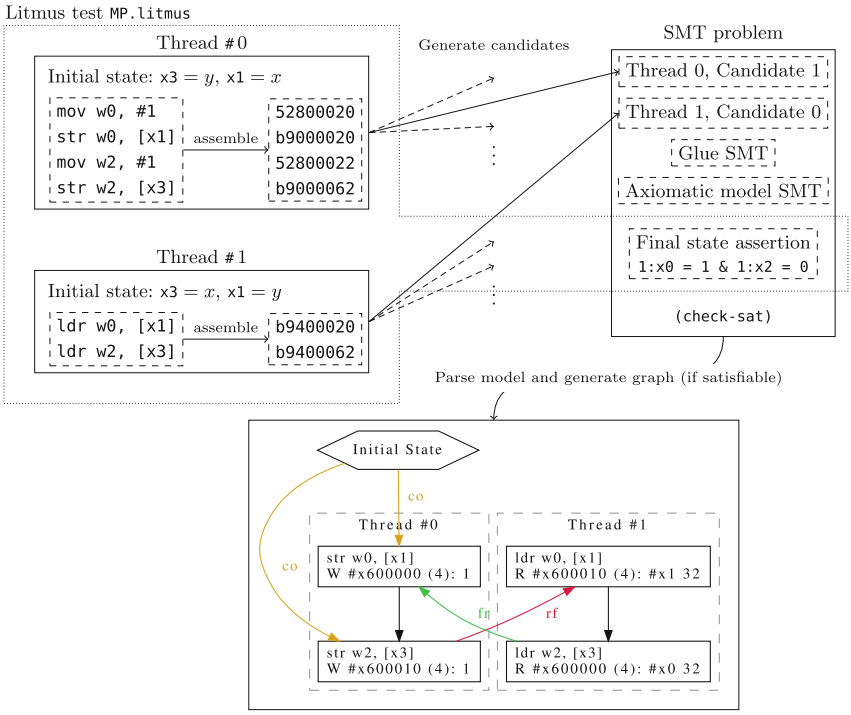

Fig. 1. Overview of process for checking the allowed executions of a litmus test

relations for address, data, and control dependencies, discussed in more detail in Sect. 2.3. Finally, there is a constraint on the final state of each test which specifies values expected in registers and memory after all threads have executed.

The Cat language represents axiomatic memory models as definitions of relations over the above events, and constraints over those relations, e.g. that specific relations are irreflexive, acyclic, or empty (or the negation of any of these). Relations are defined in a point-free relation-algebraic style, in terms of standard relational operators such as composition, intersection and union. The memory models we consider are all multi-copy-atomic, and all recursion in their definitions can trivially be replaced with (reflexive)-transitive closure. Herd's let rec construct computes the least solution to a set of equations [2], which is tricky to represent in SMT, so we do not support it. We believe even relations such as Power's (mutually recursive) preserved program order are nevertheless representable as SMT, so this limitation is mostly in our translation from Cat—we would likely want to use a different syntax to represent these relations for Isla.

A satisfiable solution to the overall SMT problem described above thus represents an execution permitted by the architecture. Parsing the model generated by the SMT solver allows us to generate a graph of the execution by instantiating each relation in the model with the various events. If all generated SMT problems

are unsatisfiable for every combination of per-thread candidate executions then there are no permitted executions. If desired we can repeatedly ask the SMT solver for additional distinct models until we have all permitted executions.

2.3 Syntactic Dependency Analysis

Axiomatic memory models for relaxed hardware architectures rely heavily on notions of address, data, and control dependencies between instructions. For example, consider the following assembly:

```
    ldr  w0, [x1]   // load 32 bits from address in x1 into x0
    cbnz w0, LC01   // compare and branch if non-zero to LC01
LC01:
    mov  w2, #1     // load 1 into x2
    str  w2, [x3]   // store 32 bit-value in x2 to the address in x3
```

Here there is a control dependency between the load (`ldr`) and the store (`str`), as the value read by the load is used to determine whether the branch instruction `cbnz` that precedes the store is taken or not. This control dependency exists irregardless of whether the branch is taken or not—its existence is purely determined by the syntactic structure of the above code.

In general, existing ISA descriptions do not cover this aspect of the architecture well, as they are principally developed only to describe the sequential behaviour. Previous tools have either hand-coded dependency information, which is acceptable for cut-down ISA models but too laborious and error-prone at the scale of the ISA models we use, or used a heavyweight taint-tracking interpreter [15]. Our approach avoids both of these. It is similar to the latter, computing dependencies from the ISA specification, but building the footprint analysis atop our symbolic execution library requires only around 500 LoC.

To express dependencies, we need to associate each event in our candidate executions with the syntactic instruction/opcode that generated them. To do this we use a Sail function `__instr_announce(opcode)`, called in each architecture's fetch-decode-execute loop just after fetching an instruction; this adds a special effect to the candidate execution recording the instruction opcode. We also have another special effect that delimits each fetch-decode-execute cycle, so each effect such as `read-mem` and `write-mem` that would give rise to an event can be associated with an opcode, as well as an index in the program order relation for its thread.

For each instruction we also need to know its *footprint*: data about the instruction including which input registers it reads, which output registers it writes, whether it is a branch instruction, and so on. It also contains *taint* information—we need to know which registers writes may contain data 'tainted' by a memory read performed by a load, or which input registers 'taint' data written to memory. The Sail ISA specifications do not explicitly describe this footprint, so we are forced to derive it from the specification.

To do this we symbolically evaluate each opcode independently in a suitably unconstrained environment so as to capture all its possible behaviours. This can be computationally expensive due to the number of possible behaviours

some instructions have, so we build a footprint cache to avoid re-computing this where possible. It turns out to be hard to distinguish ordinary branches from instructions that can cause an exception to occur, so we add a special branch address announce effect, created by a Sail function `__branch_address_announce` that we call in branch instructions. This also enables the taint tracking for branch addresses we need for control dependencies as described above. The taint tracking is achieved simply by looking at what sub-expressions in the generated SMT problem contain variables that also appear in the various effects in each trace.

Once we have this footprint information we can analyse it for the opcodes between each read and write effect and derive the necessary dependency relations over their events. Note that this dependency relation must be exact. If we under-approximate, we will allow executions that should be forbidden, and if we over-approximate we will forbid executions that should be allowed.

In some cases the current Arm-provided ISA specification does not include enough information to identify the architecturally respected dependencies, and our dependency analysis would identify a dependency when there should not be one. To solve this we add some special Sail functions that give fine-grained control of the dependency calculation. For example, in indirect branches we ignore any dependency between the target register `X`n and the link register `X30` by including a function in the Sail definition that tells the footprint analysis to ignore any relation it finds between the two registers.

```
if branch_type == BranchType_INDCALL then {
    ignore_dependency_edge(n, 30);
    X(30) = PC() + 4
};
```

This works by adding a special annotation in the candidate execution trace which can be used by the footprint analysis—for all other purposes it is a no-op. This information should properly become part of the architecture specification, as mistakes in the dependency calculations could be a source of soundness bugs. The lack of support for this information in existing ISA specifications can partly be explained by the lack of tooling to properly explore the integration of ISA specifications with concurrency, something we hope a tool such as ours can address.

2.4 Web Interface

Figure 2 shows the web interface we have developed for our tool, based on the web interface for the C memory model tool Cerberus-BMC by Lau et al. [16]. This can either be run locally, or via a website, https://isla-axiomatic.cl.cam.ac.uk.

3 System Litmus Tests

As mentioned previously, one advantage of our tool is that, because it supports the full sequential ISA, it enables easy experimentation with tests and models

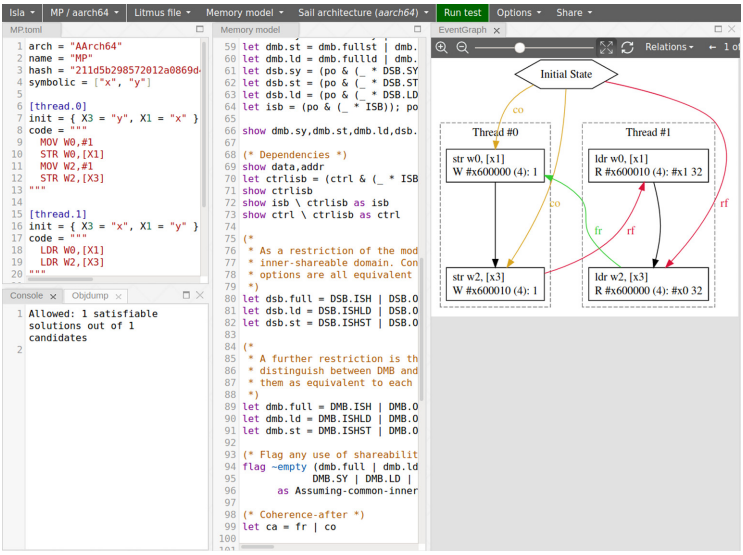

Fig. 2. Web interface for the tool

outside the scope of previous tools, e.g. involving new systems features. For example, Simner et al. developed semantics for Arm instruction fetch and I/D cache maintenance [27]. Consider the litmus test in Fig. 3 [27, §3.3], a simple test involving self-modifying code. In order to run this test and the others in [27] our tool required only minimal changes: we had to add support for data-cache and instruction-cache maintenance events and relations for them in our Cat to SMT translation. Additionally we needed to generalise how we generated the `rf` (reads-from) relation to generate both the regular `rf` relation and the new `irf` (instruction-reads-from) relation. Because our tool already runs tests using a fetch-execute-decode loop, all the instruction fetch events were already available—we in fact filter them out when running user-mode tests.

When generating candidate executions for a thread we normally do not assume anything about what other threads may be doing, but for self-modifying code this would clearly be problematic, as it would imply that any other thread could modify any of this thread's instructions arbitrarily. We therefore mark the memory locations that contain instructions that can be modified and provide in advance all the possible values they might take.

4 Results and Comparisons

We evaluate our tool for correctness and performance with respect to Herd using previous corpora of tests.

```
1        str w0, [x1]
2        dc cvau, x1
3        dsb ish
4        ic ivau, x1
5        dsb ish
6        isb
7        bl f
8        mov w2, w10
9        b Lout
10   f:  b l0
11   l1: mov w10, #2
12       ret
13   l0: mov w10, #1
14       ret
15   Lout:
```

In the initial state register `x1` contains the address of the label `f`, and register `w0` contains the opcode for the branch instruction `b l1`. Without the highlighted cache-maintenance and barrier instructions on lines 2–6, the write of that opcode to `f` performed by the store on line 1 may or may not be observed before the instruction fetch for `f`, so at the end of the test the register `w2` can contain either 1 or 2, depending on whether we branched to `l1` or `l0`.

The highlighted instructions on lines 2–6 are a sequence of data-cache (`dc`) and instruction-cache (`ic`) maintenance instructions with requisite data and instruction barriers that must occur to guarantee that the write is observed by the instruction fetch, as documented by the Armv8-A architecture reference manual [7] and captured by the axiomatic model of Simner et al. [27]

Fig. 3. Self-modifying code litmus test SM+cachesync-isb

We select 3798 litmus tests for both Armv8-A and RISC-V to compare between our tool and Herd—these tests include a representative set of features such as barriers and atomics, while exercising all of the basic litmus test shapes. All tests were run on a 2.6GHz Intel Xeon Gold 6240 CPU with 36 physical cores and 400GB of RAM. The tests are split into rough categories based on the contents of the tests. We ran 36 concurrent instances of both our tool and Herd across each set of tests, running Herd with the `-speedcheck fast` flag which causes it to stop enumerating executions when it resolves the final assertion in each test, which is the closest behaviour to how our tool behaves by default.

To assess correctness, we use a set of golden references for these above tests, for all of which the previous operational RMEM [14] and axiomatic Herd models and tools agree, and which have been extensively validated against hardware implementations. We confirm that our tool produces the same expected results as those models for all the litmus tests, including when run in exhaustive mode.

To assess performance, the table below gives the total real execution time for each batch of tests.

Test set	Number of tests	Isla	Herd
Armv8-A basic 2-thread	1377	49 s	11 s
Armv8-A basic 3-thread	161	11.7 s	1.2 s
Armv8-A exclusives	23	20.2 s	1.5 s
Armv8-A DMB/LD	70	7.4 s	0.7 s
Armv8-A PPO	2020	3 m 29.3 s	16.2 s
RISC-V basic 2-thread	36	0.7 s	0.2 s
RISC-V AMOs	111	2 s	0.7 s

In general Herd is faster for nearly all tests, but this is not surprising given the amount of detail in the full-scale instruction semantics that we are using, particularly for Armv8-A. Our goal is not to be faster, but to support those full-scale ISA semantics while remaining fast enough for practical purposes. We achieve this: most tests take only a second or so to run, which is perfectly usable interactively. For example, given the Armv8-A basic 3-thread tests, for a single sequential run of the tests, the shortest took 872 ms to run, while the longest took 1231 ms. The above batch times are similarly perfectly usable for (e.g.) regression testing while editing a model.

We also evaluate our tool with respect to that of Simner et al., for the instruction-fetch tests (which are currently not supported by Herd) in Sect. 6 of their paper. Our tool returns the expected results for all these tests, including the two tests (FOW and SM.F+ic) that were unsupported by their tool. In terms of performance, we note that their tool took 30 min to run just 90 of the 1377 basic 2-thread tests above, which is awkwardly slow for using a tool in practice, whereas when limiting our tool to 8 cores (to more closely match their experimental setup) our tool will execute all 1377 in under 3 min. We were additionally able to provide further validation that the Simner et al. model behaves as the standard Armv8-A model for non-self-modifying tests by showing that it behaves identically for all 3798 of the non-self-modifying tests above.

Acknowledgement. This work was partially supported by the UK Government Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme, to deliver a DSbDtech enabled digital platform (grant 105694), ERC AdG 789108 ELVER, EPSRC programme grant EP/K008528/1 REMS, an Arm iCASE award, Arm, and Google. Approved for public release; distribution is unlimited. This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8650-18-C-7809 ("CIFV"). The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

1. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191214-draft, 238 pages (2020). https://riscv.org/technical/specifications/. Accessed 23 Sept 2020
2. Alglave, J., Cousot, P., Maranget, L.: Syntax and semantics of the weak consistency model specification language cat. CoRR abs/1608.07531 (2016)
3. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Computer Aided Verification - 25th International Conference, CAV, pp. 141–157 (2013). https://doi.org/10.1007/978-3-642-39799-8_9
4. Alglave, J., Maranget, L.: The `diy7` tool. http://diy.inria.fr/. Accessed 28 Jan 2021
5. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Litmus: running tests against hardware. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 41–44. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_5

6. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM TOPLAS **36**(2), 7:1–7:74 (2014). https://doi.org/10.1145/2627752
7. Arm: Arm Architecture Reference Manual: Armv8, for Armv8-A architecture profile, 8248 pages (2020). https://developer.arm.com/documentation/ddi0487/fc. Accessed 23 Sept 2020
8. Arm: Memory model tool (2020). https://developer.arm.com/architectures/cpu-architecture/a-profile/memory-model-tool Accessed 26 Jan 2021
9. ARM Ltd.: ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile) (2017). ARM DDI 0487B.a (ID033117). https://developer.arm.com/documentation/ddi0487/b/?lang=en
10. Armstrong, A., et al.: ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS (2019). http://www.cl.cam.ac.uk/~pes20/sail/
11. Armstrong, A., Campbell, B., Simner, B., Pulte, C., Sewell, P.: Isla: integrating full-scale ISA semantics and axiomatic concurrency models (extended version). In: Extended version of a paper in Proceedings of CAV 2021: 33rd International Conference on Computer-Aided Verification (2021). https://www.cl.cam.ac.uk/~pes20/isla/
12. Bornholt, J., Torlak, E.: Synthesizing memory models from framework sketches and litmus tests. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June, 2017, pp. 467–481. ACM (2017). https://doi.org/10.1145/3062341.3062353
13. Deacon, W.: The ARMv8 application level memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat (2016)
14. Flur, S., French, J., Gray, K., Pulte, C., Sarkar, S., Sewell, P.: RMEM (2020). www.cl.cam.ac.uk/~pes20/rmem/. Accessed 28 Jan 2021
15. Gray, K.E., Kerneis, G., Mulligan, D., Pulte, C., Sarkar, S., Sewell, P.: An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In: Proceedings of MICRO-48, the 48th Annual IEEE/ACM International Symposium on Microarchitecture (2015). https://doi.org/10.1145/2830772.2830775
16. Lau, S., Gomes, V.B.F., Memarian, K., Pichon-Pharabod, J., Sewell, P.: Cerberus-BMC: a principled reference semantics and exploration tool for concurrent and sequential C. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. LNCS, vol. 11561, pp. 387–397. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_22
17. Mador-Haim, S., et al.: An axiomatic memory model for POWER multiprocessors. In: Proceedings of the 24th International Conference on Computer Aided Verification, pp. 495–512 (2012). https://doi.org/10.1007/978-3-642-31424-7_36
18. Martonosi Research Group: Check research tools and papers. https://check.cs.princeton.edu/. Accessed 28 Jan 2021
19. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Proceedings of TPHOLs 2009: Theorem Proving in Higher Order Logics, LNCS 5674, pp. 391–407 (2009). https://doi.org/10.1007/978-3-642-03359-9_27
20. Park, S., Dill, D.L.: An executable specification and verifier for relaxed memory order. IEEE Trans. Comput. **48**(2), 227–235 (1999)
21. Pulte, C.: The semantics of multicopy atomic ARMv8 and RISC-V. Ph.D. thesis, University of Cambridge (2018). https://www.repository.cam.ac.uk/handle/1810/292229

22. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. In: POPL 2018: Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (2018). https://doi.org/10.1145/3158107
23. Pulte, C., Pichon-Pharabod, J., Kang, J., Lee, S.H., Hur, C.K.: Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In: PLDI 2019: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (2019). https://doi.org/10.1145/3314221.3314624
24. Reid, A.: Trustworthy specifications of ARM v8-A and v8-M system level architecture. In: FMCAD 2016, pp. 161–168 (2016). https://alastairreid.github.io/papers/fmcad2016-trustworthy.pdf
25. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: Proceedings of PLDI 2011: the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 175–186 (2011). https://doi.org/10.1145/1993498.1993520
26. Sarkar, S., et al.: The semantics of x86-CC multiprocessor machine code. In: Proceedings of POPL 2009: the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 379–391 (2009). https://doi.org/10.1145/1594834.1480929
27. Simner, B., et al.: Armv8-a system semantics: instruction fetch in relaxed architectures. In: ESOP 2020: Proceedings of the 29th European Symposium on Programming (2020). http://www.cl.cam.ac.uk/~pes20/iflat/top-extended.pdf
28. Trippel, C., Manerkar, Y.A., Lustig, D., Pellauer, M., Martonosi, M.: Full-stack memory model verification with tricheck. IEEE Micro **38**(3), 58–68 (2018)
29. Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically comparing memory consistency models. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017, pp. 190–204. ACM (2017). http://dl.acm.org/citation.cfm?id=3009838
30. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Analyzing the intel Itanium memory ordering rules using logic programming and SAT. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 81–95. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39724-3_9
31. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Nemos: a framework for axiomatic and executable specifications of memory consistency models. In: 18th International Parallel and Distributed Processing Symposium (IPDPS 2004), Santa Fe, New Mexico, USA (2004). https://doi.org/10.1109/IPDPS.2004.1302944
32. Zhang, H., Trippel, C., Manerkar, Y.A., Gupta, A., Martonosi, M., Malik, S.: ILA-MCM: integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification. In: 2018 Formal Methods in Computer Aided Design (FMCAD), pp. 1–10 (2018). https://doi.org/10.23919/FMCAD.2018.8603015