Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA

Shaked Flur[1]   Kathryn E. Gray[1]   Christopher Pulte[1]   Susmit Sarkar[2]   Ali Sezgin[1]
Luc Maranget[3]   Will Deacon[4]   Peter Sewell[1]

[1] University of Cambridge, UK   [2] University of St Andrews, UK   [3] INRIA, France   [4] ARM Ltd., UK
first.last@cl.cam.ac.uk   ss265@st-andrews.ac.uk   luc.maranget@inria.fr   will.deacon@arm.com

Abstract

In this paper we develop semantics for key aspects of the ARMv8 multiprocessor architecture: the concurrency model and much of the 64-bit application-level instruction set (ISA). Our goal is to clarify what the range of architecturally allowable behaviour is, and thereby to support future work on formal verification, analysis, and testing of concurrent ARM software and hardware.

Establishing such models with high confidence is intrinsically difficult: it involves capturing the vendor's architectural intent, aspects of which (especially for concurrency) have not previously been precisely defined. We therefore first develop a concurrency model with a microarchitectural flavour, abstracting from many hardware implementation concerns but still close to hardware-designer intuition. This means it can be discussed in detail with ARM architects. We then develop a more abstract model, better suited for use as an architectural specification, which we prove sound w.r.t. the first.

The instruction semantics involves further difficulties, handling the mass of detail and the subtle intensional information required to interface to the concurrency model. We have a novel ISA description language, with a lightweight dependent type system, letting us do both with a rather direct representation of the ARM reference manual instruction descriptions.

We build a tool from the combined semantics that lets one explore, either interactively or exhaustively, the full range of architecturally allowed behaviour, for litmus tests and (small) ELF executables. We prove correctness of some optimisations needed for tool performance.

We validate the models by discussion with ARM staff, and by comparison against ARM hardware behaviour, for ISA single-instruction tests and concurrent litmus tests.

Categories and Subject Descriptors   C.0 [*General*]: Modeling of computer architecture;   D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming;   F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords   Relaxed Memory Models, semantics, ISA

1. Introduction

The ARM architecture is the specification of a wide range of processors: cores designed by ARM that are integrated into devices produced by many other vendors, and cores designed ab initio by ARM architecture partners, such as Nvidia and Qualcomm. The architecture defines the properties on which software can rely on, identifying an envelope of behaviour that all these processors are supposed to conform to. It is thus a central interface in the industry, between those hardware vendors and software developers. It is also a desirable target for software verification and analysis: software that is verified w.r.t. the architecture should run correctly on any of those processors (modulo any hardware errata, of course).

However, exactly what behaviour is and is not allowed by the architecture is not always clear, especially when it comes to the concurrency behaviour of ARM multiprocessors. The architecture aims to be rather loose, to not over-constrain the hardware microarchitectural design choices of all those different vendors, and to permit optimised implementations with high performance and low power consumption. To this end, it adopts a *relaxed memory model*, allowing some effects of out-of-order and non-multi-copy-atomic implementations to be programmer-visible. But it describes that in prose, which, as one might expect, leaves many open questions.

Our goal in this paper is to clarify this situation, developing mathematically rigorous models that capture the ARM architectural intent. But establishing such models with high confidence is intrinsically difficult, as the vendor intent has not previously been made this precise, either in public documents or internally. Black-box testing of processor implementations is useful here, comparing their behaviour against that allowed by models for a variety of concurrent test programs [3, 4, 8, 21, 22, 24], but it can only go so far; one really needs to discuss the model in detail with the architects (those with the authority to define what the architecture allows). This means it must be accessible to them, and that suggests our strategy: we first develop a concurrency model with a microarchitectural flavour, abstracting from many hardware implementation concerns but still close to hardware-designer intuition, so that it can be clearly (albeit necessarily informally) related to the processor designs they have in mind, and be tested against their extensional behaviour. In this *Flowing model* read requests, writes, and barriers flow explicitly through a hierarchical storage subsystem. We then develop a more abstract model, better suited for use as an architectural specification and programmer's model, which we prove sound w.r.t. that. This *partial-order propagation* (or *POP*) model abstracts from the storage-subsystem hierarchy to give a model in which all hardware threads are symmetrical. Both models have been discussed in detail with senior ARM staff, resolving many subtle questions about the architecture.

The concurrency semantics alone is not enough to understand or reason about concurrent code; one also needs semantics for the instruction set architecture (the ISA). The ARM architecture describes the sequential behaviour of the ISA reasonably precisely, in a proprietary pseudocode language; the problems here are (1) dealing with the mass of detail involved, and (2) integrating these sequential descriptions into the concurrent semantics. One cannot simply treat each instruction as an atomic transaction, or interleave their pseudocode. Previous work on relaxed-memory semantics has not really addressed this issue, either avoiding it entirely or defining semantics for a small ad hoc fragment of the ISA. Here we use a novel ISA description language, with a lightweight dependent type system, that lets us use a rather direct representation of the ARM architecture reference manual (the ARM ARM) instruction descriptions [7]. We model all the application-level ISA except for floating-point and vector instructions, and we validate this part of the semantics against hardware for a suite of automatically generated single-instruction tests.

Our models are defined in executable higher-order logic, in Lem [19], and we use Lem to generate executable OCaml code to make a tool allowing one to explore, either interactively or exhaustively, the behaviour of concurrent litmus tests or (small) conventional ELF binaries. For performance the tool relies on the fact that certain transitions commute, to reduce the combinatorial blow-up; we prove that those properties hold.

Our focus throughout is on the ARMv8 version of the architecture, which introduced support for 64-bit execution. Example ARMv8 cores and processors include the ARM-designed Cortex-A53 and A57 cores, in the AMD Opteron A1100, the Qualcomm Snapdragon 810, and the Samsung Exynos Octa 5433 SoCs; together with architecture-partner-designed processors such as Nvidia Denver core, used in the Tegra K1. The Apple A7 and A8 (in iPhones and iPads since the iPhone 5S) also appear to be ARMv8-compatible.
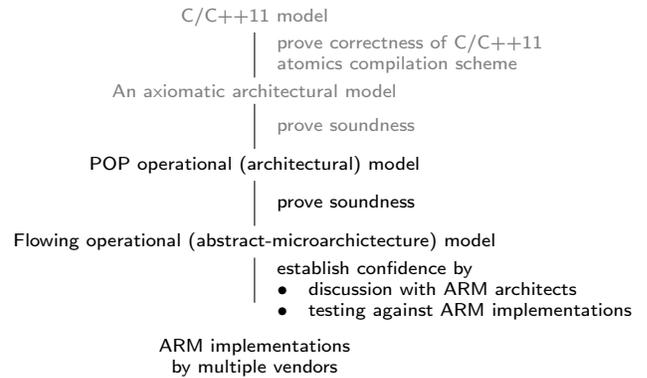
ARMv8 also added several new concurrency primitives, including the ARM load-acquire and store-release instructions, and weaker barrier instructions than the ARMv7 dmb full barrier. It includes both a 64-bit and 32-bit instruction set; we deal with the former, the A64 of AArch64, and all those concurrency primitives.

To summarise our contribution, we:

- give an abstract-microarchitectural model, the Flowing model, for ARMv8 concurrency, validated both by black-box testing and discussions with ARM staff (§3,4,7,13);
- give a more abstract model, the POP model (§5,8);
- integrate the above with an ISA semantics for all the application-level non-FP/SIMD ARMv8 instruction set (§6);
- prove that POP does indeed abstract Flowing (§9);
- prove that various model transitions commute (§10);
- develop a tool that allows one to explore, either interactively or exhaustively, the behaviour of concurrent litmus tests or (small) conventional ELF binaries (§11); and
- demonstrate this on an example of the Linux ARMv8 spinlock code (§12).

We begin with an informal description of the main ideas underlying the Flowing model, and the concurrency phenomena it has to deal with (§3,4). The web interface and other online material can be accessed from http://www.cl.cam.ac.uk/~pes20/popl16-armv8.

We envisage a stack as below, where the black-labelled edges, in this paper, enable a range of semantics and verification activities above this solid foundation, such as the gray-labelled edges showing a possible C/C++11 concurrency implementation result above ARM. Looking downwards, the Flowing model can also be used for testing and verification of aspects of real hardware, taking us closer towards a production top-to-bottom verified stack.

C/C++11 model

prove correctness of C/C++11 atomics compilation scheme

An axiomatic architectural model

prove soundness

POP operational (architectural) model

prove soundness

Flowing operational (abstract-microarchictecture) model

establish confidence by
- discussion with ARM architects
- testing against ARM implementations

ARM implementations by multiple vendors

2. Related Work

There has been extensive recent work on hardware memory models, e.g. for x86 [20], IBM Power [22, 23], and ARMv7 [4]. The Power and ARM concurrency architectures are broadly similar, both being relaxed non-multi-copy atomic models that respect only certain program-order relations, and which have cumulative memory barrier instructions. But they differ in many important aspects: they have different memory barrier and synchronisation instructions[1], and the associated microarchitectures are quite different. That is important for us here: that Power model [22, 23] does not correspond well to ARM implementations, and so cannot serve as a basis for the discussion with ARM architects that is needed for solid validation that it matches their intent.

For example, in typical Power microarchitectures (as we understand them) memory writes and read-requests propagate via separate structures, and the Power sync memory barrier implementation involves an acknowledgement being sent back to the originating hardware thread when the barrier has been processed by each other thread. That Power model, largely based on discussions with an IBM architect as it was, explicitly modelled those sync-acknowledgements. But ARM microarchitectures may keep memory writes and read-requests in the same structures, with the ARM dmb memory barrier achieving similar extensional effects to sync quite differently, by keeping pairs of barrier-separated accesses ordered within those, rather than with global acknowledgements. It is this ordering that our Flowing model captures. We shall see other more subtle differences below, all of which are important for intensional discussion with the vendors, and some of which give rise to programmer-observable effects.

The other closely related work is the ARMv7 model of Alglave et al. [4], which differs in other important aspects from what we do here. Most importantly, it is aiming to be considerably more abstract than either of the two models we present, without the microarchitectural flavour that enables us to establish a clear relationship between them and the architects' intent. That level of abstraction is thus good for simplicity but bad for validation, and the validation of that model relies heavily on black-box testing. In an ideal world (but one which we leave for future work) we would have both the low-level microarchitectural Flowing model as we describe here, validated both by discussion and testing, and a proof (via our POP model) that such an abstract model is indeed sound

---

[1] The Power sync and ARM dmb are broadly similar, but there is no ARM counterpart of the Power lwsync, and there is no Power counterpart of the ARMv8 dmb st and dmb ld barriers or of the ARMv8 load-acquire and store-release instructions.

w.r.t. the architectural intent. The models differ also in their mathematical style: following Sarkar et al. [22, 23], we adopt an operational style, which again is good for the correspondence with architect intuition, while [4] is axiomatic. The latter is faster for exhaustively checking the results of small litmus tests, while the former allows one to incrementally explore single executions of larger programs. Finally, we cover ARMv8 rather than ARMv7, and integrate with semantics for a large part of the instruction set.

There has been a great deal of previous work using domain-specific IDLs and proof assistants to describe instruction behaviour, for many different purposes. On the formal and semantics side, this includes the ARMv7 model by Fox [10] in his L3 language, that by Goel et al. [11] for x86 in ACL2, work on automatically generating compiler components, e.g. [9], and the assembly language and machine-code models used by verified compilation work, e.g. [5, 13–15, 18, 26] With the exception of CompCertTSO [26], which was w.r.t. the much simpler x86-TSO model, none of these address concurrency. A few, notably those of Fox and Goel, are rather complete in their coverage of the sequential ISA, but many include just enough for the purpose at hand (simplifying the problems of scale), and are not concerned with engineer-accessibility.

3. Introduction to the Flowing Model

Modern high-end processors are astonishingly complex. The pipeline of an ARM Cortex-A57 core can have up to 128 instructions in flight simultaneously [6], with each instruction broken down into micro-operations that are dispatched to multiple arithmetic, branch, floating-point, and load/store execution units. Shadow registers and register renaming are used to prevent the number of architected registers being a bottleneck. A very high-level view of a Cortex-A72 core showing some of this is in Fig. 1, and there are multiple levels of cache and interconnect in addition to that, some shared between cores or clusters thereof.

A detailed microarchitectural description does not make a good programmers model, or a usable basis for semantics and reasoning about concurrent software. Indeed, much microarchitectural detail is, by design, not observable to programmers (except via performance properties); there are only some aspects that give rise to behaviour that concurrent contexts can observe, notably the out-of-order execution that the pipeline permits (including shadow registers), and the lack of multi-copy-atomicity that the cache protocol, cache hierarchy and interconnect might exhibit. Our task here is to invent a new abstraction, just concrete enough to model those aspects, and just concrete enough to let hardware designers and architects relate it to the range of actual microarchitectures they have in mind, but otherwise as simple as possible. This model can be validated by discussion and by testing against hardware. Note that we are not defining a *specific* microarchitecture, but rather a model with a microarchitectural flavour that is *architecturally complete*, allowing the full envelope of behaviour that the architects intend. Then we can build higher-level models, proving them sound with respect to that abstract-microarchitectural one, to use for reasoning.

We build this new abstraction, the *Flowing model*, in three parts. The *instruction semantics* defines the behaviour of each instruction in isolation, giving, for each, a clause of an AST type denoting the machine-code instructions, a decode function that maps 32-bit opcode bitvectors into that type, and a clause of an execute function that gives imperative pseudocode for the instruction. Fig. 2 shows these for the ADD instruction (eliding the body of its decode function). We elaborate on this ISA model and the language used for it in §6. We interpret the bodies of the decode and execute functions with an operational semantics for our ISA description language; the steps of that semantics are essentially abstract micro-operations, abstracting from the actual hardware micro-operations performed by the arithmetic units etc. Even what one might think is a simple
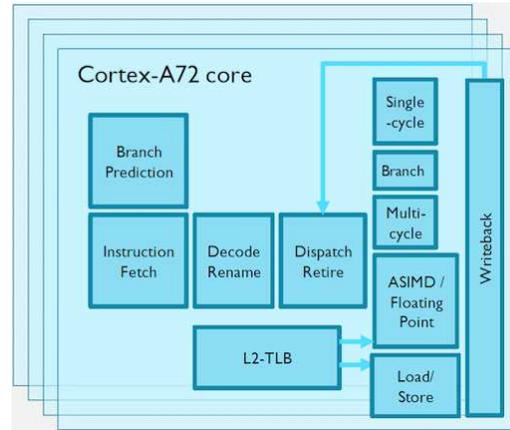

Figure 1. ARM Cortex-A72 Core Block Diagram (source: ARM)

```
typedef ast =
  forall Int 'R, 'R IN {32, 64}, (*register size*)
         Int 'D, 'D IN {8,16,32,64}. (*data size*)
  | ...
  | AddSubImmediate of
    (reg_idx,reg_idx,[:'R:],boolean,boolean,bit['R])

function forall Int 'R, 'R IN {32,64},
              Int 'D, 'D IN {8,16,32,64}.
  decodeAddSubtractImmediate
    ([sf]:[op]:[S]:0b10001:shift:(bit[12]) imm12:Rn:Rd)
    : ast<'R,'D> effect pure  =
  ...
function clause execute (AddSubImmediate(d,n,
    datasize,sub_op,setflags,imm)) = {
  (bit['R]) operand1 :=
    if n == 31 then rSP() else rX(n);
  (bit['R]) operand2 := imm;
  (bit) carry_in := 0; (*ARM:uninitialized*)
  if sub_op then {
    operand2 := NOT(operand2);
    carry_in := 1;
  }
  else
    carry_in := 0;
  let (result,nzcv) =
    (AddWithCarry(operand1,operand2,carry_in)) in {
    if setflags then
      wPSTATE_NZCV() := nzcv;
    if (d == 31 & ~(setflags)) then
      wSP() := result
    else
      wX(d) := result;
  }
}
```

Figure 2. Instruction semantics for add/adds/sub/subs

instruction like add is surprisingly intricate when one handles all the details; one execution involves 80 instruction-semantics steps.

The *thread subsystem*, analogous but different in detail to that of the IBM Power model of Sarkar et al. [22], models the execution of instructions in a single core (or hardware thread) by allowing out-of-order execution, respecting various inter-instruction dependencies as necessary. This abstracts from the out-of-order dispatch and control logic, the register renaming, the load/store queues, and the local (L1) cache of the core, modelling all that with a simple tree of instruction instances:
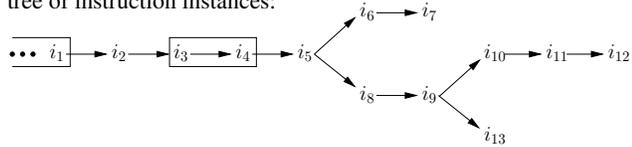

Some of these (boxed) may be finished, while others may be executing out-of-order or speculatively, after a conditional branch (the forks) which is not yet resolved. The ISA model generates *events*, principally: *register-read*, *register-write*, *memory-read*, *memory-write* and *barrier*, which are used by the thread-subsystem (§7.7 Sail interpreter step); some involve instruction-state continuations, e.g. to record an instruction state which is awaiting the result of a memory read.

The last part of the model is the *storage subsystem*, which receives memory read, write and barrier *requests* from the thread and replies to read-requests with *read-responses* containing the write to be read from. The storage subsystem abstracts from the interconnect and shared cache levels in the microarchitecture. The Flowing-model instruction semantics and thread subsystem will be reused in our more abstract POP model, while the storage subsystem will be replaced by a more abstract model.

Given all the complexity of real microarchitectures it is interesting to note that the ARM architects have a much simpler abstraction in mind when they discuss the validity of different behaviour: they can often reason about the architecture and the hypothetical behaviour of hardware without needing to unpack details of cache protocols and suchlike. The basic idea of our Flowing model is to try to formalize this abstraction, keeping it as familiar to them as much as possible. In the rest of this section we introduce it, via two motivating examples.

Lack of multi-copy atomicity   The ARM architecture is not multi-copy atomic [8]: a write by one hardware thread can become visible to some other threads before becoming visible to all of them. This is observable to the programmer in concurrent litmus tests such as WRC+addrs (write-to-read causality):
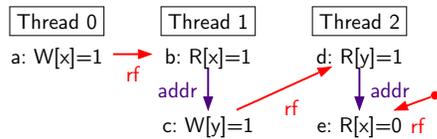

Test WRC+addrs: Allowed

Here a message x=1 is written by Thread 0 and read by a Thread 1 which then raises a flag y=1. That in turn is read by Thread 2 which then tries to read the message. To prevent local reordering in the second two threads, both of their second accesses are made address-dependent on the results of their first reads. This is not enough to prevent the unwanted behavior shown, in which Thread 2 reads from the initial state of x, not the value 1 of the message (a).

The diagram shows a particular execution, not a program: a graph over memory read and memory write events. A read event b:R[x]=1 represents a memory read from the address of x of value 1. Similarly a:W[x]=1 represents a memory write. All variables are disjoint and initialized to 0. The rf edges indicate *reads-from*

relations between writes and reads (the rf edge from a red dot indicates a read from the initial memory value).

Microarchitecturally, this behaviour can arise from multi-level cache and core clustering. Two cores that share a cache level that is not shared with a third core can observe each other's memory accesses before the third core does. ARM architects reason about the behaviour of this by abstracting it to a tree-shaped topology of queues in which requests "flow", starting from the entry point of the issuing thread through join points until they reach main memory. On their way to main memory some requests might bypass others subject to reordering conditions. In this abstraction, read requests do not need to travel all the way to main memory to get satisfied: if a read request encounters a write to the same address, a read response can be immediately sent to the issuing thread of the read request. This abstraction is the inspiration behind the Flowing storage subsystem.

For example, a particular processor might have a topology as below, with the pairs of threads 0,1, and 2,3, each sharing a queue which is not shared with the rest (many other topologies are possible, especially with more cores). At a certain point in the computation, the write (a) might have propagated down past one join point, with the read request (b) still above that point.
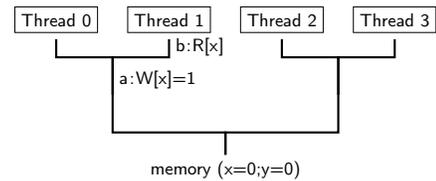
Later the read request (b) can flow down another step, and then perhaps be satisfied directly from (a). That will resolve the address dependency to write (c), which could be committed and flow to the same point:
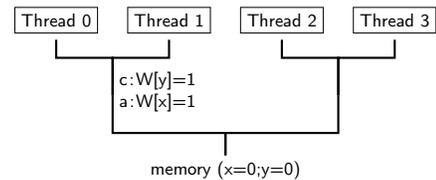

Then, as (c) and (a) are to different addresses (and not separated by a barrier), they can be *reordered* , and (c) could flow past the next point where it can satisfy Thread 2's first read. In turn that resolves the Thread 2 address dependency to its read of x, and that can flow down and be satisfied from the initial x=0 value in main memory before (a) passes the bottom join point.

The abstraction used in this description has pros and cons. It is reasonably simple and easy to understand, and to relate to hierarchical microarchitectures. But the asymmetry between hardware threads is uncomfortable from an architectural point of view (programmers will typically not know the assignment of software threads to particular hardware threads, and should not write code where the correctness depends on that assignment); it is exactly that asymmetry that our POP model will abstract from.

It is interesting to note also that the testing of ARM implementations that we have done to date has not exhibited observable non-multi-copy-atomic behaviour, even though the architecture intentionally and unambiguously permits it (to allow freedom for future implementations). This limits the extent to which a model can be validated by black-box testing alone; it also means, as usual for a loose specification, that it is important to reason about software w.r.t. the architecture rather than w.r.t. current implementations, at least if it is intended to be portable to future implementations.

Contrasting again with Power, there such non-multi-copy-atomic behaviour can be understood in a different way, from the cache protocol alone acting in a symmetric topology [16, §12]. In this sense the Flowing model is not a good intensional representation of Power microarchitectures, and there are litmus tests that are observable on Power implementations that Flowing does not allow. But the POP model permits those, and it is a plausible basis for a good common model.

Out-of-order execution Turning to the thread model that we use for both Flowing and POP, ARM and Power implementations differ in just how much out-of-order execution they exhibit. The MP+dmb.sy+fri-rfi-ctrlisb (message-passing) litmus test below is a message-passing test where the writes of Thread 0 are maintained in order by a dmb sy barrier and the reads c and f of Thread 1 are separated by another write to y (which is coherence[2] after the write to y in Thread 0, indicated by co edge), a read of y, a branch, and an isb barrier (indicated by a ctrlisb edge); the question is whether the instructions in Thread 1 create enough order to prevent the read of x reading from the initial state 0.

Test MP+dmb.sy+fri-rfi-ctrlisb: Allowed
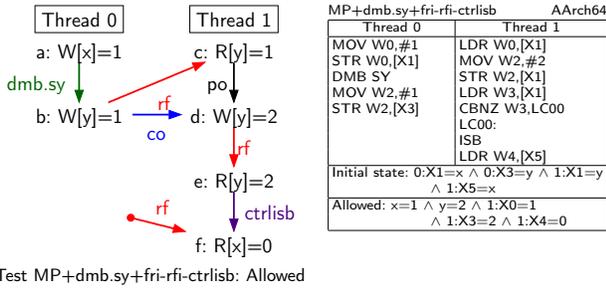
The PLDI11 Power model [22] forbids this behaviour as the read c, the write d, the read e, the isync (the Power analogous to isb) and the read f all have to commit in program order. The read c can be committed only after it has been satisfied (by the write b) which means the write b has propagated to Thread 1. Write b can propagate to Thread 1 only after the lwsync (the Power lightweight memory barrier, weaker than dmb sy) and the write a have propagated to Thread 1. Therefore, when the read f is satisfied it can only be from the write a (or coherence-after write).

In testing, the above is observed on some ARM machines ([4, Table VI] and [16, §10.6]). To allow this behaviour read e has to be able to commit before read c gets its value back. ARM architects give two distinct explanations for this behaviour. In the first explanation the microarchitecture handles reads in two distinct steps, with a read request being issued and then satisfied. In this example, after the read c has been issued the microarchitecture can guarantee its coherence order even though it has not been satisfied yet, by keeping read requests and writes to the same address ordered in the Flowing hierarchy, and continue by committing the program-order–following write to the same address d. This enables e to be committed, which resolves the control dependency and allows f to be issued and satisfied with the initial value (0), before c was satisfied.

The second explanation is that the write d is independent of the read c in every respect except coherence. This prevents d from being committed but it does not prevent the thread from forwarding d to the po-following read e which in turn can be committed before d (as the address and value of d are fixed).

Our Flowing model incorporates both of these mechanisms.

---

[2] Coherence is the per-location order over writes to that location that conventional multiprocessor architectures all provide.

4. Flowing Model Design

We now discuss a selection of further aspects of the Flowing design.

4.1 Coherence

Relaxed-memory processors exhibit much programmer-observable reordering of memory actions, but they typically do aim to guarantee some *coherence* property: with loads and stores to a single location appearing to execute in a global linear order.

The PLDI11 Power model guaranteed this in the storage subsystem by fiat, by maintaining a partial order of the coherence commitments between writes made so far, and in the thread semantics by allowing writes (to the same address) to be committed, and reads (to the same address) to be satisfied, only in program order.
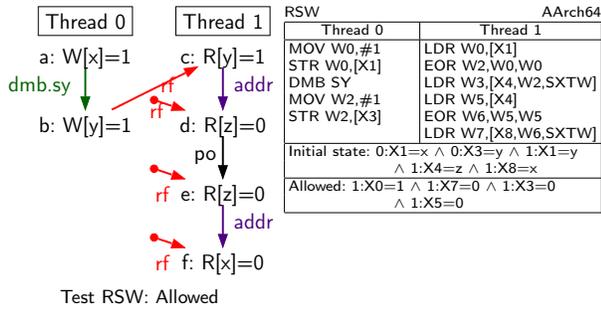
Here the Flowing storage subsystem can easily maintain writes and read requests to the same address in-order as they flow down the hierarchy, but the thread semantics has to be more liberal to match the architects' intent, and this is further complicated by the need to restart reads. In the following paragraphs we explain how the thread subsystem guarantees the coherence order of two writes will match their program order (write-write coherence), and the coherence of two writes that were read by two reads will match the program order of the reads (read-read coherence).

Write-write coherence Simply committing writes to the same address in program order would exclude the LB+data+data-wsi (load-buffering) litmus test below, which the ARM architects intend to be allowed (to date, we have not observed this behaviour in testing). We resolve this in the model by allowing writes to be committed out of order, and only passing a write to the storage when no other po-after write to the same address has been committed. This means we can commit and pass to the storage subsystem the write e: W[x]=2 before the data dependency in Thread 1 is resolved, and only later commit d: W[x]=1, without ever passing it to the storage subsystem. In hardware, the two writes might be merged together.

Test LB+data+data-wsi: Allowed
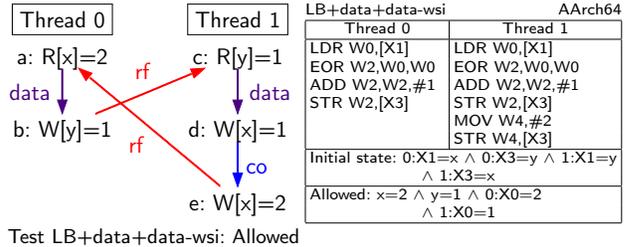
If the dependency from the read c to the write d was an address dependency the behaviour would be forbidden by the architecture, as resolving the address of the write d might raise an exception, in which case the write e must not be visible to other threads. The models therefore distinguish between the two cases by allowing a write to be committed only after all po-previous addresses have been fully determined.

Read-read coherence The RSW (read-from-same-write) litmus test below challenges the coherence order of reads. The model allows it in the same way as hardware, by allowing read e to be issued before the address dependency from c to d is resolved, and therefore before the read d to the same address is issued. After e is satisfied by the storage subsystem with the initial value, read request f can be issued and satisfied with the initial value as well. Only at that point can the instructions of Thread 0 commit in order and the writes and the barrier flow all the way into memory. In turn, Thread 1 issues read c and the storage satisfies it with the write b that just flowed into memory. This resolves the address dependency and allows read d to be issued and satisfied from the initial state.

Thread 0

a: W[x]=1

dmb.sy

b: W[y]=1

Thread 1

c: R[y]=1

rf / rf / addr

d: R[z]=0

po

rf / e: R[z]=0

addr

rf / f: R[x]=0

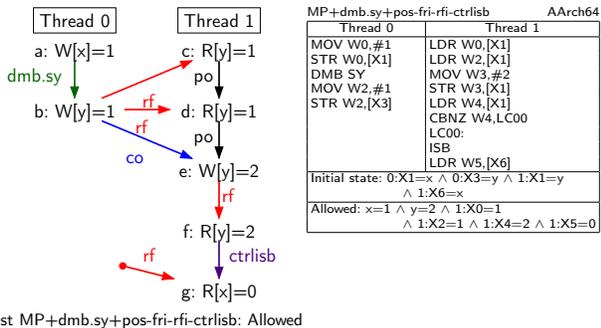| RSW	AArch64
Thread 0	Thread 1
MOV W0,#1	LDR W0,[X1]
STR W0,[X1]	EOR W2,W0,W0
DMB SY	LDR W3,[X4,W2,SXTW]
MOV W2,#1	LDR W5,[X4]
STR W2,[X3]	EOR W6,W5,W5
	LDR W7,[X8,W6,SXTW]
Initial state: 0:X1=x ∧ 0:X3=y ∧ 1:X1=y ∧ 1:X4=z ∧ 1:X8=x	
Allowed: 1:X0=1 ∧ 1:X7=0 ∧ 1:X3=0 ∧ 1:X5=0	

Test RSW: Allowed

Notice that there is no coherence violation between the reads d and e as both of them read from the same write. If read d was to read from a different write than e, then e would have had to be restarted as the write e reads from might be co-before the write d reads from.

It is thus because the model allows issuing reads out of order that the model must also perform read restarts. In fact the model also allows reads to be issued out of order with write commitments. Coherence is then maintained by restarting uncommitted reads in two situations. When a read r is satisfied by a write w, if exists a po-previous read that was issued after r (i.e. the reads were issued out of order), and has already been satisfied by a write that is different than w, and w was not forwarded to r then r is restarted; otherwise all po-following reads to the same address that have been issued before r and have already been satisfied are restarted, except for reads that have been satisfied by w and reads that have been satisfied by forwarding a write that is po-after r. When a write w is committed we restart all po-following reads to the same address that have been satisfied by a different write that is not po-after w, together with all po-following reads to the same address that have been issued and not satisfied yet.

When a read is restarted all its dataflow dependents are also restarted, and the storage subsystem removes any read request issued for the read.

The restart of reads on write commitment guarantees write-read coherence. Finally read-write coherence is maintained by requiring that when a write w is committed all po-previous reads to the same address have been issued and can not be restarted.

The MP+dmb.sy+pos-fri-rfi-ctrlisb litmus test, which is observed on ARM hardware, is explained by ARM architects by allowing the write e to be committed before the reads c and d are satisfied.

Thread 0

a: W[x]=1

dmb.sy

b: W[y]=1

Thread 1

c: R[y]=1

po

rf / d: R[y]=1

rf / po

co / e: W[y]=2

rf

f: R[y]=2

rf / ctrlisb

g: R[x]=0

MP+dmb.sy+pos-fri-rfi-ctrlisb	AArch64
Thread 0	Thread 1
MOV W0,#1	LDR W0,[X1]
STR W0,[X1]	LDR W2,[X1]
DMB SY	MOV W3,#2
MOV W2,#1	STR W3,[X1]
STR W2,[X3]	LDR W4,[X1]
	CBNZ W4,LC00
	LC00:
	ISB
	LDR W5,[X6]
Initial state: 0:X1=x ∧ 0:X3=y ∧ 1:X1=y ∧ 1:X6=x	
Allowed: x=1 ∧ y=2 ∧ 1:X0=1 ∧ 1:X2=1 ∧ 1:X4=2 ∧ 1:X5=0	

Test MP+dmb.sy+pos-fri-rfi-ctrlisb: Allowed

But to do so the model must guarantee read d will not be restarted when c is satisfied. To do so and maintain read-read coherence, the model allows write e to be committed only if the reads have been issued in order, and refrains from restarting reads that are satisfied out of order if they were issued in program order. Hence the model keeps track of the order in which reads are issued (again, following hardware).

4.2 New ARMv8 Memory Barriers

ARMv8 introduces a new variant of the dmb barrier, dmb ld. The regular dmb barrier orders arbitrary memory actions that occur before and after it, and also has cumulativity properties. Among other things, it orders stores that the barrier thread reads from before the dmb w.r.t. stores that it performs after, and orders stores (from the barrier thread) that occur before the barrier w.r.t. stores (from any thread) that appear logically after the barrier.

The new dmb ld orders loads that occur before the barrier with load and stores that occur after the barrier, and dmb st (also in ARMv7) orders stores that occur before the barrier with stores that occur after the barrier.

Ongoing discussions with ARM architects reveals that they intend these barriers to have weaker cumulative properties than the dmb sy, but their exact strength is still subject to debate. Our current understanding is that dmb st only has a cumulativity property with respect to stores that occur after the barrier; the situation for dmb ld is less clear and is currently under discussion in ARM.

4.3 Load-acquire/Store-release

Another addition in AArch64 is the load-acquire and store-release instructions. According to the ARM ARM, store-release is multi-copy-atomic when observed by load-acquires, a strong property that conventional release-acquire semantics does not imply. Furthermore, despite their names, these instructions are intended to be used to implement the C11 sequentially consistent load and store.

From their description in the ARM ARM, these two instructions behave somewhat like two halves of a dmb sy, where the store-release enforces cumulative order with po-previous instructions and load-acquire enforces order with po-later instructions. One might expect this will be enough to guarantee the multi-copy atomicity of store-release, but this is not the case. In the Flowing model one can imagine a read request from a load-acquire being satisfied by a write from a store-release when the two requests are adjacent in some (non root) queue. Before the read satisfaction, the load/store pair behaves like a dmb sy, preventing instructions from being reordered with them. But after the read is satisfied, half of the implicit dmb sy disappears, and instructions can be reordered with the write, breaking its multi-copy atomicity.

To guarantee the multi-copy atomicity of store-release, the Flowing model does not allow a read from load-acquire to be satisfied from a store-release write until the write has reached the memory. In addition, when such a read is satisfied from a regular write, the read does not disappear from the storage subsystem as regular reads do, instead it is marked as satisfied and reordered with the write that satisfied it. This semantics has some surprising effects the ARM architects did not expect, in particular, the litmus test in [7, §J10.2.4] (IRIW+poaas+LL) can be weakened by changing the second load in P3 and P4 to regular loads (IRIW+poaps+LL) and the behaviour will still be forbidden. The implications of these are currently being discussed with ARM.

4.4 Dependencies

We say there is a load-store dependency between a load and a following store instructions, if there is a data flow from the register holding the value loaded by the load and the registers used by the store, and similarly for load-load dependency. The architecture guarantees some ordering for instructions that have load-load and load-store dependencies. Currently, ARM ARM [7, §B2.7.4] makes a distinction between 'true' and 'false' load-store data dependencies, with the intuition being that for a 'false' dependency the value read does not extensionally affect the value stored. Making this precise in a satisfactory way is problematic, as too liberal a notion of 'false' dependency (allowing more hardware optimisation) can also make it impractical to compute whether a trace is

admitted by the architecture or not. This question is currently under discussion with ARM, after we drew it to their attention, and it is possible the semantics will be strengthened to not distinguish between the two kinds of dependency; our model follows that proposal at present.

4.5 Branch Prediction

Control dependencies from a load to another load are not in general respected, as hardware can speculatively reach and satisfy the second load early. For a computed branch, in principle the branch prediction hardware might guess an arbitrary address (or at least an arbitrary executable-code-page address); there seems to be no reasonable way to limit this behaviour, and one can construct litmus tests that observe it. Mathematically that is no problem, but to make the tool based on our model usable in practice, we have to approximate here, otherwise the nondeterminism (picking addresses that turn out to be incorrect) would be prohibitive.

A detailed prose description of the Flowing thread and storage subsystem is given in §7.

5. Introduction to the POP Model

We now show how the POP model abstracts from the hierarchical storage subsystem structure of the Flowing model, to give a better programming model (and to combine the best of both worlds from the pros and cons of the Flowing abstraction listed in §3).

The purpose of the queues in the Flowing storage subsystem is to maintain order between requests, relaxing it subject to the reordering condition. The topology, on the other hand, determines the order in which a request becomes visible to the threads. Consider for example the write a:W[x]=1 in the first storage subsystem state diagram of §3 for WRC+addrs. It is visible to threads 0 and 1, as a read request from each of these threads can potentially flow and encounter a, but it is not visible to threads 2 and 3, as a read request issued by those would flow all the way to memory without encountering a. In the POP model we make these two notions explicit. Instead of the queues enforcing an implicit order we record an order (as a graph) over all requests, and instead of letting a fixed topology determine the order in which a request becomes visible, we record for each thread a set of requests that have become visible to it, and we allow requests to be added to these sets (subject to conditions over its place in the order).

Notice that unlike the PLDI11 storage coherence order, which is only over writes, and only relates writes to the same address, the POP storage model records an order over all requests (writes, reads and barriers) and it relates requests with different addresses. In addition, POP records propagation sets (which do not add ordering) as opposed to the PLDI11 propagation queues (which play a significant role in ordering). Moreover, in the PLDI11 model, when an event is propagated to a new thread, it takes its place in the head of the propagation queue, while in the POP model, requests that propagated to a new thread can do so from the middle of the order. Further, in the PLDI11 model the storage subsystem receives a read request from a thread and replies to it with a read response in an atomic transition, while in POP the storage subsystem receives the request and replies to it in two distinct transitions. Finally, as mentioned in the introduction, the Power sync memory barrier requires an acknowledgement to be passed from the storage subsystem to the issuing thread, while the POP model maintain memory-barrier-induced ordering without such acknowledgements.

5.1 Example: POP Simulating Flowing

In the following we will demonstrate an execution of the MP+dmb.sy+addr litmus test below. On the thread subsystems side we will do our best to make the second load read 0. On the storage subsystem side we will first show how the Flowing model behaves and then an analogous behaviour of the POP model.

Since there is an address dependency between the first load and the second load in Thread 1, the thread has to start by issuing the first load, c. In the Flowing storage subsystem, read c is received from Thread 1 and placed at the top of the queue associated with that thread. Similarly, in the POP storage subsystem, read c is recorded as propagated to Thread 1.

In Thread 0 now, the presence of a dmb sy between the two stores forces them (and the dmb sy) to be committed in program order. The Flowing storage model places these requests in the queue associated with Thread 0, one on top of the other, in the order they were received (i.e. program order). The POP storage model records each of these instructions as propagated to Thread 0 when it is received. In addition, when the dmb sy request is received it is recorded as being ordered after the previously received write a and when the write b is received it is recorded as being ordered after the dmb sy.

Test MP+dmb.sy+addr: Forbidden
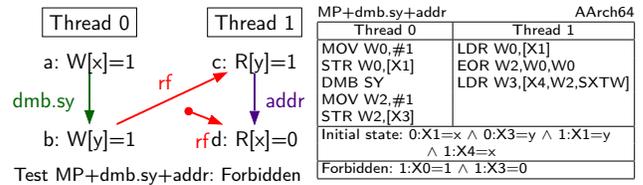
The Flowing storage model must flow the write b to the root queue before the read c gets there in order for c to be satisfied by b. But before it can do that it must flow the dmb sy and write a to the root, as none of these can be reordered with the others. Similarly, the POP storage subsystem must propagate the write b to Thread 1 before propagating the read c to Thread 0 in order for c to be satisfied by b. But before it can do that it must propagate the dmb sy and write a, as they are ordered before b. When the dmb sy and write b are propagated to Thread 1 they are recorded as being ordered before c.

Back in the Flowing storage subsystem, read c can flow to the root queue and get satisfied by b. In the POP storage subsystem, read c can propagate to Thread 0 and get satisfied by b.

After c was satisfied the address dependency is resolved and Thread 1 can issue read d. In the Flowing storage model, d is placed at the top of the queue associated with Thread 1 and it is clear it will not be able to read the initial value of x as the write a is in the root queue. Similarly in the POP storage model, d is recorded as propagated to Thread 1 and ordered after the dmb sy and write a, and therefore d will not be able to read the initial value of x.

A detailed prose description of the POP storage subsystem is given in §8. In the next section we describe the ISA model.

6. ISA Model

Handling the instruction semantics raises two problems, as noted in the introduction: the mass of detail involved, and its semantic integration with the concurrent semantics. We address both with a domain-specific language for ISA description, Sail.

The ARM ARM has 224 application-level non-FP/SIMD instructions, counting subsections of Chapter C6 *A64 Base Instruction Descriptions* (each of which defines one instruction, sometimes with several variants). Of these 224, we support all except 21 (and aspects of 4 more), relating to exceptions, debug, system-mode, and load-non-temporal-pair.

Fig. 2 shows excerpts of this. At the top is a clause of an instruction AST datatype (ast), with constructor AddSubImmediate, covering four ADD and SUB instructions. Then there is

a decoding function `decodeAddSubtractImmediate` that pattern-matches a 32-bit vector and constructs an element of that `ast` type (the body of this function is elided). For example, the ARM assembly instruction `ADD X1, X2, #1` would be assembled to `0x91000441`, which is decoded to `AddSubImmediate(1,2,64,0,0,ExtendType_UXTB,1)`. Finally there is a clause of the `execute` function that defines the behaviour of these instructions.

Hardware vendors differ widely in the level of detail and rigor of their instruction descriptions. The ARM ARM is relatively good in these respects: the pseudocode used is reasonably complete and close to something that could be executed, at least for sequential code (Shi et al. build a simulator based on pseudocode extracted from an ARMv7 pdf [25]). We therefore want to follow it closely, both to avoid introducing errors and to keep our definitions readable by engineers who are already familiar with the ARM ARM. Sail is designed with this in mind, and our instruction descriptions can often be line-for-line identical to those of the ARM ARM. Looking again at Fig. 2, the body of the Sail `execute` clause differs from the ARM ARM text [7, §C6.6.4, *ADD (immediate)*] only in minor details: the type annotations and concrete syntax for `let`. Both languages are imperative, with mutable local variables and primitives for register and memory reads and writes, and typed bit vectors.

For Sail, we developed a type system expressive enough to type-check such code without requiring excessive type annotation or partiality. The ISA description manipulates bitvectors of differing sizes and start-indices, which we support with dependent types over `Int`. For example, Fig. 2 includes a variable `'R` of kind `Int`, bounded to be either `32` or `64`, to accommodate those two flavours of instructions; the code involves bitvector types `bit['R]` of that length and also a singleton type `[:'R:]` (really, an integer range type from `'R` to `'R`). This instruction happens not to index into a vector; in those that do, the indices use such range types (not necessarily singletons). Type inference and checking involve polynomial equations and inequalities, undecidable in general but not a problem in practice, as the constraints that actually arise here are relatively simple. Sail also provides implicit type coercions between numbers, vectors, individual bits, and registers, again keeping the specification readable. A simple effect system tracks the presence of memory and register operations, identifying pure code. The language has first-order functions and pattern-matching, including vector concatenation patterns (as used in `decodeAddSubtractImmediate`). We have also used Sail for related work on IBM Power ISA semantics [12] (Power uses a broadly similar but different and less rigorous pseudocode); it is expressive enough for both.

The dynamic semantics of Sail is where we see the integration with the concurrency thread-subsystem semantics: unlike a sequential or sequentially consistent system, we cannot simply use a state-monad semantics that updates a global register and memory state. Instead, the Sail semantics (expressed as an interpreter in Lem) makes register and memory requests to the thread semantics, providing a continuation of the instruction state. It also has to announce to the thread semantics the point at which the address of a memory write becomes available – an example where we differ from the ARM ARM pseudocode as written there, adding extra information. The thread semantics further needs to know the register footprint of an instruction and intra-instruction register dataflow information (e.g. the register reads that may feed into a memory address); the Sail interpreter calculates these with an exhaustive symbolic taint-tracking execution.

7. The Flowing Model in Detail

7.1 The Storage Subsystem/Thread Interface

The model is expressed in terms of read, write, and barrier requests. Read and write requests include the kind of the memory access (e.g. exclusive, release, acquire), the ID of the issuing thread and the memory access address. Write requests also include a value. Barrier requests include the issuing thread ID.

When we refer to a write or read request without mentioning the kind of request we mean the request can be of any kind.

The storage subsystem and a thread subsystem can exchange messages through synchronous transitions:

- a write request can be passed to the storage subsystem by a thread Commit store instruction transition coupled with a storage subsystem Accept request transition;
- a (memory) barrier request can be passed to the storage subsystem by a thread Commit barrier instruction transition coupled with a storage subsystem Accept request transition;
- a read request can be passed to the storage subsystem by a thread Issue read request transition coupled with a storage subsystem Accept request transition; and
- a read response can be returned from the storage subsystem to a thread by a storage subsystem Satisfy read from segment or Satisfy read from memory transition coupled with a thread Satisfy memory read from storage response transition.

7.2 Storage Subsystem States

The Flowing storage subsystem state includes:

- *thread_ids*, the set of thread IDs that exist in the system;
- *topology*, a data structure describing how the segments are connected to each other;
- *thread_to_segment*, a map from thread IDs to segments, associating each thread with its leaf segment;
- *buffers*, a map from segments to list of requests associating each segment with the requests queued in that segment;
- *reordered*, a set of request pairs that have been reordered w.r.t. each other; and
- *memory*, a map from memory addresses to the most recent write request to that address to reach memory.

7.3 Storage Subsystem Transitions

Accept request   A request *r* from thread *r.tid* can be accepted if:
1. *r* has not been accepted before (i.e. *r* is not in *buffers*); and
2. *r.tid* is in *thread_ids*.

Action: add *r* to the top of *buffers(thread_to_segment(r.tid))*.

Flow request   A request *r* can flow from segment *s1* to *s2* if:
1. *r* is at the bottom of *buffers(s1)*; and
2. *s1* is a child of *s2* in *topology*.

Action:
1. remove *r* from *buffers(s1)*;
2. add *r* to the top of *buffers(s2)*; and
3. remove from *reordered* any pair that contains *r*.

Reorder requests   Two requests *r_new*, *r_old* that appear consecutively in *buffers(s)* (*r_new* nearer the top) can be reordered if:
1. (*r_new*, *r_old*) does not appear in *reordered* (i.e. they have not been reordered (in segment *s*) with each other before (preventing live lock); and
2. *r_new* and *r_old* satisfy the *reorder condition* (§7.4).

Action:
1. switch the positions of *r_new* and *r_old* in *buffers(s)*; and
2. record the reordering of *r_new* and *r_old* (by adding the pair *(r_new, r_old)* to *reordered*).

Satisfy read from segment Two requests, *r_read*, *r_write*, can generate a read response to thread *r_read.tid* if:

1. *r_read* is a read request ;
2. *r_write* is a write request;
3. *r_read*, *r_write* appear consecutively in *buffers(s)* for some segment *s*, with *r_read* closer to the top (newer); and
4. *r_read* and *r_write* are to the same address.

Action:

1. send a read response for request *r_read* to thread *r_read.tid*, containing *r_write*; and
2. remove *r_read*.

Satisfy read from memory A read request *r_read* from thread *r_read.tid* can generate a read response to thread *r_read.tid* if *r_read* is at the bottom of *buffers(s)*, where *s* is the root segment in *topology*. Action:

1. send a read response for request *r_read* to thread *r_read.tid*, containing the write stored in *memory* for the address *r_read.addr*; and
2. remove *r_read*.

Flow write to memory The write request *r_write* can be stored into memory if: *r_write* is at the bottom of *buffers(s)*, where *s* is the root segment in *topology*. Action:

1. update *memory* to map the address *r_write.addr* to *r_write*; and
2. remove *r_write* from *buffers(s)* and *reordered*.

Flow barrier to memory A barrier request *r_barr* can be discarded if: *r_barr* is at the bottom of *buffers(s)*, where *s* is the root segment in *topology*. Action: remove *r_barr*.

7.4 Auxiliary Definitions for Storage Subsystem

Reorder condition Two requests *r_new* and *r_old* are said to meet the *reorder condition* if:

1. neither one of *r_new*, *r_old* is a dmb sy; and
2. if both *r_new* and *r_old* are memory access requests, they are to different addresses.

7.5 Thread States

The state of a single hardware thread includes:

- *thread_id*, a unique identifier of the thread;
- *register_data*, general information about the available registers, including name, bit width, initial bit index and direction;
- *initial_register_state*, the initial values for each register;
- *initial_fetch_address*, the initial fetch address for this thread;
- *instruction_tree*, a data structure holding the instructions that have been fetched in program order; and
- *read_issuing_order*, a list of read requests in the order they were issued to the storage subsystem.

7.6 Instruction State

Each instruction in the *instruction_tree* has a state including:

- *program_loc*, the memory address where the instruction's opcode resides;
- *instruction_kind*, the kind of the instruction (e.g. load-acquire);
- *regs_in*, the input registers, for dependency calculation;
- *regs_out*, the output registers, for dependency calculation;
- *reg_reads*, accumulated register reads;
- *reg_writes*, accumulated register writes;
- *mem_read*, one of *none*, *pending a*, *requested a*, *write_read_from w*, where *a* is a memory address and *w* is a memory write;
- *mem_write*, one of *none*, *potential_address a*, *pending w*, *committed w*, where *a* is a memory address and *w* is a memory write;

- *committed*, set to *true* when the instruction can no longer affect the memory model (the Sail interpreter might still need to complete the execution of the ISA definition);
- *finished*, set to *true* only after *committed* is set to true and the Sail interpreter has finished executing the ISA definition; and
- *micro_op_state*, the meta-state of the instruction, one of *plain interpreter_state* (ready to make a Sail interpreter transition from *interpreter_state*), *pending_mem_read read_cont* (ready to perform a read from memory) or *potential_mem_write write_cont* (ready to perform a write to memory) where *read_cont* is a Sail interpreter continuation that given the value read from memory returns the next interpreter state, and *write_cont* is a Sail interpreter continuation that given a boolean value (that indicates if the store was successful) returns the next interpreter state.

Instructions that have been fetched (i.e. in *instruction_tree*) and have a *finished* value of *false* are said to be *in-flight*. Instructions that have a *committed* value of *true* are said to be *committed*. Load instructions with the value *requested* for *mem_read* are said to have an *outstanding read request*, and if they have the value *write_read_from* they are said to be *satisfied*. Store instructions with *mem_write = pending w* are said to have a *pending write*. We say instruction *i* has *fully determined address* if all instructions that write to input registers of *i* that affect memory address calculation in the ISA definition of *i* are committed. We say *i* is *fully determined* if all instructions that write to input registers of *i* are committed.

7.7 Thread Transitions

Sail interpreter step An instruction *i* in meta-state *plain interpreter_state* can perform an interpreter step as follows:

- if *interpreter_state* indicates a memory-read event: set *i.mem_read* to *pending interpreter_state.read_addr* and update the instruction meta-state to *pending_mem_read interpreter_state.read_cont*;
- if *interpreter_state* indicates a memory-write-address event: set *mem_write* to *potential_address interpreter_state.write_addr* and update the instruction meta-state to *plain interpreter_state.next*;
- if *interpreter_state* indicates a memory-write-value event: set *mem_write* to *pending w* (where *w* is a write request with the value *interpreter_state.write_value* and the address that was in *mem_write* before) and update the instruction meta-state to *potential_mem_write interpreter_state.write_cont*;
- if *interpreter_state* indicates a register-read event: look in *instruction_tree* for the most recent po-previous instruction, *i'* that has *interpreter_state.reg* in *i'.regs_out*. If *interpreter_state.reg* is not in *i'.reg_writes* the transition is disabled. Otherwise, add *interpreter_state.reg* to *i.reg_reads* and update the meta-state of *i* to *plain (interpreter_state.reg_count val)* (where *val* is the value written to the register by *i'*);
- if *interpreter_state* indicates a register-write event: add *interpreter_state.reg* to *i.reg_writes* and update the meta-state of *i* to *plain interpreter_state.next*; and
- if *interpreter_state* indicates an internal step: update the meta-state of *i* to *plain interpreter_state.next*.

Fetch instruction An instruction *i*, from address *loc*, can be fetched, following its program-order predecessor *i'* if:

1. *i'* is in *instruction_tree*;
2. *loc* is a possible next fetch address for *i'* according to the ISA model;
3. none of the successors of *i'* in *instruction_tree* are from *loc*; and
4. *i* is the instruction of the program at *loc*.

The possible next fetch addresses allow speculation past calculated jumps and conditional branches; they are defined as:

1. for a non-branch/jump instruction, the successor instruction address;
2. for a jump to constant address, that address;
3. for a conditional branch, the possible addresses for a jump[3] together with the successor; and
4. for a jump to an address which is not yet fully determined (i.e., where there is an uncommitted instruction with a dataflow path to the address), any address. This is (necessarily) approximated in our implementation, c.f. §4.5.

Action: construct an initialized instruction instance, including static information available from the ISA model such as *instruction_kind*, *regs_in*, *regs_out*, and add it to *instruction_tree* as a successor of *i'*.

This is an internal action of the thread, not involving the storage subsystem, as we assume a fixed program rather than modelling fetches with memory reads; we do not model self-modifying code.

Issue read request An in-flight instruction *i* in meta-state *pending_mem_read read_cont* can issue a read request of address *a* to the storage subsystem if:

1. *i.mem_read* has the value *pending a*, i.e., any other reads with dataflow path to the address have been satisfied, though not necessarily committed, and any arithmetic on such a path completed; and
2. all po-previous dmb sy and isb instructions are committed.

Action:

1. send a read-request to the storage subsystem;
2. change *i.mem_read* to *requested a*; and
3. update *read_issuing_order* to note that the read was issued last.

Satisfy memory read from storage response A read response with write *w*, for read request from instruction *i* in meta-state *pending_mem_read read_cont* can always be received. Action:

1. if *i.mem_read* does not have the value *requested* the response is ignored (this can happen when the read is satisfied by write forwarding while waiting for the response); else
2. if there exists a po-previous load instruction to the same address that was issued after *i* (i.e., issued out of order) and was satisfied by a write that is not *w*, set *i.mem_read* to *pending*; else
3. for every in-flight instruction *i'* that is po-after *i* and has read from a write to the same address as *i* that is not *w* and not po-successor of *i*, restart *i'* and its data flow dependents;
4. change *i.mem_read* to *write_read_from w*; and
5. update the meta-state of *i* to *plain (read_cont w.value)*.

Satisfy memory read by forwarding an in-flight write A pending memory write *w* from an in-flight store instruction *i'* can be forwarded directly to a load instruction *i* in meta-state *pending_mem_read read_cont* if:

1. *i.mem_read* has the value *pending w.address* or *requested w.address*; and
2. *i'* is po-before *i*, there is no other store instruction to the same address in between them, and there is no other load instruction in between them that has read from a different store instruction to the same address.

Action:

1. for every in-flight instruction *i"* that is po-after *i* and has read from a write to the same address as *i* that is not *w* and not po-successor of *i*, restart *i"* and its data flow dependents;
2. change *i.mem_read* to *write_read_from w*; and
3. update the meta-state of *i* to *plain (read_cont w.value)*.

---

[3] In AArch64, all the conditional branch instructions have a constant address.

Commit store instruction A store instruction *i* in meta-state *potential_mem_write write_cont* and with pending write *w* can be committed if:

1. *i* is fully determined;
2. all po-previous conditional branches are committed;
3. all po-previous dmb sy and isb instructions are committed;
4. all po-previous memory access instructions have a fully determined address; and
5. all po-previous instructions that read from the same address have either an outstanding read request or are satisfied, and cannot be restarted (see §7.8).

Action:

1. restart any in-flight loads (and their dataflow dependants) that:
   (a) are po-after *i* and have read from the same address, but from a different write and where the read could not have been by forwarding an in-flight write that is po-after *i*; or
   (b) have issued a read request that has not been satisfied yet.
2. if there is no committed po-following store to the same address, send a write request to the storage subsystem;
3. record the store as committed (i.e. set *i.mem_write* to *committed w* and *committed* to *true*); and
4. update the meta-state of *i* to *plain (write_cont true)*.

Commit barrier instruction A barrier instruction *i* in meta-state *plain interpreter_state* can be committed if:

1. *interpreter_state* indicates a barrier event is pending;
2. all po-previous conditional branches are committed;
3. all po-previous barriers (of any kind) are committed;
4. if *i* is isb, all po-previous memory access instructions have a fully determined address; and
5. if *i* is dmb sy, all po-previous memory access instructions are committed.

Action:

1. record the barrier as committed (i.e. set *committed* to *true*); and
2. update the meta-state of *i* to *plain interpreter_state.next*.

Finish instruction An in-flight instruction *i* in meta-state *plain interpreter_state* can be finished if:

1. *interpreter_state* indicates the execution of the ISA definition has finished;
2. if *i* is a load instruction:
   (a) all po-previous dmb sy and isb instructions are committed;
   (b) let *i'* be the store instruction to the same address as *i* that appears last in program order before *i*:
       i. if *i'* was forwarded to *i*, *i'* is fully determined, otherwise *i'* is committed;
       ii. all memory access instructions, po-between *i'* and *i*, have a fully determined address; and
       iii. all load instructions to the same address as i, that are po-between *i'* and *i*, are committed.
3. *i* is fully determined; and
4. all po-previous conditional branches are committed.

Action:

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

7.8 Auxiliary Definitions for Thread Subsystem

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

1. there exists an in-flight store instruction *s* such that applying the action of the Commit store instruction transition to *s* will result in the restart of *i*;
2. there exists an in-flight load instruction *l* such that applying the action of the Satisfy memory read from storage response transition to *l* will result in the restart of *i* (even if *l* is already satisfied);
3. *i* has an outstanding read request that has not been satisfied yet, and there exists a load instruction po-before *i* that has an outstanding read request to the same address (maybe already satisfied) after *i*; or
4. there exists an in-flight instruction *i'* that might be restarted and an output register of *i'* feeds an input register of *i*.

8. The POP Model

8.1 The Storage Subsystem/Thread Interface

The storage subsystem and a thread subsystem can exchange messages through synchronous transitions:

- a write request can be passed to the storage subsystem by a thread Commit store instruction transition coupled with a storage subsystem Accept request transition;
- a (memory) barrier request can be passed to the storage subsystem by a thread Commit barrier instruction transition coupled with a storage subsystem Accept request transition;
- a read request can be passed to the storage subsystem by a thread Issue read request transition coupled with a storage subsystem Accept request transition; and
- a read response can be passed from the storage subsystem to a thread by a storage subsystem Send read-response to thread transition coupled with a thread Satisfy memory read from storage response transition.

In addition to the above, when a load instruction is restarted in the thread subsystem, all its read-requests are removed from the storage subsystem.

8.2 Storage Subsystem State

The POP storage subsystem state includes:

- *thread_ids*, the set of thread IDs that exist in the system;
- *requests_seen*, the set of requests (memory read/write requests and barrier requests) that have been seen by the subsystem;
- *order_constraints*, the set of pairs of requests from *requests_seen*. The pair *(r_old,r_new)* indicates that *r_old* is before *r_new* (*r_old* and *r_new* might be to different addresses and might even be of different kinds); and
- *requests_propagated_to*, a map from thread IDs to subsets of *requests_seen*, associating with each thread the set of requests that has propagated (potentially visible) to it.

8.3 Storage Subsystem Transitions

Accept request A request *r_new* from thread *r_new.tid* can be accepted if:

1. *r_new* has not been accepted before (i.e., *r_new* is not in *requests_seen*); and
2. *r_new.tid* is in *thread_ids*.

Action:

1. add *r_new* to *requests_seen*;
2. add *r_new* to *requests_propagated_to(r_new.tid)*; and
3. update *order_constraints* to note that *r_new* is after every request *r_old* that has propagated to thread *r_new.tid*, and *r_new* and *r_old* do not meet the Flowing *reorder condition* (see Reorder condition).

Propagate request to another thread The storage subsystem can propagate request *r* (by thread *tid*) to another thread *tid'* if:

1. *r* has been seen before (i.e., *r* is in *requests_seen*);
2. *r* has not yet been propagated to thread *tid'*; and
3. every request that is before *r* in *order_constraints* has already been propagated to thread *tid'*.

Action:

1. add *r* to *requests_propagated_to(tid')*; and
2. update *order_constraints* to note that *r* is before every request *r_new* that has propagated to thread *tid'* but not to thread *tid*, where *r_new* and *r* do not meet the Flowing *reorder condition* (see Reorder condition).

Send read-response to thread The storage subsystem can send a read-response for read request *r_read* to thread *r_read.tid* containing the write request *r_write* if:

1. *r_write* and *r_read* are to the same address;
2. *r_write* and *r_read* have been propagated to (exactly) the same threads;
3. *r_write* is *order_constraints*-before *r_read*; and
4. any request that is *order_constraints*-between *r_write* and *r_read* has be *fully-propagated* (§8.4) and is to a different address.

Action:

1. send thread *r_read.tid* a read-response for *r_read* containing *r_write*; and
2. remove *r_read*.

8.4 Auxiliary Definitions for Storage Subsystem

Fully propagated Request *r* is said to be fully-propagated if it has been propagated to all threads and so has every request that is *order_constraints*-before it.

Removing read request When a read request is removed from the storage subsystem, due to restart of the instruction in the thread subsystem or satisfaction, first *order_constraints* is restricted to exclude the request; it is then further restricted by applying the *reorder condition* to each pair of ordered requests and removing pairs that can be reordered; finally the transitive closure of the result is calculated.

9. POP Abstracts Flowing

We now show that the POP model does indeed abstract the Flowing model, as intended. The detailed proof is available online; here we outline the statement of the theorem and the overall structure of the proof. A Flowing *trace* is a sequence

$$(tss_0, flo_0), a_1, (tss_1, flo_1), \ldots, a_n, (tss_n, flo_n)$$

where each tss_i denotes a thread subsystem state, flo_i denotes a Flowing storage subsystem state and for each $i \in [1, n]$ the transition a_i is enabled at the Flowing system state $f = (tss_{i-1}, flo_{i-1})$ and when taken leads to $f' = (tss_i, flo_i)$, written as $f \xrightarrow{a_i}_{\mathcal{F}} f'$. We define the dual notion for POP traces with a POP system state (tss, pop), where pop is a POP storage subsystem state. We say that a Flowing trace $tr_{\mathcal{F}}$ is equivalent to a POP trace $tr_{\mathcal{P}}$ if the last system state in each trace has identical thread subsystem states.

We prove that POP soundly relaxes Flowing by establishing a simulation relation from Flowing to POP which for a given Flowing trace generates an equivalent POP trace. Since the thread subsystem state has full information about the execution of the instructions, equivalence of traces implies identical program behaviour.

Theorem 1 (POP relaxes Flowing)**.** *Let* $tr_{\mathcal{F}}$ *be a Flowing trace ending at Flowing system state* (tss, flo). *Then there exists a POP*

trace $tr_{\mathcal{P}}$ which ends at POP system state (tss', pop) such that $tss = tss'$.

Our proof relies on a simulation relation σ and a transition map $\mu : A_{\mathcal{F}} \mapsto A_{\mathcal{P}}^*$. The simulation relation is such that whenever $(f, p) \in \sigma$, f and p have identical thread subsystem states. Besides the simulation relation, we also define a function $\mu : A_{\mathcal{F}} \mapsto A_{\mathcal{P}}^*$ which maps a Flowing transition into a (possibly empty) sequence of POP transitions. We then show inductively that for any Flowing system state f and POP system state p, if $(f, p) \in \sigma$ and there is a Flowing transition $f \xrightarrow{a}_{\mathcal{F}} f'$, then there is a POP system state p' such that $p \xrightarrow{\mu(a)}_{\mathcal{P}}^* p'$ and $(f', p') \in \sigma$.

10. Commutativities

Exploring all architecturally allowed outcomes of concurrent test programs is computationally expensive. In any state of the model multiple transitions might be enabled, and any possible trace of these transitions might produce a new outcome. From the definition of the model, however, it is clear that the order in which the abstract machine takes certain transitions does not matter: certain thread-internal transitions can be reordered and the outcome will be the same. We proved this for particular kinds of transitions and implemented an optimisation of the tool we describe below that uses this result to improve the performance of exhaustively exploring the possible behaviours of concurrent programs.

The property we proved is the following: assume transition t takes the abstract machine from state s_0 to s, and transition t' from s_0 to s'. Then for particular types of transitions t the model only needs to explore the traces starting with t, because in s transition t' is enabled again and either

- in s', transition t is enabled and taking t in s' produces the same state as taking t' in s; or
- taking transition t' in state s results in s'.

Theorem 2. *Let $t, t' \in$ enumerate-transitions-of-system s_0 and assume t is a thread-internal transition, an instruction-finish transition, a register read or write transition, a potential-memory-write transition, or a fetch transition, and neither t nor t' are fetch transitions directly po-after a branch. Then*

$$(t \in \text{enumerate-transitions-of-system } s' \wedge \tag{1}$$
$$t' \in \text{enumerate-transitions-of-system } s \wedge$$
$$\text{state-after-transition } s' \, t = \text{state-after-transition } s \, t') \vee$$
$$(t' \in \text{enumerate-transitions-of-system } s \wedge \tag{2}$$
$$\text{system-state-after-transition } s \, t' = s')$$

The proof (available online) is by case analysis on the transition kinds of t and t'. The reason the second clause in the statement above is needed is that t might be enabled by an instruction i that can be subject to restart, for example by a storage read response transition t', or part of a speculatively executed branch that might be aborted when the branch is resolved. In these cases, when taking transition t the instruction i makes progress which is "overwritten" when taking transition t'.

11. Exploration Tool

We have built a tool that lets one explore, interactively or exhaustively, all the executions admitted by the model for small test programs, either litmus tests or conventional ARMv8 ELF executables. The core of the tool is OCaml code automatically generated by the Lem compiler from the model definition (giving reasonable confidence that it is executing the model as specified), to which we add a driver and user interface code.

The exhaustive mode of the tool takes a litmus test as an input and produces a list of all the observable final states (using a memoised search), together with an example trace for each one; it also evaluates the litmus test final-state assertion. We use this on a cluster and a large server machine for bulk litmus test and ISA test validation.

The interactive mode lets the user make the nondeterministic choices between the available transitions at each state (or follow a previously identified sequence of choices). The user can also choose to let the tool eagerly take all the ISA internal transitions of each instruction and only prompt the user for the thread and storage subsystem transitions, or, further, to eagerly take transitions which are known to commute with others (§10), leaving the user only the choices that affect memory behaviour. At each point the tool displays the abstract state of the model, including the storage subsystem state, the tree of instruction instances for each hardware thread, and the Sail instruction state (the remaining Sail code and local variable environment) for each instruction. The delta from one state to the next is highlighted and the tool supports arbitrary 'undo', for ease of exploration. The tool provides both a command-line and web interface, using `js_of_ocaml` to compile to JavaScript that can run standalone in a browser. The web interface can be accessed from `http://www.cl.cam.ac.uk/~pes20/popl16-armv8`.

In small sequential tests the tool currently has a performance of the order of 90 IPS: a test adding up the numbers from 0 to 20 (an ELF binary compiled with GCC from a C program with a `for` loop), involving 212 instruction instances, takes 2.4 seconds to run using the Flowing model, 25.5 seconds in the POP model. Many optimisations are possible; for example, the POP model currently keeps all writes, but "sufficiently old" writes could be discarded (though the performance gain of doing such optimisations must be balanced against the cost of making the model less clear).

Exhaustive exploration of concurrent tests is intrinsically challenging due to the combinatorial explosion: using the POP model to compute all possible outcomes of the MP+dmb+addr litmus test without the commutativity optimisation of §10 takes 8 hours 35 minutes on an Intel Core i5 machine with 16GB RAM, 12 hours 23 minutes using the Flowing model. Using the commutativity property to avoid exploring equivalent traces improves the run time for this test to 4 seconds with POP and 5.5 seconds with Flowing.

12. Example

One of the benefits of our model and tool is to make it possible to explore the behaviour of intricate concurrent code with respect to the full envelope of behaviour allowed by the architecture, not just test it on particular implementations. Previous such tools have been useful both for Linux kernel developers [17] and ARM hardware engineers [personal communication], but were limited in many ways: to a concurrency model that was not well-validated w.r.t. ARM (and in fact did not match the intent in some respects), to a tiny fragment of the ISA, and to ARMv7; we have now relaxed all these limitations.

We demonstrate this for an ARMv8 spinlock implementation taken from the Linux kernel. The example uses two threads, each running a small critical section wrapped with lock and unlock functions. It assumes an initial state with register X0 holding the address of a lock, register X4 the address of a shared memory object, and register X5 (accessed as a half register with W5) a thread id.

The sample critical section for Thread 0 is on the next page; that for Thread 1 differs in taking the branch to `error` on loading 1 from the shared memory object (with a CBZ). If it does, then Thread 1 has been allowed into the critical section before Thread 0 has released the lock.

```
lock: LDAXR  W1, [X0]          unlock:
      ADD    W2, W1, #16, LSL #12   LDRH   W1, [X0]
      STXR   W3, W2, [X0]           ADD    W1, W1, #1
      CBNZ   W3, lock               STLRH  W1, [X0]
      EOR    W2, W1, W1, ROR #16    RET
      CBZ    W2, out
spin: LDAXRH W3, [X0]          ───────────────────────
      EOR    W2, W3, W1, LSR #16   // T0 critical section
      CBNZ   W2, spin              BL    lock
out:  RET                          STR   W5, [X4]
                                   LDR   W5, [X4]
                                   CBNZ  W5, error
                                   BL unlock
```

The spinlock uses several ARMv8 release/acquire instructions, some exclusive (load-linked/store-conditional pairs), namely LDAXR, load-acquire exclusive register, LDAXRH, load-acquire halfword, STXR, store exclusive register, and STLRH, store-release halfword. We explored this example interactively in both the Flowing and POP models, with both 32- bit and 64-bit instructions. In neither model were we able to exit the spin section of the lock code while the opposite thread had not passed the unlock section once the STXR had been executed. This was true whether the store had propagated to main memory or the opposite thread.

We then injected an error into this implementation, replacing STXR with a plain STR. In the resulting program, we found an execution trace in which the write of the store claiming the lock in Thread 0 does not propagate to Thread 1 before Thread 1 tests the lock, and so Thread 1 is also able to also claim the lock and enter its critical section, at which point the critical section of Thread 0 loads a 1 into W5 and the CBNZ instruction branches to error. Finding this error required that the store on Thread 0 propagated to Thread 1 only after Thread 1 also stored to the lock, which may or may not happen in an actual execution on hardware but which we could quickly see as an architectural possibility during exploration.

13.   Experimental Validation

Single-instruction tests   For the validation of the sequential ISA semantics we wrote a tool for automatically generating ARM assembly tests that compare hardware and model behaviour for individual instructions. Each of the tests first initialises registers and memory to particular values, then logs the relevant CPU and memory state before and after running the instruction that is tested.

The tests are generated largely automatically based on information derived from the instruction descriptions in the ARM ARM. The manual describes the encoding of the instructions and instruction fields using tables of the legal bit patterns; the instruction's pseudo code explains how the instruction's parameters are used by the instruction: some instruction fields encode immediate values, others mode strings or bits that switch between different variants of the same instruction. Based on this the tool generates tests by selecting random immediate values and all legal combinations of mode strings and bits to test, as much as possible, all behaviours of the instruction.

The test programs are statically linked Linux ELF binaries produced by GCC, that can be executed using our tool.

The tool generates around 8400 tests, all of which pass. The tests are generated uniformly for almost all instructions; branches and loads/stores need some additional setup.

Litmus tests   For experimental validation of the concurrency semantics, we use a library of litmus tests developed in previous work on Power and ARM [2, 4, 22, 23], including both hand-written tests and tests autogenerated by the diy tool of Alglave and Maranget [1], adapted to ARMv8. We use the the litmus tool [3] to make an ARM executable that is then run on different ARM devices. The executable runs millions (sometimes billions) of iterations of the test, trying to excite the cores to produce interesting behaviour, the outcome of which is a list of observed final states.

For each litmus test we then compare the final states observed on hardware with the final states reported by the exhaustive exploration of the model. In addition, we compare the results of exhaustive exploration between Flowing and POP. The exhaustive exploration of Flowing gives results for 2489 litmus tests and of POP for 2530, out of 4832; the remainder exceed time or memory limits.

The experimental comparison between Flowing and POP shows no difference between the models. One can devise exotic litmus tests on which the models will behave differently, exploiting the Flowing observable topology (e.g. 4xIRIW+addrs, combining multiple instances of the IRIW+addrs litmus test), but these are too big for our tool's exhaustive enumeration. (we checked by hand that that test is allowed by POP and proved it is forbidden by Flowing).

The comparison with hardware shows all the observable behaviour on hardware is allowed by the models (i.e. models are sound). As expected, some behaviours that are allowed by the models are not observed on current hardware.

In online material we give two tables (for ARMv7 and ARMv8) with a small sample of our experimental results, for the litmus tests cited by name in this paper (including results for Snapdragon 810, Denver, and A8X processors). The models are sound with respect to this data; indeed, the tested hardware is often less relaxed than the models and architectural intent. The data also illustrates how much hardware behaviour varies from one implementation to another, further emphasising the importance of a precise understanding of the architectural envelope.

14.   Conclusion

Well-validated semantic models for mainstream processor architectures are an important and technically challenging problem in themselves, and they are also an essential prerequisite for higher-level semantics and verification, of fine-grained concurrent algorithms, operating systems code, compilers, high-level-language concurrency models, and much more.

In this paper we have taken important steps towards this for the ARMv8 architecture, combining concurrency models, both microarchitectural and more abstract, with a complete application-level non-FP/SIMD ISA semantics. The former are validated by discussion with ARM and by black-box litmus testing; the latter by the close correspondence with the ARM ARM and by single-instruction testing. Our models will be made available for use by others, in their Sail and Lem source and as command-line and web-interface executable tools. We have also (in an initial experiment) used Lem to generate Isabelle/HOL definitions of all the model except the Sail interpreter, to support mechanised reasoning.

Much future work is possible, of course. In the short term, more validation is always desirable, here especially for the exclusive operations and for mixed-size accesses (we have not touched on the latter in this paper; our model covers them but they have not been well-tested). For work on concurrent algorithms and verified compilation, our coverage should be sufficient, but for some OS code one would also need semantics for exceptions, interrupts, and virtual memory, including all their interactions with concurrency.

References

[1] J. Alglave and L. Maranget. The diy tool. http://diy.inria.fr/.

[2] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proc. CAV*, 2010.

[3] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: running tests against hardware. In *Proceedings of TACAS 2011: the 17th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–44, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19834-2. URL http://dl.acm.org/citation.cfm?id=1987389.1987395.

[4] J. Alglave, L. Maranget, and M. Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM TOPLAS*, 36(2):7:1–7:74, July 2014. ISSN 0164-0925. .

[5] R. M. Amadio, N. Ayache, F. Bobot, J. Boender, B. Campbell, I. Garnier, A. Madet, J. McKinna, D. P. Mulligan, M. Piccolo, R. Pollack, Y. Régis-Gianas, C. S. Coen, I. Stark, and P. Tranquilli. Certified complexity (cerco). In *Foundational and Practical Aspects of Resource Analysis - Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers*, pages 1–18, 2013. . URL http://dx.doi.org/10.1007/978-3-319-12466-7_1.

[6] ARM. Cortex-a57 processor. www.arm.com/products/processors/cortex-a/cortex-a57-processor.php, Accessed 2015/07/06.

[7] *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. ARM Ltd., 2014.

[8] W. W. Collier. *Reasoning about parallel architectures*. Prentice Hall, Englewood Cliffs, 1992. ISBN 0-13-767187-3. URL http://opac.inria.fr/record=b1105256.

[9] J. a. Dias and N. Ramsey. Automatically generating instruction selectors using declarative machine descriptions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 403–416, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. . URL http://doi.acm.org/10.1145/1706299.1706346.

[10] A. C. J. Fox. Directions in ISA specification. In *Interactive Theorem Proving – Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, pages 338–344, 2012. . URL http://dx.doi.org/10.1007/978-3-642-32347-8_23.

[11] S. Goel, W. A. Hunt, M. Kaufmann, and S. Ghosh. Simulation and formal verification of x86 machine-code programs that make system calls. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 18:91–18:98, Austin, TX, 2014. FMCAD Inc. ISBN 978-0-9835678-4-4. URL http://dl.acm.org/citation.cfm?id=2682923.2682944.

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

[13] A. Kennedy, N. Benton, J. B. Jensen, and P.-E. Dagand. Coq: The world's best macro assembler? In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, pages 13–24, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2154-9. . URL http://doi.acm.org/10.1145/2505879.2505897.

[14] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: A verified implementation of ml. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. . URL http://doi.acm.org/10.1145/2535838.2535841.

[15] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009. URL http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf.

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

[17] P. McKenney. Validating memory barriers and atomic instructions. Linux Weekly News article, Dec. 2011. https://lwn.net/Articles/470681/.

[18] G. Morrisett, G. Tan, J. Tassarotti, J. Tristan, and E. Gan. Rocksalt: better, faster, stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 395–404, 2012. . URL http://doi.acm.org/10.1145/2254064.2254111.

[19] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. In *Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 175–188, 2014. . URL http://doi.acm.org/10.1145/2628136.2628143.

[20] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proceedings of TPHOLs 2009: Theorem Proving in Higher Order Logics, LNCS 5674*, pages 391–407, 2009.

[21] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. 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*, pages 379–391, Jan. 2009. . URL http://doi.acm.org/10.1145/1594834.1480929.

[22] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proceedings of PLDI 2011: the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 175–186, 2011. . URL http://doi.acm.org/10.1145/1993498.1993520.

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

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

[25] X. Shi, J.-F. Monin, F. Tuong, and F. Blanqui. First steps towards the certification of an ARM simulator using Compcert. In J.-P. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs*, volume 7086 of *Lecture Notes in Computer Science*, pages 346–361. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-25378-2. . URL http://dx.doi.org/10.1007/978-3-642-25379-9_25.

[26] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, June 2013. ISSN 0004-5411. . URL http://doi.acm.org/10.1145/2487241.2487248.