An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors

Kathryn E. Gray[1] Gabriel Kerneis[1*] Dominic Mulligan[1] Christopher Pulte[1] Susmit Sarkar[2] Peter Sewell[1]

[1]University of Cambridge *(during this work) [2]University of St Andrews

ABSTRACT

Weakly consistent multiprocessors such as ARM and IBM POWER have been with us for decades, but their subtle programmer-visible concurrency behaviour remains challenging, both to implement and to use; the traditional architecture documentation, with its mix of prose and pseudocode, leaves much unclear.

In this paper we show how a precise architectural envelope model for such architectures can be defined, taking IBM POWER as our example. Our model specifies, for an arbitrary test program, the set of all its allowable executions, not just those of some particular implementation. The model integrates an operational concurrency model with an ISA model for the fixed-point non-vector user-mode instruction set (largely automatically derived from the vendor pseudocode, and expressed in a new ISA description language). The key question is the interface between these two: allowing all the required concurrency behaviour, without overcommitting to some particular microarchitectural implementation, requires a novel abstract structure.

Our model is expressed in a mathematically rigorous language that can be automatically translated to an executable test-oracle tool; this lets one either interactively explore or exhaustively compute the set of all allowed behaviours of intricate test cases, to provide a reference for hardware and software development.

1. INTRODUCTION

1.1 Problem

Architecture definitions provide an essential interface, decoupling hardware and software development, but they are typically expressed only in prose and pseudocode documentation, inevitably ambiguous and without a tight connection to testing or verification; they do not exist as precise artifacts. This is especially problematic for the concurrency behaviour of weakly consistent multiprocessors such as ARM and IBM POWER, where programmer-visible microarchitectural optimisations expose many subtleties: the traditional documentation does not define precisely which programmer-observable behaviour is (and is not) allowed for concurrent code; the definitions are not executable as test oracles for pre-silicon or post-silicon hardware testing; they are not executable as an emulator for software testing; and they are not mathematically rigorous enough to serve as a foundation for software verification. Instead, one has an awkward combination of those PDF documents, vendor-internal "golden" simulation models (sometimes in the form of large C/C++ codebases), manually curated test-cases, and software emulators such as gem5 [1] and QEMU [2].

We argue instead that what is needed, for any architecture, is a definitive architectural envelope specification, precisely defining the range of allowed behaviour for arbitrary code. Such a specification should have many desirable properties: it should be mathematically rigorous, readable, clearly structured, sound with respect to the vendor intent, sound with respect to existing implementations (allowing all experimentally observable behaviour, modulo errata), avoid over-commitment to particular microarchitectural implementation choices, have a clear computational intuition, and be executable as a test oracle, to enumerate, check or explore the allowed behaviour of test cases, and hence to serve as a reference.

We show here how this can be achieved for a substantial fragment of the IBM POWER architecture.

1.2 Context

We build on previous work by Sarkar et al. [3, 4], who describe an architectural model for IBM POWER concurrency. That model is expressed in an *abstract microarchitectural* style, to give a clear computational intuition without committing to particular microarchitectural choices. There is a storage subsystem model that maintains a state of all the memory writes and barriers seen, the coherence relations among them that have been established so far (as a strict partial order), and the list of writes and barriers propagated to each thread; this abstracts from the microarchitectural de-

MICRO-48, December 05-09, 2015 Waikiki, HI, USA
Copyright is held by the authors. Publication rights licensed to ACM.
ACM 978-1-4503-4034-2/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2830772.2830775 .

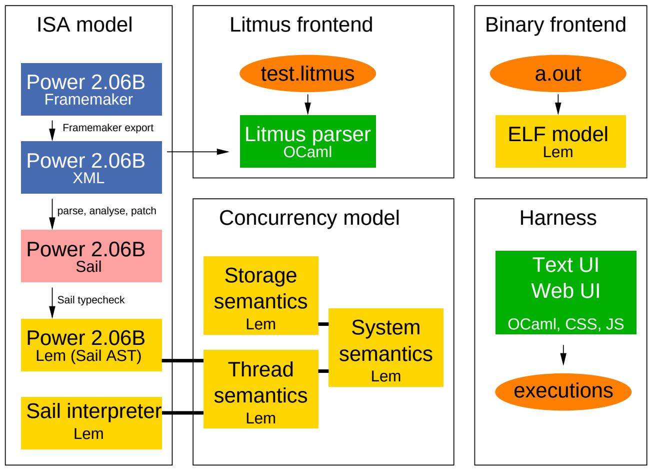


Figure 1: Overview

tails of any particular cache protocol and storage hierarchy. Then there is a model for each hardware thread that maintains a tree of in-flight and committed instruction instances, expressing the programmer-visible aspects of out-of-order and speculative computation; this abstracts from pipeline and local store queue microarchitecture. Together these form an abstract machine with a state and transitions.

That model has been experimentally validated against several generations of POWER implementations (G5, 5, 6, 7, and 8), comparing the model behaviour with that of production or pre-silicon hardware, on hand-written litmus tests and on tests produced by the diy tool of Alglave and Maranget [5], using the Litmus test harness [6]. It has been validated intensionally by extensive discussion with a senior IBM architect (clarifying the intended concurrency model in the process); it has been validated mathematically by using it in a proof that C/C++11 concurrency [7] can be correctly compiled to POWER [8, 4]; and the tool has been used by Linux-kernel software developers [9]. This work (together with related research on axiomatic models [10]) has also discovered errata in a number of multiprocessor implementations of POWER and ARM architectures, both pre- and post-silicon (ARM concurrency is broadly similar to POWER, though not identical).

However, that previous model makes many major simplifying assumptions. In particular, it includes only a tiny fragment of the POWER instruction set, and even that is given only an ad hoc semantics and only at an assembly level; and it does not handle mixed-size memory and register accesses. Effectively, it *only* defines the architectural behaviour for simple litmus-test programs, not of more general code.

1.3 Contribution

In this paper we show how a precise architectural envelope model for a weakly consistent architecture can be defined, integrating a concurrency model (extending that of [3]) with an architectural model for all of the fixed-point non-vector user-mode instruction set. Doing this in a way that achieves all the desirable properties mentioned above requires several new contributions, which we summarise here and detail below.

Concurrency/ISA Model Interface The most fundamental question we address is what the interface between the concurrency model and ISA semantics should be (§2). For a single-threaded processor one can regard instructions simply as updating a global register and memory state. The same holds for a sequentially consistent (SC) multiprocessor, and TSO multiprocessor behaviour (as in x86 and Sparc) requires only the addition of per-thread store buffers. But for weakly consistent multiprocessors such as IBM POWER and ARM, some aspects of out-of-order and speculative execution, and of the non-multi-copy atomic storage subsystem, are exposed to the programmer; we cannot use a simple state-update model for instructions. We explain this, and discuss what is required instead, with a series of concurrent POWER examples. There has been a great deal of work on modelling weakly consistent processors, e.g. [11, 12, 13, 14, 15, 16, 17, 18, 19], but none that we are aware of deals with these issues with the interface to the instruction semantics.

Sail ISA Description Language Architecture descriptions of instruction behaviour are traditionally expressed in a combination of prose and of pseudocode that looks like it is written in a sequential imperative language (of abstract micro-operations such as assignments to registers, arithmetic operations, etc.). They are also relatively large, with hundreds-to-thousands of pages of instruction description, and it is important that they be accessible to practising engineers. To achieve this, permitting instruction descriptions to be expressed in that familiar imperative style while simultaneously supporting the structure we need for integration with the concurrency model, we introduce a new instruction description language (IDL), Sail (§3). Sail has a precisely defined and expressive type system, using type inference to check pseudocode consistency while keeping instruction descriptions readable. It currently supports a concrete syntax similar to the POWER pseudocode language (front-ends tuned to other conventions are also possible in future).

ISA Description Tied to Vendor Documentation The vendor specification for the POWER architecture is provided as a PDF document [20] produced from Framemaker sources. To keep our ISA model closely tied to that description, we took an XML version exported by Framemaker and wrote a tool that extracts and analyses the instruction descriptions from it, producing Sail definitions of the decoding and instruction behaviour, and auxiliary code to parse and pretty-print instructions (§4). The vendor pseudocode is less consistent than one might hope (unsurprisingly, as it has not previously been mechanically parsed or type-checked), so producing a precise Sail definition required dealing with ad hoc variations and patching the results.

Extended Concurrency Model Scaling up the previous concurrency model of Sarkar et al. to this larger

ISA also revealed previously unconsidered architectural questions, of just what behaviour should be allowed in various concurrent cases, which we investigated in discussion with architects and with ad hoc testing (§2, §5).

Test-Oracle Tool To produce a useful tool from our model, building on the previous `ppcmem` tool [3], we use Lem to automatically generate executable OCaml code from the mathematical model. We combine this with a front-end for litmus tests, using boilerplate code generated from the XML, and with a front-end based on a formal model of the ELF executable format, to be described elsewhere. That gives us a tool, with command-line and web interfaces[1], for interactively or exhaustively exploring the architecturally allowed behaviour of small (but possibly highly intricate) concurrent test cases (§6).

Validation We validate that our model is a sound description of POWER hardware, i.e. that the envelope of behaviour it defines includes the experimentally observable behaviour of implementations, by comparing model and implementation for a range of sequential and concurrent tests (§7). Our discussion with the vendors also provides some assurance that the model captures the architectural intent on various points. Experimental testing can never provide complete assurance, of course, and we intend to maintain and refine the model.

An overview of our system is in Fig.1, showing the formally specified components (in Lem and Sail), the vendor descriptions we start from for the ISA model (in Framemaker and the derived XML), and the surrounding parsing and harness code (in OCaml). The key internal interface is that between the ISA and Concurrency models.

1.4 Limitations

This is, to the best of our knowledge, the first mathematically rigorous architectural model of weakly-consistent multiprocessor behaviour that is integrated with a substantial instruction-set model. But many important limitations remain: it handles only non-write-through cacheable coherent memory, and we do not consider exceptions and interrupts, floating point, instruction-cache effects, or supervisor features (including page table manipulation). Given those limitations, our tool provides an emulator for concurrent programs, but our focus is on making it architecturally complete, not on performance (or on performance modelling). We compile our mathematical definitions to executable code in a deliberately straightforward fashion, without optimisation, to maintain confidence that we are executing the definition, and finding all executions of concurrent programs is combinatorially challenging. The tool should therefore be seen as a reference for small-but-intricate test programs (for hardware testing, and as found in implementations of OS synchronisation primitives and concurrent data structures), not as an emulator for production-scale code.

[1]`http://www.cl.cam.ac.uk/~pes20/ppcmem2`

1.5 Background: rigorous Lem specification

To express our model in a way which is both mathematically precise and executable, we use the Lem tool [21]. Lem provides a lightweight language of definitions, of types and pure functions, that can be seen both as mathematical definitions and as executable pure functional programs; the tool typechecks them and (subject to various constraints) can export to executable OCaml code, proof assistant definitions for Coq, HOL4, and Isabelle/HOL, and typeset LaTeX. The Lem type system and library include unbounded and bounded integers, tuples `t1*..*tn`, records `<|l1:t1,..,l2:tn|>`, function types `t1->t2`, sets `set t`, maps `map t1 t2`, and user-defined types, with top-level ML-style polymorphism. The expression language combines normal pure functional-programming features (functions, pattern-matching, etc.) together with simple higher-order logic, including bounded quantification and set comprehensions.

2. THE ISA/CONCURRENCY INTERFACE

Modern multiprocessor implementations embody many sophisticated microarchitectural optimisations, and weakly consistent multiprocessor architectures make a choice to let some consequences of those be observable to programmers; they trade off what is arguably a more complex programming model for benefits in speed, power, simplicity, or verifiability. To make a precise architectural envelope model we have to identify and specify exactly what the intended range of allowed behaviour is, and these observable weakly consistent phenomena impact the structure of the model in interesting ways. At one extreme, we cannot simply model instructions as atomically updating a global register and memory state, in the way that a sequential or sequentially consistent concurrent emulator might do, as that would not be sound with respect to actual implementations (it would not admit all their observable behaviour). But our architectural model should also involve as little implementation detail as possible, both for simplicity and to be independent of any particular implementation. The details of some particular pipeline or storage hierarchy (as needed for correctness or performance evaluation of an implementation) would not make a a good definition of an architecture.

In this section we show how various weakly consistent phenomena impact the interface between the modelling of instructions and that of the concurrency behaviour, describing how we can permit the right envelope of behaviour while remaining as abstract as possible.

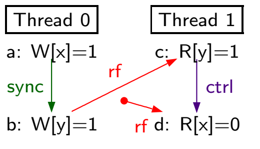
2.1 Constraints from observable behaviour

2.1.1 No single program point

We recall aspects of experimentally observed behaviour that inform our design choices, reusing notation and test naming conventions from previous work [3, 4]. First, we have observable out-of-order and speculative execution, as in the MP+sync+ctrl litmus test below. The execution of interest is on the right, with reads-from (rf), sync, and control-dependency edges between memory events; the POWER assembly, with initial and final state that identifies that execution, is on the left for reference. Here Thread 0 writes some data to x and then sets a flag y, with a strong sync barrier between to keep those two in order as far as any other thread is concerned. Thread 1 reads the flag and then, after a conditional branch, the data. It is architecturally allowed and observable in practice for the load of x to be satisfied speculatively (reading 0 from the initial state), before the conditional branch is resolved by a load of y=1 from Thread 0's write.

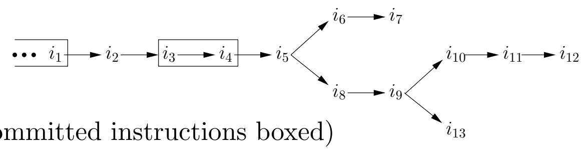
MP+sync+ctrl POWER

Thread 0	Thread 1
stw r7,0(r1) # x=1	lwz r5,0(r2) # r5=y
sync # sync	cmpw r5,r7 # if r5=1
stw r8,0(r2) # y=1	beq L
	...
	L:
	lwz r4,0(r1) # r4=x
Initial state: 0:r1=x, 0:r2=y, 0:r7=1, 0:r8=1, 1:r1=x, 1:r2=y, 1:r7=1, x=0	
Allowed: 1:r5=1, 1:r4=0	



Test MP+sync+ctrl: Allowed

This means that an operational model that permits all architecturally allowed behaviour cannot simply execute the instructions of each hardware thread one-by-one in program order. Instead, as in Sarkar et al. [3], we maintain a tree of in-flight instruction instances for each hardware thread, branching at conditional branch or calculated jump points, and discarding un-taken subtrees when branches become committed. For example:



(committed instructions boxed)

2.1.2 No per-thread register state

In a sequential, SC, or TSO model, one can treat registers simply as a per-thread map from architected register names to values, examined and updated on register reads and writes. But in a pipelined implementation many instructions can be in flight, and concurrency can make microarchitectural shadow registers and register renaming indirectly programmer-observable, as in the MP+sync+rs example below (a message-passing variant of Adir et al. [17, Test 6]). Here the three uses of r5 on Thread 1 do not prevent the second read being satisfied out-of-order (e.g. the first two uses of r5 on Thread 1 might involve one shadow register while the third usage might involve another). This is observable on some ARM and POWER processors.

MP+sync+rs POWER

Thread 0	Thread 1
stw r7,0(r1) # x=1	lwz r5,0(r2) # r5=y
sync # sync	mr r6,r5 # r6=r5
stw r8,0(r2) # y=1	lwz r5,0(r1) # r5=x
Initial state: 0:r1=x, 0:r2=y, 0:r7=1, 0:r8=1, 1:r1=x, 1:r2=y, x=0	
Allowed: 1:r6=1, 1:r5=0	

This means that a sound model cannot simply have a per-thread register state. Instead, when an instruction needs to do a register read, we have to walk back through its program-order (po) predecessors in the tree to find the most recent one that has written to that register, and take the value from there. We also have to check that there are no po-intervening instructions that *might* write to that register (and block until otherwise if so), which means that we have to be able to pre-calculate their register-write footprints. Two POWER instructions (lswx and stswx) have data-dependent register footprints; for these we pre-calculate an upper bound that we refine after the XER register read that determines them.

2.1.3 Register self-reads

Many instructions have pseudocode that reads from one of their own register writes. To simplify the definition of when register reads should block, we rewrite them to use a local variable instead. This gives the useful property that for most instructions the register-read and register-write footprints can be calculated statically from its opcode fields, and that it will dynamically read and write exactly once to each element of those.

2.1.4 Dependencies and register granularity

In POWER, dependencies between instructions arising from register-to-register dataflow are architecturally significant (shadow registers notwithstanding), as they guarantee local ordering properties that concurrent contexts can observe, and that are used in idiomatic code. Address and data dependencies create local ordering between memory reads and writes, while control dependencies do so only in some circumstances (allowing implementations to speculate branches but not to observably speculate values). Processor architectures typically have a more-or-less elaborate structure of register names and aliases. For example, POWER includes 32 64-bit general-purpose registers, GPR[0]..GPR[31] (denoted *ri* in assembly) and a 32-bit condition register CR (with bits indexed 32..63) that is partitioned into 4-bit subfields CR0..CR7; those bits are also referred to with individual flag names LT, EQ, etc. All this must be supported in the pseudocode, but the more important semantic question is the architectural granularity of mixed-size register accesses: a precise model has to define when writing to one part of a register and reading from another constitutes a dependency. The choice is observable in tests such as MP+sync+addr-cr below: Thread 0 is just the same two message-passing writes, while Thread 1 has a putative dependency chain involving a write to CR3 followed by a read form CR4. The final state of 1:r6=1, 1:r5=0 is observable in implementations, telling us that a sound architectural model cannot treat CR as a single unit.

MP+sync+addr-cr POWER

Thread 0	Thread 1
stw r7,0(r1) # x=1	lwz r5,0(r2) # r5 = y
sync # sync	mtocrf cr3,r5 # cr3 = 4 bits of r5
stw r8,0(r2) # y=1	mfocrf r6,cr4 # set 4 bits of r6 = cr4
	xor r7,r6,r6 # r7 = r6 xor r6
	lwzx r8,r1,r7 # r8 = *(&x + r7)
Initial state: 0:r1=x, 0:r2=y, 0:r7=1, 0:r8=1, 1:r1=x, 1:r2=y, x=0	
Allowed: 1:r6=1, 1:r5=0	

Related experiments show that dependencies through individual bits of `CR` are respected, as are dependencies through adjacent bits of the same `CR`*n* field, so we could treat `CR` either as a collection of 4-bit fields or as 32 1-bit fields. The vendor documents are silent on this question, but the latter seems preferable: it allows the most hardware implementation variation; we do not believe that code in the wild relies on false-register-sharing dependencies (though of course it is hard to be sure of this); and it is mathematically simplest. We follow this choice for all register accesses, with a general definition that assembles the value for a register read by reassembling fragments of the most recent writes (in POWER, the GPR registers are always fully written).

It is also important that interactions via the current instruction address (CIA) do not give rise to dependencies in the model, as that would prevent out-of-order execution. The POWER instruction descriptions read and write two pseudoregisters, `CIA` and `NIA`, which are not architected registers; our thread model treats those specially.

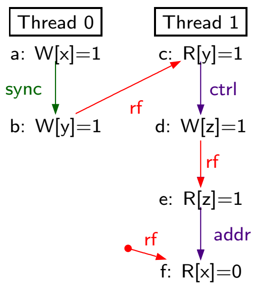
2.1.5 *Reading from uncommitted instructions*

For out-of-order execution to work, we clearly have to let instructions read from register writes of po-previous instructions that are not yet finished, just as in implementations register values might be forwarded from instructions before they are retired.

But we also have to let po-later instructions read from the *memory* writes of earlier instructions in speculative paths. The POWER architecture states that writes are not performed speculatively but, while speculative writes are never visible to other threads, they can be forwarded locally to program-order-later reads on the same thread. The PPOCA variant of MP below shows that this forwarding is observable to the programmer. Here f is address-dependent on e, which reads from the write d, which is control-dependent on c. One might expect that chain to prevent read f binding its value before c does, but in fact the write d can be forwarded directly to e within the thread while d, e, and f are all still speculative (before the branch of the control dependency on c is resolved). This is intended to be architecturally allowed and it is observable in practice; it means that the model must expose uncommitted writes to po-later instructions.

PPOCA POWER

Thread 0	Thread 1
stw r7,0(r1) # x=1	lwz r5,0(r2) # r5=y
sync # sync	cmpw r5,r7 # if r5=1
stw r8,0(r2) # y=1	beq L
	...
	L:
	stw r7,0(r3) # z=1
	lwz r6,0(r3) # r6=z
	xor r6,r6,r6 # r6=r6 xor r6
	lwz r4,r6,r1 # r4=*(&x+r6)
Initial state: 0:r1=x, 0:r2=y, 0:r7=1, 0:r8=1, 1:r1=x, 1:r2=y, 1:r3=z, 1:r7=1, x=0, z=0	
Allowed: 1:r5=1, 1:r4=0	



Test PPOCA: Allowed

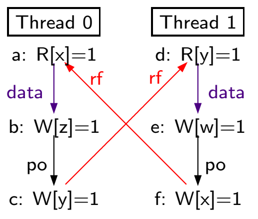
2.1.6 *Non-atomic intra-instruction semantics for register reads*

The previous examples showed that we need to be able to dynamically analyse the register and memory writes of an instruction that is partially executed but not yet committed, but so far one might imagine that one could model each instruction atomically moving between three states: not started, fully executed but not committed, and committed. The following example shows that this is not the case, even for instructions that do just one memory write: we have to be able to see that the memory footprint of an instruction becomes determined after only some of its register reads have been satisfied, in order to know that po-later instructions will definitely be to different addresses and hence can be executed out-of-order (before all the register reads of the first instruction are resolved) without violating coherence.

Without the middle writes of the example (b and e), one has a plain 'LB' test, which is intended to be architecturally allowed for POWER and ARM. The LB+datas+WW variant has extra writes, to two different addresses, inserted in the middle of each thread. These middle writes are merely data-dependent on the first reads, not address-dependent, so even before the reads have been satisfied, the middle writes can be known to be to different addresses to the last writes on each thread. In some implementations this lets those writes go ahead out-of-order and be read from by the reads on the other thread. If instead those middle writes were address-dependent on the first reads (as in a test LB+addrs+WW, not shown), then before those reads are satisfied the middle writes would not be known to be to different addresses to the last writes on each thread, and the last writes could not go ahead. The former is observable on some ARM processors; the latter is not. For the current POWER server processors, no LB variant is observable.

LB+datas+WW POWER

Thread 0	Thread 1
lwz r5,0(r1) # r5=x	lwz r6,0(r2) # r6=y
stw r5,0(r3) # z=r5	stw r6,0(r4) # w=r6
stw r9,0(r2) # y=1	stw r9,0(r1) # x=1
Initial state: x=0, y=0, 0:r1=x, 0:r2=y, 0:r3=z, 0:r9=1, 1:r1=x, 1:r2=y, 1:r4=w, 1:r9=1	
Allowed: 0:r5=1, 1:r6=1	



Test LB+datas+WW: Allowed

A write instruction typically has a register read that supplies the data to be written and one or more register reads that are used to compute the address to be written, e.g. as in pseudocode below for the `stw RS,D(RA)` instruction used for those middle writes. This calculates an effective address `EA` from register `RA` and instruction field `D` before reading the data from register `RS` and doing the memory write.

```
(bit[64]) b := 0;
(bit[64]) EA := 0;
if RA == 0 then b := 0 else b := GPR[RA];
EA := b + EXTS (D);
MEMw(EA,4) := (GPR[RS])[32 .. 63]
```

To permit LB+datas+WW, after the address register reads can be resolved (i.e., after the program-order-previous instructions that write those registers have produced values for them, whether or not they have been committed), we have to be able to compute the write address, even if the data register reads cannot yet be resolved, so that later instructions that might have been to the same address can go ahead. In contrast, whether the data register reads can be resolved has no effect on later instructions. Hence:

1. It would be unsound to block the middle writes until their entire register-read footprint is available, as that would block the later writes.
2. It would be sound to interpret the pseudocode in a dataflow style, or to rewrite it with explicit intra-instruction concurrency, but that would introduce unnecessary nondeterminism.
3. It would be sound to interpret the pseudocode as written sequentially, as the address reads are before the data reads (but it would not be if they were reversed).

We adopt the last alternative, but note that this means we have to be able to dynamically recalculate the potential memory read and write footprint of an instruction in progress, after some but not all of its register reads are resolved.

2.1.7 *Undefined values*

Instruction descriptions often leave some register bits explicitly undefined, e.g. for some flag bits, or 32 bits of the result register in POWER multiply-word instructions. There are several possible interpretations of this. One could: (a) make a nondeterministic choice at assignment time (if stable, or when read, if not); (b) feed in the concrete values from an observed hardware or simulator trace; (c) work over lifted bits, `0`, `1`, or `undef`; or (d) work with symbolic bit values and accumulate constraints on them. Option (a) is mathematically attractive and suitable for testing software above the model, but would make it combinatorially infeasible to find all allowed behaviours even for small examples, and likewise infeasible to compare our emulator results against actual hardware. Option (b) could be attractive for conformance testing vs. an existing simulator. Option (d) is also combinatorially challenging, especially if any such bits are used in addresses. We currently support (c), allowing undefined bits in register and memory values (to support testing against actual hardware) but not in address or instruction-field values (as that would make semantic exploration infeasible). This is reflected in the interface to the ISA semantics.

2.2 The ISA semantics interface

From the above, we can see that there is (so far) no need for intra-instruction parallelism in the model. We have to be able to execute multiple instructions from a thread concurrently, and to represent partially executed instructions (instructions cannot be regarded as atomic transactions, though they sometimes do need to be aborted or restarted). But their individual pseudocode can be interpreted sequentially, taking care with the sequencing of register reads leading to addresses vs those leading to data. The interface to an executing instruction has to expose its register and memory read and write events, together with memory barrier events. Other instruction instances in a thread have to be able to make progress while one is blocked waiting for a register read or memory read to be satisfied. It has to be possible to forward from a po-previous memory write in the same thread that has not yet been committed.

We also saw that it is necessary to be able to dynamically calculate the possible memory read and write footprint of a partially executed instruction instance, analyzing its future behaviour, and we have to know which pending register reads can affect those footprints.

This can all be achieved most simply, and with a tight connection to a readable pseudocode description of instructions, by a deep embedding of a formally defined pseudocode language (or instruction description language, IDL) into the generic Lem mathematical meta-language in which our model is written; such an embedding represents the typed instruction description as a Lem term of a Lem type for the IDL AST. We describe our Sail IDL in the next section. It is given semantics with a Sail interpreter, defined in Lem, and that interpreter has a simple interface to the rest of the model, with types as below (slightly condensed for presentation).

```
type instruction_state

type outcome =
| Read_mem of address*size*(memval -> instruction_state)
| Write_mem of address*size*memval*instruction_state
| Barrier of barrier_kind*instruction_state
| Read_reg of reg_slice*(regval -> instruction_state)
| Write_reg of reg_slice*regval*instruction_state
| Internal of instruction_state
| Done

val interp : instruction_state -> outcome

val decode : context
  -> opcode -> instruction_or_decode_error

val initial_state : context
  -> instruction -> instruction_state_or_error
```

The interpreter works over a type `instruction_state` which is abstract as far as the rest of the model is concerned. The main `interp` function takes a single instruction state and executes it for one step, to produce an `outcome`. This is a labelled union type in which the memory and register read cases include an instruction-state continuation, abstracted on the value that should be supplied by the rest of the model. This lets us decouple the behaviour of a single instruction from the rest of the system (other instructions can execute while that continuation is saved).

There are also functions to decode an opcode to an instruction (an element of an instruction-set abstract-syntax type, and to create an initial instruction state

Store Doubleword with Update   DS-form

stdu   RS,DS(RA)

62	RS	RA	DS	1
0	6	11	16	30 31

```
EA ← (RA) + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

```
union ast member (bit[5],bit[5],bit[14]) Stdu

function clause decode
    ( 0b111110
    : (bit[5]) RS
    : (bit[5]) RA
    : (bit[14]) DS
    : 0b01
    as instr ) =
  Stdu (RS,RA,DS)

function clause execute (Stdu (RS, RA, DS)) =
  { EA := GPR[RA] + EXTS (DS : 0b00);
    MEMw(EA,8) := GPR[RS];
    GPR[RA] := EA }

function clause invalid (Stdu (RS, RA, DS)) =
  (RA == 0)
```

Figure 2: Example instruction description, in vendor documentation and in Sail, showing the close correspondence between the two for execute and decode

from that; these are parameterised by a context which is essentially the complete ISA definition.

To calculate the potential register and memory footprints of an instruction (from either its initial state or a partially executed state) we can simply run the interpreter exhaustively, feeding in a distinguished `unknown` value to the continuations for any reads; the interpreter operations treat `unknown` similarly to `undef`. It can also calculate the register reads that feed into memory addresses by doing this with dynamic taint tracking.

3. SAIL: ISA DEFINITION LANGUAGE

To express the mass of individual instruction descriptions, we need an instruction definition language that (1) supports the interface in the previous section, including the `interp`, `decode`, and `initial_state` functions and the analysis of the potential footprints and dependencies of partially executed instructions, (2) is mathematically precise, and (3) is readable by engineers familiar with the existing vendor documentation. There has been a great deal of previous work using domain-specific IDLs and proof assistants to describe instruction behaviour: for emulation, generation of compiler components, test generation, formal verification of compilers and of hardware, etc. [22]. Emulators such as gem5 [1] and QEMU [2] each have their own internal descriptions of instruction behaviour. On the more formal side, notable recent examples include the work of Fox [23] for ARM in his L3 IDL, and Goel et al. [24] for x86 in ACL2. Some of these are both precise and readable (and some rather complete), but to the best of our knowledge none addresses instruction behaviour in the context of weakly consistent multiprocessors and the issues of the previous section. Accordingly, we have developed a new IDL, Sail, for the purpose. Sail comprises a language, with a formally defined type system, and a tool that parses and typechecks ISA definitions, exporting them to a type-annotated Lem AST, and an interpreter (written in Lem) that executes them.

We illustrate Sail with an example instruction description in Fig. 2 (`stdu`, one of the simplest of the several hundred instructions we consider). On the left is the vendor documentation, while on the right are Sail definitions of a clause of the abstract-syntax type `ast` of instructions (`Stdu`, with three bitvector fields), a clause of the `decode` function, pattern-matching 32-bit opcode values into that abstract-syntax type, and a clause of the `execute` function, defining the instruction behaviour, and a clause of the `invalid` predicate, identifying invalid instructions. Sail supports conventional imperative code, with access to memory (the write to `MEMw`) and registers (the reads and writes of `GPR[·]`), instruction fields from the AST value (`RS`, `RA`, `DS`), local variables (`EA`), sequencing, conditionals, and loops. The Sail interpreter produces `outcome`s for register and memory accesses, and for memory barriers, as described in the previous section.

We equip the language with an expressive type system to check consistency and detect errors in Sail definitions. Many POWER instructions (especially the vector instructions) involve elaborate manipulation of bitvectors, with computed indices and lengths, and registers are indexed from various start-index values. To check these we use a type system in which types can be dependent on simple arithmetic expressions. In particular, for any type t, start index s, length l, and direction d, Sail has a type `vector<`s,l,d,t`>` of vectors of t, and the start index and length can be computed, e.g. from instruction fields and loop indices. In general type-checking in such a system quickly becomes undecidable, with numeric constraints involving addition, multiplication, and exponentiation, but the constraints that arise in practice for our ISA specification are simple enough (e.g. $0 \leq 2^n + m$, given $n \geq 0$ and $m \geq 0$) that they can be handled by an ad hoc solver. Sail type constructors (including user-defined types) and functions can be parametric in types, in natural-number values, and in directions. In the POWER description indices increase along a bitvector, from MSB to LSB, while other architectures use the opposite convention; this direction polymorphism lets us use either style directly, without error-prone translation and sharing the same library of basic operations. Sail function types can be annotated by sets of effects, to identify whether they are pure or can have register or memory side-effects, and there is also simple effect polymorphism. To keep Sail definitions readable, we use type inference and some limited automatic coercions (between bits, bitvectors of length 1, and numbers; from bit vectors to unsigned numbers; and between constants and bit vectors), so very few type annotations are needed — none in the right-hand side of Fig. 2 except for the sizes of the instruction opcode fields in the `decode` function.

All this lets us have definitions of decoding and behaviour that are simultaneously precise and close enough to the vendor pseudocode to be readable, as Fig. 2 shows.

4. FROM VENDOR DOCUMENT TO SAIL

Given a metalanguage for instruction description, one could produce a description for an existing architecture manually, reading the existing prose+pseudocode documentation and rendering it into the IDL. This gives the flexibility to refactor the definition, e.g. with an eye to particular proofs in the work of Fox [23] and to fast symbolic evaluation in that of Goel et al. [24]. Sometimes one may have to follow this manual approach: the existing documentation varies widely in how rigorously defined and how complete the instruction descriptions are, from something reasonably precise for ARM through to something much less so for x86, and in any case for some purposes such refactoring is essential. But ISA definitions are moderately large (100s to 1000s of pages), making this an error-prone and tedious task, and the result is less tightly coupled to the original than one might like: it may not be readable by practicing engineers, and it may be hard to update to a new version of the vendor specification.

The obvious alternative is to try to automatically extract all the information one can, doing as little manual patching as possible. This has been done for a sequential ARM description by Shi et al. [25], from PDF to a model complete enough to boot a kernel and theorem-prover definitions in Coq.

For POWER, the vendor pseudocode is in between those mentioned above in rigour, looking reasonably precise at first sight, and so an automatic extraction seems feasible and worthwhile. It is maintained internally as a Framemaker document, publicly released as PDF. Neither is intended to be easily parsed, and so we worked instead from an alternative XML export provided by IBM from the Framemaker source. We wrote a tool that extracts instruction descriptions automatically from this, stores them in an intermediate format suitable for analyses, and produces a Sail model for decoding and executing instructions, as well as helper OCaml code to parse, execute and pretty-print litmus tests. Some instructions needed additional patching, e.g. for the setting of arithmetic flags, which are described in the manual in prose, not in pseudocode, and for a few errors in the pseudocode.

The tool has to deal with many irregularities in the XML to pull out the main blocks shown on the left of Fig. 2 (instruction name, form, mnemonic, binary representation, pseudocode, and list of special registers altered) and parse the pseudocode into a simple untyped grammar. The powerful type inference that Sail provides makes it simple to generate Sail code from this.

4.1 Current Status

We focus mainly on the user-mode *Branch Facility* and *Fixed-Point Facility* instructions of the POWER ISA User Instruction Set Architecture [20]. In the version we worked from, these include 154 normal user instructions; we currently extract decoding information and instruction pseudocode for all of these, and we describe our test generation and validation for them in §7 (these instruction counts refer to the underlying instructions as identified in the documentation, e.g. the four `add`, `add.`, `addo`, and `addo.` variants of `Add` are counted together as one). There are 5 system-call and trap instructions, which are not in our scope, in those chapters. We also handle memory barriers: the `sync`, `lwsync`, `eieio`, and `isync` instructions from Chapter 4 of *Book II: POWER ISA Virtual Environment Architecture*; for these the instruction semantics simply signals the corresponding event to the concurrency model.

The remaining user ISA for server implementations comprises the vector, floating-point, decimal floating-point, and vector-scalar floating-point instructions. We extract decoding information for almost all of these, and pseudocode for many of the vector instructions, but specifying floating-point operations is a major topic in itself [26, 27, 28], not in our scope here.

All this produces approximately 8500 lines of Sail, defining the Sail AST, decoding, and execution for 270 instructions, and around 17 000 lines of OCaml code to parse, pretty-print, and manipulate assembly instructions.

During development we were provided with an updated version of the XML export; it required less than two days of work to adapt the extraction process (despite various changes to XML tags), suggesting that this could be maintained over time and that in principle it could be gradually integrated into a vendor workflow.

5. THE CONCURRENCY MODEL

Looking back at the Fig. 1 overview, Sections 2, 3, and 4 have described the left-hand block: our ISA model and its interface. We now describe the concurrency model, with respect to that of our starting point [3].

The thread semantics is adapted throughout to handle mixed-size register accesses and the model of instruction behaviour from §2.2, and also to maintain an explicit tree of in-flight instructions.

For mixed-size memory accesses the thread model also decomposes misaligned and large (vector) writes into their architecturally atomic units. POWER is a non-multicopy-atomic architecture: two writes by one thread to different locations can become visible to two other threads in opposite orders. This is an observable consequence of the storage hierarchy and cache protocol microarchitecture of implementations, but our model abstracts from those details; it maintains an explicit description of the memory writes (and barriers) that have been propagated to each hardware thread, and of the coherence commitments between writes that have been established so far. The decomposition allows the atomic units of an ISA memory write to be separately propagated to other threads.

Handling mixed-size memory access requires a further change to the storage subsystem model: there are now coherence relationships between overlapping writes with distinct footprints (their address/size pairs).

For concreteness, we show the Lem type of our new storage subsystem state below. It is a record type, with fields containing various sets, relations, and functions. For example, the `coherence` field is a binary relation over `write`, a type of memory write events. This too is a record type (not shown), containing a unique id, an address and size, and a memory value (a list of bytes of lifted bits).

```
type storage_subsystem_state = <|
  threads: set thread_id;
  writes_seen: set write;
  coherence: rel write write;
  events_propagated_to: thread_id -> list event;
  unacknowledged_sync_requests: set barrier;|>
```

The storage subsystem model can take transitions between such states, to accept a new write or barrier from a thread, to send a response to a read request, to propagate a write or barrier to a new thread, to acknowledge a `sync` barrier to its originating thread when the relevant events have propagated to all threads, and to establish new coherence commitments. The details of our model lie in the preconditions and resulting states of such transitions. We cannot include them here, for lack of space (the interpreter and concurrency model are around 4300 and 2800 non-comment lines of specification), but we intend to make them available online. From the type and transitions described above, though, one can see that the model is abstracting from particular microarchitecture but can still be directly related to implementation behaviour. For example, some model coherence-commitment transitions will correspond to one write winning a race for cache-line ownership.

The corresponding type for the model of each thread is essentially a tree of instruction instances, each of which can take transitions for register writes or reads, for making a memory write locally visible, satisfying a memory read by locally forwarding from such a write, committing a memory write or barrier to the storage subsystem, issuing a memory read request, satisfying a memory read from the storage subsystem, or an internal step; the thread can also fetch a new instruction instance at any leaf of its tree. To relate to the interpreter interface we saw earlier, the abstract micro-op state of an instruction is an element of:

```
type micro_op_state =
  | MOS_plain of instruction_state
  | MOS_pending_mem_read of
      read_request * (memval -> instruction_state)
  | MOS_potential_mem_write of
      (list write) * instruction_state
```

storing the continuation provided by the interpreter in the memory-read case. An instruction instance combines this with the statically analysed footprint data, obtained by running the interpreter exhaustively, and a record of the register and memory reads and writes the instruction has performed (cleared if the instruction is restarted).

At present we treat instruction and data memory separately, not having investigated the interactions between concurrency and instruction-cache effects. Instruction fetches read values from a fixed instruction memory, decode them (if possible) using the Sail decode function, as in Fig. 2, and use the exhaustive interpreter to analyse their register footprint and potential next fetch addresses. The exhaustive interpreter is also used as necessary to re-analyse the possible future memory footprint of partially executed instructions.

The complete state of the model simply collects these components together:

```
type system_state = <|
  program_memory: address -> fetch_decode_outcome;
  initial_writes: list write;
  interp_context: Interp_interface.context;
  thread_states: map thread_id thread_state;
  storage_subsystem: storage_subsystem_state;
  idstate: id_state; model: model_params;  |>
```

with a function to enumerate all the possible transitions of a system state:

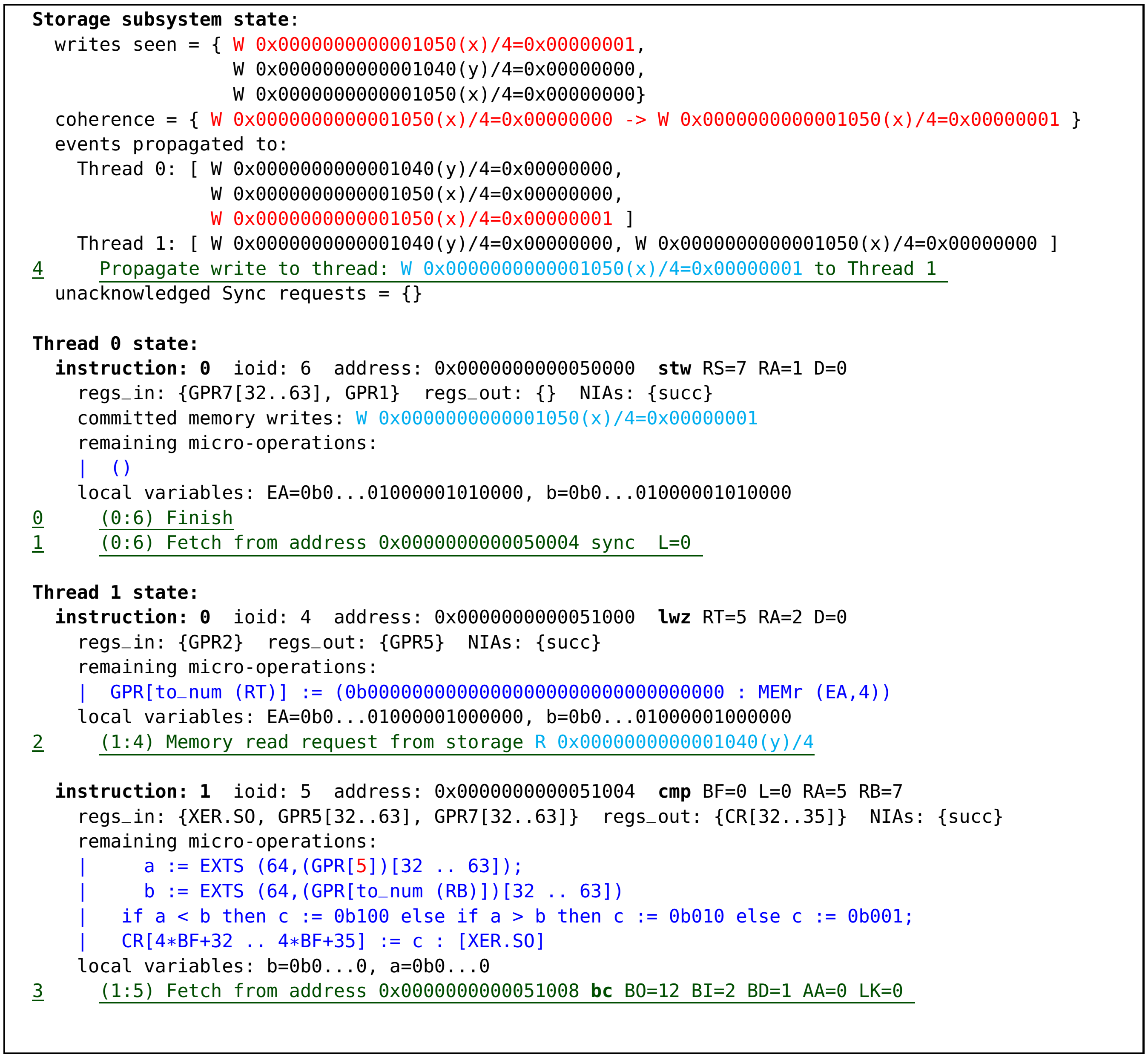
```
val enumerate_transitions_of_system :
  system_state -> list trans

val system_state_after_transition :
  system_state -> trans -> system_state_or_error
```

6. THE TOOL, WITH ELF AND LITMUS FRONT-ENDS

To make an executable tool from our mathematical model, following Fig. 1, we use the §4 extraction tool to generate a Sail definition of our POWER ISA fragment and Sail to typecheck that and translate into a Lem definition which we link with the concurrency model. Lem typechecks both and translates into executable OCaml.

We link that with an OCaml test harness for exploring the system-state transitions and with two front-ends: one to parse litmus tests such as the §2 examples, and another to parse statically linked Power64 ELF executable binaries. The former is based on the `herdtools`



```
Storage subsystem state:
  writes seen = { W 0x0000000000001050(x)/4=0x00000001,
                  W 0x0000000000001040(y)/4=0x00000000,
                  W 0x0000000000001050(x)/4=0x00000000}
  coherence = { W 0x0000000000001050(x)/4=0x00000000 -> W 0x0000000000001050(x)/4=0x00000001 }
  events propagated to:
    Thread 0: [ W 0x0000000000001040(y)/4=0x00000000,
                W 0x0000000000001050(x)/4=0x00000000,
                W 0x0000000000001050(x)/4=0x00000001 ]
    Thread 1: [ W 0x0000000000001040(y)/4=0x00000000, W 0x0000000000001050(x)/4=0x00000000 ]
4     Propagate write to thread: W 0x0000000000001050(x)/4=0x00000001 to Thread 1
  unacknowledged Sync requests = {}

Thread 0 state:
  instruction: 0  ioid: 6  address: 0x0000000000050000  stw RS=7 RA=1 D=0
    regs_in: {GPR7[32..63], GPR1}  regs_out: {}  NIAs: {succ}
    committed memory writes: W 0x0000000000001050(x)/4=0x00000001
    remaining micro-operations:
    |  ()
    local variables: EA=0b0...01000001010000, b=0b0...01000001010000
0     (0:6) Finish
1     (0:6) Fetch from address 0x0000000000050004 sync  L=0

Thread 1 state:
  instruction: 0  ioid: 4  address: 0x0000000000051000  lwz RT=5 RA=2 D=0
    regs_in: {GPR2}  regs_out: {GPR5}  NIAs: {succ}
    remaining micro-operations:
    |  GPR[to_num (RT)] := (0b00000000000000000000000000000000 : MEMr (EA,4))
    local variables: EA=0b0...01000001000000, b=0b0...01000001000000
2     (1:4) Memory read request from storage R 0x0000000000001040(y)/4

  instruction: 1  ioid: 5  address: 0x0000000000051004  cmp BF=0 L=0 RA=5 RB=7
    regs_in: {XER.SO, GPR5[32..63], GPR7[32..63]}  regs_out: {CR[32..35]}  NIAs: {succ}
    remaining micro-operations:
    |     a := EXTS (64,(GPR[5])[32 .. 63]);
    |     b := EXTS (64,(GPR[to_num (RB)])[32 .. 63])
    |   if a < b then c := 0b100 else if a > b then c := 0b010 else c := 0b001;
    |   CR[4*BF+32 .. 4*BF+35] := c : [XER.SO]
    local variables: b=0b0...0, a=0b0...0
3     (1:5) Fetch from address 0x0000000000051008 bc BO=12 BI=2 BD=1 AA=0 LK=0
```

This shows a state for the first MP+sync+ctrl example of §2.1, from the user interface of our tool (lightly edited for presentation). This state is reachable after 44 non-internal transitions, and the possible next transitions are underlined and highlighted in green; in the web interface they are clickable. The delta from the previous state is highlighted in red. Finished instruction instances are shown in a more condensed form, but there are none in this state. For each instruction, the remaining Sail abstract microoperations are shown in blue. Memory events (the writes and reads of `x` and `y` shown in the §2.1 execution diagram) are shown in cyan.
On Thread 0 the write x=1 (in full, `W 0x0000000000001050(x)/4=0x00000001`) of the first `stw` has just been committed to the storage subsystem, and in the storage subsystem state it is coherence-after an initial-state write, but it has not yet been propagated to Thread 1. The `sync` can be fetched, but it will not be propagatable to Thread 1 until that first write is.
On Thread 1 the first `lwz` and `cmp` are both executing. The former can read `y` from memory, and if that is done now (before the Thread 0 write of `y=1` has been committed and propagated to Thread 1) it will get the zero from the initial state write. The `cmp` is blocked on a register read from `GPR[5]` waiting for the `lwz` to write to it. Thread 1 continues with a `bc` conditional branch; that fetch transition is already enabled, and after that fetch the final load `lwz` could also be fetched, and indeed also speculatively satisfy its read of `x` immediately, though it could not be committed until the branch is. The `regs_in`, `regs_out`, and `NIAs` static-analysis data for the `cmp` show how its precise register footprint has been calculated, including particular subfields and bit-ranges of XER, CR, and GPR registers.

Figure 3: A tool screenshot with a system state and currently enabled transitions for MP+sync+ctrl

front-end of Maranget et al. [29], using assembly parsing and pretty-printing code produced by our extraction tool from the XML POWER definition. The latter uses a mathematical model of the ELF file format, also written in Lem. Parsed binaries are checked for static linkage and conformance with the Power64 ABI before their loadable segments are identified and loaded into the tool's code memory. Names of global variables, their addresses in the executable memory image, and their initialisation values are also extracted to initialise the tool's data memory and the user-interface symbol pretty-printer. Text and web interactive user interfaces show the current state and enabled transitions; the user can select any of those, automatically skip internal transitions, run sequentially, or (resources permitting) do an exhaustive search. Fig. 6 gives a tool screenshot, showing a model state and its enabled transitions.

7. TEST GENERATION AND VALIDATION

We validate our model in several ways. For the sequential behaviour of instructions, we generate random single-instruction tests and compare the behaviour of the model (run in sequential mode) against that of a POWER 7 server (allowing the hardware to exhibit arbitrary values where the model has `undef` bits). For concurrent behaviour, we use a range of concurrent litmus tests, running the model in exhaustive concurrent mode and checking the set of results for each test includes those previously found by testing hardware (POWER G5, 6, 7, and 8) using the `litmus` tool [6]. Experimental testing can never give complete assurance, of course, and we expect to evolve the model over time, but these results establish a reasonable level of confidence that our model is sound with respect to the behaviour of POWER hardware implementations. Assessing completeness is harder, as it is essentially a question of the architectural intent of the vendor or designer, which has not previously been expressed precisely — enabling that is the point of our work. But our discussion with IBM staff also serves to validate that our model captures the vendor architectural intent for a number of specific points. To the best of our knowledge our tool is complete, allowing all the behaviour that they intend to be allowed, except for certain exotic cases of undefined behaviour or computed branches, which would be combinatorially infeasible to enumerate.

In more detail, for sequential testing we wrote a tool to automatically generate assembly tests for each instruction supported by the model, for interesting partly-random combinations of machine state and instruction parameters, and taking care with branches and suchlike. Each test can be run either in the model or on actual hardware, logging the register values and relevant memory state before and after execution of the instruction in question, then we compare those logfiles (up to `undef`). These tests are standard ELF binaries produced with GCC (so this also exercises our ELF front-end).

Tests are generated largely automatically, from the Sail names and inferred types of instruction fields, the inferred Sail effect annotation (stating whether the instruction does register and memory reads or writes), and the vendor ISA description [20, §1.6.28] of how instruction fields are used and what registers or memory locations an instruction using this instruction field might depend on. Most of this can be done uniformly: only 13 special cases were needed for individual instructions, and 7 for certain load/store instruction forms. For single-bit mode fields, our test generation is exhaustive. For the 154 user-mode branch and fixed-point instructions, we currently generate 6984 tests (we omit only conditional branches to absolute addresses). Running these on POWER 7 hardware and in our model, all of these instructions pass all their tests. This testing has found around 33 bugs, variously in our ISA model (e.g. where we omitted flag setting that is not in the pseudocode or made multiple accesses to the same register), interpreter (e.g. arithmetic mismatches), concurrency model, and test generation. There was also one error in the manual's pseudocode and four cases where the pseudocode and text disagree.

For concurrent testing, we checked 2175 litmus tests, including those of §2 and [3], each of which identifies a non-SC execution which might or might not be allowed in the model or observed in practice. For each we ran the model exhaustively to calculate the set of possible results it allows, and compared against previous experimentally observed behaviour for POWER G5, 6, 7 and 8 hardware. This identified a small number of problems in the model, all of which were fixed. Systematic generation of good litmus tests that exercise all the new features of the model is a research problem for future work in itself, as there are now many more possibilities.

8. CONCLUSION

We have shown how one can construct a rigorous architectural model for a substantial fragment of a sophisticated multiprocessor architecture, combining instruction description and concurrency model to make a precise artifact embodying the architectural abstraction, not merely a prose document. The fact that we can compute the set of all model-allowed outcomes for extensive sequential and concurrent tests validates our claim that it can serve as a test oracle.

This opens up many possibilities for future work: using the model as a reference for hardware validation, for exploring whether future microarchitecture designs provide the same programmer-observable concurrency behaviour, and for testing whether implementations of software concurrency libraries are correct with respect to the architecture (previous sequential POWER hardware testing deemed construction of an architecture model to be a *"critical step [that] should not be underestimated"* [30]). Related work on ARM is also in progress [31].

We have focussed on precision, clarity, abstraction, soundness, and completeness, not on performance, and that is the obvious limitation of our current tool: it is just fast enough for the testing we have considered (the sequential and concurrent checking above take minutes and hours respectively, on a single machine). There is ample scope for improving both sequential and concurrent performance, though the latter will always be limited by the combinatorial challenge. Ideally one would be able to generate high performance, verified, and architecturally complete emulators from such models, but that needs a great deal of further research.

We have done all this for a pre-existing industrial architecture, but our techniques, tools, and much of the concurrency model should also be applicable to research architectures. There one can hope to escape some of the legacy issues and move directly to precise architecture descriptions that one can test against.

8.1 Acknowledgements

We acknowledge funding from EPSRC grants EP/H005633 (Leadership Fellowship, Sewell) and EP/K008528 (REMS Programme Grant), an ARM iCASE award (Pulte), and the Scottish Funding Council (SICSA Early Career Industry Fellowship, Sarkar).

9. REFERENCES

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[2] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), pp. 41–41, USENIX Association, 2005.

[3] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding POWER multiprocessors," in *PLDI*, 2011.

[4] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams, "Synchronising C/C++ and POWER," in *Proc. PLDI*, 2012.

[5] J. Alglave and L. Maranget, "The `diy` tool." `http://diy.inria.fr/`.

[6] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Litmus: running tests against hardware," in *Proc. TACAS 2011*, 2011.

[7] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing C++ concurrency," in *Proc. POPL*, 2011.

[8] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell, "Clarifying and compiling C/C++ concurrency: from C++11 to POWER," in *Proc. POPL*, 2012.

[9] P. McKenney, "Validating memory barriers and atomic instructions," *Linux Weekly News*, 2011. `http://lwn.net/Articles/470681/`.

[10] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data-mining for weak memory," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), pp. 40–40, ACM, 2014.

[11] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, (Los Alamitos, CA, USA), pp. 434–442, IEEE Computer Society Press, 1986.

[12] W. Collier, *Reasoning about parallel architectures.* Prentice-Hall, Inc., 1992.

[13] K. Gharachorloo, "Memory consistency models for shared-memory multiprocessors," *WRL Research Report*, vol. 95, no. 9, 1995.

[14] J. M. Stone and R. P. Fitzgerald, "Storage in the PowerPC," *IEEE Micro*, vol. 15, pp. 50–58, April 1995.

[15] C. May, E. Silha, R. Simpson, and H. Warren, eds., *The PowerPC architecture: a specification for a new family of RISC processors.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994.

[16] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[17] A. Adir, H. Attiya, and G. Shurek, "Information-flow models for shared memory with an application to the PowerPC architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 5, pp. 502–515, 2003.

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

[19] D. Lustig, M. Pellauer, and M. Martonosi, "Pipe check: Specifying and verifying microarchitectural enforcement of memory consistency models," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (Washington, DC, USA), pp. 635–646, IEEE Computer Society, 2014.

[20] *Power ISA Version 2.06B.* IBM, 2010. `https://www.power.org/wp-content/uploads/2012/07/PowerISA_V2.06B_V2_PUBLIC.pdf` (accessed 2015/07/22).

[21] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell, "Lem: reusable engineering of real-world semantics," in *Proc. ICFP*, 2014.

[22] P. Misra and N. Dutt, eds., *Processor Description Languages.* Morgan Kaufmann, 2008.

[23] 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* (L. Beringer and A. P. Felty, eds.), vol. 7406 of *Lecture Notes in Computer Science*, pp. 338–344, Springer, 2012.

[24] 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, (Austin, TX), pp. 18:91–18:98, FMCAD Inc, 2014.

[25] X. Shi, J.-F. Monin, F. Tuong, and F. Blanqui, "First steps towards the certification of an ARM simulator using Compcert," in *Certified Programs and Proofs* (J.-P. Jouannaud and Z. Shao, eds.), vol. 7086 of *Lecture Notes in Computer Science*, pp. 346–361, Springer Berlin Heidelberg, 2011.

[26] J. Harrison, "Floating-point verification," *Journal of Universal Computer Science*, vol. 13, pp. 629–638, may 2007.

[27] S. Boldo and G. Melquiond, "Flocq: A unified library for proving floating-point algorithms in Coq," in *Proceedings of the 2011 IEEE 20th Symposium on Computer Arithmetic*, ARITH '11, (Washington, DC, USA), pp. 243–252, IEEE Computer Society, 2011.

[28] D. M. Russinoff, "Computation and formal verification of SRT quotient and square root digit selection tables," *IEEE Trans. Comput.*, vol. 62, pp. 900–913, May 2013.

[29] L. Maranget *et al.*, "herdtools." `https://github.com/herd/herdtools`.

[30] L. Fournier, A. Koyfman, and M. Levinger, "Developing an architecture validation suite: Applicaiton to the PowerPC architecture," in *DAC*, pp. 189–194, 1999.

[31] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell, "Modelling the ARMv8 architecture, operationally: Concurrency and ISA," in *Proceedings of POPL*, 2016.