The Semantics of Power and ARM Multiprocessor Machine Code

Jade Alglave[2] Anthony Fox[1] Samin Ishtiaq[3] Magnus O. Myreen[1]
Susmit Sarkar[1] Peter Sewell[1] Francesco Zappa Nardelli[2]

[1]University of Cambridge [2]INRIA [3]Microsoft Research Cambridge

http://www.cl.cam.ac.uk/~pes20/weakmemory/

Abstract

We develop a rigorous semantics for Power and ARM multiprocessor programs, including their relaxed memory model and the behaviour of reasonable fragments of their instruction sets. The semantics is mechanised in the HOL proof assistant.

This should provide a good basis for informal reasoning and formal verification of low-level code for these weakly consistent architectures, and, together with our x86 semantics, for the design and compilation of high-level concurrent languages.

Categories and Subject Descriptors C.1.2 [*Multiple Data Stream Architectures (Multiprocessors)*]: Parallel processors; D.1.3 [*Concurrent Programming*]: Parallel programming; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]

General Terms Documentation, Reliability, Standardization, Theory, Verification

Keywords Relaxed Memory Models, Semantics, PowerPC, ARM

1. Introduction

Parallelism is finally going mainstream, but, despite 40 years of research on concurrency, programming and reasoning about concurrent systems remains very challenging. A key issue is that most research has implicitly assumed that memory is sequentially consistent, whereas the reality is that typical multiprocessor architectures —including x86, Sparc, Power, Itanium, ARM, and Alpha— only provide *relaxed* (or *weak*) memory models. For performance reasons, their implementations involve out-of-order execution (reordering operations within processor pipelines), and multiple levels of caching and write buffering, to reduce latency. These microarchitectural optimizations have observable effects at the assembly language level: different processors can see the events of a complete execution in different orders; they can have inconsistent views of the shared memory. It would not be sound to think about such a system in terms of an intuitive interleaving semantics, in which the events from different processors act on a single global state. Instead, we need to understand the subtle guarantees that each architecture provides, both for low-level programming, including implementation of concurrency libraries and OS or hypervisor kernels, and for designing the new high-level concurrent languages that are so urgently needed, so that they can be efficiently compiled down to these architectures.

Previous work on multiprocessor memory models has addressed various more-or-less idealised architectures, and several tutorials have been published [AG96, Luc01, LJV97]. However, typical vendor specifications are still expressed only in informal prose, supplemented by a few litmus-test examples of small programs. Inevitably this is ambiguous and incomplete. The Itanium and SPARC have vendor specifications in semi-formal mathematics [Ita, Spa], but these still leave ample room for interpretation [HJK06, YGLS04, PD95], and the situation for x86, Power, and ARM leaves a great deal of scope for confusion. Moreover, none of the previous work gives a complete semantics for multiprocessor programs, as it does not integrate the memory models with semantics for instructions.

In this paper we describe a semantics for multiprocessor Power and ARM programs, in a declarative axiomatic style. It is precise, formalised in the HOL proof assistant [HOL]. The memory model (in Section 2) is integrated with instruction semantics and decoding (in Section 3) for reasonable fragments of the instruction sets, including various ALU operations, branches, loads and stores, reservations, and barriers (we do not model page tables, access to device memory, or exceptions). We also discuss a number of litmus tests (Section 4) and some initial empirical testing against particular processors (Section 5). We believe the semantics to be reasonably accurate, but this is work in progress: in future work we plan more extensive testing, and we welcome feedback from programmers and architects with experience in these architectures. We plan also to prove metatheoretic results, as in our complementary work on x86-CC [SSZN$^+$09], where we proved results about data-race-free programs and an abstract-machine version of the memory model.

Such semantics should provide a good basis for informal reasoning about the majority of low-level concurrent algorithms, for formal verification and model-checking of such algorithms, and for the design of high-level language support for concurrency, all above the real multiprocessor architectures that we have today.

DAMP'09, January 20, 2009, Savannah, Georgia, USA.

2. The Power/ARM Axiomatic Memory Model

Our memory model is in a axiomatic style, specifying the legal orders of events in a valid execution of a multiprocessor program, as in previous formal work on relaxed memory models. A single instruction may involve a non-atomic collection of several reads or writes, so one cannot simply reorder whole instructions. Instead one has to work at the finer granularity of events that are individual reads and writes to memory. Our events also include individual reads and writes to processor registers (in contrast to most previous work), as the interplay between dependencies through registers and through memory is a key aspect of the model, and as the instruction semantics must involve register accesses.

We aim to capture Power and ARM *architectures*, rather than the behaviour of particular devices. The architectures are specifications of what can be relied upon by assembly-level programmers, for whole families of past and (generally) future processors. They change relatively slowly, and are very loose specifications, to admit a wide variety of processor implementations. Our Power definition is based primarily on the Power ISA Version 2.05 specification [pow07] (applicable to POWER6 and POWER5 processors), particularly Book II Ch.1, §3.4, and Appendix B, together with articles [SF95, LHF05], and reference to the PowerPC Book E [IBM02]. Our ARM definition is based upon the ARM Architecture Reference Manual [ARM08a] (applicable to ARMv7 processors), particularly §A3.8, and the Barrier Litmus Test Cookbook [ARM08b].

We choose to address Power and ARM for several reasons. First, they are both widely used: POWER processors in high-end servers, and both PowerPC and ARM in embedded devices. Even the latter are expected to become multicore, for power/performance reasons, e.g. with the ARM11 MPCore (up to 4 cores) and ARM Cortex-A9 MPCore.

Second, they give an interesting contrast to our x86 work. The current informal-prose x86 vendor specifications, formalised in our x86-CC memory model [SSZN$^+$09], are, very roughly, causal consistency: x86-CC has a single transitively closed happens-before relation, which has to be respected in all processors' views, and very strong "LOCK"'d instructions, which are atomic and are seen in the same order by all processors.[1] The Power memory model, on the other hand, is weakly consistent. It allows more local reordering, and provides load-reserve/store-conditional and a variety of barrier primitives. The ARM memory model is very close to that for Power, differing only in the barrier semantics.

Third, there is little previous literature on memory models for these architectures. For PowerPC there are models by Corella et al. [CSB] and Adir et al. [AAS03]; for ARM there is the initial work of Chong and Ishtiaq [CI08].

Fourth, there has been extensive work on precise semantics for both PowerPC and ARM in the single-processor case. Leroy's verified compiler research [Ler06] is founded on a Coq model of PowerPC instructions, and Fox has verified correspondence between an earlier ARM ISA specification and a microarchitectural model, in HOL [Fox03].

For both architectures, we aim to cover the fragment required for typical low-level concurrent algorithms in main memory, as they might appear in user or OS kernel programs. We do not deal with explicit manipulation of page tables, cache hints, self-modifying code, and so forth, and for Power at present we cover only one of the barrier instructions (`sync 0`, but not `lwsync`, `eieio`, `mbar`, or `isync`).

Our definitions are mechanised in the HOL proof assistant [HOL], and the axiomatic memory model definitions are also mechanised in the Coq proof assistant [Coq], hand-translated from the HOL. For lack of space only key extracts are included here, but the full details are freely available on-line [wea08]. Most of the definitions in this paper are automatically typeset from the HOL, reducing the scope for error.

2.1 Basic Types

The basic types of our model, introduced below, describe what a candidate execution is. They are very similar to those we used for x86, which makes it easier to compare the three models. It also lets us reuse tools, as we discuss in §5.

We take types address and value to both be the 32-bit words, and take a location to be either a memory address or a register of a particular processor. The memory model is polymorphic on a type $'reg$ of registers, later instantiated to the Power or ARM registers as appropriate. There are also pseudo-locations Location_res and Location_res_addr used to express the semantics of reservations (§2.7).

$$\begin{aligned}\mathsf{location} = {} & \textsc{Location\_reg}\ \textbf{of}\ \mathsf{proc}\ 'reg \\ & \mid \textsc{Location\_mem}\ \textbf{of}\ \mathsf{address} \\ & \mid \textsc{Location\_res}\ \textbf{of}\ \mathsf{proc} \\ & \mid \textsc{Location\_res\_addr}\ \textbf{of}\ \mathsf{proc}\end{aligned}$$

These constructors are curried, so $\textsc{Location\_reg} : \mathsf{proc} \rightarrow 'reg \rightarrow \mathsf{location}$. To identify an instance of an instruction in an execution, we specify its processor and its index in program order (i.e., in the program with an unfolding of all branches):

$$\begin{aligned}\mathsf{iiid} = {} & \langle\!\langle\ proc : \mathsf{proc}; \\ & \quad poi : \mathsf{num} \rangle\!\rangle\end{aligned}$$

An action is either a read or write of a value at some location, or a synchronisation barrier (used as a marker to define the Power `sync` and the ARM `DMB`):

$$\mathsf{dirn} = \mathrm{R} \mid \mathrm{W}$$

$$\mathsf{synchronization} = \textsc{Sync}$$

$$\begin{aligned}\mathsf{action} = {} & \textsc{Access}\ \textbf{of}\ \mathsf{dirn}\ (' reg\ \mathsf{location})\ \mathsf{value} \\ & \mid \textsc{Barrier}\ \textbf{of}\ \mathsf{synchronization}\end{aligned}$$

A typical action might be Access W (Location_mem 100) 5, for a write of 5 to memory address 100. Finally, an event has an instruction instance id, an event id (of type $\mathsf{eiid} = \mathsf{num}$, unique among the events of this iiid), and an action:

$$\begin{aligned}\mathsf{event} = {} & \langle\!\langle\ eiid : \mathsf{eiid}; \\ & \quad iiid : \mathsf{iiid}; \\ & \quad action : \mathsf{action} \rangle\!\rangle\end{aligned}$$

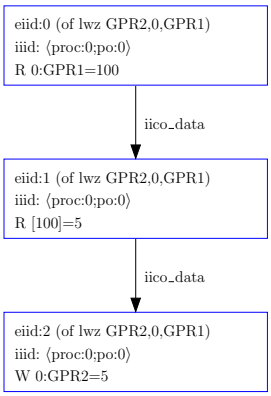
The semantics of a single instruction must also record any intra-instruction causality relationships among its events. The Power and ARM architectures make a subtle distinction between causality due to data dependency, e.g. a write to an address (or of a value) that was taken from a register, and a control dependency, e.g. a write that was conditional on a particular flag value.

For example, the Power instruction lwz GPR2,0,GPR1 below, loading register GPR2 with the value from memory at the location in register GPR1, has an event structure

[1] It appears that actual x86 processors provide an even stronger TSO-based memory model, and the vendor specifications are expected to change to reflect that [SSZN$^+$09, Addendum].

with a register read, a memory read, and a register write, all linked by intra-instruction-causality-data:

ppc-lwz	proc:0
poi:0	lwz GPR2,0,GPR1
Initial state: 0:GPR1= 100; [100]= 5	

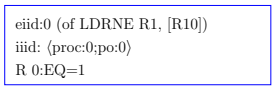


ppc-lwz: (event structure 1)

The ARM LDRNE R1, [R10], on the other hand, conditionally loads register R1 with the value of memory at the location in R10, if flag EQ is zero.

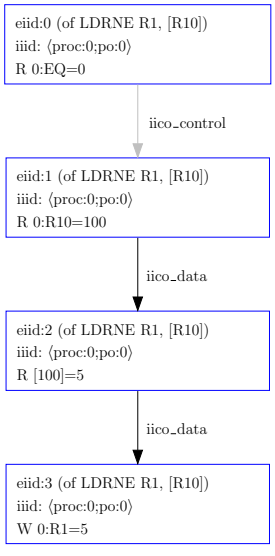
ARM-ldrne	proc:0
poi:0	LDRNE R1, [R10]

It therefore has two families of event structures — those in which there is a single event, reading a non-zero flag value, and nothing else happens:



ARM-ldrne: (event structure 2)

and those in which it reads a zero flag value, with an intra-instruction-causality-control relation to the read of R10:



ARM-ldrne: (event structure 1)

Collecting this data together, we define an event structure E to comprise a set of events, an intra-instruction data causality relation, and an intra-instruction control causality relation. It also specifies an architecture (Power205 or ARMv7, see §2.5), and a reservation granule size and atomicity relation (see §2.7).

```
event_structure =⟪ events : ('reg event)set;
                   intra_causality_data : ('reg event)reln;
                   intra_causality_control : ('reg event)reln;
                   atomicity : ('reg event)set set;
                   arch : architecture;
                   granule_size_exponent : num⟫
```

This is subject to various well-formedness conditions (such as that the event pairs in the intra-causality relation are, indeed, elements of the event set), which we omit here.

Given an event structure, a candidate execution witness consists of an initial state constraint and a processor-indexed family of view orders, together with a write serialization that we explain in §2.3. The rest of this section is devoted to defining when such a candidate is in fact a valid execution.

```
type_abbrev state_constraint = location → value option

type_abbrev view_orders : proc → ('reg event)reln

execution_witness =
⟪ initial_state : ('reg state_constraint);
  vo : ('reg view_orders);
  write_serialization : ('reg event reln)⟫
```

A well-formed view order for processor p is a strict linear order over all of its events together with all the memory write events of other processors; we write viewed_events $E\ p$ for that union.

2.2 Read Values

Our first condition simply says that any read event, of a memory or register location (or a reservation address), reads the value either of the most recent write to that location, if there is one, or otherwise from the initial state. Here "most recent" is with respect to the view order of the reading processor. The state_updates auxiliary finds the write events before a given event e that write to the same location.

```
state_updates E vo e =
{ew | ew ∈ (writes E) ∧ (ew, e) ∈ vo(proc e) ∧
      (loc ew = loc e)}


read_most_recent_value E initial_state vo =
∀e ∈ (E.events).∀l v.
  ((e.action = Access R l v) ∧ reg_or_mem_or_resaddr l)
      ⟹
  (if (state_updates E vo e) = {} then
      (Some v = initial_state l)
   else
      ((Some v) ∈ {value_of ew |
        ew ∈ maximal_elements(state_updates E vo e)
                             (vo(proc e))}))
```

2.3 Coherence

Both architectures (when in the appropriate mode) provide *coherent* memory: writes by two processors to the same memory location must be seen by all processors in the same order. In other words, for each memory location, there exists a strict linear order on the stores to this location, and this must be respected by all the view orders. We formalise this by defining the candidate write serializations, each of which

is a union of a candidate order over the writes for each memory location.

$$\begin{array}{l}\text{get\_mem\_l\_stores } E\ l = \\ \{e \mid e \in E.events \wedge \text{mem\_store } e \wedge (\text{loc } e = \text{Some } l)\}\end{array}$$

$$\begin{array}{l}\text{write\_serialization\_candidates } E\ cand = \\ (\forall (e_1, e_2) \in cand. \\ \quad \exists l. e_1 \in (\text{get\_mem\_l\_stores } E\ l) \wedge \\ \qquad e_2 \in (\text{get\_mem\_l\_stores } E\ l)) \wedge \\ (\forall l. \text{strict\_linear\_order}(cand|_{(\text{get\_mem\_l\_stores } E\ l)}) \\ \qquad\qquad (\text{get\_mem\_l\_stores } E\ l))\end{array}$$

We require that the write serialization of an execution is included in each processor's view order, and that the orderings of writes by each processor to the same location, captured with preserved_coherence_order below, are included in the write serialization.

$$\begin{array}{l}\text{preserved\_coherence\_order } E\ p = \\ \{(e_1, e_2) \mid (e_1, e_2) \in \text{po\_iico\_data } E \wedge \\ \quad (\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p) \wedge (\text{loc } e_1 = \text{loc } e_2) \wedge \\ \quad (\text{mem\_store } e_1 \wedge \text{mem\_store } e_2)\}\end{array}$$

Here po_iico_data is the union of the strict program order relation and the intra-causality-data relation:

$$\begin{array}{l}\text{po\_strict } E = \\ \{(e_1, e_2) \mid (e_1.iiid.proc = e_2.iiid.proc) \wedge \\ \qquad e_1.iiid.poi < e_2.iiid.poi \wedge \\ \qquad e_1 \in E.events \wedge e_2 \in E.events\}\end{array}$$

$$\text{po\_iico\_data } E = \text{po\_strict } E \cup E.intra\_causality\_data$$

$$\begin{array}{l}\text{po\_iico\_both } E = \text{po\_strict } E \cup E.intra\_causality\_data \cup \\ \qquad\qquad E.intra\_causality\_control\end{array}$$

Note that coherence does *not* require that writes to two different locations must be seen by all processors in the same order.

2.4 Preserved Program Order

The essence of a weakly consistent model, such as those of Power or ARM, is that a processor implementation is free to re-order its own operations rather liberally, in between synchronisation operations, so long as dependencies are respected — any reordering not forbidden is permitted. The dependencies that are respected, however, are subtly less than what one might expect, and we need to take several steps to define them. Moreover, the documentation speaks of dependencies between instructions, but again we actually need to deal in terms of dependencies between events, using the intra-instruction causality as appropriate.

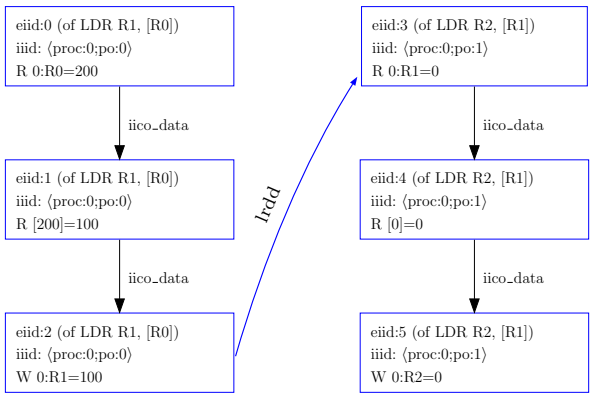
First, we identify the data dependencies through the registers of a single processor, picking out the pairs of events in program order (strictly, in po_iico_data), on a processor p, where e_1 is a write to a register and e_2 is a read from the same register, with no intervening write to the same register:

$$\begin{array}{l}\text{local\_register\_data\_dependency } E\ p = \\ \{(e_1, e_2) \mid \\ \quad (e_1, e_2) \in \text{po\_iico\_data } E \wedge \\ \quad (\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p) \wedge \\ \quad (\exists r\ v_1\ v_2. \\ \qquad (e_1.action = \text{Access W}(\text{Location\_reg } p\ r) v_1) \wedge \\ \qquad (e_2.action = \text{Access R}(\text{Location\_reg } p\ r) v_2) \wedge \\ \qquad (\neg(\exists e_3\ v_3. (e_1, e_3) \in \text{po\_iico\_data } E \wedge \\ \qquad\quad (e_3, e_2) \in \text{po\_iico\_data } E \wedge \\ \qquad\quad (e_3.action = \text{Access W}(\text{Location\_reg } p\ r) v_3))))\}\end{array}$$

For example, in the short ARM program below, there is a local register data dependency from the write of R1 in the first instruction to the read of R1 in the second.

ARM-depend-A	proc:0
poi:0	LDR R1, [R0]
poi:1	LDR R2, [R1]

A sample event structure for that program is shown below. Note that the intra-instruction causality relations are part of the event structure data, provided by the instruction semantics, whereas local register data dependency (lrdd) is calculated from the event structure and candidate execution.



ARM-depend-A: (event structure 1)

For dependencies from a memory load to a memory load, we pick out the pairs (e_1, e_2) of two memory loads, in program order and by the same processor (though perhaps to different addresses), which are transitively related by the union of local register data dependency and the intra-instruction data causality relation (perhaps counter-intuitively, the intra-instruction control relation is *not* included):

$$\begin{array}{l}\text{address\_or\_data\_dependency\_load\_load } E\ p = \\ \{(e_1, e_2) \mid \\ \quad (\text{mem\_load } e_1 \wedge \text{mem\_load } e_2 \wedge (e_1, e_2) \in \text{po } E \wedge \\ \quad (\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p) \wedge \\ \quad (e_1, e_2) \in ((E.intra\_causality\_data \cup \\ \qquad \text{local\_register\_data\_dependency } E\ p)^+))\}\end{array}$$

Here r^+ denotes the transitive closure of a relation r.

For example, in the ARM program below, the LDRNE instruction is conditional on the EQ flag.

ARM-nodep	proc:0
poi:0	LDR R1, [R4]
poi:1	CMP R1, #1
poi:2	LDRNE R2, [R5]

Even if the value read from that flag is 0, as in Fig. 1, there is no address_or_data_dependency_load_load from the first memory load to the memory load of the LDRNE.

However, in the program below [ARM08b, §6.2.1.2b] there *is* an address_or_data_dependency_load_load from the first memory load to the second memory load, despite the fact that this is a "false" dependency, i.e. the address used by the second load is not affected by the value read

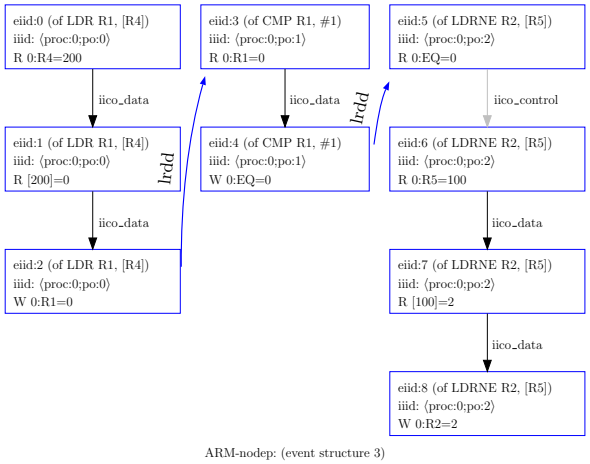


Figure 1.

by the first. Introducing such "false" dependencies can be useful to constrain execution orders.

ARM-depend-B	proc:0
poi:0	LDR R1, [R0]
poi:1	AND R1, R1, #0
poi:2	LDR R2, [R3,R1]

Dependencies from a memory load to a memory store are similar, except that here both intra-instruction relations are included:

$$\begin{array}{l}\text{address\_or\_data\_or\_control\_dependency\_load\_store } E\ p = \\ \{(e_1, e_2) \mid \\ \quad (\text{mem\_load } e_1 \wedge \text{mem\_store } e_2 \wedge (e_1, e_2) \in \text{po } E \wedge \\ \quad (\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p) \wedge \\ \quad (e_1, e_2) \in ((E.\mathit{intra\_causality\_data} \cup \\ \qquad E.\mathit{intra\_causality\_control} \cup \\ \qquad \text{local\_register\_data\_dependency } E\ p)^+))\}\end{array}$$

The program below [ARM08b, §6.2.1.2c, modified with a STR instead of a LDR at the end] illustrates such an address_or_data_or_control_dependency_load_store, passing through the intra-instruction-control relation. In this case, the execution (or not) of the memory store depends on the value returned by the memory load.

ARM-depend-C	proc:0
poi:0	LDR R1, [R0]
poi:1	CMP R1, #55
poi:2	STRNE R2, [R3]

The next example [ARM08b, §6.2.1.2d, likewise modified] is similar, except that here the result of the conditional MOV influences the address to which the store access is performed.

ARM-depend-D	proc:0
poi:0	LDR R1, [R0]
poi:1	CMP R1, #55
poi:2	MOVNE R4, #0
poi:3	STR R2, [R3,R4]

Note that the semantics does not require the concept of a pure control dependency.

We now turn to the special case of memory accesses to the same address. Here we include all such pairs: omitting load/load or load/store would make a nonsense of the coherence property; store/store is also enforced by coherence, so harmless to include here; and without store/load a load could fail to read from a program-order-past store:

$$\begin{array}{l}\text{preserved\_program\_order\_mem\_loc } E\ p = \\ \{(e_1, e_2) \mid (e_1, e_2) \in \text{po\_iico\_data } E \wedge \\ \quad (\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p) \wedge (\text{loc } e_1 = \text{loc } e_2) \wedge \\ \quad \text{mem\_access } e_1 \wedge \text{mem\_access } e_2\}\end{array}$$

Finally, we collect together the above, together with the intra-instruction data and control relations (restricted to the processor p in question):

$$\begin{array}{l}\text{preserved\_program\_order } E\ p = \\ \{(e_1, e_2) \mid (e_1, e_2) \in E.\mathit{intra\_causality\_data} \wedge \\ \qquad\qquad (\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p)\} \cup \\ \{(e_1, e_2) \mid (e_1, e_2) \in E.\mathit{intra\_causality\_control} \wedge \\ \qquad\qquad (\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p)\} \cup \\ \text{address\_or\_data\_dependency\_load\_load } E\ p \cup \\ \text{address\_or\_data\_or\_control\_dependency\_load\_store } E\ p \cup \\ \text{preserved\_program\_order\_mem\_loc } E\ p\end{array}$$

Note that program order between two register accesses, even to the same register, is *not* automatically preserved (but only if there is a local_register_data_dependency as above). The documentation [pow07, p.414] states "Because processors may implement nonarchitected duplicates of architected resources (e.g. GPRs, CR fields, and the Link Register), resource dependencies (e.g. specification of the same target register for two Load instructions) do not order storage accesses." This is illustrated by the ppc.reg example in §4, which our model admits.

2.5 Barriers

To make concurrent programming feasible, given the liberal reordering of normal operations that is permitted on Power and ARM (as described above), both architectures provide a variety of synchronisation instructions. Power 2.05 provides `sync L`, where `L` can be 0 ("heavyweight sync"), 1 ("lightweight sync") or 2 (in the Server Environment specification only); an `eieio` (Server) or `mbar MO` (Embedded), sharing the same opcode, intended for memory-mapped I/O; and an `isync`, to synchronise the instruction stream. For the fragment of the ISA we consider, the guarantees provided by `sync 0` are the same as those of `mbar 0`. Broadly, an `eieio` provides weaker synchronisation, for stores only; an `lwsync` provides synchronisation for memory accesses except store/load pairs, and `mbar` for `MO` $\neq$ `0` is implementation-defined. ARMv7 provides `DMB` ("Data Memory Barrier"), very similar to the Power `sync 0`, and the stronger `DSB` ("Data Synchronisation Barrier") which also synchronises the execution stream, including cache, branch predictor, and TLB maintenance operations. We formalise the Power `sync 0` and ARM `DMB`.

Each instance of such a barrier instruction in an execution defines two groups of memory access events: Group A, in some sense those 'before' the barrier, and Group B, in some sense 'after' the barrier. It ensures that all members of Group A precede all members of Group B, in the view orders of *all* processors. To express this, we start by defining those instructions to generate a special Barrier Sync event, which appears (only) in the view order of the processor that executed the instruction.

```
check_sync_power_2_05 E vos =
  ∀es ∈ (E.events).(es.action = BARRIER SYNC) ⟹
  let group_A = {e | ((e, es) ∈ po E ∨ (e, es) ∈ vos(proc es)) ∧ mem_access e} in
  let group_B_base = {e | (es, e) ∈ po E ∧ mem_access e} in
  let group_B_ind B0 =
    {e | mem_access e ∧
         (¬(proc e = proc es)) ∧
         ∃er. mem_load er ∧ (er, e) ∈ vos(proc er) ∧ (proc er = proc e) ∧
              ∃ew. mem_store ew ∧ ew ∈ B0 ∧ (ew, er) ∈ vos(proc er) ∧ (loc er = loc ew) ∧
                   (¬(∃ew'.(ew, ew') ∈ vos(proc er) ∧ (ew', er) ∈ vos(proc er) ∧
                        (loc ew' = loc er) ∧ mem_store ew'))} in
  let group_B = FIX group_B_ind group_B_base in
  ∀p ∈ (procs E).∀ea ∈ group_A.∀eb ∈ group_B.(ea ∈ viewed_events E p ∧ eb ∈ viewed_events E p) ⟹
    if (p = es.iiid. proc) then ((ea, es) ∈ vos p ∧ (es, eb) ∈ vos p) else (ea, eb) ∈ vos p

check_dmb_arm E vos =
  ∀es ∈ (E.events).(es.action = BARRIER SYNC) ⟹
  let group_A_base = {e | ((e, es) ∈ vos(proc es)) ∧ mem_access e} in
  let group_A_ind A0 = {er | mem_load er ∧
                             ∃e ∈ A0.(er, e) ∈ vos(proc e)} in
  let group_B_base = {e | (es, e) ∈ po E ∧ mem_access e} in
  let group_B_ind B0 = {e | mem_access e ∧
                            ∃ew. mem_store ew ∧ ew ∈ B0 ∧ (ew, e) ∈ vos(proc e)} in
  let group_A = FIX group_A_ind group_A_base in
  let group_B = FIX group_B_ind group_B_base in
  ∀p ∈ (procs E).∀ea ∈ group_A.∀eb ∈ group_B.(ea ∈ viewed_events E p ∧ eb ∈ viewed_events E p) ⟹
    if (p = es.iiid. proc) then ((ea, es) ∈ vos p ∧ (es, eb) ∈ vos p) else (ea, eb) ∈ vos p
```

Figure 2. Barrier Semantics

For Power, given such a barrier event *es* from processor p, Group A consists of all memory access events that precede *es* in either program order or in p's view order. Group B is defined inductively. It includes:

- all memory access events that follow *es* in the p program order; and
- all memory access events e, on other processors p', that follow (in the p' view order) a memory read event *er* (by p') that reads the value from a memory write event *ew* in Group B (i.e., *er* follows *ew* in the p' view order, they are to the same location, and there is no intervening write *ew'* to the same location in that view order).

For any *ea* in Group A and *eb* in Group B, and for all processors, whenever *ea* and *eb* are in the viewed_events for that processor (i.e. they are either memory reads by that processor, or memory writes by any processor), we require that *ea* precedes *eb* in that processor's view order. For the processor that executed the synchronisation instruction, we also require that the barrier event lies between *ea* and *eb*. The HOL statement of this is in Figure 2.

For ARM, the Architecture Reference Manual [ARM08a] and the Barrier Litmus Test Cookbook [ARM08b] give two distinct specifications. The Cookbook specification matches the Power semantics we describe above, except that there is no requirement that $p' \neq p$. It may be that this (rather strange) requirement is an error in the Power 2.05 specification — the text there speaks of "processors and mechanisms other than P1" [pow07, II,p.413], but perhaps means "in addition to P1".

The ARM Architecture Reference Manual specification is quite different. Its Group A is also inductive, with a base case of all memory access events that precede *es* in p's view order (but apparently not including those preceding *es* in program order), and an inductive case adding memory reads *er* by any processor that precede an event e in Group A in that processor's view order. Its Group B base case is the same as in Power, but its Group B inductive case adds all memory accesses e that follow, in the view order of proc e, a memory write in Group B. This is also formalised in Figure 2, and at present our top-level definition uses this, not the Cookbook version, for ARM.

2.6 A-Cumulativity and 'Performed'

The Power specification [pow07, p.413] says the ordering provided by a barrier is "cumulative" if

> "it also orders storage accesses that are performed by processors and mechanisms other than P1, as follows.
> - A includes all applicable storage accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
> - B includes all applicable storage accesses by any such processor or mechanism that are performed after a load instruction executed by that processor or mechanism has returned the value stored by a store that is in B."

and that `sync` is indeed cumulative. Here "performed" is defined informally as follows [pow07, p.408]:

> "A load or instruction fetch by a processor or mechanism (P1) is performed with respect to any processor or mechanism (P2) when the value to be returned by the load or instruction fetch can no longer be changed by a store by P2. A store by P1 is per-

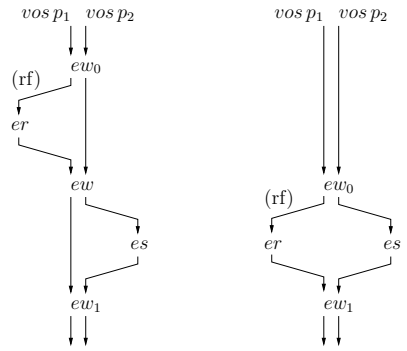


Figure 3. Additional A-cumulativity read cases *er*

> formed with respect to P2 when a load by P2 from the location accessed by the store will return the value stored (or a value stored subsequently). [...] The preceding definitions apply regardless of whether P1 and P2 are the same entity."

This does not lend itself to a straightforward direct formalisation, for two reasons. First, it implicitly refers to global time. This is not a real problem, as we only need to consider when one access is performed before or after another. Second, more seriously, it is subjunctive: the first clause refers to a hypothetical store by P2, and the second to a hypothetical load by P2. Our formalisation defines when a specific concrete execution is admitted, and it would be very awkward to express conditions in terms of modified versions of that execution with such hypothetical accesses added.

In most cases where "performed" is used in the informal specification, one is asking whether one memory access e_1 is performed before another e_2 with respect to some processor p, where both accesses are either writes (by any processor) or reads by p. It then seems sufficient to interpret "e_1 is performed before e_2 w.r.t p" by $(e_1, e_2) \in vos\ p$, i.e. that e_1 precedes e_2 in p's view order.[2]

The only exception is for A-cumulativity, where we believe that the Figure 2 definition of *group_A* is incomplete[3]. Two alternative plausible extensions can be phrased in terms of our view orders.

Conservatively, one could add the reads *er* by any other processor p_1 which are before (in the p_1 view order) some write *ew*, to the same address, that precedes the sync *es* (in the syncing processor p_2's view order). Such a read *er* must take its value from some earlier write (or the initial state).

$$\begin{array}{l}\{er \mid \text{mem\_load}\ er \wedge (\neg(\text{proc}\ er = \text{proc}\ es)) \wedge \\ \quad \exists ew.\, \text{mem\_store}\ ew \wedge (\text{loc}\ ew = \text{loc}\ er) \wedge \\ \qquad (ew, es) \in vos(\text{proc}\ es) \wedge \\ \qquad (er, ew) \in vos(\text{proc}\ er)\}\end{array}$$

More liberally, one could also include reads *er* that read from the last write ew_0 (to that location) before the `sync`, or, in other words, the reads that do not read from a write after the `sync`:

$$\begin{array}{l}\{er \mid er \in E.events \wedge \text{mem\_load}\ er \wedge \\ \quad (\neg(\text{proc}\ er = \text{proc}\ es)) \wedge \\ \quad \neg\exists ew.\, \text{mem\_store}\ ew \wedge (\text{loc}\ ew = \text{loc}\ er) \wedge \\ \qquad (es, ew) \in vos(\text{proc}\ es) \wedge \\ \qquad (ew, er) \in vos(\text{proc}\ er)\}\end{array}$$

These are illustrated in Figure 3 (with (rf) indicating the reads-from relationships). At present we do not know whether either of them matches the architect's intentions.

Another, more radical, possibility would be to add other-processor's reads to the view orders, but one would then also need subtle conditions constraining where in the view orders they appear.

2.7 Reservations

The Power and ARM instruction sets both include load-reserve and store-conditional instructions, e.g. `lwarx/stwcx` and `LDREX/STREX`. These are intended to be used in pairs: a load-reserve loads a value from a memory address and establishes a reservation for the memory granule including that address; to a first approximation, a later store-conditional by the same processor to the same address succeeds iff no other processor has written to that memory granule since.

We express this semantics with the two special locations per processor: LOCATION_RES (notionally carrying just one bit, but embedded in a word32), and LOCATION_RES_ADDR (carrying the reserved address). The latter behaves just like a register, though in the instruction semantics it is only read when the former is 1 ($1w$ in HOL). The former, LOCATION_RES, can be read and written by load-reserve and store-conditional instructions, but the value read is not the most recent value written, instead being computed by location_res_value below. This walks over the relevant view order to find the most recent (in that view order) reservation, if any, and checks that there has been no intervening (in that view order) memory write to the same reservation granule.

$$\begin{array}{l}\text{location\_res\_value}\ E\ vo\ e = \\ \textbf{let}\ \mathit{prior\_reservations} = \{ew \mid ew \in (\text{writes}\ E) \wedge \\ \quad (ew, e) \in vo(\text{proc}\ e) \wedge \\ \quad (\text{loc}\ ew = \text{SOME}\ (\text{LOCATION\_RES\_ADDR}(\text{proc}\ e)))\}\ \textbf{in} \\ \textbf{if}\ (\mathit{prior\_reservations} = \{\})\ \textbf{then}\ 0w\ \textbf{else} \\ \quad \textbf{let}\ \mathit{reservation} = \text{maximal\_elements} \\ \qquad \mathit{prior\_reservations}(vo(\text{proc}\ e))\ \textbf{in} \\ \quad \textbf{let}\ \mathit{intervening\_writes} = \{ew \mid \\ \qquad \exists ew' \in \mathit{reservation}.\exists a'.\exists a\ v. \\ \qquad\quad (ew.\mathit{action} = \text{ACCESS W}(\text{LOCATION\_MEM}\ a)v) \wedge \\ \qquad\quad (ew', ew) \in vo(\text{proc}\ e) \wedge (ew, e) \in vo(\text{proc}\ e) \wedge \\ \qquad\quad (\text{value\_of}\ ew' = \text{SOME}\ a') \wedge \\ \qquad\quad \text{same\_granule}\ E\ a\ a'\}\ \textbf{in} \\ \quad \textbf{if}\ \mathit{intervening\_writes} = \{\}\ \textbf{then}\ 0w\ \textbf{else}\ 1w\end{array}$$

$$\begin{array}{l}\text{read\_location\_res\_value}\ E\ \mathit{initial\_state}\ vo = \\ \forall e \in (E.\mathit{events}).\forall p\ v. \\ \quad (e.\mathit{action} = \text{ACCESS R}(\text{LOCATION\_RES}\ p)v) \implies \\ \quad (v = \text{location\_res\_value}\ E\ vo\ e)\end{array}$$

Additionally, we need to ensure that accesses to these special locations, and any associated memory access, are atomic. We record this in the *atomicity* field of an event structure, and check it with the predicate below.

[2] Recall that our view orders do not include memory reads by other processors. This choice seems simpler to us w.r.t. the programmer's intuition, though perhaps it is counterintuitive from an architect's point of view, thinking in terms of cache-line ownership in global time.

[3] Thanks to Paul McKenney for this observation.

check_atomicity E vo =
  $\forall p \in (\text{procs } E).\forall es \in (E.atomicity).$
    $\forall e_1\ e_2 \in es.(e_1, e_2) \in (vo\ p) \implies$
      $\forall e.(e_1, e) \in (vo\ p) \wedge (e, e_2) \in (vo\ p) \implies e \in es$

Note that any matching load-reserve/store-conditional pair will necessarily be to the same address, and so the definition of preserved_program_order_mem_loc will apply. However, because the value of LOCATION_RES read is computed specially, as above, we do not introduce any dependencies between the events of such a pair and an intervening write by another processor.

2.8 Valid Executions

Finally we can define when an execution X over an event structure E is valid. This simply collects together the axioms stated above, together with three conditions preventing intervening register writes between a preserved write/read pair. In brief: the view orders are well formed; read events read the most recent value written; the write serialisation is a proper candidate; for each processor p, its preserved coherence order is in the write serialization; for each processor p, its view order contains the write serialization, preserved program order, and local register data dependency; there are no intervening writes between a register write and read from local register data dependency, and similarly for register writes before a read from the initial state or write to the final state; the `sync` or `DMB` condition is checked; the reservation values are correct; and the atomicity relation is respected.

valid_execution E X =
  view_orders_well_formed E $X.vo \wedge$
  read_most_recent_value E $X.initial\_state$ $X.vo \wedge$
  $X.write\_serialization \in$ write_serialization_candidates $E \wedge$
  $(\forall p \in (\text{procs } E).$
    preserved_coherence_order E $p \subseteq X.write\_serialization \wedge$
    $X.write\_serialization \subseteq X.vo\ p \wedge$
    preserved_program_order E $p \subseteq X.vo\ p \wedge$
    (*no intervening writes in local register data dependency*)
    local_register_data_dependency E $p \subseteq X.vo\ p \wedge$
    $(\forall (e_1, e_2) \in (\text{local\_register\_data\_dependency } E\ p).$
      $\neg(\exists e_3.(e_1, e_3) \in X.vo\ p \wedge (e_3, e_2) \in X.vo\ p \wedge$
        $(\text{loc } e_3 = \text{loc } e_1) \wedge \text{store } e_3))) \wedge$
  (*no intervening writes before a reg read from initial state*)
  $(\forall e \in (E.events).(\text{reg\_load } e \wedge$
    $(\neg(\exists e_0.(e_0, e) \in \text{po\_iico\_both } E \wedge \text{reg\_store } e_0 \wedge$
      $(\text{loc } e_0 = \text{loc } e)))) \implies$
    $(\neg(\exists e_0.(e_0, e) \in X.vo(\text{proc } e) \wedge \text{reg\_store } e_0 \wedge$
      $(\text{loc } e_0 = \text{loc } e)))) \wedge$
  (*no intervening writes after a reg write to the final state*)
  $(\forall e \in (E.events).(\text{reg\_store } e \wedge$
    $(\neg(\exists e_1.(e, e_1) \in \text{po\_iico\_both } E \wedge \text{reg\_store } e_1 \wedge$
      $(\text{loc } e_1 = \text{loc } e)))) \implies$
    $(\neg(\exists e_1.(e, e_1) \in X.vo(\text{proc } e) \wedge \text{reg\_store } e_1 \wedge$
      $(\text{loc } e_1 = \text{loc } e)))) \wedge$
  (**case** $E.arch$ **of**
    POWER205 $\rightarrow$ check_sync_power_2_05 E $X.vo$
  $\|$ ARMv7 $\rightarrow$ check_dmb_arm E $X.vo$) $\wedge$
  read_location_res_value E $X.initial\_state$ $X.vo \wedge$
  check_atomicity E $X.vo$

2.9 Example

In Fig. 4 we show a simple example of a valid execution, produced by `memevents`, for the ppc3.1 program below. Here we store to two different locations, on two processors, and each reads from the other location. There are very few dependencies in this example, and, in the valid execution shown, both reads are from the initial state. Note that the union of the two view orders (vo:0 and vo:1) is cyclic.

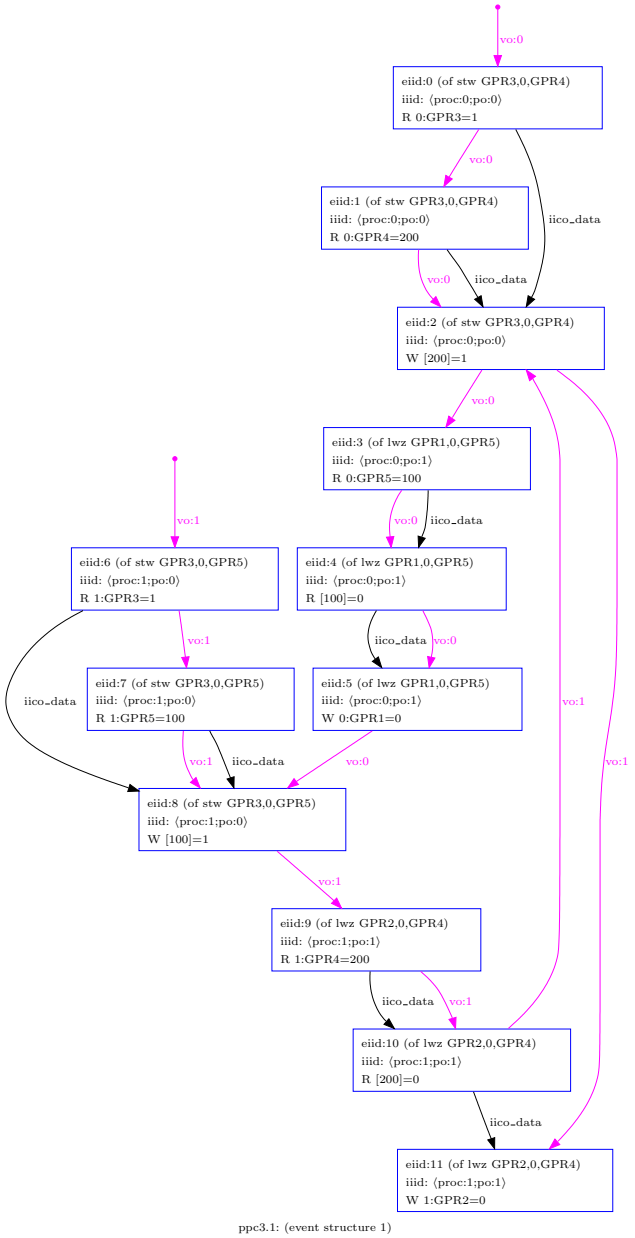


Figure 4. A Valid Execution (ppc3.1)

ppc3.1	proc:0	proc:1
poi:0	stw GPR3,0,GPR4	stw GPR3,0,GPR5
poi:1	lwz GPR1,0,GPR5	lwz GPR2,0,GPR4
Initial state: 0:GPR3= 1; 0:GPR4= 200; 0:GPR5= 100; 1:GPR3= 1; 1:GPR4= 200; 1:GPR5= 100 (elsewhere 0)		
Allowed: 0:GPR1=0 $\wedge$ 1:GPR2=0		

3. Instruction Semantics

As in the x86 model, the overall semantics is factored into two parts: the instruction semantics defines, for any program, a set of candidate event structures, and the axiomatic memory model of the previous section defines, for each event structure, its valid executions.

3.1 ARM

At the time of writing there are seven versions of the ARM instruction set architecture, ARMv1 through ARMv7.[4] The ARM11 MPCore and ARM Cortex-A9 MPCore are ARMv6 and ARMv7 architectures respectively. Versions one (never used in a commercial product) and two are now obselete. Each successive version has provided extensions and minor revisions to the previous version. The revisions have been mostly conservative; for example, behaviours previously categorised as *unpredictable*, or *implementation dependent*, have been specified later as *undefined* (which is handled by an well defined exception entry mechanism) or provided with a semantics.

Architecture ARMv3 was specified by Fox in HOL when verifying a model of the ARM6 micro-architecture [Fox03]. This ISA model was later extended to ARMv4 and that model has been used by Myreen et al. to verify machine code programs [MSG08]. Architecture ARMv4T was subsequently specified, in its entirety, in HOL. These HOL models are conventional, deterministic, functional specifications and are not immediately suited to reasoning about weak memory models. Thus, we have converted and extended these ISA models into a single monadic style specification (covering ARMv3 to ARMv5TE, together with `LDREX`/`STREX` from ARMv6 and `DMB` from ARMv7). The monadic specification supports non-determinism and reasoning about register and memory access events. ARMv6 introduced around 100 other new instructions, but these were mostly extra arithmetic and bit-manipulations (also extra exclusive loads/stores, for bytes, half-words, etc.); our semantics therefore covers essentially all the interesting concurrency cases.

3.2 Power

Our instruction semantics for Power is a HOL4 version of Leroy's specification of PowerPC assembly [Ler06], extended with the `lwarx`, `stwcx`, and `sync` instructions. Leroy's model was translated into HOL4 and a instruction decoder was attached to it in order to make his assembly-level model a machine-level model. The specification defines instructions: `add`, `addi`, `addis`, `addze`, `and.`, `andc`, `andi.`, `andis.`, `b`, `bctr`, `bctrl`, `bf`, `bl`, `bs`, `blr`, `bt`, `cmplw`, `cmplwi`, `cmpw`, `cmpwi`, `cror`, `eqv`, `extsb`, `extsh`, `lbz`, `lbzx`, `lha`, `lhax`, `lhz`, `lhzx`, `lwarx`, `lwz`, `lwzx`, `mflr`, `mr`, `mtctr`, `mtlr`, `mulli`, `mullw`, `nand`, `nor`, `or`, `orc`, `ori`, `oris`, `slw`, `sraw`, `srawi`, `srw`, `stb`, `stbx`, `sth`, `sthx`, `stw`, `stwcx`, `stwx`, `subfc`, `subfic`, `sync`, `xor`, `xori`, `xoris`. Floating-point instructions were omitted.

We have a reasonable level of confidence in the correctness of these instruction semantics: for ARM, based on the verification of the ARMv3 fragment against a microarchitectural model, and Myreen et al.'s verification of machine-code programs above the semantics; for Power, based on Leroy et al.'s extensive work. For both, further confidence will come from empirical testing: as we did for x86 [SSZN$^+$09], we intend to automatically generate HOL conjectures from instrumented execution of instructions on actual processors, and to (automatically) prove that these conjectures are true in the semantics.

[4] There are also numerous optional extensions (variants) to these architectures. For example: Thumb, Thumb-2, DSP and Jazelle extensions.

4. Litmus tests

Memory models are often illustrated (or even 'defined') in terms of behaviours of small litmus-test programs that are either allowed, required, or forbidden. In this section we present a few such tests, as space permits; many more will be required to give reasonable coverage of our model.

Total order on writes to the same location In litmus test ppc6, in Fig. 5 (an analogue of the x86 [SSZN$^+$09, iwp2.6]), two processors concurrently update the same memory location (100, in GPR2) with two different values (1 and 2), while two other processors read twice from that memory location. Coherence (§2.3) requires the existence of a strict linear order on all the stores to one location, and this order must be preserved by all the view orders. This implies that the view orders of processors 2 and 3 must agree on the order in which they see the stores performed by processors 0 and 1, forbidding the final state reported in the figure.

Local reordering of stores or loads to different locations The ARM and Power architectures allow processors to reorder their individual memory accesses rather freely. Test ppc1 below shows that a pair of stores, or a pair of loads, to non-overlapping different memory locations, can be reordered. The specified outcome can be obtained by a possible execution in which the two loads performed by proc:1 are executed in the opposite order to program order; or alternatively a possible execution in which the two stores performed by proc:0 are reordered. This is an analogue of [SSZN$^+$09, iwp2.1/amd1], except that this "Allowed" outcome is forbidden there.

ppc/iwp1	proc:0	proc:1
poi:0	stw GPR1,0,GPR5	lwz GPR3,0,GPR6
poi:1	stw GPR2,0,GPR6	lwz GPR4,0,GPR5
Initial state: 0:GPR1 = 1; 0:GPR2 = 1; 0:GPR5 = x; 0:GPR6 = y; 1:GPR5 = x; 1:GPR6 = y (elsewhere 0)		
Allowed: 1:GPR3=1 ∧ 1:GPR4=0		

Local reordering of stores and loads to different locations Test ppc3.1 (analogue of [SSZN$^+$09, iwp2.3.a/amd4]) in §2.9 shows that non-overlapping memory stores and loads can be reordered.

Write buffering In test ppc4 (an analogue of [SSZN$^+$09, iwp2.4/amd9]) each processor might see the other memory store at the end of their view order, thus admitting the outcome shown.

ppc4	proc:0	proc:1
poi:0	stw GPR1,0,GPR4	stw GPR1,0,GPR5
poi:1	lwz GPR2,0,GPR4	lwz GPR2,0,GPR5
poi:2	lwz GPR3,0,GPR5	lwz GPR3,0,GPR4
Initial state: 0:GPR1= 1; 0:GPR4= 200; 0:GPR5= 100; 1:GPR1= 1; 1:GPR4= 200; 1:GPR5= 100 (elsewhere 0)		
Allowed: 0:GPR3=0 ∧ 1:GPR3=0		

Shadow registers Test ppc.reg (Adir et al. [AAS03, Test 6]) illustrates that the existence of shadow registers is observable. The specified outcome can be obtained only by an execution in which proc:0 executes the `li` and `stw` instructions before the `lwz` and `mr` ones. Observe however

ppc/iwp6	proc:0	proc:1	proc:2	proc:3
poi:0	stw GPR1,0,GPR5	stw GPR1,0,GPR5	lwz GPR3,0,GPR5	lwz GPR3,0,GPR5
poi:1			lwz GPR4,0,GPR5	lwz GPR4,0,GPR5
Initial state: 0:GPR1 = 1; 0:GPR5 = x; 1:GPR1 = 2; 1:GPR5 = x; 2:GPR5 = x; 3:GPR5 = x (elsewhere 0)				
Forbidden: 2:GPR3=1 ∧ 2:GPR4=2 ∧ 3:GPR3=2 ∧ 3:GPR4=1				

Figure 5. Total order on writes to the same location

that the `li` instruction writes to the r1 register, which is read by the `mr` instruction that precedes it.

ppc.reg	proc:0	proc:1
poi:0	lwz GPR1,0,GPR4	lwz GPR3,0,GPR5
poi:1	mr GPR2,GPR1	addi GPR3,GPR3,1
poi:2	li GPR1,1	stw GPR3,0,GPR4
poi:3	stw GPR1,0,GPR5	
Initial state: 0:GPR4= 200; 0:GPR5= 100; 1:GPR4= 200; 1:GPR5= 100 (elsewhere 0)		
Allowed: 0:GPR1=1 ∧ 0:GPR2=2 ∧ 1:GPR3=2 ∧ [100]=1 ∧ [200]=2		

Failure of transitive causality The intuitive property that there is an acyclic transitive causality relation, including all the dependencies of the model, does *not* hold for Power and ARM, in sharp contrast to the x86. For instance, consider the following ARM test (from [ARM08b, §6.4]), in which proc:0 signals to proc:1 by storing the sentinel 1 (in R0) in the shared location [R2], and "then" proc:1 signals to proc:2 by, similarly, storing the sentinel 1 in the shared location [R3].

proc:0	proc:1	proc:2
STR R0, [R2]	b1: LDR R12, [R2]	c1: LDR R12, [R3]
	CMP R12, #1	CMP R12, #1
	BNE b1	BNE c1
	STR R0, [R3]	AND R12, R12, #0
		LDR R0, [R2,R12]

In Power and ARM, the final result 2:R0=0 is permissible. This implies that proc:2 has seen the store from proc:1 before the store from proc:0. This is the case despite the fact that proc:1's store can only happen after proc:1 has observed the store from proc:0. In contrast, in an x86 analogue, 2:R0=1 would be guaranteed [SSZN$^+$09, iwp2.5/amd8]. In these weaker architectures, to guarantee such a result (ensuring that proc:1 and proc:2 have the same view orders regarding the stores) the programmer must insert a barrier just before proc:1's store into [R3].

5. Testing

We tested the Power examples in §4, and others, on two machines: a two-cpu PowerPC G5, and an eight-core POWER5. We used our litmus tool [SSZN$^+$09]: the test, written in assembly language, is encapsulated in a C skeleton; the tool spawns the threads that compose the test, taking care to start them close to simultaneously, and checks if the final state it obtained is consistent with the constraints we specified; each test is iterated many thousands of times.

In all cases, the results observed were consistent with the Allowed or Forbidden behaviour of the test. We observed all the non-sequentially-consistent behaviour relating to the testing of address dependencies (ppc1, ppc3.1), and we found a witness exhibiting intra-processor forwarding (ppc4). The results for ppc6 were consistent with the coherence axiom of the model, and the results for a `sync` test were consistent with the cumulative ordering of `sync`, both as expected. We did not observe a witness exhibiting the existence of register duplicates (ppc.reg). It may be that we did not repeat this test enough times to exhibit this behaviour, or that the processors we used to test it do not make use of this feature, even if it is allowed by the architecture.

We are developing a single `memevents` tool for exploring the consequences of our semantics, with (hand-written) executable versions of our x86, Power, and ARM memory models, using the OCaml module system to factor out the architecture-dependent aspects. For x86 this is complete for the relevant tests [SSZN$^+$09]. For Power and ARM, it is work in progress: of the tests in the previous section, currently ppc1 and ppc3.1 can be executed by `memevents`, which analyses all possible executions; they have the behaviour specified. Litmus tests for Power and ARM tend to have rather more events than those for x86, so further engineering is required to support the other tests.

6. Related Work

The most closely related work is that of Adir, Attiya and Shurek [AAS03], who define a PowerPC memory model.

This model uses *operations* instead of events: an operation corresponds more directly to an instruction, as a pair (In, Out), where In is the set of input assignments and Out the set of output assignments to distinct resources (that is, pairs (x, v) where x is a resource, or location in our terminology, and v a value). The preserved program order and allowable view orders are given in terms of operations, which imposes a certain atomicity on the events launched by a given instruction.

They include foreign read events from a shared resource in the view order of a given processor. This corresponds to the internal behaviour of some cache protocols, but seems not to be necessary for the assembly-language-programmer visible semantics.

They also sketch semantics for reservations: the memory is assumed to be partitioned into reservation granules, although granularity discussions are omitted. The return code of a `stwcx.` is also provided, which corresponds in our setup to setting CR0[EQ] to 0 or 1 according on whether or not the instruction has succeeded. Each plain store, and `lwarx` and `stwcx` instructions take the reservations granules as both inputs and outputs. A `lwarx` will set the reservation to the address from which it loads. If a store writes to a location that has been reserved by another processor, the associated reservation is set to the special symbol $\perp$. This corresponds in our setup to the *read_value_reserved* property. It is unclear whether there might be spurious dependencies arising from accesses to the reservation data.

They provide a description of the `sync` instruction. However, it refers to a non-cumulative `sync`, which is no longer the case since PPC 1.09. Thus, several examples proposed are no longer allowed.

Recent work on memory models for high-level languages, including Java, X10, and C++ [MPA05, AS07, SJMvP07, BA08], is also relevant. There one must deal not just with reordering by the processor, but also arising from compiler optimizations. We hope that having precise descriptions of the underlying architecture will assist such work.

7. Discussion

We have presented a semantics for multiprocessor programs above the Power and ARM architectures.

In this final section we introduce several criteria that such a semantics should satisfy, and discuss the extent to which our semantics does so.

First, it should be *precise*. It may be a loose specification, but it should unambiguously define what is and is not permitted. Ours is expressed in mechanised mathematics, in a well-defined logic, and using a proof assistant (HOL).

Second, it should have good *coverage*. We do not attempt to model the entirety of each architecture (to do so, including all the page-table operations etc., would be a mammoth task). However, we do model a large enough fragment for much low-level concurrent programming, including the memory model for "Normal" (ARM) and "Memory Coherence Required" (Power) memory.

Third, it should be *accurate with respect to the architecture*. This is a subtle point, as the architectures are informal prose, not themselves precisely defined objects — indeed, establishing such precise definitions is the main goal of this paper. Nonetheless, the semantics should be informally *sound* with respect to the architecture, i.e. any behaviour that is admitted by the informal specification should be admitted by the formal specification. The semantics might be looser than the informal specification, but not too much so, otherwise it may fail to capture a property that programmers need. We have taken care to interpret the informal prose specifications as best we can. We have described our model here in its own terms, for clarity and brevity, but in a longer version of this paper we intend to spell out the correspondence between the informal prose and our specification. For example, the Power and ARM specifications speak in terms of actions being "performed" or "observed"; these concepts generally are modelled by our view orders.

Fourth, it should be *accurate with respect to processor implementations*. This can be tested empirically, as we did for x86, and we intend to do so again here. In principle, it can also be established by proof, showing that the semantics is an accurate abstraction of a microarchitectural model (e.g. as in Fox's verification for ARM6 w.r.t ARMv3 [Fox03]). That would be highly desirable, but very challenging: for modern multiprocessors, such a model would be very large, and also proprietary.

Fifth, it should be *strong enough for reasonable programs*. The semantics should constrain the behaviour of the processors enough that reasonable programs can be shown to have their intended behaviour. This applies also to the informal specifications, and there it is almost impossible to assess: one simply cannot determine all the possible behaviours of a non-trivial program, by hand, against those documents. For our precise model it should be possible to prove (for small programs) how they behave, and to test larger programs against a nondeterministic emulator which can exhibit all valid executions. It should also be possible to prove general metatheoretic results, e.g. that programs that are (in some model-specific sense) data-race-free behave sequentially consistently.

Sixth, it should be *accessible*, to low-level programmers, builders of verification tools, language designers and implementors, and hardware architects, as the interface between these four groups. Our axiomatic model is moderately complex, but we believe this reflects the complexity of the intended vendor architecture. It is written in relatively straightforward typed logic, and can be stated in full in only a few pages. Time will tell whether this, augmented by further examples, suffices. Certainly having a precise mathematical specification should make it easier to write self-consistent tutorial documents. We have primarily targeted the first three groups, writing a specification that is relatively free of microarchitectural concepts.

Acknowledgements We thank Nathan Chong for discussions about the ARM, and Luc Maranget for comments on a draft, Paul McKenney and Raul Silvera for comments on the PowerPC, and Doug Lea. We acknowledge the support of EPSRC grants GR/T11715, EP/C510712, and EP/F036345, and ANR grant ANR-06-SETI-010-02.

References

[AAS03] 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.*, 14(5):502–515, 2003.

[AG96] S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, Dec 1996.

[ARM08a] ARM. *ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition)*. 2008. Available from ARM.

[ARM08b] ARM. ARM Barrier Litmus Tests and Cookbook, February 2008.

[AS07] D. Aspinall and J. Sevcik. Formalising Java's data race free guarantee. In *Proc. TPHOLs, LNCS*, 2007.

[BA08] H.-J. Boehm and S.V. Adve. Foundations of the C++ concurrency memory model. *SIGPLAN Not.*, 43(6):68–78, 2008.

[CI08] Nathan Chong and Samin Ishtiaq. Reasoning about the ARM weakly consistent memory model. In *Proc. MSPC*, 2008.

[Coq] The Coq proof assistant. `http://coq.inria.fr/`.

[CSB] F. Corella, J. M. Stone, and C. M. Barton. A formal specification of the PowerPC shared memory architecture. Technical Report RC18638, IBM.

[Fox03] Anthony Fox. Formal specification and verification of ARM6. In *Proc. TPHOLs, Theorem Proving in Higher Order Logics, LNCS 2758*, pages 25–40, 2003.

[HJK06] L. Higham, L. A. Jackson, and J. Kawash. Programmer-centric conditions for itanium memory consistency. In *Proc. ICDCN*, 2006.

[HOL] The HOL 4 system. `http://hol.sourceforge.net/`.

[IBM02] IBM. *Book E - Enhanced PowerPC Architecture*. May 2002.

[Ita] A formal specification of Intel Itanium processor family memory ordering. `http://developer.intel.com/design/itanium/downloads/251429.htm`.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. POPL*, 2006.

[LHF05] Michael Lyons, Bill Hay, and Brad Frey. PowerPC Storage Model and AIX Programming. November 2005.

[LJV97] L.Higham, J.Kawash, and Nathaly Verwaal. Defining

and comparing memory consistency models. In *PDCS*, 1997.

[Luc01] V. M. Luchangco. *Memory consistency models for high-performance distributed computing*. PhD thesis, MIT, 2001.

[MPA05] J. Manson, W. Pugh, and S.V. Adve. The Java memory model. In *Proc. POPL*, 2005.

[MSG08] Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Machine-code verification for multiple architectures: An application of decompilation into logic. In *Proc. FMCAD, Formal Methods in Computer-Aided Design*, 2008. To appear.

[PD95] S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *Proc. SPAA '95*, 1995.

[pow07] *Power ISA Version 2.05*. October 2007.

[SF95] Janice M. Stone and Robert P. Fitzgerald. Storage in the PowerPC. *IEEE Micro*, 1995.

[SJMvP07] V.A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *Proc. PPoPP*, 2007.

[Spa] The SPARC architecture manual, v. 9. `http://developers.sun.com/solaris/articles/sparcv9.pdf`.

[SSZN+09] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86 multiprocessor machine code. In *Proc. POPL 2009*, January 2009. To appear.

[wea08] The semantics of multiprocessor machine code, 2008. `www.cl.cam.ac.uk/users/pes20/weakmemory`.

[YGLS04] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS*, 2004.