Understanding POWER Multiprocessors

Susmit Sarkar[1] Peter Sewell[1] Jade Alglave[2,3] Luc Maranget[3] Derek Williams[4]
[1]University of Cambridge [2]Oxford University [3]INRIA [4]IBM Austin

Abstract

Exploiting today's multiprocessors requires high-performance and correct concurrent systems code (optimising compilers, language runtimes, OS kernels, etc.), which in turn requires a good understanding of the observable processor behaviour that can be relied on. Unfortunately this critical hardware/software interface is not at all clear for several current multiprocessors.

In this paper we characterise the behaviour of IBM POWER multiprocessors, which have a subtle and highly relaxed memory model (ARM multiprocessors have a very similar architecture in this respect). We have conducted extensive experiments on several generations of processors: POWER G5, 5, 6, and 7. Based on these, on published details of the microarchitectures, and on discussions with IBM staff, we give an abstract-machine semantics that abstracts from most of the implementation detail but explains the behaviour of a range of subtle examples. Our semantics is explained in prose but defined in rigorous machine-processed mathematics; we also confirm that it captures the observable processor behaviour, or the architectural intent, for our examples with an executable checker. While not officially sanctioned by the vendor, we believe that this model gives a reasonable basis for reasoning about current POWER multiprocessors.

Our work should bring new clarity to concurrent systems programming for these architectures, and is a necessary precondition for any analysis or verification. It should also inform the design of languages such as C and C++, where the language memory model is constrained by what can be efficiently compiled to such multiprocessors.

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, Languages, Reliability, Standardization, Theory, Verification

Keywords Relaxed Memory Models, Semantics

1. Introduction

Power multiprocessors (including the IBM POWER 5, 6, and 7, and various PowerPC implementations) have for many years had aggressive implementations, providing high performance but exposing a very relaxed memory model, one that requires careful use of dependencies and memory barriers to enforce ordering in concurrent code. A priori, one might expect the behaviour of a multiprocessor to be sufficiently well-defined by the vendor architecture documentation, here the Power ISA v2.06 specification [Pow09]. For the sequential behaviour of instructions, that is very often true. For concurrent code, however, the observable behaviour of Power multiprocessors is extremely subtle, as we shall see, and the guarantees given by the vendor specification are not always clear. We therefore set out to discover the actual processor behaviour and to define a rigorous and usable semantics, as a foundation for future system building and research.

The programmer-observable relaxed-memory behaviour of these multiprocessors emerges as a whole-system property from a complex microarchitecture [SKT+05, LSF+07, KSSF10]. This can change significantly between generations, e.g. from POWER 6 to POWER 7, but includes: cores that perform out-of-order and speculative execution, with many shadow registers; hierarchical store buffering, with some buffering shared between threads of a symmetric multithreading (SMT) core, and with multiple levels of cache; store buffering partitioned by address; and a cache protocol with many cache-line states and a complex interconnection topology, and in which cache-line invalidate messages are buffered. The implementation of coherent memory and of the memory barriers involves many of these, working together. To make a useful model, it is essential to abstract from as much as possible of this complexity, both to make it simple enough to be comprehensible and because the detailed hardware designs are proprietary (the published literature does not describe the microarchitecture in enough detail to confidently predict all the observable behaviour). Of course, the model also has to be *sound*, allowing all behaviour that the hardware actually exhibits, and sufficiently *strong*, capturing any guarantees provided by the hardware that systems programmers rely on. It does not have to be tight: it may be desirable to make a loose specification, permitting some behaviour that current hardware does not exhibit, but which programmers do not rely on the absence of, for simplicity or to admit different implementations in future. The model does not have to correspond in detail to the internal structure of the hardware: we are capturing the external behaviour of reasonable implementations, not the implementations themselves. But it should have a clear abstraction relationship to implementation microarchitecture, so that the model truly *explains* the behaviour of examples.

To develop our model, and to establish confidence that it is sound, we have conducted extensive experiments, running several thousand tests, both hand-written and automatically generated, on several generations of processors, for up to 10^{11} iterations each. We present some simple tests in §2, to introduce the relaxed behaviour allowed by Power

PLDI'11, June 4–8, 2011, San Jose, California, USA.

processors, and some more subtle examples in §6, with representative experimental data in §7. To ensure that our model explains the behaviour of tests in a way that faithfully abstracts from the actual hardware, using appropriate concepts, we depend on extensive discussions with IBM staff. To validate the model against experiment, we built a checker, based on code automatically generated from the mathematical definition, to calculate the allowed outcomes of tests (§8); this confirms that the model gives the correct results for all tests we describe and for a systematically generated family of around 300 others.

Relaxed memory models are typically expressed either in an axiomatic or an operational style. Here we adopt the latter, defining an abstract machine in §3 and §4. We expect that this will be more intuitive than typical axiomatic models, as it has a straightforward notion of global time (in traces of abstract machine transitions), and the abstraction from the actual hardware is more direct. More particularly, to explain some of the examples, it seems to be necessary to model out-of-order and speculative reads explicitly, which is easier to do in an abstract-machine setting. This work is an exercise in making a model that is as simple as possible but no simpler: the model is considerably more complex than some (e.g. for TSO processors such as Sparc and x86), but does capture the processor behaviour or architectural intent for a range of subtle examples. Moreover, while the definition is mathematically rigorous, it can be explained in only a few pages of prose, so it should be accessible to the expert systems programmers (of concurrency libraries, language runtimes, optimising compilers, etc.) who have to be concerned with these issues. We end with discussion of related work (§9) and a brief summary of future directions (§10), returning at last to the vendor architecture.

2. Simple Examples

We begin with an informal introduction to Power multiprocessor behaviour by example, introducing some key concepts but leaving explanation in terms of the model to later.

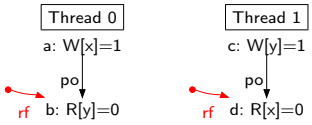
2.1 Relaxed behaviour

In the absence of memory barriers or dependencies, Power multiprocessors exhibit a very relaxed memory model, as shown by their behaviour for the following four classic tests.

SB: Store Buffering Here two threads write to shared-memory locations and then each reads from the other location — an idiom at the heart of Dekker's mutual-exclusion algorithm, for example. In pseudocode:

Thread 0	Thread 1
x=1 r1=y	y=1 r2=x
Initial shared state: x=0 and y=0	
Allowed final state: r1=0 and r2=0	

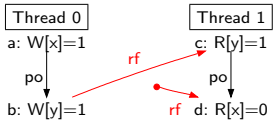
In the specified execution both threads read the value from the initial state (in later examples, this is zero unless otherwise stated). To eliminate any ambiguity about exactly what machine instructions are executed, either from source-language semantics or compilation concerns, we take the definitive version of our examples to be in PowerPC assembly (available online [SSA+11]), rather than pseudocode. The assembly code is not easy to read, however, so here we present examples as diagrams of the memory read and write events involved in the execution specified by the initial and final state constraints. In this example, the pseudocode r1 and r2 represent machine registers, so accesses to those are not memory events; with the final state as specified, the only conceivable execution has two writes, labelled a and c, and two reads, labelled b and d, with values as below. They are related by program order po (later we elide implied po edges), and the fact that the two reads both read from the initial state (0) is indicated by the incoming reads-from (rf) edges (from writes to reads that read from them); the dots indicate the initial-state writes.



Test SB : Allowed

This example illustrates the key relaxation allowed in Sparc or x86 TSO models [Spa92, SSO+10]. The next three show some ways in which Power gives a weaker model.

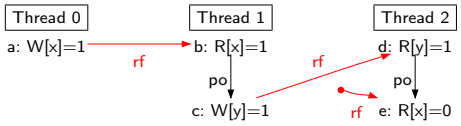
MP: Message passing Here Thread 0 writes data x and then sets a flag y, while Thread 1 reads y from that flag write and then reads x. On Power that read is not guaranteeed to see the Thread 0 write of x; it might instead read from 'before' that write, despite the chain of po and rf edges:



Test MP : Allowed

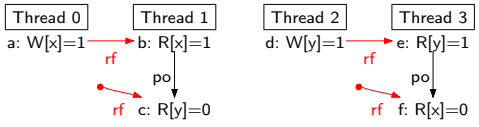
In real code, the read c of y might be in a loop, repeated until the value read is 1. Here, to simplify experimental testing, we do not have a loop but instead consider only executions in which the value read is 1, expressed with a constraint on the final register values in the test source.

WRC: Write-to-Read Causality Here Thread 0 communicates to Thread 1 by writing x=1. Thread 1 reads that, and then later (in program order) sends a message to Thread 2 by writing into y. Having read that write of y at Thread 2, the question is whether a program-order-subsequent read of x at Thread 2 is guaranteed to see the value written by the Thread 0 write, or might read from 'before' that, as shown, again despite the rf and po chain. On Power that is possible.



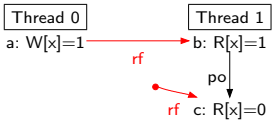
Test WRC : Allowed

IRIW: Independent Reads of Independent Writes Here two threads (0 and 2) write to distinct locations while two others (1 and 3) each read from both locations. In the specified allowed execution, they see the two writes in different orders (Thread 1's first read sees the write to x but the program-order-subsequent read does not see the write of y, whereas Thread 3 sees the write to y but not that to x).

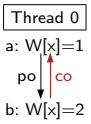


Test IRIW : Allowed

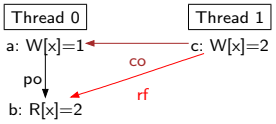
Coherence Despite all the above, one does get a guarantee of *coherence*: in any execution, for each location, there is a single linear order (co) of all writes (by any processor) to that location, which must be respected by all threads. The four cases below illustrate this: a pair of reads by a thread cannot read contrary to the coherence order (CoRR1); the coherence order must respect program order for a pair of writes by a thread (CoWW); a read cannot read from a write that is coherence-hidden by another write program-order-preceding the read (CoWR), and a write cannot coherence-order-precede a write that a program-order-preceding read read from. We can now clarify the 'before' in the MP and WRC discussion above, which was with respect to the coherence order for x.



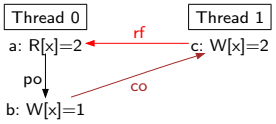
Test CoRR1 : Forbidden



Test CoWW : Forbidden



Test CoWR : Forbidden

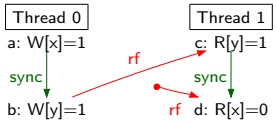


Test CoRW : Forbidden

2.2 Enforcing ordering

The Power ISA provides several ways to enforce stronger ordering. Here we deal with the sync (heavyweight sync, or hwsync) and lwsync (lightweight sync) barrier instructions, and with dependencies and the isync instruction, leaving load-reserve/store-conditional pairs and eieio to future work.

Regaining sequential consistency (SC) using sync If one adds a sync between every program-order pair of instructions (creating tests SB+syncs, MP+syncs, WRC+syncs, and IRIW+syncs), then all the non-SC results above are forbidden, e.g.



Test MP+syncs : Forbidden

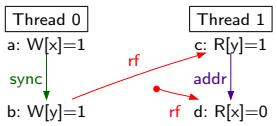
Using dependencies Barriers can incur a significant runtime cost, and in some cases enough ordering is guaranteed simply by the existence of a dependency from a memory read to another memory access. There are various kinds:

- There is an *address dependency* (addr) from a read to a program-order-later memory read or write if there is a data flow path from the read, through registers and arithmetic/logical operations (but not through other memory accesses), to the address of the second read or write.
- There is a *data dependency* (data) from a read to a memory write if there is such a path to the value written. Address and data dependencies behave similarly.
- There is a *control dependency* (ctrl) from a read to a memory write if there is such a dataflow path to the test of a conditional branch that is a program-order-predecessor of the write. We also refer to control dependencies from a read to a read, but ordering of the reads in that case is *not* respected in general.
- There is a *control+isync dependency* (ctrlisync) from a read to another memory read if there is such a dataflow path from the first read to the test of a conditional branch that program-order-precedes an isync instruction before the second read.

Sometimes one can use dependencies that are naturally present in an algorithm, but it can be desirable to introduce one artificially, for its ordering properties, e.g. by XOR'ing a value with itself and adding that to an address calculation.

Dependencies alone are usually not enough. For example, adding dependencies between read/read and read/write pairs, giving tests WRC+data+addr (with a data dependency on Thread 1 and an address dependency on Thread 2), and IRIW+addrs (with address dependencies on Threads 1 and 3), leaves the non-SC behaviour allowed. One cannot add dependencies to SB, as that only has write/read pairs, and one can only add a dependency to the read/read side of MP, leaving the writes unconstrained and the non-SC behaviour still allowed.

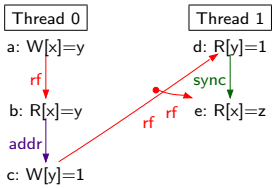
In combination with a barrier, however, dependencies can be very useful. For example, MP+sync+addr is SC:



Test MP+sync+addr : Forbidden

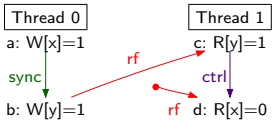
Here the barrier keeps the writes in order, as seen by any thread, and the address dependency keeps the reads in order.

Contrary to what one might expect, the combination of a thread-local reads-from edge and a dependency does *not* guarantee ordering of a write-write pair, as seen by another thread; the two writes can propagate in either order (here [x]=z initially):



Test MP+nondep+sync : Allowed

Control dependencies, observable speculative reads, and isync Recall that control dependencies (without isync) are only respected from reads to writes, not from reads to reads. If one replaces the address dependency in MP+sync+addr by a dataflow path to a conditional branch before the second read (giving the test named MP+sync+ctrl below), that does not ensure that the reads on Thread 1 bind their values in program order.

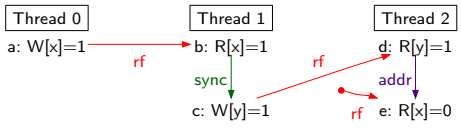


Test MP+sync+ctrl : Allowed

Adding an isync instruction between the branch and the second read (giving test MP+sync+ctrlisync) suffices.

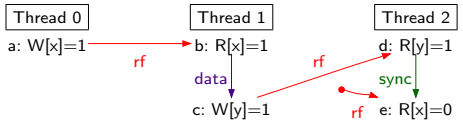
The fact that data/address dependencies to both reads and writes are respected while control dependencies are only respected to writes is important in the design of C++0x low-level atomics [BA08, BOS+11], where release/consume atomics let one take advantage of data dependencies without requiring barriers (and limiting optimisation) to ensure that all source-language control dependencies are respected.

Cumulativity For WRC it suffices to have a sync on Thread 1 with a dependency on Thread 2; the non-SC behaviour is then forbidden:



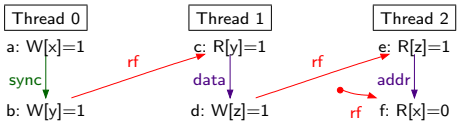
Test WRC+sync+addr : Forbidden

This illustrates what we call *A-cumulativity* of Power barriers: a chain of edges before the barrier that is respected. In this case Thread 1 reads from the Thread 0 write before (in program order) executing a sync, and then Thread 1 writes to another location; any other thread (here 2) is guaranteed to see the Thread 0 write before the Thread 1 write. However, swapping the sync and dependency, e.g. with just an rf and data edge between writes a and c, does not guarantee ordering of those two writes as seen by another thread:



Test WRC+data+sync : Allowed

In contrast to that WRC+data+sync, a chain of reads-from edges and dependencies *after* a sync does ensure that ordering between a write before the sync and a write after the sync is respected, as below. Here the reads e and f of z and x cannot see the writes a and d out of order. We call this a *B-cumulativity* property.

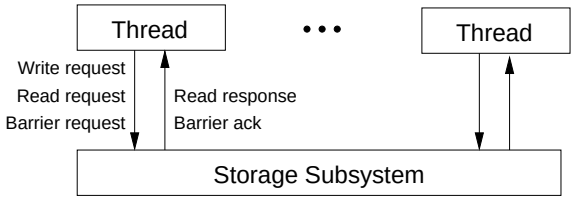


Test ISA2+sync+data+addr : Forbidden

Using lwsync The lwsync barrier is broadly similar to sync, including cumulativity properties, except that does not order store/load pairs and it is cheaper to execute; it suffices to guarantee SC behaviour in MP+lwsyncs (MP with lwsync in each thread), WRC+lwsync+addr (WRC with lwsync on Thread 1 and an address dependency on Thread 2), and ISA2+lwsync+data+addr, while SB+lwsyncs and IRIW+lwsyncs are still allowed. We return later to other differences between sync and lwsync.

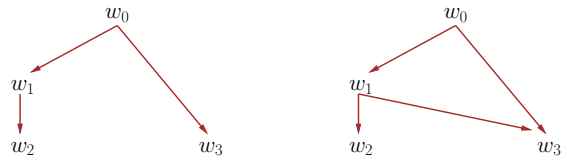
3. The Model Design

We describe the high-level design of our model in this section, giving the details in the next. We build our model as a composition of a set of *(hardware) threads* and a single *storage subsystem*, synchronising on various *messages*:



Read-request/read-response pairs are tightly coupled, while the others are single unidirectional messages. There is no buffering between the two parts.

Coherence-by-fiat Our storage subsystem abstracts completely from the processor implementation store-buffering and cache hierarchy, and from the cache protocol: our model has no explicit memory, either of the system as a whole, or of any cache or store queue (the fact that one can abstract from all these is itself interesting). Instead, we work in terms of the write events that a read can read from. Our storage subsystem maintains, for each address, the current constraint on the coherence order among the writes it has seen to that address, as a strict partial order (transitive but irreflexive). For example, suppose the storage subsystem has seen four writes, w_0, w_1, w_2 and w_3, all to the same address. It might have built up the coherence constraint on the left below, with w_0 known to be before w_1, w_2 and w_3, and w_1 known to be before w_2, but with as-yet-undetermined relationships between w_1 and w_3, and between w_2 and w_3.



The storage subsystem also records the list of writes that it has *propagated* to each thread: those sent in response to read-requests, those done by the thread itself, and those propagated to that thread in the process of propagating a barrier to that thread. These are interleaved with records of barriers propagated to that thread. Note that this is a storage-subsystem-model concept: the writes propagated to a thread have not necessarily been sent to the thread model in a read-response.

Now, given a read request by a thread *tid*, what writes could be sent in response? From the state on the left above, if the writes propagated to thread *tid* are just $[w_1]$, perhaps because *tid* has read from w_1, then:

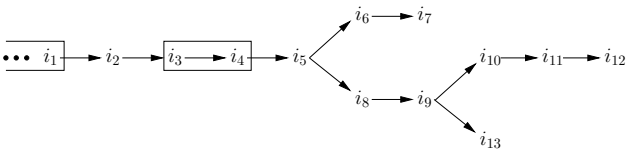
- it cannot be sent w_0, as w_0 is coherence-before the w_1 write that (because it is in the writes-propagated list) it might have read from;
- it could re-read from w_1, leaving the coherence constraint unchanged;
- it could be sent w_2, again leaving the coherence constraint unchanged, in which case w_2 must be appended to the events propagated to *tid*; or
- it could be sent w_3, again appending this to the events propagated to *tid*, which moreover entails committing to w_3 being coherence-after w_1, as in the coherence constraint on the right above. Note that this still leaves the relative order of w_2 and w_3 unconstrained, so another

thread could be sent w_2 then w_3 or (in a different run) the other way around (or indeed just one, or neither).

In the model this behaviour is split up into separate storage-subsystem transitions: there is one rule for making a new coherence commitment between two hitherto-unrelated writes to the same address, one rule for propagating a write to a new thread (which can only fire after sufficient coherence commitments have been made), and one rule for returning a read value to a thread in response to a read-request. The last always returns the most recent write (to the read address) in the list of events propagated to the reading thread, which therefore serves essentially as a per-thread memory (though it records more information than just an array of bytes). We adopt these separate transitions (in what we term a *partial coherence commitment* style) to make it easy to relate model transitions to actual hardware implementation events: coherence commitments correspond to writes flowing through join points in a hierarchical-store-buffer implementation.

Out-of-order and Speculative Execution As we shall see in §6, many of the observable subtleties of Power behaviour arise from the fact that the threads can perform read instructions out-of-order and speculatively, subject to an instruction being restarted if a conflicting write comes in before the instruction is committed, or being aborted if a branch turns out to be mispredicted. However, writes are not sent to the storage subsystem before their instructions are committed, and we do not see observable value speculation. Accordingly, our thread model permits very liberal out-of-order execution, with unbounded speculative execution past as-yet-unresolved branches and past some barriers, while our storage subsystem model need not be concerned with speculation, retry, or aborts. On the other hand, the storage subsystem maintains the current coherence constraint, as above, while the thread model does not need to have access to this; the thread model plays its part in maintaining coherence by issuing requests in reasonable orders, and in aborting/retrying as necessary.

For each thread we have a tree of the *committed* and *in-flight* instruction instances. Newly fetched instructions become in-flight, and later, subject to appropriate preconditions, can be committed. For example, below we show a set of instruction instances $\{i_1, \ldots, i_{13}\}$ with the program-order-successor relation among them. Three of those ($\{i_1, i_3, i_4\}$, boxed) have been committed; the remainder are in-flight.



Instruction instances i_5 and i_9 are branches for which the thread has fetched multiple possible successors; here just two, but for a branch with a computed address it might fetch many possible successors. Note that the committed instances are not necessarily contiguous: here i_3 and i_4 have been committed even though i_2 has not, which can only happen if they are sufficiently independent. When a branch is committed then any un-taken alternative paths are discarded, and instructions that follow (in program order) an uncommitted branch cannot be committed until that branch is, so the tree must be linear before any committed (boxed) instructions.

In implementations, reads are retried when cache-line invalidates are processed. In the model, to abstract from exactly when this happens (and from whatever tracking the core does of which instructions must be retried when it does), we adopt a *late invalidate* semantics, retrying appropriate reads (and their dependencies) when a read or write is committed. For example, consider two reads r_1 and r_2 in program order that have been satisfied from two different writes to the same address, with r_2 satisfied first (out-of-order), from w_1, and r_1 satisfied later from the coherence-later w_2. When r_1 is committed, r_2 must be restarted, otherwise there would be a coherence violation. (This relies on the fact that writes are never provided by the storage subsystem out of coherence order; the thread model does not need to record the coherence order explicitly.)

Dependencies are dealt with entirely by the thread model, in terms of the registers read and written by each instruction instance (the register footprints of instructions are known statically). Memory reads cannot be satisfied until their addresses are determined (though perhaps still subject to change on retry), and memory writes cannot be committed until their addresses and values are fully determined. We do not model register renaming and shadow registers explicitly, but our out-of-order execution model includes their effect, as register reads take values from program-order-preceding register writes.

Barriers (sync and lwsync) and cumulativity-by-fiat The semantics of barriers involves both parts of the model, as follows.

When the storage subsystem receives a barrier request, it records the barrier as *propagated to* its own thread, marking a point in the sequence of writes that have been propagated to that thread. Those writes are the *Group A* writes for this barrier. When all the Group A writes (or some coherence-successors thereof) of a barrier have been propagated to another thread, the storage subsystem can record that fact also, propagating the barrier to that thread (thereby marking a point in the sequence of writes that have been propagated to that thread). A write cannot be propagated to a thread *tid* until all relevant barriers are propagated to *tid*, where the relevant barriers are those that were propagated to the writing thread before the write itself. In turn (by the above), that means that the Group A writes of those barriers (or some coherence successors) must already have been propagated to *tid*. This models the effect of cumulativity while abstracting from the details of how it is implemented.

Moreover, a sync barrier can be *acknowledged* back to the originating thread when all of its Group A writes have been propagated to all threads.

In the thread model, barriers constrain the commit order. For example, no memory load or store instruction can be committed until all previous sync barriers are committed and acknowledged; and sync and lwsync barriers cannot be committed until all previous memory reads and writes have been. Moreover, memory reads cannot be satisfied until previous sync barriers are committed and acknowledged. There are various possible modelling choices here which should not make any observable difference — the above corresponds to a moderately aggressive implementation.

4. The Model in Detail

We now detail the interface between the storage subsystem and thread models, and the states and transitions of each. The transitions are described in §4.3 and §4.5 in terms of

their precondition, their effect on the relevant state, and the messages sent or received. Transitions are atomic, and synchronise as shown in Fig. 1; messages are not buffered. This is a prose description of our mathematical definitions, available on-line [SSA+11].

4.1 The Storage Subsystem/Thread Interface

The storage subsystem and threads exchange messages:

- a *write request* (or *write*) w specifies the writing thread *tid*, unique request id *eid*, address a, and value v.
- a *read request* specifies the originating thread *tid*, request id *eid*, and address a.
- a *read response* specifies the originating thread *tid*, request id *eid*, and a write w (itself specifying the thread tid' that did the write, its id eid', address a, and value v). This is sent when the value read is bound.
- a *barrier request* specifies the originating thread *tid*, request id *eid*, and barrier type b (sync or lwsync).
- a *barrier ack* specifies the originating thread *tid* and request id *eid* (a barrier ack is only sent for sync barriers, after the barrier is propagated to all threads.

4.2 Storage Subsystem States

A storage subsystem state s has the following components.

- $s.threads$ is the set of thread ids that exist in the system.
- $s.writes\_seen$ is the set of all writes that the storage subsystem has seen.
- $s.coherence$ is the current constraint on the coherence order, among the writes that the storage subsystem has seen. It is a binary relation: $s.coherence$ contains the pair (w_1, w_2) if the storage subsystem has committed to write w_1 being before write w_2 in the coherence order. This relation grows over time, with new pairs being added, as the storage subsystem makes additional commitments. For each address, $s.coherence$ is a strict partial order over the write requests seen to that address. It does not relate writes to different addresses, or relate any write that has not been seen by the storage subsystem to any write.
- $s.events\_propagated\_to$ gives, for each thread, a list of:
  1. all writes done by that thread itself,
  2. all writes by other threads that have been propagated to this thread, and
  3. all barriers that have been propagated to this thread.

  We refer to those writes as the writes that have been propagated to that thread. The Group A writes for a barrier are all the writes that have been propagated to the barrier's thread before the barrier is.
- $s.unacknowledged\_sync\_requests$ is the set of sync barrier requests that the storage subsystem has not yet acknowedged.

An initial state for the storage subsystem has the set of thread ids that exist in the system, exactly one write for each memory address, all of which have been propagated to all threads (this ensures that they will be coherence-before any other write to that address), an empty coherence order, and no unacknowledged sync requests.

4.3 Storage Subsystem Transitions

Accept write request A write request by a thread *tid* can always be accepted. Action:

1. add the new write to $s.writes\_seen$, to record the new write as seen by the storage subsystem;
2. append the new write to $s.events\_propagated\_to$ (*tid*), to record the new write as propagated to its own thread; and
3. update $s.coherence$ to note that the new write is coherence-after all writes (to the same address) that have previously propagated to this thread.

Partial coherence commitment The storage subsystem can internally commit to a more constrained coherence order for a particular address, adding an arbitrary edge (between a pair of writes to that address that have been seen already that are not yet related by coherence) to $s.coherence$, together with any edges implied by transitivity, if there is no cycle in the union of the resulting coherence order and the set of all pairs of writes (w_1, w_2), to any address, for which w_1 and w_2 are separated by a barrier in the list of events propagated to the thread of w_2.
Action: Add the new edges to $s.coherence$.

Propagate write to another thread The storage subsystem can propagate a write w (by thread *tid*) that it has seen to another thread tid', if:

1. the write has not yet been propagated to tid';
2. w is coherence-after any write to the same address that has already been propagated to tid'; and
3. all barriers that were propagated to *tid* before w (in $s.events\_propagated\_to$ (*tid*)) have already been propagated to tid'.

Action: append w to $s.events\_propagated\_to$ (tid').

Send a read response to a thread The storage subsystem can accept a read-request by a thread *tid* at any time, and reply with the most recent write w to the same address that has been propagated to *tid*. The request and response are tightly coupled into one atomic transition. Action: send a read-response message containing w to *tid*.

Accept barrier request A barrier request from a thread *tid* can always be accepted. Action:

1. append it to $s.events\_propagated\_to$ (*tid*), to record the barrier as propagated to its own thread (and thereby fix the set of Group A writes for this barrier); and
2. (for sync) add it to $s.unacknowledged\_sync\_requests$.

Propagate barrier to another thread The storage subsystem can propagate a barrier it has seen to another thread if:

1. the barrier has not yet been propagated to that thread; and
2. for each Group A write, that write (or some coherence successor) has already been propagated to that thread

Action: append the barrier to $s.events\_propagated\_to$ (*tid*).

Acknowledge sync barrier A sync barrier b can be acknowledged if it has been propagated to all threads. Action:

1. send a barrier-ack message to the originating thread; and
2. remove b from $s.unacknowledged\_sync\_requests$.

Storage Subsystem Rule	Message(s)	Thread Rule
Accept write request	write request	Commit in-flight instruction
Partial coherence commitment		
Propagate write to another thread		
Send a read response to a thread	read request/read response	Satisfy memory read from storage subsystem
		Satisfy memory read by forwarding an in-flight write
Accept barrier request	barrier request	Commit in-flight instruction
Propagate barrier to another thread		
Acknowledge sync barrier	barrier ack	Accept sync barrier acknowledgement
		Fetch instruction
		Register read from previous register write
		Register read from initial register state
		Internal computation step

Figure 1. Storage Subsystem and Thread Synchronisation

4.4 Thread States

The state t of a single hardware thread consists of:

- its thread id.
- the initial values for all registers, *t.initial_register_state*.
- a set *t.committed_instructions* of committed instruction instances. All their operations have been executed and they are not subject to restart or abort.
- a set *t.in_flight_instructions* of in-flight instruction instances. These have been fetched and some of the associated instruction-semantics micro-operations may have been executed. However, none of the associated writes or barriers have been sent to the storage subsystem, and any in-flight instruction is subject to being aborted (together with all of its dependents).
- a set *t.unacknowledged_syncs* of sync barriers that have not been acknowledged by the storage subsystem.

An initial state for a thread has no committed or in-flight instructions and no unacknowledged sync barriers.

Each instruction instance i consists of a unique id, a representation of the current state of its instruction semantics, the names of its input and output registers, the set of writes that it has read from, the instruction address, the program-order-previous instruction instance id, and any value constraint required to reach this instruction instance from the previous instance. The instruction semantics executes in steps, doing internal computation, register reads and writes, memory reads, and, finally, memory writes or barriers.

4.5 Thread Transitions

Fetch instruction An instruction *inst* can be fetched, following its program-order predecessor *iprev* and from address a, if

1. a is a possible next fetch address for *iprev*; and
2. *inst* is the instruction of the program at a.

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

1. for a non-branch/jump instruction, the successor instruction address;
2. for a jump to a constant address, that address;
3. for a jump to an address which is not yet fully determined (i.e., where there is an uncommitted instruction with a dataflow path to the address), any address; and
4. for a conditional branch, the possible addresses for a jump together with the successor.

Action: construct an initialized instruction instance and add it to the set of in-flight instructions. This is an internal action of the thread, not involving the storage subsystem, as we assume a fixed program rather than modelling fetches with reads; we do not model self-modifying code.

Commit in-flight instruction An in-flight instruction can be committed if:

1. its instruction semantics has no pending reads (memory or register) or internal computation (data or address arithmetic);
2. all instructions with a dataflow dependency to this instruction (instructions with register outputs feeding to this instruction's register inputs) are committed;
3. all program-order-previous branches are committed;
4. if a memory load or store is involved, all program-order-previous instructions which might access its address (i.e., which have an as-yet-undetermined address or which have a determined address which equals that one) are committed;
5. if a memory load or store is involved, or this instruction is a sync, lwsync, or isync, then
   (a) all previous sync, lwsync and isync instructions are committed, and
   (b) there is no unacknowledged sync barrier by this thread;
6. if a sync or lwsync instruction, all previous memory access instructions are committed;
7. if an isync, then all program-order-previous instructions which access memory have their addresses fully determined, where by 'fully determined' we mean that all instructions that are the source of incoming dataflow dependencies to the relevant address are committed and any internal address computation is done.

Action: note that the instruction is now committed, and:

1. if a write instruction, restart any in-flight memory reads (and their dataflow dependents) that have read from the same address, but from a different write (and where the read could not have been by forwarding an in-flight write);
2. if a read instruction, find all in-flight program-order successors that have read from a different write to the same address, or which follow a lwsync barrier program-order after this instruction, and restart them and their dataflow dependents;

3. if this is a branch, abort any untaken speculative paths of execution, i.e., any instruction instances that are not reachable by the branch taken; and
4. send any write requests or barrier requests as required by the instruction semantics.

Accept sync barrier acknowledgement A sync barrier acknowledgement can always be accepted (there will always be a committed sync whose barrier has a matching *eid*). Action: remove the corresponding barrier from $t.unacknowledged\_syncs$.

Satisfy memory read from storage subsystem A pending read request in the instruction semantics of an in-flight instruction can be satisfied by making a read-request and getting a read-response containing a write from the storage subsystem if:

1. the address to read is determined (i.e., any other reads with a dataflow path to the address have been satisfied, though not necessarily committed, and any arithmetic on such a path completed);
2. all program-order-previous syncs are committed and acknowledged; and
3. all program-order-previous isyncs are committed.

Action:

1. update the internal state of the reading instruction; and
2. note that the write has been read from by that instruction.

The remaining transitions are all thread-internal steps.

Satisfy memory read by forwarding an in-flight write directly to reading instruction A pending memory write w from an in-flight (uncommitted) instruction can be forwarded directly to a read of an instruction i if

1. w is an uncommitted write to the same address that is program-order before the read, and there is no program-order-intervening memory write that might be to the same address;
2. all i's program-order-previous syncs are committed and acknowledged; and
3. all i's program-order-previous isyncs are committed.

Action: as in the *satisfy memory read from storage subsystem* rule above.

Register read from previous register write A register read can read from a program-order-previous register write if the latter is the last write to the same register program-order before it. Action: update the internal state of the in-flight reading instruction.

Register read from initial register state A register read can read from the initial register state if there is no write to the same register program-order before it. Action: update the internal state of the in-flight reading instruction.

Internal computation step An in-flight instruction can perform an internal computation step if its semantics has a pending internal transition, e.g. for an arithmetic operation. Action: update the internal state of the in-flight instruction.

4.6 Final states

The final states are those with no transitions. It should be the case that for all such, all instruction instances are committed.

5. Explaining the simple examples

The abstract machine explains the allowed and forbidden behaviour for all the simple tests we saw before. For example, in outline:

MP The Thread 0 write-requests for x and y could be in-order or not, but either way, because they are to different addresses, they can be propagated to Thread 1 in either order. Moreover, even if they are propagated in program order, the Thread 1 read of x can be satisfied first (seeing the initial state), then the read of y, and they could be committed in either order.

MP+sync+ctrl (control dependency) Here the sync keeps the propagation of the writes to Thread 1 in order, but the Thread 1 read of x can be satisfied speculatively, before the conditional branch of the control dependency is resolved and before the program-order-preceding Thread 1 read of y is satisfied; then the two reads can be committed in program order.

MP+sync+ctrlisync (isync) Adding isync between the conditional branch and the Thread 1 read of x prevents that read being satisfied until the isync is committed, which cannot happen until the program-order-previous branch is committed, which cannot happen until the first read is satisfied and committed.

WRC+sync+addr (A-cumulativity) The Thread 0 write-request for a:W[x]=1 must be made, and the write propagated to Thread 1, for b to read 1 from it. Thread 1 then makes a barrier request for its sync, and that is propagated to Thread 1 after a (so the write a is in the Group A set for this barrier), before making the write-request for c:W[y]=1. That write must be propagated to Thread 2 for d to read from it, but before that is possible the sync must be propagated to Thread 2, and before that is possible a must be propagated to Thread 2. Meanwhile, the dependency on Thread 2 means that the address of the read e is not known, and so e cannot be satisfied, until read d has been satisfied (from c). As that cannot be until after a is propagated to Thread 2, read e must read 1 from a, not 0 from the initial state.

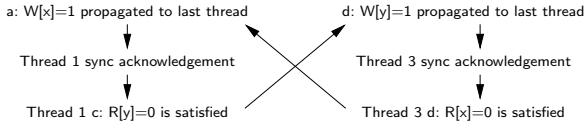
WRC+data+sync Here, in contrast, while the Thread 0/Thread 1 reads-from relationship and the Thread 1 dependency ensure that the write-requests for a:W[x]=1 and c:W[y]=1 are made in that order, and the Thread 2 sync keeps its reads in order, the order that the writes are propagated to Thread 2 is unconstrained.

ISA2 (B-cumulativity) In the ISA2+sync+data+addr B-cumulativity example, the Thread 0 write requests and barrier request must reach the storage subsystem in program order, so Group A for the sync is {a} and the sync is propagated to Thread 0 before the b write request reaches the storage subsystem. For c to read from b, the latter must have been propagated to Thread 1, which requires the sync to be propagated to Thread 1, which in turn requires the Group A write a to have been propagated to Thread 1. Now, the Thread 1 dependency means that d cannot be committed before the read c is satisfied (and committed), and hence d must be after the sync is propagated to Thread 1. Finally, for e to read from d, the latter must have been propagated to Thread 2, for which the sync must be propagated to Thread 2, and hence the Group A write a propagated to Thread 2. The Thread 2 dependency means that f cannot

be satisfied until e is, so it must read from a, not from the initial state.

The same result and reasoning hold for the lwsync variant of this test (note that the reasoning did not involve sync acks or any memory reads program-order-after the sync).

IRIW+syncs Here the two syncs (on Threads 1 and 3) have the corresponding writes (a and d) in their Group A sets, and hence those writes must be propagated to all threads before the respective syncs are acknowledged, which must happen before the program-order-subsequent reads c and f can be be satisfied. But for those to read 0, from coherence-predecessors of a and d, the latter must not have been propagated to all threads (in particular, they must not have been propagated to Threads 3 and 1 respectively). In other words, for this to happen there would have to be a cycle in abstract-machine execution time:

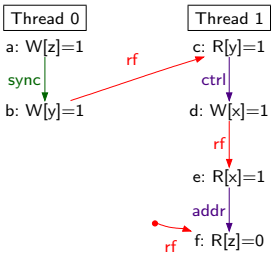


With lwsyncs instead of syncs, the behaviour is allowed, because lwsyncs do not have an analogous acknowledgement when their Group A writes have been propagated to all threads, and memory reads do not wait for previous lwsyncs to reach that point.

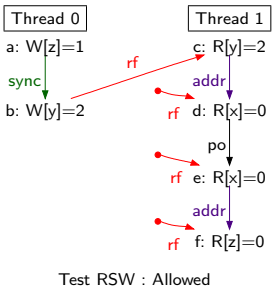
6. Not-so-simple examples

We now discuss some more subtle behaviours, explaining each in terms of our model.

Write forwarding In the PPOCA variant of MP below, 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 (with the *satisfy memory read from storage subsystem* rule) before c does, but in fact in some implementations f can bind out-of-order, as shown — the write d can be forwarded directly to e within the thread, before the write is committed to the storage subsystem. The *satisfy memory read by forwarding an in-flight write* rule models this. Replacing the control dependency with a data dependency (test PPOAA, not shown) removes that possibility, forbidding the given result on current hardware, as far as our experimental results show, and in our model. The current architecture text [Pow09] leaves the PPOAA outcome unspecified, but we anticipate that future versions will explicitly forbid it.



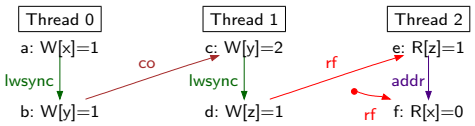
Test PPOCA : Allowed



Test RSW : Allowed

Aggressively out-of-order reads In the reads-from-same-writes (RSW) variant of MP above, the two reads of x, d and e, happen to read from the same write (the initial state). In this case, despite the fact that d and e are reading from the same address, the e/f pair can satisfy their reads out-of-order, before the c/d pair, permitting the outcome shown. The address of e is known, so it can be satisfied early, while the address of d is not known until its address dependency on c is resolved. In contrast, in an execution in which d and e read from different writes to x (test RDW, not shown), with another write to x by another thread), that is forbidden — in the model, the commit of the first read (d) would force a restart of the second (e), together with its dependencies (including f), if e had initially read from a different write to d. In actual implementations the restart might be earlier, when an invalidate is processed, but will have the same observable effect.

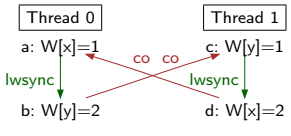
Coherence and lwsync: blw-w-006 This example shows that one cannot assume that the transitive closure of lwsync and coherence edges guarantees ordering of write pairs, which is a challenge for over-simplified models. In our abstract machine, the fact that the storage subsystem commits to b being before c in the coherence order has no effect on the order in which writes a and d propagate to Thread 2. Thread 1 does not read from either Thread 0 write, so they need not be sent to Thread 1, so no cumulativity is in play.



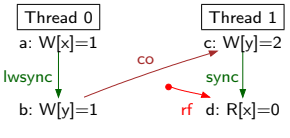
Test blw-w-006 : Allowed

In some implementations, and in the model, replacing both lwsyncs by syncs (bsync-w-006) forbids this behaviour. In the model, it would require a cycle in abstract-machine execution time, from the point at which a propagates to its last thread, to the Thread 0 sync ack, to the b write accept, to c propagating to Thread 0, to c propagating to its last thread, to the Thread 1 sync ack, to the d write accept, to d propagating to Thread 2, to e being satisfied, to f being satisfied, to a propagating to Thread 2, to a propagating to its last thread. The current architecture text again leaves this unspecified, but one would expect that adding sync everywhere (or, in this case, an address dependency between two reads) should regain SC.

Coherence and lwsync: 2+2W and R01 The 2+2W+lwsyncs example below is a case where the interaction of coherence and lwsyncs *does* forbid some behaviour. Without the lwsyncs (2+2W), the given execution is allowed. With them, the writes must be committed in program order, but after one partial coherence commitment (say d before a) is done, the other (b before c) is no longer permitted. (As this test has only writes, it may be helpful to note that the coherence order edges here could be observed either by reading the final state or with additional threads reading x twice and y twice. Testing both versions gives the same result.) This example is a challenge for axiomatic models with a view order per thread, as something is needed to break the symmetry. The given behaviour is also forbidden for the version with syncs (2+2W+syncs).



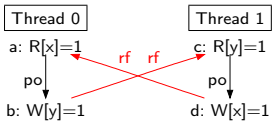
Test 2+2W+lwsyncs : Forbidden



Test R01 : Allowed

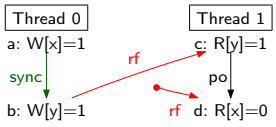
The R01 test on the right above is a related case where we have not observed the given allowed behaviour in practice, but it is not currently forbidden by the architecture, and our model permits it. In the model, the writes can all reach the storage subsystem, the b/c partial coherence commitment be made, c be propagated to Thread 0, the sync be committed and acknowledged, and d be satisfied, all before a and the lwsync propagate to Thread 1.

LB and (no) thin-air reads This LB dual of the SB example is another case where we have not observed the given allowed behaviour in practice, but it is clearly architecturally intended, so programmers should assume that future processors might permit it, and our model does. Adding data or address dependencies (e.g. in LB+datas) should forbid the given behaviour (the data dependency case could involve out-of-thin-air reads), but experimental testing here is vacuous, as LB itself is not observed.



Test LB : Allowed

Register shadowing Adir et al. [AAS03] give another variant of LB (which we call LB+rs, for Register Shadowing), with a dependency on Thread 1 but re-using the same register on Thread 0, to demonstrate the observability of shadow registers. That is also allowed in our model but not observable in our tests — unsurprisingly, given that we have not observed LB itself. However, the following variant of MP does exhibit observable register shadowing: the two uses of r3 on Thread 1 do not prevent the second read being satisfied out-of-order, if the reads are into shadow registers. The reuse of a register is not represented in our diagrams, so for this example we have to give the underlying PowerPC assembly code.



Test MP+sync+rs : Allowed

Thread 0	Thread 1

li r1,1
stw r1,0(r2)
sync
li r3,1
stw r3,0(r4)
	lwz r3,0(r2)
	mr r1,r3
	lwz r3,0(r4)

Initial state: 0:r2=x ∧ 0:r4=y ∧ 1:r2=y ∧ 1:r4=x	
Allowed: 1:r1=1 ∧ 1:r3=0	

7. Experiments on hardware

The use of small *litmus-test* programs for discussing the behaviour of relaxed memory models is well-established, but most previous work (with notable exceptions) does not empirically investigate how tests behave on actual hardware. We use our `litmus` tool [AMSS11] to run tests on machines with various Power processors: Power G5 (aka PowerPC 970MP, based on a POWER4 core), POWER 5, POWER 6, and POWER 7. The tool takes tests in PowerPC assembly and runs them in a test harness designed to stress the processor, to increase the likelihood of interesting results. This is black-box testing, and one cannot make any definite conclusions from the absence of some observation, but our experience is that the tool is rather discriminating, identifying many issues with previous models (and [AMSS10] report the discovery of a processor erratum using it).

Our work is also unusual in the range and number of tests used. For this paper we have established a library based on tests from the literature [Pow09, BA08, AAS03, ARM08], new hand-written tests (e.g. the PPOCA, PPOAA, RSW, RDW, and 2+2W in §6, and many others), and systematic variations of several tests (SB, MP, WRC, IRIW, ISA2, LB, and two others, RWC and WWC) with all possible combinations of barriers or dependencies; we call this the "VAR3" family, of 314 tests. We ran all of these on Power G5, 6, and 7. In addition, we use the `diy` tool [AMSS10] to systematically generate several thousand interesting tests with cycles of edges (dependencies, reads-from, coherence, etc.) of increasing size, and tested some of these. As an important style point, we use tests with constraints on the final values (and hence on the values read) rather than loops, to make them easily testable. We give an excerpt of our experimental results below, to give the flavour; more are available online [SSA+11]. For example, PPOCA was observable on POWER G5 (1.0k/3.1G), not observable on POWER 6, and then observable again on POWER 7 — consistent with the less aggressively out-of-order microarchitecture of POWER 6.

Test	Model	POWER 6		POWER 7	
WRC	Allow	ok	970k / 12G	ok	23M / 93G
WRC+data+addr	Allow	ok	562k / 12G	ok	94k / 93G
WRC+syncs	Forbid	ok	0 / 16G	ok	0 / 110G
WRC+sync+addr	Forbid	ok	0 / 16G	ok	0 / 110G
WRC+lwsync+addr	Forbid	ok	0 / 16G	ok	0 / 110G
WRC+data+sync	Allow	ok	150k / 12G	ok	56k / 94G
PPOCA	Allow	unseen	0 / 39G	ok	62k / 141G
PPOAA	Forbid	ok	0 / 39G	ok	0 / 157G
LB	Allow	unseen	0 / 31G	unseen	0 / 176G

The interplay between manual testing, systematic testing, and discussion with IBM staff has been essential to the development of our model. For example: the PPOCA/PPOAA behaviour was discovered in manual testing, leading us to conjecture that it should be explained by write-forwarding, which was later confirmed by discussion; the blw-w-006 test, found in systematic testing, highlighted difficulties with coherence and lwsync in an earlier model; and the role of coherence and sync acknowledgements in the current implementations arose from discussion.

8. Executing the model

The intricacy of relaxed memory models (and the number of tests we consider) make it essential also to have tool support for exploring the model, to automatically calculate the outcomes that the model permits for a litmus test, and to compare them against those observed in practice. To ease model development, and also to retain confidence in the tool, its kernel should be automatically generated from the model definition, not hand-coded. Our abstract machine is defined in Lem, a new lightweight language for machine-formalised mathematical definitions, of types, functions and inductive relations [OBZNS]. From this we generate HOL4 prover code (and thence an automatically typeset version of the machine) and executable OCaml code, using a finite set library, for the precondition and action of each transition rule. We also formalised a symbolic operational semantics for the tiny fragment of the instruction set needed for our tests. Using those, we build an exhaustive memoised search procedure that finds all possible abstract-machine executions for litmus tests.

This has confirmed that the model has the expected behaviour for the 41 tests we mention by name in this paper, for the rest of the VAR3 family of 314 systematic tests,

and for various other tests. In most cases the model exactly matches the Power7 experimental results, with the exception of a few where it includes the experimental outcomes but is intentionally looser; this applies to 60 tests out of our batch of 333. Specifically: the model allows instructions to commit out of program order, which permits the LB and LB+rs test outcomes (not observed in practice); the model also allows an isync to commit even in the presence of previously uncommitted memory accesses, whereas the specified outcomes of tests such as WRC with an lwsync and isync have not been observed; and the R01 test outcome is not observed. In all these cases the model follows the architectural intent, as confirmed with IBM staff.

Our experimental results also confirm that Power G5 and 6 are strictly stronger than Power 7 (though in different ways): we have not seen any test outcome on those which is not also observable on Power 7. The model is thus also sound for those, to the best of our knowledge.

9. Related Work

There has been extensive previous work on relaxed memory models. We focus on models for the major current processor families that do not have sequentially consistent behaviour: Sparc, x86, Itanium, ARM, and Power. Early work by Collier [Col92] developed models based on empirical testing for the multiprocessors of the day. For Sparc, the vendor documentation has a clear Total Store Ordering (TSO) model [SFC91, Spa92]. It also introduces PSO and RMO models, but these are not used in practice. For x86, the vendor intentions were until recently quite unclear, as was the behaviour of processor implementations. The work by Sarkar, Owens, et al. [SSZN$^+$09, OSS09, SSO$^+$10] suggests that for normal user- or system-code they are also TSO. Their work is in a similar spirit to our own, with a mechanised semantics that is tested against empirical observation. Itanium provides a much weaker model than TSO, but one which is more precisely defined by the vendor than x86 [Int02]; it has also been formalised in TLA [JLM$^+$03] and in higher-order logic [YGLS03].

For Power, there have been several previous models, but none are satisfactory for reasoning about realistic concurrent code. In part this is because the architecture has changed over time: the lwsync barrier has been added, and barriers are now cumulative. Corella, Stone and Barton [CSB93] gave an early axiomatic model for PowerPC, but, as Adir et al. note [AAS03], this model is flawed (it permits the non-SC final state of the MP+syncs example we show in §2). Stone and Fitzgerald later gave a prose description of PowerPC memory order, largely in terms of the microarchitecture of the time [SF95]. Gharachorloo [Gha95] gives a variety of models for different architectures in a general framework, but the model for the PowerPC is described as *"approximate"*; it is apparently based on Corella et al. [CSB93] and on May et al. [MSSW94]. Adve and Gharachorloo [AG96] make clear that PowerPC is very relaxed, but do not discuss the intricacies of dependency-induced ordering, or the more modern barriers. Adir, Attiya, and Shurek give a detailed axiomatic model [AAS03], in terms of a view order for each thread. The model was *"developed through an iterative process of successive refinements, numerous discussions with the PowerPC architects, and analysis of examples and counterexamples"*, and its consequences for a number of litmus tests (some of which we use here) are described in detail. These facts inspire some confidence, but it is not easy to understand the force of the axioms, and it describes *non-cumulative* barriers, following the pre-PPC 1.09 PowerPC architecture; current processors appear to be quite different. More recently, Chong and Ishtiaq give a preliminary model for ARM [CI08], which has a very similar architected memory model to Power. In our initial work in this area [AFI$^+$09], we gave an axiomatic model based on a reading of the Power ISA 2.05 and ARM ARM specifications, with experimental results for a few tests (described as work in progress); this seems to be correct for some aspects but to give an unusably weak semantics to barriers.

More recently, we gave a rather different axiomatic model [AMSS10], further developed in Alglave's thesis [Alg10] as an instance of a general framework; it models the non-multi-copy-atomic nature of Power (with examples such as IRIW+addrs correctly allowed) in a simple global-time setting. The axiomatic model is sound with respect to our experimental tests, and on that basis can be used for reasoning, but it is weaker than the observed behaviour or architectural intent for some important examples. Moreover, it was based principally on black-box testing and its relationship to the actual processor implementations is less clear than that for the operational model we present here, which is more firmly grounded on microarchitectural and architectural discussion. In more detail, the axiomatic model is weaker than one might want for lwsync and for cumulativity: it allows MP+lwsync+addr and ISA2+sync+data+addr, which are not observed and which are intended to be architecturally forbidden. It also forbids R01, which is not observed but architecturally intended to be allowed, and which is allowed by the model given here. The two models are thus incomparable.

We mention also Lea's *JSR-133 Cookbook for Compiler Writers* [Lea], which gives informal (and approximate) models for several multiprocessors, and which highlights the need for clear models.

10. Conclusion

To summarise our contribution, we have characterised the actual behaviour of Power multiprocessors, by example and by giving a semantic model. Our examples include new tests illustrating several previously undescribed phenomena, together with variations of classic tests and a large suite of automatically generated tests; we have experimentally investigated their behaviour on a range of processors. Our model is: *rigorous* (in machine-typechecked mathematics); *experimentally validated*; *accessible* (in an abstract machine style, and detailed here in a few pages of prose); *usable* (as witnessed by the explanations of examples); *supported by a tool*, for calculating the possible outcomes of tests; and sufficient to *explain* the subtle behaviour exposed by our examples and testing. It is a new abstraction, maintaining coherence and cumulativity properties by fiat but modelling out-of-order and speculative execution explicitly.

The model should provide a good intuition for developers of concurrent systems code for Power multiprocessors, e.g. of concurrency libraries, language runtimes, OS kernels, and optimising compilers. Moreover, as the ARM architecture memory model is very similar, it may well be applicable (with minor adaptation) to ARM.

The model also opens up many directions for future research in verification theory and tools. For example, it is now possible to state results about the correct compilation of the C++0x concurrency primitives to Power processors, and to consider barrier- and dependency-aware optimisations in that context. We have focussed here primarily on

the actual behaviour of implementations, but there is also work required to identify the guarantees that programmers actually rely on, which may be somewhat weaker — some of our more exotic examples are not natural use-cases, to the best of our knowledge. There is also future work required to broaden the scope of the model, which here covers only cacheable memory without mixed-size accesses.

We described our model principally in its own terms and in terms of the observed behaviour, without going into details of the relationship between the model and the underlying microarchitecture, or with the vendor architecture specification [Pow09]; this remains future work. A central notion of the memory model text in the latter is that of when a memory read or write by one thread is *performed* with respect to another, which has a hypothetical (or subjunctive) definition, e.g. for loads: *"A load by a processor P1 is performed with respect to a processor P2 when the value to be returned can no longer be changed by a store by P2"*, where that P2 store may not even be present in the program under consideration (again, ARM is similar). This definition made perfect sense in the original white-box setting [DSB86], where the internal structure of the system was known and one can imagine the hypothetical store by P2 appearing at some internal interface, but in the black-box setting of a commercial multiprocessor, it is hard or impossible to make it precise, especially with examples such as PPOCA. Our abstract-machine model may provide a stepping stone towards improved architectural definitions, perhaps via new axiomatic characterisations.

Acknowledgements We acknowledge funding from EPSRC grants EP/F036345, EP/H005633, and EP/H027351, from ANR project ParSec (ANR-06-SETIN-010), and from INRIA associated team MM.

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.

[AFI+09] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proc. DAMP 2009*, January 2009.

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

[Alg10] Jade Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris 7 – Denis Diderot, November 2010.

[AMSS10] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proc. CAV*, 2010.

[AMSS11] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *Proc. TACAS*, 2011.

[ARM08] ARM. ARM Barrier Litmus Tests and Cookbook, October 2008. PRD03-GENC-007826 2.0.

[BA08] H.-J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, 2008.

[BOS+11] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proc. POPL*, 2011.

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

[Col92] W.W. Collier. *Reasoning about parallel architectures*. Prentice-Hall, Inc., 1992.

[CSB93] F. Corella, J. M. Stone, and C. M. Barton. A formal specification of the PowerPC shared memory architecture. Technical Report RC18638, IBM, 1993.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *ISCA*, 1986.

[Gha95] K. Gharachorloo. Memory consistency models for shared-memory multiprocessors. *WRL Research Report*, 95(9), 1995.

[Int02] Intel. A formal specification of Intel Itanium processor family memory ordering, 2002. `developer.intel.com/design/itanium/downloads/251429.htm`.

[JLM+03] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking cache-coherence protocols with TLA+. *Form. Methods Syst. Des.*, 22:125–131, March 2003.

[KSSF10] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7: IBM's next-generation server processor. *IEEE Micro*, 30:7–15, March 2010.

[Lea] D. Lea. The JSR-133 cookbook for compiler writers. `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`.

[LSF+07] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007.

[MSSW94] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC architecture: a specification for a new family of RISC processors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[OBZNS] S. Owens, P. Böhm, F. Zappa Nardelli, and P. Sewell. Lightweight tools for heavyweight semantics. Submitted for publication `http://www.cl.cam.ac.uk/~so294/lem/`.

[OSS09] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proc. TPHOLs*, pages 391–407, 2009.

[Pow09] *Power ISA™ Version 2.06*. IBM, 2009.

[SF95] J. M. Stone and R. P. Fitzgerald. Storage in the PowerPC. *IEEE Micro*, 15:50–58, April 1995.

[SFC91] P. S. Sindhu, J.-M. Frailong, and M. Cekleov. Formal specification of memory models. In *Scalable Shared Memory Multiprocessors*, pages 25–42. Kluwer, 1991.

[SKT+05] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4-5):505–522, 2005.

[Spa92] *The SPARC Architecture Manual, V. 8*. SPARC International, Inc., 1992. Revision SAV080SI9308. `http://www.sparc.org/standards/V8.pdf`.

[SSA+11] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. `www.cl.cam.ac.uk/users/pes20/ppc-supplemental`, 2011.

[SSO+10] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

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

[YGLS03] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In *Proc. CHARME, LNCS 2860*, 2003.