An Axiomatic Memory Model for POWER Multiprocessors

Sela Mador-Haim[1] Luc Maranget[2] Susmit Sarkar[3]
Kayvan Memarian[3] Jade Alglave[4] Scott Owens[3]
Rajeev Alur[1] Milo M.K. Martin[1] Peter Sewell[3] Derek Williams[5]

[1]University of Pennsylvania [2]INRIA Rocquencourt-Paris
[3]University of Cambridge [4]University of Oxford [5]IBM Austin

Abstract. The growing complexity of hardware optimizations employed by multiprocessors leads to subtle distinctions among allowed and disallowed behaviors, posing challenges in specifying their memory models formally and accurately, and in understanding and analyzing the behavior of concurrent software. This complexity is particularly evident in the IBM® Power Architecture®, for which a faithful specification was published only in 2011 using an operational style. In this paper we present an equivalent axiomatic specification, which is more abstract and concise. Although not officially sanctioned by the vendor, our results indicate that this axiomatic specification provides a reasonable basis for reasoning about current IBM® POWER® multiprocessors. We establish the equivalence of the axiomatic and operational specifications using both manual proof and extensive testing. To demonstrate that the constraint-based style of axiomatic specification is more amenable to computer-aided verification, we develop a SAT-based tool for evaluating possible outcomes of multi-threaded test programs, and we show that this tool is significantly more efficient than a tool based on an operational specification.

1 Introduction

Modern multiprocessors employ aggressive hardware optimizations to provide high performance and reduce energy consumption, which leads to subtle distinctions between the allowed and disallowed observable behaviors of multithreaded software. Reliable development and verification of multithreaded software (including system libraries and optimizing compilers) and multicore hardware systems requires understanding these subtle distinctions, which in turn demands accurate and formal models.

The IBM® Power Architecture®, which has highly relaxed and complex memory behavior, has proved to be particularly challenging in this respect. For example, IBM® POWER® is non-store-atomic, allowing two writes to different locations to be observed in different orders by different threads; these order variations are constrained by coherence, various dependencies among instructions, and several barrier instructions, which interact with each other in an intricate way. The ARM architecture memory ordering is broadly similar.

Several previous attempts to define POWER memory consistency models [CSB93,SF95,Gha95,AAS03,AFI$^+$09,AMSS10] did not capture these subtleties correctly. A faithful specification for the current Power Architecture was published only in 2011 [SSA$^+$11] using an operational style: a non-deterministic abstract machine with explicit out-of-order and speculative execution and an abstract coherence-by-fiat storage subsystem. This specification was validated both through extensive discussions with IBM staff and comprehensive testing of the hardware. The operational specification has recently been extended to support the POWER load-reserve/store-conditional instructions [SMO$^+$12].

This paper presents an alternative memory model specification for POWER using an axiomatic style, which is significantly more abstract and concise than the previously published operational model and therefore better suited for some formal analysis tools. One of the main challenges in specifying an axiomatic model for POWER is identifying the right level of abstraction. Our goal was to define a specification that is detailed enough to express POWER's complexity, capturing the distinctions between allowed and disallowed behaviors, yet abstract enough to be concise and to enable understanding and analysis.

Our approach splits instruction instances into multiple abstract events and defines a happens-before relation between these events using a set of constraints. The specification presented here handles memory loads and stores; address, data, and control dependencies; the isync instruction, and the lightweight and heavyweight memory barrier instructions lwsync and sync. The specification does not include load-reserve and store-conditional instructions, mixed-size accesses, or the eieio memory barrier.

We show that this specification is equivalent to the existing POWER operational specification (permitting the same set of allowed behaviors for any concurrent POWER program) in two ways. First, we perform extensive testing by checking that the models give the same allowed behaviors for a large suite of tests; this uses tools derived automatically from the definitive mathematical statements of the two specifications, expressed in Lem [OBZNS11]. We also check that the axiomatic specification is consistent with the experimentally observed behavior of current IBM® POWER6®/ IBM® POWER7® hardware. We then provide a manual proof of the equivalence of the operational and axiomatic specifications, using executable mappings between abstract machine traces and axiomatic candidate executions, both defined in Lem. We have checked their correctness empirically on a small number of tests, which was useful in developing the proof.

Finally, we demonstrate that this abstract constraint-based specification is useful for computer-aided verification. The testing described above shows that it can be used to determine the outcomes of multi-threaded test programs more efficiently than the operational model. This efficiency enables a tool to calculate the allowed outcomes of 915 tests for which the current implementation of the operational tool [SSA$^+$11] does not terminate in reasonable time and space. Further, the axiomatic specification lends itself to a SAT/SMT-solving approach:

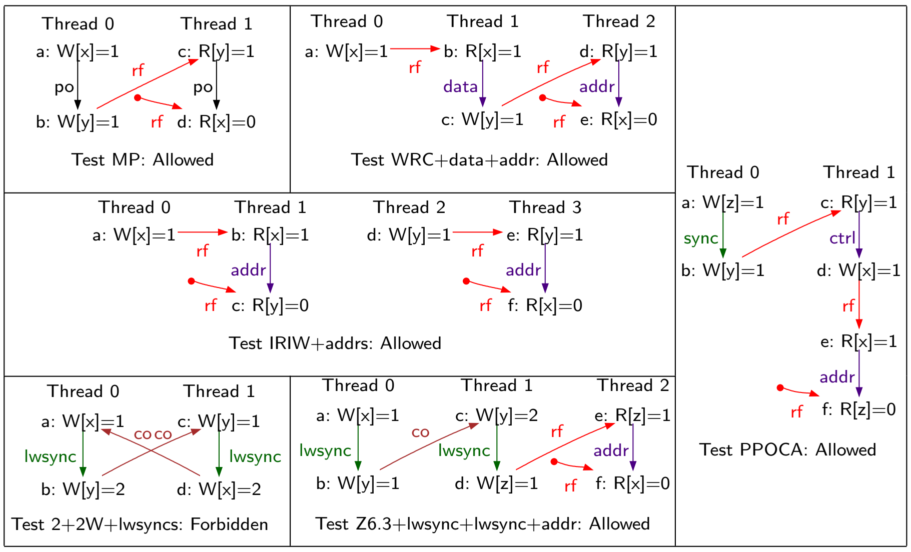


Fig. 1: Examples illustrating the POWER memory model

a hand-coded translation of the axiomatic model using minisat [ES05] reduces execution time for the full test suite radically, from 82 CPU-days to 3 hours.

The full definitions of our specifications, test suite, test results, and proof are available in on-line supplementary material [Sup].

2 Background: The POWER Memory Model

This section highlights some of the subtleties of the POWER memory model, and then overviews its abstract-machine semantics, referring to Sarkar et al. [SSA+11,SMO+12] for a complete description.

2.1 Subtleties of the POWER Memory Model

Figure 1 shows example candidate executions for several concurrent litmus tests, MP, WRC+data+addrs, etc. (each test is defined by its assembly source code and its initial and final register and memory state, which select a particular execution). For each, the diagram shows a graph with nodes for memory reads or writes, each with a label (a, b, ...), location (x, y, ...), and value (0, 1, ...). The edges indicate program order (po), data dependencies from a read to a write whose value was calculated based on the value read (data), address dependencies from a read to a read or write whose address was calculated based on the value read (addr), control dependencies from a read to instructions following a conditional branch whose condition involved the value read (ctrl), and lwsync and sync barriers. The *reads-from* edges (rf) go from a write (or a dot indicating the initial state) to any read that reads from this write or initial value. *Coherence* (co) edges give, for each location, a total order over the writes to that location. Register-only and branch instructions and are elided.

Relaxed memory behavior in POWER arises both from out-of-order and speculative execution within a hardware thread and from the ways in which writes and barriers can be propagated from their originating thread to other threads. For writes and reads to different addresses, absent any barriers or dependencies, the behavior is unconstrained. The message-passing MP example illustrates this behavior: the writes a and b might commit in either order, then propagate to Thread 1 in either order, and reads c and d can be satisfied in either order. Any of these effects can give rise to the given execution. To prevent them, a programmer could add an lwsync or sync barrier between the writes (making their commit order and their propagation order respect program order) and either add a barrier between the reads or make the second read address-dependent on the first (perhaps using the result of the first read xor'd with itself to form the address of the second read, introducing an artificial dependency) thus ensuring that it cannot be satisfied until the first read is satisfied. For example, test MP+lwsync+addr (a variation of MP with a lwsync edge in one thread and an addr edge in the other) is forbidden. A control dependency alone does not prevent reads being satisfied speculatively; the analogous MP+lwsync+ctrl is allowed. But adding an isync instruction after a control dependency does: MP+lwsync+ctrlisync is forbidden.

Dependencies have mostly local effects, as shown by the WRC+data+addr variant of MP, where the facts that b reads from a, and that c is dependent on b, are not sufficient to enforce ordering of a and c as far as Thread 2 is concerned. To enforce that ordering, one has to replace the data dependency by an lwsync or sync barrier. This example relies on the so-called *cumulative* property of the barriers, which orders all writes that have propagated to the thread of the barrier before all writes that occur later in program order, as far as any other thread is concerned.

The independent-reads-of-independent-writes IRIW+addrs example shows that writes to different addresses can be propagated to different threads (here Threads 1 and 3) in different orders; POWER is not *store-atomic*, and thread-local reorderings cannot explain all its behaviors. Inserting sync instructions between the load instructions on Thread 1 and Thread 3 will rule out this behavior. Merely adding lwsyncs does not suffice.

Returning to thread-local reordering, the PPOCA variant of MP shows a subtlety: writes are not performed speculatively as far as other threads are concerned, but here d, e, and f *can* be locally performed speculatively, before c has been satisfied, making this execution allowed.

The two final examples illustrate the interplay between the coherence order and barriers. In test Z6.3+lwsync+lwsync+addr (blw-w-006 in [SSA$^+$11]), even though c is coherence-ordered after b (and so cannot be seen before b by any thread), the lwsync on Thread 1 does not force a to propagate to Thread 2 before d is (in the terminology of the architecture, the coherence edge does not bring b into the *Group A* of the Thread 1 lwsync). This test outcome is therefore allowed. On the other hand, some combinations of barriers and coherence orders are forbidden. In 2+2W+lwsyncs, for example, there is a cycle among the writes of coherence and lwsync edges; such an execution is forbidden.

2.2 The Operational Specification

The operational specification of Sarkar et al. [SSA$^+$11,SMO$^+$12] accounts for all these behaviors (and further subtleties that we do not describe here) with an abstract machine consisting of a set of threads composed with a storage subsystem, communicating by exchanging messages for write requests, read requests, read responses, barrier requests, and barrier acknowledgments (for sync). Threads are modeled with explicit out-of-order and speculative execution: the state of each thread consists of a tree of in-flight and committed instructions with information about the state of each instruction (read values, register values etc.) and a set of unacknowledged syncs. The thread model can perform various types of transitions. Roughly speaking (without detailing all the transition preconditions), a thread can:

– Fetch an instruction, including speculative fetches past a branch.
– Satisfy a read by reading values from the storage subsystem or by forwarding a value from an in-flight write. Reads can be performed speculatively, out-of-order, and (before they are committed) can be restarted if necessary.
– Perform an internal computation and write registers.
– Commit an instruction (sending write and barrier requests to the storage subsystem).

The state of the storage subsystem consists of a set of (1) writes that have been committed by a thread, (2) for each thread, a list of the writes and barriers propagated to that thread, (3) the current constraint on the coherence graph as a partial order between writes to the same location, with an identified linear prefix for each location of those that have *reached coherence point*, and (4) a set of unacknowledged syncs. The storage subsystem can:

– Accept a barrier or write request and update its state accordingly.
– Respond to a read request.
– Perform a partial coherence commit, non-deterministically adding to the coherence graph an edge between two as-yet-unrelated writes to the same location.
– Mark that a write reached a coherence point, an internal transition after which the coherence predecessors of the write are linearly ordered and fixed.
– Propagate a write or a barrier to a thread, if all the writes and barriers that are required to propagate before it have been propagated to this thread. A write can only be propagated if it is coherence-after all writes that were propagated to a thread, but the abstract machine does not require all writes that are coherence-before it have been propagated, thus it allows some writes, which might never be propagated to some of the threads, to be skipped.
– Send a sync acknowledgment to the issuing thread, when that sync has been propagated to all other threads.

3 The Axiomatic Specification

This section introduces our new specification of the POWER memory model in an axiomatic style. We begin by defining the semantics of a multithreaded program as a set of axiomatic candidate executions. We then give an overview of the axiomatic specification, which defines whether a given axiomatic candidate execution is consistent with the model, show how the examples in Section 2 are explained using this model, and finally provide the formal specification of the model.

3.1 Axiomatic Candidate Executions

We adopt a two-step semantics, as is usual in axiomatic memory models, that largely separates the instruction execution semantics from the memory model semantics by handling each individually.

We begin with a multithreaded POWER program, or litmus test, in which each thread consists of a sequence of *instructions*. Such a program may non-deterministically display many different behaviors. For example: reads may read from different writes, conditional branches may be taken or not taken, and different threads may write to the same address in a different order. During the execution of a program, any given static instruction may be iterated multiple times (for example, due to looping). We refer to such an instance of the dynamic execution of an instruction as an *instruction instance*.

To account for the differing ways a given litmus test can execute, we define the semantics of a multithreaded program as set of *axiomatic candidate executions*. Informally, an axiomatic candidate execution consists of (1) a set of *axiomatic instruction instances*, which are instruction instances annotated with additional information as described below, and (2) a set of relations among these axiomatic instruction instances (in what follows, we will refer to instruction instances or axiomatic instruction instances for load and store instructions as *reads* and *writes*, respectively). An axiomatic candidate execution represents a conceivable execution of the program and accounts for the effects of a choice of branch direction for each branch instruction in the program, a possible coherence order choice, and a reads-from mapping showing which write a given read reads from.

An *axiomatic instruction instance* is an instruction instance of the program annotated with a thread id and some additional information based on instruction type. Axiomatic instruction instances are defined only for reads, writes, memory barriers (sync, lwsync, isync), and branches. Reads are annotated with the concrete value read from memory, while writes are annotated with the value written to memory. Barriers and branches have no additional information. Other instructions may affect dependency relationships in the axiomatic candidate execution, but are otherwise ignored by the model.

An axiomatic candidate execution consists of a set of axiomatic instruction instances, and the following relations between those axiomatic instruction instances:

– A *program order* relation *po*, providing a total order between axiomatic instruction instances in each thread.
– A *reads-from* relation *rf*, relating writes to reads to the same address.
– A *coherence* relation *co*, providing, for each address, a strict total order between all writes to that address.
– A *data dependency* relation from reads to those writes whose value depends on the value read.
– An *address dependency* relation from reads to those reads or writes whose address depends on the value read.
– A *control dependency* relation from each read to all writes following a conditional branch that depends on the value read.

For each candidate execution of a given program, the following conditions must hold: (1) for each thread, the sequence of instruction instances ordered by *po* agrees with the local thread semantics of that program, when running alone with the same read values; (2) for each read and write related by *rf*, the read reads the value written by the write; and (3) if a read is not associated with any write in *rf*, it reads the initial value.

As an example, consider the test MP+lwsync+ctrl, a variation of MP with an lwsync and a control dependency. The POWER program for this test is listed below:

```
    Thread 0              Thread 1
(a) li r1,1           (f) lwz r1,0(r2)
(b) stw r1,0(r2)      (g) cmpw r1,r1
(c) lwsync            (h) beq  LC00
(d) li r3,1           LC00:
(e) stw r3,0(r4)      (i) lwz r3,0(r4)
```

Instruction instances are not defined for register-only instructions. Therefore, there are no instances of instructions `a`, `d` and `g` in this example. The conditional branch `h` in the program may be either taken or not taken, but in this case it jumps to `i`, so in both cases the axiomatic candidate execution contains a single instance for each of the instructions: {`b,c,e,f,h,i`}. The program order *po* in all candidate executions of this program is the transitive closure of the set {`(b,c),(c,e),(f,h),(h,i)`} and the control dependency is {`(f,i)`}. The coherence order *co* is empty in this example because each write writes to a different address. Each of the two reads in Thread 1 may either read from the matching write in Thread 0 or from the initial value. Hence, there are four possible *rf* relations, and four axiomatic candidate executions.

3.2 Overview of the Specification

The axiomatic specification defines whether a given axiomatic candidate execution is *consistent* or not. This section provides an overview of our axiomatic POWER memory specification (formally described in Section 3.4).

For each axiomatic candidate execution, we construct a set of *events* that are associated with the axiomatic instruction instances, together with several relations over those events. These events and relations (as described below) capture the subtleties of the POWER memory model, including speculative out-of-order execution and non-atomic stores. The events and relations determine whether an axiomatic candidate execution is *consistent*. In more detail:

Uniprocessor correctness condition. The relations *rf* and *co* must not violate uniprocessor execution order, in the following sense: a read is allowed to read a local write only if that write precedes the read in program order, that write is the most recent (w.r.t. program order) write to that address in that thread, and there is no program-order-intervening read that reads from a different write (from another thread). Based on *rf* and *co*, we define *fr* as the relation from any read to all writes which are coherence-after the write that the read reads from. We define the communication relation *comm* as the union of *rf*, *fr* and *co*. The uniprocessor condition requires that the transitive closure of *comm* does not contain any edge which goes against program order.

Local reordering. The effects of out-of-order and speculative execution in POWER are observable, as shown by the MP variations in Fig. 1 (including PPOCA). Reads can be satisfied speculatively and speculative writes can be forwarded to local reads, although not to other threads. The specification captures this by defining *satisfy* read events, *initiate* write events, and *commit* events for both reads and writes: a read is satisfied when it binds its value, and committed when it cannot be restarted and that value is fixed; a write is initiated when its address and value can be (perhaps speculatively) calculated and it can be propagated to thread-local reads and committed when it can propagate to other threads.

Non-atomic stores. Writes in POWER need not be propagated to all other processors in an atomic fashion, as illustrated by WRC+data+addr (the write to x propagates to Thread 1 before propagating to Thread 2). As in the operational model (and previous axiomatic models [Int02,YGLS03]) to capture this behavior, we split each write into multiple *propagation* events. In our model each thread other than its own has a propagation event, whereas in the operational model some write propagations can be superseded by coherence-later propagations. A write propagating to a thread makes it eligible to be read by that thread.

Barriers and non-atomic stores. The semantics of the sync and lwsync barriers in POWER are quite subtle. As seen in WRC+lwsync+addr, lwsync has a cumulative semantics, but adding lwsync between every two instructions does not restore sequential consistency, as shown by IRIW+lwsyncs. As in the operational model, we capture this behavior by splitting barriers into multiple propagation events, analogous to those for writes, with the proper ordering rules for these. A barrier can propagate to a thread when all the writes in the cumulative Group A of the barrier have propagated to that thread.

Barriers and coherence. As shown by Z6.3+lwsync+lwsync+addr, coherence relationships between writes do not necessarily bring them into the cumulative Group A of lwsync barriers (or for that matter of sync barriers). We capture this behavior by allowing writes that are not read by a certain thread to propagate to that thread later than coherence-after writes. This weakened semantics for coherence must be handled with caution, and additional constraints are required to handle certain combinations of barriers and coherence edges, as shown by example 2+2W+lwsyncs (in Fig. 1).

To summarize, the specification uses the notion of events with possibly multiple events corresponding to an axiomatic instruction instance to capture these behaviors. There are four types of events:

1. *Satisfy events.* There is a single read satisfy event $sat(x)$ for each read axiomatic instruction instance x, representing the point at which it takes its value. Unlike in the operational model (in which a read might be satisfied multiple times on speculative paths), there is exactly one satisfy event for each read axiomatic instruction instance.
2. *Initiate events.* Each write has an initiate event $ini(x)$, the point at which its address and value are computed, perhaps speculatively, and it becomes ready to be forwarded to local reads.
3. *Commit events.* Each axiomatic instruction instance x of any type has a commit event $com(x)$. Reads and writes can commit only after they are satisfied/initiated. Writes and barriers can propagate to other threads only after they are committed.
4. *Propagation events.* For each write or barrier instruction x and for each thread t which is not the originating thread of x, there is a propagation event $pp_t(x)$, which is the point at which x propagates to thread t.

The main part of our axiomatic model is defined using *evord*, a happens-before relation between events, which must be acyclic for consistent executions. Given an axiomatic candidate execution, *evord* is uniquely defined using the rules listed below.

Intra-instruction order edges. Our specification provides two ordering rules that relate events for the same instruction: *events-before-commit* states that reads must be satisfied and writes must be initiated before they commit; *propagate-after-commit* states that an instruction can propagate to other threads only after it is committed.

Local order edges. The *local-order* rules for *evord* relate *sat*, *ini* and *com* events within each thread. For a pair of events x and y from program-ordered axiomatic instruction instances (x before y), these events must occur in program order and cannot be reordered in the following cases:

– x is a read satisfy event and y is a read satisfy or write initiate event of an instruction with either an address or data dependency on x.
– x and y are read satisfy events separated by lwsync in program order.
– x and y are read or write commit events of instructions that have either data, address, or control dependency between them.
– x and y are read or write commit events for instructions accessing the same address.
– x and y are commit events and at least one of x and y is a barrier.

– x is a conditional branch commit event and y is a commit event.
– x and y are read or write commit events and there is a program-order-intervening instruction whose address depends on x.
– x is a read commit event, y is a read satisfy event, and both reads accessing the same address but reading from different non-local writes.
– x is a commit event of sync or isync and y is a read satisfy event.

Communication order edges. Communication rules order reads and writes to the same address from different threads, based on the relations *rf*, *co*, and *fr*. The *read-from* rule ensures that a read is satisfied only after the write it reads from propagates to the reading thread. The *coherence-order* rule states that for a write w, any write w' that is coherence-after w can propagate to w's thread only after w commits. The *from-read* rule defines which writes can be observed by each read. It states that if a read r reads from w, any write w' that is coherence-after w can propagate only to the thread of r after r is satisfied.

One implication of the above definition for the from-read rule is that the reads in each thread can only observe writes in coherence order, even if they propagate out-of coherence order. For example, if w_1 is coherence-before w_2, and w_1 propagates to t after w_2, any program-order-later read in t would still read from w_2 and not w_1.

Intra-thread communication edges. If a read receives a value written by a local write, this read must be satisfied after the write is initiated. This edge is the only type of intra-thread communication edge in this specification. There are no *evord* edges arising from *fr* or *co* edges between events for axiomatic instruction instances in the same thread.

Before edges (barrier cumulativity). In the operational specification, any write reaching thread t before a barrier is committed in t must (unless superseded by a coherence successor) be propagated to any other thread before that barrier is propagated, as shown in the WRC test. Our axiomatic specification expresses the same cumulative property using a *before-edge* rule, stating that if a write propagates to a thread before a barrier commits or vice versa (a barrier propagates before a write commits), then the propagation events for these two instructions must have the same order between them in any other thread.

Before edges apply both to events associated with instructions from the same thread and instructions from different threads. For same-thread instructions, they require writes separated by a barrier to propagate to other threads in program order. For instructions from different threads, their effect is enforcement of a global propagation order between writes and barriers that are related by communication edges.

After edges (sync total order). Heavyweight syncs are totally ordered. A sync may commit only after all previously committed syncs in the program have finished propagating to all threads. We enforce this using the *after-edge* rule, which states that if a sync b propagated to a thread after committing a local sync a, then any event associated with b must be ordered after any event associated with a. Note that this is different from (and simpler than) the operational model, where sync propagations can overlap.

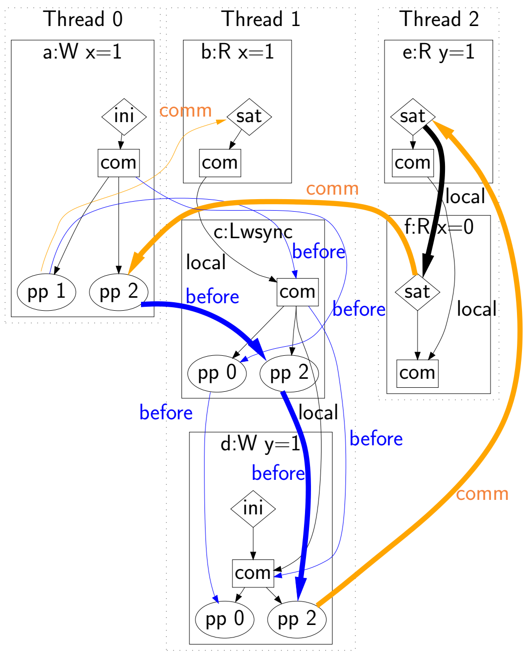
Extended coherence order. The extended coherence order, *cord*, is a relation between axiomatic instruction instances that includes *co*, as well as edges from each write w to each barrier b that commit after w propagates to b's thread, and from each barrier b to each write w that commits after b propagates to w's thread. Extended coherence must be acyclic, which captures the coherence/lwsync properties of examples such as 2+2W+lwsyncs.

3.3 Examples

We now discuss how the examples in Fig.1 are explained by this model, principally by looking at the *evord* relation between the events in each litmus test.

MP. In this example, there is a *read-from* edge from the pp_1 of b to the *sat* of c, and a *from-read* edge from the *sat* of d to the pp_1 of a. Without any additional dependencies or barriers, there are no additional edges and no cycles. Adding a dependency between the two reads on Thread 1 would add a local edge between the *sat* events of these reads. Adding an lwsync between the writes in Thread 0 would add local edges from a to the barrier to b and *before* edges between the pp_1 events of these three instruction instances, forming the cycle: $pp_1(a) \to pp_1(b) \to sat(c) \to sat(d) \to pp_1(a)$, making this forbidden.

WRC+lwsync+addr. The *evord* for this test is shown on the right. In this example, a is read by b, leading to a communication edge between them. As a result, the pp_1 event of a precedes the lwsync barrier, triggering a before-edge between them and forcing a to propagate before c in Thread 2. The result is a cycle: $pp_2(a) \to pp_2(c) \to pp_2(d) \to sat(e) \to sat(f) \to pp_2(a)$. Without the barrier, there would be no *before* edges connecting the propagation events of a and c. These two writes would be allowed to propagate to Thread 2 in any order, hence this example would be allowed.



Test WRC+lwsync+addr Candidate 4

IRIW+addrs. In this example, without barriers there would be no edges between the propagation events of a and d and therefore they could be observed in any order by Threads 1 and 3. Adding lwsync between the reads in this example adds *before* edges from a to the barrier in Thread 1 and from d to the barrier in Thread 3, but there are no edges connecting the propagation events of the two writes yet. Replacing lwsync with a heavy-weight sync, however, would add *after* edges from the propagation event of the sync in Thread 1 to the sync in Thread 3, as well as as *after* edges from the sync in Thread 3 to Thread 1, and therefore there would be a cycle.

PPOCA. In this example, there are local edges between the *com* events of Thread 1. For the *sat* and *ini* events, there are edges between d and e (intra-thread communication) and between e and f (local), but control dependency does not add edges between the satisfying c to the initiate of d, and hence there is no cycle.

Z6.3+lwsync+lwsync+addr. In this test, the only communication edge connecting Thread 0 and Thread 1 is from the *com* of b to pp_0 of c (due to coherence). This edge does not generate any *before* edges because it does not order any propagation event of Thread 0 before the barrier in Thread 1. Therefore, there are no edges between the propagation events of a and d, and hence there is no cycle.

2+2W+lwsyncs. In this example, there are *before* edges between a and b and between c and d, due to the barriers in these threads. Furthermore, the coherence order between b and c creates a communication edge between the *com* of b to the pp_0 of c (and similarly for d and a). These edges do not form a cycle in *evord*. However, the before-edges and coherence relation form a cycle in *cord*, and therefore this test is forbidden.

3.4 Formal Specification

The formal specification of the model, automatically typeset from the Lem [OBZNS11] definition, is shown in Figure 2. The following functions define the edges of *evord*: *local_order* defines which pairs of instructions form the local edges; the *events_before_commit* and *propagate_after_commit* order the events of the same instruction; *communication* defines the communication edges; *read_from_initiated* is the intra-thread communication edges; *fbefore* defines which instructions are ordered by before-edges, and *before_evord_closure* defines the actual edges; similarly, *fafter* and *after_evord_closure* define the after edges. The *evord* relation itself is defined as the least fixed point of *evord_more*, starting from *evord_base*. Also listed are the uniprocessor correctness rule, *uniproc*, and the *cord* relation, *cord_of*.

4 Experimental Validation

We establish confidence in our axiomatic specification experimentally, by checking that it gives the same allowed and disallowed behaviors as the operational model [SSA$^+$11], using a large suite of tests designed to expose a wide variety of subtle behaviors. We also check that the model is sound with respect to the observable behavior of current POWER hardware.

```
let uniproc ace =
(* Uniprocessor correctness condition *)
  let comm = communication_of ace in
  let commr = transitive_closure_of comm in
  (∀(x, y)∈commr. ¬ ((y, x) ∈ ace.ace_po))
let local_order ace events =
(* local order rules: do not reorder if true *)
  {(ex, ey)|∀ex∈events, ey∈events |
    (instruction_of ex, instruction_of ey) ∈ ace.ace_po ∧ (
      (is_sat ex ∧ is_ini ey ∧ (instruction_of ex, instruction_of ey) ∈ ace.ace_datadep) ∨
      (is_sat ex ∧ (is_ini ey ∨ is_sat ey) ∧ (instruction_of ex, instruction_of ey) ∈ ace.ace_addrdep) ∨
      (is_com ex ∧ is_com ey ∧ same_addr ex ey) ∨
      (is_com ex ∧ is_com ey ∧ (instruction_of ex, instruction_of ey) ∈ ace.ace_datadep) ∨
      (is_com ex ∧ is_com ey ∧ (instruction_of ex, instruction_of ey) ∈ ace.ace_addrdep) ∨
      (is_com ex ∧ is_com ey ∧ (instruction_of ex, instruction_of ey) ∈ ace.ace_ctrldep) ∨
      (is_com ex ∧ is_com ey ∧ is_fence ex) ∨
      (is_com ex ∧ is_com ey ∧ is_fence ey) ∨
      (is_branch ex ∧ is_com ey) ∨
      (is_com ex ∧ is_com ey ∧ (∃ez∈events. (instruction_of ex, instruction_of ez) ∈ ace.ace_po ∧
         (instruction_of ez, instruction_of ey) ∈ ace.ace_po ∧ (instruction_of ex, instruction_of ez) ∈ ace.ace_addrdep)) ∨
      (is_com ex ∧ is_read ex ∧ is_read_ext ace ey ∧
          same_addr ex ey ∧ ¬ (same_read_from ace ex ey)) ∨
      (is_lwsync ex ∧ is_read_satisfy ey) ∨
      (is_com ex ∧ is_read ex ∧ is_read_satisfy ey ∧ (∃ez∈events. (instruction_of ex, instruction_of ez) ∈ ace.ace_po ∧
        (instruction_of ez, instruction_of ey) ∈ ace.ace_po ∧ is_lwsync ez)) ∨
      (is_com ex ∧ (is_sync ex ∨ is_isync ex) ∧ is_read_satisfy ey))}
let events_before_commit ace events =
 {(ex, ey)|∀ex∈events, ey∈events | ((is_ini ex ∨ is_sat ex) ∧ is_com ey ∧ instruction_of ex = instruction_of ey)}
let propagate_after_commit ace events =
  {(ex, ey)|∀ex∈events, ey∈events | (is_com ex ∧ is_propagate ey ∧ instruction_of ex = instruction_of ey)}
let communication ace events comm =
  {(ex, ey)|∀ex∈events, ey∈events |
    (instruction_of ex, instruction_of ey) ∈ comm ∧ (
      (is_read_satisfy ex ∧ is_propagate ey ∧ propagation_thread_of ey = SOME (thread_of ex)) ∨
      (is_propagate ex ∧ is_read_satisfy ey ∧ propagation_thread_of ex = SOME (thread_of ey)) ∨
      (is_write_commit ex ∧ is_propagate ey ∧ propagation_thread_of ey = SOME (thread_of ex)))}
let read_from_initiated ace events =
  {(ex, ey)|∀ex∈events, ey∈events |
    (is_ini ex ∧ is_sat ey ∧ thread_of ex = thread_of ey ∧ (instruction_of ex, instruction_of ey) ∈ ace.ace_rf)}
let fbefore ace events ex ey =
  (is_write ex ∧ is_fence ey) ∨ (is_fence ex ∧ is_write ey)
let fbefore_evord_closure ace events evord_0 =
  {(ex, ey)|∀ex∈events, ey∈events | (∃tid∈ace.ace_threads. relevant_to_thread ex tid ∧ relevant_to_thread ey tid) ∧
    fbefore ace events ex ey ∧
    (∃(ex_1, ey_1)∈evord_0.
      relevant_to_thread ex_1 (thread_of ey) ∧ relevant_to_thread ey_1 (thread_of ey) ∧
      instruction_of ex_1 = instruction_of ex ∧ instruction_of ey_1 = instruction_of ey)}
let fafter ace events ex ey =
  (is_sync ex ∧ is_sync ey ∧ ex < > ey)
let fafter_evord_closure ace events evord_0 =
    {(ex, ey)|∀ex∈events, ey∈events |
    (∃tid∈ace.ace_threads. relevant_to_thread ex tid ∧ relevant_to_thread ey tid) ∧
    fafter ace events ex ey ∧
    (∃(ex_1, ey_1)∈evord_0. relevant_to_thread ex_1 (thread_of ex) ∧
      instruction_of ex_1 = instruction_of ex ∧ instruction_of ey_1 = instruction_of ey)}
let evord_base ace events comm =
  local_order ace events ∪
  read_from_initiated ace events ∪
  events_before_commit ace events ∪
  propagate_after_commit ace events ∪
  communication ace events comm
let evord_more ace events evord_0 =
  fbefore_evord_closure ace events evord_0 ∪
  fafter_evord_closure ace events evord_0 ∪
  {(ex, ez)|∀(ex, ey)∈evord_0, (ey', ez)∈evord_0 | ey = ey'}
let cord_of ace events evord =
   let fbefore_cord = fbefore_cord_of ace events evord in
   fbefore_cord ∪ ace.ace_co
```

Fig. 2: Formal specification of POWER in Lem

Implementations. For the operational specification, we use the `ppcmem` tool [SSA$^+$11], which takes a test and finds all possible execution paths of the abstract machine. For the axiomatic specification, we adapt the `ppcmem` front end to (straightforwardly) enumerate the axiomatic candidate executions of a test, then filter those by checking whether they are allowed by the definition of the axiomatic model. The kernel for both tools is OCaml code automatically generated from the Lem definition of the model, reducing the possibility for error.

Test suite. Our test suite comprises 4480 tests, including the tests used to validate the operational model against hardware [SSA$^+$11]. It includes the *VAR3* systematic variations of various families of tests (`http://www.cl.cam.ac.uk/users/pes20/ppc-supplemental/test6.pdf`); new systematic variations of the basic MP, S and LB tests, enumerating sequences of intra-thread relations from one memory access to the next, including address dependencies, data dependencies, control dependencies and identity of addresses. It also includes the tests of the *PHAT* experiment, used to validate the model of [AMSS10]; and hand-written tests by ourselves and from the literature. Many were generated with our diy tool suite from concise descriptions of violations of sequential consistency [AMSS11]. The tests and detailed results are available in the on-line supplementary material.

Results: comparing the axiomatic and operational models. We ran all tests with the implementations of both models. The preexisting tool for evaluating the operational model gives a verdict for only 3565 of the tests; the remaining 915 tests fail to complete by timing out or reaching memory limits. In contrast, the implementation of our axiomatic model gives a verdict for all 4480 of the tests. For all those for which the operational implementation gives a verdict, the operational and axiomatic specifications agree exactly.

Results: comparing the model to hardware implementations. We also used the test suite to compare the behavior of the axiomatic specification and the behavior of POWER6 and POWER7 hardware implementations, as determined by extensive experimental data from the `litmus` tool (`http://diy.inria.fr/doc/litmus.html`). In all cases, all the hardware-observable behaviors are allowed by the axiomatic model. As expected, the model allows behaviors not observed in current hardware implementations, because our models, following the POWER architectural intent, are more relaxed in some ways than the behavior of any current implementation [SSA$^+$11]. This result covers the 915 tests on which the operational model timed out, which gives evidence that the axiomatic model is not over-fitted to just the tests for which the operational result was known.

5 Proof of Equivalence to the Operational Specification

We establish further confidence in the equivalence of the axiomatic model presented here and the operational model, by providing a paper proof that the sets of behaviors allowed for any program are identical for both models: we show that any outcome allowed by the operational model is allowed in the axiomatic model and vice versa. In this section we provide an overview of the proof; the full proof is in the on-line supplementary material [Sup].

5.1 Operational to Axiomatic

The first part of the proof shows that any allowed test in the operational model is an allowed test in the axiomatic model. We do this by defining a mapping function *O2A* from sequences of transitions of the operational model to a program execution and *evord* relations in the axiomatic model, proving that the resulting *evord* and *cord* are always acyclic.

A witness trace $W = \{tr_1, ...tr_n\}$ is a sequence of operational-model transitions, from an initial to a final system state. Given a witness W, our *O2A* mapping generates a relation *evord′* by iterating over all labeled transitions. At each step, *O2A* adds the corresponding events to *evord′*, and adds edges to the new event if they are allowed by *evord*. Most events correspond directly to certain transition types in the machine, with two exceptions: (1) *initiate-write* events do not correspond directly to any transition type, and are added to *evord′* either before the first forwarded read or before their commit; and (2) *irrelevant write propagation* events are write propagation events that do not correspond to any write propagation transition. These are added to *evord′* either before barriers (when required by before edges), or at end of the execution.

Another difference between the two models is in the handling of sync barriers. In the axiomatic model, *after* edges enforce a total order between syncs, effectively allowing syncs to propagate one at a time. In the operational model, a thread stops after a sync and waits for an acknowledgment that it propagated to all other threads, but several syncs can propagate simultaneously. When the mapping encounters a sync-acknowledge transition, it adds after-edges between this sync and all previously acknowledged syncs.

Theorem 1. *Given a witness $W = \{tr_1, ...tr_n\}$, the mapping $O2A(W)$ produces an axiomatic program execution with co and rf that satisfy the uniproc condition and acyclic evord and cord relations.*

We prove that *evord* for the axiomatic program execution is acyclic by showing that the *evord* relation produced by *O2A* is both: (1) acyclic and (2) the same as the *evord* which is calculated from *co* and *rf*.

For all edges except the after edges, the *evord* produced by the mapping is acyclic by construction, because each newly added event is ordered after the previously added events. After edges are added between existing events when an acknowledge transition is encountered. The following Lemma guarantees that after edges do not form a cycle:

Lemma 1 (*Sync acknowledge for ordered sync propagations*). *If b_1 and b_2 are two sync instructions and b_1 is acknowledged before b_2, then there is no path in evord′ from an event of b_2 to an event of b_1.*

The mapping adds edges only if the corresponding *evord* edges are allowed. To show that all the edges in *evord* are in *O2A*(W), we show that the mapping adds events in an order than agrees with the direction of the edges in *evord*. For each type of edge in *evord*, we show that the transition rules of the operational model guarantee this order.

5.2 Axiomatic to Operational

The second part of the proof shows that each allowed execution in the axiomatic specification is allowed by the operational specification. We define a mapping that takes the the relations evaluated for the axiomatic specification, including *rf*, *co*, *evord*, and *cord*, and produces a sequence of transitions W for the operational specification.

Given an axiomatic candidate execution $CE = \{P, rf, co\}$ accepted by the axiomatic model, the *A2O* mapping generates a witness $W = \{tr_1, ...tr_n\}$ for the operational model. The mapping takes *evord* (which is acyclic for allowed executions), performs a topological sort of the events in *evord*, and then it processes these events in that order to produce W.

The *A2O* mapping translates most events directly into corresponding transitions, with a few notable exceptions: (1) there are no transitions matching write initiate events; (2) write propagation events are allowed out-of-coherence-order, whereas write propagation transitions in the operational model must be in coherence order but some writes may be skipped, as identified by the mapping; (3) no event corresponds directly to sync acknowledge transitions (which are produced after a sync propagates to all threads); and (4) no events correspond to partial-coherence-commits, which are produced by the mapping according to *co* after processing write commit events.

Theorem 2. *Given an allowed candidate execution CE, the mapping A2O(CE) produces a witness for an accepting path in the operational model.*

We prove this by induction on W. For each transition in W, we show that each type of transition is allowed based on the rules of *evord* as well as *cord* and the uniprocessor rule.

6 Evaluating the Axiomatic Specification with a SAT Solver

One advantage of the constraint-based axiomatic specification presented in this paper is that it can be readily used by constraint solvers (such as SAT or SMTs). To investigate this impact, we built a C++ implementation of the axiomatic specification using the minisat SAT solver [ES05]. Currently, this solver accepts diy sequential-consistency-violation cycles as input (rather than litmus sources), builds the corresponding tests internally and checks whether the resulting tests are allowed or forbidden. We ran this solver on the 4188 of the tests that were built from cycles.

model/tool	N	mean (s)	max (s)	effort (s)	memory
Operational/ppcmem	3565/4480	3016.19	2.4e+05	8.2e+07	40.0 Gb
Axiomatic/ppcmem	4480/4480	1394.14	2.3e+05	7.1e+06	4.0 Gb
Axiomatic/SAT	4188/4188	2.67	10.26	11170	—

Table 1: Test suite runtime in the three checkers

We compare the execution time of this SAT-based tool to the `ppcmem` checkers for the operational and axiomatic specifications described in Section 4, which were built from Lem-derived code, emphasizing assurance (that they are expressing the models exactly as defined) over performance. They do not always terminate in reasonable time and space, so we resorted to running tests with increasing space limits, using 500+ cores in two clusters. In Table 1, N is the number of tests finally completed successfully in the allocated processor time and memory limits, w.r.t. the number tried. "mean" shows the arithmetic mean of the per-test execution time of successful runs; while "max" is the execution time for the test that took longest to complete successfully. The "effort" column shows the total CPU time allocated to running the simulators (including failed runs due to our resource limits). Finally, the "memory" column shows the maximum memory limit we used.

As shown by the last row of the table, the performance improvement of the SAT-based checker over either the operational or axiomatic versions of `ppcmem` is dramatic: the SAT solver terminates on all 4188 cycle-based tests, taking no more than about 10 seconds to run any test. The total computing effort is around 3 hours (wall-clock time is about 25 minutes on an 8-core machine) compared with the 82 CPU-days of the `ppcmem` axiomatic tool and 950 days of the `ppcmem` operational tool. One obtains similar results when restricting the comparison to the tests common to all three tools.

This efficient SAT-based encoding of the model opens up the possibility of checking properties of much more substantial example programs, e.g. implementations of lock-free concurrent data-structures, with respect to a realistic highly relaxed memory model.

Acknowledgments The authors acknowledge the support of NSF grants CCF-0905464 and CCF-0644197; of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity; and funding from EPSRC grants EP/F036345, EP/H005633, and EP/H027351, ANR project ParSec (ANR-06-SETIN-010), ANR grant WMC (ANR-11-JS02-011), and INRIA associated team MM.

Power Architecture are registered trademarks of International Business Machines Corporation.

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), 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 *Workshop on Declarative Aspects of Multicore Programming*, January 2009.

[AMSS10] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *International Conference on Computer Aided Verification*, 2010.

[AMSS11] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2011.

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

[ES05] N. Een and N. Sorensson. Minisat - a SAT solver with conflict-clause minimization. In *International Conference on Theory and Applications of Satisfiability Testing*, 2005.

[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`.

[OBZNS11] S. Owens, P. Böhm, F. Zappa Nardelli, and P. Sewell. Lem: A lightweight tool for heavyweight semantics. In *Proc. ITP: Interactive Theorem Proving, LNCS 6898*, pages 363–369, August 2011. ("Rough Diamond" section).

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

[SMO$^+$12] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *Programming Language Design and Implementation*, 2012.

[SSA$^+$11] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Programming Language Design and Implementation*, 2011.

[Sup] An axiomatic memory model for Power multiprocessors — supplementary material. `http://www.seas.upenn.edu/~selama/axiompower.html`.

[YGLS03] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In *Correct Hardware Design and Verification Methods*, 2003.