Synchronising C/C++ and POWER

Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell,
Luc Maranget, Jade Alglave, Derek Williams

Synchronising C/C++ and POWER

Susmit Sarkar[1] Kayvan Memarian[1] Scott Owens[1] Mark Batty[1] Peter Sewell[1]
Luc Maranget[2] Jade Alglave[3] Derek Williams[4]

[1]University of Cambridge, {first.last}@cl.cam.ac.uk
[2]INRIA, luc.maranget@inria.fr
[3]University of Oxford, jade.alglave@comlab.ox.ac.uk
[4]IBM Austin, striker@us.ibm.com

Abstract

Shared memory concurrency relies on synchronisation primitives: compare-and-swap, load-reserve/store-conditional (aka LL/SC), language-level mutexes, and so on. In a sequentially consistent setting, or even in the TSO setting of x86 and Sparc, these have well-understood semantics. But in the very relaxed settings of IBM® POWER®, ARM, or C/C++, it remains surprisingly unclear exactly what the programmer can depend on.

This paper studies relaxed-memory synchronisation. On the hardware side, we give a clear semantic characterisation of the load-reserve/store-conditional primitives as provided by POWER multiprocessors, for the first time since they were introduced 20 years ago; we cover their interaction with relaxed loads, stores, barriers, and dependencies. Our model, while not officially sanctioned by the vendor, is validated by extensive testing, comparing actual implementation behaviour against an oracle generated from the model, and by detailed discussion with IBM staff. We believe the ARM semantics to be similar.

On the software side, we prove sound a proposed compilation scheme of the C/C++ synchronisation constructs to POWER, including C/C++ spinlock mutexes, fences, and read-modify-write operations, together with the simpler atomic operations for which soundness is already known from our previous work; this is a first step in verifying concurrent algorithms that use load-reserve/store-conditional with respect to a realistic semantics. We also build confidence in the C/C++ model in its own terms, fixing some omissions and contributing to the C standards committee adoption of the C++11 concurrency model.

1. Introduction

Synchronisation is fundamental to shared-memory concurrency, but the properties of basic real-world synchronisation primitives remain surprisingly unclear: there is a big gap between the usual descriptions of their behaviour, which presuppose a sequentially consistent (SC) [Lam79] setting, and their behaviour in actual multiprocessors and programming languages, which are not SC. Instead of interleaving operations from different threads, all with a consistent view of memory, real systems expose *relaxed memory models* to the programmer, to allow for hardware and compiler optimisations. In such a setting, even basic properties, such as locks ensuring that locations are "not concurrently accessed", become hard to understand, as one cannot reason in simple global-time terms; the semantics of synchronisation primitives are necessarily intertwined with the relaxed semantics of other code, including whatever loads, stores, and barriers are present.

This paper addresses relaxed-memory synchronisation in hardware and in software, and the relationship between the two. Our first main contribution, on the hardware side, is a usable model for the synchronisation primitives, load-reserve/store-conditional pairs, of the highly relaxed IBM® Power Architecture® (§2). While these have been present in the architecture for 20 years, their relaxed-memory behaviour have never before been clearly explained (for example, confusion on this point has led to a Linux kernel bug in the implementations of atomic operations [McK11], as we remark in §2). We do so with a novel abstraction, of a write *reaching coherence point*, with just the properties that are needed in the semantics. This removes any need for modelling the implementation detail of reservation registers, which becomes very complex in a setting with speculative and out-of-order execution.

We establish confidence in our model in multiple ways (§3), including extensive experimental testing (comparing processor behaviour against the model), detailed discussion with IBM staff, and proofs of basic properties, about the strength of load-reserve/store-conditional pairs (§3.2), and that a simple locking discipline guarantees SC for POWER programs (§5).

For our second main contribution, on the software side, we prove correctness of a mapping of the principal C/C++ synchronisation constructs to POWER (§5). We consider C/C++ locks, atomic read-modify-write (RMW) operations, fences, and atomic loads and stores, with the various possible memory-order parameters, from relaxed to SC (for locks we consider basic mutexes, not reentrant or timed, and just their lock and unlock interface, and we do not consider condition variables or futures). We take the implementation of spinlocks from the POWER documentation [Pow09, p.717,718], combining that with the POWER implementation of C/C++ RMWs, fences and atomic operations proposed by McKenney and Silvera [MS11]. Locks and RMWs both rely on the POWER load-reserve/store-conditional.

In the process of doing this proof, we identified and fixed several technical issues with the C/C++11 concurrency model and with our previous formalisation; we describe those in §4, together with a brief summary of the design of that model and an explanation of the C/C++ fences, to make this paper as self-contained as possible.

POWER and C/C++11 are both subtle models, though of quite different kinds, and there is an interplay between them: implementability on POWER (and the similar ARM) had a major influence on the design of C/C++11, and, conversely, the desire to support efficient language implementation imposes constraints on the architecture. Our work illuminates such choices by showing what is necessary for soundness; as we show in §2, the models are in some respects "tight" against each other.

More generally, our correctness proof for this lock implementation is a first step towards reasoning about more sophisticated concurrent algorithms (using load-reserve/store-conditional synchronisation directly, or in C/C++) with respect to a realistic semantics; and our models provide a basis for future work on compiler correctness, program analysis and verification, and tool building.

Our models and the statements of our results and key lemmas are expressed in the lightweight mechanised specification language

Lem [OBZNS11]; our proofs are rigorous but non-mechanised. The details of both are available online in the supplementary material [cpp], together with the data from our experimental testing and our `ppcmem` web interface for interactively exploring the model.

Related Work There is surprisingly little directly related work. We build first on our own recent line of work, where we have developed a high-fidelity abstract-machine model of the loads, stores and barriers of the POWER architecture, without load-reserve/store-conditional (Sarkar et al. [SSA$^+$11]). On the C/C++ front, Boehm and Adve describe the core of the C/C++11 design [BA08]. We have provided a mathematisation of the concurrency model of C/C++11 (Batty et al. [BOS$^+$11]), which we correct and use here. There, we also proved correctness of a compilation scheme from C/C++11 nonatomic and atomic loads and stores, and fences, to x86-TSO. The strength of the target memory model (and the omission of locks and RMWs) makes that a much simpler problem than the one we address here. We previously used these two models to show correctness of compilation from C/C++11 to POWER, for the fragment of C/C++ consisting just of nonatomic and atomic stores and loads of various kinds [BMO$^+$12]. Here our extended models and proof finally cover all of the main features of concurrency in POWER and in C/C++.

There have been, to the best of our knowledge, three previous serious attempts at realistic relaxed-memory semantics of load-reserve/store-conditional in the POWER setting, none of which are satisfactory for the current architecture. Corella, Stone, and Barton [CSB93] give an axiomatic model and show the correctness of an implementation of locks in terms of it. Their model does not cover the weaker `lwsync` barrier (which was not present in the Power Architecture of the time), and their lock implementation is therefore more conservative than the one we consider, with two `sync`s. Moreover, as Adir et al. note [AAS03], the basic model is flawed in that it permits the non-SC final state of a message-passing-with-syncs example. Adir et al. give a more elaborate model, with a view order per thread, sketching extensions for the synchronisation instructions, but this is still for the non-cumulative barriers of the time. Our previous global-time axiomatic model (Alglave et al. [AM11]) appears reasonable for load-reserve/store-conditional in isolation but too weak for the POWER `lwsync` barrier, permitting some behaviour that the C/C++11 mapping requires to be forbidden.

2. Optimistic Synchronisation on POWER: Load-reserve/Store-conditional

Load-reserve/store-conditional primitives were introduced by Jensen *et al.* [JHB87] as a RISC-architecture alternative to the compare-and-swap (CAS) instruction; they have been used in the IBM® PowerPC® architecture since 1992 and are also present in ARM, MIPS, and Alpha. They are also known as load-linked/store-conditional (LL/SC), or, on ARM, load-exclusive/store-exclusive. They provide a simple form of optimistic concurrency (very roughly, optimistic transactions on single locations).

Herlihy [Her93] uses load-reserve/store-conditional to implement various wait-free and lock-free algorithms, noting that (as for CAS, but unlike test-and-set and fetch-and-add) it is *universal* in terms of consensus number, and moreover that load-reserve/store-conditional is practically superior to CAS in that it defends against the ABA problem. These results, as for almost all other work in concurrent algorithms, use an underlying sequentially consistent (SC) semantics.

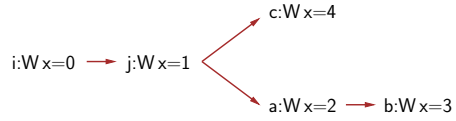
In a sequentially consistent setting, one can describe the behaviour of load-reserve/store-conditional fairly simply. A load-reserve does a load from some memory address; and a subsequent store-conditional to the same address will either succeed or fail. It will definitely fail if some other thread has written to that address in the meantime (or for some other reasons), otherwise (where the write read from by the load-reserve is still the most recent write to that address) it might succeed, performing a store. Moreover, the store-conditional sets a flag so that later instructions can determine whether or not it succeeded; load-reserve/store-conditional pairs are often repeated until success. This makes it easy to express a variety of atomic update operations, as in the following POWER assembly code for an atomic add (ARM code would be similar, with LDREX and STREX):

```
1:lwarx  r0,0,r2  \\ load-reserve from [r2] to r0
  add    r0,r1,r0 \\ add register r1 to register r0
  stwcx. r0,0,r2  \\ store-conditional r0 to [r2]
  bne-   1b       \\ ...looping until success
```

In an SC world, the store-conditional would be guaranteed to fail if some other thread writes to `[r2]` since the load-reserve; which could be achieved in hardware by monitoring whether any other thread gains write ownership of the cache line holding `[r2]` in the meantime. Note that other operations are permitted between the load-reserve and store-conditional, including memory reads and writes, though, unlike transactions, nothing is rolled back if the store-conditional fails.

However, actual POWER and ARM multiprocessors are *not* sequentially consistent, but rather have highly relaxed memory models, in which program executions are not simple interleavings of the instructions of each thread. A priori, it is unclear what it means to say that another thread has written to the relevant address *between* the load-reserve and store-conditional, as one has to understand how they interact with the thread-local out-of-order and speculative execution, and with the storage subsystem reordering, of these machines. Moreover, the vendor architecture [Pow09] does not resolve these questions as clearly as one might hope. It describes the behaviour of load-reserve/store-conditional informally in terms of *reservations*, allowing a store-conditional to succeed only if *"the storage location specified by the lwarx that established the reservation has not been stored into by another processor or mechanism since the reservation was created"*. This is simultaneously too much and too little hardware implementation detail: the *"since"* cannot refer to an SC order, but the machine execution order is not precisely specified.

We solve these problems with an abstract-machine semantics, extending the model of Sarkar et al. [SSA$^+$11] to cover load-reserve/store-conditional. That model comprises a thread semantics, with explicit out-of-order and speculative execution of each hardware thread, composed with a storage subsystem semantics. The latter abstracts from the cache and store-buffer hierarchy and from the cache protocol: for each address it maintains just a strict partial order, the *coherence order*, of the ordering commitments that have been made among the writes to that address, and for each thread it maintains a list of the writes (and barriers) that have been propagated to that thread. For example, at a certain point in abstract-machine execution, one might have established these constraints among 5 writes (by different threads) to x:



with writes [i:W x=0, a:W x=2] propagated to some arbitrary thread. In that state, a read of x by that thread would see the most recent of those two, reading 2; or write b could be propagated to that thread, appending it to that list (as it is coherence-after the writes already propagated there); or a partial coherence commitment transition could make c ordered before a, or after a but still

unrelated to b, or before b but still unrelated to a, or after b. A new write by a thread is made coherence-after all writes previously propagated to that thread but initially coherence-unrelated to other writes. This machinery guarantees coherence in the normal sense, and also supports a semantics for the POWER cumulative memory barriers in its non-multi-copy-atomic setting, where writes to different addresses can propagate to other threads in different orders. The `sync` and `lwsync` barriers constrain that propagation, as we return to in §5, and a `sync` also waits until writes that have previously been propagated to its thread (the `sync`'s *Group A*), or some coherence successors thereof, have been propagated to all threads.

Looking at a successful load-reserve/store-conditional pair in these terms, we can see that for it to provide an atomic update, the write of the store-conditional must become an *immediate coherence successor* of the write read from by the load-reserve, with no other coherence-intervening write to the same address. Moreover, that property must be maintained: no other write can be allowed to become coherence-intervening later in the abstract-machine execution. To express this, the key thing we need to introduce to the model is the concept of a write *reaching coherence point*, after which its coherence-predecessors are linearly ordered and fixed (no additional write can become coherence-before it). Our main new rule describing how the storage subsystem (in state s) can process a successful store-conditional is as follows.

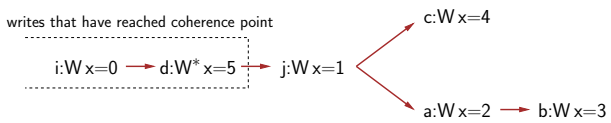
Accept a successful write-conditional request A write-conditional request w by a thread tid, with an accompanying $wprev$ that was read by the program-order-previous read-reserve, can be accepted and succeed if:

1. the write $wprev$ is to the same address as w;
2. $wprev$ has reached coherence point;
3. no coherence-successor of $wprev$ by another thread has reached coherence point or has been propagated to thread tid; and
4. all writes by tid to the same address as $wprev$ (in particular, all those since $wprev$ was propagated to tid) have reached coherence point.

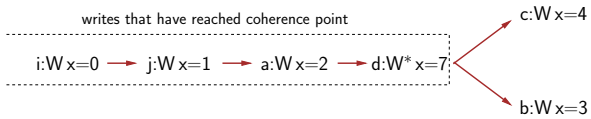
Action:

1. add the new write w to $s.writes\_seen$, to record the new write as seen by the storage subsystem;
2. append the new write w to $s.events\_propagated\_to\ (tid)$, to record the new write as propagated to its own thread;
3. record that w has reached coherence point; and
4. update $s.coherence$ by adding edges to make write w:
   (a) coherence-after all writes to the same address that have reached coherence point (including $wprev$); and
   (b) coherence-before all writes seen (except itself) to the same address that have not yet reached coherence point.

Note that a store-conditional can succeed even if there are outstanding other writes to the same address by different threads (that might have been read from by yet different threads), but that any such other writes become coherence-later than the store-conditional itself. For example, given the coherence order above, if only i has reached coherence point, and if just i has been propagated to some thread, that thread could do an atomic add of 5 by executing a load-reserve of x=0 from i, adding 5 to that value, and doing a successful store-conditional of x=5, the latter becoming an immediate coherence successor of i, and a coherence-predecessor of all the other writes:



Alternatively, in another execution, a could become coherence-before c, then j and a could reach coherence point, and that load-reserve could read 2 from a and add 5, with the resulting store-conditional of 7 becoming an immediate coherence successor, coherence-before just b and c:



A store-conditional can nondeterministically fail at any time, modelling clearing of the reservation by various events, including writes to other addresses within the same cache line, hypervisor/OS context switches, and '*implementation-specific characteristics of the coherence mechanism [which] cause the reservation to be lost*' [Pow09] (moreover, the architecture explicitly does not include a fairness guarantee). It is believed that current implementations do provide forward progress, but the architecture text does not guarantee it; without characterising those implementation-specific details, and making assumptions on the other events, we have to assume that arbitrary store-conditional failures are possible. This effectively restricts one to reasoning about safety properties.

In this paper we do not model the architecture's *reservation granules*: the rules above consider only whether write-conditionals and load-reserves are to exactly the same address (which is the normal use-case), but in the architecture the significant fact is whether the two addresses lie in the same reservation granule (in current implementations, reservation granules are the same size as cache lines). Modelling realistic (non-single-word) reservation granules would require a store-conditional to fail if there is an intervening write to the same granule, and permit a store-conditional to succeed if the preceding load-reserve was to any address in the same granule. This could be done by modifying preconditions in the above rule, to say that no write to an address in the same granule as the store-conditional has been propagated to tid since $wprev$ was read from by the load-reserve, and to say that the write $wprev$ read from by the load-reserve was to an address in the same granule as the store-conditional.

We also have to specify exactly when a write can reach coherence point, with the rule below. Note that `sync` and `lwsync` barriers constrain the order in which writes can reach coherence point, but, apart from that, barriers and load-reserve/store-conditionals are largely independent.

Write reaches its coherence point This is an internal transition of the storage subsystem, for a write it has already seen, if:

1. the write has not yet reached its coherence point;
2. all its coherence-predecessor writes have reached their coherence points; and
3. all writes (to any address, and by any thread) propagated to the writing thread before a barrier (also by the writing thread) before this write, have reached their coherence points.

Action:

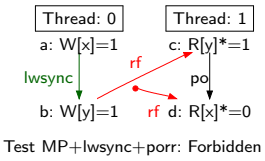
1. add this write to $s.writes\_past\_coherence\_point$; and
2. update $s.coherence$ to record that this write is coherence-before all writes seen (except itself) that are to the same address and are not past their coherence points.

Turning now to the thread semantics, load-reserve and store-conditional instructions all commit in program order relative to each other. Apart from that, load-reserves are committed just like normal loads, while store-conditionals are committed by a thread rule that synchronises with the storage-subsytem "Accept store-conditional request" rule above, the thread commit rule additionally imposing:

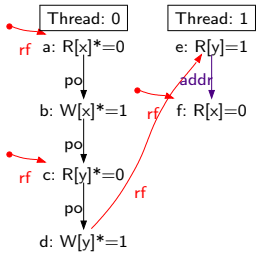
Commit in-flight instruction

> . . . if this is a write-conditional instruction: send a write-conditional request mentioning the write read from by the most recent program-order-preceding load-reserve instruction without an intervening write-conditional (if there is such a load-reserve); receive the corresponding success/fail response; and set flags accordingly. If there is no such load-reserve, simply set flags as for a fail.

Despite the commit-order constraint, a load-reserve can be *satisfied* (by reading a value from the storage subsystem) exactly like a normal load, and (like a normal load) this can be done out-of-order and speculatively, long before it is committed. In contrast to normal loads, however, there is an additional restriction on load-reserve speculation, that a load-reserve cannot be satisfied until all program-order previous load-reserves and store-conditionals are committed. This models the architectural requirement of a single reservation register per thread. In the model this leads to forbidding the example execution on the left below. Consistent with this, the example is not observable on IBM® POWER® 6 and IBM® POWER® 7 implementations.



Test MP+lwsync+porr: Forbidden



Test MP+poaa+addr: Allowed

Note also that despite the constraint on their commit order, store-conditionals to different addresses can still propagate to other threads out-of-order, as shown on the right above. This is allowed in the model and observable on POWER 6 and POWER 7.

Each diagram shows the memory read and write events of a candidate execution of an assembly program running on one or more hardware threads; the assembly source code is in the supplementary material. Events include unique ids (a, b, etc.), the addresses of reads and writes (x, y, etc.), and the values read and written. Load-reserve and store-conditional events are indicated with a star. Various edges pick out key relationships between instructions in the source, or more specifically in the particular control-flow unfolding of the source for the execution in question:

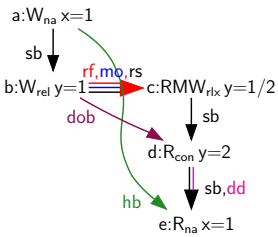
- po edges relates events from instructions in program order (in this control-flow unfolding);
- an lwsync edge indicates that there is a POWER lightweight sync barrier in program order between the instructions that gave rise to the two events;
- a sync edge indicates a POWER heavyweight sync barrier;
- an eieio edge indicates a POWER eieio barrier;
- an addr edge indicates that the address of the second event is dependent (through a dataflow path via registers and computation) on the value read by the first;
- a data edge indicates that the data written by the second event is dependent on the value read by the first;
- a ctrl edge indicates there is a dataflow path from the first read read to the test of a conditional branch that program-order-precedes the second event (branches are not shown explicitly); and
- a ctrlisync edge indicates 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.

Three further relations characterise the remainder of the dynamics of the execution:

- an rf edge from a write to a read indicates that the read read-from that write (rf edges from the initial state are marked with a red dot for the source);
- a co edge between writes to the same address gives the final coherence order between them; and
- an rcp edge between writes indicates the order in which those writes reached coherence point (usually we elide these edges).

Moreover, load-reserve and store-conditional do not impose any special constraint on normal loads and stores to different addresses: they do not act like a barrier of any kind, and so loads after a load-reserve/store-conditional pair might be speculated before it. If one wants to prevent that, one needs to add surrounding barriers. Confusion on this point seems to have been responsible for a Linux kernel bug [McK11] that was recently identified by McKenney and confirmed using our model: the Linux `atomic_add_return` implementation (from which the code above is taken), is assumed there to prevent speculation, but its implementation used overly weak barriers; the same was true for other similar read-modify-write operations. The example is in the supplementary material.

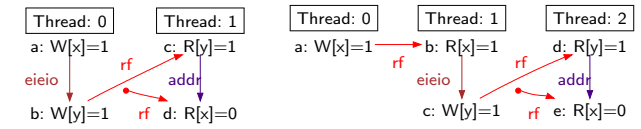
One also has to consider whether writes can be observably forwarded to a load-reserve or from a store-conditional within a thread, on speculative paths before they have reached the storage subsystem. This can happen on POWER and ARM for normal loads and stores, as shown by the PPOCA example of [SSA+11]. Interestingly, we can show that allowing such forwarding in the model from a store-conditional would make the mapping of C/C++11 atomics to POWER unsound: the C/C++ candidate execution on the right (in the notation recalled in §4) is forbidden in C/C++11, but the corresponding execution in POWER would be allowed in our model if speculative forwarding from the store-conditional of the RMW were to be permitted. The architectural intent (though this is arguably not completely clear from the text) is to rule out this behaviour in a different way, allowing implementation forwarding from store-conditionals but requiring that any store-conditionals must respect an implicit data dependence to the prior load-reserve through the architected reservation register, even when no actual data dependence is present; hence preventing such forwarding until the previous load-reserve has been satisfied. An implementation of such forwarding would be extremely aggressive and complex, and has not been attempted in practice.



This is a case where our POWER and C/C++11 models are tight against each other: one either has to rule out such forwarding becoming observable in the Power Architecture or permit the example in C/C++ (or change the mapping, but there is no obvious alternative that preserves the performance advantage of a consume over an acquire). It illustrates how constructing formal models at this level of detail can identify deep architectural and micro-architectural choices beyond current implementation practice, arising from the detailed interplay between the high-level language memory models and aggressively weakly ordered hardware models.

For completeness, we also include in this paper the POWER `eieio` barrier, in addition to the previously-covered `sync`, `lwsync`, and `isync`, though it is not used in the C/C++ mapping. Architecturally, `eieio` orders pairs of same-thread writes as far as all other threads are concerned (`eieio` has additional effects for Caching Inhibited memory, which we do not cover here), and the message-

passing test on the left is forbidden. However, the extension on the right, with the write to x pulled out to a third thread, is allowed, notwithstanding the architectural mention of cumulativity, because `eieio` does not order reads (e.g. b) w.r.t. writes.



Test MP+eieio+addr: Forbidden Test WRC+eieio+addr: Allowed

Adding `eieio` between `b` and `c` of the MP+poaa+addr example above suffices to rule out the non-SC behaviour there. We model `eieio` in the storage subsystem exactly like `lwsync`, together with a more relaxed thread commit rule. Preliminary testing suggests that current implementations have considerably stronger behaviour than the architecture, more like `lwsync`.

We also remark for completeness that we have made a technical change to our `lwsync` semantics: our [SSA$^+$11] model let reads after `lwsync`s be satisfied but restarted them, whereas now they are blocked until the `lwsync` commits. This has no observable effect but is simpler to work with.

Discussion Our semantics abstracts substantially from hardware implementation detail, focussing on the essential properties of the store-conditional. The notion of a write reaching coherence point abstracts from the hardware machinery needed to maintain coherence; it captures just the minimal properties required to express the semantics. From the literature and vendor documentation, one would be tempted to build an operational model with explicit per-thread reservation flags, set and cleared as the abstract machine executes, and indeed we did so at first, but that quickly becomes very complex when one considers reservations being set or cleared on speculative execution paths that may be rolled back. The 'implicit reservation' style that we introduce here, testing just the necessary coherence properties at write-conditional commit time, is much simpler to express and to work with.

3. Validation

Our model has been validated in six distinct ways. First, it was developed in discussion with a senior IBM POWER designer/architect, who has extensive experience of the POWER load-reserve/store-conditional implementations. Second, we built a tool, with a kernel generated from the Lem code of the model, which allows us to explore the behaviour of litmus-test example programs, interactively via a web interface (available in the supplementary material) and in batch mode (calculating all model-allowed behaviour); this tool was used in the analysis of the Linux bug mentioned above. Third, using that tool for regression testing, we checked that the addition of coherence points did not affect the observable behaviour allowed by the model for as many tests that do not involve load-reserve/store-conditional as we could test. Adding coherence points enlarges the search space needed to exhaustively explore tests, but the new version of the tool terminated in manageable memory occupancy (up to 10 GB) for 299 out of the 314 *VAR3* tests [SSA$^+$11], and also for approximately 450 new tests; for all these, the two models allow the same outcomes. Fourth, experimentally: we developed a large set of new litmus tests, as described below, ran them on generations of POWER machines (PowerPC G5, POWER 6 and POWER 7), and compared the results against those for the model (as produced by our tool). Fifth, we proved theoretical results about our POWER model on its own terms, in §3.2 below. Finally, our main theoretical result of this paper, proving that the C/C++11 synchronisation primitives can be correctly compiled to POWER, demonstrates that the model is strong enough for non-trivial usage.

3.1 Experimental Validation

First, we hand-wrote tests to exercise the basic functionalities of load-reserve and store-conditional instructions. For example, a test RSV+Fail shows that a store-conditional associated to a load-reserve will fail if there is a load-reserve to a distinct location (on actual machines, to a distinct reservation granule) in between in program order.

To systematically exercise corners of the model, we designed several series of tests, automatically generated with the diy tool [AMSS10]. The first and more important series (12499 tests) derives from the systematic tests of [SSA$^+$11]. Assuming that mapping plain accesses to load-reserves or read-modify-write sequences might forbid some behaviours previously allowed by the model, but not the other way round, we selected the tests allowed by the [SSA$^+$11] model (limited to 5 accesses to reduce the number of tests to run). We call these tests *plain tests*, since all accesses are plain reads and writes. Then we generated all the possible *atomic variations* of these plain tests, with a load-reserve or a load-reserve/store-conditional sequence in place of some plain accesses. For a read we use *fetch and no-op* (FNO), and for a write *fetch and store* (STA) constructs [Pow09, p.719]. Both FNO and STA read a value with a load-reserve, then the FNO writes back the same value with a store-conditional, while the STA writes a predefined different value with its store-conditional. Note that we cannot map a write to a store-conditional alone, because the store-conditional needs a program-order previous load-reserve to succeed. They are wrapped in loops, though in practice in our model tool we unroll the loops once.

The second series, "Rfi" (intra-thread, or internal, read-from), exercises the impact of atomic variations on store forwarding. To do so, we generated a restricted subset of our plain tests, namely some that exhibit a violation of SC in the form of a *critical cycle* [SS88], with at least one internal (intra-thread) read-from edge, and up to 3 threads; this gave a series of 1338 tests. We replaced the accesses along the internal read-from edges with FNO/STAs and load-reserves, as in the first series. Interestingly, many of the Rfi variations that involve only load-reserves were observed on POWER 6 but not on PowerPC G5 or POWER 7, e.g. a test 2+2W+lwsync+rfi-data. This suggests that the implementation of load-reserve is less liberal on the latter two.

The last series exercises in practice what we establish as a theorem about the model in §3.2: when *all* accesses of a test are replaced by FNO or STA, then the test only has SC behaviours. We tested this on a restricted subset of our plain tests, namely the 68 that exhibit a violation of SC *via* a critical cycle, and that have up to 4 threads. Running those tests on hardware showed no non-SC behaviour, consistent with the model's prediction as captured by the theorem.

Taking all these load-reserve/store-conditional tests together is 13963 tests. We have run all those on PowerPC G5, POWER 6 and POWER 7 machines many times — at least $5 \cdot 10^7$ each on PowerPC G5, $8.5 \cdot 10^8$ on POWER 6, and $2.3 \cdot 10^9$ on POWER 7 — and our model exploration tool terminated in less than 16GB memory for 10455 tests. We also have relatively thorough testing of `eieio`, with 235 tests run on POWER 6 and 7 and in the model. For all the above, all outcomes observable on the hardware are allowed by our model; in this sense the model is experimentally sound, modulo only the reservation granule issue described above.

Representative data is below (with the full dataset online).

	Model	Power6	Power7
MP+lwsync+porr	Forbid	Ok, 0/2.6G	Ok, 0/6.1G
MP+poaa+addr	Allow	Ok, 27k/1.1G	Ok, 56/1.4G
2+2W+rfipr-datarps	Allow	Ok, 58/545k	No, 0/4.4G Allow unseen

3.2 Restoring SC with FNOs and STAs

Despite the relaxed nature of POWER load-reserve/store-conditional (as illustrated by the first two MP examples of §2), if one replaces *all* loads and stores by FNO and STA respectively, one regains SC behaviour:

THEOREM 1. *Mapping all accesses to FNOs for loads and STAs for stores restores SC.*

Proof Outline: Note first that the successful store-conditionals (SSC) of an FNO or STA reach their coherence points as soon as they are committed. Thus there is a total order over all SSCs of a program (not just a per-location order), following the abstract-machine order in which the writes of this program reach their coherence points (we call this order rcp). Of course, this order is only over successful write-conditionals, not plain writes. Two SSCs in program order are committed in that order, thus reach their coherence point in that order, so rcp respects program order. Moreover, if all accesses are mapped to FNO and STA, we know that the threads communicate only through SSCs that are read by load-reserves and that a load-reserve before an SSC must read the (here unique) coherence-maximal value.

4. C/C++ Locks, RMWs, and Fences

Historically, the C and C++ standards did not cover concurrency, considering it to be a library issue. That is not a satisfactory position, as observed by Boehm [Boe05], and the recent ISO C11 and C++11 revisions incorporates concurrency for the first time [Bec11, ISO11].

The design, as outlined by Boehm and Adve [BA08], is based on a data-race-free (DRF) [AH90] model for normal code: for a program written using threads, mutexes and sequentially consistent *atomic*[1] operations, if it has no races in any execution then it is supposed to have SC semantics[2]; while if some execution does exhibit a race, the program's behaviour is undefined (and an implementation is entirely unconstrained). This permits a broad range of compiler optimisations [Šev11] and allows implementation on relaxed-memory multiprocessors without requiring expensive barriers or special load and store instructions for every memory access. However, implementing SC atomic operations can still be expensive, and therefore the language also includes a variety of *low-level atomics* for high-performance concurrent algorithms: atomic reads, writes, and read-modify-write operations with weaker properties that are cheaper to implement. They are annotated with a *memory order*: either *relaxed*, providing no synchronisation; *release* (for writes and RMWs), *acquire* (for reads and RMWs), or *release_acquire* (for RMWs), providing synchronisation for message-passing idioms; or *consume*, a weaker variant of read-acquire that lets programmers take advantage of the fact that POWER and ARM respect data and address dependencies at little or no cost. SC atomics also have release/acquire semantics. The language also includes release/acquire and SC *fences*.

[1] In C/C++11 terminology, *atomic* plays a very loosely similar role to Java *volatile*: in C/C++11, for defining whether a program has undefined behaviour, atomic operations (which might be single loads, single stores, or read-modify-writes) are not deemed to have data races with other atomic operations, even if they are not separated in happens-before.

[2] In fact this is not quite true, as Batty et al. [BMO+12] observe, due to a subtlety involving atomic initialisation.

The standard is written in prose, subject to the usual problems of ambiguity, but we have produced a formal semantics for the concurrency model [BOS+11]. In discussion with members of the ISO WG21 concurrency subgroup, we identified various issues with earlier drafts of the standard, proposing solutions that are now incorporated into the standard; there is a close correspondence between the formal semantics and the C++11 text. The C standard has followed suit: at the time of writing, the C++11 concurrency model has been partially incorporated into C11, but some of the C++11 fixes are not yet incorporated. We highlighted these (together with a simplification to the model) at a recent WG14 meeting, which has registered them as defect reports for a future Technical Corrigendum.

In this section we explain enough of the C/C++11 memory model to support our discussion and proof of how it can be implemented above POWER in §5. We recall the mathematical structure of the Batty et al. semantics and explain the lock, read-modify-write, and fence synchronisation primitives informally. The basic semantics for these is not a contribution of this paper (a version was given in [BOS+11]) but the explanation here is new, and we have had to make three significant improvements to the semantics:

- We place the responsibility for correctly using mutexes on the programmer, in accordance with intuition and with the standard. Previous work, including [BA08, BOS+11], placed it instead on the compiler, imposing unrealistic requirements, e.g. that the implementation of unlock has to check that the mutex is currently held by the unlocking thread.
- We carefully specify how lock and read-modify-write operations can block owing to store-conditional operations that can fail continually on the Power Architecture.
- We add two missing cases to the fence semantics.

We discuss these in more detail below, and the mathematically rigorous version of the improved model is available in the supplementary material.

The C/C++11 memory model is axiomatic (quite different in style to our POWER abstract machine): presuming a threadwise operational semantics that defines candidate executions consisting of the sets of memory read and write actions for threads in isolation, it defines when such a candidate is *consistent* and whether it has a race of some kind. Consistency is defined in terms of several relations, including a *happens-before (hb)* relation that embodies the language's notion of causality.

The three basic relations, *sequenced-before (sb)*, *reads-from (rf)*, and *modification order (mo)*, are closely related to our POWER abstract machine's notion of program order, reads-from, and coherence order, respectively. Sequenced-before is a threadwise partial order that does not need to be total (for example, the operands `x` and `y` of `x == y` are not related by sequenced-before). Modification order is a per-location total order over the write operations to that location. A modification order is only needed for atomic locations; it can contain atomic writes with various memory orders, together with non-atomic initialisation writes.

The *synchronises-with (sw)* and happens-before relations are calculated from sequenced-before and reads-from. A write-release synchronises with a read-acquire that reads-from the write. Read-acquires can also synchronise with some (but not necessarily all) write-releases that appear in modification order before the read-from write; this is formalised by *release sequences*, which we do not detail here. Happens-before is then built as the transitive closure of synchronises-with and sequenced-before, with some exceptions relating to read-consumes which we do not detail here. Four coherence axioms require that happens-before is consistent with modification order. Synchronises-with and happens-before have no direct counterparts in POWER: the compilation of C/C++ to POWER

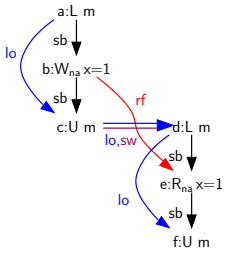
must insert enough barriers to ensure that synchronizing write/read pairs enjoy the right ordering properties (see §5).

The *sequential consistency (sc)* relation is a total order on all of the SC reads and writes. It is required to be consistent with sequenced-before, modification-order, and happens-before.

A program has a data race if there is some consistent execution with two happens-before unrelated reads/writes on the same location, where at least one is a write, and at least one is not atomic.

4.1 Locks

To model mutexes, we use three kinds of additional actions: locks, unlocks, and blocks. In a consistent execution, a total order *lock order (lo)* (analogous to modification order) over these ensures that no mutex is acquired when already held: between any pair of locks of the same mutex, there must be an unlock of that mutex. Furthermore, lock order must be consistent with happens-before. Lastly, the lock order introduces synchronisation from each unlock to later locks of the same mutex, ensuring that the actions in each critical section happen-before the actions in subsequent ones. In the example execution here, actions a and d lock mutex m, c and f unlock it, and the non-atomic writes b and read e are inside critical sections. The synchronisation between c and d ensures that e reads from b, and that e and b do not race.



These C/C++ candidate execution diagrams are analogous to the POWER execution diagrams of §2. They pick out one candidate execution of a program (the programs are in the supplementary material), and show the memory write and read actions and the lock and unlock actions of that candidate execution. Each action is annotated by the location and value, and, for memory reads and writes, the memory order parameter. The actions are arranged in vertical columns by thread. Among these actions, we depict the key C/C++ relations:

- sb edges show sequenced-before, in this control-flow unfolding (recall that sequenced-before is not necessarily total over the events of a thread);
- rf edges from write actions to read actions show that the read reads-from that write (as before, reads from an initial state are marked with a red dot for the source);
- mo edges show modification order, relating all write actions at an atomic location;
- sc edges show the SC order, a total order over all SC actions; and
- lo edges show the lock order, a total order over all lock and unlock actions.

From these, two derived relations can be calculated:

- sw edges depict the C/C++ synchronizes-with relation; and
- hb edges depict the C/C++ happens-before relation.

In previous work [BA08, BOS+11], a consistent execution also required that lock/unlock actions strictly alternate in lock order, and that the unlock of a mutex must follow the lock that acquired it in sequenced-before. Putting this requirement on consistent executions requires that the compiler or lock implementation enforce it, for example, by having each unlock call dynamically check that the mutex is held by the unlocking thread. To support efficient unlock implementations that have no such check, the C/C++11 standard places this requirement on the programmer, and so we allow consistent executions with improper calls to unlock, but give such programs undefined behavior.

Not all attempts to acquire a mutex eventually succeed: the program could be deadlocked, or a particular acquisition could be starved forever. To model this, on an attempted acquisition the threadwise operational semantics non-deterministically generates either a lock action, representing success, or a block action, representing deadlock or starvation. In the latter case, we require that the operational semantics not produce any further actions sequenced after the block. The concept of blocked mutex acquisition does not explicitly appear in the standard, or in prior work; we add it to properly separate the axiomatic memory model from the threadwise operational semantics. Because block actions appear in lock order, the memory model can place constraints on them to specify the exact fairness or liveness guarantees enjoyed by mutex acquisition. The operational semantics only needs to know whether a certain call is blocked or not. This contrasts with the typical operational treatment of mutexes in which locking and unlocking operations manipulate a piece of shared state; state which, in a relaxed memory setting, must be governed by the memory model.

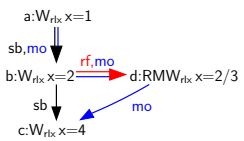
As we discuss in §2, the Power Architecture allows store-conditionals to fail arbitrarily, and there is no simple guarantee ruling out repeated failure (though implementations do go to great lengths to provide forward progress). In §5, we consider an implementation of C/C++11 mutexes based on POWER load-reserve/store-conditional instructions, and in this context a mutex acquisition can block indefinitely, even in the absence of deadlock or contention, due to continual, spurious store-conditional failures. To model this possibility in C/C++11, we allow block actions to appear anywhere in the lock order, without constraint. However, if one were modelling a particular POWER implementation that guaranteed store-conditional liveness, one could use a stronger C/C++11 model: in lock order, each block is either preceded by a lock which is never unlocked (the deadlock case), or followed by an infinite number of locks (the starvation case).

To simplify our reasoning about the connection between POWER and C/C++11 mutexes in §5, we also use an alternative model that, first, uses a separate lock order for each mutex, and, second, requires that each unlock synchronise only with the next lock in that order. We have proved the equivalence of the two models (see the supplementary material); the equivalence relies on having undefined behaviour for programs that misuse locks.

4.2 RMWs

C/C++11 has three kinds of atomic RMWs: `fetch_add` and similar, `compare_exchange_strong`, and `compare_exchange_weak`. The first updates the value at a given address, the second is a traditional CAS, and the third is a traditional CAS except that it can fail spuriously. The first two are implemented on POWER with load-reserve/store-conditional pairs with loops to account for the store-conditional failing. Thus, like mutex acquisition, the operational semantics must allow them to block. The last is implemented with a simple load-reserve/store-conditional pair (which exposes the possibility of non-deterministic spurious failures).

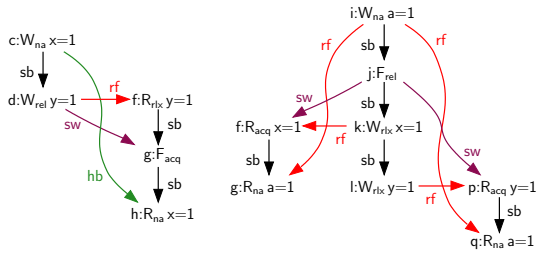
Successful RMWs are modelled by an RMW action that both reads and writes; atomicity is guaranteed by requiring that the read reads-from the immediate predecessor of the write in modification order. Thus, in the example to the right, RMW-relaxed d must read from write-relaxed b, and not write-relaxed a.



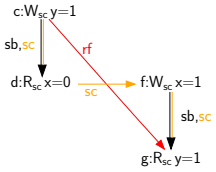
4.3 Fences

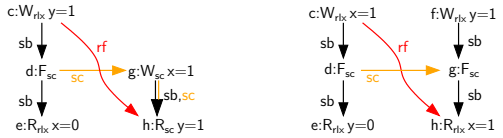
In C/C++11, fences are used to replace multiple acquire/release or SC operations with more efficient relaxed ones. There are four kinds: acquire, release, acquire/release, and SC fences.

Acquire and release fences establish synchronises-with relationships, even when the reading and writing actions are relaxed. In the message-passing example on the left below, the acquire fence g synchronises with write-release d even though the read f is relaxed; this synchronisation ensures that non-atomic write c happens before non-atomic read h. This might help performance: for example, if f were in a loop waiting to observe d's write; we only have to pay for the acquire once, rather than on each iteration. In the variation on the right, a single release fence j ensures synchronisation based on multiple relaxed writes k and l.



SC fences appear in the sc relation, and enforce a set of coherence-like axioms between it and modification order. In the store-buffering (or Dekker's) example on the right, SC atomic reads and writes are used to ensure that at least one of the reads d or g reads-from one of the writes c or f. We can get the same guarantee by making one thread's operations relaxed and putting an SC fence in between, or even making all four operations relaxed and using two SC fences, as below.





Despite their name, C/C++11 SC fences are not designed to restore sequential consistency to programs that use low-level atomics (in part because in the Itanium architecture there is no implementation fence that would do so [Int02]). For example, the IRIW litmus test from [BA08] with two SC fences is allowed in C/C++ (in the model, the 6 SC fence conditions do not impose any constraint on the *sc* order between two SC fences because of a non-SC-atomic read before one fence and a read after another).

The proof of our main result in §5, together with the preliminary results of [BMO+12], are arguably the first extensive validation of the C/C++11 model. To the best of our knowledge, large-scale software development of concurrent C/C++11 code has not yet begun, as that practical validation requires production compilers which are only just becoming available. Even when they are, and are correct, they may provide over-strong models, so at present proof is the only way to determine whether code is correct with respect to the specification rather than with respect to some particular implementation.

5. Compiling Synchronisation Primitives: from C/C++ to POWER

We now discuss our second main contribution, the correctness proof of one mapping of the principal C/C++ synchronisation constructs —locks, read-modify-writes, and fences— to POWER, using load-reserve/store-conditional and the POWER barriers. The proof can be easily adapted to show related mappings, with different barrier placement on C/C++ SC operations, or SC atomics implemented with load-reserve/store-conditional, correct as well. This subsumes the result of Batty et al. [BMO+12], which just covered C/C++ loads and stores (with their various possible memory orders).

The mapping we consider combines the scheme proposed by McKenney and Silvera [MS11] for C/C++11 atomics, fences and RMWs with the spinlock implementation given in the Power Architecture [Pow09, p.717,718]. We first give the mapping with some informal intuition about its soundness (focussing on locks) before describing the main ideas of our proof.

5.1 Compiling from C/C++ to POWER

Locks Assuming that register `r3` contains a lock location, `r4` contains the value signifying that a lock is free and `r5` the value signifying that a lock is taken, the POWER implementation of a spinlock is:

```
loop:
  lwarx r6,0,r3,1 # load-reserve lock [r3] into r6
  cmpw r4,r6      # go to wait if lock not free,
  bne- wait       #  eventually returning to loop
  stwcx. r5,0,r3  # store-cond to try to set lock
  bne- loop       # loop if lost reservation
  isync           # import barrier
```

The implementation of unlock is:

```
  lwsync          # export barrier
  stw r4,0(r3)    # normal store to release lock
```

In the lock implementation, a load-reserve/store-conditional pair works on the lock location. The load is followed by a check on the value stored at the lock location; if the lock is not free, the code branches to a waiting routine (which might be a loop doing a normal read on the lock location, or a backoff, and which will return to `loop` to try to establish a new reservation). Otherwise the code tries to set the lock with a store-conditional. As discussed in §2, this will fail if the new value cannot be made an immediate coherence successor of the write read by the load-reserve (e.g. if some other thread has taken the lock in the meantime), and in that case it again returns to `loop`, otherwise the lock has been taken successfully and atomically. The implementation of unlock has a simple store at the lock location writing the value that signifies that the lock is free.

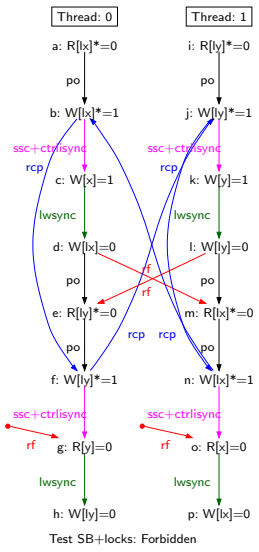
As a result, a C/C++ unlock/lock synchronisation is, at the POWER ISA level, a store-to-load communication. For this to truly be a synchronisation, the implementation also must include additional protection. The unlock store is preceded by an `lwsync` instruction. The effect of this barrier is first to restrict the reordering of instructions in its own thread: an `lwsync` must be committed after any program-order-preceding memory-access instruction, and before any memory-access instruction following in program order is satisfied (for reads) or committed. The second effect is an ordering of the propagation of stores. For any `lwsync` barrier, our POWER model keeps track of the set of stores and barriers that have been done by or propagated to its thread before it was committed. Before the `lwsync` can be propagated to another thread tid, all the members of this set, called its Group A, must first be propagated to tid. In the context of the unlock/lock synchronisation, this guarantees that any store visible to the unlocking thread before its unlocking store (to the lock location) must also be visible to the locking thread, before the load-reserve (of the successful store-conditional/load-reserve pair) on the lock location.

The propagation ordering enforced by an `lwsync` is transitive, in the following sense. Consider a chain of unlock/lock synchronisations, with each unlock directly following a lock on the

same thread, and each lock directly following an unlock on a different thread. This corresponds to a chain of thread-crossing store/load communications. In this communication, the stores come from the unlocking stores to the lock location, and the successful store-conditionals of lock acquires, while the loads correspond to the load-reserves at the lock location from the successful store-conditional/load-reserve pair of locks. For the communication to occur, each store must propagate to the thread of its associated load. If the first store of the chain is preceded by an `lwsync`, then this barrier along with its Group A must propagate to the second thread of the chain before the store it precedes. Since a load-reserve/store-conditional pair must be committed in order, the `lwsync` and its Group A stores are therefore propagated to the second thread before the store-conditional of that second thread is committed. Hence for this second thread's store-conditional to propagate to the third thread of the chain, the first thread's barrier along with its Group A must first propagate to that third thread. The argument continues to the rest of the chain.

Additionally, the last branch instruction of the lock implementation is followed by an `isync` instruction. As a result of this combination of a dependency from the result of the store-conditional to a conditional branch, followed by an `isync` (together, a kind of *ctrlisync* dependency) any program-order-following loads cannot be speculated before the store-conditional succeeds. Hence, any program-order-following loads must see the stores that were performed by the unlocking thread, before the unlock was executed.

A fully locked POWER program is SC The intuition above for locks is applicable in a wider context than C/C++. For a simple result along those lines, consider a POWER assembly program (not necessarily compiled from C/C++), which has all competing accesses (those accessing the same location, from distinct threads, at least one of them being a write) protected by locks, using a distinct lock variable per shared-memory variable (denoted below by lx, ly, etc., for locks protecting variables x, y, etc.). We assume all locations are disjoint and of the same size, and ignore the case of overlapping accesses.



Using the `isync` at the end of the lock sequence and the `lwsync` at the beginning of the unlock sequence, we get that two accesses in program order guarded by locks follow the order in which the SSC of their lock primitives reach coherence point. For example in the test SB+locks on the right, the write c to x on Thread 0 is committed after the SSC b to lx reaches coherence point, as depicted by the magenta ssc+ctrlisync arrow.

The lock write to ly on Thread 0 reaches coherence point before the lock write to ly on Thread 1, as depicted by the blue arrow rcp from Thread 0 to Thread 1, thus restoring SC for this test. Fleshing out this line of reasoning proves our next theorem:

THEOREM 2. *Protecting access to each address with an associated lock restores SC, assuming the program does not use distinct overlapping locations.*

RMWs The implementation of C/C++ RMWs also relies on load-reserve/store-conditional pairs. Here, without loss of generality, we only discuss the implementation of one kind of RMW, the "fetch_add" operation with an implementation as in the first §2 example: a load-reserve/store-conditional pair surrounding an addition instruction, the whole wrapped in a loop waiting for the store-conditional to succeed. The implementations of other kinds of RMWs use different instructions between the load-reserve/store-conditional pair but have essentially the same correctness argument.

RMWs are parameterised by a memory order, and the mapping places POWER barriers at the beginning and dependencies at the end of the load-reserve/store-conditional block, in the same style as the implementations of non-RMW C/C++ atomic accesses. The implementation of a relaxed RMW has no additional barriers; the implementation of a release RMW is preceded by an `lwsync` barrier; that of an acquire RMW is followed by a ctrlisync dependency; acquire/release combines both; and that of an SC RMW is preceded by a `sync` and followed by a ctrlisync dependency.

Fences C/C++ fences are compiled to single barrier instructions: SC fences are mapped to a `sync` and all the others to `lwsync`.

non-RMW atomics For non-RMW atomic memory accesses the compilation maps each C/C++ read or write to corresponding POWER instruction together with protecting barrier instructions, as for the RMWs above but with the addition of *consume* loads, which have no barrier but require the compiler to preserve dependencies [MS11, BMO$^+$12].

5.2 Correctness proof

Our second main result is the correctness of this compilation scheme. To focus on the concurrency issues, we abstract from the details of thread-local compilation: we consider an arbitrary non-optimising compiler (preserving memory accesses, program order and thread-local dependencies) that uses this compilation scheme for the concurrency primitives. Informally, our result is as follows; the formal statement and proof are in the supplementary material.

THEOREM 3. *Consider a non-optimising compiler that uses the mapping above. For any race-free C/C++ program, compile it to POWER and take any execution trace of our POWER abstract machine. Then there exists a C/C++ execution that is both equivalent to the POWER trace and accepted by the C/C++ memory model as a consistent execution of the original C/C++ program.*

This is in the same style as the result of Batty et al. [BMO$^+$12] for the compilation mapping in the absence of locks, fences and RMWs. We now outline the overall proof structure, common to both works, and then show how to extend it to handle the synchronisation primitives.

The proof describes a procedure for constructing a C/C++ execution from any POWER trace that guarantees their equivalence, and then shows the consistency of that C/C++ execution. But this construction does not work for racy programs. As a straightforward example, consider a C/C++ program with two threads, the first doing a non-atomic write at a location x, the second doing a non-atomic read on x. The C/C++ memory model rules out any consistent execution with the read reading from the write, because they are unrelated in C/C++ happens-before, and thus the execution would be racy. But in the compiled POWER code, which has no direct analogue of the C/C++ happens-before, nothing prevents the corresponding store propagating to the second thread to be read by the load. For the verification of the consistency of the reconstructed C/C++ execution to succeed, the following consequence of race-freedom is required: "in the C/C++ execution, a read cannot read-from a write that it is not related to by happens-before". The proof splits on whether this condition holds. If it does hold, consistency of the reconstructed execution can be directly shown, while if it does not hold, it can be shown that the original C/C++ program must have been racy, thus leading to a contradiction.

The latter proof is technically involved because one needs to build a racy, consistent C/C++ execution that does not exactly correspond to the POWER trace. There are two areas where this part of Batty et al.'s proof could fail when adding synchronisation constructs: in the top-level case split, and in the way that POWER-level speculation complicates the construction of the racy C/C++ execution. In fact, our alternate C/C++ formalisation of locks (§4.1) removes the need for modifying the case split: the new proof has a similar case split (on whether unlocks are used properly), but it is simpler because the proof is about the C/C++ model only, not in combination with POWER. Furthermore, careful modelling of the absence of store-conditional forwarding (§2) means that the construction in the existing proof works unchanged.

Returning to showing consistency of a reconstructed execution, several properties regarding the interaction of the happens-before relation with other relations defined by the C/C++ memory model have to be verified of the constructed C/C++ execution. To directly exploit our POWER model, the statements of all these properties are translated into equivalent statements regarding the POWER trace from which the C/C++ execution was built: we define relations on the elements of POWER traces and show a correspondence between these and the C/C++ relations. For most relations, clear counterparts exist. For example, the C/C++ *sequenced-before* corresponds roughly to the POWER program order (though some care needs to be taken, as sequenced-before is a partial order while the program order is total). However, there is nothing that corresponds directly to the happens-before relation. Instead, a more complex POWER relation is built from the other POWER relations.

Using this characterisation, a *propagation lemma* states that for any pair of C/C++ actions a and b related by happens-before, the instruction corresponding to b sees any stores that the instruction corresponding to a is able to see: any store propagated to the thread of a before a is executed, is also propagated to the thread of b before b is executed (a read is executed at the latest abstract-machine transition in which it is satisfied — all previous reads having been restarted — while a write is executed when it is committed). Using this lemma in conjunction with the preconditions and actions of the POWER model transitions, the conditions required by the C/C++ memory model can be verified. For example, when the POWER storage subsystem accepts a write request, one of its actions is to update the coherence order so that the store being accepted is coherence-after any store already propagated to its thread, which is needed when verifying one of the coherence properties required by C/C++.

Finally, a total order for SC actions must be built from the POWER trace, but as for happens-before there is no direct analogue of this relation in POWER. Instead, we show that the POWER `sync` barrier associated with SC actions by the mapping ensures that we never get cyclic dependencies if we take into account all the required properties of this order (e.g., a read may not read from a write after in SC order).

Including the synchronisation primitives The addition of the synchronisation primitives affects the definition of the happens-before relation. In particular, RMWs make the definition of release-sequences more complicated, and locks and fences introduce new ways for a synchronises-with edge to appear. This in turn affects the definition of the POWER analogue to happens-before. Here is the new characterisation of the transitive part of happens-before with synchronisation primitives, using relations derived from a POWER trace:

$$\begin{array}{l}\big((sync_t \cup lwsync_t)^{\text{refl}};\\ coi_t{}^{\text{refl}};\\ rfe_t;\\ (rmw_t; rf_t)^*;\\ (ctrlisync_t \cup dd_t{}^{\text{refl}} \cup lwsync_t)^{\text{refl}}\big)^+\end{array}$$

where semicolon denotes relational composition, and $\cdot^{\text{refl}}$ and $\cdot^+$ are respectively the reflexive and transitive closures. The full potentially non-transitive happens-before relation is obtained by taking the union with program order, but that does not present any difficulty for the proof. The $sync_t$ and $lwsync_t$ relations relate commit transitions of memory accesses corresponding to memory instructions from a single thread which are separated in program-order by, respectively, a `sync` or `lwsync`. This part of the characterisation corresponds to the barriers introduced by the compilation mapping before unlocks, and release atomic-stores, fences, and RMWs. The coi_t relation relates commit transitions of stores from the same thread which are related in the coherence order. In the absence of RMWs, this relation is the POWER analogue of the C/C++ release-sequence relation, and the rfe_t relation (relating the commit transition of a store to the commit transition of a load reading from that store from a different thread) was the next component. With RMWs, we need something more elaborate: the rmw_t relation identifies the presence of a load-reserve/store-conditional pair. The rf_t relation extends rfe_t by also allowing the store and load to be on the same thread. The composition $(rmw_t; rf_t)^+$ corresponds to a chain of RMWs ending a C/C++ release sequence. Note that they may be RMWs anywhere in a release-sequence, not only at the end, but in that case the RMW chain must come back to the thread of the release head, and then the last RMW is related to the release head by the coi_t relation. The last component of the characterisation is the union of $ctrlisync_t$ (the ctrlisync analogue of the $sync_t$ and $lwsync_t$ relations, dd_t (data dependency, for C/C++ consume atomics, included in our proof but not discussed here) and $lwsync_t$. This corresponds to the barriers or dependencies placed after acquire and consume reads or to acquire fences. The $lwsync_t$ here is added, corresponding to acquire fences.

Despite the expanded definition of the analogue of happens-before, the key abstraction of the propagation lemma still holds. Thus the proof of consistency of the reconstructed C/C++ execution extends smoothly to the addition of C/C++ locks, RMWs, and fences. There is one new condition to check for the consistency of a C/C++ execution: an RMW action must read from its last predecessor in the modification-order. In terms of the POWER implementation, this means that the load-reserve must read from the last coherence predecessor of the store-conditional — but that is just what we are guaranteed from the rules given in §2, for a successful store-conditional.

Finally, SC fences must be a part of the sc relation, and this interacts with modification order to impose 6 additional coherence-like conditions. As before, there is no obvious order in the POWER machine trace with the right properties. Instead, we build the sc relation from a POWER machine trace by directly checking that these 6 conditions are not violated. To take one example, having the sc relation between the SC fences go one way in the store-buffering example of §4 imposes a condition that at least one read-from edge cannot be from a write that is too old in modification order. Recalling that SC fences are mapped to the POWER `sync` barrier, we use a property of this barrier (that it must wait for all group A writes, or coherence successors thereof, to be propagated to all threads) to guarantee this condition. Indeed, the corresponding example in POWER is SB+syncs, which only has SC behaviour.

Discussion: C/C++ and POWER Our proof establishes a close correspondence between C/C++ executions and POWER abstract

machine executions of the compiled program. Nevertheless, there are programs with different behaviours on the two models. As noted previously, IRIW with two SC fences is allowed in C/C++, whereas its compilation with `syncs` is forbidden, and sometimes the mapping imposes stronger barriers than strictly necessary. For example, consider the 2+2W test of [SSA+11]. Here the only way to regain SC behaviour in C/C++ is to either use SC fences or all SC accesses everywhere. The test maps to 2+2W+syncs, which is indeed forbidden in POWER, but in fact here `lwsyncs` would have been enough.

6. Conclusion

With this work, together with [SSA+11, BOS+11, BMO+12], we finally have a semantics for real-world concurrent programming in C/C++ and on POWER multiprocessors that is complete enough to support reasoning about real OS and VM kernel code, at least in the absence of mixed-size accesses, and that is sufficiently well validated that one can have confidence in the results.

This opens the door to future work on verification techniques and analysis tools for such code. For example, looking informally at Michael's lock-free datastructure algorithms using hazard pointers [Mic04] (one of which was verified in an SC setting using separation logic [PBO07]), one can consider verification of efficient *implementations* of those abstract concurrent algorithms, either at the POWER ISA level, where one needs the various POWER barriers and load-reserve/store-conditional, or, making use of our main result here, at the C/C++ level, where one needs to add a particular placement of C/C++ fences and RMWs (interestingly, the two appear to coincide here: the C/C++ concurrency model is sufficiently expressive that one can express exactly the low-level synchronisation needed, as one would hope. Understanding how to do such verifications, especially compositionally, is a major open problem, as is the question of how to determine the best such placement. As we have seen, our work is also identifying subtle architectural choices.

We believe that the ARM architecture is broadly similar to POWER in these respects (though it is not identical to the model we present here), and hence that it will be possible to adapt the proof to ARM. We are currently engaged in further testing and in discussion with ARM staff to establish confidence there.

Acknowledgements We acknowledge funding from EPSRC grants EP/F036345, EP/H005633, EP/H027351, and EP/G026254/1, and from ANR project WMC (ANR-11-JS02-011), and thank the anonymous reviewers for their comments.

References

[AH90] S. V. Adve and M. D. Hill. Weak ordering — a new definition. In *Proc. ISCA*, 1990.

[AM11] J. Alglave and L. Maranget. Stability in weak memory models. In *Proc. CAV*, 2011.

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

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

[Bec11] P. Becker, editor. *Programming Languages — C++*. 2011. ISO/IEC 14882:2011. A non-final but recent version is available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`.

[BMO+12] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Proc. POPL*, 2012. `http://www.cl.cam.ac.uk/~pes20/cppppc/`.

[Boe05] H.-J. Boehm. Threads cannot be implemented as a library. In *Proc. PLDI*, 2005.

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

[cpp] Supplementary material. `http://www.cl.cam.ac.uk/users/pes20/cppppc-supplemental`.

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

[Her93] M. Herlihy. A methodology for implementing highly concurrent data objects. *TOPLAS*, 15(5):745–770, Nov 1993.

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

[ISO11] *Programming Languages — C*. 2011. ISO/IEC 9899:2011. A non-final but recent version is available at `http://www.open-std.org/jtc1/sc22/wg14/docs/n1539.pdf`.

[JHB87] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. (Technical Report UCRL-97663), Nov 1987.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, 1979.

[McK11] P. E. McKenney. [patch rfc tip/core/rcu 0/28] preview of rcu changes for 3.3, November 2011. `https://lkml.org/lkml/2011/11/2/363`.

[Mic04] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15:491–504, June 2004.

[MS11] P. E. McKenney and R. Silvera. Example POWER implementation for C/C++ memory model. `http://www.rdrop.com/users/paulmck/scalability/paper/N2745r.2011.03.04a.html`, 2011.

[OBZNS11] S. Owens, P. Böhm, F. Zappa Nardelli, and P. Sewell. Lem: A lightweight tool for heavyweight semantics. In *Proc. ITP, LNCS 6898*, 2011. "Rough Diamond" section.

[PBO07] M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *Proc. POPL*, 2007.

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

[Šev11] J. Ševčík . Safe optimisations for shared-memory concurrent programs. In *Proc. PLDI*, 2011.

[SS88] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *TOPLAS*, 10:282–312, 1988.

[SSA+11] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI*, 2011.