Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER

Mark Batty[1] Kayvan Memarian[1,2] Scott Owens[1] Susmit Sarkar[1] Peter Sewell[1]

[1]University of Cambridge [2]INRIA

{first.last}@cl.cam.ac.uk

Abstract

The upcoming C and C++ revised standards add concurrency to the languages, for the first time, in the form of a subtle *relaxed memory model* (the *C++11 model*). This aims to permit compiler optimisation and to accommodate the differing relaxed-memory behaviours of mainstream multiprocessors, combining simple semantics for most code with high-performance *low-level atomics* for concurrency libraries.

In this paper, we first establish two simpler but provably equivalent models for C++11, one for the full language and another for the subset without consume operations. Subsetting further to the fragment without low-level atomics, we identify a subtlety arising from atomic initialisation and prove that, under an additional condition, the model is equivalent to sequential consistency for race-free programs.

We then prove our main result, the correctness of two proposed compilation schemes for the C++11 load and store concurrency primitives to Power assembly, having noted that an earlier proposal was flawed. (The main ideas apply also to ARM, which has a similar relaxed memory architecture.)

This should inform the ongoing development of production compilers for C++11 and C1x, clarifies what properties of the machine architecture are required, and builds confidence in the C++11 and Power semantics.

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

General Terms Languages, Reliability, Standardization, Theory, Verification

Keywords Relaxed Memory Models, Semantics

1. Introduction

Most work on semantics and reasoning for shared-memory concurrency has assumed a sequentially consistent (SC) memory [Lam79], in which a single shared memory is acted upon by interleaved threads. In practice, however, mainstream multiprocessors (x86, Sparc, Power, ARM, Itanium) provide weaker and more subtle *relaxed* memory models, to permit various hardware optimisations. Mainstream concurrent programming languages (e.g. Java, C, C++) also provide relaxed memory models, although of rather different kinds, both to permit compiler optimisations and so that they can be compiled to those processors without excessive use of hardware memory barriers. Moreover, for several of those multiprocessors and languages, the actual memory model provided has long been poorly specified and not well understood.

In recent years that situation has improved. Of particular relevance here, the ISO C++ Standards Committee (JTC1/SC22/WG21) has introduced a memory model, as outlined by Boehm and Adve [BA08], into their revised standard (C++11) [Bec11], and it is expected that the upcoming revision of the C standard (informally, C1x) will adopt essentially the same model. This standard is written in prose, subject to the usual problems of ambiguity and not directly testable, but in previous work we produced a formal semantics for C++11 concurrency [BOS+11]. In discussion with members of the WG21 concurrency subgroup, we identified various issues with earlier drafts [BOS+10], proposing solutions that are now incorporated into the new ISO C++11 standard; the result has a close correspondence between the formal semantics and the standard prose.

The C++11 memory model is an axiomatic one: presuming a threadwise operational semantics that defines candidate executions consisting of the sets of memory read and write events 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 a happens-before relation. It is a data-race-free (DRF) model [AH90]: for a program that has no race in any consistent execution, the semantics is the set of all such, while other programs are undefined (and implementations are unconstrained). The design is stratified: there is a sublanguage with sequentially consistent atomic operations which is intended to have simple semantics for common use, then additional release/acquire and relaxed atomic operations (needing less hardware synchronisation to implement) for expert use in high-performance concurrency libraries and system code, and finally release/consume atomics, for maximum performance on architectures such as Power and ARM where dependencies provide useful ordering guarantees. The whole is relatively complex — for example, to accommodate release/consume atomics, it involves a happens-before relation that is, by design, not necessarily transitive.

The first contribution of this paper is to simplify the C++11 model, proving that it is equivalent to a simpler model M_1, without the technical concept of "visible sequences of side effects". Subsetting the language to remove release/consume atomics gives a further simplification, M_2, with a transitive happens-before relation. Subsetting still further, to the fragment without low-level atomics, one would like to know that that model is equivalent to a sequentially consistent model M_3. In Section 4 we show that this holds for

POPL'12, January 25–27, 2012, Philadelphia, PA, USA

programs that are race-free in M_2, but that the M_2 and M_3 notions of race can differ, with an example involving atomic initialisation; we give additional conditions under which the two do coincide.

Meanwhile, on the hardware side, Sarkar et al. [SSA+11] have established a rigorous memory model for Power multiprocessors, based on extensive testing and discussion with IBM staff. Power has a more relaxed and more subtle model than Sparc TSO or x86-TSO, but it and ARM are similar in this respect, and preliminary testing and discussion with ARM staff suggests that the same model, with minor differences, applies there (Itanium is also quite relaxed but rather different). This model is of a very different kind to that for C++11: it deals with the Power dependencies and barriers (`sync`, `lwsync`, `isync`) rather than the C++11 atomic primitives; it gives semantics to all programs, not just race-free programs; and the presence of speculative execution is observable to the programmer on these machines. Accordingly, this model is in an abstract-machine style rather than an axiomatic style: it defines labelled transition systems for a thread semantics (with explicit speculation) and for a storage subsystem (abstracting from the cache hierarchy), with interactions between the two.

Between programming language and processor, how can one compile a programming language with the C++11 memory model to Power or ARM? There are three main parts to this question: one has to consider how each C++11 atomic operation is compiled to assembly code, perhaps with barriers and/or preserved dependencies; when conventional compiler optimisations are legal in this setting; and what new optimisations are needed, e.g. to remove redundant barriers. In the relaxed-memory-concurrency setting, all of these are sufficiently delicate that it is hard to have any confidence in correctness without proof. We focus here on the first.

A naive compilation scheme would be to insert the heavyweight Power `sync` barrier between every C++11 memory access. Even here, it is not at all obvious that this would be a correct implementation (indeed, on Itanium, inserting the `mf` barrier between every access is not sufficient to regain SC), and for a realistic implementation, with good performance, it is essential not to introduce unnecessary synchronisation. McKenney and Silvera have proposed a mapping of the core C++ atomic primitives to Power assembly [MS11]. For the different flavours of (nonatomic and atomic) loads and stores, this is:

C++11 Operation	Power Implementation
Non-atomic Load	`ld`
Load Relaxed	`ld`
Load Consume	`ld` (and preserve dependency)
Load Acquire	`ld; cmp; bc; isync`
Load Seq Cst	`sync; ld; cmp; bc; isync`
Non-atomic Store	`st`
Store Relaxed	`st`
Store Release	`lwsync; st`
Store Seq Cst	`sync; st`

This compiles some C++11 operations to normal Power loads and stores, some with a Power lightweight synchronisation barrier `lwsync` or heavier `sync`, and in some cases uses just a combination of an artificial control dependency (of a compare `cmp` and conditional branch `bc`) and an `isync`.

Our second contribution is to prove the above mapping correct. While the mapping involves only a few assembly instructions, its correctness involves much of the behavioural subtlety of the C++11 and Power relaxed memory models, and is far from obvious. Indeed, the mapping above has been updated in the light of our comments following Sarkar et al. [SSA+11]: an earlier proposal included an optimisation that we observed is not sound w.r.t. the Power architectural intent (we give two counterexamples here). Our proof relies on the simplified general C++11 model M_1 from the first part. Along the way we obtain results about the Power architecture of independent interest, e.g. showing that (and why) inserting a `sync` between all accesses does restore SC. We also prove soundness of an alternative mapping, with the SC atomics implemented as below — the "trailing `sync`" convention, suggested (in the ARM context, with DMB in place of `sync` and `lwsync`) by Boehm [Boe11]:

C++11 Operation	Power Implementation
Load Seq Cst	`ld; sync`
Store Seq Cst	`lwsync; st; sync`

Depending on the relative frequencies of SC loads and store, and on the costs of the different hardware barriers (which may vary significantly from one architecture to another, and also between implementations), this may have better performance. Note that for code produced by different compilers to correctly interoperate, they must all make the same choice of mapping, which should be part of the ABI.

We also show that both these mapping are local optima: weakening any of the clauses (e.g. replacing a `sync` by a `lwsync`) yields an unsound scheme.

In the first contribution we covered the full C++11 primitives as in [BOS+11], but for these correctness results we focus just on the load and store operations in the tables above, omitting dynamic thread creation, C++11 fences, read-modify-writes, and locks. The first two are minor simplifications, but the last two are substantial questions for future work, as their Power implementations involve the load-reserve/write-conditional instructions, for which we do not yet have a well-established relaxed-memory semantics.

To make this paper as self-contained as possible, we give a brief introduction to our source and target starting points, the C++11 [BOS+11] and Power [SSA+11] models, in Sections 2 and 6 respectively. Section 3 describes our simplifications of C++11 and Section 4 the atomic initialisation issue; Section 5 discusses the correctness of the above C++11-to-Power compilation schemes informally, looking at specific examples; and Section 7 discusses our correctness statement and proof that they are correct in general. We discuss related work and conclude in Sections 8 and 9.

We focus throughout on the key ideas, as it is not possible to include full details here (the model definitions alone for C++11 and Power are around 15 and 50 pages of typeset mathematics), but our definitions and proofs are available on-line [BMO+]. Our definitions are expressed in Lem [OBZNS11], a lightweight language of machine-typechecked type, function, and relation definitions, from which one can generate LaTeX, prover definitions in HOL4, Isabelle/HOL, and (in progress) Coq, and executable OCaml code. Our proofs are a combination of machine-checked interactive HOL4 proof and rigorous hand proof (with auxiliary definitions and lemma statements in Lem).

This is a domain in which rigorously proved results are of direct interest to mainstream industrial software development, which typically relies almost entirely on testing for assurance. The nondeterminism of relaxed memory multiprocessors makes effective random testing challenging, and the fact that the programming language and processor architecture specifications are looser than current implementations makes it insufficient even in principle. Compiler groups are now implementing C++11 concurrency, and we are in ongoing discussion with members of the Red Hat GCC and ARM groups on how this should be done correctly, based in part on the work we present here (e.g., one early-draft implementation was found to put the SC store `sync` on the wrong side of the store).

2. Background: C++11 concurrency

This section gives a general introduction to the C++11 concurrency primitives and their semantics. We refer the reader to [BOS+11,

Bec11, BA08] for a full discussion of many subtle points that we cannot cover here.

2.1 The Language: C++11 concurrency primitives

C++11 is a shared-memory-concurrency language with explicit thread creation and implicit sharing: any location might be read or written by any thread for which it is reachable from variables in scope. It is the programmer's responsibility to ensure that such accesses are *race-free*. For example, in the following program the spawned thread, executing `thread_body(&x)`, writes `x` while the `main` thread reads `x`, without any synchronisation between the two. These are *non-atomic* reads and writes, and this constitutes a *data race*; the fact that the program exhibits an execution containing a race makes the program *undefined* according to the standard: an implementation has freedom to behave arbitrarily (in particular, compilers can perform optimisations that are not sound for racy programs, which means that common thread-local optimisations can still be done without the compiler needing to determine which accesses might be shared).

```
#include <thread>
using namespace std;
void thread_body(int* p) {*p = 2;}
int main() {
  int x = 1;
  thread t1(thread_body, &x);
  int r = x;
  t1.join();
  return 0; }
```

The language provides several mechanisms for concurrent programming without races. First, accesses to shared state can be protected using mutex locks of various kinds, e.g. using `m.lock()` and `m.unlock()`. We then have various *sequentially consistent atomic* operations on objects of integral type (bytes, integer types, etc.): atomic loads, stores, and various read-modify-write operations. Atomic operations do not race with each other, by definition, so changing the above to use SC atomic operations for `x` gives a well-defined program, and the allowed behaviours are intended to be those one would naively expect in an SC semantics. The intention is that locks and SC atomics will suffice for most programmers. However, making all racy accesses sequentially consistent is not always required and can incur a significant performance cost, so C++11 also provides several *low-level* atomic operations (with different memory order parameters) for expert use, e.g. in concurrency libraries and other systems code. For message-passing idioms, in which one thread writes some (perhaps multi-word) data `x` and then a flag `y`, while another waits to see that flag write before reading the data, it suffices to make the flag write a *release* and the flag read an *acquire* atomic:

```
int x;
atomic<int> y(0);
// sender thread T0
x=1;
y.store(1,memory_order_release);
// receiver thread T1
while (0==y.load(memory_order_acquire)) {};
int r = x;
```

The synchronisation between the release and acquire ensures that the sender's write of `x` will be seen by the receiver, i.e., that the read of `x` must be from the write `x=1` rather than from some uninitialised value.

On multiprocessors with weak memory orders, notably Power and ARM, release/acquire pairs are cheaper to implement than SC atomics but still significantly more expensive than plain stores and loads (c.f. the preceding `lwsync` for the release and following `cmp; bc; isync` for the acquire in Section 1; we return later to how these act). However, these architectures do guarantee that certain dependencies in an assembly program are respected, and in many cases those suffice on the reader side. This motivates a *release/consume* variant of release/acquire atomics. For example, suppose one thread writes some data (again perhaps multi-word) then writes the address of that data to a shared atomic pointer, while the other thread reads the shared pointer, dereferences it and reads the data.

```
int x;
atomic<int *> p(0);
// sender thread
x=1;
p.store(&x,memory_order_release);
// receiver thread
int* xp = p.load(memory_order_consume);
int  r = *xp;
```

Here the combination of the release/consume pair and the dependency at the receiver from the read of `p` to the read of `x` suffices to ensure the intended behaviour.

Finally, in some cases one needs only very weak guarantees, and here `memory_order_relaxed` atomics can be used. The language also includes explicit fences, in acquire, release, acquire-release, and SC forms, to ease porting of older fence-based code.

2.2 Semantics: the C++11 axiomatic memory model

Traditional shared-memory operational semantics involves an explicit memory, with interleaved transitions by threads reading and writing that memory. That is intrinsically SC, and for a relaxed-memory language such as C++11 a quite different semantic style is needed.

We say a *candidate execution* ex of a C++ program comprises a set of memory actions together with various relations over them. The actions are of the form:

action ::=	
a:R_{na} x=v	non-atomic read
a:W_{na} x=v	non-atomic write
a:R_{mo} x=v	atomic read
a:W_{mo} x=v	atomic write
a:RMW_{mo} x=v1/v2	atomic read-modify-write
a:L x	lock
a:U x	unlock
a:F_{mo}	fence

Here a is a unique id (including a thread id), x a location, and v a value. Memory orders mo range over those mentioned earlier, abbreviated thus: SC, RLX, REL, ACQ, CON, and A/R.

The relations include four binary relations determined by a threadwise operational semantics: *sequenced-before* (sb), *additional synchronises with* (asw), *data dependency* (dd), and *control dependency* (cd); together with three that determine an interrelationship between memory actions of different threads: *reads-from* (rf) relates a write to all the reads that read from it; the *sequential consistent order* (sc) is a total order over all SC actions; and *modification order* (mo) is the union of a per-location total order over writes to each atomic location.

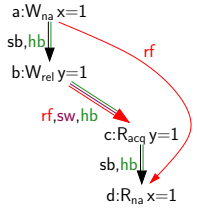
Given a candidate execution, the semantics defines various derived relations: *release-sequence*, *hypothetical-release-sequence* (a variant of *release-sequence* used in the fence semantics), *synchronizes-with* (sw), *carries-a-dependency-to* (cad), *dependency-ordered-before* (dob), *inter-thread-happens-before* (ithb), *happens-before* (hb), *visible-side-effect*, and *visible-sequences-of-side-effects*. These are used to define when a candidate execution is *consistent* and whether it contains a *race* of some kind: a *data race*, *unsequenced race*, or *indeterminate read*, which we collect into a predicate $race\ ex$. A data race is a pair of accesses to the same address, at least one of which is a non-atomic write, which are not happens-before related. The

consistent_reads_from_mapping check has subclauses as on the right below.

consistent_execution =
well_formed_threads ∧
consistent_locks ∧
consistent_inter_thread_hb ∧
consistent_sc_order ∧
consistent_modification_order ∧
well_formed_reads_from_mapping ∧
consistent_reads_from_mapping

consistent_reads_from_mapping =
consistent_non_atomic_read_values ∧
consistent_atomic_read_values ∧
coherent_memory_use ∧
rmw_atomicity ∧
sc_reads_restricted ∧
sc_fences_heeded

For a program c_prog for which all its consistent executions are race-free with respect to all three kinds of race (which we write as drf c_prog), the semantics is the set of all those consistent executions; other programs are undefined. For example, the release-acquire program excerpted above is race-free, and one of its consistent executions is shown below (suppressing the initialisation write i:W_{na} y=0). To give some flavour of the semantics, we describe why this execution is indeed consistent, referring the reader to [BOS$^+$11] for the detailed definitions. The *well_formed_threads* and *well_formed_reads_from_mapping* predicates are straightforward sanity conditions. For this simple example the inter-thread-happens-before (ithb) and happens-before relation (hb) coincide, and are just the transitive closure of the union of sequenced-before (sb) and the synchronised-with (sw) edge created by the release/acquire pair (b, c). The *consistent_inter_thread_happens_before* predicate checks that this is acyclic. The *consistent_locks* and *consistent_sc_order* checks are vacuous, as there are no lock or SC atomic operations in this example. The modification order (mo) is empty, as there is only one write to the atomic location, so *consistent_modification_order* is vacuous. For *consistent_reads_from_mapping*, the main subclause is *consistent_non_atomic_read_values*, which checks that the non-atomic read of x takes the value of a *visible side effect*, which in a race-free program is unique and is the most recent write in happens-before. Then *consistent_atomic_read_values* permits the atomic read of y to read from any element of a *visible sequence of side effects*, which is a set of writes of y, ordered by modification order, headed by a visible side effect and terminated before any write that happens-after the read. The *coherent_memory_use* does not contribute because the happens-before condition already enforces coherence here. The remaining three (*rmw_atomicity*, *sc_reads_restricted*, *sc_fences_heeded*) are vacuous, as there are no RMW operations or SC atomics.



3. Simplifying the C++11 model

In this section, we establish two simplifications of the C++ model of [BOS$^+$11]. The standard tries to give an intuitive definition of which values an atomic read might take using *visible sequences of side effects*. As Batty et al. [BOS$^+$11, BOS$^+$10] note, taken alone they are too weak, and additional coherence axioms (now incorporated into the ISO C++ Standard) are necessary to capture the intended semantics. Given those, we can prove that the concept of *visible sequences of side effects* is unnecessary, permitting a much simpler definition of *consistent_atomic_read_values* in a model M_1:

$$
\begin{array}{l}
consistent\_atomic\_read\_values_1 = \\
\quad \forall b{\in}actions. \\
\quad (\text{is\_read } b \ \wedge \ \text{is\_at\_atomic\_location } lk\ b) \ \Rightarrow \\
\quad\ (\text{if } (\exists a\_vse{\in}actions.\ (a\_vse,\ b)\ \in\ vse) \\
\quad\quad \text{then } (\exists a{\in}actions.\ ((a,\ b)\ \in\ rf)\ \wedge\ \neg\ ((b,\ a)\ \in\ hb)) \\
\quad\quad \text{else } \neg\ (\exists a{\in}actions.\ (a,\ b)\ \in\ rf))
\end{array}
$$

THEOREM 1. *A C++11 candidate execution is consistent in C++ iff it is consistent in M_1, and the races are identical.* [Proof: the happens-before relations coincide; mechanised in HOL4]

This has already been useful in automatic analysis [BWB$^+$11], as it removes the only use of an existentially quantified set of sets of actions in the semantics.

Our second simplified model, M_2, removes the complications introduced by atomic consume operations (included only for expert programmers on Power and ARM) for the subset of the language without them. The original C++ model and M_1 have a complex, non-transitive definition of happens-before, from the standard:

$$
\begin{array}{lcl}
ithb_1 & = & \text{let } r = sw \cup dob \cup (sw; sb) \text{ in } (r \cup (sb; r))^+ \\
hb_1 & = & sb \cup ithb
\end{array}
$$

If we subset the language to remove atomic consume operations, we can give an equivalent model M_2 with a much simpler

$$hb_2 = (sb \cup sw)^+$$

THEOREM 2. *A C++11 candidate execution without consume operations is consistent in M_1 iff it is consistent in M_2, and the races are identical.* [Proof: the two hb relations coincide; HOL4]

4. Atomic initialisation and SC semantics

Subsetting further to remove release/acquire and relaxed operations, leaving just nonatomics and SC atomics, one would like to prove a result along the lines of Boehm and Adve [BA08], showing that for such programs the M_2 semantics (and hence M_1 and Standard C++) and an SC-for-DRF-programs semantics are equivalent. Our third simplified model, M_3, has the obvious SC notion of consistent execution, with a total order over all operations, and requiring reads to read the most recent writes in that. For defining data races, it uses the same style of definition as C++11, using a happens-before relation, but calculates that from the total order. To prove the result, we would need to show that a program is race-free in M_2 iff it is race-free in M_3, and, for any race-free program, the sets of consistent executions are the same.

Unfortunately, this is not true in general. Given a program which is race-free in M_2, we can show that it is race-free in M_3 and that it has the same consistent executions in both:

THEOREM 3.

$$
\begin{array}{l}
drf_2\ c\_prog \Rightarrow drf_3\, c\_prog\ \wedge \\
\qquad\qquad \forall ex.\ opsem\ c\_prog\ ex \Rightarrow \\
\qquad\qquad\quad (consistent\_ex_2\ ex \iff consistent\_ex_3\ ex)
\end{array}
$$

(Here we write $opsem\ c\_prog\ ex$ to mean that candidate execution ex is admitted by the threadwise operational semantics $opsem$ for C++11 program c_prog.) This is an easy corollary of the following lemma:

LEMMA 4.

1. $consistent\_ex_2\ ex \Rightarrow consistent\_ex_3\ ex \vee race_2\ ex$
2. $consistent\_ex_3\ ex \Rightarrow consistent\_ex_2\ ex \vee race_3\ ex$
3. $consistent\_ex_3\ ex \wedge opsem\ c\_prog\ ex \Rightarrow consistent\_ex_2\ ex \vee \exists ex'.\ consistent\_ex_2\ ex' \wedge race_2\ ex' \wedge opsem\ c\_prog\ ex'$

[Proof: the first two are straightforward; for the third, we can identify a minimal race in the M_3 total order and transfer that to a consistent execution in M_2.]

However, the other direction fails, as the program below shows. In the SC M_3 semantics, it is race-free, but in M_2 it has a consistent execution that has races. The example relies on the fact that in C++11 there can be non-atomic writes to atomic locations: the initialisation of atomic locations are non-atomic (so that they can be implemented without fences).

```
atomic<int> x(0);
atomic<int> y(0);
-----------------------------------
if (1==x.load(memory_order_seq_cst))
  atomic_init(&y,1);
-----------------------------------
if (1==y.load(memory_order_seq_cst))
  atomic_init(&x,1);
```

In the SC M_3 semantics, both loads must read `0` and so neither of the (non-atomic) `atomic_init` writes can take place; the program has no races and so has defined behaviour. But in M_2, there is a consistent execution in which both loads read `1`. This execution has races, and so in M_2 the program's behaviour is undefined. (This execution also has dynamic re-initialisation; the program might also be considered undefined in M_2 for that reason.)

This issue can be eliminated by requiring atomics to be initialised when constructed, removing `atomic_init`, and requiring that atomics are accessed via a path through sequenced-before and reads-from corresponding to data-flow from their creation (i.e., not by forging pointers). Under these additional restrictions, we have proved the desired equivalence of M_2 and M_3, and hence that for this fragment C++11 has SC-for-DRF-programs semantics.

THEOREM 5. *A C++11 candidate execution that uses only nonatomic operations, SC-atomic operations, and locks, and that satisfies the conditions above, is consistent in C++11 iff it is consistent in M_3, and the races are identical.*

5. Correctness of the mapping, informally

We now instantiate the first mapping of §1 for some key examples, giving an introduction to the behaviour that Power does and does not guarantee for them, and hence explaining informally why the mapping is correct in these specific cases. We also show that both mappings are locally optimal.

In C++11, synchronisation effects are associated to atomic loads and stores with particular memory order parameters, but Power assembly language has just plain loads and stores; it provides the `sync`, `lwsync`, and `isync` instructions, together with certain dependencies, to constrain the otherwise highly relaxed memory model. We introduce these constraints and the model by example here, the key properties we depend on are summarised in §6. Applying the §1 mapping to the C++11 release-acquire example of §2.1 gives a Power assembly program as follows:

y=0			
T0		T1	
`r1=1; r2=&x; r3=&y`		`r2=&x; r3=&y`	
a: `stw r1,0(r2)`	write x=1	`loop:`	
b: `lwsync`	from write-rel	d: `lwz r4,0(r3)`	read y
c: `stw r1,0(r3)`	write y=1	e: `cmpwi r4,0`	
		f: `beq loop`	
		g: `isync`	from read-acq
		h: `lwz r5,0(r2)`	read x

Here `stw` is a store, `lwz` a load, `cmpwi` a compare-immediate, and `beq` a conditional branch. This is essentially the same as the MP+lwsync+ctrlisync example of [SSA+11]: it has an `lwsync` between the two Thread 0 writes and an `isync` before the second Thread 1 read that follows a conditional branch that is data-dependent (via the compare) on the first Thread 1 read; we call the latter a *control-isync* relationship.

For this to be a correct implementation, that Power synchronisation must be strong enough to exclude any outcomes that the C++11 semantics forbids. In particular, it must prevent the Thread 1 read of `x` from reading the initial state instead of from the Thread 0 write of `x=1`. The `lwsync` and control-isync are both necessary and sufficient. The `lwsync` keeps the two Thread 0 writes in order as observed by any other thread (otherwise, as they are to different locations, they might be re-ordered in the storage hierarchy, or even committed out-of-order). Replacing the `lwsync` by an `isync` would still permit the former reordering, as is observable on Power 6 and 7 processors (MP+isync+ctrlisync), and just by typing it cannot be replaced by a control-isync, as one cannot have a dependency from a write. Meanwhile, the control-isync relationship ensures that the read of `x` cannot be satisfied until the `isync` is committed, which requires the program-order-previous branch to be committed, which requires the read of `y=1` to be committed.

Without the `isync`, the Power architecture permits a processor to speculatively satisfy the read of `x` out-of-order, before satisfying the program-order-previous read(s) of `y`, despite the intervening conditional branch. It can thereby read from the initial state, and this is observable in practice, for a litmus test MP+lwsync+ctrl. (That test is also C++11-expressible: it is essentially the result of applying the mapping to the release-acquire example with the read-acquire replaced by a read-relaxed.) The `isync` alone, without the control dependency from the preceding read to a conditional branch, also would not suffice: in this case the `isync` could be committed first, enabling read `h` to be satisfied and committed before anything else occurs. We have not observed this outcome in practice on current Power 6 or Power 7 processors (for test MP+lwsync+isync), but it is architecturally permitted. For the analogous ARM test MP+dmb+isb, the outcome is observable.

Now consider the release-consume example of §2.1, mapped to:

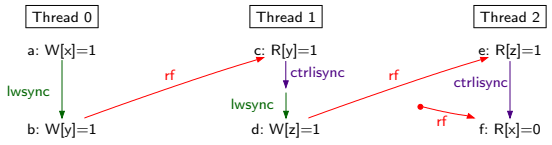
p=0			
T0		T1	
`r1=1; r2=&x; r3=&p`		`r3=&p`	
a: `stw r1,0(r2)`	write x=1	d: `lwz r4,0(r3)`	read p
b: `lwsync`	from write-rel	e: `lwz r5,0(r4)`	read *xp
c: `stw r2,0(r3)`	write p=&x		

The writing side has an `lwsync` as before, but now the reading side has no control-isync. It is correct nonetheless, because the Power architecture respects *address dependencies* between loads: here the read `e` reads from an address in register `r4` which takes its value from the program-order-previous read `d`. In such a case the reads must be satisfied (and indeed also committed) in program order.

The above examples involve only two threads, which turns out to be a very special case. With three or more threads, one can observe that on Power a write does not necessarily become visible to the other threads atomically, and indeed can be propagated to them in different orders, or even never propagated to some threads. Any transitive reasoning across multiple threads relies on the *cumulativity* of the Power `sync` and `lwsync` barriers. Consider a three-thread version of the release-acquire example, with release-acquire synchronisation between the first pair of threads and between the second pair:

```
int x;  atomic<int> y(0);  atomic<int> z(0);
-----------------------------------------------
T0  x=1;
    y.store(1,memory_order_release);
-----------------------------------------------
T1  while (0==y.load(memory_order_acquire)) {};
    z.store(1,memory_order_release);
-----------------------------------------------
T2  while (0==z.load(memory_order_acquire)) {};
    r = x;
```

The mapping introduces control-isyncs (from the load-acquires) and `lwsyncs` (from the store-releases) as in the candidate Power execution illustrated below:



Here the effect of the Thread 1 control-isync is subsumed by the following `lwsync` (eliminating the former is likely to be an important compiler optimisation). That leaves the Thread 2 control-isync, which acts just as in the first release-acquire example, and the two `lwsyncs`. To rule out the C++11-forbidden execution shown, we need to know that the writes `a` and `d`, by different threads and to different locations, are propagated to Thread 2 in that order. By the thread-local property of `lwsync` used above, the Thread 1 `lwsync` ensures that `a` and `b` are propagated to Thread 1 in that order, so `a` must have been propagated to Thread 1 before `c` reads from `b` and hence before the Thread 1 `lwsync` is committed. This means that `a` is in the *Group A* of that `lwsync`, the set of writes that have been propagated to its thread already. In turn, the cumulativity of Power `lwsyncs` means that `a` must be propagated to any other thread, e.g. Thread 2, before the `lwsync` is, and hence before write `d` is.

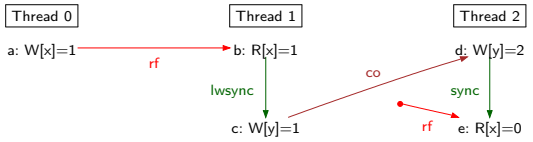
Now consider the classic four-thread IRIW example (Independent Reads of Independent Writes). Here Threads 0 and 2 write to two different locations; the question is whether Threads 1 and 3 can see those writes in opposite orders, one reading `x=1` then `y=0` while the other reads `y=1` then `x=0`.

```
atomic<int> x(0);  atomic<int> y(0);
T0  x.store(1,memory_order_release);
T1  r1=x.load(memory_order_acquire);
    r2=y.load(memory_order_acquire);
T2  y.store(1,memory_order_release);
T3  r3=y.load(memory_order_acquire);
    r4=x.load(memory_order_acquire);
```

With release-acquire atomics, this is permitted in C++11 (this is clearly not a sequentially consistent outcome, and so release-acquire atomics do *not* guarantee SC), and indeed applying the mapping gives essentially the Power test IRIW+lwsyncs, which is architecturally permitted and observable in practice.

With SC atomics, on the other hand, this non-SC outcome is forbidden in C++11. To forbid it in the implementation requires a Power `sync` between both pairs of SC-atomic reads, not just an `lwsync`. The `sync` is stronger in that its Group A writes (or some coherence-successors thereof), including any program-order-previous writes, must be propagated to all threads before the `sync` completes and subsequent memory access instructions can begin.

A `sync` is also needed between the other three combinations of an SC atomic read or write, as shown by our SB+lwsyncs and R01 examples [SSA$^+$11] for write-to-read and write-to-write, and by the WRW+WR+lwsync+sync example below for the read-to-write case, all of which are architecturally permitted. SB+lwsyncs is also observable on current implementations, whereas we have not observed the last two. Both of the latter can be explained by the fact that a Power coherence edge does not add writes into the Group A of a subsequent barrier (WRW+WR is similar to R01 but with the initial write pulled back along an rf edge). They are counterexamples to the correctness of a previous version of the mapping, which proposed an `lwsync` in place of the `sync` for SC stores in normal cacheable memory.



Test WRW+WR+lwsync+sync : Allowed

The first §1 mapping creates these intervening `syncs` by putting a `sync` before the load or store of an SC atomic. One could equally well put them after, as in the second mapping, though in C++11 SC atomics also have release/acquire semantics, so one then also needs a preceding `lwsync` for the SC atomic store (in the first mapping, the preceding `sync` does duty for both purposes).

Using the above examples, and some others, we can show:

THEOREM 6. *Both the §1 mappings are pointwise locally optimal.*

PROOF For each possible weakening there is a C++11 example with some forbidden behaviour which would map onto a Power program for which that behaviour is allowed, as checked by our `cppmem` [BOS$^+$11] and `ppcmem` [SSA$^+$11] tools using automatically generated executable versions of the models. All these examples are collected in the supplementary material [BMO$^+$].

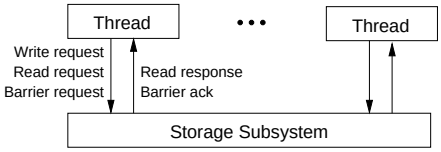
- The load-consume dependency cannot be removed (C++ message-passing with release/consume $\mapsto$ Power MP+lwsync+po).
- The load-acquire control-isync cannot be weakened to just a control dependency (MP+release/acquire $\mapsto$ MP+lwsync+ctrl), or to just an isync ($\mapsto$ MP+lwsync+isync).
- The load-seq-cst sync, whether before or after the load, cannot be weakened to an lwsync (IRIW+seq-csts $\mapsto$ IRIW+lwsyncs), or to an isync ($\mapsto$ IRIW+ctrlisyncs).
- The load-seq-cst control-isync in the original mapping cannot be weakened to just control (MP+seq-csts $\mapsto$ MP+sync+ctrl), or to just isync ($\mapsto$ MP+sync+isync).
- The store-release lwsync cannot be weakened to isync (MP+release/acquire $\mapsto$ MP+isync+ctrlisync).
- The store-seq-cst sync, whether before or after the store, cannot be weakened to lwsync (R01+seq-csts $\mapsto$ R01), or to an isync ($\mapsto$ R+isync+sync).

□

6. Background: the Power memory model

In preparation for the correctness proof in the next section, we recall the overall structure of the Power memory model [SSA$^+$11], sketching how the concepts we used informally in §5, e.g. of a write being "propagated to" a thread, reflect the actual semantics.

As we saw there, the fact that reads can be satisfied speculatively, even program-order-after a conditional branch, is observable. This pushes us towards an operational abstract-machine semantics, expressed as a parallel composition of a thread model and a storage subsystem.



At any one time, a thread may have many in-flight (uncommitted) instructions; reads can be satisfied and instructions committed out-of-order, subject to constraints from barriers and dependencies, and uncommitted instructions are subject to restart or abort. The storage subsystem abstracts from the cache and store buffering hierarchy (and does not involve speculation). It does not have an explicit memory, but instead maintains the global accumulated constraints on the coherence order between writes to the same address that have been built up, together with a list for each thread of the writes and barriers that have been propagated to that thread.

The thread and storage subsystem models are each labelled transition systems (LTSs) synchronising on labels as in the diagram above; their composition gives another LTS with labels:

label ::=
| FETCH *tid ii*
| COMMIT_WRITE_INSTRUCTION *tid ii w*

| COMMIT_BARRIER_INSTRUCTION $tid\ ii\ barrier$
| COMMIT_READ_INSTRUCTION $tid\ ii\ rr$
| COMMIT_REG_OR_BRANCH_INSTRUCTION $tid\ ii$
| WRITE_PROPAGATE_TO_THREAD $w\ tid$
| BARRIER_PROPAGATE_TO_THREAD $barrier\ tid$
| SATISFY_READ_FROM_STORAGE_SUBSYSTEM $tid\ ii\ w$
| SATISFY_READ_BY_WRITE_FORWARDING $tid\ ii_1\ w\ ii_2$
| ACKNOWLEDGE_SYNC $barrier$
| PARTIAL_COHERENCE_COMMIT $w_1\ w_2$
| REGISTER_READ_PREV $tid\ ii_1\ reg\ ii_2$
| REGISTER_READ_INITIAL $tid\ ii\ reg$
| PARTIAL_EVALUATE $tid\ ii$

Here tid ranges over thread ids, ii over instruction instances, w over write events, rr over read-response events, $barrier$ over `sync` or `lwsync` events, and reg over Power register names.

A Power execution t is simply a trace of abstract-machine states and these labels; we write machine_executions $power\_prog$ for the set of all such for a Power program $power\_prog$ (Power programs, of type POWER_PROGRAM, are essentially maps from addresses to assembly instructions, together with the initial register state, including PC, for each thread.)

Key properties required for the proofs of §7 are derived from the preconditions and postconditions of the transitions. As an example, before a barrier (`sync` or `lwsync`) is propagated to a new thread by the BARRIER_PROPAGATE_TO_THREAD transition, any writes propagated before it to its original thread (the "Group A" writes) must first be propagated to that thread (by a WRITE_PROPAGATE_TO_THREAD transition). For a `sync`, the corresponding ACKNOWLEDGE_SYNC transition cannot fire until that `sync` is propagated to all threads (and this implies that all its Group A writes, or coherence successors thereof, must have been propagated to all threads before that in the trace). The combination of a conditional branch and an `isync` instruction ("ctrlisync") stops program-order-later reads being able to do their SATISFY_READ transitions before the branch is resolved and committed.

To isolate particular events of interest, we let the following metavariables range over transition labels from a trace as follows:

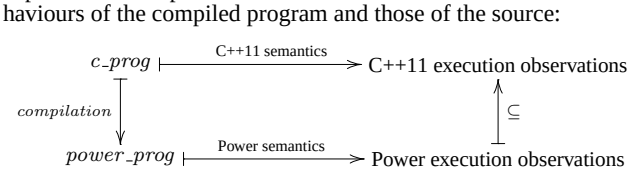
Metavariable	Transition labels ranged over
wc	COMMIT_WRITE_INSTRUCTION
rc	COMMIT_READ_INSTRUCTION
bc	COMMIT_BARRIER_INSTRUCTION
mc (memory)	COMMIT_WRITE_INSTRUCTION
	$\cup$ COMMIT_READ_INSTRUCTION
xc	COMMIT_WRITE_INSTRUCTION
	$\cup$ COMMIT_BARRIER_INSTRUCTION

7. Correct compilation from C++11 to Power

We now prove that the mappings of Section 1 are correct in general, focusing on the first mapping and then discussing the changes required for the second.

7.1 Correctness Statement

The usual form of whole-program correctness statement one might expect for a compiler is an inclusion between the observable behaviours of the compiled program and those of the source:



Here, however, we are proving correctness of compilation schemes for particular constructs (the C++11 atomic and nonatomic loads and stores), not correctness of a specific compiler. We therefore want to factor out as much as possible of the definition of the source language and of the rest of the compiler, both to remove extraneous complexity that is irrelevant to the relaxed-memory behaviour and to let us state a general result that should be re-usable in multiple later compiler correctness proofs. We do so by axiomatising the required threadwise properties of a C++11 operational semantics and a good compiler.

For the source language, we take a C++11 program, of type C_PROGRAM, to be a parallel composition of single-threaded programs from an abstract type C_PROGRAM_THREAD, with an arbitrary threadwise operational semantics for them that satisfies certain rather mild axioms. An operational semantics opsem is a relation between C_PROGRAMs and sets of C++11 actions $actions_c$ equipped with the three relevant operational-semantics relations of §2.2: sequenced-before sb_c, data-dependency dd_c, and control-dependency cd_c, packaged into a record op_c (additional-synchronises-with is empty in the absence of dynamic thread creation). Recall that this threadwise operational semantics should leave the values of memory reads unconstrained, as that is handled by the C++11 axiomatic memory model of §2.2. We return below to the axioms we need.

We take a strong intensional notion of observation: given a Power machine trace t we build read and write events $events_t$ from the COMMIT_READ_INSTRUCTION and COMMIT_WRITE_INSTRUCTION commit labels, together with a reads-from relation rf_t. The observation predicate obs_eq $t\ actions_c\ rf_c$ requires that there is an exact correspondence between these $events_t$ and the $actions_c$ of a C++11 execution, with isomorphic reads-from relations rf_t and rf_c.

Given a Power trace t, it is also straightforward to construct analogues of the threadwise C++11 relations over the constructed events, for program order po_t, data/address dependency dd_t, and control dependency cd_t; by analysing the trace, we can also identify the events from the same thread that are separated by control-isync, sync, or lwsync instructions in the trace, constructing relations $ctrlisync_t$, $sync_t$, and $lwsync_t$.

A compiler $comp$ is a function from C_PROGRAM to POWER_PROGRAM; it satisfies threadwise_good_compiler if for any C++11 program c_prog and any Power trace of $comp\ c\_prog$, there is a set of C++11 $actions_c$ with associated sb_c, dd_c, cd_c, and rf_c relations such that

1. the C++11 data has the same observations as the trace: obs_eq $t\ actions_c\ rf_c$;
2. the C++11 data is allowed by the threadwise operational-semantics: opsem $c\_prog\ actions_c\ op_c$;
3. the C++11 relations sb_c, dd_c, and cd_c are subsets of the corresponding relations po_t, dd_t, and cd_t built from the trace (note that the C++11 sequenced-before relation may not be total within a thread, as evaluation order is sometimes unconstrained, but the Power program order is total within a thread); and
4. if one applies the mapping to the C++11 actions $actions_c$, the result is included in the $ctrlisync_t$, $sync_t$, and $lwsync_t$ relations constructed from the Power trace.

For example, a C++11 action $\mathtt{a}{:}\mathsf{W}_{\mathsf{REL}}\ \mathtt{x}{=}1$ can be presumed to arise from a source program instruction `x.store(e,memory_order_release)`, and so a threadwise good compiler following the mapping should have compiled it into a Power `lwsync` preceding a store, and any trace should contain a corresponding COMMIT_BARRIER_INSTRUCTION label followed (perhaps not immediately) by a COMMIT_WRITE_INSTRUCTION.

Finally, we express correctness on a per-execution basis and can assume that the source program is race-free, otherwise C++11 regards it as undefined and compilation is unconstrained. That leads us to the following statement of our main theorem:

THEOREM 7 (Correctness).
$\forall comp\ c\_prog.$
threadwise_good_compiler $comp \wedge$
drf c_prog
$\Rightarrow$
(let $power\_prog = comp\ c\_prog$ in
($\forall t \in$machine_executions $power\_prog.$
($\exists actions_c\ op_c\ rf_c\ mo_c\ sc_c.$
obs_eq $t\ actions_c\ rf_c \wedge$
opsem $c\_prog\ actions_c\ op_c \wedge$
consistent_ex $actions_c\ op_c\ rf_c\ mo_c\ sc_c$)))

Our proof follows this structure. Consider a Power trace t of a Power program $power\_prog$ produced by a threadwise-good compiler from a C++ program c_prog. The threadwise_good_compiler assumption tells us that there are some corresponding (up to obs_eq) C++11 $actions_c$, op_c, and rf_c for which the memory actions of each thread satisfy the threadwise operational semantics. We now construct a C++11 modification order mo_c and sequential consistent order sc_c, and then have to either prove that the C++11 execution comprising all this data is consistent, or to construct another consistent execution which contains a race, thereby contradicting the drf assumption.

The construction of a C++11 modification order mo_c is straightforward, as (like sb_c, dd_c, cd_c, and rf_c) there is a direct analogue in the Power trace: the coherence constraints in the storage subsystem state increase monotonically along a trace, so we can just take their union co_t, choose an arbitrary per-location linearisation thereof, and restrict to the atomic locations. Construction of a C++11 SC order sc_c is more involved, as we see below in §7.3.

Note that, in contrast to many compiler correctness proofs, this is not a simulation argument: the C++11 definition of consistent execution is a predicate on complete candidate executions; it is not generated by an operational semantics and does not involve any notions of a C++11 whole-program state or transition.

7.2 Analysis of a Power trace w.r.t. C++11 inter-thead-happens-before

The C++11 definition of consistent execution comprises seven conditions, as listed in §2.2, almost all of which rely heavily on the *happens-before* relation, which is the union of sequenced-before and a complex *inter-thread-happens-before* relation.

To establish those conditions, we need to analyse the Power trace with the shape of the *inter-thread-happens-before* relation in mind, making use of various ordering properties of the Power semantics and the fact that the Power program is obtained using the given mapping.

In broad terms, if there is a C++11 inter-thread-happens-before relation from a write to a read, we need to show that the write has propagated to the reading thread before the read is satisfied. For read-to-read pairs, we need to show that any write propagated to the thread of the first read, before it is satisfied, has propagated similarly. For write-to-write and read-to-write pairs, we need to ensure propagation before the commit of the final write.

In more detail, we must also allow coherence successors of writes to be propagated in their place, must consider propagation of barriers, must refer to the last satisfy label of any read that reads from a different thread, and must split the different-thread/same-thread cases. To capture all this, we define a $propBefore$ relation over memory and barrier commit labels. For labels from different threads, we say:

- $bc \xrightarrow{propBefore} rc$ iff the barrier has propagated to the reading thread before the read is finally satisfied, i.e., if there is a BARRIER_PROPAGATE label for bc trace-order-before the last SATISFY_READ for rc;
- $wc \xrightarrow{propBefore} rc$ iff the write, or some coherence-successor write, has propagated to the reading thread before the read is finally satisfied; and
- $wc_1 \xrightarrow{propBefore} wc_2$ iff the write of wc_1, or some coherence-successor write, has propagated to the thread of wc_2 before the wc_2 commit.

For labels from the same thread, we say:

- $xc \xrightarrow{propBefore} rc$ iff xc is either a write commit label and is read by rc or it is trace-order-before rc is finally satisfied; and
- $xc \xrightarrow{propBefore} wc$ iff xc is trace-order-before wc.

Consider now the C++11 definition of happens-before (hb). Without relaxed or consume operations, it only crosses threads with a reads-from edge of a release/acquire pair that synchronise with each other. Under the mapping, this rf_t edge must be preceded by $sync_t$ or $lwsync_t$ and followed by a $ctrlisync_t$. With relaxed atomics, the acquire might read from something in the release sequence of the release it synchronises with. In Power terms, a release sequence is a series of same-thread co_t edges (if we covered read-modify-write operations, these would not necessarily be on the same thread). Adding consume atomics, release/consume synchronisation involves the dependency-ordered-before (dob) relation, which is similar to ithb except that on the right hand side it only reaches to dependent operations, not to all program-order successors of the read. In Power terms, in this case the $ctrlisync_t$ is replaced by ${dd_t}^*$. All this motivates the definition of a machine analogue of ithb, $machine\text{-}ithb_t$:

$$\left((sync_t \cup lwsync_t)^{\text{refl}}; {coi_t}^*; rfe_t; (ctrlisync_t \cup {dd_t}^*)^{\text{refl}}\right)^+$$

where coi_t is the machine coherence order restricted to pairs of writes by the same thread, and rfe_t is the machine reads-from relation restricted to write/read pairs on distinct threads.

LEMMA 8. *Given two C++11 memory actions m_1 and m_2 and the corresponding Power commit events mc_1 and mc_2, if $m_1 \xrightarrow{ithb_c} m_2$ then $mc_1 \xrightarrow{machine\text{-}ithb_t} mc_2$.*

PROOF By a series of lemmas about the C++11 auxiliary relations, unfolding the definitions, with reasoning as above. □

To prove the required propagation property (Corollary 10 below), we first consider the left part of this relation:

LEMMA 9 (Propagation).

1. *If $wc \xrightarrow{((sync_t \cup lwsync_t)^{\text{refl}}; {coi_t}^*; rfe_t)^+} rc$, then $wc \xrightarrow{propBefore} rc$.*
2. *If $rc_1 \xrightarrow{((sync_t \cup lwsync_t)^{\text{refl}}; {coi_t}^*; rfe_t)^+} rc_2$, then for any write commit $wc \xrightarrow{propBefore} rc_1$, we have $wc \xrightarrow{propBefore} rc_2$.*

PROOF By induction on the transitive closure of the relation. In the base case, by decomposing the large relation, two intermediate write commit events appear (wc_1 and wc_2). In the case where $mc \xrightarrow{sync_t \cup lwsync_t} wc_1$, if mc is a write, then it falls into the Group A of the intervening barrier; if it is a read, then any write propagated to its thread before it is satisfied falls into the Group A of the barrier. The ${coi_t}^*$ relation is a subset of po_t, hence we also have $mc \xrightarrow{sync_t \cup lwsync_t} wc_2$. Hence, using the A-cumulativity property of the Power machine and since we know that wc_2 does propagate to the thread of rc before rc is last satisfied (because it is read from), the writes in the Group A of the barrier (or some coherence successors) must have previously been propagated to that thread. This establishes the desired $\xrightarrow{propBefore}$ relationship. In

the other case, $mc = wc_1$, we just saw that wc_2 does propagate to the thread of rc in time and it is here a coherence-successor of mc. For the inductive case we use the same reasoning as for the first step to establish that the writes under consideration are first propagated to the thread of the intermediate read, and then the induction hypothesis. □

COROLLARY 10 (Propagation w.r.t. ithb).

1. *If* $rc \xrightarrow{machine\text{-}ithb_t} mc$ *then for any write commit* wc *with* $wc \xrightarrow{propBefore} rc$, *we have* $wc \xrightarrow{propBefore} mc$.
2. *If* $wc \xrightarrow{machine\text{-}ithb_t} mc$ *then* $wc \xrightarrow{propBefore} mc$.

PROOF $ctrlisync_t$ and dd_t are subsets of po so, without loss of generality, we can neglect all those edges except the last one:

$$wc \xrightarrow{((sync_t \cup lwsync_t)^{\text{refl}};coi_t{}^*;rfe_t)^+} rc \xrightarrow{(ctrlisync_t \cup dd_t{}^*)} mc$$

Lemma 9 establishes propagation to the thread of rc, then, using either the $ctrlisync_t$ or dd_t, if mc is a read then it is satisfied after rc is committed, and if mc is a write then it is committed after rc is committed. □

Several of the C++11 conditions are acyclicity properties (either explicitly so, e.g. the consistency of ithb, or requiring consistency between relations, e.g. rf_c and hb_c). To establish these, it is convenient to reduce them to the acyclicity of trace order, by the following lemma.

LEMMA 11 (Trace order respects $machine\text{-}ithb_t$).
If $mc_1 \xrightarrow{machine\text{-}ithb_t} mc_2$ *then* mc_1 *is before* mc_2 *in the trace.*

PROOF Along the same lines as the reasoning above, by induction on the transitive closure of $machine\text{-}ithb_t$. □

7.3 Analysis of a Power trace to construct a C++11 SC order

C++11 demands a single total order sc over all the SC atomic actions, subject to some consistency conditions. Broadly, this order must be consistent with the sequenced-before relation, with modification order, and for each SC read rc, if it reads from a SC write wc, then wc must be the last SC write to that location before rc in the SC order, while if it reads from a non-SC write wc', then wc' cannot be after (in happens-before) any SC write to the same location which is itself after rc in the SC order.

In Power terms, the first two conditions are straightforward to express: we need to construct a total order on SC actions which includes the subparts of po_t and co_t (po_t^{sc} and co_t^{sc} respectively) that are restricted to SC actions. However, there is no obvious corresponding total order visible in the trace. In particular, recall from §6 that for write commit labels trace order is not necessarily consistent with the coherence order co_t. It is also not sufficient to use an arbitrary linearisation of $(po_t^{sc} \cup co_t^{sc})^*$ (though po_t and co_t are individually acyclic), since that may introduce bad intervening writes to the same location in between a pair of a write and a read that reads-from it. Consider then an SC read rc, and the write wc (not necessarily SC) that it reads-from, and recall that we have a coherence order co_t for writes to that location. We define two new derived relations on SC reads and writes: from-reads (fr_t^{sc}) defined as $rc \xrightarrow{fr_t^{sc}} wc_1$ if $wc \xrightarrow{co_t} wc_1$, i.e. rc reads-from a coherence predecessor of the SC wc_1; and extended-reads-from (erf_t^{sc}) defined as $wc_1 \xrightarrow{erf_t^{sc}} rc$ if wc_1 is the last SC write in coherence order equal to or before the wc that rf_t reads from. From-reads relations have been used by various researchers [ABJ+93, AMSS10, Alg10], and the latter reference proves that if the union of program-order, coherence, reads-from, and from-reads is acyclic, then the execution is an SC execution by the traditional definition [Lam79] of a single global order over all events. We use this characterisation in Theorem 13 below.

We will use as the SC order an arbitrary linearisation of $(po_t^{sc} \cup co_t^{sc} \cup fr_t^{sc} \cup erf_t^{sc})^*$. We now show that this combination of relations is acyclic, and hence can be consistently extended to a total order. First note that if the above relation has a cycle, there must be at least one $mc_1 \xrightarrow{po_t^{sc}} mc_2$ edge in the cycle, as otherwise it is straightforward to deduce that the coherence relation implied by the machine trace is itself cyclic. Under the mapping, this means that mc_1 and mc_2 must be separated in the trace by a sync and its ACKNOWLEDGE_SYNC transition. Our main lemma for this (Lemma 12 below) says that for any mc with $mc \xrightarrow{(po_t^{sc} \cup co_t^{sc} \cup fr_t^{sc} \cup erf_t^{sc})^*} mc_1$, it must be in trace-order before that ACKNOWLEDGE_SYNC transition (call this the S transition), and this contradicts mc_2 being after that transition.

We will prove Lemma 12 by induction on the length of the relation, case analysing the relation in the first step. We have to strengthen the inductive hypothesis in two ways to make the proof go through. For example, consider the case where the first step is a $rc \xrightarrow{fr_t^{sc}} wc$ edge. We know inductively that wc is before S in the trace, but this does not immediately help us know that the read rc is also before S. What we need is the strong property of the sync (recall §6), that before the acknowledgement for a sync in the trace, preceding writes are not merely committed but also they (or some coherence successors) are propagated to all threads. This suffices, since if wc is propagated to the reading thread, and it is not read-from, the read must already have been done.

For the second strengthening, consider the case where the first step is a $wc \xrightarrow{rf_t} rc$ edge (one kind of an erf_t^{sc} edge is a rf_t edge). As above, we will have to inductively establish that the write wc have been propagated to all threads before the sync acknowledgement S. The Power machine guarantees this for any sync on the same thread as rc, but not necessarily for syncs on other threads. To make the induction work, we will find a new intermediate sync acknowledgement S_n, which is on the same thread as an event related to rc in the po_t^{sc}-free part of the relation. Since this involves only co_t^{sc}, fr_t^{sc} and erf_t^{sc} edges, these will be on the same location, and wc will then be correctly related by coherence to a write that is known to be propagated everywhere before S_n in the trace.

The final statement is then:

LEMMA 12 (Main Lemma for SC). *Suppose* $mc_1 \xrightarrow{po_t} mc_2$, *and there is a sync ack* S *between* mc_1 *and* mc_2 *in the trace. Suppose* $mc_n \xrightarrow{(po_t^{sc} \cup co_t^{sc} \cup fr_t^{sc} \cup erf_t^{sc})^*} mc_1$. *Then there is an* mc'_n *and a sync ack* S_n *such that:*

1. mc_n *is before* S_n *in the trace, and* S_n *is before or equal to* S *in the trace;*
2. *If* mc_n *is a write, then that write or a coherence successor has propagated to all threads before* S_n *in the trace;*
3. mc'_n *is before* S_n *in the trace;*
4. mc'_n *and* S_n *are from the same thread; and*
5. $mc_n \xrightarrow{(co_t^{sc} \cup fr_t^{sc} \cup erf_t^{sc})^*} mc'_n$ *(the subpart of the relation between* mc_n *and* mc'_n *is* po_t*-free).*

PROOF By induction on the length of the chain $mc_n \xrightarrow{(po_t^{sc} \cup co_t^{sc} \cup fr_t^{sc} \cup erf_t^{sc})^*} mc_1$. In the base case, $mc_n = mc_1$, and the required conditions are easily established by taking $mc'_n = mc_n$ and $S_n = S$.

In the inductive case, we case analyse on the kind of relation in the first step. For co_t^{sc}, fr_t^{sc} and erf_t^{sc}, we use the same intermediate event and sync acknowledgement as the inductive hypothesis, making $mc'_n = mc'_{n-1}$ and $S_n = S_{n-1}$. Then, for example, if the

edge is $wc_1 \xrightarrow{co_t^{sc}} wc_2$, we have inductively that wc_2 is propagated to all threads as required before S_n, and thus a coherence successor of wc_1 is as well. We outlined the $rc \xrightarrow{fr_t^{sc}} wc$ and $wc \xrightarrow{erf_t^{sc}} rc$ cases in the description above.

In the remaining case, $mc_n \xrightarrow{po_t^{sc}} mc_{n-1}$, we have two SC events separated in program order. By the mapping, there must be a sync between them in program order. Let S_n be the acknowledgement transition for that sync, and $mc'_n = mc_n$. Then the required conditions are easy to check. □

Notice in the proof that we use the sync program-order between any two SC actions from the same thread. This is crucial for the induction to go through. Indeed, as shown in §5, for any pair of SC actions, if we weaken the sync to anything else (lwsync or weaker), we have a counterexample to show a program with only SC actions behaving in a non-sequentially consistent manner.

The proof is applicable in a wider setting than just for C++11. The key fact we used to create the SC order is that every pair of SC actions on the same thread is separated (in program order) by a sync. Looking at the proof for the case that every memory access is a SC atomic action (*i.e.* no other memory types, or non-atomic accesses) shows that such a program has only sequentially consistent behaviour, an interesting fact in its own right about Power assembly programs.

THEOREM 13 (Syncs between every pair of accesses restore SC). *Suppose we have a Power (assembly) program with every pair of memory accesses on the same thread separated in program order by a sync. Then the program has only sequential consistent behaviour.*

PROOF Given the premises, the proof of Lemma 12 shows that $(po_t \cup co_t \cup fr_t \cup rf_t)^*$ is acyclic, then by [Alg10, §4.2.1.3] the execution is Lamport-SC. □

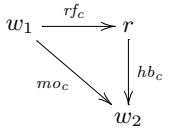
7.4 Verification of consistent_execution

We now have the tools to verify the satisfaction of all the conjuncts of the *consistent_execution* predicate, as listed in §2.2 and simplified by Theorem 1 of §3. The two *well_formed* conjuncts hold by the obs_eq and assumptions on opsem, e.g. that the sequenced-before relation only relates actions of the same thread. The *consistent_locks* conjunct is vacuous in the sublanguage we consider in this section. For the others, we use the correspondences between C++11 actions and relations and Power trace events and relations we established in §7.1 and in §7.2, mostly using Corollary 10 and Lemma 11.

First, $consistent\_inter\_thread\_happens\_before$ states the acyclicity of ithb. Using Lemma 11, we prove by contradiction that such a cycle implies a cycle in the trace order.

Next we have four coherence conditions for pairs of a read and a write (CoRR, CoRW, CoWR and CoWW), which are part of *consistent_modification_order* and *coherent_memory_use* (itself part of *consistent_reads_from_mapping*). These diagrams involve rf_c and hb_c edges relating two writes and some reads, all at the same atomic location; they require the writes to be correctly ordered in mo_c. For example:

LEMMA 14 (CoRW). *Given C++11 actions w_1, r, and w_2, all to the same location, and an rf_c and hb_c edge as below, the $w_1 \xrightarrow{mo_c} w_2$ edge exists.*



PROOF To establish this we first show that the Power write wc_1 corresponding to w_1 is propagated to the thread of the Power write wc_2 corresponding to w_2 before wc_2 is committed. In more detail: For the rf_c edge, the inclusion from §7.1 gives us a Power $wc_1 \xrightarrow{rf_t} rc$ relation between the corresponding commit events from the trace. For the hb_c edge, by the definition of C++11 happens-before, either (case 1) there is an sb_c edge, in which case we use the inclusion from §7.1 to show $rc \xrightarrow{po_t} wc_2$ (and they are on the same thread), or (case 2) an $ithb_c$ relationship, in which case there is an $rc \xrightarrow{machine\text{-}ithb_t} wc_2$ edge. We know (because of the Power rules involved in a reads-from relation, and with a case split on whether wc_1 and rc are on the same thread or not) that wc_1 propagates to the thread of rc before rc is finally satisfied. Hence wc_1 is propagated to the thread of wc_2 before the latter is committed, either (in case (1)) because the semantics of the commit transition guarantees that po_t is respected by the trace order for events at a same location, and using transitivity, or (in case (2)), by Corollary 10.

Finally, by the storage-subsystem semantics for COMMIT_WRITE_INSTRUCTION, when a write is committed, it automatically becomes coherence-after all the writes that have already been propagated to its thread, so $wc_1 \xrightarrow{co_t} wc_2$. And by the construction of mo_c, that gives $w_1 \xrightarrow{mo_c} w_2$. □

The other three coherence properties are similar, albeit administratively more complex. Two additionally use the fact that a Power read can only be satisfied from the last write (to the relevant address) that has been propagated to its thread.

The *consistent_sc_order* predicate checks that SC is a strict total order over the SC actions. The totality is by construction; strictness is by acyclicity. It also requires that hb_c and mo_c restricted to SC atomics are included in sc_c. The first is immediate from the construction of hb_c from subsets of the relations involved in the construction of sc_c. The second makes use of a lemma stating the inclusion in co_t of the relation from which we build the SC-order when restricted to pairs of writes at a same location, and the construction of mo_c.

The *consistent_modification_order* predicate consists of CoWW, dealt with above, a totality condition that is immediate from the construction of mo_c as a linearisation, and a check that it only relates writes at atomic locations, which is also by construction.

Now we have the subclauses of *consistent_reads_from_mapping*. Two, *rmw_atomicity* and *sc_fences_heeded*, are vacuous for the sublanguage we consider. We dealt with the coherence conditions of *coherent_memory_use* above.

The predicate *sc_reads_restricted* forces any SC read to read from the last sc_c-preceding write (wc) at the same location, or some non-SC write that is not hb_c-before wc. The proof is by constructing a SC-cycle for each forbidden situation, which contradicts the results of §7.3.

An interesting observation here is that, through the application of the acyclicity result of §7.3, we make use for the first time of the barriers placed by the mapping before the compilation of SC reads. And in fact, we do not rely anywhere else on these barriers as they play no part in the propagation property stated by Lemma 9.

Finally, we get to the constraints on read values. The *consistent_non_atomic_read_values* predicate says that a read r at a non-atomic location must read from a visible side-effect, i.e., a write w that happens-before r such that there is no write w' that happens-between w and r. Assuming the contrary, there are three possible situations:

1. r reads from a write that happens-after it;
2. r reads from an hb-hidden write: there exists a w' that happens-between w and r; or
3. r reads from an hb-unrelated write.

Cases 1 and 2 are lengthy but in the same style as the reasoning for CoRW and CoWR, showing that writes propagate appropriately.

Case 3 is quite different. Here we have a Power read rc that reads from a write wc for which neither $w \xrightarrow{hb_c} r$ nor $r \xrightarrow{hb_c} w$ hold. Intuitively, this is a race, but this is not a consistent execution: r cannot read from w in C++ unless $w \xrightarrow{hb_c} r$.

In the next section we show that in this case the original program has some other candidate execution that is consistent and that has a race (a data race or indeterminate read), contradicting the top-level assumption that the program was drf. A similar situation arises for the last subclause, *consistent_atomic_read_values*.

Formally, we case split at the top level on the following rf_in_hb predicate, capturing additional assumptions that ensure consistency:

$$\begin{aligned} &rf\_in\_hb = \forall w\ r.\\ &(w \xrightarrow{rf_c} r \wedge r \text{ is a non-atomic read } \Rightarrow w \xrightarrow{hb_c} r) \wedge\\ &(w \xrightarrow{rf_c} r \wedge r \text{ is an atomic read } \Rightarrow\\ &\quad (\exists w'.\ \text{is\_write } w' \wedge \text{ same\_location } w\ w' \wedge w' \xrightarrow{hb_c} r)) \end{aligned}$$

If rf_in_hb holds, case (3) is excluded and *consistent_non_atomic_read_values* and *consistent_atomic_read_values* can be established directly.

7.5 Construction of a racy consistent execution

In the situation where the rf_in_hb predicate of the previous section does not hold, our strategy is to 1) find a prefix of the trace for which rf_in_hb holds and build a consistent execution for it by applying the reasoning of Section 7.4 to that prefix; 2) add an indeterminate or data-race read to the consistent execution; 3) return any missing sequenced-before predecessor actions to the consistent execution; 4) extend the consistent execution until it is a complete execution of the original program.

For Step 1, we find the first read commit rc on the trace that causes a violation of rf_in_hb, and build a consistent execution for part of the trace that precedes it. Our ability to use Section 7.4 for this relies on the fact that the commit labels on the trace respect rf_t, dd_t, and cd_t, which are in turn consistent with rf_c, dd_c, and cd_c.

Steps 2–4 rely on the following receptiveness axiom about the C++ threadwise operational semantics, which we believe any reasonable operational semantics should satisfy.

$$\begin{aligned} &\forall c\_prog\ actions'_c\ actions_c\ op_c\ a\ new\_a.\\ &\quad \text{opsem } c\_prog\ actions_c\ op_c \wedge\\ &\quad a \in (actions_c \setminus actions'_c) \wedge new\_a \in \text{all\_values } a \wedge\\ &\quad \text{is\_read } a \wedge actions'_c \subset actions_c \wedge\\ &\quad (actions'_c \cup \{a\}) \text{ is downward closed under } (dd_c \cup cd_c)^+\\ &\quad \Rightarrow\\ &\quad (\exists new\_act_c\ new\_op_c.\\ &\qquad \text{let } stay\_act_c = actions'_c \cup\\ &\qquad\quad \{b | b \in actions_c \setminus actions'_c \wedge b \neq a \wedge \neg\, a \xrightarrow{(dd_c \cup cd_c)^+} b\}\\ &\qquad \text{in}\\ &\qquad\quad stay\_act_c \cap new\_act_c = \emptyset \wedge\\ &\qquad\quad ((op_c|_{(stay\_act_c \cup \{a\})})[new\_a/a] =\\ &\qquad\quad\ new\_op_c|_{(stay\_act_c \cup \{new\_a\})}) \wedge\\ &\qquad\quad (\forall x\ y.\ x \xrightarrow{(new\_dd_c \cup new\_cd_c)^+} y \wedge x \in new\_act_c \Rightarrow\\ &\qquad\qquad y \notin stay\_act_c \cup \{new\_a\}) \wedge\\ &\qquad\quad \text{opsem } c\_prog\ (stay\_act_c \cup \{new\_a\} \cup new\_act_c)\ new\_op_c) \end{aligned}$$

It states that if the operational semantics can perform a read at a certain point, it can instead perform any read that is of the same kind and from the same location, but that reads a (potentially) different value. This reflects the fact that the threadwise operational semantics relies on the memory model to determine how memory reads are satisfied. Furthermore, later actions that do not depend on the read via $(dd_c \cup cd_c)^+$ are unaffected by the change of value read. This reflects the fact that the threadwise operational semantics is required to correctly calculate the notions of control and data dependence. We now return to the construction of the racy execution. For Step 2, if there is a visible side effect w' of r (recall that r is the read that violated hb_in_rf), we create a new read r' according to the opsem axiom and have r' read from w'. This r' then races with the write w that r read from originally. If there is no visible side effect, we have r read from nothing, and the execution has an indeterminate read. We prove that this new execution is consistent by showing that the happens before relation is unchanged, except for switching r to r'.

Step 3 is necessary because the Power can speculatively execute reads out of program order, and so the prefix chosen in Step 1 might be missing actions that the C++ threadwise operational semantics requires. In other words, the trace ordering is not necessarily consistent with po_t, and hence the trace prefix might not be downward closed in sb_c. The addition of such a missing read r in Step 3 follows the same reasoning as in Step 2 with the added observation that r cannot be (atomic) sequentially consistent or acquire because the Power cannot speculate past the control+isync dependency that would follow such a read. The addition of missing writes is straightforward.

Step 4 is straightforward because there are no constraints on how the execution extended to completion; it already contains the race, and further application of opsem axiom will not disturb it.

This completes the proof of Theorem 7.

7.6 Alternate trailing-`sync` mapping

As mentioned in §1, a modified mapping has been proposed that has a `sync` barrier as the last instruction in the mapping for all SC actions. This keeps a `sync` barrier program-order between any pair of SC actions from the same thread, which ensures the results of §7.3 hold. The rest of the proof interacts with SC actions through the SC order, so it can be carried over unchanged. One additional subtlety is that SC stores are also release stores, and SC loads in C++ are also acquire loads. The first fact requires the `lwsync` before the store for SC stores, as for release stores. The second requires program-order-later loads to be stopped from being satisfied before the SC load is. The trailing `sync` suffices here.

8. Related Work

The most closely related work is our correctness proof of a compilation scheme from C++11 to x86 [BOS$^+$11]. That covered RMWs in addition to loads and stores, but was nonetheless comparatively simple, as x86 has a strong TSO-based semantics [SSO$^+$10]. For Section 3, Boehm and Adve gave a definition [BA08] of a memory model based on an earlier C++11 working paper, broadly in the same style as the eventual draft standard and [BOS$^+$11], but differing in many details and with the semantics for low-level atomics only sketched. They give a hand proof that in that model DRF programs without low-level atomics have SC semantics.

Some previous work proves correctness of compilers for concurrent languages, rather than compilation schemes for particular primitives, in a relaxed-memory context. Ševčík et al. prove correctness of CompCert-TSO [SVZN$^+$11], a compiler from a concurrent C-like language with TSO semantics to the x86 TSO model (building on Leroy's single-threaded CompCert [Ler09]). Here the source and target share the same relatively simple memory model, and the correctness proof is simulation-based, taking advantage of the fact that x86-TSO has a simple operational characterisation. Lochbihler proves correctness of compilation from a formalisation of multi-threaded Java to a JVM [Loc10], but this assumes SC and does no optimisation. Ševčík [Šev11] proves the correctness (or otherwise) for various optimisations for DRF languages, though not specifically for C++11. Burckhardt et al. [BMS10] consider the correctness of transformations in relaxed models. Vafeiadis and Zappa

Nardelli verify fence elimination optimisations in the context of the CompCert-TSO compiler [VZN11].

There is an extensive line of work inserting fences to restore SC, starting with Shasha and Snir [SS88]. Fang et al. [FLM03] do this in practice for sync on Power 3 machines, but they appear to implicitly assume that writes are atomic (discussing only thread-local reordering), which they are not on more recent Power implementations. Alglave et al. [AMSS10, Alg10, AM11] consider Power fence insertion with respect to an axiomatic memory model, the *CAV2010 model*. That model was a precursor to the abstract-machine model of Sarkar et al. [SSA+11] that we use in this paper. As described there, the CAV2010 model is stronger than the architectural intent for the R01 test (in this sense it is unsound w.r.t. the architecture, although not observed to be unsound w.r.t. current implementations), and it is weaker than the architecture (and current implementations) for cases such as MP+lwsync+addr. The latter might be generated by the mapping for a C++ release/consume example, so this model is too weak to prove the mapping correct.

Lea produced a guide to implementing the JSR-133 Java Memory Model on various multiprocessors [Lea]. Written when the Power architectural intent was less clear (considerably before [SSA+11]), its suggested uses of `isync` are not correct w.r.t. the architecture. It also focuses on the synchronisation required between pairs of operations on the same thread, without discussing the Power cumulativity properties that are essential for C++11 release/acquire, release/consume, and SC atomics.

9. Conclusion

We have proved the correctness of a realistic relaxed-memory compilation scheme, from the programming language memory model proposed for the mainstream C and C++ standards, to a realistic memory model for Power multiprocessors. The C++11 model was designed with implementation above the Power (and ARM) model in mind, among others, and this establishes that it is in fact implementable with what appears to be a reasonable mapping (using hardware synchronisation mechanisms commensurate with the C++11 semantics required). Moreover, our development explains why the mapping is correct and also what properties of the Power are actually required, increasing understanding and confidence in both models, and providing a good basis for compiler developers compiling C++11 and C1x to Power and ARM; we are discussing this with GCC and ARM compiler groups.

There are many interesting directions for future work. First, the development should be extended to cover other C++11 features: read-modify-write and locks operations require first a semantics for the Power load-reserve/store-conditional instructions, and one would also like to cover fences and dynamic thread creation.

Second, while a full compiler correctness proof for C++ (or even C) is still a long way off, we would like to instantiate our compilation-scheme proof to a concrete operational semantics and compiler for a small fragment.

Third, the proof suggests the construction of a more abstract axiomatic Power model, based on the derived properties we use in the proof. We can see there exactly which machine trace events are relevant, e.g. the last satisfy label for each committed read, and certain write propagation events; the more abstract model could deal just with those. This would give a simpler foundation for developing analysis and reasoning techniques for Power and ARM concurrent software.

Acknowledgements We thank Hans Boehm, Paul McKenney, Jaroslav Ševčík, and Francesco Zappa Nardelli for discussions on this work, and acknowledge funding from EPSRC grants EP/F036345, EP/H005633, and EP/H027351.

References

[ABJ+93] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *Proc. SPAA*, 1993.

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

[Alg10] J. Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris 7 and INRIA, 2010.

[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+] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. `http://www.cl.cam.ac.uk/users/pes20/cppppc`.

[BMS10] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying local transformations on relaxed memory models. In *CC*, 2010.

[Boe11] Hans Boehm. Atomic synchronization sequences, 2011. Mailing list communication, July 18th.

[BOS+10] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency: The post-Rapperswil model. Technical Report N3132, ISO IEC JTC1/SC22/WG21, August 2010. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3132.pdf`.

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

[BWB+11] J. C. Blanchette, T. Weber, M. Batty, S. Owens, and S. Sarkar. Nitpicking C++ concurrency. In *Proc. PPDP*, 2011.

[FLM03] X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *Proc. ICS*, 2003.

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

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

[Ler09] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[Loc10] A. Lochbihler. Verifying a compiler for Java threads. In *Proc. ESOP'10*, 2010.

[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.

[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.

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

[SVZN+11] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *Proc. POPL*, 2011.

[VZN11] V. Vafeiadis and F. Zappa Nardelli. Verifying fence elimination optimisations. In *Proc. SAS*, 2011.