Overhauling SC Atomics in C11 and OpenCL

Mark Batty
University of Kent, UK
m.j.batty@kent.ac.uk

Alastair F. Donaldson
Imperial College London, UK
alastair.donaldson@imperial.ac.uk

John Wickerson
Imperial College London, UK
j.wickerson@imperial.ac.uk

Abstract

Despite the conceptual simplicity of sequential consistency (SC), the semantics of SC atomic operations and fences in the C11 and OpenCL memory models is subtle, leading to convoluted prose descriptions that translate to complex axiomatic formalisations. We conduct an overhaul of SC atomics in C11, reducing the associated axioms in both number and complexity. A consequence of our simplification is that the SC operations in an execution no longer need to be totally ordered. This relaxation enables, for the first time, efficient and exhaustive simulation of litmus tests that use SC atomics. We extend our improved C11 model to obtain the first rigorous memory model formalisation for OpenCL (which extends C11 with support for heterogeneous many-core programming). In the OpenCL setting, we refine the SC axioms still further to give a sensible semantics to SC operations that employ a 'memory scope' to restrict their visibility to specific threads. Our overhaul requires slight strengthenings of both the C11 and the OpenCL memory models, causing some behaviours to become disallowed. We argue that these strengthenings are natural, and that all of the formalised C11 and OpenCL compilation schemes of which we are aware (Power and x86 CPUs for C11, AMD GPUs for OpenCL) remain valid in our revised models. Using the HERD memory model simulator, we show that our overhaul leads to an exponential improvement in simulation time for C11 litmus tests compared with the original model, making *exhaustive* simulation competitive, time-wise, with the *non-exhaustive* CDSChecker tool.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

Keywords Formal methods, graphics processing unit (GPU), heterogeneous programming, HOL theorem prover, language design, program simulation, weak memory models

1. Introduction

Atomics and memory models. C11 and OpenCL both define a collection of *atomic operations*, or 'atomics', which can be used by experts to program high-performance, lock-free algorithms in a portable manner. Atomics accept a *memory order* parameter, which controls the exposure of certain *relaxed memory* behaviours that modern CPUs and GPUs natively exhibit.

The C11 and OpenCL specifications [21, 23] define the semantics of atomics via axiomatic *memory models*; that is, sets of rules that govern the reading and writing of shared memory locations. These memory models are complex, stretching to about 19 and 30 pages, respectively, of convoluted prose. This complexity makes it extremely challenging to reason about the correctness of programs that are written in, and compilers that implement, these languages.

Correctness in any relaxed memory setting is notoriously evasive; indeed, the subtleties of relaxed memory have previously led to confirmed bugs in language specifications [7, 10], deployed processors [1], compilers [27, 40] and vendor-endorsed programming guides [3]. The importance of correctness in the context of C11 is well-known. Correctness is just as crucial in OpenCL, which is an open standard for *heterogeneous programming* that is developed and supported by major hardware vendors such as Altera, AMD, ARM, Intel, Nvidia, Qualcomm and Xilinx. OpenCL is a key player in the recent drive to exploit GPUs and FPGAs in general-purpose computing, including in safety-critical domains such as medical imaging [34] and autonomous navigation [24].

We seek in our work to tame the complexity of these memory models through *formalisation*.

The C11 memory model has been formalised by several researchers, in varying degrees of completeness, and with varying degrees of fidelity to the standard [2, 7, 38]. These formalisation efforts have proved fruitful; they have, for instance, enabled the construction of simulators that automatically explore the allowed behaviours of small C11 programs (called *litmus tests*) [2, 7, 13, 29], underpinned the design of program logics for specifying and verifying C11 programs [37, 38], and they provide a firm foundation for ongoing debate about the design of the C11 memory model itself [10, 39].

The OpenCL memory model (introduced in version 2.0 of the standard) has received comparatively little academic attention, with the notable exception of the work of Gaster et al. [17], which we discuss further in §7. OpenCL provides a framework for CPU programs to delegate the execution of massively-parallel *kernel* functions, written in a variant of C, to one or more *accelerator devices*, such as GPUs or FPGAs. Threads that execute these kernels are organised into a hierarchy: threads[1] are grouped into *work-groups*, and work-groups are grouped by device. The OpenCL memory model is broadly similar to that of C11, but is extended

[1] Threads in OpenCL are also called *work-items*.

with features such as *memory regions* (which contain locations that are accessible only to a certain subtree of the thread hierarchy), and *memory scopes* (which, when applied to an atomic operation, confine its visibility to a certain subtree of threads).

SC atomics. Our work is distinguished by its focus on the *sequentially consistent* (SC) fragment of these memory models; that is, the semantics of atomics whose memory order is `memory_order_seq_cst`. The chief guarantee provided by this memory order is that all SC atomics in a given execution will execute in some order (say, S) on which all threads mutually agree. Note that these memory models do not *construct* S; they merely postulate the existence of a suitable S.

Sequential consistency is known for its simplicity [26], and indeed, any C11 or OpenCL program using *exclusively* SC atomics would enjoy a simple interleaving semantics. However, when combined with the more relaxed memory orders that C11 and OpenCL also provide, the semantics of SC atomics becomes highly complex, and it is this complexity that we tackle in this paper.

SC atomics are in widespread use, partly because the SC memory order is used when no other is specified, and partly because programmers are routinely advised to use SC atomics prior to optimising their code with the more relaxed memory orders [42 (p. 221)]. Algorithms that make use of SC atomics include Dekker's mutual exclusion algorithm [14], and more generally, multiple-producer-multiple-consumer algorithms that require every consumer to observe the actions of every producer in the same order.[2] As such, it is important that the semantics of SC atomics is clear to programmers, to allow smooth transitioning between the exclusive use of SC (for ease of reasoning) to a mixture of SC and weaker-than-SC atomics (for performance optimisation).

In theory, SC atomics can be avoided by replacing them with mutex-protected non-atomic operations (and simple spinlock mutexes can be implemented using just release and acquire atomics [42 (p. 111)]). In practice, support for SC atomics is non-negotiable if software is to make use of concurrency libraries. This is because the aforementioned replacement of SC atomics must be performed throughout the entire program – in both library code and client code alike – and with the same mutex variable for every operation. Moreover, programs that make extensive use of spinlocks could prove less efficient than those that rely on native SC atomics, and accidental misuse of locks may lead to deadlock.

1.1 Main contributions

Our work aims to provide clearer, simpler foundations for reasoning about C11, enabling a clean extension to OpenCL for heterogeneous programming, and facilitating efficient simulation.

1. Overhauling SC atomics in C11 (§3). The C11 specification devotes around 276 words to explaining the semantics of SC atomics. In our work, we have translated these words into mathematical axioms, carefully strengthened these axioms (without imposing unreasonable demands on the compiler), and then refactored them so that they are expressed as simply as possible. Our revised text

- ✓ is shorter (requiring just 80 words in the same prose style),
- ✓ is simpler (because it reduces seven axioms to just one), and
- ✓ is amenable to more efficient simulation (see below).

Supporting the revised text is a provably-equivalent model that avoids the need to postulate the total order S. Instead, the model constructs a partial order on SC operations, preserving only the edges of S that can affect program behaviours. The enumeration of all candidate S relations is one of the most expensive tasks for memory model simulators like HERD; by reducing S to a partial order, we can dramatically improve simulation performance.

2. Overhauling SC atomics in OpenCL (§5). Our simplifications to the rules governing SC atomics in C11 can be carried over directly to OpenCL, where the same three benefits listed above can be reaped. In the OpenCL setting, however, there is an additional complexity in the semantics of SC atomics. Specifically, the total order S in which all of a program's SC atomics execute is only guaranteed to exist when one of two conditions holds: either all SC atomics in the program's execution use the widest-possible memory scope and only access memory shared between devices, or all SC atomics have their memory scope limited to the current device and never access memory shared between devices. We find that this semantics is unhelpful to programmers, because if *any* SC atomic violates these conditions, then *no* SC atomic is guaranteed to have semantics stronger than acquire/release; this may lead to additional behaviours not anticipated by the programmer. The semantics is simultaneously unhelpful to compiler-writers: a loop-hole that we discovered in the second condition above means that even device-scoped SC atomics must be implemented using expensive inter-device synchronisation.

We have amended the rules that govern SC atomics in OpenCL, so that the SC guarantees do not vanish immediately in the presence of a differently-scoped SC atomic somewhere in the program, but instead degrade gracefully. Our revised rules

- ✓ are shorter and simpler (we can replace 391 words in the specification with 89 words in the same prose style),
- ✓ enable new programming patterns in OpenCL (such as programs that use SC atomics in a natural manner, yet a manner that violates the overly restrictive conditions above),
- ✓ let device-scoped SC atomics be efficiently implemented, and
- ✓ improve the compositionality of OpenCL semantics, and hence the ability to write concurrency libraries (because the behaviour of SC atomics no longer depends on unstable, global conditions).

3. Proving the implementability of our revised models (§3.3, §5.2). Our improvements to the SC axioms in the C11 and OpenCL memory models hinge on slight strengthenings of the models; that is, tweaking some of the axioms so that fewer executions are allowed. This increases the demands on compilers that implement these memory models, so it is important to check that our changes do not invalidate existing compilation schemes. To this end, we prove that all of the formalised C11 compilation schemes of which we are aware (namely, those for Power [8] and x86 [7] machines) remain sound after our changes, and we argue informally that our OpenCL changes preserve the soundness of the only formalised OpenCL compilation scheme (namely, that for AMD GPUs [41]).

1.2 Supporting contributions

In order to justify the claims we make in our main contributions, we have established several supporting artefacts, which we believe are also valuable in their own right.

4. Formalising the OpenCL memory model (§5). The OpenCL specification contains numerous ambiguities, omissions and inconsistencies, which makes it a shaky structure upon which to build an argument about the correctness of an OpenCL program or compiler. The lack of clarity may lead programmers and compiler-writers to cautiously opt for low-efficiency implementations that are easier to guarantee correct. Moreover, there are instances where the OpenCL specification authors have made unnecessarily conservative, programmer-unfriendly decisions in the design of rules for memory consistency. We provide the first mechanised formalisation of the OpenCL memory model. Our formalisation serves to clarify the specification, and can henceforth be used to underpin

[2] `http://en.cppreference.com/w/cpp/atomic/memory_order`

future program logics for verifying OpenCL kernels, and to inform further refinements to the memory model.[3] In particular, we use our rigorous memory model to show that the design decisions of the specification can be made less conservative, offering programmers more flexibility, without placing any additional burden on efficient implementation of the language.

5. Formalising the memory models in `.cat` (§2, §4). We have encoded the C11 and OpenCL memory models in the `.cat` framework [2]. Previous formalisations of the C11 memory model exist, in Isabelle [7], Lem [8] and Coq [38]; here we contribute the first version in `.cat`. We conduct our development work in the `.cat` language because it is the native input format to the HERD memory model simulator, which has a proven record of efficiently simulating a range of CPU machine-level memory models [2 (§8.3)].

6. Extending the HERD memory model simulator (§6). During memory modelling work, tool support for simulating alternative memory models against litmus tests is invaluable. HERD is able to simulate any memory model expressed as a `.cat` file, but in its original incarnation, it supported only machine-level models of CPUs [2] and GPUs [3]. To explore our proposed changes to the C11 and OpenCL memory models, we extended HERD with a module for generating executions of C11 and OpenCL programs, and support for language-level memory models that incorporate 'undefined behaviour' (a notion that is absent from machine-level models). This involved adding around 8000 lines to the original HERD codebase.[4] All of the examples in this paper have been automatically checked with HERD. Using HERD, we have evaluated the impact of our changes to the SC axioms, and found an exponential improvement in simulation performance.

Online material. Our companion webpage provides instructions for downloading HERD and our `.cat` formalisations [11].

2. The C11 memory model in `.cat`

This section describes formally the current C11 memory model.

The semantics of multi-threaded C11 programs is formalised in two stages; the first concerning the thread-local semantics, and the second capturing the memory model. Roughly speaking, the first stage takes as input a C11 program and calculates its set of *executions* (§2.1, §2.2); the second stage then compares each execution to the memory model to determine which executions are actually allowed (§2.3).

There exist several prior formalisations of the C11 memory model [2, 7, 38]. The novelty of this section is the first comprehensive formalisation of the model in the `.cat` framework [2], which enables the use of the efficient HERD simulator [2]. For reasons of space, and because they are orthogonal to the thrust of our contributions, we omit our treatment of the 'consume' memory order, unsequenced races and C11 locks from the paper. However, our `.cat`-based formalisation fully accounts for these features, and is provided on our companion webpage [11].

2.1 C11 programs

A C11 program manipulates a set of shared memory locations.

Definition 1 (Memory locations)**.** Each memory location is declared with either a *non-atomic* or an *atomic* type. That is, $type(l) \in \{\text{atomic}, \text{non-atomic}\}$ for every memory location l.

Definition 2 (Structure of C11 programs)**.** We consider C11 programs of the form $P = \|_{t \in T}\, p_t$, where T is a set of thread identifiers, p_t is a piece of sequential code, and $\|$ is parallel composition. (This static form of parallelism is a simplification of the dynamic thread creation that C11 actually provides.)

Atomic locations can be accessed via *atomic operations*; these include reads, writes, and read-modify-writes (RMWs). C11 also defines *fence* operations. Atomic operations and fences expose the programmer to relaxed memory behaviours; which behaviours are exposed is controlled by the operation's *memory order* parameter.

Definition 3 (Memory orders)**.** The available memory orders in C11 are:

$o ::=$ `RLX` (relaxed)
| `ACQ` (acquire, only for reads/RMWs)
| `REL` (release, only for writes/RMWs)
| `AR` (acquire+release, only for RMWs)
| `SC` (sequentially consistent, the default).

Example 1 (A C11 program)**.** We give below a contrived C11 program that operates on two atomic locations, `x` and `y`, using atomic `store` and `load` operations with a variety of memory orders.

```
                atomic_int *x; atomic_int *y;
store(x,1,RLX); || r1=load(x,RLX); || store(x,2,SC); || store(y,1,SC);
                || r2=load(x,RLX); || r3=load(y,SC); || r4=load(x,SC);
```

2.2 C11 executions

The C11 memory model is defined in terms of program *executions*. An execution X takes the form of a mathematical graph, where each node $e \in E$ is labelled with a run-time memory event (see Def. 4), and the edges connect events performed by the same thread in program order. In other words, an execution is a partial order over a set E of events, and can be thought of as a 'concurrent trace'.

Definition 4 (Event labels)**.** Each event's label characterises the kind of instruction that gave rise to the event, and incorporates up to four attributes, as listed in the first five columns of the following table:

$kind$	loc	$rval$	$wval$	ord	R	W	F	A
$\mathrm{W_{na}}$	(l,		v,	)		✓		
W	(l,		v,	o)		✓		✓
$\mathrm{R_{na}}$	(l,	v,		)	✓			
R	(l,	v,		o)	✓			✓
RMW	(l,	v,	v',	o)	✓	✓		✓
F	(			o)			✓	✓

The labels represent (reading down): non-atomic writes, atomic writes, non-atomic reads, atomic reads, RMWs (which are always atomic), and memory fences. Where relevant, labels contain (reading across): the location being accessed, the value being read, the value being written, and the memory order specified by the programmer. A ✓-mark on the right-hand side of the table indicates that an event with this label belongs to the set R (resp. W, F, A) of events that read (resp. write, are a fence, are atomic). Let $\mathcal{L}$ denote the set of labels.

Definition 5 (Executions)**.** An execution is a tuple $X = (E, I, lbl, thd, sb)$ with the following components.

- E is a set of event identifiers.
- $lbl \in E \to \mathcal{L}$ associates each event with a label. For each event e, $loc(e)$ projects the loc attribute of $lbl(e)$ (if applicable); $rval(e)$, $wval(e)$ and $ord(e)$ provide similar projections.
- $I \subseteq E$ is a set of *initial* events. Every initial event $e \in I$ is a non-atomic write of zero; that is, $kind(e) = \mathrm{W_{na}}$ and $wval(e) = 0$. Moreover, there is exactly one initial event per location.
- $thd \subseteq (E \setminus I)^2$ is an equivalence relation on non-initial events that relates events from the same thread.

[3] Indeed, we have already built upon our formalisation in another piece of work that investigates a proposed extension to the OpenCL memory model [41].

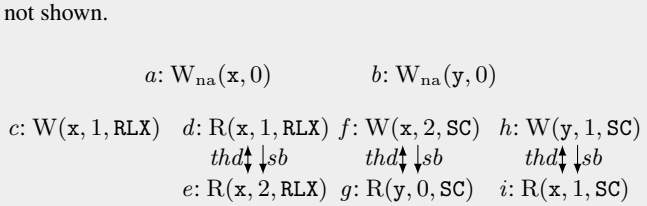
[4] As estimated by `git log`.

- $sb \subseteq thd$ is the *sequenced-before* relation: a strict partial order (i.e., irreflexive and transitive) between events from the same thread, that captures the program order.

Let $\mathbb{X}$ be the set of all executions. Next, we define a number of derived sets and relations over the events of an execution that will prove useful in describing the memory model.

Definition 6 (Derived sets and relations)**.** In the context of an execution (E, I, lbl, thd, sb), we define the relation $=_{loc}$ as $\{(e, e') \in (E \setminus F)^2 \mid loc(e) = loc(e')\}$; it holds between non-fence events that access the same location. The relation $=_{val}$, defined as $\{(e, e') \in W \times R \mid wval(e) = rval(e')\}$, holds when the first event writes the value that the second reads. For each memory order $o \in \{\texttt{RLX}, \texttt{ACQ}, \texttt{REL}, \texttt{AR}, \texttt{SC}\}$, we abbreviate the set $\{e \in A \mid ord(e) = o\}$ as just o. We also define the set $nal = \{e \in E \setminus F \mid type(loc(e)) = \text{non-atomic}\}$ of events that access a non-atomic location.

Example 2 (A C11 execution)**.** The diagram below depicts one execution of the program given in Example 1. The initial events, a and b, are placed above the events of the four parallel threads. Reflexive and transitive edges are elided, and derived relations are not shown.



Basic executions. The first stage of the C11 semantics translates a program into a set of executions called its *basic* set.[5] Each execution in this set is compatible with the instructions of the individual threads, but the set is constructed without considering the behaviour of shared memory, so it provides an over-approximation of the executions that will ultimately be allowed to happen once the whole program and the memory model are taken into account. For instance, the execution in Example 2 is a basic execution of the program in Example 1: the values of the write events correspond to the program text, but the values of the read events are arbitrary and the basic set of all executions ranges over all choices. We do not define formally how the basic executions are constructed, and simply assume their existence for any program we wish to consider. Practical tools such as HERD and Cppmem [7] implement this construction as part of litmus test simulation; the construction is investigated formally in ongoing work by Memarian et al.

Candidate executions. The second stage of the C11 semantics, which is the focus of this paper, takes as input a program's basic execution set and returns the set of *allowed executions*. In order to build the allowed executions, we employ an intermediate structure called a *candidate execution*, which extends an execution with a *witness* that comprises three additional relations, called rf (reads-from), mo (modification order) and S (sequential consistency order).

Definition 7 (Candidate executions)**.** A candidate execution is a pair (X, w) where $X = (E, I, lbl, thd, sb)$ is an execution, and $w = (rf, mo, S)$ is a witness comprising three relations $rf, mo, S \subseteq E^2$. A candidate execution is well-formed, written $wf(X, w)$, if:

- the reads-from relation links write events to read events, such that every read observes exactly one write, and the locations and values match; that is,

$$\left.\begin{array}{l} \forall e \in R.\, \exists! e' \in W.\, (e', e) \in rf \\ \text{and} \quad rf \subseteq (=_{loc} \cap =_{val}) \end{array}\right\} \quad \text{(WfRf)}$$

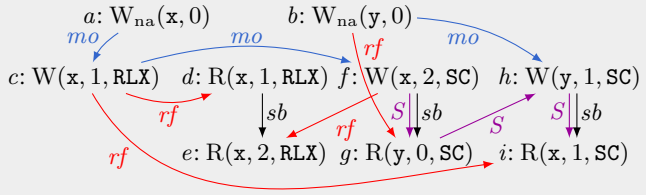
  where $\exists!$ means 'exists unique';
- the modification order relates, in a strict total order, all and only those events that write to the same atomic location; that is,

$$\left.\begin{array}{l} (mo \cup mo^{-1}) = (=_{loc} \cap\, W^2 \setminus nal^2 \setminus id) \\ \text{and} \quad \text{acy}(mo) \end{array}\right\} \quad \text{(WfMo)}$$

  where $\text{acy}(r)$ means that r is acyclic; and
- the S relation relates, in a strict total order, all and only the SC events in an execution; that is,

$$\text{acy}(S) \quad \text{and} \quad (S \cup S^{-1}) = (\texttt{SC}^2 \setminus id) \quad \text{(WfS)}$$

Example 3 (A C11 candidate execution)**.** The diagram below extends the execution in Example 2 with a witness. We elide the thd edges (each column corresponds to one thread). The candidate execution is well-formed, and consistent with the axioms of the memory model (presented next).



2.3 C11 axioms

A candidate execution is deemed *consistent* with the memory model if it satisfies the 12 consistency axioms of Def. 11, which we shall build towards in this subsection. We express the axioms using the `.cat` language [2], a concise language based on the propositional fragment of Tarski's relation calculus [36].

Definition 8 (The `.cat` language)**.** The cat language supports the construction of relations via: union, intersection, difference, complement ($\neg r$), inverse (r^{-1}), reflexive closure ($r^?$), transitive closure (r^+), and relational composition ($r_1 ; r_2$), which is defined such that $(x, z) \in r_1 ; r_2$ if $(x, y) \in r_1$ and $(y, z) \in r_2$ for some y. It also provides the syntax $[s] = \{(e, e) \mid e \in s\}$ for the identity relation (id) restricted to the set s. (These operators can be neatly combined to describe paths through graphs; for instance, $[s_1] ; r_1 ; [s_2] ; r_2 ; [s_3]$ relates s_1-events to those s_3-events that are reachable by following an r_1-edge to an s_2-event and then an r_2-edge.) Each axiom of the memory model must be expressed in the form of an acyclicity ($\text{acy}\ r$), irreflexivity ($\text{irr}\ r$), or emptiness ($\text{empty}\ r$) constraint on some relation r constructed using these operators.

In order to define these axioms, we first need to introduce several derived relations.

Remark 9. In the following, we justify our formal definitions by referring to the C11 standard [21], using the notation §N:n for section N, paragraph n. We refer to the C++11 standard [20], whenever a clause was erroneously omitted from C11. (C11 inherits its memory model from C++11). Similarly, we refer to the C++14 standard [22] in the case of an erroneous omission from C++11. We include these omitted parts because doing so leads to a cleaner model that we believe to be closer to the designers' intent.

Definition 10 (Further derived sets and relations)**.** In the context of a candidate execution $(E, I, lbl, thd, sb, rf, mo, S)$, we define the

[5] This set is sometimes called the 'pre-executions' [7] or the 'opsems' [39].

following subsets of E and relations over E:

$$
\begin{array}{rcl}
acq & \stackrel{\text{def}}{=} & \mathtt{ACQ} \cup \mathtt{AR} \cup (\mathtt{SC} \cap (R \cup F)) \\
rel & \stackrel{\text{def}}{=} & \mathtt{REL} \cup \mathtt{AR} \cup (\mathtt{SC} \cap (W \cup F)) \\
fr & \stackrel{\text{def}}{=} & rf^{-1} ; mo \\
Fsb & \stackrel{\text{def}}{=} & [F] ; sb \\
sbF & \stackrel{\text{def}}{=} & sb ; [F] \\
rs' & \stackrel{\text{def}}{=} & thd \cup (E^2 ; [R \cap W]) \\
rs & \stackrel{\text{def}}{=} & mo \cap rs' \setminus ((mo \setminus rs') ; mo) \\
sw & \stackrel{\text{def}}{=} & ([rel] ; Fsb^? ; [A \cap W] ; rs^? ; rf ; \\
 & & [R \cap A] ; sbF^? ; [acq]) \setminus thd \\
hb & \stackrel{\text{def}}{=} & (sb \cup (I \times \neg I) \cup sw)^+ \\
hbl & \stackrel{\text{def}}{=} & hb \cap =_{loc} \\
vis & \stackrel{\text{def}}{=} & (W \times R) \cap hbl \setminus (hbl ; [W] ; hb) \\
cnf & \stackrel{\text{def}}{=} & ((W \times W) \cup (W \times R) \cup (R \times W)) \cap =_{loc} \\
dr & \stackrel{\text{def}}{=} & cnf \setminus hb \setminus hb^{-1} \setminus A^2 \setminus thd
\end{array}
$$

Commentary. The set acq (resp. rel) contains all events that behave as an acquire (resp. a release).[6] The *from-read* relation (fr) links each read to all those writes that are mo-after the write the read observed [2].

The relation rs captures the *release sequence*, using rs' as a helper. The release sequence of e comprises those events that form a maximal mo-chain, starting from e, of events that either are in e's thread or are RMWs.[7]

Release/acquire synchronisation is captured by the sw relation. This relates an atomic write-release event to an atomic read-acquire event in a different thread if the read obtains its value from the write or its release sequence.[8] If the acquire (resp. release) is a fence, the synchronisation happens via an atomic read (resp. write) sequenced before (resp. after) the fence.[9]

Happens-before (hb) is a transitive relation that includes sequenced-before and synchronisation edges, and puts initial events before all other events.[10] We use hbl to abbreviate happens-before to events on the same location. A write is *visible* (vis) to a read if it is the most recent write to that location in happens-before.[11]

Two events are in *conflict* (cnf) if they access the same location and at least one is a write;[12] these events go on to form a *data race* (dr) if they are unrelated by happens-before, they are not both atomic, and they are in different threads.[13]

We now use the derived relations of Def. 10 to formalise what it means for an execution to be consistent.

Definition 11 (Consistency)**.** A candidate execution $(X, w) = (E, I, lbl, thd, sb, rf, mo, S)$ is *consistent*, written $\text{consistent}(X, w)$, if it is well-formed and it satisfies all of the following axioms:

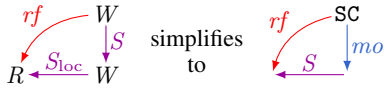
$$
\begin{array}{lll}
\text{irr}(hb) & & \text{(Hb)} \\
\text{irr}((rf^{-1})^? ; mo ; rf^? ; hb) & & \text{(Coh)} \\
\text{irr}(rf ; hb) & & \text{(Rf)} \\
\text{empty}((rf ; [nal]) \setminus vis) & & \text{(NaRf)} \\
\text{irr}(rf \cup (mo ; mo ; rf^{-1}) \cup (mo ; rf)) & & \text{(Rmw)} \\
\text{irr}(S ; r_1) & \text{where } r_1 = hb & \text{(S1)} \\
\text{irr}(S ; r_2) & \text{where } r_2 = Fsb^? ; mo ; sbF^? & \text{(S2)} \\
\text{irr}(S ; r_3) & \text{where } r_3 = rf^{-1} ; [\mathtt{SC}] ; mo & \text{(S3)} \\
\text{irr}((S \setminus (mo ; S)) ; r_4) & \text{where } r_4 = rf^{-1} ; hbl ; [W] & \text{(S4)} \\
\text{irr}(S ; r_5) & \text{where } r_5 = Fsb ; fr & \text{(S5)} \\
\text{irr}(S ; r_6) & \text{where } r_6 = fr ; sbF & \text{(S6)} \\
\text{irr}(S ; r_7) & \text{where } r_7 = Fsb ; fr ; sbF & \text{(S7)}
\end{array}
$$

Commentary. These axioms are equivalent to those in Batty et al.'s Lem formalisation [8], the fidelity of which has been endorsed by the C11 standards committee, but because they are expressed in the `.cat` language, they are markedly more concise. We have established this equivalence using the HOL theorem prover, with the help of a tool we wrote for exporting `.cat` files to Lem, and our proof script is available online [11]. We now explain each axiom in turn.

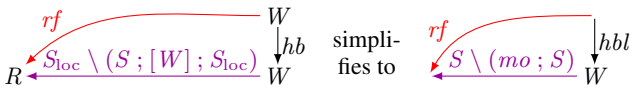
Happens-before must contain no cycles.[14] Requiring irreflexivity here is sufficient (Hb), since hb is transitive. Coherence (Coh) governs the relationship between hb and mo: if the write e_1 is mo-before the write e_2, then e_2 (and any events that read from e_2) must not happen before e_1 (nor before any events that read from e_1).[15] A read must not observe a write that happens after it (Rf),[16] and a read of a non-atomic location must observe a visible write (NaRf).[11] An RMW must observe the immediately-preceding write in mo (Rmw);[17] that is, not itself (first disjunct), nor a too-early write (second disjunct), nor a too-late write (third disjunct).

This leaves the SC axioms, which we present using *where*-clauses for ease of reference later. Axiom S1 states that S must be consistent with happens-before.[18] Axiom S2 governs the relationship between S and mo: if the write e_1 is mo-before the write e_2, then e_2 (and any fences sequenced after e_2) must not come before e_1 (nor before any fences sequenced before e_1) in S.[19]

Axioms S3 and S4 constrain the values that an SC read e_1 of a location l may observe. If there are any SC writes to l preceding e_1 in S, then e_1 must read either from the most recent of these in S – call this e_2 – or from a non-SC write that does not happen before e_2.[18] We encode this requirement as two irreflexivity constraints. First, we wish to rule out reading from an SC write that is not the most recent in S; that is, we wish to forbid cycles of the shape depicted below left, where $S_{\text{loc}} \stackrel{\text{def}}{=} S \cap =_{loc}$. Axiom S3 does this, using the simplified form shown below right.



Second, we require e_1 *not* to read from a write that happens before e_2; that is, we wish to forbid cycles of the shape depicted below left. Axiom S4 does this, using the simplified form shown below right.



Axioms S5, S6 and S7 govern SC fences. If a read e_1 of a location l is sequenced after an SC fence, then e_1 must not read from a write to l that is mo-earlier than the last write to l that precedes the fence in S.[20] In fact, 'the last write' here can be safely

[6] [21 (§7.17.3:3–4)], [21 (§7.17.4.1:2)] [7] [21 (§5.1.2.4:10)] [8] [21 (§5.1.2.4:11)] [9] [21 (§7.17.4:2–4)] [10] [21 (§5.1.2.4:18)], simplified in the absence of `memory_order_consume` [11] [21 (§5.1.2.4:19)] [12] [21 (§5.1.2.4:4)] [13] [21 (§5.1.2.4:25)]

[14] [20 (§1.10:12)] [15] [21 (§5.1.2.4:7)], [21 (§5.1.2.4:22)], [20 (§1.10:17–18)] [16] The specification uses the 'visible sequence of side effects' to phrase this clause [21 (§5.1.2.4:22)], but Batty [6 (§5.3)] has proved that 'happens after' suffices. [17] [21 (§7.17.3:12)] [18] [21 (§7.17.3:6)] [19] [21 (§7.17.3:6)], [20 (§29.3:7)], [22 (§29.3:7)] [20] [21 (§7.17.3:9)]

generalised to 'some write', because being mo-earlier than *some* write to l that precedes the fence in S implies being mo-earlier than the last write, since mo is total (S5). If a write e_2 to location l is sequenced before an SC fence, then any SC read of l that follows the fence in S must not read from a write to l that is mo-earlier than e_2 (S6).[21] Finally, if a read e_1 of location l is sequenced after an SC fence, and a write e_2 to l is sequenced before another SC fence that precedes the first fence in S, then e_1 must not read from a write mo-earlier than e_2 (S7).[22]

A final axiom formalises what it means for an execution to exhibit a fault.

Definition 12 (Faultiness)**.** A candidate execution (X, w) is *faulty*, written $\text{faulty}(X, w)$, if it is consistent and does *not* satisfy the following axiom:

$$\text{empty}(dr). \tag{Dr}$$

If any basic execution can be extended to a faulty candidate execution, then the entire program's behaviour is 'undefined' and any execution is allowed. Otherwise, the allowed executions are those basic executions that can be extended to a consistent candidate execution.

Definition 13 (Allowed executions)**.** Given a set Xs of a program's basic executions, we obtain the program's *allowed* executions as:

$$\text{allowed}(Xs) \stackrel{\text{def}}{=} \begin{array}{l}\textbf{if } \exists X \in Xs.\, \exists w.\, \text{faulty}(X, w) \textbf{ then } \mathbb{X} \\ \textbf{else } \{X \in Xs \mid \exists w.\, \text{consistent}(X, w)\}\end{array}$$

3. Overhauling the SC axioms in C11

The rules for SC axioms in C11, as demonstrated in the previous section, are highly convoluted. In this section, we describe how these rules can be improved in two fairly orthogonal ways. In §3.1, we describe how the total order over SC operations can be replaced with a partial order; this simplification will be demonstrated in §6.2 to dramatically improve the efficiency with which the model can be simulated. In §3.2, we describe a slight strengthening of the model that enables significant simplifications to be made. These simplifications lead to a model that is easier to understand, and should prove easier to work with in a formal setting.

3.1 Reducing S from a total to a partial order

We observe that all but one of the seven SC axioms (Def. 11) can be written in the form $\text{irr}(S\,;\,r)$ for some relational expression r. These r's can be seen as the constraints on the total order S. Axiom S4 is not quite of this form. However, replacing its '$S \setminus (mo\,;\,S)$' with just 'S', to obtain the axiom S4a given below, happens to coincide exactly with an amendment to the model already proposed by Vafeiadis et al. to lend the model more desirable mathematical properties [39 (§4.2)].

$$\text{irr}(S\,;\,r_4) \tag{S4a}$$

Where axiom S4 forbids an SC read to observe any write that happens before the *most recent* SC write in S, axiom S4a forbids it to observe any write that happens before *any* SC write in S. Let us assume here that the uncontroversial amendment of Vafeiadis et al. will be accommodated by the C standards committee.

Lemma 14 (SC order extension principle)**.** *For any relation r, there exists a strict total order S over all SC events that is compatible with r, if and only if r, when restricted unequal SC events, is acyclic. That is:*

$$(\exists S.\, \mathsf{WfS} \wedge \text{irr}(S\,;\,r)) = \text{acy}(\mathsf{SC}^2 \setminus id \cap r).$$

[21] [21 (§7.17.3:10)] [22] [21 (§7.17.3:11)]

Proof. This follows from the well-known order extension principle: that any (strict) partial order can be extended to a (strict) total order. □

We are now in a position to replace the seven irreflexivity axioms with a single acyclicity axiom.

Theorem 1. *There exists a strict total order on SC events that satisfies axioms* S1*,* S2*,* S3*,* S4a*,* S5*,* S6*, and* S7*, if and only if the following* $\mathsf{S}_{\mathsf{partial}}$ *axiom (which states that the union of all the constraints on S, when restricted to unequal SC events, is acyclic) holds:*

$$\text{acy}(\mathsf{SC}^2 \setminus id \cap (r_1 \cup r_2 \cup r_3 \cup r_4 \cup r_5 \cup r_6 \cup r_7)) \tag{$\mathsf{S}_{\mathsf{partial}}$}$$

That is:

$$(\exists S.\, \mathsf{WfS} \wedge \mathsf{S1} \wedge \mathsf{S2} \wedge \mathsf{S3} \wedge \mathsf{S4a} \wedge \mathsf{S5} \wedge \mathsf{S6} \wedge \mathsf{S7}) = \mathsf{S}_{\mathsf{partial}}.$$

Proof.

$$\begin{array}{ll} & \exists S.\, \mathsf{WfS} \wedge \mathsf{S1} \wedge \mathsf{S2} \wedge \mathsf{S3} \wedge \mathsf{S4a} \wedge \mathsf{S5} \wedge \mathsf{S6} \wedge \mathsf{S7} \\ = & \text{[basic properties of relations]} \\ & \exists S.\, \mathsf{WfS} \wedge \text{irr}(S\,;\,(r_1 \cup r_2 \cup r_3 \cup r_4 \cup r_5 \cup r_6 \cup r_7)) \\ = & \text{[by Lemma 14 with } r \text{ instantiated to } r_1 \cup \cdots \cup r_7\text{]} \\ & \text{acy}(\mathsf{SC}^2 \setminus id \cap (r_1 \cup r_2 \cup r_3 \cup r_4 \cup r_5 \cup r_6 \cup r_7)) \end{array}$$ □

Having replaced axioms S1–S7 with the new $\mathsf{S}_{\mathsf{partial}}$ axiom, we no longer require the S relation in execution witnesses. Memory model simulators, such as HERD, typically work by enumerating all executions of a program and then filtering out the consistent subset. Removing the need to iterate through all possible total orders of SC events – a computation that is exponential in the number of SC events – allows simulation performance to be greatly improved, as demonstrated in §6.2.

3.2 A stronger and simpler SC axiom

We now show that it is possible to strengthen the SC semantics without requiring changes to the compilation schemes of any of the C11 target architectures that have an established formal memory model, that is: x86 and Power. The strengthening we propose simplifies the $\mathsf{S}_{\mathsf{partial}}$ axiom significantly and provides stronger guarantees to the programmer.

The proposal for this simplification arises from the observation that the relations considered in the $\mathsf{S}_{\mathsf{partial}}$ axiom are nearly symmetric in hb, mo and fr. In particular, both hb and mo constrain the S order between any combination of SC fences and atomics. The treatment of fr is different: for fr edges that begin or end at a fence, the axioms S5, S6 and S7 ensure that the SC order is constrained to match. When two SC atomics are related by an fr edge (S3 and S4), ordering is only provided when the intermediate access that forms the fr is itself an SC atomic (rule S3), or when the mo edge from the intermediate access of the fr to its target is also covered by a hb edge (rule S4a).

Our proposal is to strengthen the $\mathsf{S}_{\mathsf{partial}}$ axiom, to add these missing constraints so that every fr edge between SC atomics contributes to the S order. We achieve this in our model by removing the [SC] restriction from S3, which results in the following axiom:

$$\text{irr}(S\,;\,fr). \tag{S3a}$$

This change permits a significant simplification to the SC rules that we establish in the following theorem.

Theorem 2. *If rule* S3 *is replaced by* S3a *(that is, if r_3 is replaced with fr in the* $\mathsf{S}_{\mathsf{partial}}$ *axiom) then* $\mathsf{S}_{\mathsf{partial}}$ *becomes equivalent to:*

$$\text{acy}(\mathsf{SC}^2 \setminus id \cap (Fsb^?\,;\,(hb \cup fr \cup mo)\,;\,sbF^?)). \tag{$\mathsf{S}_{\mathsf{simp}}$}$$

That is:

$$\mathrm{acy}(\mathsf{SC}^2 \setminus id \cap (r_1 \cup r_2 \cup fr \cup r_4 \cup r_5 \cup r_6 \cup r_7)) = \mathsf{S_{simp}}.$$

Proof.

$$\begin{aligned}
& r_1 \cup r_2 \cup fr \cup r_4 \cup r_5 \cup r_6 \cup r_7 \\
= \ & \text{[unfolding definitions and combining } fr, r_5, r_6 \text{ and } r_7\text{]} \\
& hb \cup (Fsb^? \,;\, mo \,;\, sbF^?) \cup (Fsb^? \,;\, fr \,;\, sbF^?) \cup r_4 \\
= \ & \text{[since } r_4 \subseteq fr\text{, by WfMo]} \\
& hb \cup (Fsb^? \,;\, mo \,;\, sbF^?) \cup (Fsb^? \,;\, fr \,;\, sbF^?) \\
= \ & \text{[since } hb = (Fsb^? \,;\, hb \,;\, sbF^?)\text{]} \\
& Fsb^? \,;\, (hb \cup fr \cup mo) \,;\, sbF^?
\end{aligned}$$

□

Programming impact. The change presented here does strengthen the memory model; there are executions that were previously allowed that are now forbidden. The simplest we found, which is similar to one used by Vafeiadis et al. [39 (Fig. 6)], is presented in Example 3. We believe Example 3 to be a counterintuitive execution, because the read event i does not observe the most recent write to `x` in S (namely, f), but c, which is mo-earlier than f. The execution is forbidden by axiom S3a because of its $f \xrightarrow{S} i \xrightarrow{fr} f$ cycle. Although the current C11 model allows this execution, mapping this example to the formalised targets of C11 (Power and x86) never yields programs that exhibit it.

3.3 Soundness of existing C11 compilation schemes

There are two C11 targets with formal architectural memory models: x86 and Power. In this subsection, we establish that for both of these architectures, the strengthening does not require a stronger compilation mapping. In both cases, we rely on an existing proof of soundness from the literature. We need only establish that our strengthened $\mathsf{S_{simp}}$ axiom holds.

To establish the soundness of our strengthening for x86, we build on the soundness proof of Batty et al. [7], which uses the axiomatic model of x86 of Owens et al. [31]. To obtain soundness for Power, we build on the soundness proof of Batty et al. [8], which uses the operational Power model of Sarkar et al. [32].

Theorem 3. *Let P be a C11 program that has no faulty executions. If we compile P to x86 according to the mapping given by Batty et al. [7], then every valid x86 execution corresponds to a C11 execution where $\mathsf{S_{simp}}$ holds. If we compile P to Power according to the mapping given by Batty et al. [8], then every valid Power trace is observationally equivalent to a C11 execution where $\mathsf{S_{simp}}$ holds. [Proof in §A]*

Remark 15 (Soundness of the ARMv8 compilation scheme)**.** At the time of writing, work to formalise the ARMv8 specification, and how it implements C11, is ongoing [16]. We understand that it is not currently clear whether the specification is intended to allow or forbid behaviours like our Example 3, and whether the effects of this decision on the C11 memory model are understood. As such, we see our work as a timely intervention in the ongoing argument about how this particular aspect of the ARMv8 specification should evolve and be formalised.

3.4 Effect on the standard

We give below a suggestion for how the wording of the standard could be changed to accommodate our proposal. Our text, which replaces paragraphs 6 and 9–11 of section 7.17.3, is considerably shorter (80 words rather than 276) while preserving the style and terminology of the original. We have retained the total order S in our wording, because we believe it is more intuitive for programmers than an acyclicity condition. Nonetheless, we enable efficient simulation of this model via the $\mathsf{S_{simp}}$ axiom (which is equivalent to the total order formulation, thanks to Lemma 14 with r instantiated to $Fsb^? \,;\, (hb \cup fr \cup mo) \,;\, sbF^?$).

1. A value computation A of an object M *reads before* a side effect B on M if B follows, in the modification order of M, the side effect that A observes.
2. If X reads before Y, or happens before Y, or precedes Y in modification order, then X (and any fences sequenced before X) is *SC-before* Y (and any fences sequenced after Y).
3. There shall be a single total order S on all `memory_order_seq_cst` operations, consistent with the SC-before order.

Summary. This section has described how, having strengthened the original set of axioms (S1 through S7) to use Vafeiadis et al.'s S4a in place of S4, the behaviour of SC operations can be captured by a single axiom ($\mathsf{S_{partial}}$) that allows the total order S to be eliminated from the model. Moreover, if the axioms are further strengthened to use our S3a in place of S3, then that axiom can be greatly simplified ($\mathsf{S_{simp}}$), while still respecting current compilation schemes.

4. Formalising the OpenCL memory model

A principal aim of the OpenCL initiative is to provide functional portability across a plethora of heterogenous many-core devices. The standard is implemented by CPU, GPU and FPGA vendors, and aims to allow applications to be device-agnostic. The OpenCL memory model, introduced in the 2.0 revision of the standard, is inherited from that of C11, but is specialised and extended for heterogeneous programming. The memory model is the sole mechanism for correctly implementing fine-grained concurrent algorithms in a device-agnostic manner. Rigorous foundations for this model are thus vital.

We now describe how our formalisation of the C11 memory model (§2, §3) can be extended to yield the first mechanised formalisation of the full OpenCL memory model. We describe the form of OpenCL programs (§4.1), their executions (§4.2), and the axioms against which these executions are judged (§4.3). We then discuss some interesting features of the memory model: some innocuous quirks (§4.4) and some serious shortcomings (§4.5). The most serious shortcoming relates to the axioms that govern SC atomics, and we propose how to fix this in §5.

For reasons of space, and because they are orthogonal to the thrust of our contributions, we omit our treatment of barrier synchronisation operations and the associated issue of *barrier divergence*. As with the omitted C11 features mentioned in §2, our `.cat`-based formalisation of the OpenCL memory model, provided on our companion webpage [11], fully accounts for these features.

4.1 OpenCL programs

Definition 16 (Structure of OpenCL programs)**.** Building on Def. 2, we consider OpenCL programs of the form

$$P = \|\!\|\!\|_{d \in D} \|\!\|_{w \in W} \|_{t \in T} \, p_{d,w,t}$$

where D, W, and T are sets of device, work-group, and thread identifiers, and each $p_{d,w,t}$ is a piece of sequential code.

Using the notation above, we can write $p \,\|\!\|\!\|\, p'$ to denote a litmus test comprising two threads to be executed on different devices, $p \,\|\!\|\, p'$ for two threads in different work-groups in the same device, and $p \,\|\, p'$ for two threads in the same work-group. We can also write, for example, $p_1 \,\|\, p_2 \,\|\!\|\, p_3 \,\|\, p_4 \,\|\!\|\!\|\, p_5 \,\|\, p_6 \,\|\!\|\, p_7 \,\|\, p_8$, to denote a litmus

test comprising two devices, each executing two work-groups, each containing two threads.

Remark 17 (Limitations)**.** This program structure does not account for *sub-groups*, an optional extension in OpenCL 2.0 that allows threads to synchronise with one another at a level of granularity finer than that of a work-group,[23] nor for further non-OpenCL threads (e.g., POSIX threads) running on the host platform. Moreover, w and t can actually be 1-, 2-, or 3-dimensional vectors, but we make the simplifying assumption that all identifiers are natural numbers.

Recall that locations in C11 are either non-atomic or atomic (Def. 1). OpenCL locations are further declared to reside in a *memory region*.

Definition 18 (Memory regions)**.** We have $region(l) \in \{\texttt{local}, \texttt{global}, \texttt{global\_fgb}\}$ for every location l, where `fgb` stands for fine-grained shared virtual memory (SVM) buffer.[24] There is one `local` region per work-group, containing locations accessible only to that work-group. Locations in the `global` or `global_fgb` region are accessible to all devices. Fences can be performed either on the global memories (`global` and `global_fgb`) or on the local memory, or both simultaneously.

The distinction between `global` and `global_fgb` locations is that the former must not be shared between different devices, while the latter enable inter-device communication. Unlike C11, in which any memory location can be shared between threads, the OpenCL memory model physically prevents certain sharing patterns. For instance, threads from different devices are *forbidden* from conflicting on `global` memory, but are *able* to do so as a result of a programmer fault; in contrast, threads from different work-groups are *unable* to conflict on local memory: the language provides no mechanism through which such a conflict can arise.

Definition 19 (Memory scopes)**.** Atomics in OpenCL are parameterised by a *memory scope*. The three options are

$$\begin{array}{lll} s ::= & \texttt{WG} & \text{(work-group scope)} \\ \mid & \texttt{DV} & \text{(device scope)} \\ \mid & \texttt{ALL} & \text{(system scope).} \end{array}$$

A memory scope specifies how widely visible the effects of the operation should be.

Example 4. The use of memory scopes is illustrated by the following code, which implements the message-passing idiom between two threads in the same work-group.

```
   global int *x; global atomic_int *y;
*x = 42;          || if(load(y,ACQ,WG)==1)
store(y,1,REL,WG); ||   r = *x;
```

Since all accesses to the global location `y` come from the same work-group, those accesses can be performed at `WG` scope (which means that on implementations where each work-group caches global memory, it suffices to read/write those cached values). This scope would be insufficient, and the program deemed faulty, if the threads were in different work-groups – both scopes would have to be upgraded to `DV`.

4.2 OpenCL executions

OpenCL executions extend C11 executions as follows.

Definition 20 (OpenCL event labels)**.** We extend C11 event labels (Def. 4) with an additional *scope* attribute, which assigns a memory scope s to all atomic events. We also subdivide the F label in order to represent fences on global ($\mathrm{F_G}$), local ($\mathrm{F_L}$) and both-global-and-local memory ($\mathrm{F_{GL}}$). The updated table is as follows:

kind	*loc*	*rval*	*wval*	*ord*	*scope*	R	W	F	A
$\mathrm{W_{na}}$	l,		v,				✓		
W	l,		v,	o,	s		✓		✓
$\mathrm{R_{na}}$	l,	v,				✓			
R	l,	v,		o,	s	✓			✓
RMW	l,	v,	v',	o,	s	✓	✓		✓
$\mathrm{F_G}$				o,	s			✓	✓
$\mathrm{F_L}$				o,	s			✓	✓
$\mathrm{F_{GL}}$				o,	s			✓	✓

Definition 21 (OpenCL executions)**.** An OpenCL execution is a tuple $(E, I, lbl, thd, wg, dv, sb)$ where (E, I, lbl, thd, sb) is a C11 execution as in Def. 5, and $wg, dv \subseteq (E \setminus I)^2$ are equivalence relations on non-initial events that relate events from the same work-group and device, respectively. In order to enforce the privacy of `local` locations to a single work-group, we require that if $loc(e) = loc(e') = l$ and $region(l) = \texttt{local}$, then $(e, e') \in wg$.

Definition 22 (Derived sets and relations)**.** In the context of an OpenCL execution $(E, I, lbl, thd, wg, dv, sb)$, we define

$$\begin{array}{rcl} fgb & \stackrel{\text{def}}{=} & \{e \in E \setminus F \mid region(loc(e)) = \texttt{global\_fgb}\} \\ G & \stackrel{\text{def}}{=} & \{e \in F \mid kind(e) \in \{\mathrm{F_G}, \mathrm{F_{GL}}\}\} \cup \\ & & \{e \in E \setminus F \mid region(loc(e)) = \texttt{global}\} \cup fgb \\ L & \stackrel{\text{def}}{=} & \{e \in F \mid kind(e) \in \{\mathrm{F_L}, \mathrm{F_{GL}}\}\} \cup \\ & & \{e \in E \setminus F \mid region(loc(e)) = \texttt{local}\} \end{array}$$

as the sets of events that access, respectively: fine-grained atomic SVM buffers, global memory, and local memory. Also, for each scope s, we abbreviate the set $\{e \in A \mid scope(e) = s\}$ as just s.

Definition 23 (OpenCL candidate executions)**.** Candidate executions in OpenCL, and their well-formedness, are defined in the same way as in C11 (Def. 7).

4.3 OpenCL axioms

We now define and discuss the consistent and faulty predicates for the OpenCL memory model, paying particular attention to each of the departures from C11. We justify our formal definitions by reference to the OpenCL specification [23], writing n/m to denote line m on page n.

Definition 24 (Further derived sets and relations)**.** In the context of a candidate execution $(E, I, lbl, thd, sb, rf, mo, S)$, we define the following subsets of E and relations over E:

$$\begin{array}{rcl} incl & \stackrel{\text{def}}{=} & (\texttt{WG}^2 \cap wg) \cup (\texttt{DV}^2 \cap dv) \cup \texttt{ALL}^2 \\ rsw(r) & \stackrel{\text{def}}{=} & ([r \cap rel]\,;\, Fsb^?\,;\, [W \cap A]\,;\, rs^?\,;\, [r]\,;\, rf\,; \\ & & [R \cap A]\,;\, sbF^?\,;\, [r \cap acq]) \cap incl \setminus thd \\ gsw & \stackrel{\text{def}}{=} & rsw(G) \cup (rsw(L) \cap (\texttt{SC}^2 \cup (G \cap L \cap F)^2)) \\ lsw & \stackrel{\text{def}}{=} & rsw(L) \cup (rsw(G) \cap (\texttt{SC}^2 \cup (G \cap L \cap F)^2)) \\ ghb & \stackrel{\text{def}}{=} & (G^2 \cap (sb \cup (I \times \neg I)) \cup gsw)^+ \\ lhb & \stackrel{\text{def}}{=} & (L^2 \cap (sb \cup (I \times \neg I)) \cup lsw)^+ \\ ghbl & \stackrel{\text{def}}{=} & ghb \cap =_{loc} \\ lhbl & \stackrel{\text{def}}{=} & lhb \cap =_{loc} \\ gvis & \stackrel{\text{def}}{=} & (W \times R) \cap ghbl \setminus (ghbl\,;\, [W]\,;\, ghb) \\ lvis & \stackrel{\text{def}}{=} & (W \times R) \cap lhbl \setminus (lhbl\,;\, [W]\,;\, lhb) \\ hr & \stackrel{\text{def}}{=} & cnf \setminus (ghb \cup lhb) \setminus (ghb \cup lhb)^{-1} \setminus incl \setminus thd \\ iddr & \stackrel{\text{def}}{=} & cnf \setminus dv \setminus fgb^2 \\ sc\text{-}all & \stackrel{\text{def}}{=} & \neg(E^2\,;\, [\texttt{SC} \setminus (\texttt{ALL} \cap fgb)]\,;\, E^2) \end{array}$$

[23] Sub-groups have become a core feature in the recent OpenCL 2.1 specification [23 (p. 22)]. [24] OpenCL also provides `private` regions, each accessible only to one thread, and a read-only `constant` region, but neither of these are interesting from a memory modelling perspective.

$$sc\text{-}dv \quad \stackrel{\text{def}}{=} \quad \neg(E^2 \,;\, [\mathtt{SC} \setminus (\mathtt{DV} \setminus fgb)] \,;\, E^2)$$

Commentary. In OpenCL, only events that have *inclusive* scopes ($incl$) can synchronise: either the events have `WG` scope and are in the same work-group, or they have `DV` scope and are in the same device, or they have `ALL` scope.[25] We shall explain in §4.5 how this notion of scope inclusion is unnecessarily conservative.

The synchronisation relation (rsw) is parameterised by a region r (global or local). The global synchronises-with relation (gsw) includes events that synchronise on global memory,[26] but also includes events that synchronise on *local* memory, providing both events have memory order `SC`,[27] or both are global-and-local fences.[28] Local synchronises-with (lsw) is analogous. Example 6 shows how synchronisation works in the presence of global-and-local fences.

Happens-before is partitioned into global and local versions: global happens-before (ghb) contains global synchronises-with and sequenced-before edges between events on global memory,[29] and local happens-before (lhb) is analogous.[30] See Example 5 for a discussion of the repercussions of this definition of happens-before. Visibility is also split into global ($gvis$) and local ($lvis$) versions.[31]

The *heterogeneous race* (hr)[32] generalises C11's data race (dr, Def. 10), to reflect the fact that in OpenCL, even atomic operations can race when memory scopes are used incorrectly.[33] If two events from different devices conflict on a location that is not in a fine-grained atomic SVM buffer, then they form an *inter-device data race* ($iddr$); such races cannot be ruled out by happens-before edges.[34]

This leaves the $sc\text{-}all$ and $sc\text{-}dv$ relations. In OpenCL, the total order S is only required to exist when

$$\mathtt{SC} \subseteq \mathtt{ALL} \cap fgb \quad \text{or} \quad \mathtt{SC} \subseteq \mathtt{DV} \setminus fgb.$$

The first condition holds when every SC event has `ALL` scope and accesses a `global_fgb` location;[35] the second holds when every SC event has `DV` scope and does *not* access a `global_fgb` location.[36] The relation $sc\text{-}all$ (resp. $sc\text{-}dv$) is the universal relation if the first (resp. $sc\text{-}dv$) condition holds and is the empty relation otherwise. In §4.5, we shall criticise these conditions as being simultaneously too strong for programmers and too weak for compiler-writers.

Definition 25 (Consistency axioms in OpenCL)**.** There are nine consistency axioms. Departures from the C11 consistency axioms (Def. 11) are highlighted.

$$\mathrm{irr}(ghb) \qquad \text{(O-HbG)}$$
$$\mathrm{irr}(lhb) \qquad \text{(O-HbL)}$$
$$\mathrm{irr}((rf^{-1})^? \,;\, mo \,;\, rf^? \,;\, ghb) \qquad \text{(O-CohG)}$$
$$\mathrm{irr}((rf^{-1})^? \,;\, mo \,;\, rf^? \,;\, lhb) \qquad \text{(O-CohL)}$$
$$\mathrm{irr}(rf \,;\, (ghb \cup lhb)) \qquad \text{(O-Rf)}$$
$$\mathrm{empty}((rf \,;\, [G \cap nal]) \setminus gvis) \qquad \text{(O-NaRfG)}$$
$$\mathrm{empty}((rf \,;\, [L \cap nal]) \setminus lvis) \qquad \text{(O-NaRfL)}$$
$$\mathrm{irr}(rf \cup (mo \,;\, mo \,;\, rf^{-1}) \cup (mo \,;\, rf)) \qquad \text{(O-Rmw)}$$
$$\mathrm{acy}(\mathtt{SC}^2 \setminus id \cap (sc\text{-}all \cup sc\text{-}dv) \cap (Fsb^? \,;\, (ghb \cup lhb \cup fr \cup mo) \,;\, sbF^?)) \qquad \text{(O-S}_{\mathsf{simp}}\text{)}$$

Commentary. Both happens-before relations are required to be acyclic (O-HbG, O-HbL).[37] OpenCL requires coherence for both global and local happens before separately (O-CohG, O-CohL).[38] The axioms governing the reads-from relation are carried over from C11 (O-Rf, O-NaRfG, O-NaRfL, O-Rmw), but appropriately divided into global and local versions.[39]

OpenCL defines the same SC axioms that we saw in Def. 11 (S1–S7), but uses $ghb \cup lhb$ in place of hb. We have incorporated into axiom O-S$_{\mathsf{simp}}$ the simplifications that we already discussed in the context of C11 (§3). Intersecting with the $sc\text{-}all$ and $sc\text{-}dv$ conditions means that the acyclicity constraint is only enforced when one of those conditions holds.[40]

Definition 26 (Faultiness in OpenCL)**.** A candidate OpenCL execution is faulty if it is consistent and does *not* satisfy both of the following axioms:

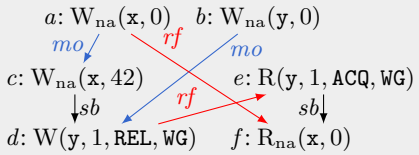
$$\mathrm{empty}(hr) \qquad \text{(O-Hr)}$$
$$\mathrm{empty}(iddr) \qquad \text{(O-Iddr)}$$

4.4 Quirks in the memory model

We present three worked examples that illustrate features of the memory model that may not be obvious from a cursory glance at its axioms. These 'quirks' in the model are distinguished from the technical shortcomings that we save for §4.5.

Our first example illustrates an interesting consequence of OpenCL's separation of happens-before into two distinct relations.

Example 5. Suppose the code of Example 4 were changed so that `y` were declared `local` rather than `global`. Executions such as the one below would then become consistent, which means that a stale value of `x` can be read (event f), even when successful release/acquire synchronisation (between d and e) has occurred.



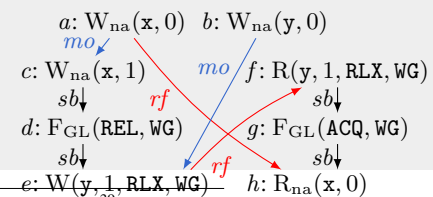
This execution is consistent because the sb edges no longer induce either variety of happens-before, since they link events that act on different memory regions. Worse still, there is now a data race between c and f, which renders the entire program undefined.

We learn from Example 5 that a flag in one memory region cannot be used to protect data in another region. To address this issue, OpenCL provides fences that act on both global *and* local memory simultaneously. These are illustrated in Example 6.

Example 6. The following program uses relaxed (`RLX`) accesses on the local flag `y`, relying instead on the fences to synchronise the threads and enable the global data `x` to be passed.

```
   global int *x; local atomic_int *y;
*x = 42;                    ||  if(load(y,RLX,WG)==1)
fence(GL,REL,WG);           ||  { fence(GL,ACQ,WG);
store(y,1,RLX,WG);          ||    r = *x; }
```

The `fence` instructions successfully prevent the stale value of `x` being read, because the following execution is inconsistent.



[25] [23 (47/16–26)] [26] [23 (51/1–9)] [27] [23 (51/32–33)]
[28] [23 (54/13–16)] [29] [23 (49/3–7)] [30] [23 (49/8–11)]
[31] [23 (49/21–26)] [32] This terminology is due to Hower et al. [18].
[33] [23 (49/29–33)] [34] [23 (58/24–27)] [35] [23 (51/15–17)]
[36] [23 (51/18–20)] [37] [23 (49/12–13)]
[38] [23 (50/11–24)] [39] [23 (49/26–27)], [23 (50/8–9)], [23 (52/22–23)]
[40] [23 (51/14)]

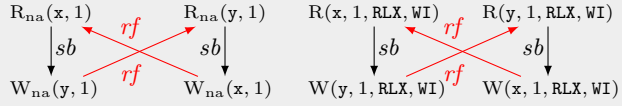
The execution is inconsistent because it has a cycle $h \xrightarrow{rf^{-1}} a \xrightarrow{mo} c \xrightarrow{ghb} h$, in violation of O-CohG. Note that $c \xrightarrow{ghb} h$ holds here because, firstly, (d, g) is in $rsw(L)$ and hence in gsw and ghb, and secondly, (c, d) and (g, h) are both in $sb \cap G^2$ and hence in ghb.

In Example 7, we illuminate the relationship between memory scopes and non-atomic operations. Since scopes can be used to limit atomic operations to certain groups of threads, it is tempting to introduce an additional 'work-item' scope, `WI`, and encode non-atomic events as atomic events whose scope is limited to the current thread. This would make the $\mathrm{W_{na}}$ and $\mathrm{R_{na}}$ labels redundant. An ordinary data race can then be cast as a failure of scope inclusion. However, the differences between non-atomic and atomic operations go beyond racy behaviours, as we shall see in the following example.

Example 7. Consider the following load-buffering litmus test:

```
global int *x, *y;
if (*x==1) *y=1; || if (*y==1) *x=1;
```

The execution of this program that exhibits the relaxed behaviour, in which both comparisons succeed (shown below left), is *not consistent*: both of its reads observe writes that are not visible, in violation of the NaRf axiom. If the non-atomic locations become atomic and the non-atomic operations become work-item-scoped atomics, then this relaxed behaviour (shown below right) becomes *consistent*, since the NaRf restriction no longer applies.



4.5 Problems with the memory model

We present three shortcomings in the OpenCL memory model, which we discovered as a direct result of our formalisation efforts.

Scope inclusion is too strong. The specification provides an overly conservative notion of scope inclusion: two events only have inclusive scopes if their scopes match exactly. This leads to such surprises as the following example.

Example 8. Suppose the code of Example 4 were changed so that the store to `y` now occurs at `DV` scope, but the load of `y` remains at `WG` scope. Although the release scope is clearly 'wide enough', it does not match the acquiring scope, so no synchronisation edge is induced. This leads to two data races: both between the non-atomic accesses of `x`, and between the ill-scoped atomic accesses of `y`.

A resolution proposed by Gaster et al. is to allow the annotated scopes to differ, as long as both are sufficiently wide [17 (§3.1)]. This enables, for instance, a `DV`-scoped write to synchronise with a `WG`-scoped read in the same work-group. The proposal can be formalised in our framework by changing the definition of the $incl$ relation (Def. 24) as follows:

$$\begin{aligned} incl1 &\stackrel{\text{def}}{=} ([\mathtt{WG}] ; wg) \cup ([\mathtt{DV}] ; dv) \cup ([\mathtt{ALL}] ; E^2) \\ new\text{-}incl &\stackrel{\text{def}}{=} incl1 \cap incl1^{-1} \end{aligned}$$

The idea here is to define a one-sided version of scope inclusion first, so that (e_1, e_2) is in $incl1$ if e_1 has a wide enough scope to 'reach' e_2. Requiring this to hold in both directions ensures that both events have sufficient scopes, if not necessarily the same.

The SC axioms are too weak. As encoded in our O-$\mathsf{S_{simp}}$ axiom (Def. 25), SC operations in OpenCL are only guaranteed to provide SC behaviour when one of the $sc\text{-}all$ and $sc\text{-}dv$ conditions holds. Since these are conditions on the whole program, we have a "clear composability problem" [17].

We find several reasons why these conditions are problematic. First, they mean that the default memory scope (which is `DV`) is not sufficient to ensure SC semantics in all situations. Second, any program that includes a `WG`-scoped SC atomic, such as `store(x,1,SC,WG)`, immediately violates the conditions. Third, the conditions are mutually exclusive, so a program that satisfies $sc\text{-}all$ can be combined with another that satisfies $sc\text{-}dv$, with the result satisfying neither. Finally, consider the following example.

Example 9. The following program, comprising two threads in different work-groups on the same device, has SC semantics, which means that it cannot exhibit the relaxed behaviour `r0` $= 0 \wedge$ `r1` $= 0$:

```
                global atomic_int *x, *y;
a: store(x,1);  ||  c: store(y,1);
b: r0 = load(y);||  d: r1 = load(x);
```

Note that the atomic `store` and `load` operations default to the `SC` memory order and the `DV` memory scope, and that condition $sc\text{-}dv$ holds. However, if `global` is changed to `global_fgb`, the relaxed behaviour becomes permissible, because neither condition $sc\text{-}all$ nor $sc\text{-}dv$ holds. Condition $sc\text{-}dv$ no longer holds now that `x` and `y` are in fine-grained atomic SVM buffers, and condition $sc\text{-}all$ does not hold either because the `ALL` scope is not being used.

It is jarring that such a small change, from `global` to `global_fgb`, can legitimise relaxed behaviours. Worse still, such a change may be invisible to the programmer, if they can see only the kernel code: the assignment of locations to SVM buffers occurs only on the host side, and such locations are only marked in a kernel as `global`.

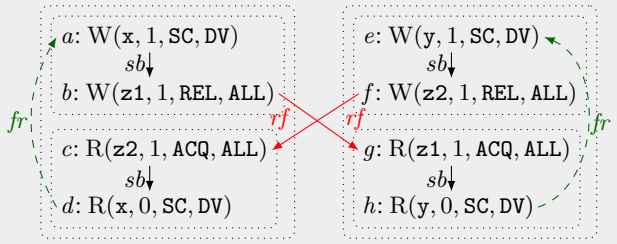
The SC axioms are too strong. Following discussion with members of the Khronos OpenCL working group, we understand that the purpose of condition $sc\text{-}dv$ is to enable efficient implementations of `DV`-scoped SC atomics. The intention of the condition is that if no SC atomic accesses memory shared between devices, they can be implemented without expensive inter-device synchronisation. It was thought not to matter that the specification requires implementations to establish a total order between SC events on different devices, because it is not possible to observe this order without creating an inter-device data race.

In fact, this is not the case. We present in Example 10 a program that satisfies condition $sc\text{-}dv$, and yet is still able to observe the order between SC events in different devices – even though these events are `DV`-scoped and access no memory shared between devices.

Example 10. Consider the following program, which comprises two devices, both executing two threads (stacked vertically). It can be thought of as a 'twisted' version of the store-buffering test.

```
                global atomic_int *x, *y;
          global_fgb atomic_int *z1, *z2;
store(x,1,SC,DV);          ||| store(y,1,SC,DV);
store(z1,1,REL,ALL);       ||| store(z2,1,REL,ALL);
r1 = load(z2,ACQ,ALL)?     ||| r2 = load(z1,ACQ,ALL)?
  load(x,SC,DV) : 1;       |||   load(y,SC,DV) : 1;
```

Two threads in different devices write, using `DV` scope, to distinct `global` locations `x` and `y`, and then write to `global_fgb` flags, using `ALL` scope, to signal that they are done. Meanwhile, two partner threads try to acquire these signals from the opposite device, and if they are successful, they read the location their partner (in the same device as they) wrote to. We are interested in whether these reads can both obtain 0; that is, whether the final state $\{\mathtt{r1} = \mathtt{r2} = 0\}$ is allowed. This final state could only be obtained via the following execution:



where the outer dotted rectangles delimit dv equivalence classes and the inner ones delimit thd equivalence classes.

The execution is inconsistent, and therefore must be forbidden by a compiler. To see this, observe that each rf edge induces a synchronisation (gsw) edge, and hence global happens-before. Since sb edges also contribute to global happens-before, we obtain the cycle $a \xrightarrow{ghb} b \xrightarrow{ghb} g \xrightarrow{ghb} h \xrightarrow{fr} e \xrightarrow{ghb} f \xrightarrow{ghb} c \xrightarrow{ghb} d \xrightarrow{fr} a$. This makes the execution fall foul of O-S$_{\text{simp}}$, which is non-vacuous here because the condition $sc\text{-}dv$ is satisfied.

That the execution in the example above is not allowed implies that OpenCL implementations must make the order of SC write operations visible to all devices, even when those writes are only performed with DV scope. In other words, the current phrasing of the OpenCL memory model demands too much from the compiler-writer to permit an efficient implementation of DV-scoped SC atomics, while in other respects offering too little to the programmer, by guaranteeing SC semantics only when an onerous condition holds.

To summarise: the intent of the Khronos working group was to enable efficient implementation of DV-scoped SC atomics by compilers, at the expense of programmer inconvenience. Instead, our formalisation shows that we have the worst of both worlds: the programmer is inconvenienced, and yet a correct compiler is obliged to enforce inter-device orderings on DV-scoped SC atomics.

5. Overhauling the SC axioms in OpenCL

We describe how the handling of SC atomics in OpenCL can be changed to address the shortcomings identified in §4.5.

Building on a suggestion by Gaster et al. [17 (§7.2)], we propose to eradicate the stringent conditions on the existence of the SC order by simply intersecting the constraints on the SC order with the scope-inclusion relation. This essentially means that the orderings imposed between events by the SC axioms only take effect if those events have inclusive scopes. Under this proposal, which recalls the way C11's synchronisation relation (sw, Def. 10) is intersected with scope-inclusion when producing OpenCL's version (rsw, Def. 24), we do not need to restrict the programmer's usage of SC atomics to certain scopes; instead, the guarantees provided by those SC atomics degrade gracefully as their scopes narrow.

Definition 27 (Proposed SC axiom for OpenCL)**.** The following axiom for SC atomics in OpenCL is obtained from O-S$_{\text{simp}}$ by removing the $sc\text{-}all$ and $sc\text{-}dv$ conditions and instead intersecting with $incl$:

$$\text{acy}(\texttt{SC}^2 \cap (Fsb^? \,;\, (ghb \cup lhb \cup fr \cup mo) \,;\, sbF^?) \cap incl) \qquad \text{(O-S}_{\text{scoped}}\text{)}$$

5.1 Effect on the standard

To accommodate our proposal, we propose that the wording of the OpenCL 2.1 standard [23 (51/14–31 and 51/34–52/13)] be changed to match the text given in §3.4, but with 'happens before' replaced with 'global or local happens before', and 'consistent with the SC-before order' replaced with 'consistent with the SC-before order restricted to operations with inclusive scopes'. This replaces 391 words with 89 words, while retaining the standard's style and terminology.

	OpenCL atomic operation	Assembly instructions
❶	$r = \texttt{load}(x, \texttt{SC}, \texttt{WG})$	$\texttt{LD}\ r\ x$
❷	$r = \texttt{load}(x, \texttt{SC}, \texttt{DV})$	$\texttt{INV}_{\texttt{L1}}$; $\texttt{LD}\ r\ x$; $\texttt{INV}_{\texttt{L1}}$
❸	$\texttt{store}(x, r, \texttt{SC}, \texttt{WG})$	$\texttt{ST}\ r\ x$
❹	$\texttt{store}(x, r, \texttt{SC}, \texttt{DV})$	$\texttt{FLU}_{\texttt{L1}}$; $\texttt{ST}\ r\ x$; $\texttt{FLU}_{\texttt{L1}}$
❺	$r = \texttt{fetch\_inc}(x, \texttt{SC}, \texttt{WG})$	$\texttt{INC}_{\texttt{L1}}\ r\ x$
❻	$r = \texttt{fetch\_inc}(x, \texttt{SC}, \texttt{DV})$	$\texttt{FLU}_{\texttt{L1}}$; $\texttt{INC}_{\texttt{L2}}\ r\ x$; $\texttt{INV}_{\texttt{L1}}$

Table 1. Compiling the revised OpenCL memory model

5.2 Implementability of the new SC axiom

The new O-S$_{\text{scoped}}$ axiom is stronger than the original O-S$_{\text{simp}}$ axiom, so we must confirm that our proposal does not place undue demands on compilers that implement the memory model.

The only published compilation scheme of the OpenCL 2.0 memory model of which we are aware is that published by AMD [30] and later formalised by Wickerson et al. [41]. The scheme compiles the release/acquire fragment of OpenCL atomics, and its soundness has been verified against an operational model of an AMD GPU [41]. In this subsection we describe how the scheme can be extended to support SC atomics, and we demonstrate via a series of examples that the extended scheme meets the requirements of our revised SC axiom. The original compilation scheme does not cater for multiple devices, and does not include fences, and we do not attempt here to extend the scheme to cover these features. As such, this scheme does not engage directly with the problems of inter-device SC atomics that we noted in the previous section; however, it does illustrate how WG- and DV-scoped SC atomics can co-exist.

The AMD compilation scheme. The operational model is quite simple. Each work-group has its own L1 cache, and each device has its own L2 cache. Since the compilation scheme considers only the single-device case, the L2 cache can be safely thought of as the main memory. No instruction reordering is permitted. At any time, the environment can flush a dirty L1 cache entry to the L2 (and thereby make it clean), can fetch an L2 entry to replace a clean L1 entry, and can evict a clean L1 entry.

The semantics of the various assembly instructions can be summarised as follows. $\texttt{LD}\ r\ x$ loads into register r from the nearest cache that contains a valid entry for x; $\texttt{ST}\ r\ x$ stores from r into the local L1 cache, first flushing x's entry therein if it is invalid; $\texttt{INC}_{\texttt{L1}}\ r\ x$ increments x in the local L1 cache; $\texttt{INC}_{\texttt{L2}}\ r\ x$ increments x in the L2 cache, first flushing any dirty entry for x in the local L1 cache; $\texttt{FLU}_{\texttt{L1}}$ flushes all dirty entries in the local L1 cache; and $\texttt{INV}_{\texttt{L1}}$ marks all entries in the local L1 cache as invalid.

The extensions to the compilation scheme are given in Tab. 1. Here, `fetch_inc` stands for 'atomic fetch and increment', and provides a representative of RMW operations in OpenCL.

Correctness of the compilation scheme. Most of the flush and invalidate instructions in the compilation scheme are necessary to ensure correct release/acquire semantics. For SC atomics, we need add only two further instructions: the $\texttt{INV}_{\texttt{L1}}$ before the load in row ❷, and the $\texttt{FLU}_{\texttt{L1}}$ after the store in row ❸. The need for these instructions can be motivated by considering the following two examples, which correspond to the classic store-buffering and IRIW litmus tests.

The memory model requires the program in Example 9 not to produce the final state $\texttt{r0} = \texttt{r1} = 0$. With only release/acquire semantics, the compilation scheme inserts no flush or invalidate instructions between the `store` and the `load` in each thread, and the relaxed behaviour can be observed: both threads might pre-fetch $\texttt{x} = \texttt{y} = 0$ into their respective L1 caches (the threads are

in different work-groups, so they have different L1 caches), then perform their stores, and finally load the L1-cached values of `x` and `y`. However, placing a $\texttt{FLU}_{\texttt{L1}}$ after the store and an $\texttt{INV}_{\texttt{L1}}$ before the load ensures that no sequence of fetching and flushing can lead to the relaxed behaviour. We do not need $\texttt{FLU}_{\texttt{L1}}$ or $\texttt{INV}_{\texttt{L1}}$ instructions before or after the SC increment instruction, because $\texttt{INC}_{\texttt{L2}}$ writes directly to the L2, invalidating the L1 as it does so.

The memory model also requires the IRIW litmus test

```
global atomic_int *x; global atomic_int *y;
store(x,1); ||| store(y,1); ||| r0=load(x); ||| r2=load(y);
                            ||| r1=load(y); ||| r3=load(x);
```

not to produce the final state $\{\texttt{r0} = \texttt{r2} = 1, \texttt{r1} = \texttt{r3} = 0\}$. (Recall that these `store` and `load` operations use memory order `SC` and scope `DV` by default.) Here, an $\texttt{INV}_{\texttt{L1}}$ instruction between each pair of loads is sufficient to rule out such executions.[41]

6. Simulating the memory models with HERD

Our overhaul of SC atomics avoids the requirement for the S relation to be explicitly constructed in execution witnesses. Our hypothesis was that this would lead to improved efficiency in the process of exhaustively enumerating the allowed behaviour of litmus tests that use SC atomics. We now explain how we extended the HERD memory model simulator in order to enable investigation of C11 and OpenCL litmus tests (§6.1), and present experimental results using HERD to compare the efficiency of simulation before and after our overhaul, and also in comparison to the CDSChecker memory model simulator [29] (§6.2). For a family of litmus tests derived from Dekker's algorithm, our results show that our revised axioms lead to an exponential speedup in simulation time using HERD, bringing performance using HERD, which is general-purpose and exhaustive on loop-free programs, much closer to that of CDSChecker, which is specifically tuned for the C11 memory model and is not guaranteed to be exhaustive, even on loop-free programs.

6.1 Extensions to HERD

The version of HERD described by Alglave et al. [2, 3] supports only assembly code: sequences of labelled instructions and gotos. In order to simulate our formalisations of the C11 and OpenCL memory models, we have extended the `.cat` format to support the definition of faulty axioms, and the HERD tool with both a routine for alerting the user when a faulty execution is detected and a module for translating C11 and OpenCL programs into their executions.

We model only a small fragment of the C11 language: enough to encode the litmus tests we found useful for testing our formalisation. We exclude, for example, the address-of operator, compound types, and function calls. We include `if` and `while` blocks, pointer dereferencing, simple expressions, and built-in atomic functions such as `atomic_thread_fence` (C11) and `atomic_work_item_fence` (OpenCL).

6.2 Simulating the C11 model: performance evaluation

We now compare the performance of HERD in enumerating the behaviours of litmus tests (a) when equipped with the original SC axioms in C11 vs. (b) when equipped with our revised SC axioms. We also provide performance results gathered using CDSChecker, a custom-built simulator for the C11 memory model [29]. The HERD tool guarantees exhaustive enumeration of allowed behaviours for a loop-free litmus test; CDSChecker aims for high coverage of behaviours, but is known to be non-exhaustive in general [29].

Recall that Dekker's mutual exclusion algorithm [14] is a key use case for SC atomics. The essential idiom underlying an N-threaded version of Dekker's algorithm is captured by the following N-threaded store-buffering litmus tests:

$$P_N \stackrel{\text{def}}{=} \left(\begin{array}{c|c|c|c} \texttt{store(x}_2\texttt{,1);} & \texttt{store(x}_3\texttt{,1);} & \dots & \texttt{store(x}_1\texttt{,1);} \\ \texttt{r}_1\texttt{=load(x}_1\texttt{);} & \texttt{r}_2\texttt{=load(x}_2\texttt{);} & \dots & \texttt{r}_N\texttt{=load(x}_N\texttt{);} \end{array}\right)$$

that operate on a collection $\{\texttt{x}_1, \dots, \texttt{x}_N\}$ of atomic locations initialised to zero. Recall that atomic `store` and `load` operations use memory order `SC` by default. Dekker's algorithm requires that it is *not* possible to observe the final state where $\texttt{r}_1 = \dots = \texttt{r}_N = 0$; only `SC` is strong enough to rule out this relaxed behaviour.

We use the family $(P_N)_{N \in \mathbb{N}}$ to assess the scalability of the two versions of HERD and of CDSChecker. Figure 1 shows the time each tool takes to simulate P_N as N increases.[42] Experiments were conducted on a 3.1 GHz MacBook Pro, and each data point represents the mean of ten runs. We do not include error bars because the standard deviation is negligible. The original memory model, naively implemented in HERD, times out on just 4 threads. This is because it iterates over all $(2N)!$ orders of the $2N$ SC events that are in every execution of P_N. When HERD is provided with our revised memory model, simulation times greatly improve. Bearing in mind the logarithmic y-axis, the performance of both HERD on the revised memory model and CDSChecker appears to scale exponentially with N, which meets expectations since P_N has $2^N - 1$ unique final states. Still, CDSChecker significantly outperforms HERD when simulating P_N, and on several other programs that we tried. This is because CDSChecker, unlike HERD, is optimised specifically for the C11 memory model, through the use of such techniques as the early elimination of infeasible executions, and a variant of dynamic partial order reduction (DPOR) [15] on the S order. In fact, we conjecture that the use of DPOR here has an effect similar to our proposal to rephrase the memory model with S as a partial order.

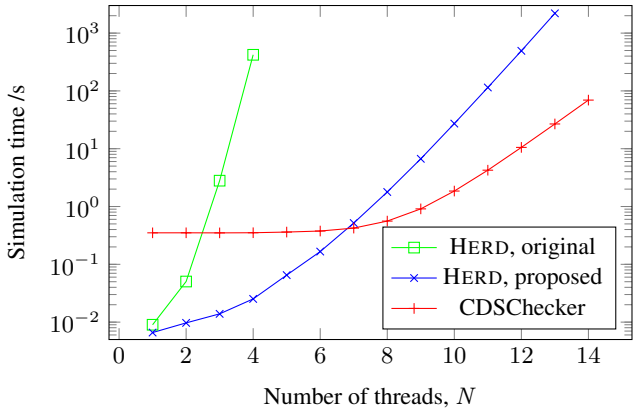


Figure 1. Time to simulate an N-threaded store-buffering test

Figure 1 demonstrates that simply by tweaking the axioms that define the memory model, simulation time can be dramatically decreased, without the need to implement complex optimisations, such as DPOR, that make it difficult to assess the soundness and completeness of the tool. It happens that CDSChecker*is* exhaustive on all of our P_N programs, but we remark that we can only be sure of this because of HERD.

7. Related work

The C11 memory model has been formalised several times. Batty et al. [7] present a comprehensive formalisation using Lem [28]. Vafeiadis et al. [38, 39] and Batty et al. [9] have also formalised

[41] On weaker models, such as Power, that are not *multi-copy atomic* [35], further synchronisation would be required between the loads.

[42] We used HERD revision 88ff189 (`http://github.com/herd/herdtools`) and CDSChecker revision 7c51087 (`git://demsky.eecs.uci.edu/model-checker.git`).

slightly simplified variations. Alglave et al. have formalised a release/acquire fragment of the C11 model (without release sequences, fences, non-atomics, or data races) in the `.cat` language, and have shown it to be an instance of their generic axiomatic memory model [2]. We use the `.cat` language in our work too, but our comprehensive model, which incorporates undefined behaviours and a richer language of events, no longer fits within their generic framework.

We remark that in the absence of fences, our $\mathsf{S}_{\mathsf{simp}}$ axiom (see Theorem 2) forbids the same dependency cycles that Shasha and Snir characterise as violations of sequential consistency [33]. In a sense, one contribution of our paper is to simplify the semantics of C11's SC atomics to the point where it can be defined, for the first time, in the Shasha–Snir style.

Criticisms of the C11 model. Batty et al. describe a fundamental problem in the structure of the C11, C++11, C++14 and OpenCL memory models: the so-called "thin-air" executions [10]. This is a difficult open problem requiring a radically different approach; we do not address it here.

Vafeiadis et al. note that the current rules governing SC atomics break desirable properties of the memory model, harming the prospect of reasoning above it, and they propose a strengthening of the model to fix this [39]. Our proposal builds on theirs (§3.1), but goes further (§3.2), arriving at a substantially simpler model. A similar proposal was in fact considered by Vafeiadis et al. in the context of the original total-order SC axioms [39], but abandoned over concerns that it would invalidate the existing Power compilation scheme. In our work, we have demonstrated that such a proposal *is* in fact valid on Power (and x86).

We note that despite our strengthening, SC fences remain too weak to restore sequential consistency in all circumstances, even when placed between every pair of accesses. This weakness was intentional in C11 to permit efficient implementation over Intel's Itanium architecture [6], but it does harm programmability. Lahav et al. [25] have proposed an alternative implementation of SC fences, in terms of acquire/release RMWs on a distinguished location, that always restores sequential consistency.

The OpenCL 2.0 memory model has recently been described by Gaster et al. [17], as an instance of a heterogeneous race-free (HRF) model [18]. Our work improves on theirs in several ways. A key shortcoming of their work is its relative informality: it lacks the mathematical precision that is required to resolve all the details of the OpenCL memory model. Our formalisation, in contrast, is precise enough to be executed by a machine (cf. §6). Moreover, their characterisation of the OpenCL memory model has several technical issues. It replaces the specification's modification order (which orders atomic write events) with a *coherence order* (which orders both read *and* write events) without proving that the intent of the specification is preserved by this change. Another infidelity to the specification is the omission of release sequences, which prohibits the correct treatment of release-fences. Indeed, Gaster et al. include no formal treatment of fences at all, describing their behaviour only in prose. Our `.cat` presentation of the OpenCL memory model treats release sequences and fences in full. Its informality aside, Gaster et al.'s work contains numerous insights into the design and workings of the OpenCL memory model, and provided a valuable basis for our formalisation efforts.

We have already begun to build on top of the formalisation of the OpenCL memory model presented here, as part of our investigations into the semantics of a proposed extension to OpenCL called *remote-scope promotion* [41]. That work, which has already been published, describes only a small 'release/acquire' fragment of the OpenCL memory model, while the current paper describes the full model, including the interesting and important SC and relaxed atomics.

Implementations of the OpenCL memory model. AMD and Intel have recently released OpenCL 2.0-compliant implementations [4, 19]. We are aware only of one implementation of the OpenCL memory model that has been formalised: namely, a compilation scheme from OpenCL (extended with a feature called remote-scope promotion [30]) to a model of next-generation AMD GPUs [41]. Alglave et al. present an experimentally-validated axiomatic model of an Nvidia GPU [3], which could provide another compilation target for our OpenCL memory model. However, we find that their model is too weak to admit an efficient mapping from OpenCL. Specifically, it does not provide the property of *cumulativity*: synchronisation at one scope cannot be chained with further synchronisation at a wider scope to induce overall synchronisation between the two end-points. Since cumulativity is a property required by the OpenCL memory model, we deduce that the OpenCL compiler must, very expensively, treat all operations as having the widest scope.

Memory model simulators other than HERD that are capable of handling the C11 model include Cppmem [7], Nitpick [13] and CDSChecker [29]. We did not include Cppmem and Nitpick in our tool comparison (§6.2) because Norris et al. have already demonstrated that CDSChecker's performance is far superior [29].

Because it is highly optimised for the C11 memory model, CDSChecker continues to outperform HERD even on the revised model. HERD on the other hand is deliberately designed *not* to be optimised for a particular model, but to be instead a generic memory model simulator. A key advantage of using a generic memory model simulator like HERD is that it is easy to tinker with the model during the development process: one must only modify a text file and restart HERD in order to explore the impact of a proposed change. Indeed, this ease of modification, together with the challenge of expressing the C11 model in the very concise `.cat` language, inspired our discovery of the simpler SC axioms described in this paper. Moreover, where CDSChecker is designed for efficiency, sometimes at the cost of fidelity to the memory model (the lack of self-satisfying conditionals, for instance, is a source of incompleteness in CDSChecker), our formalisation and simulator are designed primarily to represent the memory model as closely as possible.

CDSChecker obtains its main performance benefits by exploring partial modification orders. It is therefore natural to ask whether the memory model could be revised to accommodate partial modification orders in the same way that we have incorporated a partial S order. We believe that this is not straightforwardly possible without changing the model: our partial order reduction on S hinges on its constraints all having the form $\mathrm{irr}(S\,;r)$ for some r, but this is not the case for mo – see axiom Rmw (Def. 11) for instance.

8. Conclusion

Our overhaul of the semantics of SC atomics and fences provides four main benefits in relation to the C11 and OpenCL memory models: more efficient exploration of the behaviours of litmus tests (cf. §3.1, §6.2); refined specification text that we argue is easier for programmers and compiler-writers to understand (cf. §3.4); improved usability of the languages by programmers (cf. §5); and opportunities for compiler-writers to produce more efficient implementations (cf. §5). We argue that our proposed changes to the memory models validate all of the formalised C11 and OpenCL compilation schemes of which we are aware.

A topic for future research is the consideration of memory consistency between OpenCL devices and the host application that launches kernels on these devices; our treatment in this paper focuses solely on interactions between kernel threads. We also plan to use our memory model as a basis for reasoning about OpenCL programs, extending the capabilities of tools such as GPUVerify [12], where

existing support for atomic operations is limited and not based on formal foundations [5].

Acknowledgements. Luc Maranget kindly advised on our extensions to HERD. We thank Jade Alglave, Nathan Chong, Benedict Gaster, Vinod Grover, Lee Howes, Jeroen Ketema, Matthew Parkinson, Peter Sewell, Tyler Sorensen, and our anonymous reviewers for their feedback and encouragement. This work was supported by the EPSRC (grants EP/K011499/1, EP/I020357/1, EP/K015168/1, and EP/I01236/1), and by the EU FP7 project CARP (project number 287767).

References

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

[2] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: modelling, simulation, testing, and data-mining for weak memory. *TOPLAS*, 2014.

[3] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU concurrency: weak behaviours and programming assumptions. In *ASPLOS*, 2015.

[4] AMD Developer Central. *AMD APP SDK 3.0 released, featuring OpenCL 2.0*, 2015. URL `http://developer.amd.com/community/blog/2015/08/26/introducing-app-sdk-30-opencl-2/`.

[5] E. Bardsley and A. F. Donaldson. Warps and atomics: Beyond barrier synchronization in the verification of GPU kernels. In *NASA Formal Methods*, 2014.

[6] M. Batty. *The C11 and C++11 Concurrency Model*. PhD thesis, University of Cambridge, October 2014.

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

[8] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *POPL*, 2012.

[9] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013.

[10] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In *ESOP*, 2015.

[11] M. Batty, A. F. Donaldson, and J. Wickerson. Overhauling SC atomics in C11 and OpenCL – companion webpage, 2016. URL `http://multicore.doc.ic.ac.uk/overhauling`.

[12] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, and J. Wickerson. The design and implementation of a verification technique for GPU kernels. *TOPLAS*, 2015.

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

[14] E. W. Dijkstra. Cooperating sequential processes (1965). In P. Brinch Hansen, editor, *The Origin of Concurrent Programming*, pages 65–138. Springer, 2002.

[15] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, 2005.

[16] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *POPL*, 2016.

[17] B. R. Gaster, D. R. Hower, and L. Howes. HRF-Relaxed: Adapting HRF to the complexities of industrial heterogeneous memory models. *ACM Transactions on Architecture and Code Optimization*, 2015.

[18] D. R. Hower, B. M. Beckmann, B. R. Gaster, B. A. Hechtman, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Adapting data-race-free memory consistency for heterogeneous systems. In *ASPLOS*, 2014.

[19] Intel Developer Zone. *OpenCL 2.0 is here!*, 2014. URL `https://software.intel.com/en-us/forums/opencl/topic/531074`.

[20] ISO/IEC. *Programming languages – C++*. International standard 14882:2011, 2011.

[21] ISO/IEC. *Programming languages – C*. International standard 9899:2011, 2011.

[22] ISO/IEC. *Programming languages – C++*. International standard 14882:2014, 2014.

[23] Khronos Group. *The OpenCL Specification*. Version 2.1, Revision 8, 2015.

[24] Khronos Group News Archives. *Freescale to spark innovation and open development for autonomous driving systems with OpenCL*, 2014. URL `https://www.khronos.org/news/archives/2014/11`.

[25] O. Lahav, N. Giannarakis, and V. Vafeiadis. Taming release-acquire consistency. In *POPL*, 2016.

[26] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), 1979.

[27] R. Morriset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI*, 2013.

[28] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. In *ICFP*, 2014.

[29] B. Norris and B. Demsky. CDSChecker: Checking concurrent data structures written with C/C++ atomics. In *OOPSLA*, 2013.

[30] M. S. Orr, S. Che, A. Yilmazer, B. M. Beckmann, M. D. Hill, and D. A. Wood. Synchronization using remote-scope promotion. In *ASPLOS*, 2015.

[31] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.

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

[33] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *TOPLAS*, 10(2), 1988.

[34] M. Steuwer and S. Gorlatch. High-level programming for medical imaging on multi-GPU systems using the SkelCL library. In *ICCS*, 2013.

[35] J. M. Stone and R. P. Fitzgerald. Storage in the PowerPC. In *IEEE Micro*, 1995.

[36] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6 (3):73–89, 1941.

[37] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, 2014.

[38] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*, 2013.

[39] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, 2015.

[40] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, 2008.

[41] J. Wickerson, M. Batty, A. F. Donaldson, and B. M. Beckmann. Remote-scope promotion: clarified, rectified, and verified. In *OOPSLA*, 2015.

[42] A. Williams. *C++ Concurrency in Action*. Manning, 2012.

A. Proof of Theorem 3

The following theorem states that the x86 and Power compilation schemes for C11, as given in Tab. 2, remain sound in the presence of our revised SC axiom, $\mathsf{S}_{\mathsf{simp}}$.

	C11 operation	x86	Power
❶	$r = \texttt{load}(x, \texttt{SC})$	lock xadd(0)	sync; ld; cmp; bc; isync
❷	$\texttt{store}(x, r, \texttt{SC})$	lock xchg	sync; st
❸	$r = \texttt{fence}(x, \texttt{SC})$	mfence	sync

Table 2. Compiling the C11 SC atomics

Theorem 3 (repeated from §3.3)**.** *Let P be a C11 program that has no faulty executions. If we compile P to x86 according to the mapping given by Batty et al. [7], then every valid x86 execution corresponds to a C11 execution where $\mathsf{S_{simp}}$ holds. If we compile P to Power according to the mapping given by Batty et al. [8], then every valid Power trace is observationally equivalent to a C11 execution where $\mathsf{S_{simp}}$ holds.*

Proof (x86 case). The axiomatic model of Owens et al. restricts the behaviour of memory using a partial order over x86 memory events called $memory\text{-}order$. The proof of Batty et al. [7] constructs the relations of the C11 execution using $memory\text{-}order$; modification order (mo) and reads-from (rf), in particular, are projected from it. Here we rely on several properties of $memory\text{-}order$ as set out by Owens et al.: when restricted to writes, it is a linear order; program order between two events is included in memory order if there is an intervening fence or if either instruction is locked; program-order edges from reads to later events are included; and a read observes the most recent preceding write in $memory\text{-}order$.

We proceed by contradiction, showing that given the construction of rf and mo used in the proof of Batty et al., any cycle in $\mathtt{SC}^2 \setminus id \cap (Fsb^? ; (hb \cup fr \cup mo) ; sbF^?)$ implies either the existence of a cycle in $memory\text{-}order$, or an inconsistent rf edge.

Any cycle in the relation is made up of mo, hb and fr edges, possibly linked with sb edges. The mo, hb and sb edges all imply corresponding $memory\text{-}order$ edges. To see this, note:

- $memory\text{-}order$ is a linear order over writes;
- any hb edge in the $\mathsf{S_{simp}}$ relation begins with either a fence or a locked instruction, so sb edges correspond to $memory\text{-}order$ edges, then there may be a chain of edges in mo ; rf ; sb, where the final sb edge is headed by a read, so transitivity implies that hb corresponds to $memory\text{-}order$; and
- any sb edges are either between locked instructions or have a fence between accesses, and so correspond to $memory\text{-}order$ edges.

Finally, if the cycle contains an fr edge, then $memory\text{-}order$ cannot contradict this: the read would become inconsistent in the x86 execution. Then for a given $\mathsf{S_{simp}}$ cycle, we have a sequence of $memory\text{-}order$ edges that would form a cycle if not for holes corresponding to fr edges.

We now show that there is indeed a cycle in $memory\text{-}order$. Consider an fr edge: its head is a read, so the preceding edge must either be a sb edge or a hb edge. If it is a sb edge then either the head of that edge is a write, or the edge that precedes that is an hb edge. In all cases, the read at the head of the fr edge is preceded in hb by a write. As $memory\text{-}order$ is total over writes, it must order this preceding write's x86 counterpart before the write in the tail of the fr edge. We use this fact to construct a cycle in $memory\text{-}order$, a contradiction. □

Proof (Power case). In their proof, Batty et al. construct a C11 execution from a Power trace such that Power coherence and rf edges match their constructed C11 counterparts. In proving the SC axioms hold over the execution, they prove a property, $good\text{-}sc$, that establishes a total order over the SC atomics of the execution that contains po, co, fr and an extended variant of reads from, erf, each restricted to the SC atomics. The SC fences are added to this relation in a way that is consistent with co and rf in the rest of the execution, preserving the invariant that it is a strict partial order. The following edges become part of the SC order: $[\mathtt{SC}]$; $po^?$; $(rf^{-1})^?$; co ; $rf^?$; $po^?$; $[F \cap \mathtt{SC}]$ and $[F \cap \mathtt{SC}]$; $po^?$; $(rf^{-1})^?$; co ; $rf^?$; $po^?$; $[\mathtt{SC}]$. The proof goes on to show that hb restricted to the SC actions is a subset of the total order. The construction of mo and rf in the C11 execution follow the Power trace directly, so $good\text{-}sc$ together with the addition of the fence edges, which contain $Fsb^? ; fr ; sbF^?$ and $Fsb^? ; co ; sbF^?$, show the acyclicity of $\mathtt{SC}^2 \setminus id \cap (Fsb^? ; (hb \cup fr \cup mo) ; sbF^?)$ directly. □