Taming the complexities of the C11 and OpenCL memory models

John Wickerson[1] and Mark Batty[2]

[1] Imperial College London
[2] University of Cambridge

Abstract. We study how the C11 memory model can be simplified and how it can be extended. Our first contribution is to propose a mild strengthening of the model that enables the rules pertaining to sequentially-consistent (SC) operations to be significantly simplified. We eliminate one of the total orders that candidate executions must range over, leading to a model that is significantly faster to simulate. Our endeavours to simplify the C11 memory model are particularly timely, now that it provides a foundation for memory models of more exotic languages – such as OpenCL 2.0, an extension of C for programming heterogeneous systems composed of CPUs, GPUs and other devices. Our second contribution is the first mechanised formalisation of the OpenCL 2.0 memory model, extending our simplified C11 model. Our C11 and OpenCL memory model formalisations are expressed in the `.cat` language of Alglave et al., the native input format of the herd memory model simulator. Originally designed for the efficient simulation of hardware memory models, we have extended herd to support language-level memory models.

1 Introduction

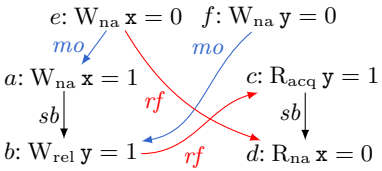
1.1 Motivation

Computer processors, be they CPUs or GPUs, have intricate memory hierarchies that expose *relaxed* behaviours: those that violate sequential consistency (SC) [14]. Relaxed memory is extremely subtle, and has led to bugs in language specifications [7, 6], deployed processors [2], compilers [18, 15], and vendor-endorsed programming idioms [1]. As such, defining *formal models* of relaxed memory – mathematical specifications of the allowable behaviours of a multi-threaded program with respect to accesses to shared memory locations – is an important area of research. Formal models can help to resolve ambiguities and inconsistencies in the specification, can underpin work to build verification techniques and tools, and can be used to build simulators for exploring the effects of the memory model. Such simulators are useful for confirming the fidelity of a formalisation to the original specification, and for ultimately acting as a machine-independent oracle to check systematically that compilers, machines and programs all conform to their specification.

The C11 memory model The 2011 revision of the C programming language (C11) provides an *atomics* library – atomic reads, writes, read-modify-writes (RMWs) and

```
int *x; atomic_int *y;
*x = 1;                  ||  if(load(y,o.acq))
store(y,1,                   r = *x;
  o.rel);                ||
```

(a) An C11 program

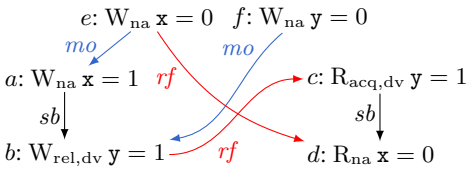


(b) An execution *inconsistent* in C11

Fig. 1. The MP idiom in C11

```
global int *x; global atomic_int *y;
*x = 1;                  ||  if(load(y,o.acq,s.dv))
store(y,1,                   r = *x;
  o.rel,s.dv);           ||
```

(a) An OpenCL program



(b) An execution *inconsistent* in OpenCL

Fig. 2. The MP idiom in OpenCL

fences – that enable fine-grained lock-free programming, but that expose the programmer to relaxed behaviours. Which behaviours are exposed is controlled by each operation's *memory order* parameter. C11 defines the semantics of atomic operations using an *axiomatic* memory model: a model that takes an over-approximation of a program's behaviours (its *candidate executions*) and provides axioms to whittle this set down to the *consistent* executions that are allowed to happen. C11 has a *catch-fire* semantics [9, §5]: it defines certain consistent executions (e.g. data races) as programmer faults, and allows programs that admit a faulty execution to do anything – even 'catch fire'.

Example 1. The message-passing (MP) idiom, illustrated in Fig. 1a, is central to the C11 design. The left thread writes to a normal *non-atomic* (na) location x and then to an atomic flag y using the *release* memory order ($o.rel$). The right thread reads y using the *acquire* order ($o.acq$), and if it sees 1, its non-atomic read of x is also guaranteed to see 1. C11 makes this guarantee by rejecting as *inconsistent* those executions – such as the one shown in Fig. 1b – in which the right thread observes the initial value of x. (As we shall explain in §2, this execution violates an axiom governing non-atomic reads.)

The OpenCL 2.0 memory model Exploiting GPUs for general-purpose, high-performance computing has seen huge recent success in such areas as statistical modelling, computational finance, and medical imaging. A key player in this endeavour is OpenCL [13], which is a framework for programming *heterogeneous systems* composed of CPUs, GPUs, and other devices, and is supported and developed by major hardware vendors such as Altera, AMD, ARM, Intel, Nvidia, and Qualcomm.

OpenCL 2.0 provides a C-like language whose memory model inherits most of the complexity of C11, but also adds nuances peculiar to heterogeneous programming. A key idea of OpenCL is that threads (called *work-items*) are organised into *work-groups*, and work-groups into *devices*. This reflects the underlying structure of a typical GPU device, in which the (many thousands of) processing cores are organised into *compute units*. In the quest for maximal performance, memory locations and memory consistency guarantees can be localised to certain subsets of threads: memory can be allocated in *private* (per thread), *local* (per work-group) and *global* regions, and the

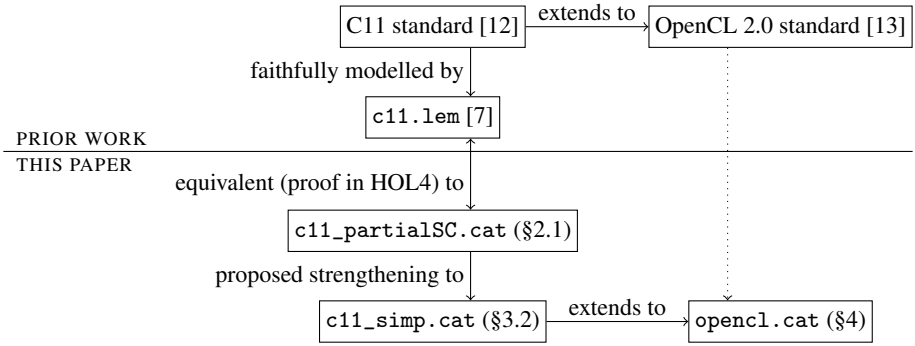


Fig. 3. Our main contributions and the section in which each is described

effects of individual instructions can be confined to a particular part of the hierarchy by the use of *memory scopes*.

Example 2. Figure 2a revisits the MP idiom in OpenCL. Both `x` and `y` reside in global memory, and the two threads are in different work-groups but the same device, as indicated by $|||$. (We use $\|$ to place threads in the same work-group, and $||||$ to place them in different devices.) The atomic operations are now parameterised by a memory scope: here, the $s.dv$ (device) scope ensures, informally, that their effects need only be propagated throughout the current device. The candidate shown in Fig. 2b is ruled inconsistent just as in the C11 case. Note, though, that the execution would be consistent (and moreover, racy) if either memory scope were narrowed to $s.wg$ (work-group): this would be insufficient to ensure the necessary release–acquire synchronisation.

1.2 Our work

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.

Extending the herd simulator to language-level models We have extended the herd memory model simulator to handle language-level models with catch-fire semantics. We target herd because it has a proven record of efficiently simulating machine-level memory models [3, §8.3]. Herd is a generic simulator, able to simulate any memory model specified in `.cat` [3], a concise language based on the relation calculus [19]. Accordingly, we use `.cat` to express our memory model formalisations.

Simplifying the C11 model We propose a significant simplification to the rules governing SC operations in the C11 memory model (a change that can be inherited by OpenCL). The specification states that a candidate execution is consistent only if there exists a total order (say S) among all of its SC operations and S satisfies certain properties. We describe an alternative axiom, proven equivalent in the HOL4 theorem prover, that implies the existence of a suitable total order, *without actually constructing it*. Since

iterating over all possible total orders is a very expensive computation, this observation leads to a model (see `c11_partialSC.cat` in Fig. 3) that is significantly faster to simulate. Moreover, we describe how the model can be slightly strengthened to enable this axiom to be substantially simplified (see `c11_simp.cat` in Fig. 3).

Formalising and improving the OpenCL 2.0 memory model We extend our C11 memory model to obtain the first mechanised formalisation of the OpenCL memory model (see `opencl.cat` in Fig. 3). How our work improves on a previous paper formalisation by Gaster et al. [10] is discussed in §5. We also pin down the precise conditions under which the C11 model is a special case of the OpenCL one.

Contributions The original contributions of this paper are:

- an extension to the herd memory model simulator, enabling support for language-level memory models such as C11 and OpenCL (§2.2);
- a provably-equivalent formal C11 model that eliminates the total order over SC operations for efficient simulation (§3.1);
- a proposed strengthening of the C11 memory model that greatly simplifies the rules governing SC atomics (§3.2); and
- a mechanised formalisation of the OpenCL 2.0 memory model, and an explanation of how the C11 model can be obtained as a special case (§4).

Companion material We provide appendices, our HOL4 proof scripts, and instructions for downloading herd, online at `http://multicore.doc.ic.ac.uk/opencl/`.

2 The C11 memory model

This section introduces the C11 memory model, describes the structure our formalisation of it in the `.cat` language, and explains how we extended the herd tool in order to be able to simulate it.

Memory orders As mentioned in the introduction, atomic operations in C11 take a parameter, called a *memory order*, that defines which memory consistency guarantees they provide. The strongest guarantees are provided by $o.sc$ (sequentially-consistent); $o.rlx$ (relaxed) provides none; while $o.acq$ (acquire, for reads), $o.rel$ (release, for writes) and $o.ar$ (acquire-and-release, for RMWs) lie in-between. For simplicity, we omit the 'consume' memory order, which does not appear in OpenCL and, as Vafeiadis et al. remark, is little used elsewhere [21, §2].

Memory locations Each memory access acts on a particular *location* in memory, and these are partitioned into 'non-atomic' and 'atomic' types. Atomic operations must only be performed on 'atomic' locations.

Candidate executions provide the input to the C11 memory model. As exemplified by Fig. 1b, each execution is a graph whose vertices (called *events*) represent run-time memory accesses or fences, and whose edges represent various relations between these events.

Event labels:

$$\mathrm{W}_{\mathrm{na}}(l, v) \;\mid\; \mathrm{W}_o(l, v) \;\mid\; \mathrm{R}_{\mathrm{na}}(l, v) \;\mid\; \mathrm{R}_o(l, v) \;\mid\; \mathrm{RMW}_o(l, v_{\mathrm{old}}, v_{\mathrm{new}}) \;\mid\; \mathrm{F}_o$$

where l ranges over memory locations, o over memory orders, and v over a set $\mathbb{V}$ of values assumed to contain at least 0 and 1.

Sets of events:

- R, W, F: read or RMW events, write or RMW events, fence events
- I: initialisation events (one non-atomic write to zero per location)
- naL: events accessing a non-atomic location
- na: events representing a non-atomic operation
- $o.rlx$, $o.acq$, $o.rel$, $o.ar$, $o.sc$: events parameterised by the respective memory order

Relations over events:

- thd: an equivalence relation over all (non-initialisation) events from the same thread
- loc: an equivalence relation over all (non-fence) events that access the same location
- sb (sequenced before): a partial order, contained in thd, depicting the order of instructions in the source code
- rf (reads from): contained in $W \times R$, relating writes to reads when the locations and values match, each read reads from exactly one write
- mo (modification order): a total order on writes to the same atomic location
- S: a total order on events in $o.sc$

Fig. 4. C11 events and relations

Events are labelled according to the grammar given at the top of Fig. 4. Write (W), read (R), and read-modify-write (RMW) events (which are both reads and writes) are parameterised by a location and the value(s) read and/or written. All events are parameterised by a memory order o or, optionally, in the case of reads and writes, marked as 'non-atomic' (na). Write events cannot use memory order $o.acq$ or $o.ar$ and read events cannot use $o.rel$ or $o.ar$. Figure 4 lists several predefined subsets of events.

Relations Figure 4 also lists the relations that make up a candidate execution. Of these, some (thd, loc and sb) are determined from the program's syntactic structure by a thread-local semantics, while others (rf, mo, and S) can be chosen in any way that meets the conditions given in Fig. 4. (The full C11 memory model includes further dependency relations, but in the absence of the 'consume' order, we need not include them.) We suppose that (concurrent) C programs take the form $P = \|_{t=0}^{N-1} p_t$, where $N \geq 1$ and each p_t is a sequential C program.

Consistent and faulty executions As mentioned in the introduction, the memory model considers a subset of the candidate executions to be *consistent*, and a subset of the consistent executions to be *faulty*. These subsets are identified by a pair of predicates over executions: $consistent$ and $faulty$. If any candidate execution is both $consistent$ and $faulty$, the behaviour of the entire program is undefined; if not, the behaviour of the program is given by its consistent executions.

Example 1, continued. The candidate execution depicted in Fig. 1b, in which the right thread does not see the updated value of x, fails the $consistent$ predicate because the

events a, b, c, and d are consecutively ordered by the *happens before* relation (as induced by the order in the source code and the release–acquire synchronisation between b and c) but the consistency axiom called *coherence* requires that if a read (such as d) obtains its value from a write (such as e), then any mo-later write (such as a) must not happen before that read.

2.1 The C11 memory model in `.cat`

In our work, we define the memory model's $consistent$ and $faulty$ predicates using the `.cat` format [3]. This involves defining several axioms, each stating that a given relation is irreflexive ($\mathbf{irr}$), acyclic ($\mathbf{acyclic}$) or empty ($\mathbf{empty}$). The $consistent$ predicate is satisfied when *all* of the consistency axioms hold; the $faulty$ predicate is satisfied when *any* of the faulty axioms hold. The relations that our axioms constrain include the relations that appear in candidate executions and any relation derived therefrom using the operators listed below.

(The following notation is a more readable representation of the ASCII-limited language that is used in the real `.cat` files.) If r is a relation, then r^{-1} is its inverse, $r^?$ its reflexive closure, and r^+ its transitive closure. If s is a set, then s^2 abbreviates $s \times s$, and $[s] = \{(e, e) \mid e \in s\}$ is the identity relation (id) restricted to s. We also use complement ($\neg$), the universal relation (unv), and relational composition (;), 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. These operators can usefully be 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-node and then an r_2-edge.

We defer the definitions of the full set of C11 axioms to an appendix (§B) because they will shortly appear in the OpenCL model in §4. Instead, we shall focus in this paper (§3) on a subset of the C11 model – the axioms that govern SC operations, and how they can be simplified.

2.2 Extending the herd simulator

The version of herd described by Alglave et al. [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 candidate 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 the address-of operator, compound types, and function calls. We include `if` and `while` blocks, pointer deferencing, simple expressions, and built-in atomic functions such as `atomic_thread_fence` (C11) and `atomic_work_item_fence` (OpenCL). As an example, we model C11's

`atomic_compare_exchange_strong_explicit`(`obj`, `exp`, `des`, `succ`, `fail`)

instruction (where `obj` is an atomic location, `exp` points to the value that `obj` is expected to contain, `des` is the desired new value, `succ` is the memory order to use for

the RMW operation in the case of success, and `fail` is the memory order to use when loading from `obj` in the case of failure) as having the following candidate executions:

$$\left\{ \begin{array}{c} \mathrm{R}_{\mathrm{na}}(\mathtt{exp}, v_{\mathtt{exp}}) \\ \downarrow sb \\ \mathrm{R}_{\mathtt{fail}}(\mathtt{obj}, v_{\mathtt{obj}}) \\ \downarrow sb \\ \mathrm{W}_{\mathrm{na}}(\mathtt{exp}, v_{\mathtt{obj}}) \end{array} \middle| \begin{array}{l} v_{\mathtt{exp}} \in \mathbb{V} \\ v_{\mathtt{obj}} \in \mathbb{V} \\ v_{\mathtt{exp}} \neq v_{\mathtt{obj}} \end{array} \right\} \cup \left\{ \begin{array}{c} \mathrm{R}_{\mathrm{na}}(\mathtt{exp}, v_{\mathtt{exp}}) \\ \downarrow sb \\ \mathrm{RMW}_{\mathrm{succ}}(\mathtt{obj}, v_{\mathtt{exp}}, \mathtt{des}) \end{array} \middle| v_{\mathtt{exp}} \in \mathbb{V} \right\}$$

and we omit the '$v_{\mathtt{exp}} \neq v_{\mathtt{obj}}$' condition to model the `weak` version, which may fail spuriously.[3] These instructions are rather subtle to model because the memory order that must be used to access the `obj` location depends on the value `obj` contains, which of course can only be determined having performed the load. To our knowledge, previous work on C11 memory model simulation has either modelled these instructions incorrectly [7, 17] or not at all [3, 21].

3 Simplifying the rules for SC operations

In this section we describe how the rules for SC operations can be improved in two fairly orthogonal ways: first by replacing a total order over SC operations with a partial order (§3.1), and second by slightly strengthening the model in such a way that enables significant simplifications to be made (§3.2).

3.1 Partial-order SC axioms

The C11 specification includes a total order, S, over all SC events in each candidate execution, and the behaviour of the SC atomics is defined in terms of this order. A conjunct of the *consistent* predicate captures these governing rules in the formal model. We observe that there is an equivalent model that does away with the total S relation, instead providing an axiom that implies the existence of a suitable total order, without actually constructing it. In this subsection, we describe this new axiom, its relationship both to the standard and the Lem model of Batty et al. [7] and we describe a HOL4 proof of the equivalence.

The overall scheme of the new partial model is to take a candidate execution that is consistent in all respects except the consistency of its SC events, and to represent the axioms of the total model as constraints on the S relation. If the constraints form a partial order, i.e. they are acyclic, then we can extend this to a total order, completing the candidate execution with a concrete S that would satisfy the SC axioms in the total model.

To this end, we build up a list of relations that capture the constraints implied by the total model, unifying rules that are currently weakly linked in the C11, C++11 and C++14 standards. These rules have been a source of problems: important rules were missing from C11 and C++11 and were added in C++14 [20], and C11 imperfectly transcribed them from C++11. As a consequence, we discuss our model in relation to the C++11 standard text, including rules that were added later in C++14.

[3] [12, §7.17.7.4]

$$Fsb := [F] \,;\, sb \qquad sbF := sb \,;\, [F] \qquad rb := (rf^{-1} \,;\, mo) \setminus id$$

(a) Auxiliary definitions

Constraint	Lem definition (see [4])	Reference to standard
$X_1 := hb$	`sc_accesses_consistent_sc`	[12, §29.3:3]
$X_2 := Fsb^? \,;\, mo \,;\, sbF^?$	`sc_accesses_consistent_sc` and `sc_fenced_sc_fences_heeded`	[12, §29.3:3,7] and [20, §29.3:7]
$X_3 := rf^{-1} \,;\, [o.sc] \,;\, mo$	`sc_accesses_sc_reads_restricted`	[12, §29.3:3]
$X_4 := rf^{-1} \,;\, (hb \cap loc) \,;\, [W]$	`sc_accesses_sc_reads_restricted`	[12, §29.3:3]
$X_5 := Fsb \,;\, rb$	`sc_fenced_sc_fences_heeded`	[12, §29.3:4]
$X_6 := rb \,;\, sbF$	`sc_fenced_sc_fences_heeded`	[12, §29.3:5]
$X_7 := Fsb \,;\, rb \,;\, sbF$	`sc_fenced_sc_fences_heeded`	[12, §29.3:6]

(b) Constraints on S (for the unsimplified axiom)

$S_p := X_1 \cup X_2 \cup X_3 \cup X_4 \cup X_5 \cup X_6 \cup X_7$
$\mathbf{acyclic}(o.sc^2 \cap S_p)$ (Sc-orig)

(c) Unsimplified axiom

$S_p := Fsb^? \,;\, (rb \cup mo \cup hb) \,;\, sbF^?$
$\mathbf{acyclic}(o.sc^2 \cap S_p)$ (Sc)

(d) Simplified axiom (see §3.2)

Fig. 5. C11 axiom for partial S order, and a simplified version

Figure 5b lists the constraints. It provides references to their appearance in the standards, the name of the predicate that models each rule in the formalisation of Batty et al., and it defines a relation that captures the implied constraint on S in our partial model. The axiom (Sc-orig) in Fig. 5c then requires the union of these constraints, when restricted to SC operations ($o.sc$), to be acyclic.

Constraint X_1 follows from a requirement that happens-before be consistent with S. X_2 collects together a similar requirement – that modification order be consistent with S – with three other rules that describe the interaction of SC fences, SC atomics and modification order. We observe that C11 prevents S contradicting mo directly, and it prevents S contradicting mo with a fence both before and after, but conspicuously missing are similar axioms to cover the cases when there is only *one* fence, before or after. In fact, these missing axioms are included in the C++14 specification, and we include them in the model presented here.

The rest of the constraints, X_3 through X_7, capture the interaction of reads-before (rb) with SC atomics and SC fences. (Borrowing a notation from Vafeiadis et al. [21], a read event 'reads-before' any write that is not the same event as the read and is later in modification order than the write from which the read obtained its value.) There are five cases, rather than the one case for modification order, because there is a peculiar weakness in the definition of the C11 memory model: a reads-before edge between two SC atomic accesses does not constrain the S order, but it does in the presence of SC fences. This suggests a strengthening of the model, which we explore in the next subsection.

3.2 Stronger and simpler SC

In this section, we propose a simpler axiom for SC operations (axiom Sc in Fig. 5d). Our proposal requires a slight strengthening of the specification. We include in §A a suggested change to the wording of the standard that would accommodate our proposal.

First, we observe that a simplification can almost be made regarding the consistency of S with respect to rb, to match its consistency with mo (constraint X_2). We already have axioms that prevent S contradicting an rb edge that is preceded and/or followed by a fence (X_5, X_6, X_7), but we lack an axiom that prevents S contradicting rb directly. X_3 almost provides this, except that it only rules out cycles in which all writes are sequentially-consistent. We propose to generalise this axiom to remove the $[o.sc]$ restriction, leaving simply $X_3 := rb$. Our proposal allows X_3, X_5, X_6 and X_7 to be replaced with:

$$X_{rb} := Fsb^? \,;\, rb \,;\, sbF^?,$$

and moreover, since X_4 is now subsumed by X_3 (as a result of mo being a total order per location), X_4 can be safely removed.

Second, we observe that the same transformation can be applied to hb. Because sb is included in hb, the original definition of X_2 has the same force as:

$$X_{hb} := Fsb^? \,;\, hb \,;\, sbF^?.$$

Thus, our set of eight constraints on S can be reduced to X_2, X_{rb}, and X_{hb}, and hence, with a little further symbolic manipulation, to the Sc axiom given in Fig. 5d.

Even stronger and simpler SC? One might hope that placing an SC fence between every pair of accesses in the program would restore SC behaviour, but this is not the case [20, 21]. Our partial-order representation of the model makes evident a further strengthening that would provide this property: make an arbitrary chain of mo, rf and rb between accesses associated with two SC fences constrain S. Unfortunately, this is too strong: it would require a more expensive implementation (with additional hardware synchronisation) on the ARM architecture.

4 Formalising the OpenCL 2.0 memory model

We now describe our formalisation of the OpenCL memory model, and the precise sense in which it extends our formalisation of the C11 memory model.

OpenCL programs are executed by a *host* (typically a CPU) in partnership with one or more *devices* (e.g. GPUs). In this paper we consider only the device code. Each thread (or: *work-item*) has its own *private memory*, each *work-group* of threads has its own *local memory*, and the *global memory* is accessible to all threads in all work-groups, on all devices. Accordingly, we shall take a (concurrent) OpenCL program to be of the form

$$P = \bigparallel_{d \in D} \bigparallel_{w \in W} \bigparallel_{t \in T} p_{d,w,t}$$

where D/W/T are sets of device/work-group/thread identifiers, and each $p_{d,w,t}$ is a sequential OpenCL program. That is, $p \parallel\!\!\!\parallel p'$ indicates threads to be executed on different

devices, $p \mid\mid\mid p'$ indicates threads to be executed in different work-groups in the same device, and $p \parallel p'$ indicates two threads in the same work-group.

In practice, all of the threads executing on the same device are defined by a single piece of sequential code called a *kernel*, that is parametric in w (obtained from `get_group_id`) and t (`get_local_id`). 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.

Memory scopes Atomics in OpenCL can be parameterised not only by a memory order, but also by one of the four levels of the execution hierarchy, in order to specify how widely visible the effects of the instruction should be. This parameter is called a *memory scope*, and the four options are na (non-atomic, a.k.a. work-item scope), $s.wg$ (work-group scope), $s.dv$ (device scope) and $s.all$ (all devices scope).

Memory locations are either 'non-atomic' or 'atomic', as in C11, but are also declared as residing in the global or local memory regions. Atomic operations can be performed on atomic locations in global or local memory. Fences can be performed on either the global or the local memory, or both simultaneously.

Candidate executions take broadly the same form as in C11, but are extended (as described in Fig. 6a) with a memory scope per event, fences parameterised by one or more regions, and two additional relations to account for the execution hierarchy. Note that we no longer require separate event labels for non-atomic reads/writes: these can simply be modelled as atomic reads/writes with na scope.

4.1 Consistent executions and faulty executions

The $consistent$ and $faulty$ predicates for the OpenCL memory model are defined in Fig. 6 and discussed below. We pay particular attention to each of the departures from C11, which are highlighted in the figure.

Whenever possible, we justify our formal definitions by reference to the OpenCL specification [13]. Unlike the C11 standard on which it is based, it does not use paragraph numbers. Hence, when referring to the OpenCL specification, we use the notation n/m to denote line m on page n.

Synchronisation (lines 1–8) Different threads synchronise via release and acquire operations. Let acq (resp rel) denote the sets of events that are annotated with $o.acq$ (resp. $o.rel$) or stronger and are hence able to acquire (resp. release).[4] The rs relation links a write w_1 to each write w_2 in its *release sequence*.[5] (This means that w_2 must not be mo-before w_1, and it must either be in the same thread as w_1 or be an RMW, as must every event mo-between w_1 and w_2.) Only events that have *inclusive* scopes ($incl$) can synchronise: either the events have na scope and are in the same thread, or they have $s.wg$ scope and are in the same work-group, or they have $s.dv$ scope and are in the same device, or they have $s.all$ scope. We define sw as the events (e_1, e_2) that synchronise with each other. The events must be in different threads, and e_2 must acquire from e_1's release sequence.[6] Note that if the acquire (resp. release) is a fence, the synchronisation happens via an atomic write sequenced before (resp. after) the fence.[7]

[4] [13, 47/27–31] [5] [13, 45/19–23] [6] [13, 47/34–48/4] [7] [13, 50/14–51/10]

Event labels (extending the grammar of Fig. 4):

$$\mathrm{W}_{o,s}(l,v) \quad | \quad \mathrm{R}_{o,s}(l,v) \quad | \quad \mathrm{RMW}_{o,s}(l,v_{\mathrm{old}},v_{\mathrm{new}}) \quad | \quad \mathrm{F}_{o,s}(r)$$

where s is a memory scope, and $r \in \{\mathrm{global}, \mathrm{local}, \mathrm{global\text{-}and\text{-}local}\}$.

Sets of events (plus those from Fig. 4):
- G: global events; i.e., either a fence on global memory or an access to a global location
- L: local events; i.e., either a fence on local memory or an access to a local location
- $s.wg$, $s.dv$, $s.all$: events parameterised by the respective memory scope

Relations over events (plus those from Fig. 4):
- wg: an equivalence relation over all events from the same work-group
- dv an equivalence relation over all events from the same device

(a) OpenCL events and relations

Synchronisation

1. $acq := (o.acq \cup o.ar \cup o.sc) \cap (R \cup F)$
2. $rel := (o.rel \cup o.ar \cup o.sc) \cap (W \cup F)$
3. $Fsb := [F] \,;\, sb$
4. $sbF := sb \,;\, [F]$
5. $rs' := thd \cup (unv \,;\, [R \cap W])$
6. $rs := mo^{?} \cap rs' \cap \neg((mo \cap \neg rs') \,;\, mo)$
7. $incl := thd \cap na^2 \cup wg \cap s.wg^2 \cup dv \cap s.dv^2 \cup s.all^2$
8. $sw(r) := ([r \cap rel] \,;\, Fsb^{?} \,;\, [W \setminus na] \,;\, rs \,;\, [r] \,;\, rf \,;\, [R \setminus na] \,;\, sbF^{?} \,;\, [acq \cap r]) \cap incl \cap \neg thd$
9. $scf := o.sc^2 \cup (G \cap L \cap F)^2$
10. $gsw := sw(G) \cup scf \cap sw(L)$
11. $lsw := sw(L) \cup scf \cap sw(G)$

Happens-before and coherence

12. $ghb := (G^2 \cap (sb \cup (I \times \neg I)) \cup gsw)^{+}$
13. $lhb := (L^2 \cap (sb \cup (I \times \neg I)) \cup lsw)^{+}$
14. $\mathbf{irr}(ghb)$ (O-Hb-G)
15. $\mathbf{irr}(lhb)$ (O-Hb-L)
16. $coh(hb) := (rf^{-1})^{?} \,;\, mo \,;\, rf^{?} \,;\, hb$
17. $\mathbf{irr}(coh(ghb))$ (O-Coh-G)
18. $\mathbf{irr}(coh(lhb))$ (O-Coh-L)

Consistency of reads

19. $\mathbf{irr}(rf \,;\, (ghb \cup lhb))$ (O-Rf)
20. $vis(hb) := (W \times R) \cap hb \cap loc \cap \neg((hb \cap loc) \,;\, [W] \,;\, hb)$
21. $\mathbf{irr}(rf \,;\, [G \cap naL] \,;\, \neg vis(ghb)^{-1})$ (O-Narf-G)
22. $\mathbf{irr}(rf \,;\, [L \cap naL] \,;\, \neg vis(lhb)^{-1})$ (O-Narf-L)
23. $\mathbf{irr}(rf)$ (O-Rmw1)
24. $\mathbf{irr}(mo \,;\, mo \,;\, rf^{-1})$ (O-Rmw2)
25. $\mathbf{irr}(mo \,;\, rf)$ (O-Rmw3)

Sequential consistency

26. $rb := (rf^{-1} \,;\, mo) \setminus id$
27. $S_p := Fsb^{?} \,;\, (rb \cup mo \cup hb) \,;\, sbF^{?}$
28. $S_p := \neg(unv \,;\, [o.sc \cap \neg(s.all \cup s.dv)] \,;\, unv) \cap S_p$
29. $\mathbf{acyclic}(o.sc^2 \cap S_p)$ (O-Sc)

(b) The OpenCL consistency predicate. $consistent :=$ (O-Hb-G) $\wedge \cdots \wedge$ (O-Sc)

30. $conflict := ((W \times W) \cup (W \times R) \cup (R \times W)) \cap loc$
31. $dr := conflict \cap \neg(ghb \cup lhb) \cap \neg(ghb \cup lhb)^{-1} \cap \neg incl$
32. $\mathbf{empty}(dr)$ (O-Dr)

(c) The OpenCL faulty predicate. $faulty :=$ (O-Dr)

Fig. 6. The OpenCL memory model in `.cat`, with departures from C11 highlighted.

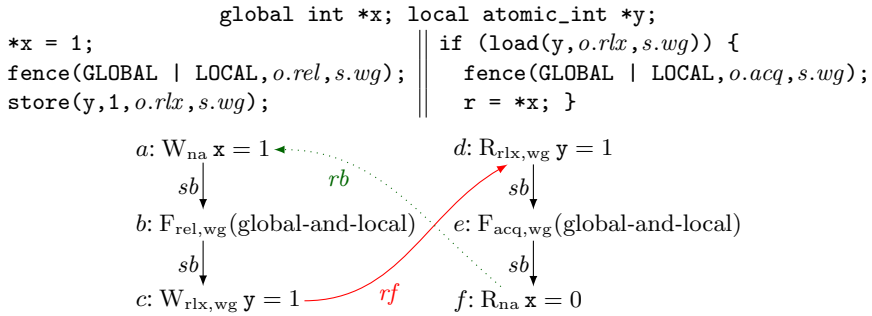


Fig. 7. An MP kernel (featuring global data, a local flag, and additional global-and-local fences) and an (inconsistent) candidate execution. Initialisation events and some relations omitted.

Example 3. In Fig. 2, the release-store on y synchronising with the acquire-load prevents a race on x. The threads are in the same device (as denoted by $|||$), so scope $s.dv$ is sufficient. If the figure were modified to put the threads in different devices (as denoted by $||||$), the synchronisation could not occur unless both scopes were upgraded to $s.all$.

Global and local synchronisation (lines 9–11) The synchronisation relation (sw) is parameterised by a region r (global or local). The global synchronises-with relation (gsw) includes events that synchronise on global memory or – in the case that both events are have memory order $o.sc$ or both are global-and-local fences – synchronise on *local* memory. Local synchronises-with (lsw) is defined symmetrically.

Happens-before (lines 12–18) contains sequenced-before and synchronisation edges.[8] Note that initialisation events happen before non-initialisation events. Happens-before is partitioned into global and local versions: global happens-before (ghb) only takes global synchronisation edges, and sequenced-before edges between events on global memory (as ensured by intersecting with G^2); local happens-before (lhb) is analogous. Both are required to be irreflexive (O-Hb-G and O-Hb-L).[9] The principle of *coherence* essentially states that happens-before must not contradict the modification order (mo): if the write w_1 is mo-before the write w_2, then w_2 (and its reads) must not happen after w_1 (and its reads).[10] OpenCL requires coherence for both global and local happens before separately (O-Coh-G and O-Coh-L).

Example 4. One repercussion of this definition of happens-before is that if y were in `local` rather than `global` memory in Fig. 2, then even successful release–acquire synchronisation could not rule out x being read at 0, because neither happens-before relation would include (a, b) or (c, d). As such: a flag in local memory cannot be used in this manner to protect data in global memory, and vice versa.

Example 5. To address the issue raised in Example 4, OpenCL provides global-and-local fences, which can be inserted as shown in Fig. 7 to provide synchronisation across both global and local memory. Here, reading x at 0 is ruled out because $(a, f) \in ghb$,

[8] [13, 45/38–46/3] [9] [13, 46/8–9] [10] [13, 45/14–15] and [13, 47/7–20]

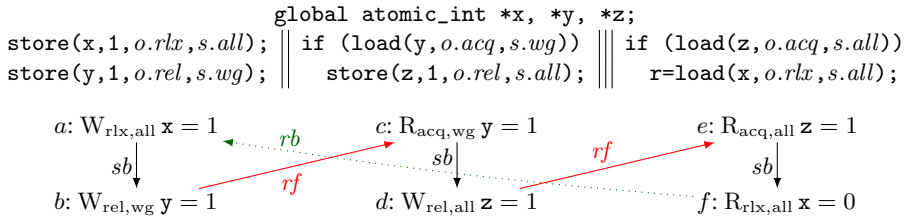


Fig. 8. An ISA2 kernel and an (inconsistent) candidate execution. Initialisation events and some relations omitted.

which in turn is justified by (a, b) and (e, f) being in $sb \cap G^2$, and (b, e) being in $sw(L)$ and hence in gsw. A similar result is obtained, without fences, by making both accesses to y use the $o.sc$ memory order.

Consistency of reads (lines 19–25) A read must not obtain its value from a write that happens after it (O-Rf).[11] Moreover, a read of a non-atomic location must obtain its value from a write that is *visible*: the write must happen before the read, with no write to the same location happening in-between (O-Narf-G and O-Narf-L).[12]. An RMW must read from the immediately-preceding write in mo;[13] that is, it must not read from: itself (Rmw1), a read that is too early (Rmw2), or a read that is too late (Rmw3).

Example 6. Figure 8 illustrates the 'ISA2' idiom (message passing via an intermediary). Here, the left thread synchronises with the middle one (using $s.wg$ scope since they are both in the same work-group), which then synchronises with the right thread in another work-group (using $s.all$ scope). This implies that a global-happens-before f – this despite a releasing only at $s.wg$ scope. Coherence (O-Coh-G) then prevents f reading x's initial value. However, if y were in `local` memory, a would *not* global-happen-before f, so x could be read at 0 or 1.

Sequential consistency (lines 26–29) The conditions on the S relation are only required to hold if every SC event has memory scope $s.dv$ or $s.all$,[14] so we redefine the S_p relation to be empty (and thus to satisfy the axiom vacuously) if this is not so.

Example 7. We note that the above restriction on S_p spoils the compositionality of the OpenCL memory model: one cannot exploit the SC order when reasoning about library code, in case an unknown client uses an SC operation with a scope narrower than $s.dv$, thus invalidating the SC guarantee for the entire program. This observation recalls the lack of compositionality in C11 observed by Batty et al. [5], which is caused by the possibility of cycles appearing when library and client executions are combined.

Faulty behaviour (lines 30–32) Two events *conflict* if at least one is a write and they access the same location.[15] There is a *data race* (dr) between a pair of events if they conflict, neither happens before the other, and their scopes are not inclusive.[16] Unless

[11] The specification uses the 'visible sequence of side effects' to phrase this clause [13, 47/1–6], but Batty [4, §5.3] has proved that 'happens after' suffices. [12] [13, 47/17–24] [13] [13, 49/18–20] [14] [13, 48/9–16] [15] [13, 45/11–13] [16] [13, 46/25–31]

the dr relation is empty, the execution is faulty (Dr). The specification additionally requires that 'at least one of [the events] is not atomic' and that they are 'in different [threads]'; that is:

$$dr := \mathit{conflict} \cap \neg(\mathit{ghb} \cup \mathit{lhb}) \cap \neg(\mathit{ghb} \cup \mathit{lhb})^{-1} \cap (\neg \mathit{incl} \cup \neg(na^2)) \cap \neg \mathit{thd}$$

However, we observe that these requirements are already covered as a consequence of inclusive scopes, and so our simpler definition of dr in Fig. 6c suffices.

4.2 Relating the OpenCL and C11 memory models

The specification provides a conservative notion of scope inclusion. An alternative proposed by Gaster et al. is to allow the annotated scopes to differ, as long as both are sufficiently wide [10]. This enables, for instance, an $s.dv$ write to synchronise with an $s.wg$ read in the same work-group. The proposal, which we call 'asymmetric scope inclusion', can be implemented by changing the definition of the incl relation as follows:

$$\begin{aligned} \mathit{incl1} &:= ([na] \,;\, \mathit{thd}) \cup ([s.wg] \,;\, \mathit{wg}) \cup ([s.dv] \,;\, \mathit{dv}) \cup ([s.all] \,;\, \mathit{unv}) \\ \mathit{incl} &:= \mathit{incl1} \cap \mathit{incl1}^{-1}. \end{aligned}$$

The idea here is to define a one-sided version of scope inclusion first, so that (e_1, e_2) is in $\mathit{incl1}$ 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.

With this change in place, the OpenCL memory model becomes a conservative extension of the C11 model (without the 'consume' memory order), in the sense that the latter can be obtained as a special case of the former, in which all threads belong to a single work-group, all memory is global, and memory scopes are restricted to na, for all non-atomic instructions, and $s.all$, for all atomic instructions. The key to this observation is to note that the incl relation in the C11 model (currently defined as $\mathit{thd} \cup (\neg na)^2$) can be equivalently formulated as

$$\begin{aligned} \mathit{incl1} &:= ([na] \,;\, \mathit{thd}) \cup ([s.all] \,;\, \mathit{unv}) \\ \mathit{incl} &:= \mathit{incl1} \cap \mathit{incl1}^{-1} \end{aligned}$$

which is identical to the new incl relation given above, minus the $s.wg$ and $s.dv$ scopes.

5 Related work

The C11 memory model has been formalised several times. Batty et al. [7] present a comprehensive formalisation using Lem [16]. Vafeiadis et al. [22, 21] and Batty et al. [5] have also formalised 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 [3], and have shown it to be an instance of their generic axiomatic memory model. We use the `.cat` language in our work but our more comprehensive model, with its catch-fire semantics and richer language of events, does not fit into the generic framework.

Criticisms of the C11 model Batty et al. describe a fundamental problem in the structure of the C11, C++11, C++14 and OpenCL 2.0 memory models: the so-called "thin-air" executions [6]. 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 covering the 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 [21]. Our proposal goes further than this, arriving at a substantially simpler model.

The OpenCL 2.0 memory model has recently been formalised by Gaster et al. [10], as an instance of a heterogeneous race-free (HRF) model [11]. (A heterogeneous race is an appropriate generalisation of a data race in the context of a memory model with scopes, where memory accesses can race not only by being non-atomic, but also by using inadequate memory scopes.) Although Gaster et al.'s model captures some details that we omit, such as shared virtual memory, their model is not mechanised, and it lacks fidelity to the specification – for instance, they omit details of the interplay between the local and global happens-before relations, and they do not model SC atomics soundly.

Memory model simulators capable of handling the C11 model include CPPMEM [7], Nitpick [8] and CDSCHECKER [17]. Quick experiments show that herd is significantly faster and can handle larger litmus tests than CPPMEM. For instance, CPPMEM takes 23 seconds to calculate the candidate executions of the classic IRIW litmus test when all accesses are $o.sc$, while herd, thanks to our partial S optimisation (§3) is almost instantaneous. Nitpick and CDSCHECKER are also faster than CPPMEM, but the former does not have a user interface, existing only within the Isabelle proof environment, and the latter is incomplete, in the sense that it does not generate all candidate executions (self-satisfying conditionals, for instance, are omitted). 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.

—

Acknowledgements Luc Maranget kindly advised on our extensions to herd. We thank Jade Alglave and Tyler Sorensen for their support and many helpful discussions about this work, and credit to Tyler the idea of simplifying the definition of the dr relation. Alastair Donaldson suggested the OpenCL memory model as a topic of study; we thank him and Nathan Chong, Benedict Gaster, Jeroen Ketema, Matthew Parkinson, and Peter Sewell for their feedback and encouragement. This work is supported by the EPSRC grants EP/H005633/1, EP/K008528/1, EP/K011499/1, and the EU FP7 project CARP (287767).

References

1. Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: weak behaviours and programming assumptions. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14–18 2015*, 2015. To appear.
2. Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 258–272, Berlin, Heidelberg, 2010. Springer-Verlag.
3. Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7, 2014.
4. Mark Batty. *The C11 and C++11 Concurrency Model*. PhD thesis, University of Cambridge, October 2014.
5. Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 235–248, New York, NY, USA, 2013. ACM.
6. Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon, and Peter Sewell. C concurrency challenges. In *ESOP*, 2015. To appear.
7. Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 55–66. ACM, 2011.
8. Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. Nitpicking C++ concurrency. In Peter Schneider-Kamp and Michael Hanus, editors, *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, pages 113–124. ACM, 2011.
9. Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 68–78. ACM, 2008.
10. Benedict Gaster, Derek Hower, and Lee Howes. HRF-relaxed: Adapting HRF to the complexities of industrial heterogeneous memory models. *ACM Transactions on Architecture and Code Optimization*, 2015. To appear.
11. Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous-race-free memory models. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 427–440. ACM, 2014.
12. ISO/IEC. *Programming languages - C*, 2011.
13. Khronos. *The OpenCL Specification (Version 2.0)*, November 2013. Current version available from: `https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`. A cache of document revision 22, the version referenced in this paper, is included in our online companion material (`http://multicore.doc.ic.ac.uk/opencl/`).
14. L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor,. *Computers, IEEE Transactions on*, 46(7):779–782, 1997.
15. Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 187–196, New York, NY, USA, 2013. ACM.

16. Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In *Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 175–188, 2014.
17. Brian Norris and Brian Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 131–150. ACM, 2013.
18. Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, 2008.
19. Alfred Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, 1941.
20. Stefanus Du Toit, editor. *Programming Languages — C++*. ISO/IEC, September 2014. ISO/IEC CD 14882. A draft of the 2014 standard, this document is available at `http://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf`.
21. Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are unsound in the C11 memory model and what we can do about it. In *The 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, Mumbai, India – January 12–18, 2015*, 2015. To appear.
22. Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages' Applications*, OOPSLA 13, pages 867–884, New York, NY, USA, 2013. ACM.