Remote-Scope Promotion: Clarified, Rectified, and Verified

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

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

Bradford M. Beckmann
Advanced Micro Devices, Inc., USA
brad.beckmann@amd.com

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

Abstract

Modern accelerator programming frameworks, such as OpenCL™, organise threads into work-groups. *Remote-scope promotion* (RSP) is a language extension recently proposed by AMD researchers that is designed to enable applications, for the first time, both to optimise for the common case of intra-work-group communication (using *memory scopes* to provide consistency only within a work-group) and to allow occasional inter-work-group communication (as required, for instance, to support the popular load-balancing idiom of *work stealing*).

We present the first formal, axiomatic memory model of OpenCL extended with RSP. We have extended the HERD memory model simulator with support for OpenCL kernels that exploit RSP, and used it to discover bugs in several litmus tests and a work-stealing queue, that have been used previously in the study of RSP. We have also formalised the proposed GPU implementation of RSP. The formalisation process allowed us to identify bugs in the description of RSP that could result in well-synchronised programs experiencing memory inconsistencies. We present and prove sound a new implementation of RSP that incorporates bug fixes and requires less non-standard hardware than the original implementation.

This work, a collaboration between academia and industry, clearly demonstrates how, when designing hardware support for a new concurrent language feature, the early application of formal tools and techniques can help to prevent errors, such as those we have found, from making it into silicon.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel Architectures; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

Keywords Formal methods, graphics processing unit (GPU), Isabelle, OpenCL, programming language implementation, weak memory models, work stealing

1. Introduction

Remote-scope promotion (RSP) is a new accelerator programming feature that was recently proposed by a team of AMD researchers [19]. In a nutshell, RSP allows two popular accelerator programming paradigms – *memory scopes* and *work stealing* – to be unified for the first time. Simulation of a prototype GPU implementation has demonstrated that applications using RSP perform on average 17% faster than those using only memory scopes, and 6% faster than those that use only work stealing. This encouraging result indicates that the feature has the potential to be included in future GPUs.

In this work we scrutinise the complex and subtle design of RSP and reason rigorously about its correctness using formal techniques. We report on our findings, presenting the technical details of our RSP formalisation, and highlighting the role our formal approach played in identifying bugs in the original design.

This work stems from a collaboration between AMD researchers and an academic team of formal semanticists. Our collaboration was made possible by AMD's decision to publish details of new processor features very early in their design cycle, a departure from the more closed approach typical among processor vendors. We report on bugs that we found in the RSP design, demonstrating the value of applying formal techniques from the academic research community and collaborating *early* during the design of hardware support for a new concurrent programming language construct.

Using a combination of a proof assistant (Isabelle [18]),[1] a cutting-edge memory modelling tool (HERD [3]), and recent advances in modelling the behaviour of heterogeneous programming languages (e.g., [24]), we have translated the original proposal for RSP, which encompasses both high-level programming language extensions and low-level architectural extensions, into rigorous mathematics. We have discovered and identified fixes for several bugs with the previously-described design, improved its clarity both for users and implementers, proposed several modifications that simplify the design and may improve its performance, and proved the soundness of the implementation (after applying our fixes and improvements).

Our work is distinguished from other efforts to formalise the semantics of complex processor architectures [1, 3, 16, 22, 23], by its focus on a *prototype* architecture that is several years away from fabrication. We therefore deliberately focus on an idealised model. This enables rapid development, and leads to what we regard as the key value of our work: identifying fixes and improvements to the RSP design to be made *at very little cost*. In contrast, errors and bugs discovered late in the design cycle or errors that make it into silicon can be extremely expensive to fix or work around.

We report the following research contributions:

1. Formalising the OpenCL+RSP language (§3) We describe precisely how the OpenCL™ framework for heterogeneous parallel programming can be extended to capture RSP; we call the extension OpenCL+RSP.

2. Testing OpenCL+RSP programs (§4) We have extended the HERD litmus test simulator [3] to enable enumeration of the allowable outcomes of small OpenCL+RSP programs. This allows developers to understand how key concurrency idioms at the heart of their algorithms might behave on *any* (current or future) correct implementation of OpenCL+RSP. We applied HERD to a set of 12 AMD OpenCL+RSP litmus tests and an AMD work-stealing queue implementation, revealing issues in four of the litmus tests and a data race in the work-stealing queue implementation. These issues have been confirmed and fixed.

3. Formalising the implementation of OpenCL+RSP (§5) Informed by the published design [19] and collaboration between the authors and other AMD designers, we present a formalisation of the original implementation of OpenCL+RSP. This comprises a mathematical model of a simple GPU device, semantics for a minimal assembly language for this device, and a scheme for compiling OpenCL+RSP to this assembly language.

The formalisation process led to the discovery of two bugs in the initially proposed design of RSP, one that violates the atomicity of atomic read-modify-write (RMW) operations, and one that renders the *message-passing* idiom unusable. This idiom is central in lock-free concurrency and used at the heart of work stealing, the key motivator for OpenCL+RSP. We confirmed the message-passing bug to occur twice in the prototype implementation. A third potential instance of this bug was actually avoided in the prototype implementation by the inclusion of cacheline stalls, which prevent certain problematic interactions; but these stalls were not mentioned in the description of the implementation. These two scenarios respectively illuminate how formalisation can help designers not only to make correct designs, but to understand which details make their designs correct.

4. Improving the implementation (§5.4, §5.5) Our formalisation also identified several significant simplifications that can safely be made to the OpenCL+RSP implementation: by reordering certain low-level instructions, the bugs we found can be fixed without the need for expensive cacheline stalls. Importantly, this avoids the need for non-standard hardware to support cacheline stalling, a prerequisite of the original implementation proposal. Additionally, the avoidance of stalling may improve the efficiency of the implementation.

5. A proof of soundness (§6) Finally, we prove that our improved implementation is sound, in that it does provide the required high-level semantics. More precisely: we show that every low-level execution of a compiled OpenCL+RSP program is contained in the set of executions that are allowed for the program with respect to the high-level semantics. Our soundness result provides a firm basis for designers looking to support RSP in their next-generation GPU architectures.

Online companion material We provide Isabelle scripts containing type-checked definitions for our formalisation of RSP, and an extended write-up of our (non-mechanised) soundness proof, at the following webpage:

`http://multicore.doc.ic.ac.uk/RSP/`

2. Background: Remote-Scope Promotion

2.1 Heterogeneous Programming with OpenCL

The OpenCL programming framework [17] provides a *hierarchical execution model*, geared towards heterogeneous systems made up of CPUs, GPUs, and other accelerators. Each thread[2] is identified according to the *work-group* to which it belongs, and the device on which that work-group is executing. A thread executes on a *processing element* and a work-group on a *compute unit*. A similarly-structured *memory hierarchy* comprises a private memory region per thread, a region shared among threads in a work-group (e.g., an L1 cache), a region shared among work-groups on a device (e.g. an L2 cache), and a global region available to the whole system. High performance can be achieved by restricting data sharing to lower levels of the memory hierarchy where possible, minimising memory latency. Figure 1 illustrates the execution and memory hierarchies.

[1] As discussed further in §5, we have used Isabelle to formalise and type-check our definitions and theorem statements; we have *not* mechanically proved the theorems.

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

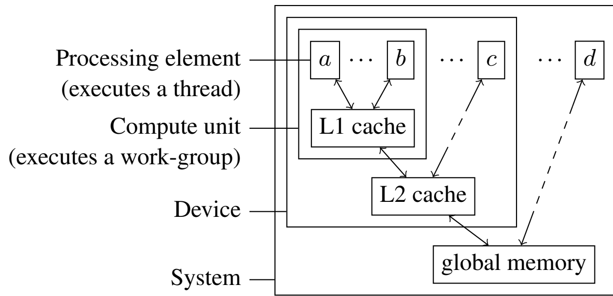


Figure 1. Illustration of the OpenCL execution hierarchy

Atomic operations OpenCL 2.0 provides *atomic operations*, which enable fine-grained lock-free synchronisation both within and between work-groups and devices. The operations provide a range of memory-consistency guarantees according to semantics defined by a detailed C11-based *memory model* [17: §3.3]. Operations with weaker guarantees may offer superior performance, but have more subtle semantics.

Memory scopes Where the OpenCL memory model departs significantly from C11 is in its introduction of *memory scope* constants. The three constants are:

$$s ::= \texttt{WG} \quad \text{current work-group} \mid \texttt{DV} \quad \text{current device} \mid \texttt{ALL} \quad \text{all devices}$$

and, when attached to an atomic operation, govern how far through the execution hierarchy the memory consistency guarantees must be enforced. For instance, if a global memory location `x` is currently being accessed only by threads in the same work-group, such as a and b in Fig. 1, the accesses can be scoped at `WG` so that they need travel no further than the L1 cache that a and b share.

Both participants in a synchronisation operation are required to use a memory scope that is wide enough to encompass the other. This rule would be violated if, for instance, thread a in Fig. 1 writes `x` at `WG` scope, thread c (in a different work-group) reads `x`, and there is no synchronisation in between. As we shall explain further in §2.2, such a situation is deemed by OpenCL to be a *race*: a programmer fault that renders the whole program undefined.

Work stealing is a technique for achieving dynamic load balancing in high-performance computing. In an OpenCL context, work stealing involves each work-group owning a task queue, and idle threads popping tasks from another work-group's queue should their own queue become empty. The ability to steal work is valuable when it is impractical to distribute tasks evenly among work-groups at compile-time – either because of a non-uniform computational cost per task that depends on input data, or because new tasks can be created dynamically at run-time [8].

To scope or to steal? The current design of OpenCL allows programmers to exploit work stealing, or to exploit the scoping mechanism, but not both. To see this, consider an application that exploits scopes by having threads use work-group scope when pushing to or popping from their local task queue. This makes stealing impossible, regardless of the stealer's scope, because synchronisation fails unless *both* of the operations that are synchronising use wide-enough scopes. To make stealing possible, we could arrange that every queue operation uses a wider scope, but then we lose the benefit of using scopes: the common case (accessing one's own queue) would take a performance hit to allow the uncommon case (stealing from another's queue).

A solution: remote scope promotion Previous work proposed an extension to OpenCL's scoping mechanism [19], *remote-scope promotion* (RSP), that *is* compatible with work stealing. It was shown that on a range of benchmarks from the Pannotia suite [9] (including Google's PageRank), RSP combined with stealing leads to an average speedup of 17% over scopes alone, and 6% over stealing alone.

In OpenCL extended with RSP (OpenCL+RSP), each atomic operation has an extra Boolean flag, indicating whether the operation is *remote*. In ordinary scoped synchronisation, both of the synchronising operations must use a wide-enough scope. The essence of RSP is to add another sufficient condition for synchronisation; namely, that just one of the participants has a wide-enough scope and is flagged as remote. In this case, the scope of the other participant is irrelevant; it is silently promoted to match the first participant's scope.

A mapping from OpenCL+RSP to hardware primitives was described previously. Remote operations are compiled to special instructions for flushing or invalidating caches that belong to other work-groups or other devices. It was informally argued that the implementation is correct. The implementation has been realised in a simulator and has been tested using a number of examples.

The need for formality Like all new concurrency-related language constructs, RSP has a subtle semantics that may be hard to implement correctly, and is hard to reason about. (For instance, from the prose description above: what should happen when *both* participants in a pair of synchronising operations are flagged as remote?) Due to the inconclusive nature of testing, and the fundamental problems associated with testing concurrent systems, we turn to formal methods for a rigorous treatment of RSP.

2.2 The OpenCL Memory Model

The OpenCL 2.0 memory model, which builds on the C11 memory model, is the part of the OpenCL language specification that covers reading and writing shared memory locations. It defines which values are allowed to be read at a given program point, and whether two memory accesses have a *data race*. It is principally concerned with the collection of *atomic functions*, which can expose to programmers the various *weak memory* behaviours of the underlying hardware.

The C11 memory model was first formalised by Batty et al. [6]; this formalisation was then extended to the OpenCL case by Wickerson et al. [24]. In §3, we extend the memory model further to formalise OpenCL+RSP.

Axiomatic memory models All of these memory models are defined *axiomatically*. To define the set of a program's allowed executions in this style, one first generates a superset thereof, called the set of *pre-executions*, which comprises those executions that could be obtained with the use of a completely non-deterministic memory that returns an arbitrary value for each load. An 'execution', in this context, comprises a set of run-time memory events (such as $\mathrm{R}_{\mathtt{DV}}(\mathtt{x}, 42)$, which indicates the value 42 being read from `x` using device scope), and several relations between them (such as the *program order* of the corresponding instructions). One then whittles this down, using a set of axioms, to the set of *consistent* executions. A pre-execution is consistent if it can be extended to a *candidate execution* that satisfies all of the axioms of the memory model. A candidate execution additionally contains a *reads from* relation (representing data flow from write events to read events) and a *modification order* among the writes to each location. These two relations together constitute the *execution witness*. If any of a program's candidate executions is deemed to have a data race, the behaviour of the program is *undefined*, which means that it may behave arbitrarily (and the behaviours of candidate executions that do not have data races become irrelevant).

Syntax of OpenCL programs We restrict our attention to a small OpenCL-like language that includes non-atomic loads and stores ($\mathtt{load}_{\mathtt{na}}$ and $\mathtt{store}_{\mathtt{na}}$), scoped atomic acquire-loads and release-stores ($\mathtt{load}_s$ and $\mathtt{store}_s$), and a scoped 'atomic increment' operation ($\mathtt{fetch_inc}_s$) that demonstrates an acquire+release RMW. Full OpenCL includes other varieties of atomic memory access, such as relaxed and sequentially-consistent.

In real OpenCL, all threads execute the same *kernel program*, but can obtain differing control or data flows by querying their own thread identifiers. In our simplified setting, we suppose that each thread is programmed independently. Reflecting the execution hierarchy in OpenCL, we formalise an OpenCL program P as a list of lists of lists of sequential programs:

$$\begin{array}{rcl} P & ::= & P_{\mathrm{dv}} \parallel\!\parallel \ldots \parallel\!\parallel P_{\mathrm{dv}} \\ P_{\mathrm{dv}} & ::= & P_{\mathrm{wg}} \mid\!\mid\!\mid \ldots \mid\!\mid\!\mid P_{\mathrm{wg}} \\ P_{\mathrm{wg}} & ::= & p \parallel \ldots \parallel p \end{array}$$

where p is a piece of sequential code, $\parallel\!\parallel$ separates code executed by different devices, $\mid\!\mid\!\mid$ separates code executed by different work-groups in the same device, and $\parallel$ separates code executed by different threads in the same work-group.

Example 1. The following program comprises two threads in two different work-groups on the same device.

$$\mathtt{fetch_inc}_{\mathtt{DV}}(\mathtt{x}) \mid\!\mid\!\mid \mathtt{store}_{\mathtt{DV}}(\mathtt{x}, 2)$$

One thread increments `x` and the other sets `x` to 2. The use of `DV`-scope for both operations ensures that these conflicting accesses do not race. □

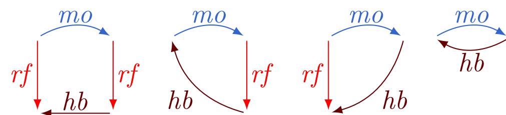
2.3 Details of the Memory Model

The details of the OpenCL memory model are summarised in Fig. 2. We give the language from which event labels are drawn – this ranges over read (R), write (W), and read-modify-write (RMW) events (which represent, for instance, an atomic increment or a successful compare-and-swap). We provide identifiers for particular subsets of events in any given candidate execution, and list the basic and derived relations between events. We finally give the axioms of the memory model, following the `.cat` format of Alglave et al. [3]. Five axioms (the consistency axioms) characterise consistent executions, and one further axiom (the non-faultiness axiom) characterises the absence of data races. We shall explain these axioms further, after introducing the following notational conventions.

Notation. We write r^+ for the transitive closure of a relation r, r^{-1} for its inverse, and we abbreviate $r \cup id$ as $r^?$, where id is the identity relation. We write $\neg$ for complement, $\setminus$ for set difference, unv for the universal relation, and $[s]$ to abbreviate $\{(x, x) \mid x \in s\}$. We define relational composition (;) such that $(x, z) \in r_1 \,;\, r_2$ if $(x, y) \in r_1$ and $(y, z) \in r_2$ for some y. (This notation is convenient for describing shapes in execution graphs; for instance, the relation $[s_1];r_1;[s_2];r_2;[s_3]$ relates events in s_1 to those in e_3 that can be reached by taking an r_1-edge to an event in s_2 and then taking an r_2-edge.) □

The rs' and rs relations define the *release sequence*, which is inherited without modification from C11 and can be safely ignored by the unfamiliar reader. The *incl1* relation connects event e_1 to event e_2 whenever e_1's scope is no narrower than the distance between the events in the execution hierarchy at run-time. When this relation also holds in the opposite direction ($incl^{-1}$), then the events are deemed to have inclusive scopes, as captured by the *incl* relation. Inter-thread synchronisation (sw, 'synchronises-with') relates an atomic write to an atomic read in another thread that reads from it, providing the two events have inclusive scopes. Happens-before (hb) edges are induced both by the program order and by synchronisation. They must not form cycles (Hb); that is, the shape ⟲hb is forbidden.

The principle of *coherence* (Coh) governs the relationship between hb and mo: if the write w_1 is mo-before the write w_2, then w_2 (and any events that read from w_2) must not happen before w_1 (nor before any events that read from e_1). Coherence forbids the following shapes.



The reads-from axiom (Rf) forbids reads to observe writes that happened after them (hb↻rf). The non-atomic reads-

Location types: Each location is atomic or non-atomic

Event labels: For locations x, scopes s, and values v:

- $\mathrm{W}_{\mathtt{na}}(x, v)$ / $\mathrm{R}_{\mathtt{na}}(x, v)$: non-atomic write/read
- $\mathrm{W}_s(x, v)$ / $\mathrm{R}_s(x, v)$: atomic write/read
- $\mathrm{RMW}_s(x, v, v')$: atomic read-modify-write

Predefined subsets of events:

- R: the set of read and RMW events
- W: the set of write and RMW events
- I: the set of initialisation events (each a non-atomic write of 0, one per location)
- *nal*: events that access a non-atomic location
- na: non-atomic events
- WG / DV / ALL: the set of events that are parameterised by the respective memory scope

Primitive relations:

- *thd*/*wg*/*dv*: an equivalence relation over all (non-initialisation) events, relating events from the same thread/work-group/device
- *loc*: an equivalence relation over all events, relating events that access the same location
- *sb* (sequenced before): a strict partial order specifying the order of each thread's instructions, and also linking initialisation events to non-initialisation events
- *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 strict partial order that relates all and only writes to the same atomic location

Derived relations:

- $rs' \stackrel{\text{def}}{=} thd \cup (unv \,;\, [R \cap W])$
- $rs \stackrel{\text{def}}{=} mo \cap rs' \setminus ((mo \setminus rs') \,;\, mo)$
- $incl1 \stackrel{\text{def}}{=} ([\mathtt{WG}] \,;\, wg) \cup ([\mathtt{DV}] \,;\, dv) \cup ([\mathtt{ALL}] \,;\, unv)$
- $incl \stackrel{\text{def}}{=} incl1 \cap incl1^{-1}$
- $sw \stackrel{\text{def}}{=} ([W \setminus \mathtt{na}] \,;\, rs^? \,;\, rf \,;\, [R \setminus \mathtt{na}]) \cap incl \setminus thd$
- $hb \stackrel{\text{def}}{=} (sb \cup sw)^+$
- $hbl \stackrel{\text{def}}{=} hb \cap loc$
- $vis \stackrel{\text{def}}{=} hb \setminus (hb \,;\, [W] \,;\, hbl)$
- $conflict \stackrel{\text{def}}{=} ((W \times W) \cup (W \times R) \cup (R \times W)) \cap loc$
- $dr \stackrel{\text{def}}{=} conflict \setminus hb \setminus hb^{-1} \setminus incl$

Consistency axioms:

$\mathbf{irreflexive}(hb)$ (Hb)

$\mathbf{irreflexive}((rf^{-1})^? \,;\, mo \,;\, rf^? \,;\, hb)$ (Coh)

$\mathbf{irreflexive}(rf \,;\, hb)$ (Rf)

$\mathbf{empty}((rf \,;\, [nal]) \setminus vis)$ (Narf)

$\mathbf{irreflexive}(rf \cup (mo \,;\, mo \,;\, rf^{-1}) \cup (mo \,;\, rf))$ (Rmw)

Non-faultiness axiom:

$\mathbf{empty}(dr)$ (Dr)

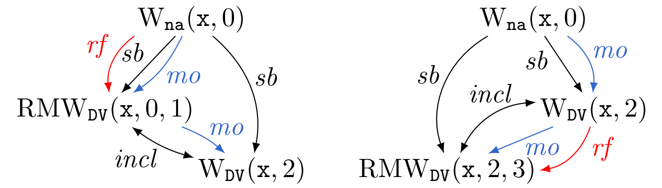
Figure 2. The OpenCL memory model (simplified)

from axiom (Narf) requires non-atomic reads to observe an immediate predecessor in hb, called a *visible write*: i.e. we must have hb ⟲ rf but no hb-intervening write to the same location (hb ; W ; hbl vs. rf).

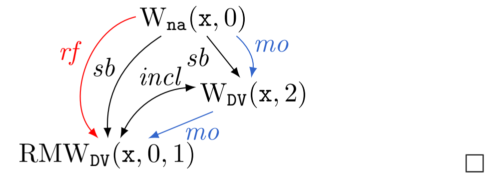
The principle of *RMW atomicity* (Rmw) dictates that each RMW event must observe the mo-latest write to that location; that is, it must not observe itself (⟲ rf), it must not observe a write that is too late (mo ⟲ rf), and it must not observe a write that is too early (mo ; mo vs. rf).

Finally, an execution has a data race (dr) if two conflicting events are unrelated by happens-before and do not have inclusive scopes. The non-faultiness axiom (Dr) detects data races.

Example 2. Here are two executions of the program in Example 1, both of which satisfy all of the consistency axioms and the non-faultiness axiom. The initial event is drawn above the events of the two parallel threads. Reflexive and transitive edges are elided. The left-hand execution gives rise to the final state $\mathtt{x} = 2$, while the right-hand one finishes with $\mathtt{x} = 3$.



Other final values for $\mathtt{x}$ are not allowed. For instance, the following execution, which would result in $\mathtt{x} = 1$, falls foul of the Rmw axiom: it constitutes a violation of RMW atomicity.



□

Example 3. If the program in Example 1 were changed so that the increment had work-group scope:

$$\mathtt{fetch_inc}_{\mathtt{WG}}(\mathtt{x}) \;|||\; \mathtt{store}_{\mathtt{DV}}(\mathtt{x}, 2)$$

then the scope-inclusion ($incl$) edges seen in Example 2 would all be replaced with data race (dr) edges. □

Simplifications and other discrepancies with the standard We do not consider the distinction between global memory (shared among all threads) and local memory (shared among threads in a work-group), and instead treat all memory as global; local memory is not interesting in the context of memory scopes, since the only allowable scope with which local memory can be accessed atomically is WG. Fences, barriers, relaxed atomics, and sequentially-consistent atomics

were not discussed in previous work on RSP [19] and are largely orthogonal.

OpenCL employs a stricter form of scope inclusion, in which both events must additionally use the *same* memory scope. The version we use here follows a proposal called HRF-relaxed [10], and is a necessary prerequisite for RSP.

3. Formalising OpenCL+RSP

We now describe our first research contribution: how to extend the OpenCL memory model with RSP. The purpose of this extension is: (a) to enable programs that exploit RSP to be analysed (§4), and (b) to enable a proof that the implementation of the language features is correct (§5). Note that (b) is an important enabler for (a), because the program analysis would be meaningless were the OpenCL+RSP memory model impossible to implement.

Adding RSP to OpenCL first involves extending the syntax of the language, and to this end, we propose simply to add an additional parameter to each existing atomic function, which accepts either `N` (for non-remote) or `R` (for remote) – see Example 4 below. Second, we must extend the semantics of the language (i.e., the memory model). This requires changing just one definition, the $incl$ relation in Fig. 2, as follows:

$$\begin{aligned} incl \stackrel{\text{def}}{=}\ & incl1 \cap incl1^{-1} \\ & \cup ([rem] ; incl1) \cup (incl1^{-1} ; [rem]) \end{aligned}$$

where rem identifies the set of events representing a remote operation. In the original definition, both events must have wide enough scopes; in the new version, the events (say e_1 and e_2) may *also* synchronise if e_1's scope is wide enough to reach e_2 and e_1 is remote, or if e_2's scope is wide enough to reach e_1 and e_2 is remote.[3]

Example 4. If the program in Example 3 were changed so that the store became *remote*:

$$\texttt{fetch_inc}_{\texttt{WG,N}}(\texttt{x}) \;|||\; \texttt{store}_{\texttt{DV,R}}(\texttt{x}, 2)$$

then the scope-inclusion ($incl$) edges seen in the executions in Example 2 would be restored. □

The simple manner in which we can adapt the memory model for RSP illustrates the elegance of an axiomatic memory model. Our extension is conservative in the sense that it does not affect the semantics of OpenCL programs that do not exploit RSP. It is significantly simpler than a previous outline formalisation [19], and is further distinguished by being founded on an existing, comprehensive formalisation of OpenCL [24]. Although the details of implementing RSP are rather involved (as we discuss in detail in §5), the effect of RSP on the memory model is minimal. The minor modification to the $incl$ relation is all that is required to enable simulation of litmus tests that exploit RSP, which we discuss next.

[3] We initially sought to encode RSP by instead adding an extra disjunct to the $incl1$ relation: $incl1 \stackrel{\text{def}}{=} \ldots \cup ([rem] ; unv)$. Although seductively simple, this version does not capture the intended behaviour of RSP in the case where *both* participants are marked remote; rather, it would erroneously allow two remote `WG`-scoped operations to synchronise across different devices.

4. Testing OpenCL+RSP Programs

We extended the memory model simulator HERD to support the simulation of small OpenCL+RSP programs against the newly-extended memory model (§4.1). We then used HERD to analyse a suite of test programs that we obtained from the broader group of original RSP developers, uncovering several faults in the process (§4.2), and further exercised HERD to debug a larger OpenCL+RSP application: a work-stealing queue (§4.3).

4.1 Extending HERD

HERD is a generic memory model simulator [3]. Its basic operation is to generate and iterate through a set of candidate executions of a given litmus test, and assess whether each is consistent and/or faulty according to the axioms of a given memory model, as described using the `.cat` specification language. Originally designed for CPU assembly programs [3], HERD has recently been extended with the capacity to simulate C11 and OpenCL programs [24]. For the current work, we have extended HERD further, to support OpenCL+RSP. This entailed two sub-tasks: extending the front end of HERD to understand remote versions of atomic operations in litmus tests, and extending the memory model specification language with an additional identifier, rem, to stand for the set of remote events in an execution.

4.2 Litmus Testing

The original developers of RSP used a suite of 12 litmus tests to gain confidence in the correctness of their implementation. These tests, which are mostly variants on standard litmus tests, characterise the building blocks of parallel algorithms that use RSP. We obtained the suite from the RSP developers, and used HERD to simulate each test against our formalisation of the OpenCL+RSP memory model.

Preparation We encoded each program into the `.litmus` format that is accepted by HERD. In doing so, we observed that the original litmus tests make several uses of empty loops that spin until a given location holds a specific value, such as '`while (load(x) != 1); C`'. Because such programs have arbitrarily-many candidate executions (one for each possible iteration count) we followed standard practice in preparing litmus tests (e.g., Alglave et al. [1]) and replaced the pattern above with '`if (load(x) == 1) {C}`', ignoring those executions where the conditional test failed.

Results We performed simulation on a 2.8 GHz MacBook Pro, and observed that each litmus test was fully simulated in less than one second, except for two tests that made use of several compare-exchange instructions; each of these tests was fully simulated in less than three minutes.

Seven of the 12 litmus tests included postconditions on the final state of memory that we found, through analysis with HERD, to be satisfied by all consistent executions. Another test exhibited a deliberate data race that was confirmed by HERD.

Our analysis exposed bugs in the other four litmus tests. The first test had an unintentional data race, resulting from a discrepancy between the HSA 1.0 memory model [13] and the OpenCL 2.0 memory model. Specifically, OpenCL does not enforce happens-before between two operations that access different regions of memory even if they belong to the same thread, so one cannot assume the HSA behaviour, which does enforce happens-before here. The second test also contained a data race. The third had an incorrect postcondition due to a simple arithmetic error. The fourth had a postcondition that was too strong: it forbade certain executions that were allowed by the axioms of the memory model. As it happens, the proposed implementation does not give rise to the executions that this litmus test forbids; it can therefore be deemed a conservative implementation of the OpenCL specification in this regard.

4.3 Case Study: A Work-Stealing Queue

We used HERD to probe the correctness of a more realistic OpenCL+RSP application: a work-stealing queue. This application, is a key motivator for RSP [19: §3.2], and exploits RSP by simultaneously optimising for the common case of accessing the local task queue (by using `WG`-scoped non-remote operations in `push` and `pop`) and enabling the uncommon case of accessing a different work-group's queue (by using `DV`-scoped remote operations in `steal`).

Alglave et al. [1] have uncovered two bugs in a similar CUDA implementation [8] that led to tasks occasionally being dropped from queues; both bugs arose from assuming an overly-strong memory model. This was demonstrated by hand-compiling suspect slices of the CUDA code into GPU assembly litmus tests (named *dlb-mp* and *dlb-lb*) and showing experimentally that these tests could produce results that would lead to bugs at the CUDA level.

We have been able to demonstrate the complementary result for the OpenCL+RSP work-stealing queue. We produced OpenCL+RSP litmus tests capturing the same thread interactions that *dlb-mp* and *dlb-lb* captured at the GPU assembly level. Using HERD, we were able to verify the *absence* of the bugs associated with the original CUDA implementation, thanks to sufficient use of store-release and load-acquire functions to ensure necessary synchronisation. Because our result demonstrates correctness at the level of the programming language, it extends to *any* correct implementation of OpenCL+RSP.

We emphasise that we have not verified the entirety of the work-stealing queue implementation; we merely state that it is free from two specific bugs. Indeed, on the contrary, we found, reported, and confirmed a data race that could arise when performing a `pop` and a `steal` on the same queue. The race, which arises because `pop` can non-atomically write to the queue's tail pointer while the `steal` atomically reads from it, can be rectified by upgrading the non-atomic write to a relaxed atomic write.

5. A Formalised Implementation of OpenCL+RSP

Having studied the programming language semantics of RSP, we now turn our attention to formalising a low-level implementation of RSP, transforming the published description of the implementation of OpenCL+RSP into rigorous mathematics. Our formalisation comprises a mathematical model of a simple GPU device (§5.1), the syntax and semantics of a minimal assembly language for this device (§5.2 and §5.3) and a scheme for compiling OpenCL+RSP to assembly (§5.4 and §5.5).

Our formalisation effort found several opportunities to improve the original compilation scheme, ranging from improving inefficiencies to eliminating errors. Our revised compilation scheme is simpler than the original and addresses all of the errors and inefficiencies we found, hence we present the revised scheme first (§5.4), then explain the original scheme in terms of how it differs from our proposal (§5.5). In §6 we prove that our revised scheme is sound.

Tool support from Isabelle The definitions in this section have been formalised using the Isabelle proof assistant [18], and the scripts are available in our online companion material. We have also formalised the statement of our soundness theorem (Thm. 1, §6), but have not mechanised its proof. We found the type-checking and custom syntax that Isabelle provides to be invaluable while designing our model. We remark that the semantics of assembly instructions (§5.2), each of which updates various components in a deeply-nested structure of records and lists, is naturally expressed in an imperative style; because Isabelle demands a functional style, our formalisation differs in this respect from the current presentation.

5.1 A Model of GPU Hardware

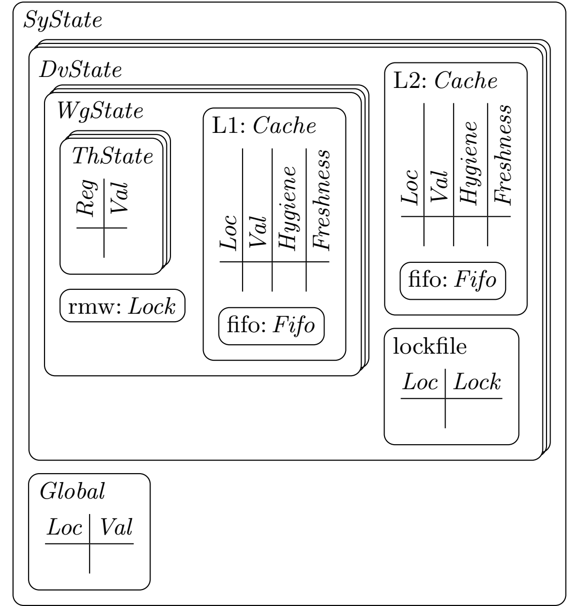
Our model of GPU hardware closely resembles the illustration in Fig. 1, which in turn is based on the model used for the original design of RSP [19: Fig. 4].

Figure 3a defines the set of states ($SyState$) that the machine can inhabit; this is defined in terms of numerous other sets, and in some cases we provide the name of a variable we shall use to range over the elements of the set. We use d, w and t to range over device, work-group and thread identifiers, all of which are natural numbers. In Fig. 3a we use an identifier followed by colon to name a component of a tuple so that, for instance, we can refer to the third component of a $CacheEntry$ E by writing $E.\mathrm{fr}$.

Complementing the formal definitions, Fig. 3b gives a pictorial representation of the machine state, rendering each

$$
\begin{array}{rrcl}
x \in & \mathit{Loc} & & \\
r \in & \mathit{Reg} & & \\
v \in & \mathit{Val} & \stackrel{\text{def}}{=} & \mathbb{Z} \\
& \mathit{FifoEl} & \stackrel{\text{def}}{=} & \mathit{Loc} \cup \{\text{FLUSH}_{d,w,t} \mid d,w,t \in \mathbb{N}\} \\
& \mathit{Fifo} & \stackrel{\text{def}}{=} & \mathit{FifoEl}\ \text{queue} \\
& \mathit{Hygiene} & \stackrel{\text{def}}{=} & \{\text{CLEAN}, \text{DIRTY}\} \\
& \mathit{Freshness} & \stackrel{\text{def}}{=} & \{\text{VALID}, \text{INV'D}\} \\
& \mathit{CacheEntry} & \stackrel{\text{def}}{=} & \mathit{Val} \times (\text{hy: } \mathit{Hygiene}) \times (\text{fr: } \mathit{Freshness}) \\
C \in & \mathit{Cache} & \stackrel{\text{def}}{=} & (\mathit{Loc} \rightharpoonup \mathit{CacheEntry}) \times (\text{fifo: } \mathit{Fifo}) \\
l \in & \mathit{Lock} & \stackrel{\text{def}}{=} & \{🔓\} \cup \{🔒_{d,w,t} \mid d,w,t \in \mathbb{N}\} \\
& \mathit{ThState} & \stackrel{\text{def}}{=} & \mathit{Reg} \to \mathit{Val} \\
& \mathit{WgState} & \stackrel{\text{def}}{=} & \mathit{ThState}\ \text{list} \times (\text{L1: } \mathit{Cache}) \times (\text{rmw: } \mathit{Lock}) \\
& \mathit{DvState} & \stackrel{\text{def}}{=} & \mathit{WgState}\ \text{list} \times (\text{L2: } \mathit{Cache}) \times \\
& & & (\text{lockfile: } \mathit{Loc} \to \mathit{Lock}) \\
& \mathit{Global} & \stackrel{\text{def}}{=} & \mathit{Loc} \rightharpoonup \mathit{Val} \\
\Sigma \in & \mathit{SyState} & \stackrel{\text{def}}{=} & \mathit{DvState}\ \text{list} \times (\text{gl: } \mathit{Global})
\end{array}
$$

(a) Formal definitions



(b) A machine state Σ, pictorially

Figure 3. Machine states

component as a rounded rectangle, and using a stacking effect to indicate a multiplicity of similar components.

The state of the system ($\mathit{SyState}$) comprises the state of each device plus the contents of global memory, which is a partial function from locations (Loc) to values (Val). We assume that all values are mathematical integers, and that global memory contains any location that is requested.

The state of a device ($\mathit{DvState}$) comprises the state of each of its work-groups, the contents of the L2 cache, and a 'lock file' that records, for each location, whether it is locked in the L2 cache ($🔒_{d,w,t}$, where d, w, t identifies the thread holding the lock) or unlocked (🔓). While a location is locked in the L2 cache by one thread, no other thread can read, write, evict, fetch, or flush it.

The state of a work-group ($\mathit{WgState}$) comprises the state of each of its threads, the contents of the L1 cache, plus an additional lock that stalls the execution of RMW operations (and is taken by threads executing remote RMW and store operations to ensure atomicity). The state of a thread ($\mathit{ThState}$) comprises its register file, which is a total function from registers to values. We assume an unlimited number of registers.

A cache (Cache) comprises two components: a partial function from locations to cache entries, and a synchronisation fifo. Each entry ($\mathit{CacheEntry}$) comprises a value, a hygiene bit (CLEAN or DIRTY), and a freshness bit (VALID or INV'D). Note that each location in memory has a separate cache entry (cf. Remark 1). The synchronisation fifo (Fifo) is a hardware component introduced as part of AMD's QuickRelease technology [11]. It is a queue whose elements (FifoEl) are locations that may need to be flushed to the lower levels of the cache; by inserting flush markers (FLUSH) among the locations, tagged with their own device/work-group/thread identifier, threads can ascertain which locations have been flushed. We assume that the queue datatype supports in-place $\mathrm{enqueue}()$ and $\mathrm{dequeue}()$ methods, and exposes a tail field.

Notation. We write Σ_d for the state of device d, Σ_{dw} for the state of work-group w in that device, and Σ_{dwt} for the state of thread t in that work-group. When we pass an $\mathit{Loc}\ x$ to a $\mathit{Cache}\ C$, writing $C(x)$, we are implicitly looking up x in the first of C's two components. □

5.2 Assembly Language

We formalise our assembly language so that the behaviour of each thread in each work-group in each device is specified independently. Accordingly, and in keeping with our formalisation of OpenCL (§2.2), an assembly program is a list (devices) of lists (work-groups) of lists (threads) of lists (instructions of a thread) of assembly instructions.

The assembly language instructions are listed in the left-hand column of Table 1. In summary, we have: loading from a location to a register, storing from a register to a location, atomically incrementing a location in the L1/L2 cache (this being the simplest representative of the class of atomic RMW operations), inserting a FLUSH marker into one or more L1 or L2 caches, invalidating all entries in one or more L1 caches, locking/unlocking a location in the L2 cache, and obtaining/releasing all of the RMW locks in the current work-group/device/system. Other standard instructions, and in particular control flow instructions, would be required to

Instr.	Effect on state Σ when executed by thread (d, w, t)
`LD` $r\ x$	**if** Σ_{dw}.L1(x).fr $=$ VALID **then** $\Sigma_{dwt}(r) := \Sigma_{dw}$.L1$(x)$ **else block**
`ST` $r\ x$	$store(\Sigma_{dw}.\mathrm{L1}, x, \Sigma_{dwt}(r))$

`INC`$_{\texttt{L1}}$ $r\ x$	**if** $\neg ready_{d,w,t}(\Sigma_{d\,w}.\mathrm{rmw})$ **then block**
	else if $\Sigma_{dw}.\mathrm{L1}(x) = (v, _, \mathrm{VALID})$ **then**
	  $\Sigma_{dwt}(r) := v$
	  $store(\Sigma_{dw}.\mathrm{L1}, x, v+1)$
	else block

`INC`$_{\texttt{L2}}$ $r\ x$	**if** $\neg ready_{d,w,t}(\Sigma_{d\,w}.\mathrm{rmw})$ **then block**
	else if $\Sigma_{dw}.\mathrm{L1}(x).\mathrm{hy} = \mathrm{DIRTY}$ **then block**
	else if $\neg ready_{d,w,t}(\Sigma_{d}.\mathrm{lockfile}(x))$ **then block**
	else if $\Sigma_{d}.\mathrm{L2}(x) = (v, _, \mathrm{VALID})$ **then**
	  $invalidate(\Sigma_{dw}.\mathrm{L1}, x)$
	  $\forall d'.\ invalidate(\Sigma_{d'}.\mathrm{L2}, x)$
	  $\Sigma_{dwt}(r) := v$
	  $store(\Sigma_{d}.\mathrm{L2}, x, v+1)$
	else block

`FLU`$_{\texttt{L1}}$ `WG`	$\Sigma_{dw}.\mathrm{L1.fifo.enqueue}(\mathrm{FLUSH}_{d,w,t})$
`FLU`$_{\texttt{L1}}$ `DV`	$\forall w'.\ \Sigma_{dw'}.\mathrm{L1.fifo.enqueue}(\mathrm{FLUSH}_{d,w,t})$
`FLU`$_{\texttt{L1}}$ `SY`	$\forall d'.\ \forall w'.\ \Sigma_{d'w'}.\mathrm{L1.fifo.enqueue}(\mathrm{FLUSH}_{d,w,t})$
`FLU`$_{\texttt{L2}}$ `DV`	$\Sigma_{d}.\mathrm{L2.fifo.enqueue}(\mathrm{FLUSH}_{d,w,t})$
`FLU`$_{\texttt{L2}}$ `SY`	$\forall d'.\ \Sigma_{d'}.\mathrm{L2.fifo.enqueue}(\mathrm{FLUSH}_{d,w,t})$
`INV`$_{\texttt{L1}}$ `WG`	$\forall x.\ invalidate(\Sigma_{dw}.\mathrm{L1}, x)$
`INV`$_{\texttt{L1}}$ `DV`	$\forall w'.\ \forall x.\ invalidate(\Sigma_{dw'}.\mathrm{L1}, x)$
`INV`$_{\texttt{L1}}$ `SY`	$\forall d'.\ \forall w'.\ \forall x.\ invalidate(\Sigma_{d'w'}.\mathrm{L1}, x)$
`LK`$_{\texttt{L2}}$ x	**if** $\neg ready_{d,w,t}(\Sigma_{d}.\mathrm{lockfile}(x))$ **then block** **else** $\Sigma_{d}.\mathrm{lockfile}(x) :=$ 🔒$_{d,w,t}$
`UL`$_{\texttt{L2}}$ x	$\Sigma_{d}.\mathrm{lockfile}(x) :=$ 🔓
`LK`$_{\texttt{rmw}}$ `DV`	**if** $\exists w'.\ \neg ready_{d,w,t}(\Sigma_{d\,w'}.\mathrm{rmw})$ **then block** **else** $\forall w'.\ \Sigma_{d\,w'}.\mathrm{rmw} :=$ 🔒$_{d,w,t}$
`LK`$_{\texttt{rmw}}$ `SY`	**if** $\exists d'.\ \exists w'.\ \neg ready_{d,w,t}(\Sigma_{d'\,w'}.\mathrm{rmw})$ **then block** **else** $\forall d'.\ \forall w'.\ \Sigma_{d'\,w'}.\mathrm{rmw} :=$ 🔒$_{d,w,t}$
`UL`$_{\texttt{rmw}}$ `DV`	$\forall w'.\ \Sigma_{d\,w'}.\mathrm{rmw} :=$ 🔓
`UL`$_{\texttt{rmw}}$ `SY`	$\forall d'.\ \forall w'.\ \Sigma_{d'\,w'}.\mathrm{rmw} :=$ 🔓

where:

$store(C, x, v) \stackrel{\mathrm{def}}{=} C(x) := (v, \mathrm{DIRTY}, \mathrm{VALID})$; $C.\mathrm{fifo.enqueue}(x)$

$ready_{d,w,t}(l) \stackrel{\mathrm{def}}{=} (l =$ 🔒$_{d,w,t}) \vee (l =$ 🔓$)$

$invalidate(C, x) \stackrel{\mathrm{def}}{=}$ **if** $C(x) \neq \bot$ **then** $C(x).\mathrm{fr} := \mathrm{INV'D}$ **else nop**

Table 1. Semantics of assembly instructions

provide a complete set; we limit the presentation here to those that manipulate the memory system.

Table 1 also defines the effect of each assembly instruction when executed from state Σ by thread t in work-group w in device d. Formally, each instruction is modelled as a non-deterministic state transformer: a function from $SyState$ to $\mathcal{P}(SyState)$. A blocked instruction returns the empty set, denoted **block**. For the time being, no instruction produces more than one final state,[4] so we define each instruction using deterministic, imperative pseudocode. We overload the $\forall$-operator to provide an imperative **foreach** construct, leaving the bounds implicit.

These pieces of pseudocode leave only one other aspect of the instructions' behaviours implicit: each piece of pseudocode, $action$, should be made conditional as follows:

$$\textbf{if } unflushed_{d,w,t}(\Sigma) \textbf{ then block else } action$$

where

$$unflushed_{d,w,t}(\Sigma) \stackrel{\mathrm{def}}{=} (\exists d'.\ \mathrm{FLUSH}_{d,w,t} \in \Sigma_{d'}.\mathrm{L2.fifo}) \vee (\exists d', w'.\ \mathrm{FLUSH}_{d,w,t} \in \Sigma_{d'w'}.\mathrm{L1.fifo}).$$

That is, a thread that has placed a FLUSH marker in an L1 or L2 fifo must block until its marker is dequeued.

Loads and stores Regarding loads (`LD`) from location x: if x's L1 cache entry is valid, the cached value is copied into the register file accordingly. Otherwise, the instruction blocks, waiting for the environment to fetch a valid entry from deeper in the cache hierarchy. In practice, the load would initiate this fetch, but since our interest is in checking safety properties, the existence of an environmental transition that will fetch the new entry means that it suffices to suppose that the load simply blocks. We describe environmental transitions in §5.3.

Stores (`ST`) to location x simply write to x's L1 entry, adding a new entry if none exists, and overwriting any previous entry. The entry is marked dirty and valid (via the $store$ helper function) and the location is enqueued to the cache's synchronisation fifo.

Atomic increments The `INC`$_{\texttt{L1}}$ and `INC`$_{\texttt{L2}}$ instructions are RMW operations, so they block if the rmw lock is held by another thread. `INC`$_{\texttt{L1}}$ increments x in the L1 cache, writing the original value to the given register; it blocks until the L1 cache holds a valid entry for x (as with loads, we rely on environmental transitions to provide this valid entry). If the L1 entry for x is dirty, the instruction blocks until it is flushed; otherwise, the L1 entry, if present, is invalidated. If access to x's L2 entry is forbidden (by another thread holding x's lockfile entry), then the instruction blocks. The instruction also blocks if x's L2 entry is invalid or absent; otherwise it increments x in the L2 cache. When storing to the L2 cache,

[4] While conducting our soundness proof, we make use of an alternative semantics that 'disengages' the memory system, making loads completely non-deterministic.

all of x's entries in other devices' L2 caches are invalidated (via the $\forall d'.\, invalidate(\Sigma_{d'}.\mathrm{L2}, x)$ step), to preserve cache coherence.

Flushes and invalidates The $\mathtt{FLU}_{\mathtt{L1}}$ instruction enqueues a flush marker, tagged with the current thread's identifier, into the current L1 cache, or all L1 caches in the current device, or all L1 caches in the system, depending on whether the instruction is parameterised by `WG`, `DV`, or `SY`, respectively. $\mathtt{FLU}_{\mathtt{L2}}$ enqueues a flush marker into the current L2 cache, or all L2 caches in the system, depending on whether it is parameterised by `DV` or `SY`, respectively. The $\mathtt{INV}_{\mathtt{L1}}\,\{\mathtt{WG}, \mathtt{DV}, \mathtt{SY}\}$ instruction invalidates all entries in all L1 caches in the {work-group, device, system}.

Locks $\mathtt{LK}_{\mathtt{L2}}\, x$ and $\mathtt{UL}_{\mathtt{L2}}\, x$ respectively lock and unlock the location x in the current L2 cache, the former blocking if the lock is currently held by another thread. $\mathtt{LK}_{\mathtt{rmw}}$ and $\mathtt{UL}_{\mathtt{rmw}}$ respectively obtain and release all the rmw locks in the given scope. While a work-group's rmw lock is held by one thread, no other thread in that work-group can perform an RMW operation.

Reducing non-standard hardware requirements Some of the instructions above require non-standard hardware support: specifically, the ability for a thread to flush/invalidate caches that are not in its direct path to global memory, to lock cachelines, and to lock RMW operations [19: §4.4]. An attractive feature of the revised RSP implementation inspired by our formalisation effort (§5.4) is that it does *not* use cacheline locking, and thus requires less non-standard hardware.

5.3 Environmental Transitions

At any time, the 'environment' can transform the system state. Environment transitions do not correspond to program instructions, but each is triggered by a particular thread.

Locks aside, we point out that if an instruction is blocked, there is always an environmental transition, or a series of environmental transitions, that will result in the instruction becoming unblocked. Locks, meanwhile, present the possibility of deadlock if used carelessly.

The available environmental transitions are defined, in Tab. 2, by their effect on the current system state Σ. Each cell in the right-hand column of the table takes the form **if** *precondition* **then** *action*, to reflect the fact that the transition can only occur under certain conditions.

The transitions are: evicting a clean cache entry ($\textsc{Evict}_{\mathrm{L1}}$ and $\textsc{Evict}_{\mathrm{L2}}$), flushing a dirty cache entry and marking it clean ($\textsc{Flush}_{\mathrm{L1}}$ and $\textsc{Flush}_{\mathrm{L2}}$), replacing a clean-or-absent cache entry by fetching from the level below ($\textsc{Fetch}_{\mathrm{L1}}$ and $\textsc{Fetch}_{\mathrm{L2}}$), removing a location whose cache entry is clean-or-absent from the tail of a fifo ($\textsc{DeqLoc}_{\mathrm{L1}}$ and $\textsc{DeqLoc}_{\mathrm{L2}}$), and removing a $\textsc{flush}$ marker from the tail of a fifo ($\textsc{DeqMarker}_{\mathrm{L1}}$ and $\textsc{DeqMarker}_{\mathrm{L2}}$).

Regarding the $\textsc{Fetch}_{\mathrm{L1}}$ action, notice that the newly-fetched entry is always marked $\textsc{clean}$, even if the L2 entry is $\textsc{dirty}$. There is no need to mark the L1 copy as dirty: the value it holds is the same as the value that will be propagated to global memory once the L2 entry (eventually) flushes.

Remark 1 (On caching protocols)**.** We model caches as if each cacheline holds the contents of a single location, but real cachelines hold the contents of several consecutive locations. Therefore, real caches may fetch more than just the requested location; we model this by allowing any location to be fetched at any time. Real caches may flush multiple locations simultaneously, but since they use a dirty bit mask,

Name	Effect on state Σ when triggered by thread (d, w, t)
$\textsc{Evict}_{\mathrm{L1}}(x)$	**if** $\Sigma_{dw}.\mathrm{L1}(x).\mathrm{hy} = \textsc{clean}$ **then** $\Sigma_{dw}.\mathrm{L1}(x) := \bot$
$\textsc{Evict}_{\mathrm{L2}}(x)$	**if** $\Sigma_{d}.\mathrm{L2}(x).\mathrm{hy} = \textsc{clean}$ **and** $ready_{d,w,t}(\Sigma_d.\mathrm{lockfile}(x))$ **then** $\Sigma_{d}.\mathrm{L2}(x) := \bot$
$\textsc{Flush}_{\mathrm{L1}}(x, v)$	**if** $\Sigma_{dw}.\mathrm{L1}(x) = (v, \textsc{dirty}, _)$ **and** $ready_{d,w,t}(\Sigma_d.\mathrm{lockfile}(x))$ **then** $\forall d'.\, \Sigma_{d'}.\mathrm{L2}(x).\mathrm{fr} := \textsc{inv'd}$ $store(\Sigma_d.\mathrm{L2}, x, v)$ $\Sigma_{dw}.\mathrm{L1}(x).\mathrm{hy} := \textsc{clean}$
$\textsc{Flush}_{\mathrm{L2}}(x, v)$	**if** $\Sigma_{d}.\mathrm{L2}(x) = (v, \textsc{dirty}, _)$ **and** $ready_{d,w,t}(\Sigma_d.\mathrm{lockfile}(x))$ **then** $\Sigma.\mathrm{gl}(x) := v$ $\Sigma_{d}.\mathrm{L2}(x).\mathrm{hy} := \textsc{clean}$
$\textsc{Fetch}_{\mathrm{L1}}(x, v)$	**if** $notDirty(\Sigma_{dw}.\mathrm{L1}, x)$ **and** $\Sigma_d.\mathrm{L2}(x) = (v, _, \textsc{valid})$ **and** $ready_{d,w,t}(\Sigma_d.\mathrm{lockfile}(x))$ **then** $\Sigma_{dw}.\mathrm{L1}(x) := (v, \textsc{clean}, \textsc{valid})$
$\textsc{Fetch}_{\mathrm{L2}}(x, v)$	**if** $notDirty(\Sigma_{d}.\mathrm{L2}, x)$ **and** $\Sigma.\mathrm{gl}(x) = v$ **and** $ready_{d,w,t}(\Sigma_d.\mathrm{lockfile}(x))$ **then** $\Sigma_{d}.\mathrm{L2}(x) := (v, \textsc{clean}, \textsc{valid})$
$\textsc{DeqLoc}_{\mathrm{L1}}(x)$	**if** $\Sigma_{dw}.\mathrm{L1.fifo.tail} = x$ **and** $notDirty(\Sigma_{dw}.\mathrm{L1}, x)$ **then** $\Sigma_{dw}.\mathrm{L1.fifo.dequeue}()$
$\textsc{DeqLoc}_{\mathrm{L2}}(x)$	**if** $\Sigma_{d}.\mathrm{L2.fifo.tail} = x$ **and** $notDirty(\Sigma_{d}.\mathrm{L2}, x)$ **then** $\Sigma_{d}.\mathrm{L2.fifo.dequeue}()$
$\textsc{DeqMarker}_{\mathrm{L1}}$	**if** $\Sigma_{dw}.\mathrm{L1.fifo.tail} = \textsc{flush}$ **then** $\Sigma_{dw}.\mathrm{L1.fifo.dequeue}()$
$\textsc{DeqMarker}_{\mathrm{L2}}$	**if** $\Sigma_{d}.\mathrm{L2.fifo.tail} = \textsc{flush}$ **then** $\Sigma_{d}.\mathrm{L2.fifo.dequeue}()$

where:

$$notDirty(C, x) \stackrel{\text{def}}{=} (C(x) = \bot) \lor (C(x).\mathrm{hy} = \textsc{clean})$$

Table 2. Environmental transitions

it is as if the flush is per-location. Real cacheline locking may restrict access to more locations than our model suggests, but this extra locking can only lead to fewer behaviours, thus making our model sound. The caches used in the evaluated design are *write-through* and *write-allocate* [19: Tab. 1];[5] we safely capture write-through behaviour by allowing the environment to flush at any time, and write-allocate behaviour by having ST create a cache entry if none exists. □

5.4 Compilation Scheme

We now consider the compilation of OpenCL+RSP programs into the assembly language of §5.2.

Although our assembly language can apply to a multiple-device system, this subsection, in line with the original RSP proposal, considers only the single-device case [19: §5]. This means that our compilation scheme does not extend to OpenCL+RSP operations that use ALL-scope.

The compilation scheme shown in the 'Original' column of Tab. 3 is what we believe to be a faithful representation of the original proposed scheme [19], informed by a series of interviews with the broader set of RSP designers. As a result of our formalisation work, we have found problems with this scheme, which we elucidate in §5.5. We propose instead the compilation scheme shown in the 'Proposed' column of Tab. 3, which addresses these problems. We now explain and justify our proposed scheme; a proof that it is sound follows in §6.

Our explanation centres on how the compilation scheme ensures correct release/acquire semantics (so that inter-work-group message-passing programs, such as

$$\begin{array}{l|||l} \mathtt{store}_{\mathtt{na}}(\mathtt{x}, 42); & \mathtt{if}(\mathtt{r0} = \mathtt{load}_{\mathtt{DV,N}}(\mathtt{y})) \\ \mathtt{store}_{\mathtt{DV,N}}(\mathtt{y}, 1); & \quad \mathtt{r1} = \mathtt{load}_{\mathtt{na}}(\mathtt{x}); \end{array}$$

can never yield $\{\mathtt{r0} = 1, \mathtt{r1} = 0\}$) and also ensures RMW atomicity (so that programs like the one in Example 3 can never finish with $\mathtt{x} = 1$).

	OpenCL+RSP operation	Original	Proposed
❶	$r = \mathtt{load}_{\mathtt{na}}(x)$ $r = \mathtt{load}_{\mathtt{WG},_}(x)$	LD r x	LD r x
❷	$r = \mathtt{load}_{\mathtt{DV,N}}(x)$	$\mathtt{INV}_{\mathtt{L1}}$ WG LD r x	LD r x $\mathtt{INV}_{\mathtt{L1}}$ WG
❸	$r = \mathtt{load}_{\mathtt{DV,R}}(x)$	$\mathtt{LK}_{\mathtt{L2}}$ x $\mathtt{FLU}_{\mathtt{L1}}$ DV $\mathtt{INV}_{\mathtt{L1}}$ WG LD r x $\mathtt{UL}_{\mathtt{L2}}$ x	LD r x $\mathtt{FLU}_{\mathtt{L1}}$ DV $\mathtt{INV}_{\mathtt{L1}}$ WG
❹	$\mathtt{store}_{\mathtt{na}}(x, r)$ $\mathtt{store}_{\mathtt{WG},_}(x, r)$	ST r x	ST r x
❺	$\mathtt{store}_{\mathtt{DV,N}}(x, r)$	$\mathtt{FLU}_{\mathtt{L1}}$ WG ST r x	$\mathtt{FLU}_{\mathtt{L1}}$ WG ST r x
❻	$\mathtt{store}_{\mathtt{DV,R}}(x, r)$	$\mathtt{LK}_{\mathtt{L2}}$ x $\mathtt{FLU}_{\mathtt{L1}}$ WG ST r x $\mathtt{INV}_{\mathtt{L1}}$ DV $\mathtt{UL}_{\mathtt{L2}}$ x	$\mathtt{LK}_{\mathtt{rmw}}$ DV $\mathtt{FLU}_{\mathtt{L1}}$ DV $\mathtt{INV}_{\mathtt{L1}}$ DV ST r x $\mathtt{FLU}_{\mathtt{L1}}$ WG $\mathtt{INV}_{\mathtt{L1}}$ DV $\mathtt{UL}_{\mathtt{rmw}}$ DV
❼	$r = \mathtt{fetch_inc}_{\mathtt{WG},_}(x)$	$\mathtt{INC}_{\mathtt{L1}}$ r x	$\mathtt{INC}_{\mathtt{L1}}$ r x
❽	$r = \mathtt{fetch_inc}_{\mathtt{DV,N}}(x)$	$\mathtt{FLU}_{\mathtt{L1}}$ WG $\mathtt{INV}_{\mathtt{L1}}$ WG $\mathtt{INC}_{\mathtt{L2}}$ r x	$\mathtt{FLU}_{\mathtt{L1}}$ WG $\mathtt{INC}_{\mathtt{L2}}$ r x $\mathtt{INV}_{\mathtt{L1}}$ WG
❾	$r = \mathtt{fetch_inc}_{\mathtt{DV,R}}(x)$	$\mathtt{LK}_{\mathtt{rmw}}$ DV $\mathtt{LK}_{\mathtt{L2}}$ x $\mathtt{FLU}_{\mathtt{L1}}$ DV $\mathtt{INV}_{\mathtt{L1}}$ WG $\mathtt{INC}_{\mathtt{L2}}$ r x $\mathtt{INV}_{\mathtt{L1}}$ DV $\mathtt{UL}_{\mathtt{L2}}$ x $\mathtt{UL}_{\mathtt{rmw}}$ DV	$\mathtt{LK}_{\mathtt{rmw}}$ DV $\mathtt{FLU}_{\mathtt{L1}}$ DV $\mathtt{INV}_{\mathtt{L1}}$ DV $\mathtt{INC}_{\mathtt{L2}}$ r x $\mathtt{FLU}_{\mathtt{L1}}$ DV $\mathtt{INV}_{\mathtt{L1}}$ DV $\mathtt{UL}_{\mathtt{rmw}}$ DV

Table 3. Compilation schemes, original and proposed

Loads An OpenCL load that is non-atomic (na) or at work-group (WG) scope is compiled to a lone LD instruction ❶. No further instructions are required because consistency need only be enforced as far as the L1 cache, which LD already targets natively. For a load at DV scope ❷, we ensure 'acquire' semantics by invalidating the L1 cache after the LD. This ensures that subsequent loads observe values from the L2 cache.

To upgrade the load to a remote load, the invalidation is preceded by a flush of all the L1 caches in the device ❸; this ensures that subsequent loads observe values that have been written to any L1 cache.

Stores As for loads, non-atomic or wg-scope stores are compiled to lone ST instructions ❹. For a store at DV scope, we ensure 'release' semantics by flushing the L1 cache before the ST. This ensure that prior stores are visible to operations that subsequently read from the L2 cache ❺.

To upgrade to a remote store ❻, we must also ensure that prior stores are visible to operations that subsequently read from their own L1 cache. For this, we precede the ST instruction with a remote invalidate ($\mathtt{INV}_{\mathtt{L1}}$ DV). The other instructions are present to ensure that any wg-scoped increments, simultaneously executing on different work-group, are performed atomically. Without them, we might observe such violations of RMW atomicity as were seen earlier in Example 2. Naturally, these increments cannot happen between the $\mathtt{LK}_{\mathtt{rmw}}$ and $\mathtt{UL}_{\mathtt{rmw}}$ instructions. In case one happens *before* the $\mathtt{LK}_{\mathtt{rmw}}$, we use a remote flush ($\mathtt{FLU}_{\mathtt{L1}}$ DV) to ensure that it is promptly flushed, and thus unable to overwrite our upcoming store. In case one happens *after* the $\mathtt{UL}_{\mathtt{rmw}}$, we use a local

[5] The original RSP description reported the caches as being *write-no-allocate*, but we confirm that this was in error.

flush and a remote invalidate after our store, to ensure that the increment will observe our stored value.

Atomic increments We map the atomic fetch-and-increment operation, `fetch_inc`, to the `INC` assembly instruction. At `WG` scope, the `INC` operation acts on the L1 cache ❼, and at `DV` scope, it acts directly on the L2 cache ❽. Performing the operation directly on the L2 cache (rather than on the L1 cache and then flushing) ensures the atomicity of RMW operations. Moreover, at `DV` scope, the `fetch_inc` must provide acquire+release semantics. Accordingly, it begins with a flush (inherited from the release store, ❺) and finishes with an invalidate (inherited from the acquire load, ❷).

Upgrading to a remote increment imposes several requirements ❾. To ensure acquire semantics, the remote increment must end with (at least) a remote-flush and a local-invalidate (as for the remote load discussed above). To ensure release semantics, it must begin with (at least) a local-flush and a remote-invalidate (as for the remote store). For RMW atomicity, it must observe any concurrent increments on another L1 cache that occur before the RMW lock is acquired, and any that happen after the RMW lock is released must observe the newly-incremented value. Therefore, the remote increment must begin with (at least) a remote-flush and a local-invalidate, and end with (at least) a local-flush and a remote-invalidate. Merging all these constraints together, we find remote flushes and invalidates are required both before and after the $\mathtt{INC}_{\mathrm{L2}}$.

5.5 Problems with the Original Scheme

We describe two errors in the original compilation scheme (Tab. 3, 'Original' column), and explain how our proposed scheme ('Proposed' column) addresses them. Finding these issues early in the design process is, we believe, the key value of formalisation efforts such as ours.

5.5.1 Non-Remote Loads Can Violate Message-Passing

Because `DV`-scoped non-remote loads (❷ in Tab. 3) invalidate their L1 cache *before* the `LD` [19: §2.3], coherence (axiom Coh in Fig. 2) is violated. To explain this, Fig. 4 exhibits a machine execution, obtained by compiling a basic inter-work-group message-passing idiom, that leads to a prohibited final state. The vertical order in the figure illustrates the interleaving of each thread's instructions and environment transitions, and we include frequent snapshots of the machine's state. In braces, we show the cache entries (using C = CLEAN, D = DIRTY, V = VALID, and I = INVALID), and in brackets we show the contents of non-empty synchronisation fifos. Since our compilation scheme covers only the single device case, we treat the shared L2 cache as if it is global memory.

Initially, all memory is zeroed and all caches are empty. If `r0` observes 1, then coherence on `x` dictates that `r1` must observe 42. Thread 2 invalidates its L1 cache, but then the environment immediately repopulates it with `x`'s initial value. Thread 1 (which, being in a different work-group, has its own

Thread 1 (executing $\mathtt{store}_{\mathtt{na}}(\mathtt{x}, 42)$; $\mathtt{store}_{\mathtt{DV,N}}(\mathtt{y}, 1)$;)	**L2 cache**	**Thread 2** (executing $\mathtt{if}(\mathtt{r0} = \mathtt{load}_{\mathtt{DV,N}}(\mathtt{y}))$ $\mathtt{r1} = \mathtt{load}_{\mathtt{na}}(\mathtt{x})$;)
$\{\}$	$\{\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 0\}$	$\{\}$
		$\mathtt{INV}_{\mathrm{L1}}$ `WG`
		$\{\}$
		$\mathrm{FETCH}_{\mathrm{L1}}(\mathtt{x}, 0)$
		$\{\mathtt{x} \mapsto_{\mathrm{C,V}} 0\}$
ST 42 x		
$\{\mathtt{x} \mapsto_{\mathrm{D,V}} 42\}$ $[\mathtt{x}]$		
$\mathrm{FLUSH}_{\mathrm{L1}}(\mathtt{x}, 42)$		
$\{\mathtt{x} \mapsto_{\mathrm{C,V}} 42\}$ $[\mathtt{x}]$	$\{\mathtt{x} \mapsto 42, \mathtt{y} \mapsto 0\}$	
$\mathtt{FLU}_{\mathrm{L1}}$ `WG`		
$\{\mathtt{x} \mapsto_{\mathrm{C,V}} 42\}$ $[\mathrm{FLUSH}\ \mathtt{x}]$		
$\mathrm{DEQLOC}_{\mathrm{L1}}$		
$\{\mathtt{x} \mapsto_{\mathrm{C,V}} 42\}$ $[\mathrm{FLUSH}]$		
$\mathrm{DEQMARKER}_{\mathrm{L1}}$		
$\{\mathtt{x} \mapsto_{\mathrm{C,V}} 42\}$		
ST 1 y		
$\{\mathtt{x} \mapsto_{\mathrm{C,V}} 42, \mathtt{y} \mapsto_{\mathrm{D,V}} 1\}$ $[\mathtt{y}]$		
$\mathrm{FLUSH}_{\mathrm{L1}}(\mathtt{y}, 1)$		
$\{\mathtt{x} \mapsto_{\mathrm{C,V}} 42, \mathtt{y} \mapsto_{\mathrm{C,V}} 1\}$ $[\mathtt{y}]$	$\{\mathtt{x} \mapsto 42, \mathtt{y} \mapsto 1\}$	
		$\mathrm{FETCH}_{\mathrm{L1}}(\mathtt{y}, 1)$
		$\{\mathtt{x} \mapsto_{\mathrm{C,V}} 0, \mathtt{y} \mapsto_{\mathrm{C,V}} 1\}$
		LD r0 y
		LD r1 x

Figure 4. An execution of inter-work-group message-passing that leads to the illegal final state $\{\mathtt{r0} = 1, \mathtt{r1} = 0\}$

L1 cache) then stores new values for `x` and `y` into the L2 cache. When Thread 2 resumes, it fetches the updated `y`, but reads the stale `x` from its own L1 cache.

This bug recurs in non-remote `DV`-scoped increments (❽ in Tab. 3): the invalidate preceding the $\mathtt{INC}_{\mathrm{L2}}$ instruction leads to a similar message-passing violation.

The inversion of the $\mathtt{INV}_{\mathrm{L1}}$ and `LD` instructions is also carried through to remote loads (❸) [19: §4.2]. However, an undocumented detail of the original implementation actually prevents the bug recurring in this case. Specifically, accesses to the relevant L2 cacheline during remote loads are stalled using $\mathtt{LK}_{\mathrm{L2}}$ and $\mathtt{UL}_{\mathrm{L2}}$ instructions. The problematic execution depicted in Fig. 4 relies on Thread 2 fetching `x` after its L1 invalidation and before Thread 1 flushes its new `x`. Cacheline locking prevents this from happening, and thus restores coherence; however, it could lead to unnecessary inter-thread interference, and is more complicated to reason about than our proposed lock-free version.

Thread 1 (executing `fetch_inc`$_{\mathtt{WG,N}}$`(x);`)	**L2 cache**	**Thread 2** (executing `store`$_{\mathtt{DV,R}}$`(x, 2);`)
	$\{\mathtt{x} \mapsto 0\}$	
		$\textsc{Fetch}_{\mathrm{L1}}(\mathtt{x}, 0)$
		$\{\mathtt{x} \mapsto_{\mathrm{C,V}} 0\}$
$\textsc{Fetch}_{\mathrm{L1}}(\mathtt{x}, 0)$		
$\{\mathtt{x} \mapsto_{\mathrm{C,V}} 0\}$		
$\mathtt{INC}_{\mathtt{L1}}\ \mathtt{x}$		
$\{\mathtt{x} \mapsto_{\mathrm{D,V}} 1\}$ $[\mathtt{x}]$		
		$\mathtt{LK}_{\mathtt{L2}}\ \mathtt{x}$
		$\mathtt{FLU}_{\mathtt{L1}}\ \mathtt{WG}$
		$\{\mathtt{x} \mapsto_{\mathrm{C,V}} 0\}$ $[\textsc{flush}]$
		$\textsc{DeqMarker}_{\mathrm{L1}}$
		$\{\mathtt{x} \mapsto_{\mathrm{C,V}} 0\}$
		`ST 2 x`
		$\{\mathtt{x} \mapsto_{\mathrm{D,V}} 2\}$ $[\mathtt{x}]$
		$\mathtt{INV}_{\mathtt{L1}}\ \mathtt{DV}$
$\{\mathtt{x} \mapsto_{\mathrm{D,I}} 1\}$ $[\mathtt{x}]$		$\{\mathtt{x} \mapsto_{\mathrm{D,I}} 2\}$ $[\mathtt{x}]$
		$\mathtt{UL}_{\mathtt{L2}}\ \mathtt{x}$
		$\textsc{Flush}_{\mathrm{L1}}(\mathtt{x}, 2)$
	$\{\mathtt{x} \mapsto 2\}$	$\{\mathtt{x} \mapsto_{\mathrm{C,I}} 2\}$ $[\mathtt{x}]$
$\textsc{Flush}_{\mathrm{L1}}(\mathtt{x}, 1)$		
$\{\mathtt{x} \mapsto_{\mathrm{C,I}} 1\}$ $[\mathtt{x}]$	$\{\mathtt{x} \mapsto 1\}$	

Figure 5. An execution of the program in Example 3 that leads to the illegal final state $\{\mathtt{x} = 1\}$

Our proposed scheme avoids these problems by invalidating the L1 cache *after* performing the `LD` instruction (❷) or the $\mathtt{INC}_{\mathtt{L2}}$ instruction (❸). In the context of Fig. 4, this ensures that `x` is re-fetched after loading `y`.

5.5.2 Remote Stores Can Violate RMW Atomicity

The original implementation of `DV`-scoped remote stores (❻ in Tab. 3) does not ensure RMW atomicity (axiom Rmw in Fig. 2) in the presence of a `WG`-scoped increment in a different work-group. Figure 5 exhibits an execution, obtained by compiling the program in Example 3, that leads to a prohibited final state.

Both threads begin by fetching $\mathtt{x} = 0$ into their L1 caches. Thread 1 performs its increment, which leaves its L1 cache holding $\mathtt{x} = 1$, then thread 2 writes $\mathtt{x} = 2$ into its own L1 cache. Thread 2's write is flushed to the shared L2 first, and then overwritten by Thread 1's write, to leave the final state of $\mathtt{x} = 1$.

Thus the remote store does not force local increments in other work-groups to flush their results to the shared L2.

Our proposed scheme rectifies this by including a `DV`-scoped flush (before the `ST` instruction). This alone does not restore RMW atomicity because a similar problem arises if Thread 1 fetches $\mathtt{x} = 0$ from the L2 cache after Thread 2 has released its lock on `x`'s L2 cacheline but before it flushes $\mathtt{x} = 2$. To deal with this situation, we include a further flush (this time at `WG`-scope) *after* the `ST` instruction.

6. Proving Soundness

We now prove our remedied implementation of RSP (Tab. 3, right column) to be sound. The existence of this proof, which links the high-level OpenCL+RSP memory model (§3) to the low-level implementation of OpenCL+RSP, lends assurance that the analysis we performed on the litmus tests and work-stealing queue in §4 is not vacuous.

6.1 Soundness Statement

The soundness theorem relates three components: the formal semantics of the OpenCL+RSP language, the formal semantics of the assembly language, and the compilation scheme for mapping programs for the former to the latter. It essentially states that if Q is the assembly program that results from compiling an OpenCL+RSP program P, then every execution of Q is observationally equivalent to some consistent execution of P. The precise statement is as follows.

Theorem 1 (Soundness)**.** *For an OpenCL+RSP program P:*

$$\forall Y \in \mathcal{A}[\![\mathit{compile}\ P]\!].\, \exists X \in \mathcal{O}[\![P]\!].\, X \cong Y$$

where $\mathcal{A}[\![-]\!]$ returns the set of executions of a given assembly program, compile applies the mapping in Tab. 3 (right column), $\mathcal{O}[\![-]\!]$ returns the set of executions allowed of a given OpenCL+RSP program, and $\cong$ captures a notion of observational equivalence between executions.

OpenCL+RSP executions Recall from §2.2 that the pre-executions of an OpenCL+RSP program are those that can be obtained under the assumption of a completely non-deterministic memory (i.e., loads can obtain any value). We write $\mathcal{O}\langle - \rangle$ for the set of candidate executions whose pre-executions match a given OpenCL+RSP program. The set of executions allowed of a given OpenCL+RSP program P, written $\mathcal{O}[\![P]\!]$, is defined as either the set of consistent executions (when none of them has a race) or the universal set (otherwise); that is,

$$\mathcal{O}[\![P]\!] \stackrel{\text{def}}{=} \begin{cases} \mathit{unv} & \text{if } \exists X \in \mathit{Xs}.\, \mathit{faulty}(X) \\ \mathit{Xs} & \text{otherwise} \end{cases}$$

where $\mathit{Xs} \stackrel{\text{def}}{=} \{X \in \mathcal{O}\langle P \rangle \mid \mathit{consistent}(X)\}$.

Assembly executions The set of executions allowed of a given assembly program Q, written $\mathcal{A}[\![Q]\!]$, is the set of all possible complete executions of the program under the semantics of assembly instructions defined in §5.2. Given an initial state, each of an assembly program's completed executions can be obtained by interleaving program actions (Tab. 1) and environmental actions (Tab. 2) until all of the program's instructions have been dispatched (respecting the order in each thread) and all of the caches have been flushed down to global memory. Then, from each of these completed executions, we can project an execution graph comprising the memory accesses that were made. It is the set of these projected execution graphs that are calculated by $\mathcal{A}[\![-]\!]$.

In fact, we implement the projection by augmenting the semantics of instructions so as to transform not only the current state, but also to accumulate the execution graph. This execution graph is initially empty, and is extended with an additional event at each program step. For instance, `LD` appends an R event, while $\mathtt{INC}_{\mathtt{L1}}$ appends an RMW event. Ultimately, $\mathcal{A}[\![-]\!]$ is defined as the set of execution graphs that accompany terminal states.

Assembly events Recall the OpenCL memory events defined in Fig. 2:

$$\begin{aligned} &\mathrm{W}_{\mathtt{na}}(x,v) \mid \mathrm{R}_{\mathtt{na}}(x,v) \\ &\mid \mathrm{W}_s(x,v) \mid \mathrm{R}_s(x,v) \mid \mathrm{RMW}_s(x,v,v'). \end{aligned} \tag{1}$$

The events in an assembly execution are broadly similar to the OpenCL events, except that they omit the scope parameter s (since the abstraction of scopes does not exist at the assembly level) and include additional events to represent low-level memory events that are not exposed in OpenCL:

$$\begin{aligned} &\mathrm{W}(x,v) \mid \mathrm{R}(x,v) \mid \mathrm{RMW}(x,v,v') \\ &\mid \mathrm{FlushL1}_{\{\mathtt{WG},\mathtt{DV}\}} \mid \mathrm{InvalL1}_{\{\mathtt{WG},\mathtt{DV}\}} \\ &\mid \mathrm{LockRMW}_{\{\mathtt{WG},\mathtt{DV}\}} \mid \mathrm{UnlockRMW}_{\{\mathtt{WG},\mathtt{DV}\}}. \end{aligned} \tag{2}$$

Since our proposed compilation scheme does away with cacheline locking, we need not include events to represent $\mathtt{LK}_{\mathtt{L2}}$ and $\mathtt{UL}_{\mathtt{L2}}$ instructions.

Execution equivalence The $\cong$-relation of Thm. 1 captures a notion of equivalence between an OpenCL execution and an assembly execution. Simply put, $X \cong Y$ holds if X and Y are isomorphic graphs once the features that are specific to high-level or low-level executions – scope parameters and low-level memory events – are erased. We effectively reduce both X and Y to executions whose event labels are restricted to the common sub-language of (1) and (2):

$$\mathrm{W}(x,v) \mid \mathrm{R}(x,v) \mid \mathrm{RMW}(x,v,v').$$

6.2 Structure of the Proof

For a given program, the axiomatic OpenCL memory model acts on a set of pre-executions, $\mathcal{O}\langle P\rangle$, computed by a thread-local counterpart to the semantics. A faithful model of OpenCL's thread-local semantics would be an extensive work in itself, and is beyond the scope of this paper. Instead, we follow Batty [4] and simply require our thread-local semantics to only generate sets, $\mathcal{O}\langle P\rangle$, that exhibit *receptiveness*; given a pre-execution in $\mathcal{O}\langle P\rangle$ with a read event r, for every value v, there is another pre-execution in $\mathcal{O}\langle P\rangle$ where all $(rf \cup sb)^+$ predecessors of r are identical, but r reads the value v, and the rest of the execution may differ. This requirement is integral to the calculation of the semantics: if the candidates in $\mathcal{O}\langle P\rangle$ do not cover the range of values that might be read, then executions will be missed by the memory model. To complement this, we prove *completion*: that any consistent $rf \cup sb$ down-closed prefix of a pre-execution in $\mathcal{O}\langle P\rangle$ can be completed to a consistent execution of $\mathcal{O}[\![P]\!]$.

The theorem describes the behaviour of programs, but because we are working with an axiomatic memory model, the proof is most conveniently performed at the level of executions. As a consequence, we must lower some concepts to the level of executions, in particular the compilation mapping and the notion of consistency for assembly executions. We write $\mathit{comp}\,X\,Y$ when the assembly execution Y is a valid compilation of the OpenCL execution X, according to the mapping in Tab. 3 (right column). We write $\mathit{consistent}_{\mathcal{A}}$ when the assembly execution Y is permitted by the machine.

Now we present a soundness lemma over executions:

Lemma 1 (Execution soundness)**.** *For all OpenCL+RSP programs P:*

$$\begin{aligned} &\forall X_{pre} \in \mathcal{O}\langle P\rangle.\, \forall Y.\, \mathit{comp}\, X_{pre}\, Y \wedge \mathit{consistent}_{\mathcal{A}}\, Y \implies \\ &\quad \exists X.\, (\mathit{pre}\, X = X_{pre}) \wedge X \in \mathcal{O}[\![P]\!] \wedge X \cong Y \end{aligned}$$

where pre projects the pre-execution from a candidate execution, removing mo and rf.

We use this theorem to establish the program-level soundness theorem (Thm. 1) by constructing a Y that satisfies comp and $\mathit{consistent}_{\mathcal{A}}$ in such a way that the set of consistent executions of the compiled program, $\mathcal{A}[\![\mathit{compile}\,P]\!]$, is equal to the set of consistent assembly executions that comp matches with the pre-executions of the program.

In establishing execution soundness, we must be able to prove that a given OpenCL pre-execution can be extended to a consistent execution. To do so, we must find a modification order (mo) and a reads-from relation (rf) that together satisfy the various consistency axioms. To this end, we further augment the semantics of assembly programs so that they construct witnesses for these relations as they execute.

Auxiliary state We augment machine states with two extra components. We emphasise that these components are auxiliary (sometimes called 'ghost'): they do not affect the operation of the machine, existing only to accumulate useful information for the proof.

We modify the types of cache entries and global memory so as to accompany each value with both an event identifier and a list of event identifiers. That is:

$$\begin{aligned} \mathit{Aux} &= (\mathrm{rfsource}\colon \mathit{EventId}) \times (\mathrm{pmo}\colon \mathit{EventId}\ \mathrm{list}) \\ \mathit{CacheEntry} &= \mathit{Val} \times \mathit{Aux} \times (\mathrm{hy}\colon \mathit{Hygiene}) \times (\mathrm{fr}\colon \mathit{Freshness}) \\ \mathit{Global} &= \mathit{Loc} \rightharpoonup (\mathit{Val} \times \mathit{Aux}). \end{aligned}$$

The new $\mathrm{rfsource}$ component identifies the event responsible for writing the currently-held value. Whenever a location is written to or flushed, this component is overwritten appropriately. If the location is subsequently loaded, an rf-edge is drawn from this event to the loading event.

The new pmo component (for 'partial modification order') identifies the sequence of events that have written to this cache entry since it was last flushed. In the case of global memory, which is never flushed, this component records a

history of all the events that have ever written to this location. Whenever a cache entry is fetched, the newly fetched entry has its pmo component initialised to the empty list. Whenever an entry is flushed, the contents of its pmo list is appended to the end of the pmo list in the deeper cache entry or global memory location, and its pmo list is then cleared. When an assembly program completes its execution and all the caches have been flushed down to global memory, then a witness for the mo relation can be read off by taking the union of all the pmo components in global memory.

Consistency It remains to show either that the pre-execution, combined with the mo and rf relations generated by the abstract machine, is consistent, or that the program was faulty, in which case every execution is a member of $\mathcal{O}[\![P]\!]$.

If the program is racy, then the mo and rf relations may be inconsistent, and these executions must be treated differently. Consequently, we split the proof into two cases. In one case, every rf edge on a non-atomic location is coincident with an hb edge, and there are no increment accesses that race with badly scoped writes. In the other case, there is at least one rf edge that is not coincident with a read, or there is a write that races with an increment that follows it in mo.

We prove the first case, omitting the assumption that X_{pre} is a pre-execution of the program P, and then rely on this in the proof of the second.

Case 1: hb ***covers non-atomic*** rf Five consistency axioms must be established (Fig. 2): Hb, Coh, Rf, Narf, and Rmw. For each axiom, we relate the forbidden shape to an assembly execution, and consider the sequences of operations that the abstract machine might perform to generate that execution. We collect intermediate lemmas that relate the OpenCL+RSP candidate execution to the states of the abstract machine.

More precisely, we build a machine execution version of the happens-before relation, and we show that events related by hb at the language level correspond to machine-execution events related by the new relation. We then establish that when a write or a read precedes an access of the same location in machine happens-before, then this write, or the write that is read, in the case of a read, is sure to have propagated through the memory hierarchy of the other thread prior to its access. We use this property to establish the axioms of the language model.

Take for example the axiom Rf. To prove this, we establish that given an rf edge from a write, w, to a read, r, the write must have propagated to the reading thread prior to the read, and then, because of the hb edge, the write must also have propagated to the writing thread before the write, a contradiction. Then the cycle forbidden by Rf is in fact forbidden by the compilation scheme over the abstract machine.

This part of the proof is reminiscent of a previous compilation-scheme proof for C++11 above the Power architecture [21], so we elide the details for brevity.

Case 2: there is a racy non-atomic read, or increment In this case, there is either an rf or mo edge that constitutes a race, but the execution, X, may not be consistent. We will show that the program is in fact racy by constructing a consistent execution with a race. First, find all of the offending edges in X, and find the edge amongst these whose tail access is added in the earliest reduction step of the abstract machine, and name its head w and its tail r. Now identify the set of events in X that precede w and r in $(rf \cup sb)^+$. Restrict the executions X and Y to these events and w, producing smaller executions X' and Y', adding steps of the abstract machine at the end of Y that flush the caches and complete the execution.

The new execution X' does not contain racy rf or mo edges of the sort we identified, so we can apply case 1 above to establish consistency of X'. Now, observing that X' contains all $(rf \cup sb)^+$ predecessors of r in X, we use the assumption of receptiveness to add the read r, having it read from a visible write if it is at a non-atomic location, or the final write in mo if it is an increment. This new execution is consistent but racy. We then apply the completion lemma to extend the execution to a complete consistent execution of P with a race, X_{fault}, as required.

7. Related Work

There has been a large and sustained effort to understand both the behaviour of relaxed memory hardware and the concurrent language interface that ought to be presented to programmers. The behaviour of relaxed CPUs is relatively well understood, through formal models of x86 and Power processors that have been validated through testing [3, 23] and communication with architects [16, 22]. This line of work has unearthed bugs in deployed CPUs [2], and unexpected behaviours in GPUs that break programming idioms taught in vendor-endorsed documentation [1]. Our approach differs from this prior work in that our industrial–academic collaboration is taking place at the research stage, following publicly-released descriptions of the architecture [11, 19], so the bugs we have found can be rectified before the hardware is deployed.

The PipeCheck tool focuses, as we do, on the correctness of a microarchitectural implementation [15]. It measures correctness against an *architectural* specification, whereas we compare directly to the *programming language* specification. PipeCheck accommodates more microarchitectural features than those we model in this work, such as instruction reordering. However, it requires microarchitectures to be specified *axiomatically*, and hence is not directly compatible with the *operational* model that we use in this work.

Formal modelling of language-level memory models has led to the discovery of bugs, particularly in the specification of the C/C++11 memory model [6]. With a formal model of concurrency in the underlying hardware and the programming language, it is possible to prove the correctness of compiler

mappings for concurrency primitives, and this has been done for mappings of C/C++11 to x86 [6] and Power [5, 21] processors. Our work represents the first correctness proof of compiler mappings targeting a concrete GPU architecture.

As the understanding of concurrency in existing hardware and languages has matured, other work has developed new approaches that expose alternative behaviours to the programmer. Rajaram et al. propose a weaker semantics for RMW accesses in the total store order (TSO) memory model [20], arguing that performance can be improved without affecting the language memory model. Another theme is to offer the programmer more control of the locality of synchronisation by exposing scoped accesses [10, 12, 14, 24]. Our work presents a new mechanised semantics for remote-scoped accesses, incorporated in to a model of OpenCL 2.0 concurrency. As well as probing the underlying architectural definition, this validates the new features in the programming model, establishing that they are efficiently implementable.

8. Conclusion

By formalising remote-scope promotion at both the language level and for a low-level implementation, we have demonstrated the value of applying formal techniques early when designing hardware support for a new concurrency feature. Our axiomatic memory model, and the corresponding extensions to HERD, allowed us to find bugs in litmus tests and a work-stealing queue implementation that were designed to illustrate the semantics and benefits of RSP. Our formalisation of the proposed implementation of RSP revealed other issues in the design. By proving that our remedied implementation is correct with respect to the memory model, we increase confidence in full-blown GPU designs based on this implementation proposal.

Our remedied RSP implementation is potentially more efficient than the original proposal because it does not require cacheline locking. In future work, we plan to assess this hypothesis through architectural simulation. We plan also to extend the compilation scheme to a multiple-device setting. Longer term, we plan to formalise more complex hardware designs. One promising target is AMD's QuickRelease technology [11], parts of which already appear in the design of RSP. The full QuickRelease scheme, which involves separate reading and writing caches, and cacheline-specific L1 invalidations, involves greater hardware complexity but promises superior performance. Another topic for future research is the extension of OpenCL verification tools, such as GPUVerify [7], to handle remote-scope promotion.

Acknowledgements AMD is a trademark of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple Inc. used by permission by Khronos. We gratefully acknowledge support from the EU FP7 CARP Project (project number 287767), the EPSRC (projects EP/K011499, EP/I020357/1, EP/K015168/1, and EP/I01236/1), a Pathways to Impact project supported by the Imperial EPSRC Impact Acceleration Account. We thank John Alsop, Jeroen Ketema, Mark Hill, Yatin Manerkar, Marc Orr, and David Wood for useful discussions and feedback on this work.

References

[1] 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.

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

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

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

[5] 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.

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

[7] 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. *ACM Trans. Program. Lang. Syst.*, 37(3):10, 2015.

[8] D. Cederman and P. Tsigas. Dynamic load balancing using work-stealing. In *GPU Computing Gems*. Elsevier, 2012.

[9] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding irregular GPGPU graph applications. In *IISWC*, 2013.

[10] B. R. Gaster, D. R. Hower, and L. Howes. HRF-Relaxed: Adapting HRF to the complexities of industrial heterogeneous memory models. *ACM TACO*, 2015.

[11] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. QuickRelease: A throughput-oriented approach to release consistency on GPUs. In *HPCA*, 2014.

[12] D. R. Hower, B. M. Beckmann, B. R. Gaster, B. A. Hechtman, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous-race-free memory models. In *ASPLOS*, 2014.

[13] G. Kyriazis. Heterogeneous system architecture: A technical review. Technical report, AMD, 2012.

[14] C. Lin, V. Nagarajan, and R. Gupta. Fence scoping. In *SC*, 2014.

[15] D. Lustig, M. Pellauer, and M. Martonosi. PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models. In *MICRO*, 2014.

[16] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *CAV*, 2012.

[17] A. Munshi. *The OpenCL Specification (Version 2.0)*. Khronos OpenCL Working Group, November 2013.

[18] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[19] 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.

[20] B. Rajaram, V. Nagarajan, S. Sarkar, and M. Elver. Fast RMWs for TSO: Semantics and implementation. In *PLDI*, 2013.

[21] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *PLDI*, 2012.

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

[23] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *CACM*, 53(7), 2010.

[24] J. Wickerson, M. Batty, and A. F. Donaldson. Overhauling SC atomics in C11 and OpenCL. *CoRR*, July 2015. http://arxiv.org/abs/1503.07073.