# HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models

BENEDICT R. GASTER, DEREK HOWER, and LEE HOWES, Qualcomm

Memory consistency models, or memory models, allow both programmers and program language implementers to reason about concurrent accesses to one or more memory locations. Memory model specifications balance the often conflicting needs for precise semantics, implementation flexibility, and ease of understanding. Toward that end, popular programming languages like Java, C, and C++ have adopted memory models built on the conceptual foundation of Sequential Consistency for Data-Race-Free programs (SC for DRF). These SC for DRF languages were created with general-purpose homogeneous CPU systems in mind, and all assume a single, global memory address space. Such a uniform address space is usually power and performance prohibitive in heterogeneous Systems on Chips (SoCs), and for that reason most heterogeneous languages have adopted split address spaces and operations with nonglobal visibility.

There have recently been two attempts to bridge the disconnect between the CPU-centric assumptions of the SC for DRF framework and the realities of heterogeneous SoC architectures. Hower et al. proposed a class of Heterogeneous-Race-Free (HRF) memory models that provide a foundation for understanding many of the issues in heterogeneous memory models. At the same time, the Khronos Group developed the OpenCL 2.0 memory model that builds on the C++ memory model. The OpenCL 2.0 model includes features not addressed by HRF: primarily support for relaxed atomics and a property referred to as scope inclusion. In this article, we generalize HRF to allow formalization of and reasoning about more complicated models using OpenCL 2.0 as a point of reference. With that generalization, we (1) make the OpenCL 2.0 memory model more accessible by introducing a platform for feature comparisons to other models, (2) consider a number of shortcomings in the current OpenCL 2.0 model, and (3) propose changes that could be adopted by future OpenCL 2.0 revisions or by other, related, models.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: Hardware/Software Interfaces, Systems Specifications Methodology; D.3 [**Programming Languages**]; D.3.1 [**Formal Definitions and Theory**]: Semantics

General Terms: Theory, Languages

Additional Key Words and Phrases: Memory models, programming languages, computer architecture, formal models

## 1. INTRODUCTION

A memory (consistency) model specifies how individual memory operations can be ordered relative to one another, giving both system users and implementers the ability

Authors' addresses: B. R. Gaster, University of West Of England; email: benedict.gaster@uwe.ac.uk; D. Hower, 8041 Arco Corporate Drive, Raleigh, NC 27617; email: dhower@qti.qualcomm.com; L. Howes, 3165 Kifer Rd, Santa Clara, CA 95051; email: lhowes@qti.qualcomm.com.

to reason about concurrent accesses to one or more memory locations. Memory model specifications exist at both low-level (e.g., ISA) and high-level (e.g., general-purpose programming language) interfaces, and balance the often conflicting needs for precise semantics, implementation flexibility, and ease of understanding.

Sequential Consistency (SC) is an intuitive model that in effect states a program will execute as if each operation were completed atomically and one at a time [Lamport 1979]. SC is easy to reason about but unfortunately prohibits a large number of important implementation optimizations.

For that reason, programming languages for homogeneous CPU systems have started to converge on memory model specifications that belong to a class called Sequential Consistency for Data-Race-Free programs (SC for DRF) [Adve and Hill 1990]. These languages, including Java [Oracle 2014], C, and C++ [ISO. International Organization for Standardization 2011], have chosen the SC for DRF framework because it is easy for most programmers to understand yet allows many implementation optimizations. SC for DRF specifications, guarantee an SC execution, *but only if all concurrent accesses to shared memory are protected by synchronization such as a mutex*. SC for DRF implementations have considerable flexibility because if, in some DRF program, it is impossible to determine the order that two memory operations occur (e.g., because they are independent), then an implementation has the flexibility to reorder those operations. For more on SC for DRF, see Section 2.1.

While SC for DRF has proven to be a good framework for traditional CPU systems, it has limitations in platforms like mobile Systems on Chips (SoCs) that contain heterogeneous components with fine-grained shared memory. In SC for DRF, races are defined under the assumptions that all synchronization operations (e.g., C++ atomics) complete in a total, global order and that synchronization operations have globally observable side effects. This assumption is reasonable in systems with conventional snooping or directory coherence, but is difficult to enforce when, for example, coherence is maintained through heavyweight cache maintenance operations (as is done in many GPUs [Hechtman et al. 2014]). For that reason, existing heterogeneous platforms provide *scoped synchronization* operations that have nonglobal side effects. For example, OpenCL has a barrier operation that only guarantees visibility among work items (equivalent to CPU threads for memory model purposes) in the same work group (a cluster of work items sharing physical resources).

Recently, Hower et al. proposed a class of memory models called Sequential Consistency for Heterogeneous-Race-Free (SC for HRF) [Hower et al. 2014] that merge SC for DRF with scoped synchronization operations. They introduced the concept of a *heterogeneous race*, which can occur when concurrent accesses are protected with synchronization of "insufficient" visibility. They identified several options for defining sufficiency, and discuss the usability and implementation trade-offs of each choice.

While the original SC for HRF models do an excellent job of taming the complexity of scoped synchronization for a simplified system model, real heterogeneous languages must deal with additional complications that can make it hard to apply the insights of SC for HRF. For example, OpenCL, which at its core is also a race-free memory model, supports well-defined but non-SC executions, disjoint address spaces, and limited observability of some memory locations. In contrast, SC for HRF assumes all well-defined executions are sequentially consistent and a system model with a single, flat address space.

In this article we show how to add four new features to the HRF framework that together allow us to fully specify a real model like OpenCL. We also show how to restrict a program so that users of systems with complex models can revert to the original, and far simpler, SC for HRF models.

In particular, we show how to add *scope inclusion*, *relaxed atomics*, *observability*, and *multiple address spaces* to the HRF framework. Scope inclusion permits more

well-defined programs at essentially zero implementation cost. It was a property known in the original SC for HRF work but excluded due to the complexity it adds to the formalization. Relaxed atomics permit well-defined but non-SC executions. They exist in languages like OpenCL and C++, and are exceedingly difficult to understand. Observability is necessary because systems like OpenCL allow what is called coarse-grain allocations in which a location mapped into a global address space may only be observable by a subset of the agents (e.g., to represent a buffer allocated into memory that is not coherent at the full-system level). Finally, we add multiple address spaces to account for the fact that some locations in heterogeneous systems exist in an entirely separate address space. For example, in OpenCL, the local memory region that represents a scratchpad cache is an entirely different address space from the global memory region that represents coherent caches. However, OpenCL has operations that can synchronize the two address spaces.

In summary, we make the following contributions to the state of the art:

—**HRF-∗-relaxed**: We extend the definition of HRF to support the complications of a real heterogeneous platform. This includes support for a property called scope inclusion, relaxed synchronization that could result in non-SC executions, a notion of location observability, and multiple address spaces (Sections 4 and 5).
—**Equivalence to SC for HRF**: We show how to constrain programs so that they result in an SC execution. This allows the majority of users to ignore the significant complexity discussed in this article and instead reason in terms of SC for HRF (Section 4).
—**Describe and Clarify OpenCL**: We describe the OpenCL memory model using the HRF framework. In doing so, we also clarify several features of the OpenCL 2.0 memory model that are handled informally in the specification (Section 6).
—**Propose OpenCL Extensions**: After describing OpenCL in terms of HRF, we use the HRF insights to demonstrate (1) that OpenCL is overdesigned with respect to address spaces and could benefit from a simplified model, and (2) that OpenCL could easily permit more well-defined sequentially consistent executions (Section 7).

To our knowledge this work is the first to extend HRF to handle the complexities of an industrial heterogeneous memory model, in general, and the first to provide an alternative formalization of OpenCL 2.0's memory model, in particular.

## 2. BACKGROUND

In this section we summarize the work on DRF and HRF memory models, as well as some of the concepts implemented in existing heterogeneous memory models, particularly OpenCL.

### 2.1. DRF Memory Models

In 1990, Adve and Hill [1990] defined a class of memory consistency models collectively termed SC for DRF. These models enable high-performance implementations, have precise semantics, and are relatively simple to understand. SC for DRF models describe rules that programs must follow in order to avoid data races. In the absence of races, an SC for DRF implementation will guarantee an SC execution. In Adve and Hill's original formulation, the system provides no guarantees when a program contains a data race, though subsequent work has developed specifications that still provide basic ordering guarantees such as write causality [Oracle 2014] or that raise an exception when a non-SC execution occurs [Marino et al. 2010; Lucia et al. 2010].

Informally, a data race is usually understood to mean any two ordinary (for C++ that means nonatomic) memory accesses that are unprotected by synchronization and could therefore occur "at the same time" [Boehm and Adve 2008]. An SC for DRF model guarantees that any execution of a DRF program will *appear* sequentially

```
Thread t1:
  101:  X = 1;
  102:  Y = 2;
  103:  atomic_store(&F, 100);              // synchronization store
Thread t2:
  201:  while (atomic_load(&F) != 100); // synchronization load
  202:  R1 = X;
  203:  R2 = Y;
```

Fig. 1. Independent operations 101 and 102 can be reordered in SC for DRF. Initially $X = Y = F = 0$.

consistent—though is under no obligation to ensure that the *actual* memory access completion order, if it could be observed, is sequentially consistent.

Many memory operations in a program are independent from each other, and an SC for DRF implementation is free to reorder any such independent accesses. In Figure 1 an implementation can perform the accesses on lines 101 and 102 in any order as long as they complete before the accesses on lines 202 and 203, respectively. In the example, it would be impossible for thread t2 to determine the order that 101 and 102 complete without introducing a data race, and therefore there is no valid execution that could determine the global completion order of accesses 101 and 102. Rather, all we can say is that line 101 "happens before" line 202 and that line 102 "happens before" line 203. This example highlights the fact that SC for DRF is a relaxed model—implementations must maintain the *appearance* of SC but are not obligated to produce an *actual* sequentially consistent order of memory accesses.

## 2.2. Heterogeneous Execution Models

To understand why conventional SC for DRF memory models are insufficient for heterogeneous platforms, we need a basic understanding of heterogeneous execution and memory models. In this section, we describe a heterogeneous platform using OpenCL terminology, though other platforms like HSA [HSA Foundation 2012] and CUDA [NVIDIA Corporation 2013] have similar organizations.

OpenCL exposes a hierarchical execution environment to reflect the fact that some actors in a platform have a special relationship to others. An OpenCL application is composed of a number of concurrent actors, including host threads that execute on a conventional CPU and device work items that can execute on a variety of attached devices from GPUs to FPGAs. As shown in Figure 2(a), work items belong to several groups that capture locality and visibility relations with respect to other actors in the system. First, all host threads and device work items belong to the single *system* group. All work items in a single *NDRange* execute on the same device. The device itself may comprise several compute units (similar to a CPU core). Work items belonging to the same *work group* execute together on a single compute unit. Finally, work items in a *subgroup* may execute together as part of a Single Instruction, Multiple Data (SIMD) vector (e.g., when running on a GPU). Separate subgroups must execute independently: in effect, a subgroup is an abstraction of a hardware thread. It is useful to expose these groups at the language level because locality is critical in most heterogeneous devices. For example, to achieve good performance on a GPU, an OpenCL application should ensure that there is little divergence, either in control flow or memory accesses, among work items in a subgroup (as these may execute in lock-step in a hardware vector unit).

## 2.3. Scopes

Since their inception, heterogeneous platforms like OpenCL have incorporated the execution hierarchy into the memory model to reflect the fact that communication costs vary depending on the actors involved. For example, in a typical GPU implementation, shown in Figure 2(b), that synchronizes by using heavyweight cache maintenance

(a) The OpenCL execution hierarchy          (b) Example GPU cache organization.
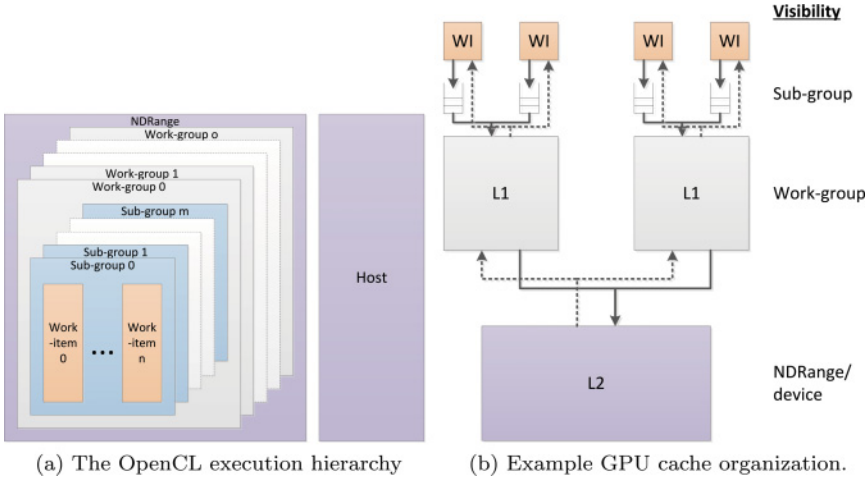
Fig. 2. OpenCL execution hierarchy and example mapping to a GPU cache organization. Consistency may be maintained at programmer-defined synchronization points by flushing/invalidating caches to the visibility specified by the scope of the operation.

operations like flush/invalidate [Hechtman et al. 2014], it is much less costly to synchronize among work items in a work group that share an L1 cache than it is among work items in an NDRange that only share an L2 cache, and both are more efficient than synchronizing with the host through an L3 cache or DRAM.

To keep the discussion concrete, we describe the OpenCL 2.0 notion of scopes here. Scope definitions are similar in other languages. Informally, a scope is a subset of actors (e.g., work items in a single work group), and a scoped synchronization operation only affects other actors within the same subset. In OpenCL, each atomic operation or fence is performed with respect to a single scope that can be specified by a programmer. OpenCL 2.0 defines five scopes that correspond to the execution hierarchy shown in Figure 2(a):

(1) memory_scope_work_item
(2) memory_scope_sub_group
(3) memory_scope_work_group
(4) memory_scope_device
(5) memory_scope_all_svm_devices

For simplicity, in the rest of the article we often abbreviate a scope name as $ms\_wi$, $ms\_sg$, $ms\_wg$, $ms\_dev$, and $ms\_svm$, respectively. The $ms\_svm$ scope corresponds to shared virtual memory, and includes all actors (work items and host threads) in an OpenCL execution.[1]

When dealing with scopes, it is useful to distinguish the static name for a scope from the dynamic group of actors that correspond to a scope at runtime.

*Definition* 2.1 (*Static Scope*).   The scope named in the program text. For example, the static scope of the OpenCL atomic operation *atomic_load_explicit*(&*A*, . . . , *memory_scope_work_group*) is work group, or $ms\_wg$.

*Definition* 2.2 (*Dynamic Scope*).   The set of agents in the hierarchy at a given scope. For example, the dynamic scope of the OpenCL atomic operation

---

[1]With some caveats to allow support for devices without coherent memory as we will see in Section 5.

```
Device D1
   Work group X
      Work item X1
         101:    T = 1;
         102:    A.store(1, memory_scope_work_group);
      Work item X2
         201:    while (!A.load(memory_scope_work_group));
         202:    R2 = T;
         203:    B.store(1, memory_scope_system);
Device D2
   Work group Y
      Work item Y1
         301:    while (!B.load(memory_scope_system));
         302:    R3 = T;
```

Fig. 3. This program contains a race (between lines 101 and 302) in HRF-direct, but is heterogeneous-race-free in HRF-indirect. Note that we use simplified OpenCL syntax in this example for consistency throughout the article. *T*, *A*, and *B* are all initialized with the value 0.

*atomic_load_explicit*(&*A*, . . . , *memory_scope_work_group*), when executed by work item *WI*, is the set of work items {*WI*, *WI′*, . . .} that execute together in a particular work group *WG*.

Given two OpenCL atomic operations, it is possible for them to share the same static scope but have different dynamic scope, for example, two work items in different work groups, each executing an atomic with static scope work group. Two operations with identical dynamic scope will always have the same static scope.

We say that two static scopes are equivalent, written $S1 ==_{sms} S2$, if their syntactic scopes are the same. We say two dynamic scopes are equivalent, written $S1 ==_{dms} S2$, if they correspond to exactly the same set of actors.

### 2.4. HRF Memory Models

When a program contains synchronization operations that are performed with respect to different scopes, it introduces the possibility of what Hower et al. [2014] call a *heterogeneous race*. A heterogeneous race occurs when a program correctly protects two ordinary accesses with synchronizing atomics such that they cannot occur simultaneously (i.e., they are intuitively DRF) but fails to use sufficient dynamic scope to guarantee visibility. They showed that a heterogeneous race, like a normal race, can lead to undefined behavior in a heterogeneous system. To help describe the types of heterogeneous races that can occur, they introduced the class of memory models called SC for HRF.

Hower et al. have defined two alternative SC for HRF memory models that trade off implementation flexibility and program expressiveness. The first, called *HRF-direct*, requires that both the source (e.g., producer) and destination (e.g., consumer) actors synchronize with respect to the same dynamic scope whenever communicating through shared memory. HRF-direct allows aggressive system optimizations and appears to be a safe abstraction of many existing heterogeneous systems. Alternatively, the authors also proposed *HRF-indirect* that allows two communicating actors to synchronize with inexact dynamic scope when there is a transitive chain of synchronization through a third actor. HRF-indirect reduces the allowable hardware optimizations but may permit higher-performing software on today's heterogeneous implementations.

In Figure 3 we show an example of a program that contains a race in HRF-direct but is HRF in HRF-indirect. In HRF-direct, the store at line 101 races with the load at

```
Device D1
  Work group X
    Work item X1
       101:   T = 1;
       102:   A.store(1, ..., memory_scope_device);
    Work item X2
       201:   while (!A.load(..., memory_scope_work_group));
       202:   R2 = T;
```

Fig. 4. Example of correct synchronization in a model with scope inclusion support. Work items X1 and X2 can directly synchronize with each other using different scopes because one scope is a subset of the other. *T* and the atomic variable *A* are both initialized to 0.

line 302 because work items X1 and Y1 have not synchronized in the same scope. In HRF-indirect, the program is HRF because work item X2 forms a transitive synchronization link between work items X1 and Y1.

To implement HRF-indirect, a system must ensure that the visible side effects of synchronization operations include prior accesses from all actors in the scope of the operation, not just from the actor performing the synchronization. To the best of our knowledge, all current CPU/GPU heterogeneous systems meet this requirement, but the restriction may prevent future optimizations. For example, in a GPU that synchronizes by flushing dirty data from a cache (e.g., at line 203 in Figure 3), HRF-indirect would prevent an optimization where only the data modified by the actor requesting synchronization is made visible to other actors. More details on the differences between the two models are discussed in Hower et al. [2014].

## 3. NEW FEATURES

In this section we describe the four HRF feature additions that are required to describe the complexities of industrial models. The first, called scope inclusion, allows synchronization operations performed with different scopes to directly pair without forming a heterogeneous race. The second is support for relaxed, non-SC atomics that are available in both C/C++ and OpenCL 2.0. The third adds the ability to describe coarse-grain memory regions with limited observability, and the fourth adds support for multiple address spaces in the same platform image. In the remainder of this section we discuss the motivation and concepts behind each feature.

### 3.1. Scope Inclusion

Figure 4 shows an example of an OpenCL-like program that synchronizes using operations performed with respect to different scopes. The dynamic scope of the operation on line 102 (device *D*) *includes* the dynamic scope of the operation on line 201 (work-group *X*). Intuitively, one might expect this example to work as intended, such that the value of *R2* is guaranteed to be 1, because the release to device scope *D* is in effect a communication to all of the actors contained in the work-group scope *X*, plus more. However, Figure 4 is a race in both *HRF-direct* and *HRF-indirect*.

*Definition* 3.1 (*Scope Inclusion*). For now, let us say that two scoped synchronization operations, $O_s$ and $O'_{s'}$, have dynamic scopes $S$ and $S'$ and are executed by agents $A$ and $A'$, respectively. The operations are inclusive, written $O_s \approx_{incl} O'_{s'}$, if $S$ contains both $A$ and $A'$, $S'$ contains both $A$ and $A'$, and either the dynamic scope $S$ of $O_s$ is a subset of the dynamic scope $S'$ of $O'_{s'}$ or vice versa.

As Hower et al. have previously observed, reasonable implementations of both *HRF-direct* or *HRF-indirect* will likely ensure that Figure 4 works as expected, and thus we expect the implementation cost of scope inclusion is negligible. To see why,

```
Device D1
    Work group X
        Work item X1
            101:    A.store(1, ..., memory_scope_work_group);
            102:    R1 = B.load(..., memory_scope_device);
        Work item X2
            201:    B.store(1, ..., memory_scope_device);
            202:    R2 = A.load(..., memory_scope_work_group);
```

Fig. 5. A race-free program in both *HRF-direct* and *HRF-indirect*. Implementation must ensure there is a total observable order of all operations, even though they are performed with respect to different scopes. Atomic variables *A* and *B* are both initialized to 0.

consider the example in Figure 5 that is race-free in both *HRF-direct* or *HRF-indirect*. The program is race-free, so an implementation must guarantee a sequentially consistent execution: a total order of all operations. Notably, this means that an implementation must establish an order between the atomic loads and stores even though they are performed with respect to different, inclusive scopes. It would be exceptionally difficult for an implementation to dynamically distinguish the difference between the programs in Figures 4 and 5, leading us to believe the implementation cost of scope inclusion is negligible.

Aside from being trivial to implement, scope inclusion could also lead to more composable functions. Without scope inclusion, a library function cannot, in general, concurrently modify a data structure because the callers may use a different scope of synchronization on the same data structure. With scope inclusion, a library can safely use the largest possible scope (usually *ms_svm*) regardless of how the callers synchronize.

## 3.2. Relaxed Atomics

The implementation flexibility afforded by SC for DRF (and discussed in Section 2.1), and by extension SC for HRF, is good enough for the majority of programs and implementations. However, the pure models do have constraints that can unnecessarily degrade performance in specific cases. For that reason, some derivative languages like C/C++11 and OpenCL 2.0 support relaxation of the SC requirement in limited cases while still staying a core DRF model. These relaxations are exceptionally difficult to understand, and are intended to be used rarely and only by expert programmers [Boehm and Adve 2008].

In Figure 6, we show an example of when SC for DRF/HRF can be overly restrictive. In this example, a service thread periodically checks whether a client thread has requested a service by reading from an incoming message queue. If there are no messages, the service thread continues to do other, unrelated work involving only local data. Let us say that response time for the incoming request is critical, meaning that the periodic check must be frequent, but that requests are rare. In the common case the queue is empty, and for that reason a high-performance implementation might wish to avoid the overhead of synchronization (e.g., a low-level fence instruction) when checking the queue for an incoming request. In a strict SC for DRF/HRF model, there is no way to check the occupancy of the shared queue without using synchronization; any attempt to read the state of the queue using an ordinary load or store would form a data race with the requestor. Thus, the program may be unnecessarily slow on a system with high synchronization costs.

To support higher-performance programs, both C/C++11 and OpenCL 2.0 include what they call *low-level atomics*, which are atomic operations explicitly marked with an ordering property weaker than SC. Specifically, programmers can mark an atomic

```
001: struct Task;
002: struct MsgQueue {
003:    int _occupancy;
004:
005:    Task* dequeue() {
006:       if (atomic_load(&_occupancy) == 0) {
007:          return NULL;
008:       } else { ... }
009:    }
010:    ...
011: } globalQueue;

Thread t1:
   101: void periodicCheck() {
   102:    Task* t = globalQueue.dequeue();
   103:    if (t != NULL)
   104:       t.execute();
   105: }
```

Fig. 6.  A C++ program where SC for DRF unnecessarily prohibits some operation reordering.

```
001: struct Task;
002: struct MsgQueue {
003:    int _occupancy;
004:
005:    Task* dequeue() {
006:       if (atomic_load(&_occupancy) == 0) {
007:          return NULL;
008:       } else { ... }
009:    }
010:    int occupancy() {
011:       return atomic_load(&_occupancy, memory_order_relaxed);
012:    }
013:    ...
014: } globalQueue;

Thread t1:
   101: void periodicCheck() {
   102:    // goal: avoid global synchronization on occupancy check
   103:    if (globalQueue.occupancy() > 0) {
   104:       Task* t = globalQueue.dequeue();
   105:       if (t != NULL)
   106:          t.execute();
   107:    }
   108: }
```

Fig. 7.  An example showing how relaxed atomics can lead to better performing programs.

access as a release, an acquire, or as a relaxed operation.[2] An access marked as a release or an acquire has global ordering side effects on ordinary loads and stores but the atomic access itself does not have to be sequentially consistent relative to other atomic accesses. This is similar to how synchronization operations are treated in the well-known $RC_{PC}$ model [Gharachorloo et al. 1990]. Accesses with relaxed ordering, on the other hand, have no side effects on ordinary loads and stores and can likewise be reordered relative to other non-SC accesses.

Using relaxed atomics, the service thread in Figure 6 can avoid costly synchronization in the common case, as shown in Figure 7. In this example, the service thread

_____
[2]For simplicity, we treat consume ordering like release ordering in this article.

reads the occupancy of the shared queue using a relaxed atomic that will not produce any synchronization side effects. The thread will only perform costly synchronization (through the *dequeue* function on line 104) if it finds the queue has content. In this example, the implementation is under no obligation to ensure that the occupancy check is sequentially consistent with respect to the rest of the execution.

While relaxed atomics can be useful in limited circumstances, we reiterate that relaxed atomics are quite complex and error prone. Their inclusion in C/C++ was controversial, and their use is generally discouraged [Boehm and Adve 2008; Sutter 2012]. The pitfalls of relaxed atomic complexity are not limited to C++ users and implementers; the C++11 standard [ISO. International Organization for Standardization 2011] has a known issue with the formulation of relaxed atomics such that while the intent was clear to readers, in an effort to avoid out-of-thin-air values, the committee inadvertently added text that required relaxed atomics to behave as if they were SC. This was a far stronger change than was intended [Boehm and Demsky 2014].

Despite the challenges and shortcomings of relaxed atomics [Adve 2010], we address them here because the OpenCL standards committee has included them in the OpenCL 2.0 specification.

### 3.3. Observability

Heterogeneous models separate memory into regions that are shared between discrete devices at *coarse-grain* synchronization points. For example, OpenCL provides coarse-grain buffers that are allocated with Application Programing Interface (API) calls and that only become visible at coarse-grain synchronization points that map or unmap the region. Notably, the visibility of memory in a coarse-grain buffer is not generally affected beyond a given level by fine-grain synchronization such as atomics.

Coarse-grain regions complicate an HRF model because they set a strict limit on the observability of a location within the region. For example, in OpenCL, a coarse-grain buffer allocated to a particular device will never be visible to an agent outside of that device, even if a work item on the device synchronizes globally. OpenCL has this feature in order to allow the runtime to use any device-specific physical memory, such as noncoherent GPU DRAM.

We incorporate observability into an HRF model in Section 5.

### 3.4. Multiple Address Spaces

In addition to coarse-grain regions within a single address space, heterogeneous platforms may also provide some memory regions with an entirely different address space. To complicate things further, these address spaces may have different ordering rules. In the case of OpenCL, the *local* and *global* address space orders are almost entirely separate, as is evident from the separate local and global flags that can be passed to OpenCL fences.

Address spaces play a fundamental role in describing data locality. They allow developers to explicitly manage where data lives in the memory hierarchy during program execution. For example, OpenCL's local memory generally corresponds to a physical scratchpad memory, which is why it is modeled as an independent address space from coherent shared memory and that is only visible to a subset of agents. Because synchronization on local memory does not affect global memory, implementations can keep local memory operations fast (e.g., an implementation does not need to flush caches on a local memory release).

We show how to incorporate multiple address spaces into an HRF model in Section 5.2.

## 4. HETEROGENEOUS-RACE-FREE-RELAXED (HRF-*-RELAXED)

In this section we formalize the extensions described in Section 3 into a fully specified HRF model. Here we assume the basic notion of scope inclusion described in Definition 3.1, though show later that the model can also support more restricted rules for scope inclusion, such as those in OpenCL 2.0, in Section 6.3.

Like Hower et al. [2014] before us, we propose two versions of our relaxed HRF models that differ only in whether or not they support transitive synchronization involving different scopes. We will present the model for the nontransitive variant, called *HRF-direct-relaxed*, first and then will show the necessary change to support scope transitivity in Section 4.6.

The formal definition of HRF-direct-relaxed in Figure 9 is considerably more complex than its predecessor HRF-direct, due mostly to to the fact that it does not start with the same *a priori* assumption that all candidate executions are sequentially consistent. Luckily, if a program only uses *mo_sc* atomics (the default in OpenCL) and only synchronizes with pairs of atomics using the exact same scope (the only synchronization currently defined by OpenCL), then the *HRF-direct-relaxed* model is equivalent to the simpler *HRF-direct* model developed in Hower et al. [2014]. Thus, the majority of users do not need to concern themselves with the complexities of *HRF-direct-relaxed*. We prove the equivalence of the two models in supplementary material [Gaster et al. 2015].

### 4.1. Model Structure

*System Model*. We define the model for an abstract system consisting of a collection of disjoint memory locations. Loads (stores) read (write) a value from (to) a single location. For simplicity, assume for now that all loads and stores are aligned to their natural width and that there are no overlapping loads or stores of different widths. Some loads and stores are marked as atomic, and all atomic operations are qualified with a specific ordering (e.g., *mo_sc*) and scope (e.g., *ms_wg*). A thread of execution is a set of operations, including loads and stores, performed by a single agent (host thread or work item). Given the values returned from memory, a thread of execution must respect the control flow semantics of the static program. We call the order of operations that a thread of execution performs *program order* and assume that it is a total order.[3]

*Logical Flow*. In Figure 8 we show how to use an HRF-relaxed model to determine the possible executions of a program. Given a static program, a user will first construct a set of all *plausible executions* that result when each load observes either the initial value of a location or the value of some other store in the execution to the same location. Many plausible executions will eventually be discarded from consideration because they violate the rules of the model. Next, the set of plausible executions are reduced to a set of *candidate executions* that respect the apparent orders and rules of the HRF model, for example, those listed in Section IV of Figure 9. A program will result in an undefined execution if any candidate execution contains a heterogeneous race, for example, as defined in Section V of Figure 9. Otherwise, a user can precisely determine the set of possible executions:

> If all candidate executions are race-free, then a conforming implementation must produce one of the candidate executions.

*Notes*. The reader should be careful not to interpret the rules regarding candidate executions in Section IV as rules that are *always* strictly enforced by an *HRF-direct-relaxed*

---

[3]Some languages have an undefined evaluation order in certain situations such that program order may be a partial order. We omit this complication from the model since it is well known how to handle it and would only distract from the main contributions.
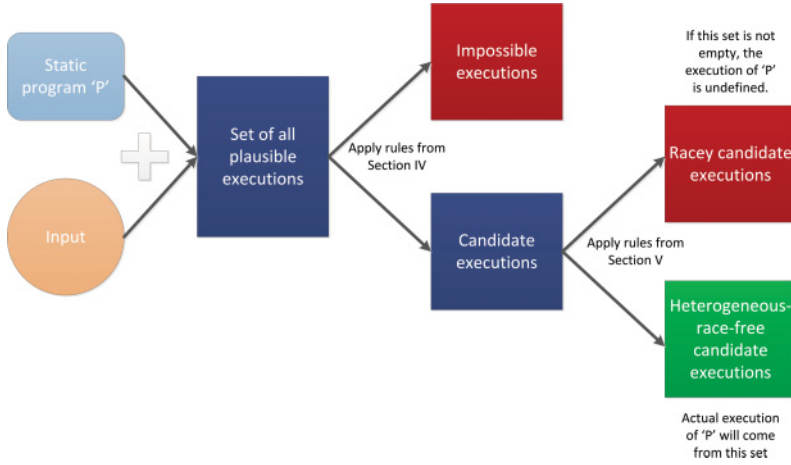
Fig. 8.   Logical structure of the HRF-relaxed models.

implementation. For example, while it is correct to say that there is an apparent total order of *mo_sc* atomic operations, regardless of scope, in the execution of a HRF program (the $\vec{sc}$ order in Figure 9), an implementation is under no obligation to provide a total order of all *mo_sc* atomics that are executed by software. If the executing program contains a heterogeneous race, then the implementation can execute correctly in a fashion that would break the total order because we do not attempt to define that execution. Also note that the orders in Section IV are apparent orders, and are not necessarily a strict indication of the order in which an implementation must complete operations even if an execution is HRF. Given the values observed during an execution, it may be impossible to determine the actual order in which some operations completed. For example, the operations may be independent. When constructing the apparent orders for purposes of the model, these independent operations are put in an arbitrary order relative to one another. For example, if a HRF execution contains two *mo_sc* atomic accesses with the same static work-group scope but different dynamic scope (because they are performed in different work groups), an implementation does not need to ensure that those two atomics are serialized because an agent cannot observe the actual completion order without introducing a heterogeneous race.

In Section 4.5 we show how a system can take advantage of these observations to implement performance optimizations, especially in relation to scoped operations.

### 4.2. Discussion

*Scope Inclusion*. To add scope inclusion, an HRF model must exclude inclusive synchronization from the set of potentially racing operation pairs. In the formalism, this is handled by the definition of a synchronization conflict. This is a relatively simple change over *HRF-direct*; the majority of the complexity in the *HRF-direct-relaxed* formalism comes from the support for relaxed atomics.

*Relaxed Atomics*. A goal of *HRF-direct-relaxed* is to define non-SC executions, so we cannot start with the same simplifying assumption made in *HRF-direct* that all candidate executions are sequentially consistent. The difference is not unlike the changes made when moving from DRF0 [Adve and Hill 1990] to C++11 [ISO. International Organization for Standardization 2011], and which is explained in detail by Boehm and Adve [2008].

---

### I. Inclusion and Consistent Orders

**Scope Inclusion**: We say two atomic operations $O_{S,A}[\ell]$ and $O'_{S',A'}[\ell']$, with respect to scopes $S$ and $S'$, locations $\ell$ and $\ell'$, and executed by agents $A$ and $A'$, respectively, are inclusive iff $O_{S,A} \approx_{incl} O'_{S',A'}$. For now, assume the operator from Definition 3.1:

$$O_{s,A} \approx_{incl} O'_{s',A'} \equiv (A \in S \wedge A' \in S \wedge A \in S' \wedge A' \in S') \wedge ((S \subseteq S') \vee (S' \subseteq S))$$

**Consistent Orders**: We say two orders, $\overrightarrow{a}$ and $\overrightarrow{b}$, are consistent iff $(\overrightarrow{a} \cup \overrightarrow{b})^+$ is acyclic. Informally, orders are consistent only if they do not contradict one another, e.g., $\overrightarrow{a}$ and $\overrightarrow{b}$ are inconsistent if $O\overrightarrow{a}O'$ and $O'\overrightarrow{b}O$ for some operations $O$ and $O'$.

---

### II. Conflict Definitions

**Ordinary Conflict**: Memory actions $O[\ell]$ and $O'[\ell']$ conflict iff $\ell = \ell'$, at least one is a store, and at least one is ordinary (non-atomic).

**Atomic Conflict:**: Two atomic operations $O_{S,A}[\ell]$ and $O'_{S',A'}[\ell']$ conflict iff $\ell = \ell'$, at least one is a write or read-modify-write, and $O_{S,A} \not\approx_{incl} O'_{S',A'}$.

---

### III. Executions

**Plausible Execution**: An execution $E$ of program $P$ is plausible iff:

(1) In $E$, the value of any load $L[\ell]$ is either the initial value of $\ell$ or the value produced by some other store, $S[\ell]$, to the same location in $E$

(2) No loaded value depends on itself. (A precise definition of this constraint is an unrelated and not fully solved problem already present in languages such as C/C++11; see Section 4.4).

**Candidate Execution**: A candidate execution is any plausible execution that respects the apparent orders and rules in Section IV below.

---

### IV. Consistent Apparent Orders in a Candidate Execution

**Program Order $(\overrightarrow{po_a})$**: Operations $O$ and $O'$ are in program order, written $O \ \overrightarrow{po_a} \ O'$ iff both are from the same agent $a$ and $O$ comes before $O'$ in the execution control flow. When referenced without a subscript, $\overrightarrow{po}$ refers to $\bigcup_{a \in \mathbb{A}} \overrightarrow{po_a}$ for all agents $\mathbb{A}$.

**Sequentially Consistent Atomic Order $(\overrightarrow{sc})$**: There is an apparent total order, $\overrightarrow{sc}$ of all the $memory\_order\_seq\_cst$ atomic operations. $\overrightarrow{sc}$ must be consistent with $\overrightarrow{po}$.

**Coherent Order $(\overrightarrow{coh_\ell})$**: There is an apparent total order, $\overrightarrow{coh_\ell}$, of all accesses by all actors to any single location $\ell$. $\overrightarrow{coh_\ell}$ must be consistent with $\overrightarrow{sc}$ and $\overrightarrow{po}$ for all $\ell$. The read and write components of an atomic read-modify-write must be adjacent in $\overrightarrow{coh_\ell}$. When referenced without a subscript, $\overrightarrow{coh}$ refers to $\bigcup_{\ell \in \mathbb{L}} \overrightarrow{coh_\ell}$ for all locations $\mathbb{L}$.

**Scoped Synchronization Order $(\overrightarrow{so_a})$**: $\overrightarrow{so_a}$ captures the synchronization operations visible to an agent $a$. Given a release memory action, $Rel_{S,A}[\ell]$ that specifies scope $S$ and is executed by agent $A$, and an acquire memory action, $Acq_{S',A'}[\ell]$ that specifies scope $S'$ and is executed by agent $A'$, $Rel_{S,A}[\ell] \ \overrightarrow{so_a} \ Acq_{S',A'}[\ell]$ iff $a \in S$, $a \in S'$, $Rel_{S,A}[\ell] \approx_{incl} Acq_{S',A'}[\ell]$, and $Rel_{S,A}[\ell] \ \overrightarrow{coh_\ell} \ Acq_{S',A'}[\ell]$.

Fig. 9. Part 1 of the formalization of *HRF-direct-relaxed*.

---

[*Continued from page 13*]

### IV. Consistent Apparent Orders (continued) and Load Values in a Candidate Execution

---

**Heterogeneous-Happens-Before-Direct-Relaxed($\overrightarrow{hhb.dr}$):** The union of the irreflexive transitive closures of all scope synchronization orders with program order:

$$\bigcup_{a \in \mathbb{A}} ((\overrightarrow{po} \cup \overrightarrow{so_a})^+)$$

Where $\mathbb{A}$ is the set of all agents.
Further, $\overrightarrow{hhb.dr}$ cannot contain a cycle and is consistent with $\overrightarrow{coh_\ell}$ for all locations $\ell$ and with $\overrightarrow{sc}$.

**Value of a Load:** A load $L[\ell]$ observes the value produced by the most recent store $S[\ell]$ in $\overrightarrow{coh_\ell}$:,

$$valueof(L[\ell]) = valueof(S[\ell]) : (S[\ell] \;\overrightarrow{coh_\ell}\; L[\ell]) \wedge (\nexists S'[\ell] : S[\ell] \;\overrightarrow{coh_\ell}\; S'[\ell] \;\overrightarrow{coh_\ell}\; L[\ell])$$

**Ordinary Load Constraint:** If an ordinary load $L[\ell]$ observes ordinary store $S[\ell]$, then $S[\ell] \;\overrightarrow{hhb.dr}\; L[\ell]$.

### V. Races and Actual Executions

---

**Heterogeneous Race:** A candidate execution contains a heterogeneous race iff two conflicting (ordinary or atomic) actions $O$ and $O'$ are unordered in $hhb.dr$:

$$\neg(O \;\overrightarrow{hhb.dr}\; O' \vee O' \;\overrightarrow{hhb.dr}\; O)$$

**Heterogeneous-race-free Program** A program is heterogeneous-race-free iff all of its candidate executions are heterogeneous-race-free.

**Racy Program:** Any program containing a heterogeneous race is considered racy.

**Result of a Heterogeneous-race-free Program:** The result of any heterogeneous-race-free program will be one of its candidate executions.

**Result of a Racy Program** The outcome of a racy program is undefined on a conforming implementation.

Fig. 9. Part 2 of formalization of *HRF-direct-relaxed*.

*Apparent Orders.* We explicitly define two apparent orders that must exist in a candidate execution and that are not explicitly defined in *HRF-direct*.[4] The first, $\overrightarrow{sc}$, is a total order of all atomics using *mo_sc* order. As we have already discussed, this does not necessarily mean that an implementation must serialize all *mo_sc* atomics that it executes. The second, $\overrightarrow{coh}$, is a total order of all loads and stores to the same location. This constraint essentially restricts HRF to systems that support hardware coherence, though that coherence mechanism does not need to be a conventional read-for-ownership style CPU protocol (e.g., a MESI protocol), and can instead be something more basic similar to what modern GPUs implement.

The other orders defined in the model, $\overrightarrow{so_a}$ and $\overrightarrow{hhb.dr}$, are derived from $\overrightarrow{sc}$ and $\overrightarrow{coh}$. Scoped synchronization order is defined per-work-item because of scope inclusion. It is not sufficient to say, for example, that there is an order of synchronization operations within a single scope (as *HRF-direct* does) because agents can directly synchronize using different scopes. It would also be too strong to say there is an order among

---

[4]However, these orders do in fact still exist in *HRF-direct* because all candidate executions are sequentially consistent.

```
Device D1
    Work group W
        Work item W1
            101:    X.store(1, mo_rel, ms_dev);
    Work group X
        Work item X1
            201:    Y.store(1, mo_rel, ms_dev);
    Work group Y
        Work item Y1
            301:    A = X.load(mo_acq, ms_dev);
            302:    B = Y.load(mo_acq, ms_dev);
    Work group Z
        Work item Z1
            301:    C = Y.load(mo_acq, ms_dev);
            302:    D = X.load(mo_acq, ms_dev);
```

Fig. 10. Assuming that $X$ and $Y$ are initialized to 0, in *HRF-direct-relaxed*, an implementation is allowed to produce the non-SC result A = C = 1 and B = D = 0.

all synchronization regardless of scope. Therefore, *HRF-direct-relaxed* effectively defines an order of synchronization among any group of work items that could directly synchronize.

*Load Value*. The value of an atomic load in *HRF-direct-relaxed* will either be the value of *some most recent* ordinary or atomic store in $\overrightarrow{hhb.dr}$ (i.e., given a store S a load observes, there is no store S′ that comes between S and the load in $\overrightarrow{hhb.dr}$) or the value of *some* atomic store that is unordered with respect to the load in $\overrightarrow{hhb.dr}$. For example, HRF-relaxed permits the HRF example in Figure 10 to obtain the non-SC result A = C = 1 and B = D = 0 in some executions.

### 4.3. Sketch of Equivalence to HRF-direct

As expected, with the HRF-relaxed formulation we can guarantee that any HRF program that only uses *mo_sc* atomics will always result in a sequentially consistent execution (see supplemental material for formalization [Gaster et al. 2015]). With this property, it is safe for nonexpert users to revert to the more simple HRF-direct model and thereby never have to reason about the valid but complex non-SC orderings.

At a high level, we prove that HRF-direct-relaxed is equivalent to HRF-direct for any program that only uses *mo_sc* atomics and exact-scope synchronization by proving that those conditions always produce sequentially consistent candidate executions in both models. Because the definitions in Section V of Figure 9 are the same as those in *HRF-direct*, the two models are equivalent.

### 4.4. Other Considerations

The HRF model presented here takes inspiration from the C++ formalization of relaxed atomics, and as such has inherited a known issue in the C++ model relating to so-called "out-of-thin-air" values [Boehm 2013]. Solutions to the problem have been proposed, though they are controversial [Boehm and Demsky 2014]. Thus, we do not take a stance in this article or propose any new solutions, instead focusing on the novel aspects of *HRF-∗-relaxed* relating to heterogeneous systems.

We also do not handle other complications that could arise in a real model in an attempt to keep our presentation as simple as possible. These other complications, such as unaligned loads and partial program order, have well-known and uncontroversial solutions, and we therefore assume could be easily addressed.

### 4.5. HRF-Direct-Relaxed Base Implementation

We assume a system organized like the one in Figure 2(b). We define mapping $c(s)$ and $C(s)$ of dynamic scope to physical cache(s) as follows:

| | |
|---|---|
| **c(Subgroup)** | nil |
| **c(Work group)** | The local write buffer of the executing agent. |
| **c(Device)** | The local write buffer and L1 cache of the executing agent. |
| **c(System)** | The local write buffer, L1 cache, and L2 cache of the executing agent. |
| | |
| **C(Subgroup)** | The local write buffer of the executing agent. |
| **C(Work group)** | The local L1 cache of the executing agent. |
| **C(Device)** | The local L2 cache of the executing agent. |
| **C(System)** | Main memory (DRAM). |

We assume the agents execute loads and stores in program order and that the write buffer drains in program order. Ordinary requests waiting in input queues for caches may be reordered if the requests are to different locations. Ordinary requests to the same location cannot reorder, though loads (stores) may coalesce into a single memory system request if there is no store (load) in program order between them. Requests cannot reorder around any other request that originated from a sequentially consistent atomic.

Caches maintain valid, dirty, and invalid states. On a flush, all request queues are drained and all dirty data is evicted to the next level of cache or main memory. On an invalidate, all request queues are drained and all valid data is invalidated. No new request can be serviced while a maintenance operation is pending. Caches will only return valid or dirty data to an agent. Dirty lines are cleaned periodically by writing back to the next level of cache. A line is guaranteed to be cleaned a finite time after the line was written.

A load completes when the value it will return is read from a cache or memory. A store completes when the value it produces is written into the local write buffer. A read-modify write completes when the store portion completes at the target scope.

The system operates as listed in Figure 11. For simplicity, we assume that memory orders *acquire* and *release* both map to *seq_cst*.

We prove in the supplemental material [Gaster et al. 2015] that the base implementation produces all the required apparent orders and therefore results in a sequentially consistent execution for any race-free program using sequentially consistent atomics.

### 4.6. HRF-Indirect-Relaxed

We define a variant of *HRF-∗-relaxed* that supports transitive synchronization through scopes similar to *HRF-indirect*. *HRF-indirect-relaxed* is identical to *HRF-direct-relaxed* in all ways except for a fully transitive happens-before relation shown in Figure 12.

## 5. ADDING OBSERVABILITY AND MULTIPLE ADDRESS SPACES

To represent OpenCL, we must extend *HRF-relaxed* futher to account for coarse-grain buffers and multiple address spaces. We tackle each in turn.

### 5.1. Observability

Real heterogeneous languages allow coarse-grain data sharing in addition to fine-grain sharing achieved through atomics. Updates to data in a coarse-grain region only propagate between nodes at explicit synchronization points (like a kernel being/end),

| | |
|---|---|
| **Ordinary Load** | On an ordinary load, the system searches for a valid copy of the location starting in the local subgroup write buffer and continuing out to main memory. When a valid copy is found, that value is written into any cache that has been searched, the value is returned to the agent, and the load terminates the search. |
| **Ordinary Store** | On an ordinary store, the system inserts the store value into the local write buffer. |
| **Relaxed Atomic Load** | Same as an ordinary load. |
| **Relaxed Atomic Store** | Same as an ordinary store. |
| **Relaxed Read-modify-write** | The read-modify-write operation is performed atomically at cache or memory corresponding to C(S). |
| **Seq_cst Atomic Load w/ scope S** | All write buffers, and/or caches in $c(S)$ are flushed/invalidated. After the cache operations complete, the load proceeds the same as an ordinary load. Operations later in program order cannot execute before the atomic load completes. |
| **Seq_cst Atomic Store w/ scope S** | All write buffers and/or caches in $c(S)$ are flushed. After the cache operations complete, the store proceeds as an ordinary store. Operations earlier in program order must be completed before the atomic store can execute. |
| **Seq_cst Atomic Read-modify-write w/ scope S** | All write buffers and/or caches in $c(S)$ are flushed. After the cache operations complete, the read-modify-write completes in the cache or memory corresponding to C(s). Operations earlier in program order must be issued and completed before the atomic read-modify-write can execute and operations later in program order cannot issue before the read-modify-write completes. |

Fig. 11.   Memory subsystem actions in the *HRF-direct-relaxed* example implementation.

**Heterogeneous-Happens-Before-Indirect-Relaxed($\overrightarrow{hhb.ir}$):** The irreflexive transitive closure of program order with the union of all scope synchronization orders:

$$(\overrightarrow{po} \cup \bigcup_{a \in A} \overrightarrow{so_a})^+$$

Further, $\overrightarrow{hhb.ir}$ cannot contain a cycle and is consistent with $\overrightarrow{coh_{[\ell]}}$ for all locations $\ell$ and with $\overrightarrow{sc}$.

Fig. 12.   Happens-before order in *HRF-indirect-relaxed*. The full model follows identically to that in Figure 9 but with $\overrightarrow{hhb.dr}$ replaced with $\overrightarrow{hhb.ir}$.

and cannot be synchronized through atomics. Coarse-grain buffers allow, for example, a graphics processor to use its own device memory that is not coherent with the rest of the system and thus requires bulk data copying at coarse-grain synchronization points. In OpenCL these synchronization points may be event dependencies between data-parallel kernels, or map and unmap calls that synchronize with the host thread.

To add coarse-grain buffer support to the HRF-relaxed models, and hence to fully support the range of current and future heterogeneous programming models, we must extend the model to include the concept of observability as seen in Figure 13. The goal is to allow for sets of memory locations that are migrated by some external entity in and

---

**Maximum Dynamic Memory Scope**: For a given memory location $L$ the maximum dynamic scope $MaxS(L)$ at which a write to location $L$ may become visible, irrespective of the scope of subsequent synchronization operations. Maximum scope is dependent on an actor being a valid modifier of a given region as defined outside of the memory model.

**Observability** $(O(A, B))$: An operation $A$ is observable by an operation $B$ iff $A_{MaxS(locationof(A))} \approx_{incl} B_{MaxS(locationof(B))} \wedge (locationof(A) == locationof(B))$.

**Heterogeneous Race**: A heterogeneous race occurs between two conflicting (ordinary or synchronization) memory actions $A$ and $B$, iff $A$ *is not observable* to $B$ or $A$ and $B$ are unordered in $hhb.dr$:

$$\neg(O(A, B) \wedge (A \overrightarrow{hhb.dr} B \vee B \overrightarrow{hhb.dr} A))$$

**Value of a Load**: In a heterogeneous-race-free program, a load observes the most recent *observable* ordinary or synchronization store in in $\overrightarrow{coh}$.

---

Fig. 13. Formalization of HRF-direct-relaxed-observable as a set of changes from HRF-direct-relaxed in Figure 9. As for HRF-indirect-relaxed, HRF-indirect-relaxed-observable is modified with the happens-before order from Figure 12.

out of given visibility zones. This migration will be performed at coarse synchronization points.

At any given point in time a given location will be available in a particular set of scope instances out to some maximum instance and by some set of actors. Only memory operations that are inclusive with that maximal scope instance will observe changes to those locations.

The default maximum scope instance for traditional memory locations in a fully coherent system is *system* scope. For a given point in time a coarse-grained allocation will be visible to a particular device $D$, and hence synchronization can only happen at *device* scope. Under the terms of the model actions to that allocation from another device $E$ would not be able to observe actions from $D$.

Note, in particular, that even in the presence of observability, and like scopes in general as described in Hower et al. [2014], a HRF program will produce a consistent total coherent order. The intuition behind this is that while different memory regions may strictly order differently, no such reordering is observable to clients. This is little different from the way that between synchronization operations memory updates may reorder in the basic SC for DRF models.

The addition of observability definitions to the model offers three obvious benefits in describing a model like OpenCL. The first is that it formalizes the rules around coarse-grained memory. The second is that it offers an opportunity to simplify the rules about SC, beyond even the simplification that HRF offers. The third relates to a simplification of OpenCL address spaces, which we discuss in more detail in Section 7.1.

### 5.2. Multiple Address Spaces

OpenCL and related languages aim to support a wide range of memory architectures, including those with scratchpad memories. One consequence of this is that OpenCL has two separate address spaces, *global* and *local*, that are covered by almost entirely separate *happens-before* relations. These relations may be rejoined carefully using specific fences that join the two (see Section 6.1).

Separate happens-before relations allow, for example, a local memory fence to complete before operations to global have completed and vice versa. The consequence of this flexibility is complexity in the memory model. We argue later in Section 7.1 that OpenCL and similar languages are overdesigned in this regard; scopes are sufficient to achieve most of the potential optimizations from multiple-happens-before relations, and thus the multiple-happens-before relations could be dropped in future revisions.

**Per-Address-Space Program Order($\overrightarrow{po_{a,as}}$):** An operation O occurs before operation O' in $\overrightarrow{po_{a,as}}$ iff O $\overrightarrow{po_a}$ O', O is a memory operation affecting a location in $as$, and O' is a memory operation affecting a location in $as$.

**Per-Address-Space Scoped Synchronization Order($\overrightarrow{so_{a,as}}$):** A release memory action, $Rel_{S,A}$, appears before an acquire memory action, $Acq_{S',A'}$ in $\overrightarrow{so_{a,as}}$ iff $Rel_{S,A} \approx_{incl} Acq_{S',A'}$, $Rel_{S,A}$ and $Acq_{S',A'}$ are both in address space $as$, S and S' both include agent $a$, and either

— $Rel_{S,A}$ occurs before $Acq_{S',A'}$ in $\overrightarrow{coh}$, or
— $Rel_{S,A}$ is a release fence, $Acq_{S',A'}$ is an acquire fence, and there exist two atomic operations X and Y such that $Rel_{S,A}$, $Acq_{S',A'}$, X and Y all act on address space $as$, $Rel_S$ $\overrightarrow{po_A}$ X, Y $\overrightarrow{po_{A'}}$ $Acq_{S',A'}$ and X $\overrightarrow{coh_\ell}$ Y.

**Per-Address-Space Heterogeneous-Happens-Before-Indirect-Relaxed($\overrightarrow{hhb.ir_{as}}$):** The union of the irreflexive transitive closures of all scope synchronization orders in address space $as$ with program order:

$$(\bigcup_{a \in A} \overrightarrow{po_{a,as}} \cup \bigcup_{a \in A} \overrightarrow{so_{a,as}})^+$$

Further, $\overrightarrow{hhb.ir_{as}}$ cannot contain a cycle and is consistent with both $\overrightarrow{coh}$ and $\overrightarrow{sc}$.

**Bridging-Synchronization-Order($\overrightarrow{bso}$): ($\overrightarrow{bso}$):** A release memory action, $Rel_{S,A}$, appears before an acquire memory action, $Acq_{S',A'}$, in $\overrightarrow{bso}$ iff $Rel_{S,A}$ $\overrightarrow{hhb.ir_{as}}$ $Acq_{S',A'}$ for all address spaces $as$.

**Heterogeneous-Happens-Before-Indirect-Relaxed($\overrightarrow{hhb.ir}$):** The transitive closure of the union of the happens-before orders for address space locations along with the bridging orderings:

$$(\bigcup_{as \in AS} \overrightarrow{hhb.ir_{as}} \cup \overrightarrow{bso})^+$$

As for the per-address-space versions, $\overrightarrow{hhb.ir}$ cannot contain a cycle and is consistent with $\overrightarrow{coh_\ell}$ for all locations $\ell$ and with $\overrightarrow{sc}$.

Fig. 14. Formalization of the multiple address-space extension to HRF-indirect-relaxed as a set of changes from Figure 9. The heterogeneous race, values of loads, and optionally, incorporation of Figure 13 reapply identically to $\overrightarrow{hhb.ir}$. Note that we add formalization for memory fences, which were until now omitted for clarity, in the per-address-space scoped synchronization order because fences are the bridging operations for OpenCL.

We can represent multiple happens-before orders by instantiating the model for each address space independently. All atomic operations will order separately for each distinct address space with its own order. We further assume that the final order for the entire program results from the transitive closure of the individual orders.

The precise additions to the *happens-before* order that would cause the closure to bridge the individual orders might vary from one language to another. In OpenCL's case it is the specific fences that create this connection.

Figure 14 shows this formalization as applied to the OpenCL case of having two separate orders, but the concept would generalize to any number of distinct happens-before orders were that to be required.

## 6. DETAILS OF THE OpenCL 2.0 MEMORY MODEL

The OpenCL 2.0 memory model maintains the features of the OpenCL 1.x model, including the execution hierarchy and basic synchronization discussed in Section 2.2, and extends it to include support for shared virtual memory (SVM) in a global address space. SVM allocations are distinguished as being either fine-grain or coarse-grain,
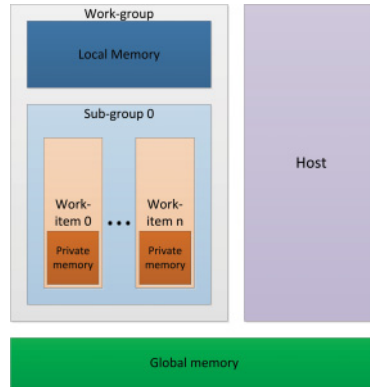
Fig. 15.   OpenCL address spaces.

which affects the observability of memory as well as the types of synchronization that are supported.[5]

When using what OpenCL calls *fine-grain SVM with platform atomics* and restricting to sequentially consistent atomic operations, the system appears similar to the basic model assumed by the original HRF work. A pointer to global memory is valid on any actor, and the host CPU thread does not need to explicitly manage data allocations, transfers, or mapping in device memory that would otherwise be required. In addition, the model provides non-SC atomic operations similar to those found in C/C++, but augmented with the ability to control scope visibility. With these features, it is possible to create programs that take advantage of the hardware support for system-wide shared memory in recent SoCs.

Like C++ and HRF, OpenCL 2.0 fundamentally follows a race-free memory model, such that only race-free programs have well-defined behavior. OpenCL race-free executions are sequentially consistent by default (i.e., when using the default atomic ordering and scope), but can be relaxed to produce well-defined but nonsequentially consistent executions with explicitly relaxed atomic operations (see Section 6).

In the remainder of this section, we provide the nuanced details of the OpenCL 2.0 memory model and then show how to describe it in terms of HRF.

### 6.1. OpenCL Address Spaces

Figure 15 shows OpenCL's address spaces, in which each colored box with the word "memory" represents a different address space. A work item has access to a *private* memory visible only to itself, a *local* memory that is shared between work items in the same work group, and finally a *global* memory shared between all concurrently executing work items as well as the host. The address spaces are disjoint and are assumed to not overlap. In OpenCL 1.x, these address spaces were explicit. In OpenCL 2.0, a programmer can map the preceding address spaces into a single generic virtual memory map, though that mapping does not change the properties of the memory model. For example, virtual addresses corresponding to private memory locations correspond to different physical locations for each work item.

---

[5]We note here that the OpenCL 2.0 specification adds FIFO data structures called pipes and includes support for image data structures with some extensions beyond those available in OpenCL 1.2. The restrictions OpenCL 2.0 applies to both of these leaves them outside of the core memory model and therefore out of the scope of this discussion.

In general, OpenCL's memory model treats address spaces separately and an operation on one address space does not affect the others. The local and global address spaces are the only address spaces that are both shared between work items and writable, and are therefore governed by the properties of the memory model. Private memory is never visible outside of a work item, and so its ordering properties are handled trivially. The memory orderings on local and global are independent and atomic operations applied to one do not affect memory orderings on the other.

It is, however, possible to *join* the global and local address space memory orders such that two work items can communicate between local and global memory. For example, a work item could write data into global memory and then synchronize via a local memory flag with another work item in its own work group. When doing so, the work item would need to issue a special fence operation that joins the two address spaces.

More specifically, OpenCL uses the | operator in a memory fence to combine local and global memory. When two memory fences each specify $CLK\_LOCAL\_MEM\_FENCE|CLK\_GLOBAL\_MEM\_FENCE$, the individual happens-before relations from local and global memory are merged for two issuing work items. This synchronization corresponds to the bridging operations that form $\overrightarrow{bso}$ in Figure 14. No other synchronization operations appear in $\overrightarrow{bso}$ for the OpenCL model.

## 6.2. Forms of Shared Virtual Memory

Shared virtual memory allocations in OpenCL 2.0 can be categorized in three ways:

(1) Coarse-grained buffer
(2) Fine-grained buffer
(3) Fine-grained system

Coarse-grained SVM buffers[6] only guarantee consistency between different agents at coarse-grained synchronization points (*map* and *unmap* operations or intercommand dependencies) and at the granularity of the entire memory allocation. An implementation is only required to present the same virtual address space for a coarse-grained SVM allocation to the subset of devices using that buffer.

At coarse-grained synchronization points, an implementation may copy the data to and from physical locations that are visible only to a specific device. For example, an implementation might copy data in/out of a GPU's physically separate and noncoherent DRAM. As we will see, this property plays an important role in the definition of sequentially consistent atomics and the perceived single total order that is expected to exist for such operations.

Fine-grained buffers can be supported with and without platform atomics. Without platform atomics visibility between different OpenCL devices is, as for coarse-grained buffer allocations, only guaranteed at explicit synchronization points like kernel beginning and end. Unlike coarse-grained buffers, visibility is defined at a byte granularity and does not require map and unmap operations to ensure visibility on the host.

Fine-grained system SVM extends fine-grained support to all host memory. This extends the set of locations visible to OpenCL 2.0's memory model but has no effect otherwise.

When platform atomics are enabled, memory consistency for both fine-grained SVM modes may be achieved by atomic operations directly without the need to wait for coarse synchronization points.

---

[6]We include non-SVM allocations in this category because they behave the same way according to the memory model.

Device D1
    Work group X
        Work item X1
    101   C = 1;
    102   Y = 1;
    103   Z.store(1, mo_sc, ms_svm)

Device D2
    Work group Y
        Work item Y1
    201   if(Z.load(mo_sc, ms_svm)) {
    202       T = C;
    203       U = Y;
    204   }

Fig. 16. Coarse-grained memory ordering in OpenCL. If $C$ is in a coarse-grain buffer, $Y$ and $Z$ are in fine-grained buffers, and $D1 \neq D2$, then $T$ will be 0 and $U$ will be 1.

Visibility for fine-grained memory conceptually maps to the following scopes:

*Fine-grained with platform atomics.*
    *memory_scope_all_svm_devices* or platform wide visibility. Allows for concurrent access from any agent that can participate in the OpenCL shared virtual address space and with ordering and visibility arising directly from atomic operations.
*Fine-grained without platform atomics.*
    *memory_scope_device* or device-only visibility. Concurrent access by different agents to the same byte is not permitted (formally such access is defined as a data race) and coherency is guaranteed only at well-defined synchronization points such as kernel begin and end.

Coarse-grained memory affects the observability of memory locations. For example, in Figure 16, if the storing actor and the loading actor run on the same device, then happens-before guarantees that both T and U will be 1. If, however, they run on different devices, it is possible that T will remain 0 while U was 1. While the synchronization through Z guaranteed a happens-before relationship, coarse-grained memory properties do not guarantee the visibility beyond their maximum scope (in this case Device).

Both of the more coarse forms of shared virtual memory in OpenCL can be covered by bounding observability. The maximum dynamic memory scope for a coarse-grained allocation, or a fine-grained allocation without platform atomics, is device scope. Therefore happens-before actions on a different device will not order operations relative to happens-before actions on the current device. Any store performed on a coarse-grained buffer by one device will not be in the observable coherent order for the location on another device, as described in Figure 13.

The difference between fine-grained allocations without platform atomics and coarse-grained allocations is in the definition of a race. For a coarse-grained buffer the required map, unmap, and event dependencies operations add an entry to the coherent update order for every memory location in the allocation, thus conflicting with updates performed by any other device. A fine-grained allocation will only update side effects caused by the executing kernel and thus conflict only at update granularity.

## 6.3. Scopes

OpenCL 2.0 has a different definition of scope inclusion from the basic one in Definition 3.1. This arises for two reasons. First, the different types of shared memory complicate the notion of dynamic scope equivalence. In particular, the dynamic SVM scope is defined differently depending on whether or not operations use fine-grained buffers. If two operations both use fine-grained buffers, then the SVM scope includes all actors in the system. Otherwise, SVM scope is limited to actors on the same device.

Second, OpenCL only supports synchronization between atomics with identical dynamic scope, which is similar to the rules for the original *HRF-direct* model.

With these two changes, we can define the scope inclusion property for OpenCL. Let us define $fga(O)$ to be true if the operation $O$ affects a location in a fine-grained with platform atomics memory allocation. Then:

*Definition* 6.1 (*OpenCL 2.0 Dynamic Scope Equivalence*). Two scoped synchronization operations, $O_S$ and $O'_{S'}$, with static scopes $S$ and $S'$ that execute on subgroups $SG$ and $SG'$, work groups $WG$ and $WG'$, and devices $D$ and $D'$ have equivalent dynamic scopes iff:

$$
\begin{aligned}
&(S == S' == ms\_sg \quad \wedge \quad SG = SG') &\vee\\
&(S == S' == ms\_wg \quad \wedge \quad WG = WG') &\vee\\
&(S == S' == ms\_dev \quad \wedge \quad D == D') &\vee\\
&(S == S' == ms\_svm \ \wedge \ fga(O_S) \wedge fga(O'_{S'})) &\vee\\
&(S == S' == ms\_svm \ \wedge \ (\neg fga(O_S) \vee \neg fga(O'_{S'})) \wedge D == D')
\end{aligned}
$$

*Definition* 6.2 (*OpenCL 2.0 Scope Inclusion (clscin)*). Two scoped synchronization operations, $O_S$ and $O'_{S'}$, are OpenCL 2.0-inclusive, written $O_s \approx_{clincl} O'_{s'}$, iff the dynamic scope of $O_S$ is equivalent to the dynamic scope of $O'_{S'}$.

### 6.4. HRF-OpenCL

We can now describe the OpenCL 2.0 memory model in terms of HRF concepts.

OpenCL supports scope transitivity, so we base the model on *HRF-indirect-relaxed*. We need to include the simpler OpenCL version of the bridging synchronization order applied to the local and global address spaces described in Section 6.1, OpenCL's notion of scope inclusion from Definition 6.2, and observability for coarse-grained allocations. We also include OpenCL's restrictive definition of sequentially consistent atomics. In the end, we arrive at the model in Figure 17, which picks up at Section IV from Figure 9.

## 7. SIMPLIFICATIONS TO THE OpenCL MODEL USING HRF-INDIRECT-RELAXED

### 7.1. Simplifying the Local/Global Ordering Separation

Unfortunately, maintaining two separate orderings for local and global memory complicates the OpenCL memory model. For most code we can view local and global memory entirely separately, such that we instantiate the entire HRF-indirect-relaxed model twice, once for local accesses and once for global accesses. We saw this in Figure 14.

In addition to the combining of orders, local memory is only observable to *work-group* scope and thus updates are never visible to other work groups. It is a quirk of the OpenCL model that SC may not be applied simultaneously to allocations with platform atomics and allocations without platform atomics, but which can be applied to global allocations without platform atomics and local allocations, even though similar observability restrictions apply in both cases.

By treating local memory as a range of locations $L$ such that $MaxS(L)$ is *memory_scope_work_group* local memory operations need not become visible to other work groups, even when added into a single happens-before ordering. This would allow a simplification of the OpenCL definition to a single happens-before ordering in the memory model and further trivially allow a single sequentially consistent total order $S$ to apply to local memory as well as global order, in all cases as viewed by a valid observer.

There may be other valid reasons for maintaining a separate ordering for local memory; for example, separate hardware scratchpad memories might use different operations with separate ordering guarantees to the global memory cache hierarchy. However, the same might apply to the use of buffers that do and that do not support

## IV. Consistent Apparent Orders And Load Values in a Candidate Execution

**Sequentially Consistent Atomic Order ($\overrightarrow{sc}$):** There is an apparent total order, $\overrightarrow{sc}$ of all the *memory_order_seq_cst* operations *iff* all atomics use scope *mo_svm* and are to a fine-grained allocation with platform atomics or all atomics use scope *mo_dev* and none are to a fine-grained allocation with platform atomics. $\overrightarrow{sc}$ must be consistent with $\overrightarrow{po}$.

**Local Scoped Synchronization Order ($\overrightarrow{so_{a,L}}$):** Given a release memory action, $Rel_{S,A}$, and an acquire memory action, $Acq_{S',A'}$, $Rel_{S,A}$ $\overrightarrow{so_{a,L}}$ $Acq_{S',A'}$ iff $Rel_{S,A} \approx_{clincl} Acq_{S',A'}$, $S$ and $S'$ both include agent $a$, and $Rel_{S,A}$ and $Acq_{S',A'}$ both touch locations in local memory, and either $Rel_S \overrightarrow{coh} Acq_{S'}$, or $Rel_{S,A}$ is a release fence, $Acq_{S',A'}$ is an acquire fence, and there exist two local atomic operations $X$ and $Y$ such that $Rel_{S,A} \overrightarrow{po_A} X$, $Y \overrightarrow{po_{A'}} Acq_{S',A'}$ and $X \overrightarrow{coh_\ell} Y$.

**Global Scoped Synchronization Order ($\overrightarrow{so_{a,G}}$):** Given a release memory action, $Rel_{S,A}$, and an acquire memory action, $Acq_{S',A'}$, $Rel_{S,A}$ $\overrightarrow{so_{a,G}}$ $Acq_{S',A'}$ iff $Rel_{S,A} \approx_{clincl} Acq_{S',A'}$, $S$ and $S'$ both include agent $a$, and $Rel_{S,A}$ and $Acq_{S',A'}$ both touch locations in global memory, and either $Rel_{S,A} \overrightarrow{coh} Acq_{S',A'}$, or $Rel_{S,A}$ is a release fence, $Acq_{S',A'}$ is an acquire fence, and there exist two global atomic operations $X$ and $Y$ such that $Rel_{S,A} \overrightarrow{po_A} X$, $Y \overrightarrow{po_{A'}} Acq_{S',A'}$ and $X \overrightarrow{coh_\ell} Y$.

**Local-happens-before ($\overrightarrow{lhb}$):** The irreflexive transitive union of program order and local scoped synchronization order:

$$(\bigcup_{a \in A} \overrightarrow{po_{a,L}} \cup \bigcup_{a \in A} \overrightarrow{so_{a,L}})^+$$

**Global-happens-before ($\overrightarrow{ghb}$):** The irreflexive transitive union of program order and global scoped synchronization order:

$$(\bigcup_{a \in A} \overrightarrow{po_{a,G}} \cup \bigcup_{a \in A} \overrightarrow{so_{a,G}})^+$$

**Bridging Synchronization Order ($\overrightarrow{bso}$):** A release fence, $R_{SO}$, ordered by local-happens-before order $O$ and global-happens-before order $O'$ appears before an acquire fence, $A_{S'O'}$, ordered by local-happens-before order $O$ and global-happens-before order $O'$, in $\overrightarrow{bso}$ iff $R_{SO} \approx_{clincl} A_{S'O'}$ and $R_{SO}$ is ordered before $A_{S'O'}$ in $\overrightarrow{so_{a,L}}$ or in $\overrightarrow{so_{a,G}}$ for some $a \in A, A'$.

**OpenCL-Happens-Before($\overrightarrow{clhb}$):**

$$(\overrightarrow{lhb} \cup \overrightarrow{ghb} \cup \overrightarrow{bso})^+$$

$\overrightarrow{clhb}$ cannot contain a cycle and is consistent with both $\overrightarrow{coh}$ and $\overrightarrow{sc}$.

**Observability($O(A, B)$:** An operation to global memory $A[\ell]$ executed on device $D$ is observable by another operation to global memory $B[\ell']$ executed on device $D'$ iff $\ell = \ell'$ and $(fga(A) \land fga(B)) \lor (D = D')$.

An operation to local memory $A[\ell]$ is observable to an operation $B[\ell']$ iff $\ell = \ell'$ and both are executed in the same work-group.

**Value of a Load:** A load $L[\ell]$ observes the value produced by the most recent *observable* store $S[\ell]$ in $\overrightarrow{coh}$

## V. Races

**Heterogeneous Race:** A candidate execution contains a heterogeneous race iff two conflicting (ordinary or atomic) actions $A$ and $B$ are not observable or are unordered in *clhb*:

$$\neg(A \overrightarrow{clhb} B \lor B \overrightarrow{clhb} A) \lor \neg O(A, B)$$

Fig. 17. *HRF-OpenCL*. We reuse Sections I–III from Figure 9, but substitute *clincl* for *incl*. We also re-use part of Section IV, specifically coherent order. We use a per-address-space program order from Figure 14. Note that these relations represent a subset of the full HRF-relaxed model as it applies to OpenCL's current behavior.

platform atomics (e.g., the need to use (Peripheral Component Interconnect) PCI-express atomics rather than local cache atomics) in the same application, and no separate order is currently maintained for those cases. In particular, the generic address space applied to local memory maintains a separate ordering for local addresses from that of global addresses. Two seemingly identical pointers both passed to a single function might have accesses ordered entirely separately when used. Applying a single happens-before order to both address spaces would remove this concern.

This simplification would improve the usability of the overall model for developers.

## 7.2. Generalizing SC

The HRF models already demonstrate that a heterogeneous race can be present across scopes even in the presence of a SC ordering. The principle behind this is the same as that behind the DRF models in general: that an ordering only matters for operations that are well defined; everything else can be relaxed. As a result, while OpenCL limits SC ordering to particular scopes, HRF shows that this ordering could be carried through all scopes and still be well -defined. In effect, a clean extension of the model that OpenCL already discusses.

The current OpenCL 2.0 specification limits SC to either one of two situations. For all buffers $b$ and sequentially consistent memory operations $o$ in a given execution:

$$(\forall(b, o)((scopeof(o) = memory\_scope\_all\_svm\_devices) \land \quad fga(b))) \lor$$
$$(\forall(b, o)((scopeof(o) = memory\_scope\_device) \qquad\qquad \land \neg \; fga(b))).$$

This results in a weaker ordering in any program that combines a fine-grained-with-atomics allocation with device scope, or all-svm-devices scope with coarse-grained or nonatomic fine-grained allocations. This is a clear composability problem because the scope is controlled by kernel code and the allocation type is controlled by host code, with no guarantee that the two are written by the same developer.

HRF-indirect already shows how, in the absence of these coarser allocations, sequentially consistent properties can be extended across scopes, removing the need for the restriction to a single scope. Observability allows us to assume a single SC ordering on all buffer types with well-defined semantics. Operations to coarse allocations will be sequentially consistent to the operating device. Other devices are not valid observers and hence the order they see the operations in is undefined.

## 8. CONCLUSION

In this article we have described how to extend the class of HRF memory consistency models to incorporate four complex features of industrial memory models. This includes support for nonsequentially consistent operations, a property called scope inclusion, limited observability of memory locations, and multiple address spaces. By building from the more basic HRF-direct and/or HRF-indirect models, we have shown how users of industrial models can restrict their programs to comply with a pure SC for HRF model and ignore the hard-to-understand complications. We have shown this explicitly with the OpenCL 2.0 model.

Using our formalization, we have shown how OpenCL could be extended to support a simpler notion of local memory and a wider range of sequentially consistent executions.

## REFERENCES

Sarita Adve. 2010. Data races are evil with no exceptions: Technical perspective. *Commun. ACM* 53, 11 (Nov. 2010), 84–84.

Sarita V. Adve and Mark D. Hill. 1990. Weak ordering—a new definition. In *Proceedings of the International Symposium on Computer Architecture*. ACM, New York, NY, 2–14.

Hans-J. Boehm. 2013. N3710: Specifying the absence of out of thin air results (LWG2265).

Hans-J. Boehm and S. V Adve. 2008. Foundations of the C++ concurrency memory model. In *Proceedings of the International Symposium on Programming Language Design and Implementation*.

Hans-J. Boehm and Brian Demsky. 2014. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC'14)*. ACM, New York, NY, Article 7, 6 pages.

Benedict R. Gaster, Derek Hower, and Lee Howes. 2015. HRF-Relaxed: Adapting HRF to the complexities of industrial heterogeneous memory models (Supplemental material). (Jan. 2015). Retrieved from http://benedictgaster.org/?page_id=278.

Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*. ACM, New York, NY, 15–26. DOI:http://dx.doi.org/10.1145/325164.325102

Blake A. Hechtman, Shuai Che, Derek R. Hower, Yingying Tian, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. QuickRelease: A throughput-oriented approach to release consistency on GPUs. In *HPCA*. 189–200.

Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free memory models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (APLOS'14)*. ACM, New York, NY, 427–40.

HSA Foundation. 2012. *Heterogeneous System Architecture: A Technical Review*.

ISO. International Organization for Standardization. 2011. ISO/IEC 14882:2011 information technology—programming languages—C++.

Leslie Lamport. 1979. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.* 1, 1 (Jan. 1979), 84–97.

Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. 2010. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 210–221. DOI:http://dx.doi.org/10.1145/1815961.1815987

Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2010. DRFX: A simple and efficient memory model for concurrent programming languages. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, New York, NY, 351–362. DOI:http://dx.doi.org/10.1145/1806596.1806636

NVIDIA Corporation. 2013. *CUDA 5.5 C Programming Guide*.

Oracle. 2014. *The Java Langauge Specification*.

Herb Sutter. 2012. Atomic weapons: The C++ memory model and modern hardware. In *C++ and Beyond*.