# Heterogeneous-race-free Memory Models

Derek R. Hower[†], Blake A. Hechtman[†§], Bradford M. Beckmann[†], Benedict R. Gaster[†], Mark D. Hill[‡†], Steven K. Reinhardt[†], David A. Wood[‡†]

[†]AMD Research
{derek.hower, brad.beckmann, benedict.gaster,steve.reinhardt}@amd.com

[§]Duke University
Electrical and Computer Engineering
blake.hechtman@duke.edu

[‡]University of Wisconsin-Madison
Computer Sciences
{markhill, david}@cs.wisc.edu

## Abstract

Commodity heterogeneous systems (e.g., integrated CPUs and GPUs), now support a unified, shared memory address space for all components. Because the latency of global communication in a heterogeneous system can be prohibitively high, heterogeneous systems (unlike homogeneous CPU systems) provide synchronization mechanisms that only guarantee ordering among a subset of threads, which we call a scope. Unfortunately, the consequences and semantics of these scoped operations are not yet well understood. Without a formal and approachable model to reason about the behavior of these operations, we risk an array of portability and performance issues.

In this paper, we embrace scoped synchronization with a new class of memory consistency models that add scoped synchronization to data-race-free models like those of C++ and Java. Called sequential consistency for heterogeneous-race-free (SC for HRF), the new models guarantee SC for programs with "sufficient" synchronization (no data races) of "sufficient" scope. We discuss two such models. The first, HRF-direct, works well for programs with highly regular parallelism. The second, HRF-indirect, builds on HRF-direct by allowing synchronization using different scopes in some cases involving transitive communication. We quantitatively show that HRF-indirect encourages forward-looking programs with irregular parallelism by showing up to a 10% performance increase in a task runtime for GPUs.

***Categories and Subject Descriptors*** C.0 [**Computer Systems Organization**]: Hardware/software interfaces, systems specifications methodology

***Keywords***: memory consistency model; heterogeneous systems; data-race-free; task runtime

## 1 Introduction

Though it took nearly 30 years, languages for CPU systems like C++ [7] and Java [26] have started to adopt a class of memory models called sequential consistency for data-race-free (SC for DRF) [3]. These models are formal and precise definitions that allow many low-level optimizations. SC for DRF models are also accessible to most programmers because they guarantee sequential consistency to any program that is synchronized correctly – meaning that most programmers need not concern themselves with the gory details. A goal of this work is to achieve the same benefits for heterogeneous systems in significantly less time.

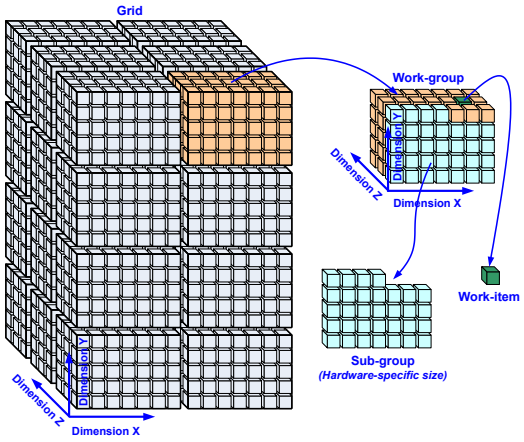


**Figure 1. The OpenCL execution hierarchy.**

Unfortunately, it is not straightforward to apply SC for DRF models in heterogeneous systems because current heterogeneous systems support synchronization operations with non-global side effects. For performance reasons, languages like OpenCL™ [29] and CUDA [12] decompose a problem into execution groups. Figure 1 shows the OpenCL execution model, in which a work-item (like a CPU thread) belongs to four different groups (which we call scopes): sub-group[1], work-group, device, and system[2]. Starting with OpenCL 2.0 [28], programmers can synchronize a work-item through shared memory[3] with respect to any one of these groups using what we call *scoped synchronization* operations. For example, OpenCL provides a fence opera-


*ASPLOS'14,* March 1–5, 2014, Salt Lake City, Utah, USA.

http://dx.doi.org/10.1145/2541940.2541981

[1] Sub-groups are optional in OpenCL, but will usually be defined on an SIMT GPU, and correspond to vector units.
[2] We use OpenCL terminology in this paper. In Section 6 we discuss the CUDA equivalents.
[3] As in CPU shared memory, not the CUDA "shared memory" scratchpad.

tion that synchronizes a work-item with other work-items in its own work-group but not with work-items in a different work-group.

Scoped synchronization breaks some fundamental tenets of SC for DRF models. For example, in OpenCL 2.0 it is possible to write a racey program that is composed entirely of atomics if those atomics do not use scopes correctly. It may also be possible to write a program in which all ordinary accesses are protected with atomics such that there can be no concurrent conflicting accesses, yet that program may still contain a race if those atomics are not scoped correctly. We find that many of the consequences of these differences are not yet well understood.

In this paper, we propose a new class of memory models for platforms that use scoped synchronization called sequential consistency for heterogeneous-race-free (SC for HRF). SC for HRF models define correctness in terms of a sequentially consistent execution and rules for what is considered "enough" synchronization to avoid races. The SC for HRF models achieve two fundamental goals: they provide a precise definition of memory behavior in a heterogeneous execution and they provide a framework to describe that behavior in a way that typical programmers can understand. While we focus on GPUs in this paper due to their prevalence, we expect the SC-for-HRF concepts and insights to apply to other heterogeneous parts (e.g., DSPs).

We describe two possible SC for HRF variants:

- **HRF-direct** requires communicating work-items to synchronize using the exact same scope.
- **HRF-indirect** extends HRF-direct to support transitive communication using different scopes.

In *HRF-direct*, if a producer wants to synchronize with a consumer, the producer must execute a release to scope S and the consumer must later execute an acquire to that exact same scope S. Even though it is (relatively) simple, we find that HRF-direct is a perfectly adequate model to use with highly regular data-parallel algorithms like the ones that currently exist in most general-purpose GPU (GPGPU) applications. In these workloads, it is easy to precisely determine the work-items involved in a communication, thereby making it easy to choose the correct minimal scope for synchronization (Section 3.1). HRF-direct can also be used to understand the OpenCL 2.0 (and likely CUDA; see Section 5.1.3) memory model, thereby giving programmers a more intuitive description of the memory behavior of existing languages.

While HRF-direct is good for today's workloads, it may not be sufficient for emerging heterogeneous applications. First, it makes it difficult to optimize applications with irregular parallelism in which the producer and consumer may not be known *a priori*. For this same reason it can be difficult to write composable library functions in HRF-direct. Second, HRF-direct is overly conservative for current GPU memory systems; some actual hardware will support a more permissive model (from the software perspective). For these reasons, we define an alternative model that relaxes HRF-direct's strict scoping requirements and may enable a wider range of parallel software on heterogeneous accelerators.

In the forward-looking *HRF-indirect* model, two threads can synchronize indirectly through a third party even if the two threads interact with that third party using different scopes. For example, threads A and C can communicate if A synchronizes with another thread B using scope S1 and then B synchronizes with C using scope S2. This type of transitive interaction can enable more irregular parallelism in future heterogeneous applications (e.g., in an algorithm in which A does not know who C is or where C is located). HRF-indirect better supports programmability features like composability and is currently supported by existing GPU hardware. However, HRF-indrect may be harder to support if heterogeneous memories are non-hierarchical and it can prevent some future low-level optimizations allowed by HRF-direct (Section 6).

To explore the practical differences between HRF-direct and HRF-indirect, we have developed a task-parallel runtime for GPUs that performs automatic load balancing as an example of irregular parallelism. We describe the runtime in Section 7 focusing on the design differences between HRF-direct and HRF-indirect implementations of the runtime. In Section 8 we evaluate the performance consequences of those differences, showing that the HRF-indirect version can outperform the HRF-direct version by 3-10% in a system resembling modern hardware. However, we also point out the caveats to this result, including considerations about system optimizations prohibited by HRF-indirect that could be used in HRF-direct systems.

HRF-direct and HRF-indirect are two of many possible formulations of an SC for HRF model. In Section 5.2 we discuss some other properties that SC for HRF models could adopt in the future, notably including a property called *scope inclusion* that exploits the hierarchical nature of most scope definitions. In summary, in this paper we make the following contributions to the state of the art:

***Define with Scoped Synchronization:*** We identify the need to more formally define and better describe the semantics of scoped synchronization operations, especially when work-items interact through different scopes.

***SC for HRF Models:*** We propose a class of programmer-centric memory models called sequential consistency for heterogeneous-race-free to describe systems with scoped synchronization support.

***HRF-direct:*** We define the HRF-direct model for today's highly regular GPGPU programs with current standards (e.g., OpenCL 2.0).

***HRF-indirect:*** We define the forward-looking HRF-indirect model for future irregular GPGPU programs with runtime-determined data flow from producers to consumers.

```
atomic<int> A = {0};
atomic<int> B = {0};
int X;

Thread t1

X = 1;
A.store(1, memory_order_seq_cst);     // release

Thread t2

  while(!A.load(memory_order_seq_cst)); // acquire
  int R2 = X; // R2 will receive X=1 from t1
  B.store(1, memory_order_seq_cst);     // release

Thread t3

    while(!B.load(memory_order_seq_cst)); // acquire
    int R3 = X;  // R3 will receive X=1 from t1
```

**Figure 2. An example of a data-race-free execution in C++11. t1 and t3 do not race on location X because of a transitive interaction through t2.**

# 2 Background and Setup

In this section, we first provide a history of memory consistency models for CPUs, focusing particularly on the data-race-free models that we build on in this paper. Then we describe the features of modern GPU hardware that necessitate scoped synchronization for good performance.

## 2.1 Memory Consistency Models

A *memory (consistency) model* specifies the semantics of shared memory, so that both users and implementers can push limits using precise correctness definitions [2, 36]. *Low-level models* interface low-level software (e.g., compilers, JITs, and runtime) to hardware, while *high-level models* interface high-level languages (e.g., C++ and Java) to the "system" that includes low-level software and hardware.

For microprocessors and multi-core chips, low- and high-level memory models have arguably taken (at least) *three decades to mature*. Microprocessors emerged in 1971 for uniprocessors and were also put in multiprocessors in the early 1980's (e.g., Sequent Balance [37]). Even though Lamport [24] specified sequential consistency (SC) in 1979, most multiprocessors do not implement SC due to their use of write buffers. In 1991, Sindhu et al. [35] formalized total store order (TSO) as a low-level memory consistency model that captured what microprocessors often do in the presence of a write buffer. Meanwhile, to increase flexibility, academics investigated more relaxed low-level memory models, such as weak ordering [14], release consistency [15], and data-race-free (DRF) [3]. Notably, providing sequential consistency to data-race-free programs (SC for DRF) [3], became a cornerstone of the Java [26] and C++ [7] models in 2005 and 2008, respectively. In 2008, a full 15 years after the first multiprocessor x86 architecture came to market, Intel released a formalization of x86-TSO (summarized by Owen, et al. [33]).

A goal of this work is to accelerate the process of defining high- and low-level memory models for heterogeneous systems so that it takes much less than three decades.

## 2.2 Sequential Consistency for Data-race-free

Sequentially consistency guarantees that the observed order of all memory operations is consistent with a theoretical execution in which each instruction is performed one at a time by a single processor [24]. SC preserves programmer sanity by allowing them to think about their parallel algorithms in sequential steps. Unfortunately, true SC can be difficult to implement effectively without sacrificing performance or requiring deeply speculative execution [17]. As a result, most commercially relevant architectures, runtimes, and languages use a model weaker than SC that allows certain operations to appear out of program order at the cost of increased programmer effort.

To bridge the gap between the programming simplicity of SC and the high performance of weaker models, a class of models, called SC for DRF, was created that guarantees an SC execution if the program has no data races (i.e., is protected by control synchronization and memory fences). In the absence of data races, the system is free to perform any reordering as long as it does not cause an observable violation of SC. SC for DRF models differ on the defined behavior in the presence of race, and vary from providing no guarantees [3, 7] to providing weak guarantees like write causality [26].

In Figure 2 we give a concrete example of a sequentially consistent data-race-free program in C++.[4] In this program, the load into R2 does not race with the store X=1 because they are separated by paired synchronization accesses on atomic variable A. The later load into R3 also does not cause a race because in SC for DRF models, synchronization has a transitive effect (a property that will be important in our subsequent discussion of HRF models). In the example, the load into R3 does not form a race with the store X=1 by thread t1 even though t1 and t3 do not use the same synchronization variable. In this case, the threads are synchronized indirectly through the causal chain involving atomics A and B.

## 2.3 Modern GPU Hardware

Because current GPU hardware is optimized to stream through data (as is common for graphics), GPU caches are managed differently than CPU caches. CPU memory systems optimize for reads and writes with high locality via read-for-ownership (RFO) cache coherence protocols. RFO protocols obtain exclusive permission for a line before writing it. Fortunately, good locality makes it likely that this initial cost gets amortized over many subsequent low-cost cache hits.

[4] Those familiar with C++ will notice that the explicit memory order syntax is unnecessary. We use the verbose form to match with OpenCL examples later.

RFO protocols make less sense for GPUs whose data streams afford little opportunity to amortize the latency of obtaining exclusive coherence permission because subsequent caches hits are less common. Current GPU caches behave less like conventional CPU caches and more like streaming write-combining buffers that complete writes without obtaining exclusive coherence permission (Section 3.2).

With GPU write-combining (non-RFO) caches, when two work-items in a GPU synchronize, hardware must ensure both work-items read/write from a mutually shared cache or memory level. To implement this correctly, current hardware may make potentially shared data uncachable, disable private caches for all potentially shared data, or flush/invalidate caches at synchronization points [4, 30]. In the final approach, the particular caches that need to be flushed/invalidated depend on which work-items are synchronizing (i.e., what the scope of synchronization is), and *therefore some synchronizations are more costly than others*. This is in stark contrast to a CPU using RFO coherence, in which the hardware actions necessary at synchronization events (e.g., handling probes or flushing a store buffer) are the same regardless of which threads are involved in the communication.

### 2.4 Data-race-free is Not Enough

There are two ways to achieve the benefits of SC for DRF models in heterogeneous systems. First, we could use an existing SC for DRF model as-is by forcing all synchronization to use global scope. However, this would eschew the benefits of scoped operations, likely resulting in inefficient and poorly performing software.

Second, we could adapt the SC for DRF models to deal with the behavior of scoped operations. In our view, this approach fundamentally changes the understanding of race-free. Informally, many understand data-race-free to mean "conflicting accesses do not occur at the same time" [1]. As we will show in the next section, that informal understanding is not strong enough with scoped synchronization. With scoped synchronization, control flow (with synchronization semantics) can ensure that the conflicting data accesses will not be concurrent, but because of scope there is no guarantee that the result will be sequentially consistent. Also, as we will show, certain properties of an SC for HRF model are quite foreign to SC for DRF models – for example, in the models presented in this paper, it is possible for two atomic accesses to race with each other.

That is why we next extend the understanding of race-free to include what we call heterogeneous races involving synchronization of insufficient scope.

## 3 HRF-direct: Basic Scope Synchronization

In the HRF-direct synchronization model, if two threads communicate, they must synchronize using operations of the ***exact same*** scope. If − in some sequentially consistent execution of a program − two conflicting data accesses are

### Formal Definition of HRF-direct

We formally define the HRF-direct model using set relational notation, using the style adopted by Adve and Hill [3].

*Conflict Definitions*

**Ordinary Conflict:** Two operations op1 and op2 conflict iff both are to the same address, at least one is a write, and at least one is an ordinary data operation.

**Synchronization Conflict:** Two synchronization operations op1 and op2 conflict iff both are to the same location, at least one is a write (or a read-modify-write), and are performed with respect to different scopes.

*Definitions for a Sequentially Consistent Candidate Execution*

**Program Order ($\overrightarrow{po}$):** op1 $\overrightarrow{po}$ op2 iff both are from the same work-item or thread and op1 completes before op2.

**Scoped Synchronization Order ($\overrightarrow{so_S}$):** Release rel1 appears before acquire acq2 in $\overrightarrow{so_S}$ iff both are performed with respect to scope S, and rel1 completes before acq2.

**Heterogeneous-happens-before-direct ($\overrightarrow{\text{hhb.d}}$):** The union of the irreflexive transitive closures of all scope synchronization orders with program order:

$$\bigcup_{\forall S \in \mathbb{S}} (\overrightarrow{po} \cup \overrightarrow{so_S})^+$$

where $\mathbb{S}$ is the set of all scopes in an execution. Note that in the above equation, the closure applies only to the inner union and is not applied to the outer union.

**Heterogeneous Race:** A heterogeneous race occurs iff a pair of operations op1 and op2 that are either an ordinary or a synchronization conflict are unordered in $\overrightarrow{\text{hhb.d}}$.

**Heterogeneous-race-free Execution:** An execution is heterogeneous-race-free iff there are no heterogeneous races.

*Program and Model Definitions*

**Heterogeneous-Race-Free Program:** A program is heterogeneous-race-free iff all possible sequentially consistent executions of the program are heterogeneous-race-free.

**Sequential Consistency for Heterogeneous-race-free-direct (HRF-direct):** A system obeys the HRF-direct memory model iff all actual executions of a heterogeneous-race-free program are sequentially consistent.

*Corollary*

**Other Total Orders:** Because any heterogeneous-race-free program is sequentially consistent in HRF-direct, implementations must ensure an apparent total order of all heterogeneous-race-free operations, including these notable subsets:

- *A total order of all synchronization operations with respect to the same scope.*
- *A total order of all the heterogeneous-race-free synchronization operations that could be observed by any single work-item or thread.*

```
For t=0; t < nsteps; t+= group_step_size:
    group_step_size times do:
        <stencil calculation on shared grid>
        BarrierSync(work-group scope)
    BarrierSync(device scope)
    <exchange work-group boundary conditions>
```

**Figure 3. An example race-free HRF-direct algorithm**

performed without being separated by paired synchronization of identical scope, then those accesses form a *heterogeneous race* and the result on an HRF-direct system is undefined. Otherwise, a conforming implementation will guarantee that the execution is sequentially consistent. HRF-direct is a good match for *existing*, highly regular GPGPU workloads and provides considerable hardware implementation flexibility (Section 6).

If a memory model is effective, programmers should be able to easily determine the minimal amount of synchronization that must be performed for correctness. In a system with scoped synchronization, this includes selecting the smallest (i.e., fastest) scope operation possible. In HRF-direct, programmers can follow a simple rule to achieve a correct and fast execution: *use the smallest scope that includes all producers and all consumers of the data being shared*. If, for example, work-items in a work-group are coordinating to share data only within the work-group, then a work-group scope should be used. If, on the other hand, some of that data may be shared beyond that work-group in the future, a larger (e.g., device) scope must be used *by all of the work-items*.

In the sidebar on page 2 we provide a formal definition of HRF-direct that can be used to reason rigorously about program behavior or system optimizations in terms of a well-defined happens-before relation. There are two key differences in HRF-direct compared to a traditional DRF model. First, HRF-direct considers synchronization operations performed at different scopes to be conflicting operations. Second, HRF-direct creates a separate happens-before order relative to each scope. For each scope S1, the S1 happens-before order is the conventional notion of DRF happens-before but where all synchronization with respect to a different scope S2 ≠ S1 is treated as an ordinary operation. An execution contains a race in HRF-direct if any two conflicting operations are unordered in *all* the scope-relative happens-before orders.

For now, we focus on the high-level concepts behind HRF-direct by applying it to an example to show how it can be used to reason about a high-performance heterogeneous application. We will discuss more in-depth details of the model in Section 3.3.

### 3.1 Race-free Example

In Figure 3 we show the skeleton of a simple GPGPU kernel that is synchronized correctly under HRF-direct. The algorithm, loosely modeled after the HotSpot application in the

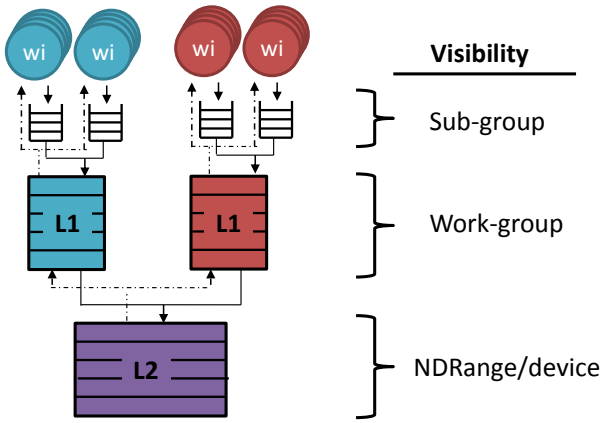


**Figure 4. Example hardware implementation for HRF-direct that closely approximates a GPU architecture.**

Rodinia benchmark suite [11], performs a stencil computation over a series of timesteps. The application uses a small amount of data and computation replication to reduce global synchronization. Rather than globally synchronize between each timestep, the program instead computes several timesteps locally within a work-group before having to exchange boundary conditions with neighboring work- groups. Notably, within the inner loop of Figure 3, work-items communicate only with other work-items inside their work-group and in the outer loop work-items communicate only with other work-items outside their work-group.

Because the communication pattern in the application is highly regular and predictable, it is easy to choose minimal scope for the synchronization points that will be race-free in HRF-direct. The timestep barrier in the inner loop should use work-group scope and the timestep barrier in the outer loop should use device (assuming the CPU is not involved) scope. The programmer can be sure this is race-free because between any work-item communication there is (a) synchronization using (b) scope to which both work-items belong.

While this example may seem simple, this pattern of communication is common in today's GPGPU applications. However, readers familiar with Rodinia or OpenCL / CUDA programming in general will notice two notable simplifications in the pseudocode. First, in the actual applications, the inner loop uses the GPU scratchpad memory ("local memory" in OpenCL and "shared memory" in CUDA) for all work-item communication. Second, the outer "barrier" is actually a series of distinct kernel launches. We believe that both of these characteristics will be less common in future GPGPU applications as global memory caches increase in size and inter-work-group communication becomes more possible. Models like HRF-direct will help ease that transition and ultimately make GPUs more programmable.

### 3.2 Implementation

In Figure 4, we show a baseline memory architecture similar to a modern GPU. It has a two-level cache hierarchy in which each compute unit (which runs a work-group) has a private L1 cache and the device as a whole shares an L2 cache. We assume that all caches use a write-combining strategy, and notably do not use a read-for-ownership coherence protocol. Because the caches write without exclusive permission, they keep track of partial cache block writes to avoid data corruption in the presence of false sharing. For each block, the caches maintain a bitmask of dirty bytes. When a block is evicted to the next level of cache/memory, only those dirty bytes are written back.

In general, we make this example implementation compatible with HRF-direct by (1) on a release event, flushing dirty data from any cache/memory that are not visible to all threads in the scope of synchronization, (2) on an acquire event, invalidating the now-stale data from caches/memories that are not visible to all threads in the scope of synchronization, and (3) completing all atomic operations in the cache/memory corresponding to the scope of the atomic. For example, on a work-group-scope release, the sub-group (i.e., SIMT vector)-private write buffers will be flushed to the L1 because the write buffers are not visible to all work-items in the work-group. On a device-scope acquire, the system would invalidate the acquiring work-item's L1 cache because it is not visible to all work-items on the device. On a device-scope atomic increment, the system would perform the read-modify-write entirely in the L2 cache controller.

This policy results in an SC execution of race-free HRF-direct programs. To see why, recall that HRF-direct requires conflicting accesses (same location, at least one is a write) to be separated by synchronization of the same scope. In the example implementation, that means the conflicting accesses both will have reached (at least) the same level in the memory hierarchy, and the location's most recent update will be returned. The location could not see some other value because that would constitute a race.

While this example implementation is simple, it is also representative of current hardware. Qualitatively, it also clearly shows the performance benefit of scoped synchronization; there is an obvious difference in the cost of a work-group-scope release (flush write buffers, if any) compared to a system-scope release (flush dirty data from L1 and L2 caches). When combined with the previous observation that HRF-direct is easy to use in regular heterogeneous applications, we conclude that HRF-direct is a reasonable and useful model for heterogeneous systems.

### 3.3 Subtleties of Synchronization

In HRF-direct, a program composed entirely of atomic synchronization operations could contain a race – and therefore the execution would be undefined. This is a major divergence from SC for DRF models in which any program composed entirely of synchronizing atomics is race-free. In Figure 5 we show an example all-atomic program that is racey in HRF-direct when the work-items are in different work-groups.

```
__global atomic<int> A = {0};
__global atomic<int> B = {0};
// __global means "not in a scratchpad" in OpenCL

Work-item wi1

  A.store(1, memory_order_seq_cst,
          memory_scope_work_group);
  B.load(memory_order_seq_cst,
         memory_scope_device);

Work-item wi2

      B.store(1, memory_order_seq_cst,
              memory_scope_device);

      // A.load forms a race when
      // workgroup of wi1 ≠ workgroup of wi2
      A.load(memory_order_seq_cst,
             memory_scope_work_group);
```

**Figure 5. Subtleties of synchronization order.**

**This is race-free if work-group of wi1 = work-group of wi2.**
**This is racey if work-group of wi1 ≠ work-group of wi2.**

In general, to avoid confusion about racing synchronization, a best practice when using HRF-direct is to associate a single scope with each atomic or synchronization variable. For example, software could be constructed so that variable A is always used with work-group-scope, variable B with device-scope, etc. If this practice is followed, atomic or synchronization operations will never race with each other (though other types of heterogeneous races can certainly still occur). Future languages may want to provide first-class support for mapping synchronization variables to scopes in this way.

#### 3.3.1 Digging Deep: The Order of Atomics

An HRF-direct implementation must ensure an observable total order among some atomics but not others. In a hierarchical scope layout (like OpenCL's in Figure 1), an HRF-direct implementation is free to reorder two atomics if they are performed with respect to disjoint scopes. If the scopes are disjoint, no work-item or thread in the system could observe the atomic operations without forming a race[5]. Because all attempts to observe an order directly would form a race, an implementation is not bound to produce any reliable order or even ensure atomicity. In the racey version of Figure 5, in which the two work-items belong to different work-groups, an implementation does not need to ensure the two accesses to A are ordered or atomic because the two accesses use disjoint scopes.

However, an implementation must still produce an observable total order (1) among all race-free atomics per-

[5] Unless the atomic operations are ordered as ordinary loads/stores through other, race-free, synchronization

## Formal Definition of HRF-indirect

The structure of HRF-indirect is identical to the HRF-direct definition but with a different happens-before relation:

**Heterogeneous-happens-before-indirect ($\overrightarrow{\text{hhb.i}}$):** The irreflexive transitive closure of all scope synchronization orders and program order:

$$\left(\overrightarrow{po} \cup \bigcup_{\forall S \in \mathbb{S}} \overrightarrow{so_S}\right)^+$$

where $\mathbb{S}$ represents the set of all scopes in an execution.

formed with respect to the same scope and (2) among all race-free atomics performed with respect to overlapping scopes. We illustrate this point in the race-free version of Figure 5, in which both work-items belong to the same work-group. Because the example is race-free, an implementation must ensure a total order between the four accesses to A and B (i.e., A = B = 0 not allowed) even though the accesses to A and B are performed with respect to different (but overlapping) scopes.

Our example implementation in Section 3.2 seeks high performance by exploiting the weaker order of atomics relative to an SC for DRF-compliant system. In that implementation, atomics are completed in the cache or memory associated with target scope (e.g., a work-group scope atomic increment is completed in the L1 cache without obtaining exclusive global permission). This implementation meets the requirements of an HRF-direct system: all race-free atomic accesses will use the same physical resource, ensuring that all work-items will observe race-free accesses in the same order.

### 3.3.2 Function Composition

Scopes complicate generic library functions that synchronize. In HRF-direct, because producers and consumers must use the same scope, it is not good enough for a generic library function to assume the worst and always synchronize with global scope. A future consumer of the data produced by that function may synchronize with a different scope, forming a race. To get around this problem, libraries may need to accept a scope as a parameter. For example, a library might provide a `lock(memory_scope s)` routine that guarantees the caller is synchronized with respect to scope `s` after the lock is acquired.

### 3.4 Limitations

While we have shown HRF-direct is a sufficient and (relatively) easy-to-understand model for programs with highly regular parallelism, it does have limitations that may impede the adoption of forward-looking software. Most notably, the HRF-direct requirement that all threads use the exact same scope could make it difficult to write high-performance synchronization in software with irregular parallelism. In situations when the producer and consumer are not known *a priori*, HRF-direct software will likely be forced to conservatively use the largest scope that includes any work-item that could be a consumer. Often, this will be the much slower device or system scope (see Section 7 for a more in-depth analysis).

```
  __global atomic<int> A = {0};
  __global atomic<int> B = {0};
  __global int X;
// __global means "not in a scratchpad" in OpenCL

Work-item wi1 -- in work-group wgX

  11: X = 1;
  12: A.store(1, memory_order_seq_cst,      // release
              memory_scope_work_group);

Work-item wi2 -- in work-group wgX

   21: while(!A.load(memory_order_seq_cst,  // acquire
                     memory_scope_work_group));
   22: int R2 = X; // R2 will get value 1
   23: B.store(1, memory_order_seq_cst,     // release
               memory_scope_device);

Work-item wi3 -- in work-group wgY (Note! not wgX)

   31: while(!B.load(memory_order_seq_cst,  // acquire
                     memory_scope_device));
   32: int R3 = X;  // R3 will get value 1
```

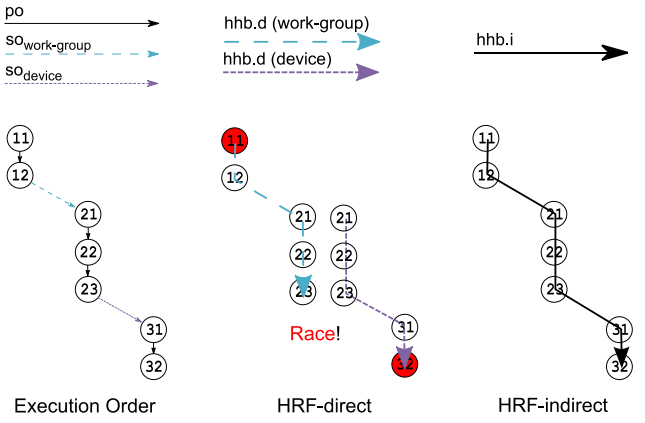


**Figure 6. An example of a racey execution in HRF-direct that is data-race-free in HRF-indirect.**

**In the diagram below the code, we show the orders established in both HRF models (left) and the happens-before order in HRF-direct (middle) and HRF-indirect (right).**

## 4 HRF-indirect: Adding Transitivity

HRF-indirect builds on the HRF-direct model by permitting communication that involves a transitive chain of synchronization that uses different scopes. We show an example of HRF-indirect's scope transitivity in Figure 6 in which work-

item wi1 synchronizes with wi2 in their shared work-group scope and then wi2 synchronizes with wi3 with a different device scope. Note that this is the same execution as the DRF example in Figure 2, but altered so that the two synchronization pairs use different scopes. Notably, each pair of *direct* synchronization (e.g., between wi1 and wi2 using A) must still use the same scope (we discuss relaxing this requirement in Section 5.2).

From the programming standpoint, HRF-indirect has several advantages relative to HRF-direct. First, as Figure 6 shows, in HRF-indirect producers and consumers do not have to be aware of each other, which can expand the types of algorithms practically possible on accelerator architectures. In Section 7 we show how this situation may arise in practice. Second, it gives programmers more flexibility in the choice of scope and could lead to improved performance. If, for example, a producer knows that a work-item outside its work-group will eventually consume its data, but also knows that several steps must first occur within the work-group, then the producer may choose to perform a smaller work-group-scope synchronization and rely on work-group peers to synchronize with the more distant work-items later.

HRF-direct is compatible with HRF-indirect: all programs that are race-free in HRF-direct are also race-free in HRF-indirect. This property opens the possibility of extending existing models to support HRF-indirect in the future; such a change will not break the compatibility of existing well-behaved software.

We provide a formal definition of HRF-indirect in the sidebar on page 2. In HRF-indirect, operations are ordered in the happens-before relation if they can be linked by any transitive combination of synchronization and program order. Figure 6 visually shows the difference between the happens-before relations of HRF-direct and HRF-indirect, and emphasizes the fact that in HRF-direct, order can only be established with respect to a single scope. In Figure 6, we break the happens-before relation of HRF-direct into components attributable to the work-group and device scopes, respectively.

Like HRF-direct, in HRF-indirect synchronization release and acquire operations are only paired if they are from the same scope. As such, it is still possible to write a racey program consisting entirely of atomics. In fact, the same subtleties of synchronization mentioned in Section 3.3 for HRF-direct also apply to HRF-indirect.

### 4.1 Example Implementation

As a testament to practicality of HRF-indirect, the example implementation described in Section 3.2 for HRF-direct works as-is. That simple system, which is a good approximation of existing hardware, is a conservative implementation of HRF-direct. With HRF-indirect, software can better exploit the capabilities of existing hardware. Though, as we discuss in Section 6, HRF-indirect does not allow as many potential optimizations in the future as HRF-direct.

To see why the example implementation is compatible with HRF-indirect, consider Figure 4. When the system performs a release to device scope by flushing the L1 cache, it is not just flushing prior writes performed by that work-item – it is also flushing any prior write in the L1 performed by a different work-item in the same work-group. Therefore, in the implementation, the release actions effectively perform on behalf of the entire scope associated with that cache/memory. That is why the example in Figure 6 works. When wi2 performs the device-scope release and flushes the L1, it is also flushing the prior store to X by wi1 that must be in (at least) the L1 cache because of the prior work-group-scope release.

## 5 Exploring HRF Properties

In this section, we compare the properties of HRF-direct and HRF-indirect to existing academic and commercial memory models. Then we discuss another property that is not present in either HRF variant that future models may consider.

### 5.1 Memory Model Comparisons

#### 5.1.1 DRF

Both HRF-direct and HRF-indirect are compatible with a DRF model. In other words, any program that exclusively uses system-scope synchronization and is free of ordinary data races will be sequentially consistent with either model. Notably, this means that programmers have the option of writing data-race-free programs to begin with for functional correctness, and then adjusting scope later for performance.

#### 5.1.2 OpenCL

The first OpenCL specifications (1.x) have very restrictive memory models that do not support *any* communication outside of a work-group between kernel invocations. OpenCL 2.0 solves this particular issue with a broader definition of atomic operations.

OpenCL 2.0's execution model [28] is extended to support, among other things, fine-grain shared virtual memory (SVM), a single flat address space, and an extended subset of the recently introduced C11-style atomics [7]. The specification also contains a rigorously defined memory model that brings each of these disjoint features together into a single framework.

```
__global atomic<int> A = {0};
__global atomic<int> B = {0};
// __global means "not in a scratchpad" in OpenCL

Work-item wi1 – in workgroup wgX

  A.store(1, memory_order_seq_cst,
          memory_scope_device);
  B.load(memory_order_seq_cst,
         memory_scope_device);

Work-item wi2 – in same workgroup wgX

      B.store(1, memory_order_seq_cst,
              memory_scope_work_group);
      A.load(memory_order_seq_cst,
             memory_scope_work_group);
```

**Figure 7. An example of scope inclusion.**

The OpenCL 2.0 memory model is an SC for HRF model when the sequentially consistent ordering (`memory_order_seq_cst`) is used. OpenCL 2.0 is more restrictive than either HRF-direct or HRF-indirect– for example, in OpenCL 2.0, a program can use either device scope or system scope for atomics but not both within the same kernel. This intermix of different scoped atomics is disallowed even if individual locations are consistently accessed using atomics of the same scope. Should the designers want to relax the model in the future to allow intermixing atomics with different scope, they will need to consider the issues discussed above.

#### 5.1.3 CUDA

Recent versions of CUDA (compute capability ≥ 2.0) support global shared memory and scoped synchronization. More specifically, CUDA supports a `__threadfence` instruction that acts as a memory fence with block (work-group), device (GPU), or system (GPU + CPU) scope. Other instructions with control-flow effects like `__syncthreads` also have scoped synchronization semantics.

Because CUDA uses a different framework to define its memory model, it is hard to make a direct comparison to the HRF models. CUDA does support a form of scope transitivity (e.g., example in Section B.5 of the CUDA Programming Guide [12]). However, because CUDA is not defined axiomatically, it is difficult to say if it has the same properties of scope transitivity defined for HRF-indirect.

#### 5.1.4 Heterogenous System Architecture and Parallel Thread Execution

HRF models are not limited to high-level languages. Low-level intermediate languages like HSAIL [21] and PTX [30] also use synchronization scopes and could benefit from a formal HRF definition. HSAIL is particularly well-suited for the HRF models in this paper because it uses a variant of scoped acquire/release synchronization.

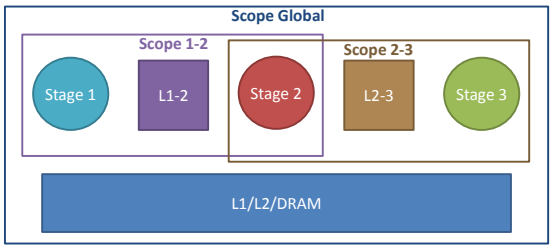


**Figure 8. Architecture and scope layout of a possible programmable pipeline.**

#### 5.1.5 Non-global Memory Types

In our discussion we have disregarded the scratchpad memories that exist in GPUs and that are exposed by both the OpenCL and CUDA programming models. While our models could be extended to include these software-controlled memories, we assume they will be less relevant in the future as heterogeneous software becomes more portable. As evidence for this trend, the more recent OpenACC programming model [32] does not include scratchpads.

### 5.2 Future HRF Models -- Scope Inclusion

Other HRF models are possible that support properties beyond those in HRF-direct/indirect. One such property is what we call *scope inclusion*[6], in which two work-items are allowed to synchronize directly (e.g., non-transitively) with each other using operations of different scope. Informally, scope inclusion would support race-free paired synchronization when the scope of the first is a subset of the scope of the second (i.e., one includes the other). We show an example in Figure 7.

Scope inclusion would require an implementation to ensure that synchronization operations with respect to inclusive scopes are ordered. This requirement is very similar to the requirement already present both in HRF-direct and HRF-indirect (and discussed in Section 3.3.1) – that atomics from overlapping scopes must be ordered. Therefore, we expect any conceivable implementation of HRF-direct or HRF-indirect would also support scope inclusion.

We did not define scope inclusion as part of the model because precisely defining when scope inclusion does and does not apply can be nuanced (as demonstrated by the complexity of Section 3.3.1 describing a similar property). However, given a concise and understandable definition, scope inclusion could be added to either HRF model by altering the definition of a synchronization conflict.

[6] OpenCL 2.0 uses the term "scope inclusion" to mean "the exact same scope." However, the term is defined in such a way that extending to the notion of inclusion described here would be straightforward in the future.

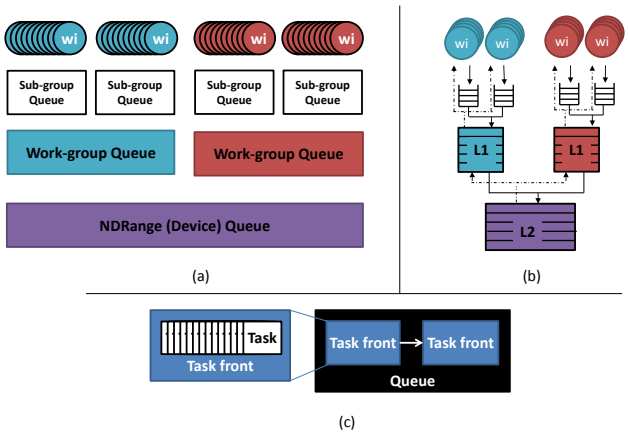


**Figure 9. (a) Task queue hierarchy, (b) the heterogeneous hardware it maps to (same as Figure 4), and (c) queue contents.**

## 6 System Optimizations

While the more permissive HRF-indirect model enables higher performance on current hardware, it may limit some hardware optimizations in the future compared to HRF-direct. To support HRF-direct in a system like the one in Figure 4, on a release, an implementation only needs to flush locations that have been written by the releasing work-item. Other blocks in the cache could remain untouched, potentially reducing the negative effect synchronization has on other work-items sharing those resources.

The same targeted flush optimization would be incorrect on an HRF-indirect-compatible system. Because of the transitivity property, a release in HRF-indirect does not just guarantee visibility of locations touched by the releasing work-item. Rather, it also guarantees visibility of any location touched by another work-item that previously synchronized with the releasing work-item. Thus, while an implementation with more targeted flush operations is possible, it would take considerable design effort to determine which blocks are safe to ignore.

An efficient implementation of HRF-indirect in a non-RFO system relies on a hierarchical memory layout and scopes that do not partially overlap. This is likely to be a feature of foreseeable GPU architectures but could be an issue with future non-hierarchical memory systems. For example, the architecture of the recently proposed Convolution Engine by Qadeer, et al. [34] resembles a programmable pipeline. To reduce the latency of communicating between neighboring pipeline stages, a designer might want to build a cache between the stages that can be controlled with an HRF scope, as shown in Figure 8. While possible, it would likely take considerable engineering effort to avoid flushing the inter-stage caches on all synchronization events, even when an inter-stage scope is used.

## 7 Case Study: Work-sharing Task Runtime

We have previously argued that HRF-direct is suitable for current regular GPGPU programs and that both HRF-direct and HRF-indirect are supported by some current hardware.

Here we seek to show that programmers who confront HRF-indirect's complexity can be rewarded with higher performance than with HRF-direct. To this end, we have developed a work-sharing, task-parallel runtime for GPUs that serves as an example of a workload with irregular parallelism and unpredictable synchronization. This section describes the design of that runtime and discusses how HRF-direct and HRF-indirect affect software design decisions. In Section 8, we will show how those decisions affect performance on a system resembling modern GPU hardware.

We have built a work-sharing runtime whose internal task queue is split into a hierarchy resembling the entities in the OpenCL execution model and, by extension, physical components in GPU hardware. For now, the runtime only supports GPU work-item workers and does not support CPU (host) thread workers. In the future the runtime could be extended to support participation from the CPU. The work-sharing task queue is broken into three levels, namely the device queue, multiple work-group queues, and multiple sub-group queues, as shown in Figure 9(a).

### 7.1 Runtime Description

Applications use the runtime by defining tasks from the perspective of a single, independent worker. Notably, the programmer does not need to consider the OpenCL execution hierarchy, because the runtime scheduler effectively hides those details. The runtime itself is built using the persistent threads [18] execution model, and takes complete control of task scheduling and load balancing by bypassing the hardware scheduler (by monopolizing all execution units).

To map the task-parallel execution model exposed by the runtime onto the OpenCL hierarchy, the runtime collects tasks created by workers into sub-group-sized (e.g., 64) groups called *task-fronts*. As shown in Figure 9c, task-fronts are the objects held in the task queues, and are assigned to ready sub-groups. Each task in a task front will run on a different work-item in the sub-group. To reduce control divergence that could decrease performance, the runtime ensures that all tasks in a task front call the same function. A sub-group will generally wait until a task front is full before dequeueing work except in some corner cases where it may be beneficial to grab a partial task-front and tolerate idle work-items (e.g., at the beginning of execution).

The runtime implements a work-sharing, as opposed to a work-stealing, load-balancing policy. When a sub-group has produced enough tasks to fill a task front, it will first enqueue it onto its own private sub-group queue. Then, the sub-group may decide to donate one or more task fronts from its private queue if the occupancy of either the work-group or device queues is below a minimum threshold. A

**Table 1. Configuration Parameters**

| Parameter | Value |
|---|---|
| # Compute Units | 8 |
| # SIMD Units / Compute Unit | 4 |
| L1 cache | 32Kb, 16-way |
| L2 cache | 2MB, 16-way |

sub-group always donates the oldest task in the queue under the assumption that the oldest task will create the most new work and thus avoid future donations.

### 7.2 HRF-direct/HRF-indirect Runtimes

In the runtime, inter-worker communication occurs at task enqueue/dequeue points. At those points, the runtime must decide which scope to use for synchronization. The decision is complicated by the fact that the worker does not know at enqueue time who the eventual consumer will be. For example, if a worker makes a donation to the work-group queue, it must assume that another worker could later donate the work to the device queue. Thus, the eventual consumer could be anywhere in the system.

This unknown-consumer aspect makes it difficult to efficiently synchronize with the HRF-direct model. Recall that HRF-direct mandates that both producer and consumer use identical scope to synchronize. In a straightforward HRF-direct implementation, all synchronization must use device (NDRange) scope since neither producer nor consumer know each other. There may be ways to avoid an always-global scope policy, but it would complicate the runtime and could require invasive policies such as a restriction on which tasks can be donated from a queue (potentially impacting performance since "oldest first" donation, which has been shown to have optimal performance in some task runtimes [6], would not be straightforward).

In contrast, the transitive property of HRF-indirect makes it possible to perform smaller scope synchronization even when the exact producer/consumer pair is not known *a priori*. Consider the problematic (for HRF-direct) case above in which a task front is donated by a different worker than the one that originally performed the enqueue. When the donation occurs, the donating worker will perform a larger scope synchronization operation and will form a transitive link between the original producer and the eventual consumer, much like the interaction shown in Figure 6. Thus, the execution will be race-free and workers can use smaller scope in the common (non-donating) case.

## 8 Evaluation

In this section, we seek to qualitatively show that there are practical differences between the HRF-direct and HRF-indirect models in current heterogeneous hardware. We do so by evaluating the performance of HRF-direct and HRF-indirect versions of the task runtime previously discussed.

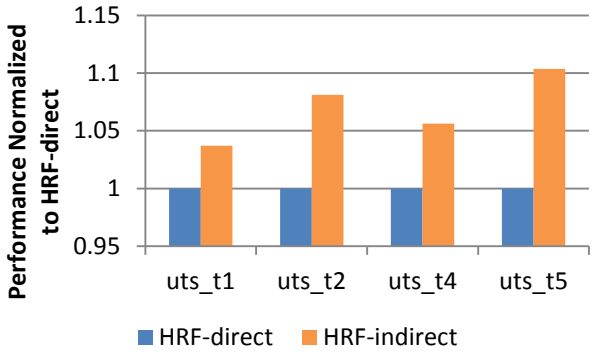


**Figure 10. Performance normalized to HRF-direct.**

### 8.1 Methodology

We wrote the task runtime in OpenCL 2.0, extended to support HRF-indirect in addition to HRF-direct. On top of that, we ported the Unbalanced Tree Search (`uts`) synthetic workload [31]. `uts` represents the traversal of an unbalanced graph whose topology is determined dynamically as the program runs. On visiting a node, the program computes how many children to create using (deterministic) SHA-1 hash computation. Thus, the tasks themselves are short and the number of children varies widely, making the workload a good test of task-parallel runtimes.

We evaluate the runtime using four different input sets to `uts`. The four represent scaled-down versions of the T1, T2, T4, and T5 graphs suggested in the source code release [38], and contain approximately 125K nodes each. We do not include results from the T3 graph because it does not have enough parallelism for GPU execution.

To calculate the performance of the workload, we run it in a version of the gem5 simulator [5] that has been extended with a GPU execution and memory model. We configured the GPU to resemble a modern mid-sized device with eight compute units, with the precise options shown in Table 1. The memory system for the GPU operates like the example system in Section 3.2, in which caches are write-combining and are flushed/invalidated based on the scope of synchronization. We do not implement the potential future hardware optimizations discussed in Section 6.

### 8.2 Results

Figure 10 shows the performance of each input set for the HRF-direct and HRF-indirect versions of the task runtime. In all cases, the HRF-indirect version has higher performance due to the reduction in the number of expensive L1 flush/invalidate operations. The exact speed-ups vary between 3% and 10%, and correlate with the number of donations that occur during the traversal of a particular graph. Generally, more donations (caused by a more unbalanced tree) lead to better relative performance with the HRF-indirect implementation.

### 8.2.1 Limitations and Caveats

We have not implemented the HRF-direct hardware optimizations discussed in Section 6 nor the HRF-direct software improvements discussed in Section 7.2. Doing any one or a combination of these optimizations may alter the results and conclusions discussed above. However, for reasons discussed in Section 7.2, including a sub-optimal task ordering, our intuition tells us it is unlikely to surpass the performance of HRF-indirect in this workload.

## 9 Related Work

Hechtman and Sorin recently argued that GPU systems should implement SC, in part, by showing that the performance of an SC GPU is comparable to one implementing a weaker model [20]. Their analysis, however, assumes a GPU that implements read-for-ownership coherence and does not take interaction with a CPU core into account. In this paper, we assume hardware that better represents current GPUs (e.g., write combining buffers, scoped synchronization, and no read-for-ownership coherence), and propose SC for HRF as a class of models to reason about consistency in similar current and future GPUs.

Other implementations of an HRF-direct/indirect compatible system are also possible. Hechtman et al. proposed a compatible cache hierarchy that will perform better that our basic example implementation in Section 3.2 for programs that make use of fine-grain synchronization [19].

Scoped synchronization is not limited to GPU systems. The Power7 CPU system uses scoped broadcasts in its coherence protocol [22]. Though this scoping is not ex-posed to the programmers, it nonetheless represents the trend toward scoped synchronization in modern hardware.

One could argue that message-passing models like MPI provide scoped consistency by allowing threads to specify senders and receivers explictly [16]. Of course, message-passing models do not use shared memory, and as a result are difficult to use with algorithms involving pointer-based data structures like linked lists. In addition, shared memory programs are easier for compilers to optimize because in MPI compilers must have semantic knowledge of the API to perform effective operation reordering [13].

Recently, there has been an effort in the high-performance community to push programming models that make use of a partitioned global address space (PGAS). These include languages like X10 [10], UPC [8], and Chapel [9]. Like SC for HRF models, these PGAS languages present a single shared address space to all threads. However, not all addresses in that space are treated equally. Some addresses can be accessed only locally while others can be accessed globally but have an affinity or home node. As a result, PGAS programs still explicitly copy data between memory regions for high performance. In contrast, in an SC for HRF model, a particular address is not bound to a home node and there is no need for application threads to copy data explicitly between memory regions. Instead, an SC for HRF model uses scoped synchronization operations to limit the communication overhead between a subset of threads.

There has been recent work on coherence alternatives for shared memory in GPU architectures. Cohesion is a system for distinguishing coherent and incoherent data on GPU accelerators [23]. The incoherent data has to be managed by software with explicit hardware actions like cache flushes. SC for HRF models, on the other hand, abstract away hardware details for programmers and rely on an implementation to manage memory resources.

One downside of both the SC for DRF and SC for HRF models is that racey software has undefined behavior. This can be especially problematic in codes that use intentional (benign) data races or in codes containing unintentional bugs. To address this, Marino, et al. proposed the DRFx model that, in addition to guaranteeing sequential consistency for data-race-free programs, will raise a memory model exception when a racey execution violates sequential consistency [27]. To do so, the authors propose adding SC violation detection hardware similar to conflict detection mechanisms in hardware transactional memory proposals. Either of the HRF variants discussed in this paper could benefit from a similar memory model exception should designers wish to support it. Lucia et al. concurrently proposed conflict exceptions that, like DRFx, raise a memory model exception on a race [25]. However, conflict exceptions are more precise and mandate that the exception be raised at the point of conflicting address.

## 10 Conclusions

We have introduced sequential consistency for heterogeneous-race-free (SC for HRF), a class of memory models to intuitively and robustly define the behavior of scoped synchronization operations in heterogeneous systems. We motivated the need for SC for HRF by showing why the existing understanding of race freedom in homogeneous CPU systems is insufficient when applied to systems using synchronization scopes. We proposed two initial SC for HRF models. HRF-direct represents the capabilities of existing heterogeneous languages and requires synchronizing threads to use the exact same scope. In the forward-looking HRF-indirect model, threads can synchronize transitively using different scopes. Our results show that HRF-indirect may have performance advantages for software with irregular parallelism, but may also restrict some future hardware optimizations.

## Acknowledgements

We thank the anonymous reviewers and Hans Boehm for their insightful comments and feedback. We also thank Marc Orr and Shuai Che for their participation in many constructive discussions.

## References

[1] Adve, S.V. and Boehm, H.-J. 2010. Semantics of shared variables & synchronization a.k.a. memory models.

[2] Adve, S.V. and Gharachorloo, K. 1996. Shared memory consistency models: A tutorial. *Computer*. 29, 12 (1996), 66–76.

[3] Adve, S.V. and Hill, M.D. 1990. Weak ordering—a new definition. *Proceedings of the International Symposium on Computer Architecture* (New York, NY, USA, 1990), 2–14.

[4] AMD, Inc. 2012. Southern Islands series instruction set architecture. Advanced Micro Devices.

[5] Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T. and Sardashti, S. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News*. 39, 2 (2011), 1–7.

[6] Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H. and Zhou, Y. 1995. *Cilk: An efficient multithreaded runtime system*. ACM.

[7] Boehm, H.-J. and Adve, S.V. 2008. Foundations of the C++ concurrency memory model. *International Symposium on Programming Language Design and Implementation (PLDI)* (Tuscon, AZ, Jun. 2008), 68–78.

[8] Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E. and Warren, K. 1999. *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses.

[9] Chamberlain, B.L., Callahan, D. and Zima, H.P. 2007. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*. 21, 3 (2007), 291–312.

[10] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C. and Sarkar, V. 2005. X10: an object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices* (2005), 519–538.

[11] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H. and Skadron, K. 2009. Rodinia: a benchmark suite for heterogeneous computing. *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009* (Oct. 2009), 44 –54.

[12] CUDA 5.5 C programming guide: 2013. *http://docs.nvidia.com/cuda/cuda-c-programming-guide/*. Accessed: 2013-12-19.

[13] Danalis, A., Pollock, L., Swany, M. and Cavazos, J. 2009. MPI-aware compiler optimizations for improving communication-computation overlap. *Proceedings of the 23rd international conference on Supercomputing* (2009), 316–325.

[14] Dubois, M., Scheurich, C. and Briggs, F. 1986. Memory access buffering in multiprocessors. *ISCA '86 Proceedings of the 13th annual international symposium on Computer architecture* (1986), 434–442.

[15] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. and Hennessy, J. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Proceedings of the 17th annual International Symposium on Computer Architecture* (1990), 376–387.

[16] Gropp, W., Lusk, E. and Skjellum, A. 1999. *Using MPI: portable parallel programming with the message passing interface*. MIT press.

[17] Guiady, C., Falsafi, B. and Vijaykumar, T.N. 1999. Is SC+ILP=RC? *Proceedings of the 26th International Symposium on Computer Architecture, 1999* (1999), 162 –171.

[18] Gupta, K., Stuart, J. and Owens, J.D. 2012. A study of persistent threads style GPU programming for GPGPU workloads. *Proceedings of Innovative Parallel Computing (InPar '12)* (May 2012).

[19] Hechtman, B.A., Che, S., Hower, D.R., Tian, Y., Beckmann, B.M., Hill, M.D., Reinhardt, S.K. and Wood, D.A. 2014. QuickRelease: a throughput oriented approach to release consistency on GPUs. *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)* (Orland, FL, Feb. 2014).

[20] Hechtman, B.A. and Sorin, D.J. 2013. Exploring memory consistency for massively-threaded throughput-oriented processors. *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)* (Tel Aviv, Israel, Jun. 2013).

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

[22] Kalla, R., Sinharoy, B., Starke, W.J. and Floyd, M. 2010. Power7: IBM's next-generation server processor. *IEEE Micro*. 30, 2 (2010), 7–15.

[23] Kelm, J.H., Johnson, D.R., Tuohy, W., Lumetta, S.S. and Patel, S.J. 2010. Cohesion: a hybrid memory model for accelerators. *Proceedings of the 37th annual international symposium on Computer architecture* (New York, NY, USA, 2010), 429–440.

[24] Lamport, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*. C-28, 9 (Sep. 1979), 690 –691.

[25] Lucia, B., Ceze, L., Strauss, K., Qadeer, S. and Boehm, H.J. 2010. Conflict exceptions: providing simple concurrent language semantics with precise hardware exceptions. *International Symposium on Computer Architecture (ISCA)* (2010).

[26] Manson, J., Pugh, W. and Adve, S.V. 2005. The Java memory model. *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2005), 378–391.

[27] Marino, D., Singh, A., Millstein, T., Musuvathi, M. and Narayanasamy, S. 2010. DRFX: a simple and efficient memory model for concurrent programming languages. *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2010), 351–362.

[28] Munshi, A. ed. 2013. The OpenCL Specification, Version 2.0 (Provisional). Khronos Group.

[29] Munshi, A., Gaster, B. and Mattson, T.G. 2011. *OpenCL programming guide*. Addison-Wesley Professional.

[30] NVIDIA Corporation 2012. *Parallel Thread Execution ISA Version 3.1*.

[31] Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P. and Tseng, C.-W. 2007. UTS: An unbalanced tree search benchmark. *Languages and Compilers for Parallel Computing*. Springer. 235–250.

[32] OpenACC, Inc 2011. The OpenACC™ Application Programming Interface, Version 1.0.

[33] Owens, S., Sarkar, S. and Sewell, P. 2009. A better x86 memory model: x86-TSO. *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (Berlin, Heidelberg, 2009), 391–407.

[34] Qadeer, W., Hameed, R., Shacham, O., Venkatesan, P., Kozyrakis, C. and Horowitz, M.A. 2013. Convolution engine: balancing efficiency & flexibility in specialized computing.

*Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), 24–35.

[35] Sindhu, P.S., Frailong, J.-M. and Cekleov, M. 1992. Formal specification of memory models. *Scalable Shared Memory Multiprocessors: Proceedings*. (1992), 25.

[36] Sorin, D.J., Hill, M.D. and Wood, D.A. 2011. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*. 6, 3 (2011), 1–212.

[37] Thakkar, S., Gifford, P. and Fielland, G. 1988. The balance multiprocessor system. *IEEE Micro*. 8, 1 (Jan. 1988), 57–69.

[38] UTS source distribution: *http://sourceforge.net/p/uts-benchmark/wiki/Home/*.