Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems

Matthew D. Sinclair Johnathan Alsop Sarita V. Adve
University of Illinois at Urbana-Champaign
{mdsincl2,alsop2,sadve}@illinois.edu

ABSTRACT

An unambiguous and easy-to-understand memory consistency model is crucial for ensuring correct synchronization and guiding future design of heterogeneous systems. In a widely adopted approach, the memory model guarantees sequential consistency (SC) as long as programmers obey certain rules. The popular data-race-free-0 (DRF0) model exemplifies this SC-centric approach by requiring programmers to avoid data races. Recent industry models, however, have extended such SC-centric models to incorporate relaxed atomics. These extensions can improve performance, but are difficult to specify formally and use correctly. This work addresses the impact of relaxed atomics on consistency models for heterogeneous systems in two ways. First, we introduce a new model, *Data-Race-Free-Relaxed* (*DRFrlx*), that extends DRF0 to provide SC-centric semantics for the common use cases of relaxed atomics. Second, we evaluate the performance of relaxed atomics in CPU-GPU systems for these use cases. We find mixed results – for most cases, relaxed atomics provide only a small benefit in execution time, but for some cases, they help significantly (e.g., up to 51% for DRFrlx over DRF0).

CCS CONCEPTS

• **Computing methodologies** → **Shared memory algorithms**; • **Computer systems organization** → **Single instruction, multiple data**; • **Hardware** → **Communication hardware, interfaces and storage**; • **Software and its engineering** → **Consistency**;

KEYWORDS

memory consistency, data-race-free models, relaxed atomics, GPGPU

ACM Reference format:
Matthew D. Sinclair Johnathan Alsop Sarita V. Adve University of Illinois at Urbana-Champaign. 2017. Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 14 pages.
https://doi.org/10.1145/3079856.3080206

1 INTRODUCTION

As the benefits from transistor scaling slow down, future processors will increasingly use parallelism and specialization (heterogeneity) to provide energy-efficient performance growth. Recently, for improved efficiency and programmability, heterogeneous systems have begun to provide a global address space across CPUs and accelerators (primarily GPUs), with a multi-level cache hierarchy with private and shared caches accessing shared data [33, 34]. As a result, coherence protocols and memory consistency models (or memory models) for heterogeneous systems are becoming increasingly important. Recently, the Heterogeneous Systems Architecture (HSA) Foundation and OpenCL 2.0 have adopted a memory model based on the recently proposed Heterogeneous-Race-Free (HRF) model [9, 26, 32, 33, 38].

The HSA, OpenCL, and HRF models are largely influenced by the decades of work on multicore CPU memory models. Programming languages such as C, C++, and Java recently converged around the data-race-free-0 memory model which promises sequential consistency (SC) to data-race-free programs (SC-for-DRF0 or DRF0) [1, 14, 42]. The popularity of DRF0 stems from its SC-centric nature. Programmers can reason with the familiar SC model as long as there are no data races, and the absence of data races allows the system to exploit many optimizations without violating SC.

DRF0 requires programmers to distinguish between data and synchronization accesses – any access that may be involved in a race (in any SC execution) must be explicitly identified as synchronization using the atomic (for C, C++, OpenCL, HSA) or volatile (for Java) declarations. This paper refers to synchronization accesses as atomics. DRF0 allows the hardware and compiler to optimize data accesses, but imposes strict constraints on atomics. Since atomics are relatively infrequent and data races are generally considered to be bugs, DRF0 provides a reasonable balance between performance and programmability.

In practice, however, there are cases where DRF0's constraints on atomics can be relaxed with acceptable results, including some acceptable violations of SC. This motivated the addition of *relaxed atomics* to DRF0 for C++ (and later for other languages) and a departure from SC-centric semantics. Unfortunately, this departure has resulted in one of the most significant challenges in specifying concurrency semantics; despite more than a decade of effort, semantics that are weak enough to accommodate all desired optimizations but strong enough to enable reasonable analysis of programs have remained elusive [3, 10, 14, 15] (Section 1.1).

Furthermore, it is generally acknowledged that relaxed atomics are extremely difficult to use correctly; therefore, it is widely recommended that they be avoided and their use be left to experts [14, 61]. This was reasonable for CPUs since atomics are generally infrequent and SC (non-relaxed) atomics are implemented relatively efficiently, leveraging sophisticated coherence protocols. The situation, however, has been different for accelerators, exemplified by GPUs. Given their focus on simplicity, current GPUs implement consistency through heavyweight coherence actions on conventional SC atomics (Section 2.1), making such atomics far more expensive than on CPUs and the potentially more lightweight relaxed atomics more tempting.

To demonstrate the benefits of using relaxed atomics in existing GPUs, we identified several GPU applications that use relaxed atomics (Section 4.4) and evaluated them on an NVIDIA GeForce

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080206

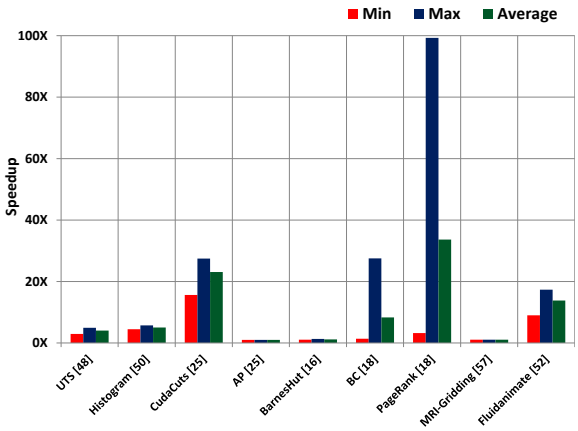


Figure 1: Relaxed atomics speedup on a discrete GPU.

GTX680. Figure 1 shows the speedup of using relaxed atomics instead of SC atomics for the 9 applications [16, 18, 25, 48, 50, 52, 57] with the highest percentage of atomics (as determined from dynamic instruction profiling). Although relaxed atomics do not affect performance for some benchmarks, other benchmarks see huge benefits (e.g., up to a 99X speedup for PageRank), motivating the need for reasonable semantics for relaxed atomics. The main results of our paper (Section 6) show that for future integrated CPU-GPU systems with coherent caches and a global address space (with arguably faster synchronization than discrete systems), the benefits are less dramatic, but high enough that relaxed atomics will still be tempting.

This paper makes two broad contributions. First, we develop new SC-centric semantics for relaxed atomics using a distinct approach. Second, we provide, to our knowledge, the first quantitative evaluation of relaxed atomics on integrated CPU-GPU systems.

1.1 Current Semantics with Relaxed Atomics

The key difficulty underlying efforts to formalize relaxed atomics has been the requirement to prohibit executions that generate values from "out-of-thin-air" due to self-satisfying speculations within causal loops [14, 42]. Much has been written about the many subtleties of this problem [3, 9, 10, 14, 15, 36, 49]; for lack of space, we accept its difficulty and only discuss the state-of-the-art for dealing with it.

The Java memory model was the first to struggle with the out-of-thin-air issue. It attempted to define semantics for programs with data races that were weak enough to enable all desirable optimizations but strong enough to preserve security considerations by minimally prohibiting out-of-thin-air values. The resulting model is extremely complex [42], was later discovered to have a flaw that unintentionally precluded key optimizations, and remains unfixed [62].

C++11 is different from Java in that it was deemed reasonable to give programs with data races undefined semantics (C++ gives undefined semantics for other situations as well). However, it ran into Java-like problems when attempting to formalize semantics for relaxed atomics that were purported to avoid the thin-air problem while enabling desirable optimizations for such accesses. Unfortunately, a fatal flaw was again discovered in the C++11 formalism that remains to be fixed [12, 14]. C++14 therefore just gives informal wording that systems should not produce out-of-thin-air values [13].

Boehm and Demsky have proposed a system restriction to deal with the problem, but it has not yet been accepted by vendors [15]. Recently, two models have been proposed that claim to enable all desired optimizations and prevent out-of-thin-air values [36, 49]; however, they are based on complex theories such as event structures and promises, which seem difficult for most programmers.

In summary, all current approaches to formalize relaxed atomics are acknowledged to have significant limitations. Furthermore, all *require giving up the familiar interface of SC, even if there is a single relaxed atomic in the program, including one buried in invisible library code.* We discuss related work further in Section 7.

1.2 Our Approach for Semantics

Prior approaches focused on defining a system that enables desirable optimizations for relaxed atomics (specifically, reordering relaxed atomics with respect to each other and data accesses) without allowing "out-of-thin-air" values. So far, this approach has failed because what constitutes "out-of-thin-air" has been difficult to pin down and arbitrary accesses may be distinguished as relaxed atomics.

We take a different approach motivated by how we see developers wanting to use relaxed atomics. Specifically we ask the following questions. *What are the common uses of relaxed atomics? Can we characterize these uses in terms of their properties in SC-centric executions? Can we then express the model as ensuring SC-centric behavior for programs that use relaxed atomics only for the specified use cases (i.e., only if the specified properties are obeyed by the program)?* This is precisely the approach that led to the programmer-centric data-race-free class of models [1, 2]. By stating a priori a set of requirements for accesses that can be distinguished as relaxed atomics, we reduce the scope of the problem and make it easier to find a reasonable solution.

Comparing again to the approach of the original DRF models [1, 4], that work examined the optimizations that were being proposed by the "hardware-centric" models of the day (e.g., weak ordering [24], processor consistency [29], release consistency [28], etc.) and determined how to characterize memory accesses where such optimizations would be safe (i.e., not violate SC). Thus, a key insight of DRF0 was that memory accesses not involved in a race, informally referred to as data accesses, could be reordered without violating SC. Later versions discovered other characterizations that led to more optimizations; e.g., the DRF1 model characterized paired vs. unpaired atomics where unpaired atomics did not require any ordering constraints relative to data accesses [4].

To identify use cases for relaxed atomics, we reached out to vendors, developers, and researchers active in this area. We developed a new model, *Data-Race-Free-Relaxed* or *DRFrlx*, that captures these use cases within an SC-centric form. We discovered five use cases.

(1) *Unpaired atomics*: Several relaxed atomics were the unpaired atomics already characterized by DRF1 [4], which is already SC-centric (Section 2.3).
(2) *Commutative atomics*: These relaxed atomics incurred racy interactions only using operations that are commutative. They required a minor adjustment to the definition of SC to accommodate standard relaxed atomic optimizations within an SC-centric framework.
(3) *Non-ordering atomics*: These atomics are involved in racy interactions, but these interactions are never responsible for creating an order between other accesses. Again, relaxed atomics style optimizations can be performed on such accesses without violations of SC.
(4) *Quantum atomics*: Some uses of relaxed atomics truly violate SC.

Programmers justify such atomics as being truly robust and resilient to a large range of approximate (non-SC) values (e.g., split counters [44]). We call such cases quantum atomics and explicitly exploit the intuition that their values are resilient – we require programmers to reason about correctness given that a quantum load may return (almost) any value. To facilitate this, we define a quantum-equivalent program that (logically) replaces quantum accesses with functions returning random values and require SC semantics for such programs (with some additional properties). This may seem bizarre at first; however, these uses of relaxed atomics have been justified in the past as being resilient to many bizarre outcomes, we simply make that expectation explicit, and only for this sub-class of relaxed atomics. Our expectation is that by clearly stating this requirement, the use of such atomics will be restricted to scenarios where such analysis is reasonable; e.g., where a quantum atomic cannot affect the address of a reference or lead to intuitively impossible control flow.
(5) *Speculative atomics*: To avoid the high overhead of synchronizing, some applications (e.g., seqlocks [11]) speculatively read shared data to enable concurrent readers, without proper synchronization. If a write occurs concurrently, the speculative reads are discarded. Even though the speculative reads may produce inconsistent, non-SC values, these values do not affect the final result. We call such accesses speculative atomics and provide SC-centric semantics for them by effectively adjusting the definition of SC to ignore accesses that do not affect the final result.

In summary, like other DRF models, DRFrlx is specified as a contract between the programmer and the system. It requires that all atomics be distinguished as SC atomics or one of the above relaxed atomics (which must obey the above properties). In return, the system will appear SC (for that program or its quantum-equivalent program). Although DRF0/1 are simpler than DRFrlx, in practice their implementations in modern programming languages are made complicated by the addition of relaxed atomics. DRFrlx provides the same semantics as DRF0/1 when relaxed atomics are not used and simpler semantics for relaxed atomics than the state-of-the-art.

We do not claim that our approach covers every possible use case of relaxed atomics. Further, we focus on the `memory_order_relaxed` version of relaxed atomics as defined by C++. We did not explore other relaxed orderings such as `memory_order_acquire` and `memory_order_release` (briefly discussed in Section 7). Instead we cover all common use cases of `memory_order_relaxed` with reasonable-to-use semantics. In particular, a relaxed atomic within a library function of a legal DRFrlx program does not require a user to understand the function's implementation as long as the library writer can convey the expected pre- and post-conditions for SC executions of the (quantum-equivalent) program.

1.3 Evaluation

Although DRFrlx also applies to multicore CPUs, we evaluate DRFrlx for GPU based systems since, as discussed above, heavyweight GPU actions on atomics make relaxed atomics attractive. To determine if the complexity of relaxed atomics is worthwhile for CPU-GPU systems, we created benchmarks based on the use cases we gathered and identified applications in standard benchmark suites that use relaxed atomics. Then we analyzed all the microbenchmarks and benchmarks for the DRF0, DRF1, and DRFrlx memory models and the conventional GPU (Section 2.1) and DeNovo (2.2) coherence protocols. We do not compare to HRF (discussed further in Section 7) because only one application (UTS) and one microbenchmark (Flags) could benefit from HRF's locally scoped synchronizations.

Our evaluation shows mixed results for the effectiveness of DRF1 and DRFrlx. For the microbenchmarks, DRF1 and DRFrlx provide only small benefit (on average, 6% execution time reduction for GPU and 10% for DeNovo). For two applications (BC and PageRank), the benefits of DRF1 were significant – depending on the input, DRF1 reduces execution time by up to 53% for DeNovo and 49% for GPU coherence and improves energy by increasing reuse. Moreover, DRFrlx provides additional benefits over DRF1 for both – up to 29% for DeNovo and 37% for GPU coherence. Comparing the interaction between the different protocols and consistency models, we find that (as shown in past work), DeNovo improves performance relative to GPU or is comparable for DRF0 (for all but 3 use cases). For DRF1 and DRFrlx, the gap between DeNovo and GPU coherence stays roughly constant. On average, compared to GPU coherence, DeNovo reduces execution time by 14%, 14%, and 12% and energy by 16%, 18%, 18% for DRF0, DRF1, and DRFrlx, respectively.

2 BACKGROUND

2.1 Modern GPU Coherence

In conventional GPU coherence protocols, synchronization happens infrequently and at a coarse granularity. As a result, GPUs use simple, software-driven coherence protocols that rely on data-race-freedom, invalidate the entire cache on paired synchronization reads, write-through all dirty data to the shared last level cache (LLC) on paired synchronization writes, and require all atomics to execute at the LLC (e.g., the L2). While this scheme provides high performance for conventional GPU applications, it is sub-optimal for emerging applications with fine-grained synchronization. To mitigate some inefficiencies, the HRF memory model introduced scoped synchronization for GPUs, but has been shown to be insufficient and unnecessarily complex [7, 53] (discussed further in Section 7).

2.2 The DeNovo Coherence Protocol

Previous work has demonstrated that the DeNovo coherence protocol is a good fit for heterogeneous CPU-GPU systems because it provides high performance for a wide variety of applications without the complexities of scoped synchronization [53]. DeNovo is a hybrid of both GPU-style and ownership-based (e.g., MESI) coherence protocols. Like ownership-based protocols, DeNovo obtains ownership for stores and uses writeback caches. Like GPU-style coherence protocols, DeNovo also exploits data-race-freedom to do reader-initiated self-invalidations. In contrast with GPU-style coherence which performs atomics at the LLC, DeNovo obtains ownership for all atomics at the L1, exploiting reuse for atomics.

2.3 DRF1 Consistency Model

The DRF1 memory model removes some ordering constraints from DRF0 by distinguishing paired synchronization read-write atomics from unpaired atomics that do not order data operations [4]. It allows unpaired atomics to be reordered with respect to data operations without violating SC for DRF1 programs (defined below).

2.3.1 *Terminology.*

We use the following terminology throughout the rest of our paper [4]. An *operation* is a memory access that either reads a memory

location (a load) or modifies a memory location (a store). Two memory operations *conflict* if they access the same memory location and at least one of them is a store. Two memory operations, *op1* and *op2*, are ordered by *program order* ($op1 \xrightarrow{\textbf{po}} op2$) if and only if both are from the same thread and *op1* is ordered before *op2* by the program text. An execution is *sequentially consistent (SC)* if there exists a total order, *T*, on all its memory operations such that (i) *T* is consistent with *program order* and (ii) a load *L* returns the value of the last store *S* to the same location ordered before *L* by *T*. We refer to *T* as an *SC total order* for the execution.[1]

2.3.2 Formal Definition of DRF1.

All memory operations are distinguished to the system as either data or atomic. An atomic operation is distinguished as either paired or unpaired.[2]

Definitions for an SC Execution with *SC total order T*:

Synchronization Order 1 ($\xrightarrow{\textbf{so1}}$): Let **X** and **Y** be two memory operations in an execution. $\mathbf{X} \xrightarrow{\textbf{so1}} \mathbf{Y}$ if and only if **X** and **Y** conflict, **X** is a paired synchronization write, **Y** is a paired synchronization read, and **X** is ordered before **Y** in the *SC total order*.

Happens-before-1 ($\xrightarrow{\textbf{hb1}}$): The happens-before-1 relation is defined on the memory operations of an execution as the irreflexive transitive closure of po and so1: $hb1 = (po \cup so1)+$.

Race: Two operations **X** and **Y** in an execution form a race (under DRF1) if and only if **X** and **Y** are from different threads, they conflict with each other, and they are not ordered by happens-before-1.

Data Race: Two operations **X** and **Y** form a data race (under DRF1) if and only if they form a race and at least one of them is distinguished as a data operation.

Program and Model Definitions:

DRF1 Program: A program is DRF1 if and only if for every SC execution of the program, all operations can be distinguished by the system as either data, paired atomic, or unpaired atomic, and there are no data races (under DRF1) in the execution.

DRF1 Model: A system obeys the DRF1 memory model if and only if the result of every execution of a DRF1 program on the system is the result of an SC execution of the program.

Optimizations: In addition to allowing all of the optimizations of DRF0, DRF1 also allows unpaired atomics to be reordered with respect to data operations, without violating SC for DRF1 programs.

Mechanism for distinguishing memory operations:

Data-race-free-1 requires that data operations can be distinguished from atomic operations, and that paired atomics can be distinguished from unpaired atomics. We reuse existing C++ support to distinguish data and atomic operations, and we add a new annotation to distinguish paired and unpaired operations, similar to how C++ distinguishes SC atomics and relaxed atomics [54].

3 RELAXED ATOMIC USE CASES AND DRF-RELAXED MODEL

We collected examples of how developers use relaxed atomics and categorized them in Table 1 based on what type of race occurs in the program: unpaired, commutative, non-ordering, quantum, or speculative. Although our sources contain additional examples that use relaxed atomics, we do not discuss them because they are similar to our examples. Based on these use cases, we introduce DRFrlx, which extends DRF0 and DRF1 [2] to allow certain relaxed atomics to be reordered without compromising SC-centric semantics.

Relaxed Atomic Category	Application
Unpaired (Section 3.1.1)	Work Queue [26]
Commutative (Section 3.2.1)	Event Counter [14, 17, 50, 61]
Non-Ordering (Section 3.3.1)	Flags [61]
Quantum (Section 3.4.1)	Split Counter [44], Reference Counter [46, 61]
Speculative (Section 3.5.1)	Seqlocks [11]

Table 1: GPU relaxed atomic use cases.

```
struct Task;
struct MsgQueue {
  atomic<int> _occupancy = 0;

  Task * dequeue() {
    if (_occupancy.atomic_load(mem_order_seq_cst) == 0) {
      return NULL;
    } else { ... }
  }
  int occupancy() {
    return _occupancy.atomic_load(mem_order_relaxed);
  }
  ...
} globalQueue;

// Thread t1 (service thread):
  void periodicCheck() {
    if (globalQueue.occupancy() > 0) {
      Task * t = globalQueue.dequeue();
      if (t != NULL)
        t.execute();
    }
  }
```

Listing 1: Work Queue example [26].

3.1 Unpaired Atomics

3.1.1 Unpaired Atomics Use Case.

Work Queue [26]: In Listing 1, a service thread and client thread use a shared work queue to communicate.[3] The service thread periodically checks whether the client thread has requested service from it by reading from an incoming message queue. When there are no new messages (the common case), nothing needs to be done and the service thread continues to do other work (on other data, not shown). Although the occupancy checks occur frequently, the service threads' atomics only need to order data when the queue is not empty.

If this example were constrained to using paired (i.e., SC) atomics, then every occupancy check must invalidate the entire L1 cache with current GPU protocols. In the common case of an empty queue, this invalidation is unnecessary and precludes data reuse. Moreover, Work Queue can use relaxed atomics in `occupancy` without violating SC, because all memory accesses will be ordered by the SC atomic in `dequeue`. By using an unpaired atomic for the occupancy check, DRF1 removes the need to invalidate the cache when the queue is empty, enables unpaired operations to be reordered with respect to data operations, and provides benefits similar to relaxed atomics.[4]

[1] For brevity, we refer the reader to [2] for formal definitions of several intuitive notions. Informally, an *execution* must obey correctness requirements for a single core. To accommodate *read-modify-writes* (RMW), the read (load) and write (store) of a RMW must appear together in an SC total order.

[2] Paired atomics are the equivalent of SC atomics in the C and C++ models [14].

[3] All listings use C/C++ convention – mem_order_seq_cst and mem_order_relaxed identify SC and relaxed atomics, respectively.

[4] If Work Queue uses multiple occupancy queues, then relaxed atomics could potentially violate SC. However, since these accesses are amenable to approximation, and the

```
atomic<int> count[NUM_BINS]; // all bins initialized to 0

// Threads 1..N:
threadNum = ...
chunkSize = ...
baseLoc = (threadNum * chunkSize);
...
for (i = 0; i < chunkSize; ++i) {
  binNum = data[baseLoc + i] % NUM_BINS;
  count[binNum].atomic_inc(mem_order_relaxed);
}
...

// Main Thread:
main() {
  launch_workers(); // launch worker threads
  ...
  join_workers();
  for (i = 0; i < NUM_BINS; ++i) {
    int r1 = count[i].atomic_load(mem_order_relaxed);
    ... // uses r1
  }
}
```

Listing 2: Event counters example [14, 17, 50, 61].

DRF1 provides most of the benefits of relaxed atomics for **Work Queue** by removing the ordering constraint between data and unpaired atomics (while preserving SC); however, unlike relaxed atomics, DRF1 constrains unpaired atomics to respect program order with respect to other unpaired atomics. New classes of relaxed atomics discussed in the rest of this section relax this constraint.

3.2 Commutative Atomics

3.2.1 Commutative Atomics Use Case.

Event Counter [14, 17, 50, 61]: In event counters, such as histograms, multiple worker threads concurrently increment shared global counters, as illustrated in Listing 2. Since these increments form a race, they must be distinguished as atomic. Straightforward DRF0/1 implementations would serialize program ordered increments and, for current GPU protocols, invalidate the L1 cache, and flush the store buffer. On inspection, however, one can reason that reordering the increments produces acceptable results; therefore, common uses distinguish the increments as relaxed atomics.

3.2.2 Commutative Atomics Informal Intuition.

We make the following key observations that enable us to formalize the intuition behind the safe reordering of the increments in Listing 2: (i) racing increments in an execution of Listing 2 are commutative and give the same result regardless of their order of execution, (ii) the values they load are not used elsewhere (and so need not be considered as part of the result of the execution), and (iii) the final incremented value is loaded only after another paired synchronization interaction (a barrier from the join in the listing). We refer to atomics with the above properties as *commutative atomics*, formalized below.

We can now reason that the execution order of racy commutative atomics does not impact the final result of the execution and cannot be used to infer the ordering of other operations in the execution. Further, the load of the final value after all the racy, conflicting commutative atomics is always separated by paired (SC) synchronization; therefore, the load does not rely on the ordering of commutative atomics (with respect to other relaxed atomics) for its correct value. Thus, reordering commutative atomics with respect to other program ordered relaxed atomics (unpaired, commutative, and others discussed later) does not violate SC.

The above reasoning uses a slight departure from the conventional notion of what constitutes the "result" of an (SC) execution. Much literature considers the value returned by each load to be part of the result. We define the result to be the memory state at the end of the (SC) execution.[5] Thus, we can ignore the values of reads that do not affect the final memory state when considering if an execution is SC.

3.2.3 DRFrlx Formal Definition (Version 1).

Since DRFrlx extends DRF1, we only list the components of DRFrlx version 1 that differ from DRF1. All memory operations need to be identified as data, paired, unpaired, or *commutative*.

Definitions for an SC Execution:

Result of an execution: The memory state at the end of the execution.

Commutativity: Two stores or RMWs to a single memory location *M* are *commutative with respect to each other* if they can be performed in any order and yield the same value for *M*.

Commutative Race: Two operations **X** and **Y** form a commutative race if and only if:

(1) **X** and **Y** form a race,
(2) at least one of **X** or **Y** is distinguished as a commutative operation, and
(3) **X** and **Y** are not commutative with respect to each other or the value loaded by either operation is used by another instruction in its thread.

Program and Model Definitions:

DRFrlx Program: A program is DRFrlx if and only if for every SC execution of the program:

- all operations can be identified by the system as either data or as paired, unpaired, or commutative atomics, and
- there are no data races or commutative races in the execution.

DRFrlx Model: A system obeys the DRFrlx memory model if and only if the result of every execution of a DRFrlx program on the system is the result of an SC execution of the program.

3.3 Non-Ordering Atomics

3.3.1 Non-Ordering Atomics Use Case.

Flags [61]: Listing 3 uses shared global flags to communicate between threads. Worker threads repeatedly loop until the `stop` flag is set. Within the loop, if certain conditions are met, a worker sets the `dirty` flag to signify something has been accessed that the main thread needs to clean up later (`cleanup_dirty_stuff`). Once the main thread has set `stop`, the workers exit. A global barrier (`join_workers`) ensures that all worker threads exit before the main thread loads the `dirty` flag.

Both `dirty` and `stop` must be distinguished as atomics. The stores to `dirty` can be distinguished as commutative since they obey

queues' values will be double-checked by the dequeue function, we can retain SC-centric semantics by distinguishing these accesses as quantum atomics (Section 3.4).

[5] For brevity, our formalism assumes finite SC executions. We can handle infinite executions as in [2] – we assume that any prefix of an SC total order is finite and consider the memory state at the end of appropriate finite prefixes as the results. For simplicity, we also ignore external outputs in our definition of result; again, this can be easily incorporated similar to [2] and does not affect our model specifications.

```
atomic<bool> dirty = false, stop = false;

// Threads 1..N:
...
while (!stop.atomic_load(mem_order_relaxed)) {
  if (...) {
    dirty.atomic_store(true, mem_order_relaxed);
    ...
  }
  ...
}

// Main Thread:
main() {
  launch_workers(); // launch threads 1..N
  ...
  stop.atomic_store(true, mem_order_relaxed);
  join_workers();
  if (dirty.atomic_load(mem_order_relaxed))
    cleanup_dirty_stuff();
}
```

Listing 3: Flags example [61].

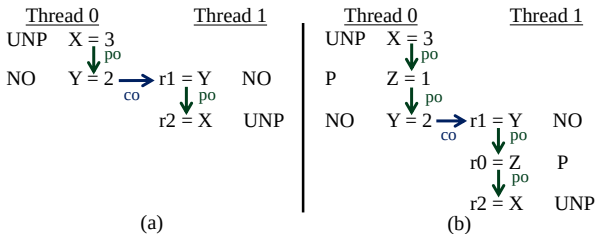


Figure 2: Executions with program/conflict graphs and ordering paths, (a) with a non-ordering race and (b) without a non-ordering race. UNP = unpaired, NO = non-ordering, P = paired.

all the necessary requirements. Before the barrier, `stop` is simultaneously accessed by all threads without any intervening paired atomic. We can informally reason that making the operations to `stop` relaxed will not violate SC for this code. Similarly, we can also reason that the load of `dirty` after the global barrier can also be relaxed.

3.3.2 Non-Ordering Atomics Informal Intuition.

The key intuition behind why the operations to `stop` and `dirty` can be relaxed is that they are not being used to order any other operations – the global barrier orders any conflicting operations that need to be ordered. Thus, reordering the operations to `stop` and `dirty` with respect to other relaxed operations will not violate SC.

To exploit the above intuition, we formalize what it means for a pair of conflicting (racing) operations to "not order" other operations (using formalism developed in [2]), and call such operations *non-ordering* atomics. We use the notion of a program/conflict graph, which captures program order and the execution order of conflicting operations in an execution. For SC, this graph must be acyclic. That is, if there is a path in this graph from an operation **X** to a conflicting operation **Y**, then **X** must execute before **Y** to prevent a cycle. In general, there can be many such "ordering paths" from **X** to **Y**. As long as the system guarantees that one such path is enforced, a cycle will be avoided – operations on other paths between **X** and **Y** may be reordered. Thus, non-ordering operations are those that either don't occur on ordering paths, or are absolved of ordering responsibilities because there is always another path that enforces the ordering. We refer to the latter path as a *valid path*. These concepts are formalized next and illustrated in Figure 2.

3.3.3 DRFrlx Formal Definition (Version 2).

For simplicity, we only show the new DRFrlx components and the components that are modified from Section 3.2.3 to support non-ordering atomics. In version 2 of DRFrlx, all memory operations must be distinguished as data, paired, unpaired, commutative, or *non-ordering*.

Definitions for an SC Execution with *SC total order T*:

Conflict Order ($\xrightarrow{co}$): **X** $\xrightarrow{co}$ **Y** if and only if **X** and **Y** conflict and **X** is ordered before **Y** in *T*.

Program/Conflict Graph and a Path [2]: The **program/conflict graph** of an execution is a directed graph where the vertices are the (dynamic) operations of the execution and the edges represent program order and conflict order. Below all paths refer to paths in the program/conflict graph.

Ordering Path [2]: A path from **X** to **Y** is called an **ordering path** if it has at least one program order edge and **X** and **Y** conflict.

Valid Path [2]: An ordering path is **valid** if all its edges are: (1) $\xrightarrow{hb1}$, or (2) between atomic accesses to the same address, or (3) between paired and/or unpaired accesses.

Non-ordering Race: Two operations, **X** and **Y** form a non-ordering race if and only if:

(1) **X** and **Y** form a race, both are distinguished as atomics, and at least one of them is distinguished as a non-ordering atomic, and
(2) **X** $\xrightarrow{co}$ **Y** is on an ordering path from **A** to **B**, but there is no valid path from **A** to **B**.

Figure 2 shows two example executions with their program/conflict graphs and ordering paths. In Figure 2(a), there is only one ordering path between the conflicting operations on X: X=3 $\xrightarrow{po}$ Y=2 $\xrightarrow{co}$ r1=Y $\xrightarrow{po}$ r2=X. Since this path contains a non-ordering atomic, a non-ordering race occurs. Figure 2(b) adds a new ordering path: X=3 $\xrightarrow{po}$ Z=1 $\xrightarrow{co}$ r0=Z $\xrightarrow{po}$ r2=X. Since the operations on Z are paired, this forms a valid path between the operations on X and there is no longer a non-ordering race in this execution.

Program and Model Definitions:

DRFrlx Program: A program is DRFrlx if and only if for every SC execution of the program:

- all operations can be distinguished by the system as either data or as paired, unpaired, commutative, or non-ordering atomics, and
- there are no data races, commutative races, or non-ordering races in the execution.

3.4 Quantum Atomics

3.4.1 Quantum Atomics Use Case.

So far, the relaxed atomics use cases do not violate SC. However, in some situations, SC violations may be tolerable. Two use cases where this occurs are split counter, described next, and reference counter (Section 3.4.4).

Split Counter [44]: In Listing 4, some threads update their counter while other threads load the current partial sum of all counters, all without adequate synchronization to preserve mutual exclusion. Since multiple threads can concurrently load and store the counters in `myCount`, the operations form races and need to be distinguished as atomics. Commutative atomics may not be used here because the

```
atomic<unsigned long> myCount[NUM_THREADS];
add_split_counter(v, tID) {
  val = myCount[tID].atomic_load(mem_order_relaxed);
  newVal = val + v;
  myCount[tID].atomic_store(newVal, mem_order_relaxed);
}
read_split_counter(tID) {
  sum = 0;
  for (i = 0; i < NUM_THREADS; ++i) {
    loc = ((tID + i) % NUM_THREADS);
    sum += myCount[loc].atomic_load(mem_order_relaxed);
  }
  return sum;
}

add_split_counter(1, 0); // Thread 0
r1 = read_split_counter(1); // Thread 1
add_split_counter(2, 2); // Thread 2
r2 = read_split_counter(3); // Thread 3
```

Listing 4: Split counters example [44].

return value of a racing operation is observed by other instructions in the thread. Non-ordering atomics may not be used because these operations form unique ordering paths between other racing operations. More fundamentally, relaxing these atomics can cause non-SC behavior. Even so, relaxed atomics are often used for split counter operations because developers are willing to trade off SC semantics (and precise partial sums) for improved performance [44]. By allowing atomics in `read_split_counter` to be reordered with both data and other atomics, Split Counter provides a fast, reasonable approximation of the current partial sum.

3.4.2 Quantum Atomics Informal Intuition.

To obtain the performance benefits of relaxed atomics in Split Counter with SC-centric semantics, we define another class of relaxed atomics: *quantum atomics*. Quantum atomics can be reordered with respect to all relaxed atomics (and data). To isolate the non-SC behavior that may result from quantum relaxation, we conceptually build a new program where each quantum load returns a random, approximate value and each quantum store stores a random, approximate value. The transformed program must obey the appropriate race-free properties and appear SC. This transformation isolates the parts of the application that are not dependent on the quantum atomics from the parts that are dependent on it, thereby allowing the reasoning about the latter part in terms of SC. We refer to this transformation as the *quantum transformation*, the transformed program as the *quantum-equivalent program*, and use the term SC-centric to refer to models that provide SC semantics but only to quantum-equivalent programs (with the appropriate race-free properties, formalized below).

When inspecting a quantum-equivalent program for illegal races, quantum accesses are only allowed to race with other quantum accesses. In this way, quantum differs from other types of atomics, which can safely upgrade to a stronger atomic type without introducing new races. The reason quantum accesses may not race with stronger atomic types is because the presence of a racy non-quantum access imposes constraints on the possible intermediate values of the target data in a quantum-equivalent program. These constraints may be inconsistent with the (possibly non-SC) behavior of the original program on a compliant DRFrlx system (defined in Section 3.7).

To provide some constraint on the values of quantum operations, we impose happens-before consistency and per-location SC (sometimes referred to as cache coherence) on these operations (similar to relaxed atomics in C/C++). However, these constraints are post facto – programmers must still commit to reasoning about race-free properties and SC executions only with the quantum-equivalent programs. While it may appear bizarre and against the grain of SC to require the programmer to reason about paths taken for random values for a quantum load, it directly exploits the fact that the reason programmers want to use relaxed loads in Split Counter is that they are amenable to imprecision.

3.4.3 DRFrlx Formal Definition (Version 3).

We only show the new components of DRFrlx and the components that are modified from Section 3.3.3. All memory operations must be distinguished as data, paired, unpaired, commutative, non-ordering, or *quantum*.

Definitions for an SC Execution:

Happens-Before Consistency: A load **L** must always return the value of a store **S** to the same memory location *M* in the execution. It must not be the case that **L** $\xrightarrow{\textbf{hb1}}$ **S** or that there exists another store **S'** to *M* such that **S** $\xrightarrow{\textbf{hb1}}$ **S'** $\xrightarrow{\textbf{hb1}}$ **L**.

Per-Location SC: There is a total order, T_{loc}, on all operations to a given memory location such that T_{loc} is consistent with happens-before-1, and that a load **L** returns the value of the last store to this location before **L** by T_{loc}.

Quantum-Equivalent Program: We generate a quantum-equivalent program *Pq* from a program *P* as follows. Each quantum atomic load *ri = Y;* in *P* is replaced with a call to a conceptual random function *ri = random();* in *Pq*. Similarly, each quantum atomic store *Y = rj* is replaced with a quantum store of a random value *Y = random()*. A quantum RMW's load and store are both replaced as above.

Quantum Race: Two operations, **X** and **Y** form a quantum race if and only if they form a race, either **X** or **Y** is a quantum atomic, and the other is not a quantum atomic.

Program and Model Definitions:

DRFrlx Program: A program is DRFrlx if and only if for every SC execution of its quantum-equivalent program:

- all operations can be distinguished by the system as either data or as paired, unpaired, commutative, non-ordering, or quantum atomics, and
- there are no data races, commutative races, non-ordering races, or quantum races in the execution.

DRFrlx Model: A system obeys the DRFrlx memory model if and only if the result of every execution *E* of a DRFrlx program *P* on the system is the same as the result of an SC execution *Eq* of the quantum-equivalent program *Pq* of *P*. In addition, *E* must obey happens-before consistency and per-location SC.

3.4.4 Using Quantum Atomics in RefCounter.

Reference Counter [61]: Quantum atomics can also be used for some reference counters. The reference counter example in Listing 5 has shared global counters that are incremented and decremented by multiple threads to track the number of threads accessing shared objects. The constructor increments the shared reference counters to signify that a thread is now accessing the shared objects; similarly, the destructor decrements the counters to signify that it is no longer

```
atomic<unsigned long> refcount1, refcount2;

// Thread 1
refcount1.atomic_inc(mem_order_relaxed);
refcount2.atomic_inc(mem_order_relaxed);
...
if (refcount1.atomic_dec(mem_order_relaxed) == 0)
  mark cb_ptr1 to be deleted
if (refcount2.atomic_dec(mem_order_relaxed) == 0)
  mark cb_ptr2 to be deleted

// Thread 2
refcount1.atomic_inc(mem_order_relaxed);
refcount2.atomic_inc(mem_order_relaxed);
...
if (refcount2.atomic_dec(mem_order_relaxed) == 0)
  mark cb_ptr2 to be deleted
if (refcount1.atomic_dec(mem_order_relaxed) == 0)
  mark cb_ptr1 to be deleted
```

Listing 5: Reference counter example [61].

```
atomic<unsigned> seq;
atomic <int> data1, data2;

T reader() {
  int r1, r2;
  unsigned seq0, seq1;
  do {
    seq0 = seq.atomic_load(mem_order_seq_cst);
    r1 = data1.atomic_load(mem_order_relaxed);
    r2 = data2.atomic_load(mem_order_relaxed);
    seq1 = seq.atomic_fetch_add(0, mem_order_seq_cst);
  } while ((seq0 != seq1) || (seq0 & 1));
  // uses r1 and r2
}

void writer(...) {
  unsigned seq0 = seq.atomic_load(mem_order_seq_cst);
  while ((seq0 & 1) || !seq.cmp_exchange_weak(seq0, seq0+1)) { ; }
  data1.atomic_store(..., mem_order_relaxed);
  data2.atomic_store(..., mem_order_relaxed);
  seq.atomic_store(seq0 + 2, mem_order_seq_cst);
}
```

Listing 6: Seqlocks example [11].

accessing the object. If this thread is the last thread accessing the shared counter, then the thread marks the object to be deleted because it is the last thread accessing it. Later, after some synchronization (e.g., a global barrier), a thread will check if the object has been marked for deletion and delete it (not shown).[6]

Since multiple threads concurrently increment and decrement the reference counters, the operations need to be distinguished as atomics. Although relaxing the counter accesses can cause SC violations, like **Split Counter**, **Reference Counter** can tolerate some SC violations. For example, it does not matter whether the accesses to the set of reference counters are sequentially consistent, as long as the final decrement to each counter marks the shared object to be freed. Because of this, **Reference Counter** implementations often trade SC semantics for improved performance by using relaxed atomics.

DRFrlx can use quantum atomics for the increments and decrements, as long as any potentially racy accesses that delete the object are protected by some non-relaxed synchronization (e.g., a global barrier). In the quantum-equivalent program, quantum increments may write a random value and quantum decrements may return (and write) a random value. Therefore, extra care must be taken to avoid race conditions in any possible quantum-equivalent SC execution. If races in the quantum-equivalent SC executions are avoided, then DRFrlx guarantees SC execution for all non-quantum accesses and per-location SC and hb-consistency for all quantum accesses. Although this constraint can limit the use of quantum atomics, the resulting guarantees are stronger than those provided for relaxed atomics in existing consistency models.

3.5 Speculative Atomics

3.5.1 Speculative Atomics Use Case.

Seqlocks [11]: In applications where updates are infrequent, it is often safe for a thread to load shared data without acquiring a lock because usually there are no concurrent writes. In Listing 6, a reader *speculatively* loads shared data (`data1`, `data2`). If there are no concurrent writers (the common case), then the readers can safely use `data1` and `data2` in subsequent instructions (not shown in Listing 6). However, the reader must reload the shared data if a writer is concurrently updating the shared data.

[6]This example differs slightly from Sutter's [61] in order to emphasize the benefits of relaxation in a multi-counter context.

Seqlocks uses a shared sequence number (`seq`) to synchronize the concurrent loads and stores to the shared data. A reader loads `seq` before and after the speculative data loads to check for concurrent writers. If the reader's sequence numbers do not match or are odd, then there is a concurrent writer. Writers make `seq` odd to indicate that an update is in progress. Once the update is complete, the writer updates `seq` to be the next even value.

Both `data` and `seq` must be distinguished as atomics. However, as discussed previously, requiring SC atomics unnecessarily hurts performance. The data accesses can be relaxed – the stores only race with loads and the results of racy loads get discarded, ensuring that these races do not affect the final result. The `seq` accesses ensure that the final data accesses whose values are used do get properly synchronized and ordered.[7]

3.5.2 Speculative Atomics Informal Intuition.

Although relaxing the loads to `data1` and `data2` may read some inconsistent, non-SC values, any misspeculated values will not be used because the sequence numbers will not match. Thus, speculatively accessing the shared data does not violate SC. The stores `data1` and `data2` can also be relaxed without violating SC because they only race with the misspeculated loads. To exploit this intuition, we formalize what it means for a racing access to be "speculative" and call such operations *speculative atomics*. One way to formalize this and ensure the final result is always SC is to require that values returned by racy speculative loads are never used, as in Seqlocks.[8] We formalize this next.

3.5.3 DRFrlx Formal Definition (Version 4).

We only show the parts that change from Section 3.4.3. All memory operations must be distinguished as data, paired, unpaired, commutative, non-ordering, quantum, or *speculative*.

Definitions for an SC Execution:

[7]The reader's `seq` accesses can also be relaxed to acquire and release ordering, which is outside the scope of this paper (Section 7). We note that the `seq1` access uses an unusual "read-don't-modify-write" operation (instead of a plain read) to generate release semantics as explained further in [11].

[8]This concept can be generalized to allow speculative atomics to use their returned values, but only within the speculative part of the program (so they do not affect the final result). It can also be potentially generalized to "read-copy-update" patterns [23, 44]. We omit these generalizations for space.

Speculative Race: Two operations, **X** and **Y**, form a speculative race if and only if they form a race, at least one of **X** or **Y** is distinguished as a speculative atomic, and either:

- both operations are stores, or
- the result of the load is observed by another instruction in the execution (i.e., the returned value is used by another instruction in the thread).

Program and Model Definitions:

DRFrlx Program: A program is DRFrlx if and only if for every SC execution of its (quantum-equivalent) program:

- all operations can be distinguished by the system as either data or as paired, unpaired, commutative, non-ordering, quantum, or *speculative* atomics, and
- there are no data races, commutative races, non-ordering races, quantum races, or *speculative races* in the execution.

3.6 Distinguishing Memory Operations

DRFrlx requires a mechanism in the programming language for distinguishing data operations from atomics, and for distinguishing paired, unpaired, commutative, non-ordering, quantum, and speculative atomics from one another. We reuse the C++ mechanism that DRF0 already uses to distinguish data and atomics. To distinguish the different types of atomics, we introduce five new keywords, unpaired, commutative, non-ordering, quantum, and speculative, to allow programmers to identify which type of relaxed atomics they are using (analogous to how C and C++ specify relaxed atomics). In practice, for the last four categories, the distinctions are important only to enable reasoning about the correctness of the program. For system optimizations, all four can be merged into a single category of relaxed since they allow the same optimizations.

3.7 DRFrlx Correctness Theorem

Theorem 3.1 describes a system with properties that we assert are sufficient to correctly implement DRFrlx. The system used in our evaluation conforms to these properties. Although we omit a proof for space, it follows the basic structure of DRF proofs in prior work [2].

THEOREM 3.1. *Assume a heterogeneous system is DRF1 compliant and enforces happens-before consistency and per-location SC for atomics. Assume the system additionally constrains DRFrlx's commutative, non-ordering, quantum, and speculative operation completion/propagation in the same way as data operations. Such a system is DRFrlx compliant.*

3.8 Formalizing DRFrlx

We formalize DRFrlx with Herd [6], a tool for formalizing memory models in terms of allowed relations between memory accesses in different threads. Given a model definition and a program, Herd produces all possible executions of the program as constrained by the model, and flags any relations of interest as specified by the model (e.g., race conditions). Since Herd does not support reads/writes of random values, this model is only able to identify races in SC executions of the original program, not the quantum-equivalent program. Therefore it is not an exhaustive exploration, and some manual inspection is necessary when quantums are used. Additionally, Herd does not have a built-in way to determine if the value returned by a memory operation is observable by any other instruction in the thread. Therefore, for commutative and speculative atomics we approximate observability by defining it as any return value which

```
let at-least-one a = a*_ | _*a

let PairedR = (Paired & R)
let PairedW = (Paired & W)
let sol = (PairedW * PairedR) & (rf | fr | co)+
let hb1 = (po | sol)+
let conflict = at-least-one W & loc
let race = (conflict & ext & ~(hb1 | hb1^-1)) \ (IW*_)
let data-race = race & (at-least-one Data)

(* comm-pair relates any two memory operations which are pairwise
     commutative (we omit the precise definition for space) *)
(* commutative race: a race involving a commutative access where
     either a) the accesses are not pairwise commutative *)
let comm-race1 = (race & (at-least-one Comm)) \ comm-pair
(* or b) the return value of an operation is observable *)
let comm-race2 = (race & (at-least-one Comm)) ; (addr | data |
     ctrl)
let comm-race = comm-race1 | comm-race2

(* pco: program-conflict order, pcoPO: pco that contains a po edge
     *)
let pco = (po | co | rf | fr)+
let pco-po = po | (po ; pco) | (pco ; po ; pco) | (pco ; po)

(* opath-aloNO: ordering path with at least one NO atomic *)
let aloNO = (at-least-one NonOrder)
let pcoPO-NO-pco = (pcoPO & aloNO) ; pco
let pco-NO-pcoPO = pco ; (pcoPO & aloNO)
let pcoPO-aloNO = (pcoPO & aloNO) | pcoPO-NO-pco | pco-NO-pcoPO
let opath-aloNO = pcoPO-aloNO & conflict

(* valid ordering path 1: accesses to the same address *)
let valid-pco1 = ((po | co | rf | fr) & loc)+
let valid-po1 = po & loc
let valid-pcoPO1 = valid-po1 | (valid-po1 ; valid-pco1) | (valid-
     pco1 ; valid-po1 ; valid-pco1) | (valid-pco1 ; valid-po1)
let valid-opath1 = valid-pcoPO1 & conflict

(* valid ordering path 2: Unpaired/Paired accesses *)
let valid-pco2 = ((po | co | rf | fr) & (Paired | Unpaired)*(
     Paired | Unpaired))+
let valid-po2 = po & (Paired | Unpaired)*(Paired | Unpaired)
let valid-pcoPO2 = valid-po2 | (valid-po2 ; valid-pco2) | (valid-
     pco2 ; valid-po2 ; valid-pco2) | (valid-pco2 ; valid-po2)
let valid-opath2 = valid-pcoPO2 & conflict

(* non-ordering race: there is an ordering path between two
     accesses which contains a NonOrdering edge, and there are no
     alternate valid ordering paths *)
(* note: for simpler herd construction, this relation is defined
     between the accesses at the ends of the ordering path *)
let non-order-race = ((race \ data-race \ comm-race) & opath-aloNO
     ) \ valid-opath1 \ valid-opath2

(* quantum race: Quantum races with non-quantum *)
let quantum-race = (race & (at-least-one Quantum)) \ (Quantum *
     Quantum)

(* speculative race: a race involving a speculative access where
     either a) both accesses are writes *)
let speculative-race1 = (race & (at-least-one Spec) & (W * W))
(* ... or b) the racy load is observable *)
let speculative-race2 = (race & (at-least-one Spec)) ; (addr |
     data | ctrl)
let speculative-race = speculative-race1 | speculative-race2

let illegal-race = data-race | comm-race | non-order-race |
     quantum-race | speculative-race

(* limit to SC executions *)
acyclic (po | rf | co | fr)
(* RMWs to happen atomically *)
empty rmw & (fre ; coe)

(* Identify any races in SC executions *)
flag ~empty (illegal-race) as IllegalRace
```

Listing 7: DRFrlx's programmer-centric model in Herd: defining and identifying illegal races in a program.

(directly or indirectly) affects the address used by a future memory access, the value stored by a future memory access, or the path taken by a future branch. This is also imprecise and requires some manual inspection when using racy commutative and speculative accesses.

Our Herd evaluation consists of two models. Listing 7 shows our programmer-centric model, which defines and identifies illegal races under DRFrlx. Each illegal race type is specified using terms such as the program order, modification order, and reads-from relations in a dynamic execution. Given an example program, Herd generates all possible SC executions and determines whether any illegal race conditions exist in the generated executions. We also defined a system-centric model (omitted for space) which generates all possible executions in a straightforward example DRFrlx system. This model restricts program executions in a way that preserves intuitive atomic reordering invariants. For example, successive unpaired accesses must occur in program order, paired reads may not be reordered with subsequent memory accesses, and paired writes may not be reordered with prior memory accesses. Using this model we can determine whether a program can exhibit non-SC behavior on such a system.

We created numerous litmus tests to stress our models. These include the use cases in Table 1, incorrectly labeled versions of these use cases, and various other tests designed to stress various racy and non-racy patterns. Although we omit detailed results for space, for all litmus tests, the programmer-centric model correctly identifies races in the SC execution, and the system-centric model can only produce non-SC executions when the model allows it (i.e., when there is an illegal race or when quantum atomics are used).

4 METHODOLOGY

Our work is influenced by previous work on DeNovo [21, 37, 53, 59, 60]. We leverage the project's existing infrastructure [53] and extend it to support relaxed atomics and the DRFrlx memory model.

4.1 Baseline Heterogeneous Architecture

We model a tightly coupled CPU-GPU architecture with a unified shared memory address space and coherent (conventional GPU or DeNovo style) caches, using methodology similar to other work (e.g., [53]). The system connects all CPU cores and GPU Compute Units (CUs) via an interconnection network. Like prior work, each CPU core and each GPU CU (analogous to an NVIDIA SM) is on a separate network node. Each network node has an L1 cache (local to the CPU core or GPU CU), a bank of the shared L2 cache (logically shared by all CPU cores and GPU CUs), and a scratchpad [37].

4.2 Simulation Environment and Parameters

We simulate the above architecture using an integrated CPU-GPU simulator built from the Simics full-system functional simulator to model the CPUs, the Wisconsin GEMS memory timing simulator [43], and GPGPU-Sim v3.2.1 [8] to model the GPU (the GPU is similar to an NVIDIA GTX 480). The simulator also uses Garnet [5] to model a 4x4 mesh interconnect with a GPU CU or a CPU core at each node. We use CUDA 3.1 [47] since this is the latest version of CUDA that is fully supported in GPGPU-Sim. Table 2 summarizes the key system parameters. Additionally, we assume support for performing atomics at the L1 (DeNovo) and L2 (GPU coherence).

CPU Parameters	
Frequency	2 GHz
Cores	1
GPU Parameters	
Frequency	700 MHz
CUs	15
Memory Hierarchy Parameters	
L1 size (8 banks, 8-way assoc.)	32 KB
L2 size (16 banks, NUCA)	4 MB
Store buffer size	128 entries
L1 MSHRs	128 entries
L1 hit latency	1 cycle
Remote L1 hit latency	35−83 cycles
L2 hit latency	29−61 cycles
Memory latency	197−261 cycles

Table 2: Simulated heterogeneous system parameters.

Benchmark	Input	Atomic Types
Microbenchmarks		
Hist (H)[50]	256 KB, 256 bins	Commutative
Hist_global (HG)[50]	256 KB, 256 bins	Commutative
HG-Non-Order (HG-NO)	256 KB, 256 bins	Non-Ordering
Flags[61]	90 Thread Blocks	Commutative, Non-Ordering
SplitCounter (SC)[44]	112 Thread Blocks	Quantum
RefCounter (RC)[61]	64 Thread Blocks	Quantum
Seqlocks (SEQ)[11]	512 Thread Blocks	Speculative
Benchmarks		
UTS[32, 48]	16K nodes	Unpaired
BC[18]	rome99 (1), nasa1824 (2), ex33 (3), c_22 (4)	Commutative, Non-Ordering
PageRank (PR)[18]	c-37 (1), c-36 (2) ex3 (3), c-40 (4)	Commutative

Table 3: Benchmarks, input sizes, and relaxed atomics used.

Our energy model uses GPUWattch [40] for the GPU CUs and McPAT v1.1 [41] for the NoC energy measurements (our architecture more closely resembles a multicore NoC than GPUWattch's NoC). We do not model the CPU core or CPU L1 energy since the CPU is only functionally simulated and not the focus of this work.

4.3 Configurations

We evaluate all combinations of a traditional GPU and the DeNovo coherence protocols with the DRF0, DRF1, and DRFrlx memory models. We use the following abbreviations to refer to these combinations: *GD0* = GPU+DRF0; *GD1* = GPU+DRF1; *GDR* = GPU+DRFrlx; *DD0* = DeNovo+DRF0; *DD1* = DeNovo+DRF1; and *DDR* = DeNovo+DRFrlx.

4.4 Benchmarks

We evaluate the effectiveness of relaxed atomics on heterogeneous CPU-GPU systems with a mix of microbenchmarks (based on the examples discussed in Section 3) and benchmarks, summarized in Table 3. For all benchmarks, we found that that the CUDA compiler would put as much independent computation as possible between atomics. Although this optimization makes sense for current GPUs, where atomics are infrequent, it also prevents some relaxed atomics from being overlapped. Thus, we wrote hand-optimized assembly to increase the overlap of relaxed atomics by grouping atomics together.

The microbenchmarks represent the use cases we obtained from developers. Historically, relaxed atomics are necessary to obtain high performance for these applications. We designed these microbenchmarks to stress the benefit that relaxed atomics could provide from

Benefit	DRF0	DRF1 (if unpaired)	DRFrlx (if unpaired or relaxed)
Avoid cache invalidations at atomic loads	✗	✓	✓
Avoid store buffer flushes at atomic stores	✗	✓	✓
Overlap atomics in the memory system	✗	✗	✓

Table 4: Benefits of DRF0, DRF1, and DRFrlx.

overlapping atomics in the memory system – although relaxed atomics can also benefit from reusing data, the microbenchmarks have very few global data operations. We use a histogram [50] for the Event Counters example, and created several variants to highlight different types of access patterns. In Hist (H), each thread locally bins its values in the scratchpad before updating the shared global histogram once all its data has been binned. To model high contention, Hist_global (HG) performs all updates on the shared global histogram instead of locally binning its values first. Unlike H and HG, HG-Non-Order (HG-NO) reads the final values of the histogram bins, like the bottom of Listing 2. To examine how this part of the Event Counter performs, we do not include the update portion (i.e., the HG portion) in its results.

Although omitted for space, we examined different levels of contention and number of bins for the histogram applications. More bins and reduced contention improve performance for all configurations, but did not change the observed trends. We wrote the remainder of the microbenchmarks based on the code listings in Section 3.

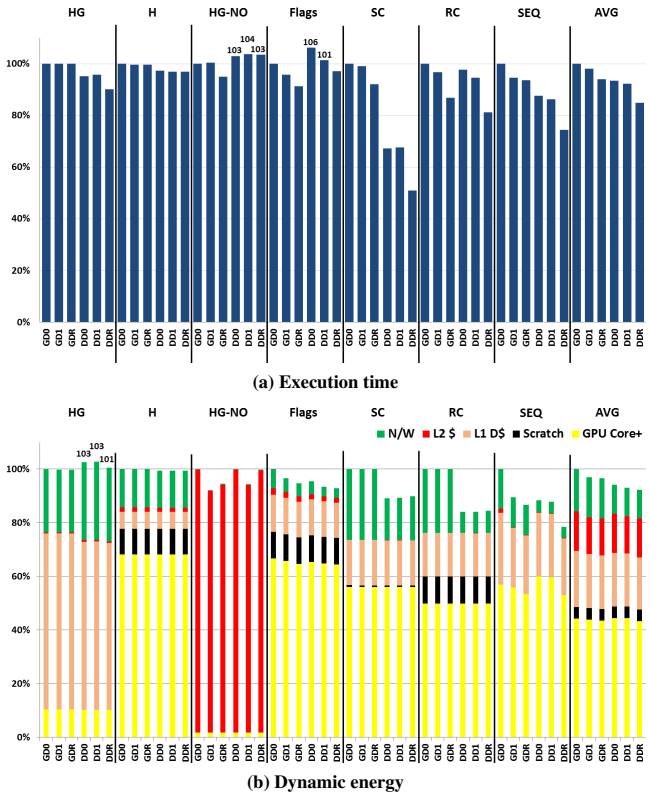
For the full benchmarks, we first identified which (standard) GPGPU benchmarks [16, 18–20, 25, 31, 47, 48, 52, 57, 58] use atomics and categorized them, focusing on the 12 benchmarks that use relaxed atomics. We use the two benchmarks from Figure 1 that obtain the highest max speedups: BC and PageRank. In addition, we chose UTS, which is representative of future workloads that perform dynamic load balancing. Unlike the microbenchmarks, these benchmarks benefit from overlapping relaxed atomics, reusing data that would invalidated by SC atomics, and avoiding store buffer flushes. UTS uses unpaired atomics, similar to the Work Queue example, while BC and PageRank use commutative and non-ordering atomics. For BC and PageRank, we studied 33 Matrix Market graphs [22] and show results for four representative graphs.

5 QUALITATIVE ANALYSIS

In CPUs, the main benefit for relaxed atomics is overlapping relaxed atomics in the memory system. Unlike multicore CPUs, heterogeneous systems largely use simple, software-based coherence protocols. As a result, relaxed atomics allow heterogeneous systems to reuse data across synchronization points by avoiding full cache invalidations on atomic loads and avoiding store buffer flushes on atomic stores. Table 4 qualitatively compares DRF0, DRF1, and DRFrlx.

DRF0 vs. DRF1: DRF0 treats all atomics as paired, so they cannot be overlapped, must invalidate all valid data on atomic loads, and must flush the store buffer on atomic stores. By distinguishing unpaired from paired atomics, DRF1 does not need to invalidate valid data or flush the store buffer on an unpaired atomic, which reduces overhead and improves valid data reuse compared to DRF0.

DRF1 vs. DRFrlx: Although DRF1 provides several benefits over DRF0, it does not allow atomics to be overlapped. DRFrlx improves



(a) Execution time

(b) Dynamic energy

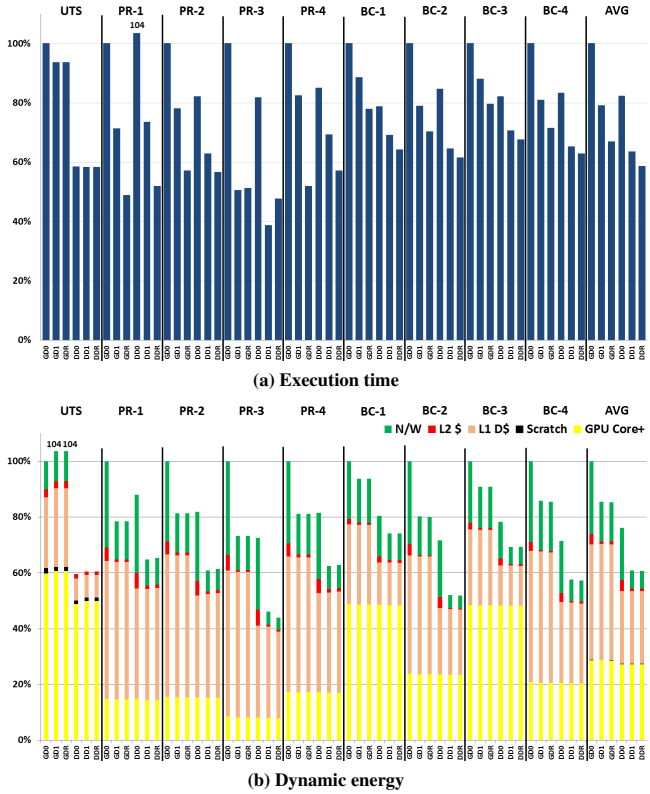
Figure 3: Results for all microbenchmarks, normalized to *GD0*.

performance and memory-level parallelism over DRF1 by allowing relaxed atomics to be overlapped in the memory system.

GPU coherence vs. DeNovo: The choice of coherence protocol also affects performance. Since DeNovo obtains ownership for written data and atomics, it can reuse them for all three consistency models. Obtaining ownership also allows DeNovo's L1 MSHRs to locally coalesce multiple requests for the same address, which reduces network traffic, improves performance, and allows DeNovo with DRFrlx to quickly service many overlapped atomic requests. However, obtaining ownership can hurt performance if an address is highly contended because DeNovo may have to get ownership from a remote L1. Conversely, GPU coherence writes through all dirty data to the LLC on a store buffer flush. Thus, relaxed atomics are important because they allow GPU coherence to avoid flushing the store buffer. GPU coherence also performs all atomic operations at the LLC. As a result, it never needs to go to a remote core for ownership. Although this may help for addresses where reuse is unlikely (e.g., highly contended or sparsely accessed addresses), GPU coherence also cannot coalesce multiple atomic requests for the same address. This exacerbates LLC contention for applications with large amounts of atomic parallelism.

6 RESULTS

Figures 3 and 4 show results (normalized to the GD0 configuration) for the microbenchmarks and benchmarks, respectively, for all 6 configurations (Section 4.3). Parts (a) and (b) of the figures show the execution time and energy consumption, respectively. Energy



(a) Execution time

(b) Dynamic energy

Figure 4: Results for all benchmarks, normalized to *GD0*.

is divided into multiple components based on the source of energy: GPU core+,[9] scratchpad, L1, L2, and network.

Our experiments show mixed results for the effectiveness of DRF1 and DRFrlx over DRF0. For the microbenchmarks, DRF1 and DRFrlx provide small benefits: on average, DRFrlx reduces execution time by 6% for GPU coherence and 10% for DeNovo; DRF1's average benefits are negligible. Of the microbenchmarks, relaxed atomics help the most for SC, RC, and SEQ: up to 13% reduction in execution time for GPU coherence and 25% for DeNovo, compared to DRF0. For BC and PR, the benefits of DRF1 are higher, depending on the graph (up to 49% for GD1 and 53% for DD1 compared to GD0 and DD0, respectively). DRFrlx further reduces execution time for several BC and PR graphs (up to 29% for DDR and 37% for GDR compared to DD1 and GD1, respectively). In most cases, DeNovo's ability to reuse data and atomics also improves energy compared to GPU. However, accessing data remotely sometimes increases DeNovo's energy (e.g., HG). Comparing the interaction between the different protocols and consistency models, we find (as also shown in past work) that DD0 generally provides improved or comparable performance relative to GD0, except for HG-NO, Flags, and PR-1. As we weaken the memory models, the gap between DeNovo and GPU coherence stays roughly the same. On average, DeNovo reduces execution time by 14% for DRF0, 14% for DRF1, and 12% for DRFrlx and energy by 16%, 18%, and 18%.

[9]GPU core+ includes the instruction cache, constant cache, register file, SFU, FPU, scheduler, and the core pipeline.

6.1 DRF0 vs. DRF1

DRF1's unpaired atomics can improve performance by avoiding the store buffer flushes and self-invalidations associated with paired synchronization writes and reads. However, since the microbenchmarks have few data accesses, DRF1 has little impact on them. Unlike the microbenchmarks, the full-sized benchmarks are able to benefit from DRF1. DRF1 reduces UTS's execution time by 6%, relative to GD0, by increasing data reuse, although DD1 does not reduction execution time compared to DD0 because DD0 already obtains ownership for the data. However, using unpaired atomics removes DRF0's ordering constraints and increases the rate at which atomics load a polled value, increasing UTS's energy. BC and PR benefit the most from DRF1 because they have frequent relaxed atomics and high levels of data reuse – avoiding cache invalidations in DRF1 increases their data reuse compared to DRF0. On average, DRF1 reduces BC's execution time by 18% for DeNovo (16% for GPU) and energy by 17% for DeNovo (12% for GPU). Across all the benchmarks and microbenchmarks, DRF1 reduces DeNovo's (GPU's) execution time by 11% (11%) and energy by 12% (10%).

6.2 DRF1 vs. DRFrlx

DRFrlx allows relaxed atomics to be overlapped in the memory system, which increases memory-level parallelism over DRF1. All microbenchmarks except H and HG-NO with DDR see some benefit from this, although the benefit is sometimes small due to increased contention. Since H locally bins its data before updating the global histogram, it has few atomics to overlap, while HG-NO with DDR suffers from the overhead of obtaining ownership from a remote core. Conversely, obtaining ownership for atomics enables DeNovo to reuse them and often improves performance. As a result, DDR reduces SC, RC, and SEQ's execution time by 25%, 14%, and 14%, respectively, compared to DD1. As expected, DRFrlx does not affect UTS's execution time, because UTS uses unpaired atomics. However, BC and PR see benefits from DRFrlx (up to 29% reduction in execution time for DDR and 37% for GDR compared to DD1 and GD1, respectively). PR benefits more than BC because it does not have as many control and data dependencies as BC, although in PR-3 the added contention increases execution time. In general, DRFrlx does not improve energy because the overhead from the increased memory contention cancels out the additional reuse benefits. On average DRFrlx reduces DeNovo's (GPU coherence's) execution time by 7% (9%) and provides the same energy efficiency as DRF1.

6.3 DeNovo vs. GPU Coherence

Obtaining ownership for written data and atomics allows DeNovo to reuse them regardless of consistency model. Normally this is beneficial, but in some cases the overhead of accessing data remotely increases execution time (PR-1, HG-NO, Flags) and energy (HG). However, obtaining ownership usually helps and on average DD0 reduces execution time by 14% and energy by 16% compared to GD0. DRF1 allows reuse of valid data across unpaired atomics and avoids excessive store buffer flushes. Increased reuse helps GD1 for all of the full-sized benchmarks, especially BC, which has lots of potential data reuse. However, GPU coherence cannot reuse atomics, which is why DD1 still outperforms GD1. On average DD1 reduces execution time by 14% and energy by 18% compared to GD1. By overlapping the relaxed atomics, GDR is able to hide the latency of

performing the atomics at the L2. This helps GPU coherence overcome its inability to reuse atomics and provide similar performance to DeNovo with DRFrlx for some benchmarks. However, in many other cases (PR-3, BC 1-4, HG, SC, RC, SEQ), DeNovo provides additional benefits with DRFrlx by coalescing atomics in the L1 MSHR, which filters requests, reduces traffic, and allows DeNovo to support a higher atomic access bandwidth. On average DDR reduces execution time by 12% and energy by 18% over GDR.

7 RELATED WORK

The HSA, HRF, and OpenCL memory models seek to mitigate the overhead of atomics with another construct: scoped synchronization [9, 26, 32, 33, 38]. These models allow the programmer to distinguish some atomics as having local scope (vs. global scope) while retaining SC semantics. However, scoped synchronization based models do not address the overheads for globally scoped synchronization. Additionally, previous work has shown that with an appropriate coherence protocol (e.g., the DeNovo protocol), scopes are not worth the added complexity to the memory model [7, 53].

Other work has tried to improve support for relaxed atomics in C, C++, Java, HSA, HRF, and OpenCL [9, 26, 33, 35, 36, 39, 49]. We take a different approach, motivated by how developers use relaxed atomics in heterogeneous systems, and extend the existing DRF memory models to incorporate these use cases with SC-centric guarantees. Previous work has also examined how applications with relaxed atomics behave on various multicore CPUs with weak memory models [30, 51] and GPUs [55, 56]. This work demonstrates the difficulty in correctly synchronizing applications, which further motivates designing simpler, SC-centric consistency models.

This paper focuses on `memory_order_relaxed`. However, some applications use other relaxed memory orderings such as the release and acquire memory orderings. For example, Seqlocks' reader-side `seq` accesses can use release-acquire ordering [11]. Since these memory orderings are not our focus, we do not explore them in this paper. However, PL_{pc}'s [2, 27] unessential operations and loop reads/writes could be used to ensure SC for some of these applications.

`Memory_order_consume` can improve performance compared to `memory_order_acquire` by relaxing the ordering of subsequent memory accesses with respect to the consume operation [45]. Consume provides some similar relaxations to quantum, but allows less reorderings because it relies on dependencies for ordering. Moreover, it is hard for compilers to correctly identify dependence ordering [45] and the C++17 standard advises against its use [54].

Coup also exploits commutativity to improve performance of updates to shared data [63]. Although our work also exploits commutativity, Coup focuses on how to efficiently support commutative operations in the coherence protocol, whereas we created a new memory model that provides more robust semantics for several classes of atomic operations, not just commutative atomics.

8 CONCLUSION

Despite more than a decade of research, no acceptable semantics for relaxed atomics have been found. Unlike previous work, which tries to formalize the semantics by prohibiting "out-of-thin-air" executions, we focus on how developers want to use relaxed atomics in heterogeneous systems. After examining numerous GPGPU benchmarks and reaching out to vendors, developers, and researchers, we identified five use cases: unpaired, commutative, non-ordering, quantum, and speculative. Next, we designed a new memory model, DRFrlx, that extends DRF0 and DRF1 to provide SC-centric semantics for these use cases. Finally, we evaluate relaxed atomics in heterogeneous CPU-GPU systems for these use cases. Compared to DRF0, we find that DRF1 and DRFrlx provide small benefits for all benchmarks except SplitCounter, RefCounter, Seqlocks, BC, and PageRank; BC and PageRank benefit significantly from DRF1 (up to 53% execution time reduction) and see additional benefits from DRFrlx (up to 37% execution reduction compared to DRF1). Our results also show that the recently proposed DeNovo coherence protocol outperforms a conventional GPU coherence protocol, regardless of memory model: on average DeNovo reduces execution time over GPU coherence by 14%, 14%, and 12% and energy by 16%, 18%, and 18% for DRF0, DRF1, and DRFrlx, respectively.

ACKNOWLEDGMENTS

This work was supported in part the National Science Foundation under grants CCF 13-02641 and CCF 16-19245 and by the Center for Future Architectures Research (C-FAR), one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. We are grateful to Brad Beckmann, Paul Blinzer, Hans Boehm, Olivier Giroux, Doug Lea, Paul McKenney, Hakan Persson, Marc Snir, and Dmitry Vyukov for helping us identify uses of relaxed atomics and/or offering feedback that improved the quality of this paper.

REFERENCES

[1] Sarita Adve and Mark Hill. 1990. Weak Ordering – A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*.
[2] Sarita V. Adve. 1993. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. Ph.D. Dissertation. University of Wisconsin, Madison.
[3] Sarita V. Adve and Hans-J. Boehm. 2010. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Communications of the ACM* (August 2010), 90–101.
[4] S. V. Adve and M. D. Hill. 1993. A Unified Formalization of Four Shared-Memory Models. *TPDS*, Article 6 (June 1993), 613-624 pages.
[5] N. Agarwal, T. Krishna, Li-Shiuan Peh, and N.K. Jha. 2009. GARNET: A Detailed On-chip Network Model Inside a Full-system Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*.
[6] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Transactions on Programming Languages and Systems* 36, 2, Article 7 (July 2014), 7:1–7:74 pages.
[7] Johnathan Alsop, Bradford Beckmann, Marc Orr, and David Wood. 2016. Lazy Release Consistency for GPUs. In *Proceedings of the 49th International Symposium on Microarchitecture*.
[8] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*.
[9] Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC Atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 634–648.
[10] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Programming Languages and Systems*. Lecture Notes in Computer Science, Vol. 9032. 283–307.
[11] Hans-J. Boehm. 2012. Can Seqlocks Get Along with Programming Language Memory Models?. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. 12–20.
[12] Hans-J. Boehm. 2013. N3710: Specifying the absence of "out of thin air" results (LWG2265). (2013).
[13] Hans-J. Boehm. 2013. N3786: Prohibiting "out of thin air" results in C++14. (2013).
[14] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 68–78.
[15] 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. 7:1–7:6.

[16] M. Burtscher, R. Nasre, and K. Pingali. 2012. A Quantitative Study of Irregular Programs on GPUs. In *IEEE International Symposium on Workload Characterization*. 141–151.

[17] C++. 2015. C++ Reference: Memory Order. http://en.cppreference.com/w/cpp/atomic/memory_order. (2015).

[18] Shuai Che, B.M. Beckmann, S.K. Reinhardt, and K. Skadron. 2013. Pannotia: Understanding Irregular GPGPU Graph Applications. In *IEEE International Symposium on Workload Characterization*. 185–195.

[19] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization*. 44–54.

[20] Shuai Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, Liang Wang, and K. Skadron. 2010. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP workloads. In *IEEE International Symposium on Workload Characterization*. 1–11.

[21] Byn Choi, R. Komuravelli, Hyojin Sung, R. Smolinski, N. Honarmand, S.V. Adve, V.S. Adve, N.P. Carter, and Ching-Tsun Chou. 2011. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*. https://doi.org/10.1109/PACT.2011.21

[22] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software* 38, 1 (2011), 1:1–1:25.

[23] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems* 23, 2 (Feb 2012), 375–382.

[24] M. Dubois, C. , and F. Briggs. 1986. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*. 434–442.

[25] Wilson W. L. Fung and Tor M. Aamodt. 2013. Energy Efficient GPU Transactional Memory via Space-time Optimizations. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 408–420.

[26] Benedict R. Gaster, Derek Hower, and Lee Howes. 2015. HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models. *ACM Transactions on Architecture and Code Optimizations* 12 (April 2015), 7:1–7:26.

[27] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. 1992. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing* 15 (1992), 399–407.

[28] 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*. 15–26.

[29] James R Goodman. 1989. *Cache Consistency and Sequential Consistency*. Technical Report.

[30] Vincent Gramoli. 2015. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–10.

[31] J. Hestness, S. Keckler, and David A. Wood. 2015. GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors. In *IEEE International Symposium on Workload Characterization*. 87–97.

[32] 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*. 427–440.

[33] HSA Foundation. 2015. HSA Platform System Architecture Specification. http://www.hsafoundation.com/?ddownload=4944. (2015).

[34] IntelPR. 2014. Intel Discloses Newest Microarchitecture and 14 Nanometer Manufacturing Process Technical Details. *Intel Newsroom* (2014).

[35] Alan Jeffrey and James Riely. 2016. On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. 759–767.

[36] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 175–189.

[37] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Prakalp Srivastava, Maria Kotsifakou, Sarita V. Adve, and Vikram S. Adve. 2015. Stash: Have Your Scratchpad and Cache it Too. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 707–719.

[38] L. Howes and A. Munshi. 2015. The OpenCL Specification, Version 2.0. Khronos Group. (2015).

[39] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-Acquire Consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 649–662.

[40] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*.

[41] Sheng Li, Jung-Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 469–480.

[42] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the 32nd Symposium on Principles of Programming Languages*. 378–391.

[43] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. 2005. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News* (2005). https://doi.org/10.1145/1105734.1105747

[44] Paul McKenney. 2015. Some Examples of Kernel-Hacker Informal Correctness Reasoning. In *Proceedings of the Dagstuhl Workshop on Compositional Verification Methods for Next-Generation Concurrency*. 15.

[45] Paul E. McKenney, Torvald Riegel, and Jeff Preshing. 2014. N4036: Towards Implementation and Use of memory_order_consume. (2014).

[46] David S. Miller. 2016. Semantics and Behavior of Atomic and Bitmask Operations. (2016).

[47] NVIDIA. 2010. CUDA SDK 3.1. (2010).

[48] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. 2007. UTS: An Unbalanced Tree Search Benchmark. In *Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, Vol. 4382. 235–250.

[49] Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 622–633.

[50] Victor Podlozhnyuk. 2007. Histogram calculation in CUDA. (2007), 11 pages.

[51] Carl G. Ritson and Scott Owens. 2016. Benchmarking Weak Memory Models. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Article 24, 11 pages.

[52] Matthew Sinclair, Henry Duwe, and Karthikeyan Sankaralingam. 2011. *Porting CMP Benchmarks to GPUs*. Technical Report. Department of Computer Sciences, The University of Wisconsin-Madison.

[53] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2015. Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*. 647–659.

[54] Richard Smith. 2017. N4659: Working Draft, Standard for Programming Language C++. (2017).

[55] Tyler Sorensen, Jade Alglave, Ganesh Gopalakrishnan, and Vinod Grover. 2013. ICS: U: Towards Shared Memory Consistency Models for GPUs. In *Towards Shared Memory Consistency Models for GPUs*. 489–490.

[56] Tyler Sorensen and Alastair F. Donaldson. 2016. Exposing Errors Related to Weak Memory in GPU Applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 100–113.

[57] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and WMW Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report. Department of ECE and CS, University of Illinois at Urbana-Champaign.

[58] Jeff A. Stuart and John D. Owens. 2011. Efficient Synchronization Primitives for GPUs. *CoRR* abs/1110.4623 (2011).

[59] Hyojin Sung and Sarita V. Adve. 2015. DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*. 545–559.

[60] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. 2013. DeNovoND: Efficient Hardware Support for Disciplined Non-determinism. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*. 13–26.

[61] Herb Sutter. 2012. Atomic Weapons: The C++ Memory Model and Modern Hardware. In *C++ and Beyond*.

[62] Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. 867–884.

[63] Guowei Zhang, Webb Horn, and Daniel Sanchez. 2015. Exploiting Commutativity to Reduce the Cost of Updates to Shared Data in Cache-coherent Systems. In *Proceedings of the 48th International Symposium on Microarchitecture*. 13–25.