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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference format:

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
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 DRFRxl, 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 atoms 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 atomic 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 atoms 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 atoms. Our expectation is that by clearly stating this requirement, the use of such atoms 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 atoms: 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 atoms 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 atoms be distinguished as SC atoms or one of the above relaxed atoms (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 atoms. DRFRlx provides the same semantics as DRF0/1 when relaxed atoms are not used and simpler semantics for relaxed atoms than the state-of-the-art.

We do not claim that our approach covers every possible use case of relaxed atoms. Further, we focus on the memory_order_relaxed version of relaxed atoms 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 atoms make relaxed atoms attractive. To determine if the complexity of relaxed atoms 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 atoms. 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 atoms 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 atoms at the LLC, DeNovo obtains ownership for all atoms at the L1, exploiting reuse for atoms.

2.3 DRF1 Consistency Model

The DRF1 memory model removes some ordering constraints from DRF0 by distinguishing paired synchronization read-write atoms from unpaired atoms that do not order data operations [4]. It allows unpaired atoms 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 \(\rightarrow\) 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** (\(\rightarrow\)): Let \(X\) and \(Y\) be two memory operations in an execution. \(X \rightarrow Y\) 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** (\(\rightarrow\)): The happens-before-1 relation is defined on the memory operations of an execution as the irreflexive transitive closure of \(p_0\) and \(s_0\): \(\text{hb} = (p_0 \cup s_0)\).

- **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 atomic can be distinguished from unpaired atomic. 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.

#### 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’ atomic operations only need to order data when the queue is not empty.

If this example were constrained to using paired (i.e., SC) atomic, 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 queue. 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
We make the following key observations that enable us to formalize commutative atomics

(i) racing increments in an execution of Listing 2 are commutative centric semantics by distinguishing these accesses as quantum atomics (Section 3.4).

\[
\text{atomic <int> count[NUM_BINS]; // all bins initialized to 0}
\]

// 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 rl = count[i].atomic_load(mem_order_relaxed);
        ...
    } // uses rl
}

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

---

5For 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.
We can informally reason that making the operations to \( X \) and \( Y \) reordered. Thus, non-ordering the operations to \( Y \) will be avoided – operations on other paths between \( X \) and \( Y \) will not violate SC for this code. Similarly, we can also reason that dirty \( Y \) 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 \((\rightarrow)\): \( X \rightarrow Y \) 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) \( hhi \), 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. \( X \rightarrow Y \) 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 \( X \): \( X=3 \rightarrow \), \( Y=2 \rightarrow r1=Y \rightarrow r2=X \). Since this path contains a non-ordering atomic, a non-ordering race occurs. Figure 2(b) adds a new ordering path: \( X=3 \rightarrow P Z=1 \rightarrow r0=Z \rightarrow r2=X \). 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

```c
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].

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNP X = 3</td>
<td>Thread 0</td>
</tr>
<tr>
<td>NO Y = 2</td>
<td>UNP X = 3</td>
</tr>
<tr>
<td>r2 = X</td>
<td>Y</td>
</tr>
<tr>
<td>r0 = Z</td>
<td>P</td>
</tr>
<tr>
<td>r2 = X</td>
<td>UNP</td>
</tr>
<tr>
<td>co</td>
<td></td>
</tr>
</tbody>
</table>

(a) 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.
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 \rightarrow bb \rightarrow S or that there exists another store S' to M such that S \rightarrow bb \rightarrow S' \rightarrow bb \rightarrow L.

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

Quantum-Equivalent Program: We generate a quantum-equivalent program Pq from a program P as follows. Each quantum atomic load r suffix = Y is replaced with a call to a conceptual random function r suffix = random(); in Pq. Similarly, each quantum atomic store Y = r suffix 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 atoms, 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

Listing 4: Split counters example [44].

```c
atomic unsigned long> myCount[NUM_THREADS];
add_split_counter(y, tID) {
  val = myCount[tID]; atomic_load(mem_order_relaxed);
  newVal = val + y;
  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
```
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).\footnote{This example differs slightly from Sutter’s [61] in order to emphasize the benefits of relaxation in a multi-counter context.}

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.

Listing 5: Reference counter example [61].

```c
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;
```


```c
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);
  seq = seq.atomic_fetch_add(0, mem_order_seq_cst);
  while ((seq0 != seq1) || (seq0 & 1)) {
    // uses r1 and r2
  }
}

void writer(...) {
  seq.atomic_store(..., mem_order_seq_cst);
  while ((seq0 & 1) || !seq.compare_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);
}
```

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.\footnote{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].}

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.\footnote{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.}

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:

```c
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);
    seq = seq.atomic_fetch_add(0, mem_order_seq_cst);
  }
  
  // uses r1 and r2
  }

void writer(...) {
  seq.atomic_store(..., mem_order_seq_cst);
  while ((seq0 & 1) || !seq.compare_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);
}
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:

**DRFrIx Program**: A program is DRFrIx 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 atoms, and
- there are no data races, commutative races, non-ordering races, quantum races, or speculative races in the execution.

3.6 Distinguishing Memory Operations

DRFrIx requires a mechanism in the programming language for distinguishing data operations from atomics, and for distinguishing paired, unpaired, commutative, non-ordering, quantum, and speculative atoms from one another. We reuse the C++ mechanism that DRF0 already uses to distinguish data and atomics. To distinguish the different types of atoms, 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 DRFrIx Correctness Theorem

Theorem 3.1 describes a system with properties that we assert are sufficient to correctly implement DRFrIx. 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-locaton SC for atomics. Assume the system additionally constrains DRFrIx’s commutative, non-ordering, quantum, and speculative operation completion/propagation in the same way as data operations. Such a system is DRFrIx compliant.

3.8 Formalizing DRFrIx

We formalize DRFrIx 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

Listing 7: DRFrIx’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 atoms 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 atoms 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 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).

Table 2: Simulated heterogeneous system parameters.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input</th>
<th>Atomic Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microbenchmarks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hist (H)[50]</td>
<td>256 KB, 256 bins</td>
<td>Commutative</td>
</tr>
<tr>
<td>Hist_global (HG)[60]</td>
<td>256 KB, 256 bins</td>
<td>Commutative</td>
</tr>
<tr>
<td>HG-Non-Order (HG-NO)</td>
<td>256 KB, 256 bins</td>
<td>Non-Ordering</td>
</tr>
<tr>
<td>Flags[61]</td>
<td>90 Thread Blocks</td>
<td>Commutative, Non-Ordering</td>
</tr>
<tr>
<td>SplitCounter (SC)[44]</td>
<td>112 Thread Blocks</td>
<td>Quantum</td>
</tr>
<tr>
<td>Relcounter (RC)[51]</td>
<td>94 Thread Blocks</td>
<td>Quantum</td>
</tr>
<tr>
<td>Seqlocks (SEQ)[11]</td>
<td>512 Thread Blocks</td>
<td>Speculative</td>
</tr>
<tr>
<td>Benchmarks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BC[18]</td>
<td>rame99 (1), nas1b24 (2), ex33 (3), c_22 (4)</td>
<td>Commutative, Non-Ordering</td>
</tr>
<tr>
<td>PageRank (PR)[18]</td>
<td>c-37 (1), c-36 (2), ex3 (3), c-40 (4)</td>
<td>Commutative</td>
</tr>
</tbody>
</table>

Our energy model uses GPUWatch [40] for the GPU CUs and McPAT v1.1 [41] for the NoC energy measurements (our architecture more closely resembles a multicore NoC than GPUWatch’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 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 atoms could provide from
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.

<table>
<thead>
<tr>
<th>Benefit</th>
<th>DRF0</th>
<th>DRF1</th>
<th>DRFRlx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoid cache invalidations at atomic loads</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Avoid store buffer flushes at atomic stores</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Overlap atomics in the memory system</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

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

Although 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 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
is divided into multiple components based on the source of energy: GPU core+, 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%.

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 reduce execution time compared to DD0 because DD0 already obtains ownership for the data. However, using unpaired atoms 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
Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems

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

performing the atoms at the L2. This helps GPU coherence overcome its inability to reuse atoms 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 atoms 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 atoms with another construct: scoped synchronization [9, 26, 32, 33, 38]. These models allow the programmer to distinguish some atoms 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 atoms 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 atoms 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 atoms behave on various multicores 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, PLpc'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 atoms.

8 CONCLUSION

Despite more than a decade of research, no acceptable semantics for relaxed atomic results 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 atoms 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 atoms in heterogeneous GPU-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 Blinder, Hans Boehm, Olivier Giroux, Doug Lea, Paul McKenney, Hakan Persson, Marc Snir, and Dmitry Vyukov for helping us identify uses of relaxed atoms and/or offering feedback that improved the quality of this paper.

REFERENCES
