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

MATTHEW D. SINCLAIR, JONATHAN ALSOP, SARITA V. ADVE

Contents

The problem

A new Model

Implementation and Results

Contents

The problem

A new Model

Implementation and Results

Introduction

Moore's law is dead and buried, how to keep making HW better?

- Parallelization
- Specialization

However, **concurrent heterogenous** systems provide a new problem in terms of memory, consistency and coherence models

What are Heterogenous Systems?

From Wikipedia:

"Systems that use more than one kind of processor or cores. These systems gain performance or energy efficiency not just by adding the same type of processors, but by adding dissimilar coprocessors..."

Why Does it Matter?

- Different methods for execution optimization
- Different methods for cache coherence
- Different compiler optimizations

Performance Performance Performance!

Heterogenous systems are built to improve performance. We want to utilize them. We need to:

- Allow as many programs as possible
- Keep programmatic model reasonably close to SC with DRF
- Don’t sacrifice compiler optimizations
- Don’t sacrifice HW memory caches

Cache Coherence is Key

GPUs:

- large number of Compute Units (CUs)
- very simplistic coherence protocols (or none at all)

Adding such coherence protocols result in significant performance dip for target application

Coherence Limitations

- Software driven
- Invalidate the entire cache on non-relaxed synchronization reads
- Write-through all dirty data to the shared LLC (e.g. L2) on non-relaxed synchronization writes
- Require atomics to execute on the LLC

Mitigations

Heterogenous Race Free (HRF) introduced Scoped synchronization – insufficient and overly complex?

DeNovo

- No scoped synchronization
- Uses ownership for writes
- Self invalidation for reads
- Ownership and execution of atomics on L1

Optimizations

Relaxed order atomics allow compiler optimizations that prevent the need to invalidate the cache and write-through to the LLC

Bottom Line

If every atomic operation will cause full L1 cache invalidation, the impact on performance will be critical

The model presented in this paper attempts to allow more cases to use relaxed atomics ➔ no cache invalidation

Contents

The problem

A new Model

Implementation and Results

DRF0

DRF0 classifies all memory variables as either *data* or *atomic*

Data-races on data variables is considered a bug

Programs are kept SC by abolishing all compiler optimizations on atomic variables. It does not support relaxed atomics at all and therefore doesn't allow even optimizations that we would consider sensible (or even remain SC)

DRFrlx

Relaxed Atomic Category	Application
Unpaired	Work Queue
Commutative	Event Counter
Non-Ordering	Flags
Quantum	Split Counter, Reference Counter
Speculative	Seqlocks

DRFrlx

Relaxed Atomic Category	Application
Unpaired	Work Queue
Commutative	Event Counter
Non-Ordering	Flags
Quantum	Split Counter, Reference Counter
Speculative	Seqlocks

Unpaired Atomics – Work Queue

```
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();
    }
  }
```

Unpaired Atomics – Work Queue

- *occupancy* result is used to determine the flow but is validated
- All accesses to are synchronized around the SC load in *dequeue*

Unpaired Atomics – Another example

```
atomic<bool> eventRaised = false;
mutex mx;

while (true) {
	while (!eventRaised.atomic_load(mem_order_relaxed)) {
		sleep(1000);
	}
	mx.Lock()
	if (eventRaised.atomic_load(mem_order_relaxed)) {
		//do something
	}
	mx.Unlock()
}
```

DRF1

- Distinguishes paired from unpaired (relaxed) atomics
- Paired atomics – no optimizations, strongly consistent, ordered access only
- Unpaired atomics – allow optimizations and reordering in regards to non-atomic (data) operations. Keep ordering in regards to other atomics operations

DRF1 – Formal Definition

Synchronization Order 1 ($so_1 \rightarrow$): X $so_1 \rightarrow$ Y iff

✓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 ($hb_1 \rightarrow$): The irreflexive transitive closure of *po* and so_1: $hb_1 = (po \cup so_1)+$

DRF1 – Formal Definition

Race (under DRF1): Two operations X and Y in an execution form a race iff

- ✓X and Y are from different threads
- ✓X and Y conflict
- ✓They are not ordered by hb_1

If X or Y is distinguished as data, we refer to it as a data race

DRF1 – Formal Definition Continued

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.

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.

A Basic Example

```
X_unp = 1     |   while (X_unp != 2);
Y_unp = 1     |   printf(Y_unp);
X_unp = 2     |
```

- SC results require printf to print 1
- DRF1 compliancy guarantees that by keeping *po* between unpaired operations
- By allowing reordering of unpaired operations we might get any other result non-SC result (0)

DRF1 – conclusion

The good

DRF1 provides the benefits of relaxed atomics by removing the ordering constraint between data and unpaired atomics (while preserving SC)

The not good enough

DRF1 constrains unpaired atomics to respect program order with respect to other unpaired atomics

DRFrlx

Relaxed Atomic Category	Application
Unpaired	Work Queue
Commutative	Event Counter
Non-Ordering	Flags
Quantum	Split Counter, Reference Counter
Speculative	Seqlocks

Commutative Atomics – Event Counter

```
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
  }
}
```

Commutative Atomics – Event Counter

- Racing increments are commutative
- Intermediate results are not observable
- Full paired synchronization on join

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

Commutative Atomics – Formal Definition

Commutativity: Two stores 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 result

X and Y form a commutative race iff:

- X and Y form a race
- X or Y is distinguished as a commutative operation
- X and Y are not commutative with respect to each other

Or

- Loaded value is visible to other operations

DRFrlx – first attempt

A program is DRFrlx iff 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

there are no data races or commutative races in the execution

DRFrlx

Relaxed Atomic Category	Application
Unpaired	Work Queue
Commutative	Event Counter
Non-Ordering	Flags
Quantum	Split Counter, Reference Counter
Speculative	Seqlocks

Non-Ordering Atomics - Flags

```
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();
}
```

Non-Ordering Atomics - Flags

- *stop* and *dirty* do not order any other operation
- Global barrier (join) orders any conflicts
- Operation on *stop* and *dirty* can be reordered with respect to other relaxed operations without violating SC

Non-Ordering Atomics – Formal Definition

Conflict Order ($co \rightarrow$): X $co \rightarrow$ Y iff X and Y conflict and X is ordered before Y in T.

Program/Conflict Graph: A directed graph where the vertices are the operations of the execution and the edges represent program order and conflict order

Non-Ordering Atomics – Formal Definition

Ordering Path: 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: An ordering path is valid if all its edges are either:

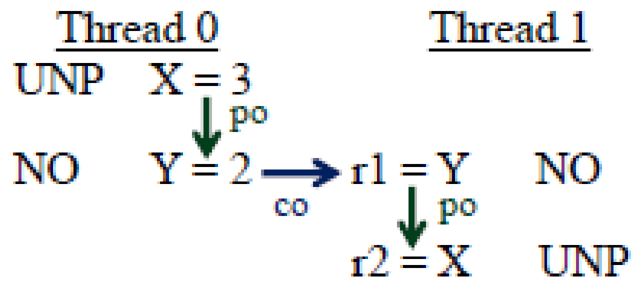
- hb_1
- between atomic accesses to the same address
- between paired or (strictly) unpaired accesses

Non-Ordering Atomics – Formal Definition

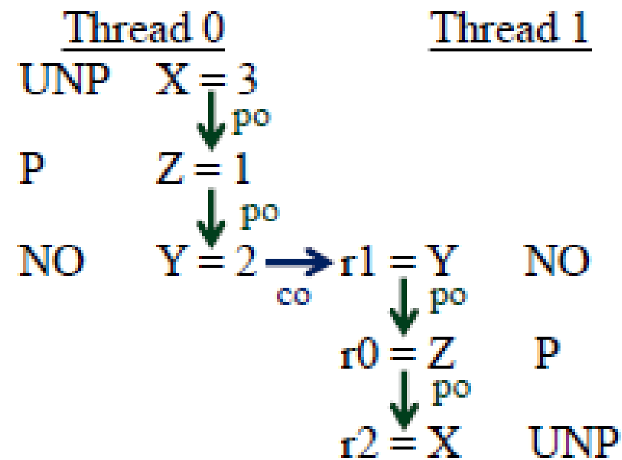
X and Y form a non-ordering race iff:

- X and Y form a race, both atomics, and at least one of them is distinguished as a non-ordering atomic
- X $co \rightarrow$ Y is on an ordering path from A to B, but there is no valid path from A to B.

Non-Ordering Atomics – Program/Conflict Graph



(a)



(b)

DRFrlx – second attempt

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 or non-ordering atomics

there are no data races or commutative races or non-ordering races in the execution

DRFrlx

Relaxed Atomic Category	Application
Unpaired	Work Queue
Commutative	Event Counter
Non-Ordering	Flags
Quantum	Split Counter, Reference Counter
Speculative	Seqlocks

Quantum Atomics – Reference Counters

```
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
```

Quantum Atomics – Split Counter

```
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
```

Quantum Atomics – Characteristics

- Not commutative – intermediate results are observed
- Not non-ordering – Read/Add Increase/Decrease affect other operations
- Split Counters increase concurrently, at every point, the sum is an estimate
- Reference Counters can only reach 0 once

Quantum Atomics – Quantum Transformation

- Replace quantum reads with a random value
- Replace quantum writes with a random value
- *Receive quantum-equivalent program*

Quantum Atomics – Formal Definition

Quantum Race: Two operations, X and Y form a quantum race iff:

✓ They form a race

✓ Exactly one of them is a quantum atomic

DRFrlx – third attempt

A program is DRFrlx iff for every SC execution of the program:

all operations can be identified by the system as either data or as paired, unpaired, commutative, non-ordering or quantum atomics

there are no data races or commutative races, non-ordering races, or quantum races in the execution

DRFrlx – third attempt

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

DRFrlx

Relaxed Atomic Category	Application
Unpaired	Work Queue
Commutative	Event Counter
Non-Ordering	Flags
Quantum	Split Counter, Reference Counter
Speculative	Seqlocks

Speculative Atomics - Seqlocks

```
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);
}
```

Speculative Atomics - Seqlocks

- Data must be atomic
- Data Stores race with loads
- Loads discarded if race occurred → Final result is consistent

Speculative Atomics – Formal Definition

X and Y, form a speculative race iff:

✓They form a race

✓X or Y is a speculative atomic

✓Both operations are stores

or

✓The (speculative) result is observed by another operation

DRFrlx – fourth and final attempt

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

there are no data races, commutative races, non-ordering races, quantum races, or speculative races in the execution

Contents

The problem

A new Model

Implementation and Results

Model Formalization

- New keywords:
unpaired, commutative, non-ordering, quantum and speculative
- Simulator was used to formalize the model
- Simulations successfully identified races in SC executions or produce non-SC executions permitted by the model

Correctness Theorem

"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"*

Breaking it down

- Heterogeneous
- DRF1 compliant
- happens-before
- per-location SC (for atomics)
- DRFrlx atomics as Data

Simulation Parameters

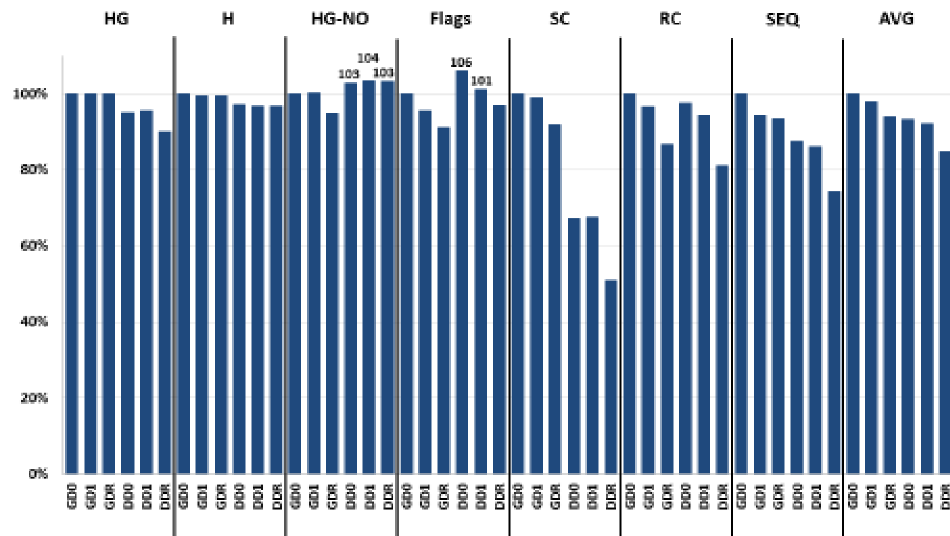
CPU Parameters	
Frequency	2 GHz
Cores	1
GPU Parameters	
Frequency	700 MHz
CUs	15
Memory Hierarchy Parameters	
L1 size (8 banks, 8-way assoc.)	32 KB
L2 size (16 banks, NUCA)	4 MB
Store buffer size	128 entries
L1 MSHRs	128 entries
L1 hit latency	1 cycle
Remote L1 hit latency	35−83 cycles
L2 hit latency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

Table 2: Simulated heterogeneous system parameters.

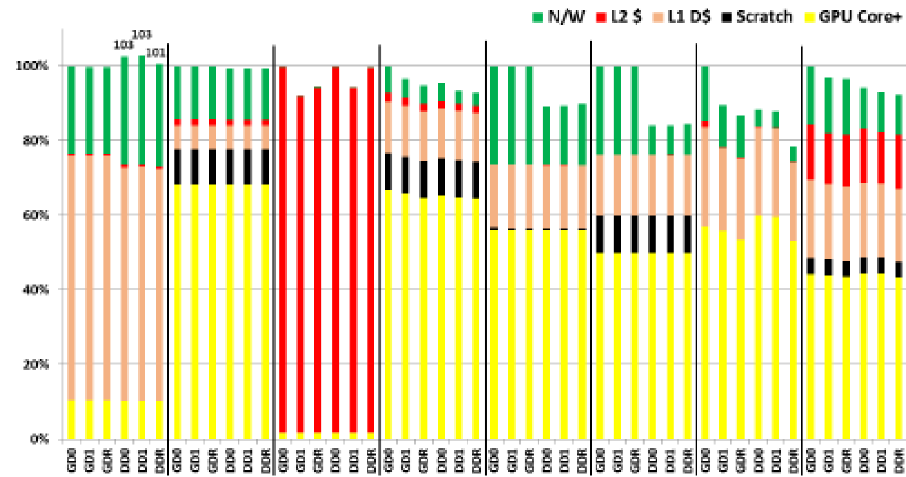
Benchmark	Input	Atomic Types
Microbenchmarks		
Hist (H)[50]	256 KB, 256 bins	Commutative
Hist_global (HG)[50]	256 KB, 256 bins	Commutative
HG-Non-Order (HG-NO)	256 KB, 256 bins	Non-Ordering
Flags[61]	90 Thread Blocks	Commutative, Non-Ordering
SplitCounter (SC)[44]	112 Thread Blocks	Quantum
RefCounter (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.

Simulation Results

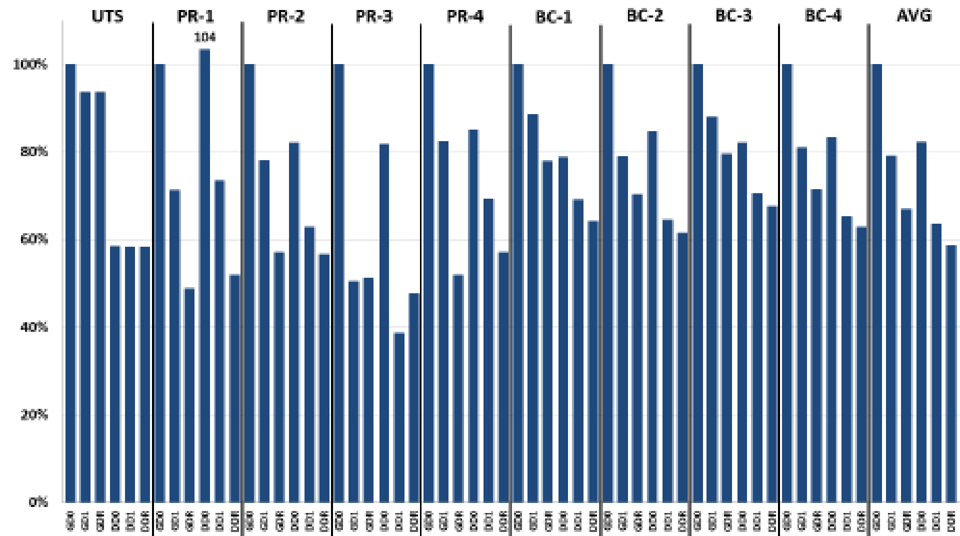


(a) Execution time

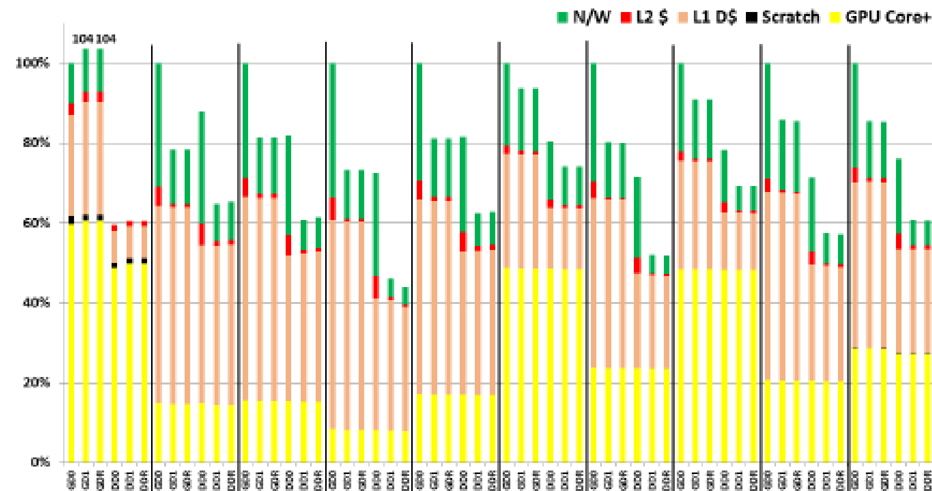


(b) Dynamic energy

Simulation Results



(a) Execution time



(b) Dynamic energy

Simulation Results - Summary

- Micro-benchmarks – usually around 15% improvement from DRFrlx
- Benchmarks – BC and PR show much more significant improvements (up to 37%)
- Combined with DeNovo the impact becomes more significant (> 50% for some cases)

Summary

New consistency model to allow further relaxing atomic variables while keeping the program semantically close to SC

➔ **Theoretically less cache invalidations**

Simulation results show slight improvements, more if combined with DeNovo, probably more work to be done in this direction