EECS 570
Lecture 17
HSA & Spandex
Winter 2020
Prof. Satish Narayanasamy
http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by B. Beckman & S. Adve
Readings

For today:
- Alsop et al - Spandex - ISCA 2018

For Monday 3/30:
- Jerger & Peh - On-Chip Networks - Chapter 3
- Kim, Dally, & Abts - Flattened Butterfly : A Cost-Efficient Topology for High-Radix Networks
Heterogeneous System Architecture

Slides courtesy B. Beckman
ENABLING EFFICIENT COMMUNICATION IN LARGE HETEROGENEOUS PROCESSORS

BRAD BECKMANN
AMD RESEARCH
MARCH 10, 2015
AGENDA

Why Heterogeneous Processors?

Current Heterogeneous Processors: AMD Kaveri and HSA

Research: Scalable Communication and Synchronization
CPU FREQUENCY TREND

Mark Horowitz, ISSCC 2014 Keynote, “Computing’s Energy Problem (and what we can do about it)”

Data from http://cpudb.stanford.edu
CPU POWER DENSITY TREND

Mark Horowitz, ISSCC 2014 Keynote, “Computing’s Energy Problem (and what we can do about it)"

Data from http://cpudb.stanford.edu
POWER & ENERGY ARE THE LIMITING FACTORS FOR COMPUTING

- Cooling
- Battery life
- Power delivery
- Operating expense
TWO WAYS TO IMPROVE ENERGY EFFICIENCY

▲ Increase parallelism
- energy/op vs. performance is non-linear

Reduce overheads
- Programmability
- Data movement

Mark Horowitz, ISSCC 2014
Keynote, “Computing’s Energy Problem
(and what we can do about it)”
WHY HETEROGENEOUS PROCESSORS?

Increased parallelism
  - Many ops per instruction (e.g., 64)
  - Many threads per compute engine (tens)
  - Many compute engines per GPU (tens)

Reduced overheads
  - SIMD operation: one instruction → many ops
  - Simple in-order pipeline
  - Very little cache capacity per ALU

Bonus! You already have one in your desktop/laptop/tablet/smart phone...
WHAT’S THE CATCH?

- Application must have parallelism
- SIMD efficiency depends on extracting data parallelism
- Great for some algorithms, not so great for others
- Typically programmed in specialized languages
- Require explicit data copying to GPU memory
- Need large tasks to amortize high overheads

Fundamental!
- GPUs are specialized for these types of workloads.
- Heterogeneity (CPUs + GPUs) needed to cover a full range of workloads.

Superficial!
- Merely historical artifacts of traditional approaches
- AMD and HSA to the rescue!
Current Heterogeneous Processors: AMD Kaveri and HSA
WHAT IS HSA?
HETEROGENEOUS SYSTEM ARCHITECTURE

Processor design that makes it easy to harness the entire computing power of an APU for faster and more power-efficient devices, including personal computers, tablets, smartphones, and cloud servers.
KEY FEATURES OF HSA

hUMA  Heterogeneous Unified Memory Architecture

hQ    Heterogeneous Queuing

HSAIL  HSA Intermediate Language
**TRADITIONAL DISCRETE GPU**

- Separate memory
- Separate addr space
  - No pointer-based data structures
- Explicit data copying
  - High latency
  - Low bandwidth
- Need lots of compute on GPU to amortize copy overhead
- Very limited GPU memory capacity
hUMA UNIFIED MEMORY

Unified address space
- GPU uses user virtual addr
- Fully coherent

No explicit copying
- Data move on demand

Pointer-based data structures shared across CPU & GPU

Pageable virtual addresses
- No GPU capacity constraints
TRADITIONAL COMMAND AND DISPATCH FLOW
hQ COMMAND AND DISPATCH FLOW

- User-mode application talks directly to the hardware
  - HSA Architected Queuing Language (AQL) defines vendor-independent format
  - No system call
  - No kernel driver involvement

- Hardware scheduling

- Greatly reduced dispatch overhead
  - less overhead to amortize
  - profitable to offload smaller tasks

- GPU kernels can self-enqueue additional tasks for dynamic parallelism
PROGRAMMING LANGUAGES PROLIFERATING ON HSA

- OpenCL™ App
- Java App
- C++ AMP App
- Python App
- OpenCL Runtime
- Java JVM (Sumatra)
- Various Runtimes
- Fabric Engine RT
- HSA Helper Libraries
- HSA Core Runtime
- HSA Finalizer
- Kernel Fusion Driver (KFD)

HSAIL
HSA BUILDING BLOCKS

HSA Hardware Building Blocks
- Shared Virtual Memory
  - Single address space
  - Coherent
  - Pageable
  - Fast access from all components
  - Can share pointers
- Architected User-Level Queues
- Signals
- Context Switching
- Platform Atomics
- Defined Memory Model

HSA Software Building Blocks
- HSAIL
  - Portable, parallel compiler IR
  - Instruction definition
- HSA Runtime
  - Create queues
  - Allocate memory
  - Device discovery
- Multiple high level compilers
  - CLANG/LLVM/HSAIL
  - C++, OpenMP, OpenACC, Python, OpenCL™, etc

Industry standard, architected requirements for how devices share memory and communicate with each other

Industry standard compiler IR and runtime to enable existing programming languages to target the GPU
# HSA Foundation Today

A Growing and Powerful Family

<table>
<thead>
<tr>
<th>Founders</th>
<th>AMD</th>
<th>ARM</th>
<th>MediaTek</th>
<th>Texas Instruments</th>
<th>Imagination</th>
<th>Qualcomm®</th>
<th>Samsung</th>
</tr>
</thead>
<tbody>
<tr>
<td>Promoters</td>
<td>LG Electronics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Supporters</td>
<td>Arteris®</td>
<td>codeplay®</td>
<td>FABRIC ENGINE®</td>
<td>Multicore ware</td>
<td>Sandia National Laboratories</td>
<td>Linaro</td>
<td>Khonoul</td>
</tr>
<tr>
<td>Contributors</td>
<td>ANALOG DEVICES</td>
<td>apical</td>
<td>symbio</td>
<td>ORACLE®</td>
<td>OMP®</td>
<td>CANONICAL</td>
<td>SYNOPSYS®</td>
</tr>
<tr>
<td>Universities</td>
<td>National Tsinghua University</td>
<td>NTHU Programming Language Lab</td>
<td>NTHU System Software Lab</td>
<td>University of Bristol</td>
<td>University of Edinburgh informatics</td>
<td>ILLINOIS COMPUTER SCIENCE</td>
<td></td>
</tr>
</tbody>
</table>
HSA FEATURES OF "KAVERI"

UNLOCKING ALL OF KAVERI’S GFLOPS

- Access to full potential of Kaveri’s APU compute power

EQUAL ACCESS TO ENTIRE MEMORY

- GPU and CPU have uniform visibility into entire memory space

ALL PROCESSORS EQUAL

- GPU and CPU have equal flexibility to be used to create and dispatch work items

![Graph showing equal access to memory and unlocking all of Kaveri’s GFLOPS](image-url)
KEY CHALLENGES FOR SCALABILITY

Avoid unnecessary communication and coherence overhead
- Coherent shared memory (hUMA) is a key part of HSA, but hardware overhead can be significant
- “Heterogeneous system coherence for integrated CPU-GPU systems”, Power et al., Micro 2013

**Today’s Focus:** Reduce synchronization penalties
- Synchronizing across all threads on a large APU can be expensive
  - e.g., making a write operation globally visible
- Yet many synchronization operations have locality
Parallel synchronization semantics
- acquire: pull latest data (to me)
- release: push latest data (to others)

Scopes bound synchronization:
- Smaller scope $\rightarrow$ less synchronization overhead

<table>
<thead>
<tr>
<th>scope</th>
<th>abbrev.</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>work-item</td>
<td>wi</td>
<td>Like a CPU thread</td>
</tr>
<tr>
<td>wavefront</td>
<td>wv</td>
<td>work-items executing in lockstep on SIMD</td>
</tr>
<tr>
<td>work-group</td>
<td>wg</td>
<td>wavefronts executing on the same CU</td>
</tr>
<tr>
<td>component</td>
<td>cmp</td>
<td>work-groups executing on the same GPU</td>
</tr>
<tr>
<td>system</td>
<td>sys</td>
<td>All work-items/threads in the process</td>
</tr>
</tbody>
</table>
void incX_component() {
    while (!CAS_acq_cmp(&L, 0, 1));
    X = X + 1;
    st_rel_cmp(&L, 0);
}

void incX_workgroup() {
    while (!CAS_acq_wg(&L, 0, 1));
    X = X + 1;
    st_rel_wg(&L, 0);
}
SCOPED SYNCHRONIZATION’S STRENGTHS

On current hardware, \( \text{wg scope} \) can yield \( >20\% \) speedup over \( \text{cmp scope} \)
HETEROGENEOUS-RACE-FREE (HRF) MEMORY MODELS
HOWER ET AL. ASPLOS 2014

- First to formalized scoped synchronization

- Transitivity matters
  - A sync B, B sync C → Does A sync C?
  - Classic tradeoff: simplicity vs. performance

- Enables task donation by an intermediary

- Introduced 2 memory models:
  - HRF-direct: disallows transitivity
  - HRF-indirect: allows transitivity
CASE STUDY – TASK-SHARING RUNTIME RESULTS

Performance Normalized to HRF-direct

input sets: uts_t1, uts_t2, uts_t4, uts_t5

- HRF-direct
- HRF-indirect

Graph shows the performance of different input sets normalized to HRF-direct, comparing HRF-direct and HRF-indirect.
SCOPED SYNCHRONIZATION’S LIMITATIONS

- **Dynamic local sharing**: some threads access shared data less frequently than others in an ad-hoc manner
- **Example: work stealing**
Insight: $w_g_1$ needs to *trigger* the promotion of scope $w_g_0$

Paper discusses the HW support for scope promotion
Prior memory models: HRF-direct, HRF-indirect

- **Invariant**: acquire/release pair must occur at the same scope

**Three new memory orders:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>remoteAcquire</td>
<td>Promote the scope of last release to the scope of this acquire, then perform acquire</td>
</tr>
<tr>
<td>remoteRelease</td>
<td>Promote the scope of next acquire to the scope of this release, then perform release</td>
</tr>
<tr>
<td>remoteAcquire+Release</td>
<td>Combine remote acquire &amp; remote release</td>
</tr>
</tbody>
</table>

**Promotion Semantic**

- **work-item 0**
  - `st(V, 2)`
  - `st_rel_wg(L, 0)`
  - Synchronizes-with relationship

- **work-item 1 (different wg)**
  - `cas_rm_acq_cmp(&L, 0, 1)`
  - RACE!
  - `ld(R1, V)`
  - Synchronizes-with relationship

**Promotion**

- Promote the scope of the last release to the scope of this acquire, then perform acquire.
- Promote the scope of the next acquire to the scope of this release, then perform release.
- Combine remote acquire & remote release.
METHODOLOGY

- Prototyped remote scoped synchronization in gem5
  - Extended with internal GPU model

- Refactored 3 Pannotia workloads to retrieve graph nodes from task queues
  - SSSP, Color, PageRank (each run with 3-4 inputs)
RESULTS

<table>
<thead>
<tr>
<th>scenario</th>
<th>Scope of sync.?</th>
<th>Work stealing?</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline</td>
<td>global</td>
<td>no</td>
</tr>
<tr>
<td>scope-only</td>
<td>local</td>
<td>no</td>
</tr>
<tr>
<td>steal-only</td>
<td>global</td>
<td>yes</td>
</tr>
<tr>
<td>rem-sync</td>
<td>local</td>
<td>yes</td>
</tr>
</tbody>
</table>

Speedup

- baseline
- scope-only
- steal-only
- rem-sync

SSSP-1, SSSP-2, SSSP-3, color-1, color-2, color-3, color-4, PR-1, PR-2, PR-3, geo. mean

MARCH 10, 2015
CONCLUSIONS

Heterogeneous systems are here to stay
- Specialization to reduce overheads and increase parallelism is needed to address power/energy limits
- GPUs are an efficient specialization for highly data-parallel workloads
  - Including, but definitely not limited to graphics

AMD and HSA are making heterogeneous systems more programmable
- Unified coherent memory (hUMA), user-level queuing (hQ), standard intermediate lang (HSAIL)
- A standard platform for a wide variety of languages: C++, Java, Python, ...
- Available now with AMD’s “Kaveri” APU and open-source SW stack

AMD Research is pushing ahead to define the heterogeneous systems of the next decade
- Ex. scalable communication and synchronization
- Many other aspects under investigation as well
Spandex: A Flexible Interface for Efficient Heterogeneous Coherence

Johnathan Alsop*, Matthew D. Sinclair*†‡, Sarita V. Adve*

*Illinois, †AMD, ‡Wisconsin

Sponsors: NSF, C-FAR, ADA (JUMP center by SRC, DARPA)
Specialized architectures are increasingly important in all compute domains
Specialization Requires Better Memory Systems

Traditional heterogeneity:

- CPU
- Accelerator

Shared coherent memory:

- CPU
- Accelerator

Existing solutions: complex and inflexible

- No fine-grain synchronization
- No irregular access patterns
- Wasteful data movement

✓ Fine-grain synchronization
✓ Irregular access
✓ Implicit data reuse
Heterogeneous devices have diverse memory demands

- Spatial locality
- Temporal locality
- Fine-grain Synch
- Latency Sensitivity
- Throughput Sensitivity
Heterogeneous devices have diverse memory demands

Typical CPU workloads:
- fine-grain synch
- latency sensitive
Heterogeneous devices have diverse memory demands

Typical **GPU** workloads: spatial locality, throughput sensitive
# MESI protocol fits CPU workloads

<table>
<thead>
<tr>
<th>Properties</th>
<th>MESI</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Granularity</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Invalidation</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Updates</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Good for:**

- **CPU**
## GPUs prefer simpler protocols

<table>
<thead>
<tr>
<th>Properties</th>
<th>MESI</th>
<th>GPU coherence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Granularity</td>
<td>Line</td>
<td></td>
</tr>
<tr>
<td>Invalidation</td>
<td>Writer-invalidate</td>
<td></td>
</tr>
<tr>
<td>Updates</td>
<td>Ownership</td>
<td></td>
</tr>
</tbody>
</table>

**Good for:**
- CPU
- GPU coherence
- GPUs prefer simpler protocols
- MESI
- Good for: GPU
DeNovo is a good fit for CPU and GPU

<table>
<thead>
<tr>
<th>Properties</th>
<th>MESI</th>
<th>GPU coherence</th>
<th>DeNovo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Granularity</td>
<td>Line</td>
<td>Reads: Line</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Writes: Word</td>
<td></td>
</tr>
<tr>
<td>Invalidation</td>
<td>Writer-invalidate</td>
<td>Self-invalidate</td>
<td></td>
</tr>
<tr>
<td>Updates</td>
<td>Ownership</td>
<td>Write-through</td>
<td></td>
</tr>
</tbody>
</table>

**Good for:**
- CPU
- GPU
- CPU or GPU
Existing Solutions: Inflexible and Inefficient

Examples: ARM ACE, IBM CAPI, AMD APU
Existing Solutions: Inflexible and Inefficient

If the glove doesn’t fit... There’s limited benefit!

Examples: ARM ACE, IBM CAPI, AMD APU
Existing Solutions: Inflexible and Inefficient

If the glove doesn’t fit... There’s limited benefit!

Examples: ARM ACE, IBM CAPI, AMD APU
Spandex: Flexible Heterogeneous Coherence Interface

Adapts to exploit individual device’s workload attributes
Better performance, lower complexity
⇒ Fits like a glove for any heterogeneous system!
Spandex Overview

Key Components

- Flexible device request interface
- DeNovo-based LLC
- External request interface

Device may need a translation unit (TU)
Key Components

• Flexible device request interface

• DeNovo-based LLC

• External request interface

Device may need a translation unit (TU)
### Device Request Interface

<table>
<thead>
<tr>
<th>Action</th>
<th>Request</th>
<th>Indicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>ReqV</td>
<td>Self-invalidation</td>
</tr>
<tr>
<td></td>
<td>ReqS</td>
<td>Writer-invalidation</td>
</tr>
</tbody>
</table>

Requests also specify granularity and (optionally) a bitmask.
Spandex Overview

Key Components

- Flexible device request interface
- DeNovo-based LLC
- External request interface

Device may need a translation unit (TU)
Spandex LLC

- States: I, V, O, S
- Allocation at line granularity
- Ownership at word granularity
- Data field tracks owner ID
- May generate requests to owner/sharer

✓ No false sharing
✓ Non-blocking ownership transfer
Spandex Overview

Key Components

- Flexible device request interface
- DeNovo-based LLC
- External request interface

Device may need a translation unit (TU)
External Request Interface

- **CPU**
  - MESI L1
  - States: I, S, O
- **GPU**
  - GPU coh. L1
  - States: I, V
- **DeNovo L1**
  - States: I, V, O

- **External Request**
  - Must handle if supports state:
    - ReqV: O
    - ReqO: O
    - ReqO+data: O
    - RvkO: O
    - Inv: S
    - ReqS: S and O

- **Translation Unit** may implement functionality if not supported by device
Evaluation: Configurations

<table>
<thead>
<tr>
<th>Configuration</th>
<th>LLC protocol</th>
<th>CPU protocol</th>
<th>GPU protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMG</td>
<td>Hierarchical MESI</td>
<td>MESI</td>
<td>GPU coherence</td>
</tr>
<tr>
<td>HMD</td>
<td>Hierarchical MESI</td>
<td>MESI</td>
<td>DeNovo</td>
</tr>
</tbody>
</table>

CPU-GPU workloads from **Pannotia** and **Chai** benchmark suites
Evaluation: CPU-GPU Applications

- Different workloads prefer different protocols
- Spandex flexibility ⇒ consistently better execution time (avg 16% lower)
Evaluation: CPU-GPU Applications

- Spandex flexibility ⇒ consistently better NW traffic (avg 27% lower)
Conclusion and Future Work

Future Work: exploit SW or HW hints about data access patterns

- Dynamic Spandex request selection
- Producer-consumer forwarding
- Extended granularity flexibility

⇒ Simple, Flexible, Efficient
Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems

Matthew D. Sinclair, Johnathan Alsop, Sarita V. Adve
University of Illinois @ Urbana-Champaign
hetero@cs.illinois.edu

Paper Available at: http://rsim.cs.illinois.edu/pubs.html
“Everyone (thinks they) can cook” use relaxed atomics (RAts)

Correctness Health code violations:

Incorrect usage ❄️ No formal definition 🔐 Not portable

Hard to debug 🐜 Out-of-thin-air value 🐜
Consistency is Complex

“If you think you understand quantum computers, it’s because you don’t. Quantum computing is actually harder than memory consistency models.”

- Luis Ceze, video in ISCA ’16 Keynote

Memory consistency: gold standard for complexity

Relaxed atomics add even more complexity
No Formal Specification for Relaxed Atomics

C++17 "specification" for relaxed atomics
  • Races that don't order other accesses
  • Implementations should ensure no “out-of-thin-air” values are computed that circularly depend on their own computation

"C++ (relaxed) atomics were the worst idea ever. I just spent days (and days) trying to get something to work. ... My example only has 2 addresses and 4 accesses, it shouldn’t be this hard. Can you help?"

- Email from employee at major research lab

Formal specification for relaxed atomics is a longstanding problem
Why Use Relaxed Atomics?

![Bar Chart]

- Speedup
  - UTS: 0X
  - Histogram: 27X
  - CudaCuts: 28X
  - AP: 44X
  - BarnesHut: 99X
  - BC: 64X
  - PageRank: 27X
  - MRI-Gridding: 28X
  - Fluidanimate: 99X

But generally use simple, SW-based coherence.

Cost of staying away from relaxed atomics too high!
Our Approach

• Previous work
  • Goal: formal semantics for all possible relaxed atomics uses
  • Unsuccessful despite ~15 years of effort

• Insight: analyze how real codes use relaxed atomics
  • What are common uses of relaxed atomics?
  • Why do they work?
  • Can we formalize semantics for them?
Contributions

- Identified common uses of relaxed atomics
  - Work queues, event counters, ref counters, seqlocks, ...

- Data-race-free-relaxed (DRFrlx) memory model:
  - Sequentially consistent (SC) centric semantics + efficiency

- Evaluated benefits of using relaxed atomics
  - Up to 53% less cycles (33% avg), 40% less energy (20% avg)

Everyone can safely use RAts
Outline

• Motivation
• Background
  • Atomics
  • Prior Approaches
• Data-race-free-relaxed
• Results
• Conclusion
Atomics in Data-Race-Free-0

- Default: DRF0 [ISCA ‘90]
  - Identify all races as synchronization accesses (C++: atomics)
  - All atomics order data accesses
  - Atomics order other atomics

⇒ Ensures SC semantics if no data races

Precludes data reuse and overlapping atomics
Atomics in Data-Race-Free-1

- Data-race-free-1 (DRF1): unpaired atomics [TPDS ‘93]
  + Unpaired atomics do not order data accesses
  - Atomics order other atomics
    ⇒ Ensures SC semantics if no data races

Can reuse data but cannot overlap atomics
Relaxed Atomics

- Relaxed atomics [PLDI ‘08]
  + Do not order data or other atomics
  + Reorder, overlap with all other memory accesses

But can violate SC and no formal specification
Outline

• Motivation
• **Background**
  • Atomics
  • Prior Approaches
• Data-race-free-relaxed
• Results
• Conclusion
C++ Support for Relaxed Atomics

\[ X = Y = 0 \]

**Thread 1**
- \( R1 = \text{ATOM.LD}(X) \) // RLX
- \( \text{ATOM.ST}(Y, R1) \) // RLX

**Thread 2**
- \( R2 = \text{ATOM.LD}(Y) \) // RLX
- \( \text{ATOM.ST}(X, R2) \) // RLX

- C++11 specification: Relaxed atomic loads can load
  - Initial value (0)
  - Or the value of the store in the other thread
- Allows out-of-thin-air values
  - Accesses have a circular dependency

Hard to forbid out-of-thin-air and allow legitimate opts
Prior Approaches for Relaxed Atomics

- C++
  - Preclude some desired optimizations [Boehm MSPC ‘15]
    - Don’t allow relaxed load \(\rightarrow\) store to be reordered
    - High overhead for architectures with weaker memory models
  - C++17 specification:
    - Systems should not produce out-of-thin-air values

- HSA
  - Global dependence ordering prohibits out-of-thin-air values
  - SC not guaranteed for programs with relaxed atomics
Outline

• Motivation
• Background
• Data-race-free-relaxed
• Results
• Conclusion
Identifying Relaxed Atomic Use Cases

• Our Approach
  • What are common uses of relaxed atomics?
  • Why do they work?
  • Can we formalize semantics for them?

• Contacted vendors, developers, and researchers

How do relaxed atomics work in Event Counters?
• Threads concurrently update counters
  • Read part of a data array, updates its counter
• Threads concurrently update counters
  • Read part of a data array, updates its counter
  • Increments race, so have to use atomics
Threads concurrently update counters
- Read part of a data array, updates its counter
- Increments race, so have to use atomics

Commutative increments: order does not affect final result
How to formalize?
Incorporating Commutativity Into DRFrlx

- New relaxed atomic category: commutative
- Formalism:
  - Accesses are commutative
  - Intermediate values must not be observed

⇒ Final result is always SC
Commutative Definitions for an SC Execution

• Result of execution: memory state at end of execution

• Commutativity
  • Two stores/RMWs to a memory location $M$ are commutative if:
    • Can be performed in any order and
    • Yield the same value for $M$

• X and Y form a commutative race iff:
  1. X and Y form a race,
  2. At least one of X and Y is distinguished as commutative, &
  3. X and Y are:
    • Not commutative or
    • Value loaded by either is used by another instr. in its thread
Commutative Program and Model Definitions

- **DRFrlx Program**
  - A program is DRFrlx iff for every SC execution of program:
    - All opers. identified as data, paired, unpaired, or commutative
    - No data races or **commutative races** in the execution

- **DRFrlx Model**
  - A system obeys DRFrlx iff for every SC execution of program:
    - *Result of every execution* of DRFrlx program is result of an SC execution of the program

---

How do relaxed atomics work in Seqlocks?
Speculative – Seqlocks

- Use shared *sequence number* instead of a *lock*
- Data accesses race, must use atomics
- Readers read sequence number before/after data access(es)
- Writers update sequence number and data

**Readers’ seq values don’t match, retry**
Speculative – Seqlocks (Cont.)

- Use shared sequence number instead of a lock
- Data accesses race, must use atomics
- Readers read sequence number before/after data access(es)
- Writers update sequence number and data

Retry non-SC data accesses – final result always SC
Incorporating Speculative Into DRFrlx

- New relaxed atomic category: speculative
- Formalism:
  - Values returned by racy speculative loads never used

$\Rightarrow$ DRFrlx: final result is always SC

What about the other use cases?
Incorporating Other Use Cases Into DRFrlx

**How to incorporate overlapped atomics that do not order?**

**How to incorporate SC violations in approximable applications?**

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Category</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work Queues</td>
<td>Unpaired</td>
<td>SC</td>
</tr>
<tr>
<td>Flags</td>
<td>Non-Ordering</td>
<td></td>
</tr>
<tr>
<td>Event Counters</td>
<td>Commutative</td>
<td>Final result always SC</td>
</tr>
<tr>
<td>Seqlocks</td>
<td>Speculative</td>
<td></td>
</tr>
<tr>
<td>Split Counters</td>
<td>Quantum</td>
<td>SC-centric: non-SC parts isolated</td>
</tr>
<tr>
<td>Ref Counters</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

- Cost of avoiding relaxed atomics too high
- Difficult to use correctly: no formal specification
- Insight: Analyze how real codes use relaxed atomics

DRFrlx: SC-centric semantics + efficiency

Everyone can safely use RAts