New Programming Abstractions for Concurrency

Torvald Riegel
Red Hat
12/04/05

Concurrency and atomicity

C++11 atomic types	Transactional Memory
Provide atomicity for concurrent accesses by different threads Both based on C++11 memory model	
Single memory location	Any number of memory locations
Low-level abstraction, exposes HW primitives	High-level abstraction, mixed SW/HW runtime support

- Talk’s focus is on C++ but C11 has (very) similar support

Atomic types and accesses

- Making a type T atomic: atomic<T>
- Load, store:
  - `atomic<int> a; a = a + 1; a.store(a.load() + 1);`
- CAS and other atomic read-modify-write:
  - ```
    int exp = 0; a.compare_exchange_strong(exp, 1);
    previous = a.fetch_add(23);
    ```
- Sequential consistency is default
  - All s-c ops in total order that is consistent with per-thread program orders
- Other weaker memory orders can be specified
  - `locked_flag.store(false, memory_order_release);`
  - Orders: acquire, acq_rel, release, relaxed, seq_cst, consume


    ```

Why a memory model?

- Defines multi-threaded executions (undefined pre C++11)
  - Normal, nonatomic memory accesses
  - Ordering of all operations enforced by atomic/synchronizing memory accesses
- Common ground for programmers and compilers
  - Formalizations of the model exist (Batty et al. [1])
  - Base for testing tools, compiler testing, verification, ...
- Unified abstraction for HW memory models
  - Portable concurrent code (across HW and compilers)
  - Simpler than several HW memory models

Happens-before (HB)

- Order of operations in a particular execution of a program
- Derived from / related to other relations:
  - Sequenced-before (SB): single-thread program order
  - Reads-from: which store op's value a load op reads
  - Synchronizes with (SW)
    - Example: acquire-load reads from release-store (both atomic)
  - Total orders for seq_cst operations, lock acquisition/release
  - Simplified: HB = transitive closure of SB U SW
- Compiler generates code that ensures some valid HB:
  - Must be acyclic and consistent with all other relations/rules
  - Generated code ensures HB on top of HW memory model

Data-race freedom (DRF)

- Data race: Nonatomic accesses, same location, at least one a store, not ordered by HB
- Any valid execution has a data race?
  => Undefined behavior

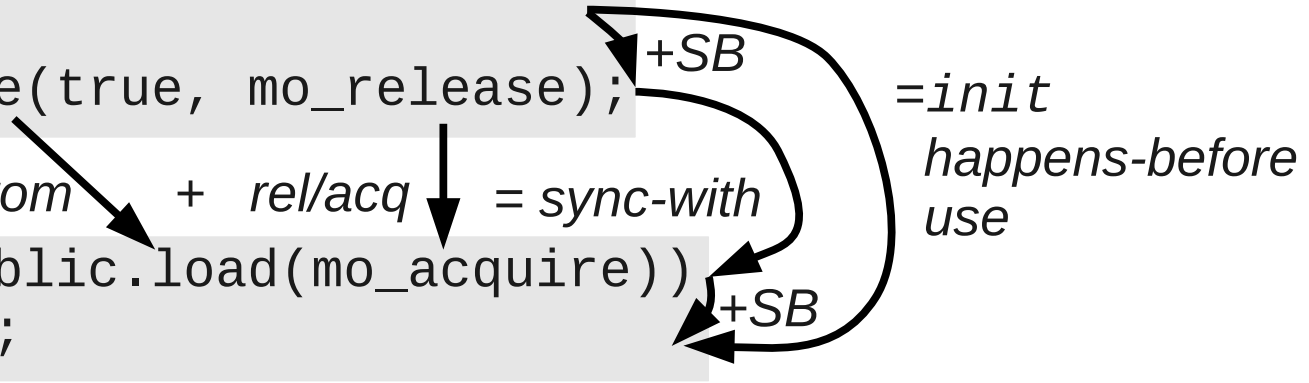
- Programs must be DRF
  - Allows compiler to optimize
- Compiler must preserve DRF
  - Access granularity (e.g., bitfields)
  - Speculative stores, reordering, hoisting, ...

Example: Programmer's POV

- Publication:

Thread1
```
init(data);
data_public.store(true, mo_release);
```

Thread2
```
if (data_public.load(mo_acquire))
  use(data);
```



- Programmers must beware of data races:

```
temp = data;
if (data_public.load(mo_acquire))
  use(temp);
```

Races with init
Program behavior is undefined

Example: Can the compiler hoist a load?

- Can we load `data` earlier than the conditional?
-
```
if (data_public.load(mo_acquire))
  use(data);
```
  - No! Introduces data race, defeats acquire HW barrier.
-
```
if (data_public.load(mo_relaxed))
  use(data);
```
  - Yes! `mo_relaxed` doesn't contribute to happens-before.
-
```
if (data_public.load(mo_acquire))
  use(data);
use_again(data);
```
  - Yes!? `data` always loaded, nonatomic, no ordering by other sync. Can assume DRF program, so no concurrent write.

GCC status

- Atomics: efficient code generated on the major archs
  - New `__atomic*` builtins (replace `__sync*`)
  - Frontend: C++ works, C11 atomics are WIP
  - libatomic: library fallback for non-native atomic ops
  - Issues:
    - Back-off in CAS loops?
    - CAS_strong on LL/SC archs vs. C++ progress guarantees
    - libatomic: Don’t use 2-word CAS for 2-word atomics?
- Memory model: seems to work, more or less
  - Need to audit GCC passes
    - Is optimizing across atomics worthwhile?
  - Need comprehensive testing to prevent future regressions

Transactional Memory (TM): What is it?

- TM is a programming abstraction
  - Declare that several actions are atomic
  - But don’t have to implement how this is achieved

- TM implementations
  - Are generic, not application-specific
  - Several implementation possibilities:
    - STM: Pure SW TM algorithms
      - Blocking or nonblocking, fine- or coarse-granular locking, ...
    - HTM/HyTM: using additional HW support for TM
      - E.g., Intel TSX, AMD ASF

Transactional language constructs for C/C++

- Declare that compound statements, expressions, or functions must execute atomically
  - `__transaction_atomic { if (x < 10) y++; }`
  - No data annotations or special data types required
  - Existing (sequential) code can be used in transactions
  - Language-level txns are a portable interface for HTM/STM
    - HTM support can be delivered by a TM runtime library update

- Draft specification for C++ [2]
  - HP, IBM, Intel, Oracle, Red Hat
  - C++ standard study group on TM (SG5)
  - C will be similar (GCC supports txns in C and C++)
  - Feedback welcome!

How to synchronize with transactions?

- TM extends the C++11 memory model
  - All transactions totally ordered
  - Order contributes to Happens-Before (HB)
  - TM implementation ensures some valid order that is consistent with HB
  - Does not imply sequential execution!

- Data-race freedom still required

```
init(data); __transaction_atomic { data_public = true; }
```

Correct:

```
__transaction_atomic {
    if (data_public) use(data); }
```

Incorrect:

```
__transaction_atomic { temp = data;  // Data race
    if (data_public) use(temp); }
```


Atomic vs. relaxed transactions

	Atomic	Relaxed
Atomic wrt.:	All other code	Only other transactions
Restrictions on txnal code:	No other synchronization (conservative, WIP)	None
Keyword:	`__transaction_atomic`	`__transaction_relaxed`

- Restrictions/safety of atomic txns checked at compile time
  - Compiler analyzes code
  - Additional function attributes to deal with multiple CUs
  - WIP: dynamic checking at runtime instead of static checking (optional)

TM supports a modular programming model

- Programmers don’t need to manage association between shared data and synchronization metadata (e.g., locks)
  - TM implementation takes care of that
- Functions containing only txnal sync compose w/o deadlock, nesting order does not matter
- Example:

```
void move(list& l1, list& l2, element e)
{ if (l1.remove(e)) l2.insert(e); }
```

  - TM: `__transaction_atomic { move(A, B, 23); }`
  - Locks: ?
- Early university user studies [3,4] suggest that txns lead to simpler programs with fewer errors compared to locking

GCC status

- Runs common TM benchmarks correctly
- Compiler: most of TM spec implemented
  - Publication safety not guaranteed
    - Need to restrict reordering across conditionals
  - Uninstrumented code path not yet created
  - TM type annotations not checked when casting
- libitm: general-purpose TM algorithms
  - Common ABI with ICC
    - Some divergence since most recent version of the spec
  - Performance tuning (problem: lack of real world usage)
  - Use HTMs that become available
- We need your workloads and use cases!

References

- [1] http://www.cl.cam.ac.uk/~pes20/cpp
- [2] https://sites.google.com/site/tmforcplusplus/
- [3] Pankratius & Adl-Tabatabai, “A study of transactional memory vs. locks in practice”, in SPAA 2011
- [4] Rossbach et al., “Is Transactional Programming Actually Easier?”, in PPoPP 2010