New Programming Abstractions for Concurrency in GCC 4.7

Torvald Riegel
Red Hat
12/04/05

Concurrency and atomicity

C++11 atomic types	**Transactional Memory**
Single memory location	Any number of memory locations
Low-level abstraction, exposes HW primitives	High-level abstraction, mixed SW/HW runtime support

Provide atomicity for concurrent accesses by different threads

Both based on C++11 memory model

- Complemented by C++11 threading 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);`
  - Important orders: acquire, acq_rel, release, relaxed, seq_cst

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 [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 <u>some</u> valid HB:
  - Must be consistent with all other relations and rules
  - Must be acyclic
  - 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 preserves DRF
  - Access granularity
  - Speculative stores, reordering, hoisting, ...

Examples

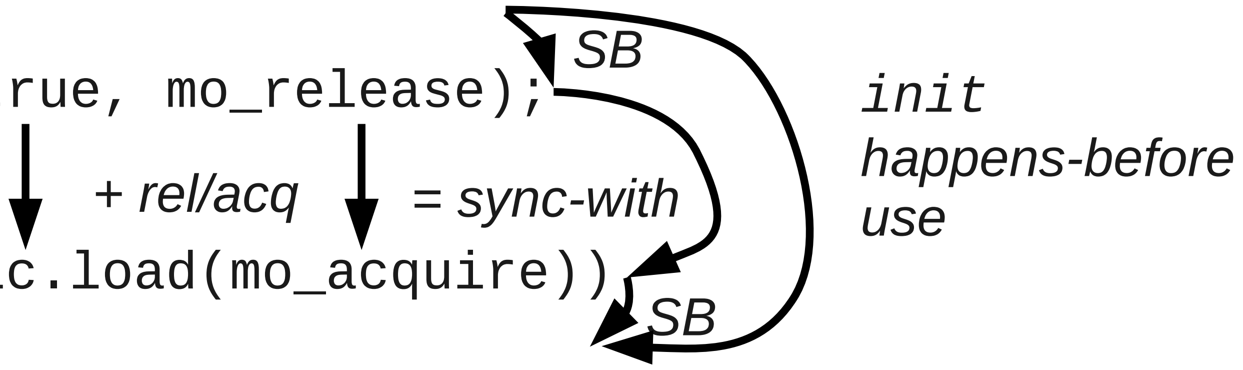
- Simple statistics counter:

```
counter.fetch_add(1, memory_order_relaxed);
counter.store(counter.load(mo_relaxed) + 1, mo_relaxed);
```

- Publication:

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

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



- Beware of data races:

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

Races with init
Program behavior is undefined

Transactional Memory (TM): What is it?

- Always faster than custom algorithm X? ...?
- A HW feature?
- Concurrent algorithm X?
- Optimistic synchronization in SW?
- Much too slow anyway? ...?

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

Transactional Memory (TM): What is it?

- ~~Always faster than custom algorithm X? ...?~~
- A HW feature?
- Concurrent algorithm X?
- Optimistic synchronization in SW?
- ~~Much too slow anyway? ...?~~

Implementation possibilities

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

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
- Language integration increases ease of use
  - Let the compiler help!
  - Allows reuse of existing (sequential) code
- Draft specification for C++ [2]
  - HP, IBM, Intel, Oracle, Red Hat
  - Spec group proposed standardization as C++ tech report [3]
  - C will be similar

TM supports a modular programming model

- Programmers don’t need to manage association between shared data and synchronization metadata (e.g., locks)
  - TM takes care of that
- Functions containing only txnal sync compose w/o deadlock, nesting order does not matter
- User studies suggest that txns lead to simpler programs with fewer errors compared to locking [4,5]
- Example:

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

  - TM: `__transaction_atomic { move(A, B, 23); }`
  - Locks: ?

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`

- Atomic / relaxed checked at compile time
  - Compiler analyzes code
  - Additional function attribs to deal with multiple Cus
- Work-in-progress: `tm_waiver`
  - Programmer-controlled synchronization for parts of a txn

How to synchronize with transactions?

- TM extends the C++11 memory model
  - All transactions totally ordered
  - Order contributes to Happens-Before (HB)
  - TM ensures <u>some</u> 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); }
```

- No changes to memory model of nontxnal code
  - If you don’t use it, you don’t pay for it

Implementation options

- Most of the TM implementation is in a library (GCC’s libitm)
- Software only (STM):
  - Global lock, two-phase locking, nonblocking, locked writes and efficiently validated reads (array of locks), ...
- Hardware TM (HTM):
  - x86: Intel’s TSX / RTM, AMD’s Advanced Synch. Facility
  - Hybrid HW/SW TM (HyTM)
- With compiler support (e.g., points-to analysis):
  - Automatic partitioning (divide-and-conquer), finding locking schemes at compile time, ...
- Language-level txns are a portable interface for HTM/STM
  - Compiler creates HTM + STM fallback code from one source
  - HTM support can be delivered by a library update

Current GCC 4.7 status and outlook

	Atomics, memory model	TM
Status:	• C++11 atomics implemented • Works fine for libitm	• Supports most of the specification • Runs standard TM benchmarks correctly
Outlook:	• C11 atomics • Fix memory access granularity issues (e.g., bitfields) • Audit GCC passes • More testing	• Optimize libitm (need your workloads and use cases!) • Generate better txnal code • HTM support

References

- [1] http://www.cl.cam.ac.uk/~pes20/cpp
- [2] https://sites.google.com/site/tmforcplusplus/
- [3] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3341.pdf
- [4] Pankratius & Adl-Tabatabai, “A study of transactional memory vs. locks in practice”, in SPAA 2011
- [5] Rossbach et al., “Is Transactional Programming Actually Easier?”, in PPoPP 2010