Transactional Memory in C++

Hans-J. Boehm

Google

and

ISO C++ Concurrency Study Group chair
ISO C++ Transactional Memory Study Group participant
Disclaimers

- I’ve been writing concurrent programs for decades, but
- I’m really at best a TM theoretician.
- Some of this is just my opinion ...
Background: C++ TM Standardization

Intel/Sun/Oracle/IBM TM specification effort more or less became ISO JTC1/SC22/WG21/SG5. Other recent participants include: Michael Wong (chair), Justin Gottschlich, Victor Luchangco, Jens Maurer, Maged Michael, Torvald Riegel, Michael Scott, Tatiana Shpeisman, Michael Spear ...

Technical Specification is about to be published.

Supports:

relaxed transactions ⇒ synchronized { ... }  
atomic transactions ⇒ atomic_noexcept { ... }  
                              atomic_commit { ... }  
                              atomic_cancel { ... }
Transaction-unsafe operations in block:

- **synchronized** blocks fall back to locking.
- **atomic_...** blocks are almost entirely statically checked to preclude that.
  - `transaction_safe_dynamic` is allowed in virtual function declarations

`transaction_safe` is part of the type system.

Atomic blocks differ only in exception handling.

No support for explicit abort except `atomic_cancel { ... }

- and that requires closed nesting support.
C++ TM is designed as

Simpler general purpose synchronization mechanism. Possibly a failure atomicity mechanism.

Not:
A way to get direct HTM access.

Possibly not:
The best way to build high performance concurrent data structures.
Transactions in C++ Memory Model

- Transactions follow C++ data-race-free model.
  - Data races ⇒ undefined behavior.
- Explicitly aborted (cancelled) transactions can participate in data races.
- atomic/synchronization operations disallowed in atomic blocks.
  - except for function-local statics, malloc/free.
- Transactions essentially behave like lock acquisitions.
Late foundational change

- Single Global Lock Atomicity $\Rightarrow$ Disjoint Lock Atomicity (Menon et al. 08)
  - Driving consideration:
    - Should be able to optimize transactions operating only on local data

```java
synchronized {
    t = new thread ([] { 
        atomic_noexceet { }; global = x;});
    x = 42;
};
t -> join(); // global = 42
```
Memory Model Implications

- **atomic/synchronized** blocks help prevent data races.
- Data-race-freedom $\Rightarrow$ synchronization-free regions are atomic.
- No synchronization inside atomic blocks $\Rightarrow$ atomic blocks are atomic.
- strong/weak isolation are indistinguishable.
- Publication safety is implied.
- Privatization safety is implied.
- No explicit opacity condition.
Future in the C++ committee

● A Technical Specification is not part of the standard.
● It may eventually be proposed as a standard addition, e.g. for C++17.
● I don’t think this is currently a slam dunk.
  ○ … in spite of influential supporters.
  ○ Would be a major imposition on implementors.
  ○ Need applications!
● … in spite of influential supporters.
Why transactional memory? My personal view:

- Generic (templatized) code is useful and at the heart of modern C++.
- Locks require ordering to avoid deadlocks.
- Lock ordering isn’t feasible with generic programming (or pervasive callbacks).

```cpp
template <class T>
T my_swap(T& x, T a) {
    lock_guard _(m);
    T result = x;
    x = a;      // Which locks does this acquire?
    return result;
}
```
Transactional memory provides modular / usable synchronization

- Locks don’t.
- Somewhat useful even with low performance.
  - Implementations often start with global lock.
  - Even STM can beat that!
- Unlikely to be the only synchronization mechanism.
  - Verdict still out on condition variable replacement?
  - Locks at
    - outermost / coarse level
    - & at system/leaf level?
What I think we got right

- Lock-like semantics
- *Synchronized* blocks
  - Tolerate transaction-unsafe actions on exceptional paths.
  - Allow implementations of transactions that the compiler can’t prove atomic
    - Transaction-unsafe on exceptional paths.
    - Hidden communication, e.g. helper threads for the hard cases.
    - Logically, but not bit-wise atomic actions.
What I think is questionable

- **atomic_blocks**
  - But they’re growing on me …
  - … and others think *synchronized* is questionable.

- **atomic_cancel**
  - Allowable exceptions are severely restricted.
  - Exceptions tend to arise mostly from transaction-unsafe operations.
What I think we should look at for version 2

- Commit actions
- Some kind of transactional escape
  - Make transaction-safe malloc user implementable?
- Abort actions?
- Relax restrictions on synchronization use in atomic transactions.
  - I’m not optimistic.
  - Easy to implement C++11 atomics as transactions, but
    ■ Slows down existing code.
  - Could defer unlocks, but
    ■ Adds deadlocks to existing libraries.
Questions ?

Discussion ?