Concurrency and Transactional Memory in C++:
50000 foot view

Hans-J. Boehm
Google
Concurrency in the C++ Standard

Most additions start in “Concurrency Study Group” (ISO JTC1/SC22/WG21/SG1).

- Transactional memory is separate (SG5).
- Proposals are also reviewed by other groups.
- Specifications are intended to represent community consensus.

SG1 (and SG5) tend to be relatively inventive.

C++ standard describes language semantics, not implementation rules or allowable optimizations.

But they are not:

- Formal mathematical specifications.
- Textbooks
Concurrency Changes in C++11

- **Threads API**
  - Benefits from lambda-expressions, etc.

- **Memory model/shared variable semantics**
  - Formalized by Mark Batty, Peter Sewell et al
  - Starting to impact hardware ISAs.
  - Undefined behavior for data races.
  - Sequential consistency by default.
  - `trylock()`, `wait()` may spuriously fail/return.

- **Atomic operations library**
  - Provides explicit weak ordering as an option:
    - `memory_order_acquire`, `memory_order_release`, `memory_order_relaxed`, `memory_order_consume`
Concurrency Changes in C++14

Relatively minor cleanups.

- `shared_timed_mutex`
- Add some hand-waving for known issues.
Conspicuous holes in C++11/C++14

Memory model mostly solid, but:

- `memory_order_relaxed` spec is wrong in C++11.
  - Serious hand-waving in C++14.
- We don’t know how to fix that without adding overhead.
- `memory_order_consume` design needs work.

`async()` beginner thread creation facility has serious design flaw: Working on replacement.

No concurrent data structures.

Incomplete synchronization library.
Moving forward: near term

“Technical Specification”:
- Optional addition to the standard.
- Candidate for future inclusion in standard.

Two technical specifications in the works:
- Parallel/vector algorithms (STL + a bit)
- Miscellaneous concurrency extensions
  - future.then, etc.
  - latches and barriers
  - atomic “smart pointers”
Moving forward: Slightly longer term

- Replace `async()` with executors.
- Fork-join task-based parallelism. ("Task regions")
- Asynchronous computation without explicit continuations. ("resumable functions")
- Low level waiting API: `synchronous<T>`.
- More general vector parallelism support.
- Various concurrent data structures.
Further out

- Fix `memory_order_relaxed`.
- Fix `memory_order_consume`.
- Mix atomic and non-atomic operations on same location.
- Better specification of execution agents (beyond bare OS threads) and progress properties.
Transactional Memory

- Separate study group. (SG5)
- I am one of many participants. Others in attendance:
  - Torvald Riegel
  - Michael Scott
  - Maged Michael
- Michael Wong (IBM) and Justin Gottschlich (Intel) are the main organizers.
- Jens Maurer has done much of the recent writing.
- Technical specification currently out for initial ballot and comments. (“Preliminary Draft Technical Specification”)
- Viewed as experimental.
  - When we can’t decide, include both options.
Why transactional memory? (many views, here’s mine)

- Locks require lock ordering to prevent deadlocks.
- Lock ordering is essentially intractable with callbacks, i.e. functions passed as parameters.
- In generic (templatized) programs, essentially every operator represents a call of a function parameter.
  - What locks does `x = y;` acquire?
  - If `x` might be a reference counted “smart pointer”?
- ⇒ Modern C++ programming is (nearly?) incompatible with locks.
Not a full replacement for mutexes

- Condition variables do not play with transactions.
- Address the 95% of the cases non-experts are more likely to write.
Proposal


Four transaction-like constructs:

  synchronized { ... }
  atomic_noexcept { ... }
  atomic_cancel { ... }
  atomic_commit { ... }

In the absence of nested non-transactional synchronization and exceptions, they all have the same semantics.
Shared semantics

No exceptions, nested synchronization:

All constructs behave as though the same single global lock were acquired before the compound statement and released at the end.

A reasonable quality implementation is expected to scale better than that …

Data-race-freedom $\Rightarrow$ strong atomicity
Semantic differences (1)

- `synchronized {}` supports nested non-transactional synchronization. e.g.
  
  ```
  synchronized {
    parallel_sort(a.begin(), a.end());
  }
  ```
  
  or more likely
  
  ```
  synchronized {
    ...
    if (unlikely_event) { cerr << "disaster"; }  
  }
  ```

- `atomic_x {}` does not. `atomic_x {}` is atomic.

- It is a compile-time error to invoke “unsafe” potentially synchronizing constructs from within `atomic_x {}`. 
Semantic differences (2)

- `atomic_commit {}` commits the transaction if an exception is thrown out of the body.
- `atomic_cancel {}` aborts the transaction in that case.
- `atomic_noexcept {}` disallows exceptions.

`atomic_cancel { ...; throw ... ; ... }` is currently the only way to explicitly abort a transaction.

Explicitly aborted transactions can participate in data races.
atomic_cancel {}

- Intuitively the most natural.
- Surprisingly rarely useful?!
  - Only a very restricted set of exception types is supported.
  - Many C++ objects (e.g. shared_ptr) cannot be safely copied out of a rolled-back transaction.
  - Exception handling seems most important for transaction-unsafe (I/O) operations.
- Difficult to implement: Requires full closed nesting.
  - Cannot roll back entire transaction if exception is caught in outer transaction.
  - Usually requires software fallback for HTM.
- Transactions are primarily a synchronization mechanism.
- Unclear whether they will be used for failure atomicity.
**synchronized {}** vs. **atomic_commit {}**

- If the body is compatible with both, there is currently no semantic difference.
  - In a data-race-free language, synchronization-free regions are atomic.
  - **atomic_commit {}** is a pure subset.
  - Allows compiler to diagnose atomicity violations.
  - Recurring discussion of C++11 atomics inside **atomic_commit** with different semantics.
  - Inclusion of both was controversial.

- But there seems to be increasing sentiment for both:
  - Statically guaranteed atomicity appears useful,
    - even if it relies on data-race-freedom.
  - **synchronized {}** is often easier to use.
  - Michael Spear’s empirical evidence seems consistent with that.
Transaction-safety

- `atomic_x {}` blocks may only contain transaction-safe statements.
- Functions may be declared `transaction_safe`, making them safe to call from atomic blocks.
- Function pointers and virtual functions may also be declared `transaction_safe`.
- Many standard library functions are declared `transaction_safe`.
- Transaction-safety is part of the type system.
Remaining concern

- C++11 mutexes and single-variable atomics allow synchronization removal for single-threaded use.
- Transactions do not have corresponding property.
  ○ `int x; atomic_noexcept { ++x; }` vs.
  ○ `atomic<int> x; ++x;`
  ○ Empty transactions are not no-ops.
- Should transactions logically lock individual objects rather than single-global lock?
- *Likely to be revisited* ...
Other interesting corner cases

```c
atomic_noexcept {
    static int x(foo());
    ...
}
```

has nested synchronization, but is allowed.

Memory allocation is another synchronization construct allowed in `atomic {}` blocks.

We need to support occasional dynamic checking of virtual function safety.
Future issues

Low level escape for non-transactional code in transaction?

C++11 atomics in atomic blocks, with semantics that preserve atomicity?
Semantically easy, but:
- Seems to impact C++11 performance.
- Surprising behavioral difference?
Questions?