Draft Specification of Txnal Language Constructs for C++

Torvald Riegel
Red Hat
11/09/23

Draft specification

- Scope:
  - C++, shared memory
  - C will be similar
- Who:
  - HP, IBM, Intel, Oracle, Red Hat
  - External contributors, comments, ...
- Implementations:
  - GCC: `transactional-memory` branch
  - ICC: What-if prototype

TM must be practical

- Aim for this spec is inclusion in C++ standard (extending C++11’s successors)
- Rather systems research than pure TM theory
- Constraints / requirements:
  - Integrate well with current C++ ecosystems (language, libraries, compilers, runtime environments)
  - Easy to understand and use
  - Efficient implementations must be possible
  - Must not slow down nontxnal code

High-level feature overview

- Central language construct: **Transaction statements**
  - `__transaction <statement>`
  - Embedded into C++ memory model
- Atomic and relaxed txns
- Several C++ thread-related features compatible with atomic txns, but not all
- Commit-on-throw is default

Embedding TM into C++11

- Language tie-in is necessary
  - Abstract machine spec and as-if rules guide implementations (TM runtime and compiler)
  - Existing memory model for txnal and nontxnal code
- C++11 memory model:
  - Sequenced-before * synchronizes-with
    = happens-before (HB)
- Reliance on data-race freedom (DRF):
  - Conflicting accesses to same location that are not ordered by HB -> data race
  - Data races have catch-fire semantics

Transactions: The basics

- Want transactions to virtually execute sequentially in some total order (like with a global lock)
- **Txnal Synchronization Order (TSO)**
  - Virtual StartTxn, EndTxn ops for each outermost txn
    - Demarcate txns in sequenced-before
  - StartTxn, EndTxn ordered according to TSO
    - Txns don’t overlap
  - EndTxn/StartTxn pairs contribute to synchronizes-with
  - Program cannot enforce a specific TSO directly, can only constrain choice
  - Txns thus affect and are affected by happens-before

Transactions: The basics (2)

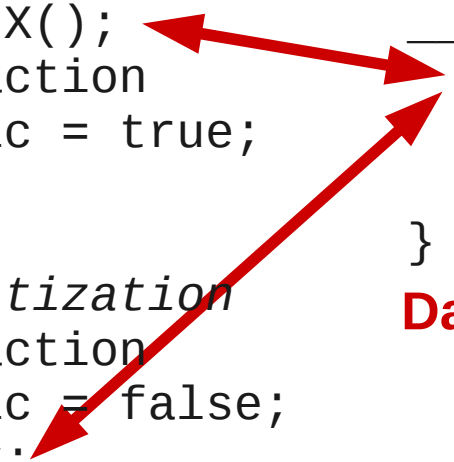
- Assuming just ordinary, nonsynchronizing code in txns
  - Can only observe TSO via txns
    - Anything else would be a data race!
  - Txns are atomic and sequentially consistent
- TM runtime
  - Can rely on DRF
  - Can determine TSO independently
  - Only has to consider existing nontxnal happens-before

Publication / privatization as example

- Responsibilities:
  - Publication: Programmer and compiler
  - Privatization: Programmer, compiler, TM runtime

```
// Publication
x = new X();
__transaction
  xpublic = true;


// Privatization
__transaction
  xpublic = false;
delete x;
```



```
// Incorrect
__transaction{
  y = x;
  if (xpublic)
    foo(y);
}
```

Data race!

```
// Correct
__transaction{
  if (xpublic)
    foo(x);
}
```

Correct dependency:
x only read if `xpublic` is true

Relaxing atomicity

- Basic txns on previous slides are **atomic txns**
  - But what if they contain sync code? No races anymore..
  - But what if the compiler can’t instrument the code?

- **Relaxed txns**:
  - Atomicity wrt. other txns but not wrt. nontxnal code
  - Can execute unsafe code
  - More permissive but relaxed atomicity

- Open question:
  - Middle ground?
    Can we mix atomic/relaxed in a single txn in a safe way?

Syntax

- Keywords for txn statements/expressions/functions
  - Current spec with atomic as default:
    - `__transaction`, `__transaction [[atomic]]`, `__transaction [[relaxed]]`
  - Alternative with no default:
    - `__transaction_atomic`, `__transaction_relaxed`

- Attributes for function types
  - `transaction_safe`: No volatile accesses, asm,...
  - `transaction_unsafe`: Prevent implicit transaction_safe
  - `transaction_callable`: Called from relaxed txn

C++ feature compatibility with atomic txns

- C++11 atomics: Maybe. IMO: No
- Locks: Can be allowed.
  - Order of acquisition matters (as in publication example)
  - Locks are held until the end of the transaction
- Block-scope static vars (ctors): Allowed
  - Initialization will appear atomically to other threads
- call_once(): Same.
- Condition variables: No?
- Futures/promises: ?

Atomic txns: Static vs. dynamic checking

- Current spec: Purely static checking
  - Atomic txns must contain only safe code (No sync, etc.)
  - Must only contain calls to `transaction_safe` functions
  - Conservative, complete atomicity check at compile time
- Fully dynamic checking (runtime)?
  - Can contain all code, call all functions
  - Fatal runtime error upon execution of unsafe code
  - Easier code reuse
- Semi-dynamic checking?
  - Mark unsafe code w/ `__not_executed_in_atomic_txns{}`
  - Makes programmers aware, but also requires code changes

Exceptions

- Default behavior:
  - Commit on throw
  - Don’t change exception semantics

- Basic support for failure atomicity
  - `__transaction_cancel` statement
    - Rolls back the enclosing atomic txn
    - With `[[outer]]`, rolls back outermost txn
  - `__transaction_cancel throw <`*`expr`*`>`
    - Cancel and throw exception of integral/enumerated type
    - Extend to std exception types with constraints on derived classes?

Please contribute!

- Making the spec robust for the C++ standard
  - Specification: Use, formalize, verify
    - We plan to present this draft to the C++ committee in February
  - Implementations: Use, test, improve
    - GCC 4.7 feature freeze at end of October

http://groups.google.com/group/tm-languages

http://gcc.gnu.org/lists.html

triegel@redhat.com