Document number: N4338
Authors: Michael Wong, Jens Maurer
Reply-to: michaelw@ca.ibm.com
Date: 2014-11-21
Revision: revision of parts of N4265, N3999, N4179, N4180

Editor's Report: Technical Specification for C++ Extensions for Transactional Memory, post-Urbana

Acknowledgment

We wish to thank comments from the Review Committee of Jens Maurer, Mike Miller, and Alisdair Meredith as well as the following:

Hans Boehm
Justin Gottschlich
Victor Luchangcom
Jens Maurer
Paul McKenney
Maged Michael
Mark Moir
Torvald Riegel
Michael Scott
Tatiana Shpeisman
Michael Spear
Michael Wong

History

Post-Urbana (2014-11-21)

N4301 is the latest TM TS Working Draft. It contains changes to the TM TS as directed by the committee at the Urbana meeting combining N4272 and N4265.

N4302 is document N4301 reformatted as a PDTS ballot document.

N4338 is this document - TM TS Editor's report, post Urbana.

Urbana (2014-11-07)

Motion 3 (approved unanimously)

Move to create a working paper for the Technical Specification on C++ Extensions for Transactional Memory with N4272, "Technical Specification on C++ Extensions for Transactional Memory, Working Draft Header" and N4265, "Transactional Memory Support for C++: Wording (revision 3)" as its initial content.

N4272 has the Working Draft front matter.

N4265 presents the wording proposal including changes from Urbana (both core and library) for integrating transactional memory support into C++. For motivation and introductory overview, see the predecessor paper N3999 "Standard Wording for Transactional Memory Support for C++".

Pre-Urbana (2014-10-10)

The companion paper N4180 motivates and explains the additional features that have been integrated since the Rapperswil meeting. Those were approved by EWG in Urbana.

N4179 presents the pre-Urbana integrated TM wording for core and library.

Pre-Rapperswil (2014-05-23)

N3999 presents a concise set of introduction, motivation, syntax, semantics and initial wording.

N4000 documents our effort to transactionalize a C++ Standard Template Library (STL) container to demonstrate the feasibility of the transactional language constructs.

Changes

- Addressed comments from 2014-06-02 review teleconference, clarifying the wording.
- Address CWG review comments from Rapperswil:
    - allow "synchronizes with" for non-library constructs
    - volatile subobjects are tx-unsafe
    - remove cross-translation unit safe-by-default
    - simplified lambda transaction-safety
    - redeclarations can omit `transaction_safe`
    - comparing pointers is unspecified unless tx-safety is the same
- Address LWG review comments from Rapperswil:
    - added more cross-references in the library section

    - prohibit additional tx-safe annotations by library implementations using the same words as for constexpr
    - for exception types that support tx cancellation, add notes about required implementation support
    - remove rand/srand, because some implementation use global state where accesses are synchronized using a mutex
- add implicit conversion from "pointer to transaction_safe member function" to "pointer to member function" (see 4.14)
- clarify definition of transaction-safe (defined term)
- added [[optimize_for_synchronized]] attribute (see section 7.6.6 [dcl.attr.sync])
- added `transaction_safe noinherit` for virtual functions (non-viral)
- added transaction-safety for all containers and iterator-related functions
- added `tx_exception`
- Addressed comments from 2014-09-15 CWG review teleconference
    - matching a *handler* performs transaction-safety conversions
    - allow, but do not require `transaction_safe` on lambdas
    - a `transaction_safe noinherit` function overriding a `transaction_safe` function is ill-formed
    - template argument deduction performs transaction-safety conversions
    - allow forming a composite pointer type using a transaction-safety conversion
- rename `maybe transaction_safe` to `transaction_safe noinherit`
- Addressed comments from 2014-11-04 LWG review<
    - for the containers, make one blanket statement covering all required functions and operations instead of detailed per-function lists
    - reflect that not only user-provided functions, but also built-in operations can make an instantiation of standard library function template unsafe (e.g. std::copy invoked with "volatile int *" parameters)
    - consider additional overloads in <cxxx> headers (compared to the corresponding xxx.h header) for tx-safety
    - C library functions should not be "declared" transaction-safe to leave more room for implementers to have special compiler magic, not requiring actual modifications of C headers
    - restrict `tx_exception` to trivially copyable types
    - iterators of containers and rebound allocators are required to be transaction-safe
    - also cover numerics algorithms (section 26.7)
- Renamed "transaction_safe noinherit" to "transaction_safe_noinherit" per 2014-11-05 EWG review
- Addressed comments from 2014-11-05 LWG review
- Addressed comments from 2014-11-05 CWG review
    - do not inherit transaction_safe for redeclarations; allow that explicit specializations differ

Resolved issues

- Under which circumstances is a lambda function (implicitly) declared transaction-safe? [It is declared tx-safe if its definition is (directly) safe and all invoked functions are tx-safe.] Do we want to allow an explicit transaction_safe/unsafe annotation? [Yes to the former.]
- Is the term "transaction-statement" still ok, or should that be renamed to "atomic-statement", in line with the (new) spelling of the keywords? *Telco 2014-03-31: use "atomic block" and "transaction-safe".*
- Allow conversion of "pointer to transaction-safe member function" to "pointer to member function"
- add "transaction_safe noinherit" for virtual functions
- `std::exception` is the base of the exception hierarchy. Should we declare its virtual what() function `transaction_safe`? [no; instead review definition of derived classes such as length_error] This has serious ripple effects to user code, in particular if that user code is totally unaware of transactions. *Telco 2014-08-11 and 2014-09-08: introduce "maybe transaction_safe" for virtual functions, which is not viral, but accepts undefined behavior.*
- For "composite pointer type" in 5 [expr], address that T might be "pointer to function" (not transaction-safe) vs. "pointer to transaction-safe function". This can be unified to "pointer to function" (not transaction-safe).
- Introduce a

  ```
  template<class T>
  class tx_exception : exception { ... };
  ```

  with a transaction-safe "what()" function and where "T" can be memcpy'd.
- add "template specialization can be tx-safe even when corresponding member of the template is not"