Document No: WG21 N4571
Date: 2015-05-08
References: ISO/IEC PDTS 19841, SC22/WG21 N4488
Reply To: Barry Hedquist <beh@peren.com>
INCITS/PL22.16 International Representative

# Record of Response: National Body Comments

# ISO/IEC PDTS 19841

# Technical Specification: C++ Extensions for Transactional Memory

Attached is WG21 N4571, Record of Response to National Body Comments for ISO/IEC PDTS 19841, Technical Specification – C++ Extensions for Transactional Memory. Also attached is WG21 N4488, containing further details for the responses.

Document numbers referenced in this Record of Response are WG21 documents unless otherwise stated.

# Record of Response: NB Comments, PDTS 19841, C++ Extensions for Transactional Memory. See WG21 N4488.

| Date:2015-05-06 | Document: SC22/N 5019 WG21/N4517 | Project: PDTS 19841 |
|---|---|---|

| MB/ NC[1] | Line number | Clause/ Subclause | Paragraph/ Figure/Table | Type of comment[2] | Comments | Proposed change | Observations of the secretariat |
|---|---|---|---|---|---|---|---|
| JP 1 | | | | ge | We are concerned that there could be a performance degradation even in an environment lacking of transactional memory feature by adopting this technical specification, e.g. making some standard libraries like <math.h> transactional safe.<br>It is the reason for our disapproval. If we can reasonably confirm that there's no degradation, we will change our position to approval. | Please make us sure there's no degradation. | REJECT<br>See WG21 N4488 (Attch) |
| US 1 | | | | te | Memory ordering requirements of transactions are problematically strict. Even empty or purely local transactions have observable synchronization effects and can usually not be removed by an optimizing compiler. This introduces a performance penalty when transactional library code is reused in a clearly thread-local context. | Consider weakening ordering requirements to allow such optimizations. | ACCEPT<br>See WG21 N4488 (Attch) |
| CA 1 | N/A | N/A | N/A | ge | Request to add a Feature Test Macro __cpp_transactional_memory based on http://isocpp.org/std/standing-documents/sd-6-sg10-feature-test-recommendations | The value of the macro will be the year and month of the release of the TS. It does not need any experimental or TS tag. | ACCEPT<br>See WG21 N4488 (Attch) |
| CA 3 | N/A | 4.3 [conv.func] | Para 1 | ge | Make helper functions in 20.2 transaction-safe. Here is an example where std::move is not transaction-safe<br><br>template <class T><br>  void safe_swap(T &a, T &b) transaction_safe<br>  {<br>    atomic_commit<br>    {<br>      using std::move;<br>      T temp = move(a); // Note that std::move is not transaction-safe according to draft, but it should be<br>      a = move(b);<br>      b = move(temp); | Add std::move and other utilities in 20.2 to be transaction_safe. | ACCEPT<br>See WG21 N4488 (Attch) |

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

| MB/ NC[1] | Line number | Clause/ Subclause | Paragraph/ Figure/Table | Type of comment[2] | Comments | Proposed change | Observations of the secretariat |
|---|---|---|---|---|---|---|---|
| | | | | |     }<br>  }<br><br>template &lt;class T&gt;<br>  void apply(T &a, T &b, void f(T&,&T))<br>  {<br>    f(a,b); // Ok<br>    assert(f == safe_swap&lt;int&gt;); // result unspecified according to 5.10, paragraph 2, right?<br>  }<br><br>int main()<br>{<br>  int x = 2, y = 3;<br>  apply(x, y, safe_swap&lt;int&gt;); // Ok even though transaction_safe is lost<br>} | | |
| CA 2 | N/A | 5.2.2 [expr.call] | Para 1 | te | This addition states:<br>A call to a virtual function that is evaluated within a synchronized (6.9 [stmt.sync]) or atomic block (6.10 [stmt.tx]) results in undefined behavior if the virtual function is declared transaction_safe_noinherit and the final overrider is not declared transaction_safe.<br>It is Undefined Behavior if you call into a virtual function declared as tx_safe_noinherit but it is not tx_safe in the final overrider. This ensures that the dynamic call is safe, no matter what the dynamic object is since tx_safe_noinherit gives no such guarantee.<br><br>Our concern is this is excessive for a | Please fix for synchronized block so that it is not part of this requirement. Suggested wording:<br>A call to a virtual function that is evaluated within ~~a synchronized (6.9 [stmt.sync]) or~~ an atomic block (6.10 [stmt.tx]) results in undefined behavior if the virtual function is declared transaction_safe_noinherit and the final overrider is not declared transaction_safe. | ACCEPT<br>See WG21 N4488 (Attch) |

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

**Record of Response: NB Comments, PDTS 19841, C++ Extensions for Transactional Memory. See WG21 N4488.**

| Date:2015-05-06 | Document: SC22/N 5019 WG21/N4517 | Project: PDTS 19841 |
|---|---|---|

| MB/ NC[1] | Line number | Clause/ Subclause | Paragraph/ Figure/Table | Type of comment[2] | Comments | Proposed change | Observations of the secretariat |
|---|---|---|---|---|---|---|---|
| | | | | | synchronized block because these can call tx_unsafe functions. | | |
| FI 1 | | 8 | 4 | te | It seems that the applicability of transaction_safe_noinherit is likely going to be wider in the future than just in virtual functions. If that wider applicability appears, new keywords need to be added. Generalizing the name transaction_safe_noinherit would possibly avoid that problem. | Rename transaction_safe_noinherit to transaction_safe_dynamic. Transaction safety of calls to such functions is ultimately a runtime property, hence _dynamic seems like a suitable suffix. | ACCEPT<br>See WG21 N4488 (Attch) |
| JP 2 | | 8.4.4 | 1 | te | A function-local static variable initialization should be transactional-unsafe. The initialization in an atomic execution needs to be synchronized with non-atomic executions. | Add "a function-local static variable initialization" in the list of conditions for a transactional-unsafe statement . | REJECT<br>See WG21 N4488 (Attch) |
| CA 4 | n/a | 8.4.4 [dcl.fct.def.tx] | After Para 1, bullet 5 | ge | In the first sequence of dash bullets (-- ) indicating transaction-unsafe expressions, the fifth one states «an implicit call of a non-virtual function that is not transaction_safe». I wonder why the «implicit» call is being explicitly (sorry for the pun! :) ) specified, as it seems to me that an explicit call to a non-virtual function would yield the same consequences. Unless I'm missing out on something, an implicit call could be something like:<br><br>struct B<br>{<br>  int f(); // not transaction_safe, not virtual<br>  virtual ~B() = default;<br>};<br><br>struct D : B<br>{ | This seems a possible confusion for other user, please clarify. | REJECT<br>See WG21 N4488 (Attch) |

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

| MB/ NC[1] | Line number | Clause/ Subclause | Paragraph/ Figure/Table | Type of comment[2] | Comments | Proposed change | Observations of the secretariat |
|---|---|---|---|---|---|---|---|
| | | | | | `int g()`<br>`{`<br>`return f() + // implicit call?`<br>`this->f() + // explicit call?`<br>`B::f(); // explicit call?`<br>`}`<br>`};` | | |

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

ISO/IEC JTC 1/SC 22/WG 21 N4488
revises N4410
Jens Maurer
2015-05-07

# N4488: Responses to PDTS comments on Transactional Memory, version 2

Jens Maurer, jens.maurer@gmx.net

with other members of the transactional memory study group (SG5), including (in alphabetical order):
Hans Boehm, hboehm@google.com
Victor Luchangco, victor.luchangco@oracle.com
Paul McKenney, paulmck@linux.vnet.ibm.com
Maged Michael, maged.michael@gmail.com
Mark Moir, mark.moir@oracle.com
Torvald Riegel, triegel@redhat.com
Michael Scott, scott@cs.rochester.edu
Tatiana Shpeisman, tatiana.shpeisman@intel.com
Michael Spear, spear@cse.lehigh.edu
Michael Wong, michaelw@ca.ibm.com (chair of SG5)

## Introduction

This paper presents the proposed responses to N4396 "National Body Comments, ISO/IEC PDTS 19841, C++ Extensions for Transactional Memory".

## Changes compared to N4410

- fix a typo in the spelling of the feature test macro (two underscores in the middle)
- remove "L" suffix in value of feature test macro
- declval does not need to be transaction-safe, since it is never odr-used
- rework section CA 1

## JP 1: Performance degradation

REJECT

The specification of transaction safety ensures that it is possible to compile code so that whether a given function call is executed in transaction context or outside of transaction context can be determined at compile time. Therefore, existing code that is executed outside of transactions and that does not use any of the transactional memory constructs will execute as before. The performance of code that actually uses transactions will depend on the available hardware support, similar to the fact that the performance of mutexes vs. accesses to atomic variables depends on a number of hardware and other factors.

A conservative approach was chosen for mandating the transaction-safety of standard library functions. Functions that conceivably access global state are not touched. In particular, the functions in the header <math.h> were intentionally not made transaction-safe in this Technical Specification, because the interaction of transactional memory with accesses to a potentially global rounding mode setting was deemed to require further study. As an exception, based on early user feedback, memory allocation is mandated to be transaction-safe, although it might access the global free store. Implementation experience shows that this does not negatively impact the performance of non-transactional executions.

## US 1: Relax synchronization to allow optimizations on local transactions

ACCEPT

In section 1.10 intro.multithread, change the added paragraph 9 as follows:

> There is a global total order of execution for all outer blocks. If, in that total order, T1 is ordered before T2,
>
> - **no evaluation in T2 happens before any evaluation in T1 and**
> - **if T1 and T2 perform conflicting expression evaluations,** then the end of T1 synchronizes with the start of T2.

## CA 1: Feature Test Macro

ACCEPT
Change in section 1.3 [general.references]:
... Beginning with section ~~1.4~~ **1.10** below, all clause and section numbers, titles, and symbolic references in [brackets] refer to the corresponding elements of the C++ Standard. Sections 1.1 through ~~1.3~~ **1.5** of this Technical Specification are introductory material and are unrelated to the similarly-numbered sections of the C++ Standard.
Change in section 1.4 [intro.compliance]:
Conformance requirements for this specification are the same as those defined in **section** 1.4 [intro.compliance] **of the C++ Standard**. [ Note: Conformance is defined in terms of the behavior of programs. -- end note ]
Add a new section 1.5 [intro.features]:

**1.5 Feature testing [intro.features]**

**An implementation that provides support for this Technical Specification shall define the feature test macro in Table 1.**

**Table 1 -- Feature Test Macro**

| Name | Value | Header |
|---|---|---|
| `__cpp_transactional_memory` | 201505 | *predeclared* |

## CA 3: Make helper functions transaction-safe

ACCEPT

Add the following to the appropriate sections:

> **Add in 20.2 [utility] after the synopsis:**
> A function in this section is transaction-safe if all required operations are transaction-safe.
> **In 20.2.4 [forward], add "`transaction_safe`" to the declaration of all functions.**

## CA 2: Virtual function calls in synchronized blocks

ACCEPT

Change the added text in 5.2.2 [expr.call] paragraph 1:

> A call to a virtual function that is evaluated within ~~a synchronized (6.9 [stmt.sync]) or~~ an atomic block (6.10 [stmt.tx]) results in undefined behavior if the virtual function is declared ~~`transaction_safe_noinherit`~~ `transaction_safe_dynamic` and the final overrider is not declared `transaction_safe`.

## FI 1: Rename `transaction_safe_noinherit` to `transaction_safe_dynamic`

ACCEPT

Change all mentions of `transaction_safe_noinherit` to `transaction_safe_dynamic`, including sections 2.11 [lex.name], 5.2.2 [expr.call] (see also CA 2, above), clause 8 [dcl.decl], 8.3.5 [dcl.fct], 10.3 [class.virtual], 18.6.2.1 [bad.alloc], 18.6.2.2 [new.badlength], 18.7.2 [bad.cast], 18.7.3 [bad.typeid], 18.8.1 [exception], 18.8.2 [bad.exception], and 19.2 [std.exceptions].

## JP 2: Initialization of function-local static variables

REJECT

We agree that initialization of function-local statics should be atomic with respect to both transactional and non-transactional uses. We do not believe that the specification as drafted, taking into consideration the requirements on transaction safety, necessitates any additional overhead on the non-transactional code path once the initialization is complete.

## CA 4: Redundant case for transaction-unsafe expressions

REJECT

The specification is carefully crafted to ensure that calls through function pointers or member function pointers fall into the sixth bullet. Omitting "implicit" in the fifth bullet would (arguably) defeat that purpose.