Disclaimer and Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

This document contains information on products in the design phase of development.


* Other names and brands may be claimed as the property of others.

Copyright (C) 2007-2008 Intel Corporation. All rights reserved.
## Revision History

<table>
<thead>
<tr>
<th>Document Number</th>
<th>Revision Number</th>
<th>Description</th>
<th>Revision Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>318253-001US</td>
<td>1.0</td>
<td>Intel® C++ STM Compiler, Prototype Edition</td>
<td>September 2007</td>
</tr>
<tr>
<td>318253-001US</td>
<td>2.0</td>
<td>Intel® C++ STM Compiler, Prototype Edition 2.0</td>
<td>March 31, 2008</td>
</tr>
</tbody>
</table>
# Contents

1  About this Document ................................................................................................................. 6
   1.1 Intended Audience .................................................................................................................. 6
   1.2 Document Organization ......................................................................................................... 6
   1.3 Conventions and Symbols ...................................................................................................... 7
   1.4 Related Information .............................................................................................................. 7
2  Introduction ................................................................................................................................ 7
   2.1 Requirements ....................................................................................................................... 7
   2.2 Not Covered by this Document ............................................................................................. 8
3  Language Extensions ................................................................................................................. 8
   3.1 Conditional Compilation ....................................................................................................... 8
   3.2 Specifying Atomic Code Blocks for TM ................................................................................ 8
   3.3 Specifying Waiver Code Blocks for TM ............................................................................... 10
   3.4 Specifying a Function as `tm_callable` ................................................................................ 11
   3.5 Specifying a Function as `tm_safe` ....................................................................................... 12
   3.6 Specifying a Function as `tm_unknown` ................................................................................ 13
   3.7 Specifying a Function as `tm_seh` ....................................................................................... 14
   3.8 Specifying a Function as `tm_pure` ..................................................................................... 15
   3.9 Specifying a Function as `tm_wrapping` .............................................................................. 16
4  C++ Class Annotation .............................................................................................................. 18
   4.1 C++ Class Level Attribute ................................................................................................... 18
   4.2 Virtual Function Overriding Rules ....................................................................................... 21
5  Failure Atomicity ...................................................................................................................... 22
   5.1 The `__tm_abort` Statement .................................................................................................. 22
6  Usage Examples ........................................................................................................................ 23
7  C/C++ Exceptions ..................................................................................................................... 26
   7.1 Commit on Exceptions ......................................................................................................... 26
   7.2 Examples .............................................................................................................................. 26
   7.3 Abort on Exceptions ............................................................................................................. 27
8  Application Programming Interface ....................................................................................... 29
   8.1 User registered commit and undo actions ........................................................................... 29
      8.1.1 Commit function ............................................................................................................ 29
      8.1.2 Undo action .................................................................................................................. 30
   8.2 Thread Id function .............................................................................................................. 30
   8.3 Internal library references removal ..................................................................................... 30
9  Generating Runtime Statistics .................................................................................................. 30
   9.1 Enabling Statistics ............................................................................................................... 31
1 About this Document

This document describes language extensions for writing a Software Transactional Memory (STM) program and the use of the Intel® C++ STM Compiler Prototype Edition 3.0, including the following items:

- Language syntax and rules
- Semantics definition
- Usage examples

This document covers the functionality that will be provided in the prototype release of the compiler. More functionality is expected to be available in subsequent releases.

1.1 Intended Audience

This document is intended for C/C++ developers who are interested in C/C++ language extensions for writing STM applications. The C++ STM language constructs proposed in this document are intended for users interested in exercising new language constructs for parallel programming, understanding the transactional memory programming model, and providing community feedback on the usefulness of these extensions with Intel® C++ STM Compiler Prototype Edition 3.0.

1.2 Document Organization

This document contains the following sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using this Document</td>
<td>Contains a summary of this document including the intended audience, national conventions, and related information.</td>
</tr>
<tr>
<td>Introduction</td>
<td>Contains the requirements for transactional memory and the areas of discussion not covered in this document.</td>
</tr>
<tr>
<td>Language Extensions</td>
<td>Contains descriptions and examples of language extensions</td>
</tr>
<tr>
<td>C++ Class Annotation</td>
<td>Contains descriptions and examples of C++ class level annotation, inheritance and overriding rules, and examples</td>
</tr>
<tr>
<td>Failure Atomicity</td>
<td>Contains description of failure atomicity and example</td>
</tr>
<tr>
<td>Usage Examples</td>
<td>Contains descriptions and examples of usage models.</td>
</tr>
<tr>
<td>C/C++ Exceptions</td>
<td>Contains definitions, descriptions and examples of C/C++ exceptions</td>
</tr>
<tr>
<td>Application Programming Interface</td>
<td>Describes the user level interface of commit function and undo action.</td>
</tr>
</tbody>
</table>
Generating runtime statistics | Contains descriptions and examples of simple transactional statistics generated from C/C++ STM runtime library
---|---
Transactional memory semantics | Describes the transactional memory semantics and memory model

### 1.3 Conventions and Symbols

The following conventions are used in this document.

**Table 2  Conventions and Symbols used in this Document**

| Courier New | Indicates an element of syntax, reserved word, keyword, filename, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant. |
| Lucida Console | Indicates the exact characters you type as input. Also used to highlight the elements of a graphical user interface such as buttons and menu names. |
| Verdana Italics | Indicates a placeholder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the placeholder. |

### 1.4 Related Information


A related document includes:

- Intel® C++ STM Compiler Prototype Edition 3.0 Compiler Release Notes
- Intel® C++ STM Compiler Prototype Edition 3.0 Runtime Release Notes
- Intel® Transactional Memory Compiler-Runtime Application Binary Interface (ABI) 1.0

### 2  Introduction

#### 2.1 Requirements

Transactional memory eases parallel programming by offering advantages for concurrency control compared to locks. To support Software Transactional Memory (STM) in unmanaged programming languages such as C/C++, the scope of software support must include at least the following items:

- Language extensions to specify and define a transaction region (or atomic section)
Compiler and runtime support to handle memory reads and writes that access shared memory locations that are not private for each thread in the transaction region executed by multiple threads.

2.2 Not Covered by this Document
This document does not provide discussions of all variants of STM semantics and STM library implementation. These materials are mentioned only in some sections where they are necessary to describe problems specific to the STM support in the Intel® C++ STM Compiler Prototype Edition 3.0.

This document does not cover internal implementation of Intel® C++ STM Compiler Prototype Edition 3.0, unless it is needed for the discussion of language extension semantics. Additionally, this document does not describe compiler optimizations to reduce STM operation overhead.

3 Language Extensions
This specification describes the C++/C constructs and keywords supported in the Intel® C++ STM Compiler Prototype Edition 3.0.

3.1 Conditional Compilation
Users may not always want to enable the STM code, thus we define a macro that can be used to mark regions of code that will be compiled only when the STM mode is enabled. This macro must never be the subject of a #define or a #undef in the user’s code.

```c
#define _TM

__tm_atomic { <statement> }
__tm_atomic { <statement> } [ else { <statement> } ]
__tm_atomic { <statement> }
__tm_atomic <statement> [ else <statement> ]
```

3.2 Specifying Atomic Code Blocks for TM
The following keyword is the fundamental construct that annotates a statement or a code block as a Transactional Memory (denoted as TM) statement or region; such statements and code blocks are referred to as the TM region for performing TM execution. A TM region marks a code block that will have automatic concurrency control when it is executed in parallel. All statements inside a __tm_atomic block execute as a single transaction. The statements in a transaction can include both direct and indirect function calls.

```c
__tm_atomic { <statement> }
__tm_atomic { <statement> } [ else { <statement> } ]
__tm_atomic <statement>
__tm_atomic <statement> [ else <statement> ]
```

Semantics:
A TM region is a code block that can be executed in parallel with automatic concurrency control. Each memory read/write in a TM region is a TM read/write with support from the compiler and runtime library. During the execution, if TM read/write conflicts are detected by TM runtime software, before committing TM reads and writes, an abort will occur and trigger the retry of the transaction from the beginning of this TM region by rolling back to the saved machine state and program state.

Two transactions, in two threads of a process, are isolated from each other. They cannot see intermediate values of memory operands. The execution acts as if all operations in one thread are completed before, or after, all the operations in the other thread.
Note, however, that there is no similar guarantee between transactional code in one thread and non-transactional code in another thread.

A transaction specified by `__tm_atomic` can be nested inside another transaction. The effects of a nested transaction are visible only when the outermost transaction commits. This is often described as “closed nesting”; on a data conflict the runtime may roll back to any level in the transaction nest (including to the outermost transaction of all) and re-execute the transaction.

The code in the `else` clause associated with an atomic block gets executed if a user abort (described in Section 5) is triggered inside of the atomic block. The code in the `else` clause is executed non-transactionally unless both the atomic block and its `else` clause are executed in a context of another (outer) transaction.

**Rules:**

The transactional memory (TM) region must be a single-entry code section but it's allowed to have more than one exit.

- If the transactional region has multiple entries program behavior is implementation dependent; the native compiler implementation will issue error messages whenever this is statically detectable.
- `continue/break/return/goto` statements are permitted while implying a commit operation of this `__tm_atomic` region with a warning message issued by the compiler if they are bounded to a `loop/switch/if–statement outside the __tm_atomic region`.
- In Intel® C++ STM Compiler Prototype Edition 3.0, `__tm_atomic` supports calling legacy functions and treats them as `irreversible` operations, that is, operations whose side effects cannot be rolled back. A transaction that executes an irreversible operation is guaranteed to commit without rolling back by switching to `serial irreversible mode` (the transaction itself is made irreversible by ensuring that it is executed serially with respect to all other transactions). Under irreversible mode, a transaction gets undefined behavior if a `__tm_abort` statement is executed in the transaction.

The following operations are not supported inside of transactions in the Intel® C++ STM Compiler Prototype Edition 3.0:

- `setjmp/longjmp` and Linux* signal operations such as `sigsetjmp/siglongjmp` in transactions.
- `__try/__finally constructs in transactions on Windows* platforms`
- unstructured parallelism, for example, forking threads in transaction, and the join that happens after the transaction commits.

Note that the compiler will issue a compile-time error, a linker error, or a runtime error if any of the above restrictions is violated.

**Example 3.1: Usage Example of __tm_atomic**

```c
// stmt1 and stmt2 execute with automatic concurrency
// control if foo is called by multiple threads. When a memory conflict is
// detected by TM runtime, it rolls back and re-executes
// transaction A.

void foo(void)
{
    __tm_atomic { // Transaction A
        stmt1;
        stmt2;
    }
}
```
Example 3.2: Nested transaction semantics

```c
void foo(int *x, int *y)
{
    __tm_atomic {                      // Transaction A
        *x = *x + 1;
        __tm_atomic *y = *y + *x;     // Transaction B
        // if memory conflicts detected, rollback and re-execute B (or A)
        } commit A, or if memory conflicts detected, rollback and re-execute A
}
```

Note that only when `commit A` occurs will the entire transaction be visible to other threads. In addition, it is semantically legal to flatten nested transactions, in which case any abort or memory conflict triggered retry in B will always cause A to be aborted/retried.

Example 3.3 – Use of continue/break in Transactions.

```c
void foo(int *x, int *y)
{
    for (int k=0; k<100; k++)
        __tm_atomic {                  // Transaction A
            if (cond) {
                *x = *x + 1;
                continue;
            } else if (test) {
                break;
            }
        } // continue/break in transaction A jumps out of this transaction
    // and commits it
}
```

Example 3.4: – Use of return/break in Transactions

```c
void foo(int *x, int *y)
{
    __tm_atomic {                  // Transaction A
        for (int k=0; k<100; k++) {
            if (*x) return;
        } // return in transaction A jumps out of this transaction
        // and commits it
        switch (cond) {
            case 1: stmt1; break;
            default: break;
        } // break in transaction A does not jump out of this transaction
    }
}
```

3.3 Specifying Waiver Code Blocks for TM

The following keyword is a construct that annotates a statement or a code block that escapes from executing in a Transactional Memory region;

```c
__tm_waiver { <statement> }
__tm_waiver <statement>
```

**Semantics:**

Operations executed in the dynamic scope of `__tm_waiver` behave the same as if they were executed outside transactions except for static checking of `__tm_waiver` rule. The execution behavior is specified below.
memory accesses are not monitored by STM runtime for conflict detection and rollback.

function invocation behaves the same as if they were invoked from non-transactional codes, in particular

- un-annotated functions are allowed and invoked without the transaction becoming irrevocable.
- annotated functions are allowed, and the non-transactional clones are invoked.

Note: The non-transactional clone of a function is the version of the function being invoked from outside transactions.

Rules:

- Nesting (static or dynamic) of _tm_waiver is allowed, but inner _tm_waiver blocks are flattened into the outermost _tm_waiver. The rationale is: inner __tm_waiver is essentially a no-op as __tm_waiver applies to the dynamic scope.

- __tm_atomic is prohibited in the lexical scope of __tm_waiver. The compiler checks __tm_atomic directly nested in _tm_waiver blocks and issues error messages. __tm_atomic dynamically nested inside __tm_waiver is prohibited and may result in undefined execution behavior.

- Explicit control-flows out of __tm_waiver block such as goto, break, continue, return, and setjmp/longjmp are prohibited. The __tm_waiver block must be a structured single-entry and single-exit block.

Example 3.5: Usage Example of __tm_waiver

```c
// the 'if' stmt with I/O operation escapes from the transaction execution
// of transaction A.

void foo(void)
{
    __tm_atomic {
        Stmt1;
        _tm_waiver if (debug) printf("I am here. \n");
        Stmt2;
    }
}
```

3.4 Specifying a Function as tm_callable

The tm_callable function attribute can be specified by using __declspec on Windows* and __attribute__ on Linux*. The tm_callable function annotates a function that can be called inside transactions. These are also referred to as tm_callable functions. Programmers can annotate STM function declarations in source and header files using tm_callable.

Windows* Syntax: Annotate a function as a tm_callable function using __declspec

```c
__declspec(tm_callable)
function declaration statement
```

Linux* Syntax: Annotate a function as a tm_callable function using __attribute__

```c
__attribute__((tm_callable))
function declaration statement
```
**Semantics:**

For every `tm_callable` function the compiler generates its transactional clone including TM instrumentation: each memory read and write is translated to a TM read barrier function and a TM write barrier function by the compiler with support from the software runtime library. A transactional clone of the original function will be used inside transactions. The original function with normal reads and writes will be used outside transactions.

**Rules:**

- A `tm_callable` function may contain `__tm_atomic` and `__tm_waiver` blocks, and `tm_callable` function may be called inside `__tm_atomic` blocks.
- A `tm_callable` function may call other `tm_callable`, `tm_safe`, `tm.wrapping`, `tm_pure`, and `tm_unknown` functions in its lexical and dynamic scopes.
- A `tm_callable` function may contain indirect function calls and virtual function calls even if it is unknown at compile-time whether the target of a function pointer is `tm_callable`, `tm_safe`, `tm.wrapping`, and `tm_pure`.
- A `tm_callable` function may have irrevocable operations and legacy function calls inside its lexical and dynamic scope.

**Example 3.6: Usage Example of `__declspec(tm_callable)`:**

```c
__declspec(tm_callable)
void UserFoo(int);
void UserGoo(float);  // it is implicitly annotated with tm_unknown
void func(void)
{
    __tm_atomic {
        UserFoo(100);    // legal use
        UserGoo(128.8);  // switch to serial irrevocable mode
    }
}
```

Note that for the indirect function calls, the compiler will generate runtime check and dispatch code to invoke either instrumented routine or non-instrumented function calls. No function pointer annotation is needed.

**3.5 Specifying a Function as `tm_safe`**

The `tm_safe` function attribute can be specified by using `__declspec` on Windows* and `__attribute__` on Linux*. The `tm_safe` function annotates a function that can be called inside transactions. These are also referred to as `tm_safe` functions. Programmers can annotate STM function declarations in source and header files using `tm_safe`.

**Windows* Syntax:** Annotate a function as a `tm_safe` function using `__declspec`

```c
__declspec(tm_safe)
function declaration statement
```

**Linux* Syntax:** Annotate a function as a `tm_safe` function using `__attribute__`

```c
__attribute__((tm_safe))
function declaration statement
```

**Semantics:**
A tm_safe function is one that has been compiled for transactional execution and is thus safe to execute atomically. A tm_safe function may contain atomic blocks. In effect, a tm_safe function contains only those operations that the system can execute atomically. For every tm_safe function, similarly to tm_callable functions, the compiler generates its transactional clone: within the transactional clone, each memory read and write is translated to a TM read barrier function and a TM write barrier function by the compiler with support from the software runtime library. A transactional clone of the original function will be used inside transactions. The original function with normal reads/writes will be used outside transactions.

**Rules:**

- A tm_safe function may contain __tm_atomic and __tm_waiver blocks, and tm_safe function may be called inside __tm_atomic blocks.
- A tm_safe functions may call other tm_safe, tm_wrapping and tm_pure functions.
- A tm_safe function may not call tm_callable functions in its lexical and dynamic scope.
- A tm_safe function may not call functions indirectly as it is unknown statically whether the target of a function pointer is tm_safe. But, it can call virtual function annotated as tm_safe or tm_pure.
- A _tm_safe function may not have irrevocable operations and legacy function calls inside its lexical and dynamic scope.

**Example 3.7: Usage Example of __declspec(tm_safe):**

```c
__declspec(tm_safe)
void UserFoo(int);

__declspec(tm_callable)
void UserGoo(float);

void func(void)
{
    __tm_atomic {
        UserFoo(100);    // legal use
        UserGoo(128.8);  // legal use
    }
}
```

Note that for the virtual tm_safe function calls, the compiler will generate runtime check and dispatch code to invoke either instrumented or non-instrumented version of a function.

### 3.6 Specifying a Function as tm_unknown

The tm_unknown function attribute can be specified by using __declspec on Windows* and __attribute__ on Linux* to annotate a function. The tm_unknown attribute annotates a function whose appropriate TM attribute cannot be determined. In case of such functions, programmers have to rely on the TM compiler analysis and runtime to decide the execution mode. The programmers can annotate TM function declarations in source and header files using this attribute.

**Windows* Syntax: Annotate a function as a tm_unknown function using __declspec**

```c
__declspec(tm_unknown)
function-declaration-statement
```

**Linux* Syntax: Annotate a function as a tm_unknown function using __attribute__**

```c
__attribute__((tm_unknown))
function-declaration-statement
```
Semantics:

By itself, tm_unknown does not have any TM semantics. When tm_unknown attribute is used to annotate a class member function, the compiler performs analysis to promote the function to tm_safe or tm_callable, or tm_pure if the function definition is seen by the compiler and recompiled by the compiler. If the function definition is not seen and compiled by the compiler, the compiler will generate a TM call to request a dynamic mode switch in the transaction calling the tm_unknown function, so TM runtime will switch to serial irrevocable mode to guarantee safe execution.

Rules:

- **tm_unknown function attribute** can be specified by using __declspec to annotate a function, this is the default if a function is not annotated.
- If a function that is not a class member function is not annotated with a TM attribute, this function is treated implicitly as it is annotated with tm_unknown. Note that tm_unknown is not permitted for non class member functions.
- If a function that is a class member function is annotated with tm_unknown, the tm_unknown attribute overrides its class-level TM attribute. See Section 4 for details.

Example 3.8 legal use of __declspec(tm_unknown)

```c
__declspec(tm_unknown) //legal use
void UserFoo(int);

void Func(void)
{
    __tm_atomic {
        UserFoo(100);       // the compiler will generate mode switch code
    }
}
```

3.7 Specifying a Function as tm_seh

The tm_seh function attribute can be specified by using __declspec. The tm_seh attribute annotates a function to indicate that Microsoft* Structured Exception Handling (SEH) should be generated for any atomic region inside that function. No C++ EH construct is permitted in the same function for transaction memory execution. It can be called both inside and outside transactions. This attribute is ignored if there is no atomic region in the function. There is no corresponding Linux* attribute, since this a Windows* specific property.

Windows* Syntax: Annotate a function as a tm_seh function using __declspec

```c
__declspec(tm_seh) function-declaration-statement
```

Semantics:

When the __declspec(tm_seh) is used, its associated function is annotated as a TM SEH function for the compiler to check for no mixed use of Microsoft* SEH and C++ EH, and to generate SEH style code for TM.

Rules:

- __declspec(tm_seh) must be used for a function with __tm_atomic constructs or else it is ignored
- __declspec(tm_seh) is legal if the function has no C++ EH construct inside
The C++ EH compiler switch (/Gx) has no effect to the function specified with __declspec(tm_seh).

Example 3.9: Legal use of tm_seh

```c
__declspec(tm_seh)
void func(void)
{
__tm_atomic {
__try {
stmt1;
exception
} __except (1) {

}
stmt2;
}
}
```

3.8 Specifying a Function as tm_pure

A tm_pure function can be specified by using __declspec on Windows* and __attribute__ on Linux*. The tm_pure attribute annotates a function that can be called both inside and outside transactions; we refer to such functions as tm_pure functions. Programmers can annotate TM functions declarations in source and header files using this attribute.

Windows* Syntax: Annotate a function as a tm_pure using __declspec

```c
__declspec(tm_pure)
function-declaration-statement
```

Linux* Syntax: Annotate as function as a tm_pure function using __attribute__

```c
__attribute__((tm_pure))
function-declaration-statement
```

Semantics:

For tm_pure functions the compiler does not generate a transactional clone. It is the programmer’s responsibility to guarantee that a tm_pure function can be executed safely inside of a transaction without requiring transactional instrumentation or switching transaction’s execution to serial irrevocable mode.

Rules:

- A function specified with tm_pure must not access, even to read, any static or non-local memory which might be modified during its execution by some other transaction. If it does so then the execution behavior is undefined.
- A tm_pure function may call tm_pure functions only, and a tm_pure function can be called in any function.
- A tm_pure function must not contain __tm_atomic blocks.

Restrictions:

The native compiler allows a transaction to call tm_pure functions without becoming irrevocable. A tm_pure function can be called both inside and outside transactions. However, it is the user’s responsibility to make sure to get expected behavior by using it properly, as no operations inside a
tm_pure function can be undone on transaction abort or retry, and conflicts with operations of other transactions cannot be detected.

Example 3.10: Legal use of tm_pure:

```c
__declspec(tm_pure)
static int UserFoo(int b)
{
    int i, x=0;
    for (i=0; i<100; i++) {
        x = x + b;
    }
    return x;
}

void func(int *a)
{
    __tm_atomic {
        --*a = *a + UserFoo(100);
    }
}
```

A few annotations have been introduced for programmers to specify a function to be called inside/outside transactions. The following table summarizes the nesting/caller/callee rules of TM constructs and TM attributes excluding tm_unknown. The first column is the enclosing construct or caller. Obviously, a function annotated with tm_pure is permitted to call functions annotated with tm_pure.

**Table 3 - nesting/caller/callee rules and restrictions**

<table>
<thead>
<tr>
<th>callee</th>
<th>__tm_atomic</th>
<th>__tm_atomic + tm_seh</th>
<th>tm_callable</th>
<th>tm_safe</th>
<th>tm_pure</th>
</tr>
</thead>
<tbody>
<tr>
<td>__tm_atomic</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>__tm_atomic + tm_seh</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>Yes</td>
</tr>
<tr>
<td>tm_callable</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>Yes</td>
</tr>
<tr>
<td>tm_safe</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>tm_pure</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>Yes</td>
</tr>
</tbody>
</table>

3.9 Specifying a Function as tm_wrapping

The tm_wrapping annotation declares a transactional wrapper function. A transactional wrapper redirects in-transaction calls of a function to a user-specified wrapper function that escapes the transaction. Transactions that execute in serial irrevocable mode may or may not execute wrapper functions as such transactions might execute uninstrumented code.

**Windows* Syntax:** Annotate a function as a tm_wrapping using __declspec

```c
__declspec(tm_wrapping(<wrapped-function-name>))
function-declaration-statement
```

**Linux* Syntax:** Annotate as function as a tm_pure function using __attribute__

```c
__attribute__((tm_wrapping(<wrapped-function-name>)))
function-declaration-statement
```
The following example shows how to declare a transactional wrapper fooTxn() for a function foo():

```c
__declspec(tm_wrapping(foo)) void fooTxn();
```

Besides C functions, the `tm_wrapping` annotation is permitted for static class members, namespace members, operators, template instantiations, and template specializations. See more usage examples listed below.

**Example 3.11: Use of `tm_wrapping` for static class members:**
```c
struct A { static int qqq(int x) }
__declspec(tm_wrapping(A::qqq))
int qqqTxn(int x)
```

**Example 3.12: Use of `tm_wrapping` for namespace members:**
```c
namespace B {
    int Q1(int x);
    int Q2(int x);
    int Q3(int x);

    __declspec(tm_wrapping(Q2)) int Q2Txn(int x);
}
__declspec(tm_wrapping(B::Q1)) int Q1Txn(int x);
using B::Q3;
__declspec(tm_wrapping(Q3)) int Q3Txn(int x);
```

**Example 3.13: Use of `tm_wrapping` for operators:**
```c
enum Intencity {verylight, light, medium, dark, verydark};
bool operator < (Intencity, Intencity);
__declspec(tm_wrapping(operator <))
bool less_int_tm(Intencity, Intencity);
```

**Example 3.14: Use of `tm_wrapping` for template instantiations/specializations:**
```c
template <typename T>
int foo(T q) {
    return sizeof(q);
}

template <> int foo<char>(char) {
    return 1;
}
__declspec(tm_wrapping(foo<char>)) int foo_char_Txn(char);
template int foo<int>(int q);
__declspec(tm_wrapping(foo<int>)) int foo_int_Txn(int);
```

**Semantics:**

Like `tm_pure` functions, the wrapper function is executed without transactional instrumentation so the programmer takes responsibility for their correct behavior. After seeing this `tm_wrapping` annotation in
the function declaration, the compiler translates every in-transaction call to `foo()` into a call to its wrapper function `fooTxn()`, which executes without TM instrumentation.

Rules:

- Inside a wrapper function, the programmer must be careful not to access memory that has been accessed transactionally as such an access may see an inconsistent or speculative value; that is, the programmer must segregate the data accessed inside the wrapper function from data that is accessed transactionally by any thread including the thread making the call. The programmer must also not execute any atomic blocks inside the wrapper function.

Inside wrapper functions, the programmer must register the proper undo and commit actions to roll back or finalize the effects of the wrapper function on an abort or commit, respectively. A commit action executes when the transaction commits. An undo action executes when the transaction rolls back due to a user abort or a conflict. The undo and commit actions are described in more detail in Section 8.

# 4 C++ Class Annotation

In addition to the function level attributes described above, which are also applicable to member functions of a class, attribute specification can also be specified at a class level to ease TM programming in C++. Although attributes are defined at a class level, they are applicable only to member functions in the class. Inconsistency can arise when inheriting base class attributes to derived methods, a set of rules are defined as to what combinations are allowed. The goal is to keep the attribute inheritance simple yet enable programmability.

## 4.1 C++ Class Level Attribute

The two attributes that are permitted at class level declaration are `tm_callable` and `tm_safe`.

**Windows* Syntax:** Annotate a class as a `tm_callable` or `tm_safe` using `__declspec`

```cpp
__declspec(tm_callable)
class-declaration-statement

__declspec(tm_safe)
class-declaration-statement
```

**Linux* Syntax:** Annotate as function as a `tm_callable` or `tm_safe` function using `__attribute__`

```cpp
__attribute__((tm_callable))
class-declaration-statement

__attribute__((tm_safe))
class-declaration-statement
```

**Semantics:**

When `tm_callable` (or `tm_safe`) attribute is used, all the member functions both virtual and non virtual become `tm_callable` (or `tm_safe`). This is equivalent to using `tm_callable` (or `tm_safe`) annotation on every member function in the class including constructors, destructors and operators (e.g. `+`, `-`) and excluding operator “new” and “delete”.

**Rules:**
A class can have \texttt{tm\_callable}, \texttt{tm\_safe} attribute or no attribute. If a class is annotated with the either \texttt{tm\_callable} or \texttt{tm\_safe} attribute, then all member functions with no attribute will become \texttt{tm\_callable} or \texttt{tm\_safe}, respectively.

Individual attributes on a member function override the class level attribute for that member function. Thus \texttt{tm\_callable}, \texttt{tm\_safe}, \texttt{tm\_pure} or \texttt{tm\_unknown} on individual member functions can override a \texttt{tm\_callable} or \texttt{tm\_safe} class level attribute. The \texttt{tm\_callable} or \texttt{tm\_safe} attribute on a member function in a class with class level \texttt{tm\_callable} or \texttt{tm\_safe} attribute is redundant.

A derived class with no explicit attribute inherits the \texttt{tm\_safe} attribute if its base class is annotated with \texttt{tm\_safe}. The \texttt{tm\_safe} on the derived class is redundant if the base class it derives from is annotated with \texttt{tm\_safe}.

A derived class with no explicit attribute inherits the \texttt{tm\_callable} attribute if its base class is annotated with \texttt{tm\_callable}. The \texttt{tm\_callable} on the derived class is redundant if the base class it derives from is annotated with \texttt{tm\_callable}.

If there are multiple base classes, a derived class inherits \texttt{tm\_safe} as long as one of its base classes is annotated with \texttt{tm\_safe}. If there are no base classes is annotated with \texttt{tm\_safe}, a derived class inherits \texttt{tm\_callable} as long as one of its base classes is annotated with \texttt{tm\_callable}.

A virtual member function in derived class inherits \texttt{tm\_attribute} based on its base class virtual member function annotation. All base class virtual member functions should have \texttt{tm\_safe}, \texttt{tm\_callable}, or \texttt{tm\_unknown} attribute. Note that \texttt{tm\_unknown} functions include functions that are not marked.

Example 4.1: Legal use of class level \texttt{tm\_callable} annotation:

\begin{verbatim}
__declspec(tm_callable) class foo {
  int     func1();
  float   func2();
  virtual int func3();
}

// the declaration above is equivalent to the declaration below

class foo {
  __declspec(tm_callable) int     func1();
  __declspec(tm_callable) float   func2();
  __declspec(tm_callable) virtual int func3();
}
\end{verbatim}

Example 4.2: Legal use of class level \texttt{tm\_callable} annotation:

\begin{verbatim}
__declspec(tm_callable)
__class foo {
  __declspec(tm_callable) int     func1();
  __declspec(tm_callable) float   func2();
  __declspec(tm_callable) int     func3();
  __declspec(tm_pure) int       func4();
  __declspec(tm_unknown) float   func5();
  __declspec(tm_callable) virtual int func6();
}

// the declaration above is equivalent to the declaration below

__class foo {
__declspec(tm_callable)

int     func1();
float   func2();
int     func3();
int     func4();
float   func5();
virtual int func6();
}
\end{verbatim}
Example 4.3: Legal use of class level \texttt{tm\_callable} inheritance:

\begin{verbatim}
__declspec(tm_callable) class aaa { ... } 
class bbb { ... }
class ccc : aaa, bbb { ... }
\end{verbatim}

// the declaration above is equivalent to the declaration below

\begin{verbatim}
__declspec(tm_callable) class aaa { ... }
class bbb { ... }
__declspec(tm_callable) class ccc : aaa, bbb { ... }
\end{verbatim}

Example 4.4: Legal use of \text{tm\_callable} for template member functions

\begin{verbatim}
template <class T>
class Water {
public:
    Water(int = 100) ;
    ~Water() { delete [] waterLoc; }
    __declspec(tm_callable,noinline)
    int check(T& rValue)
    {
        if (!isClean()) {
            rValue = waterLoc[rate];
            return 1 ;
        } else {
            return 0 ;
        }
    }
private:
    int rate;
    T* waterLoc;
};
\end{verbatim}

Example 4.5: Illegal use of \text{tm\_callable} and \text{tm\_pure} for template member functions

\begin{verbatim}
template <class T>
class Water {
public:
    Water(int = 100) ;
    ~Water() { delete [] waterLoc; }
    __declspec(tm_callable,noinline)
    int check(T& rValue)
    {
        if (!isClean()) {
            rValue = waterLoc[rate];
            return 1 ;
        } else {
            return 0 ;
        }
    }
private:
    int rate;
    T* waterLoc;
};
template <class T>
__declspec(tm_pure) // conflicts with tm\_callable in template class
int Water<T>::check(T& rValue)
{
    if (!isClean()) {

```cpp
rValue = waterLoc[rate];
return 1;
}
return 0;
}
```

**Example 4.6: Legal use of class level **tm_safe** annotation:**

```cpp
declspec(tm_safe)
class foo {
    int func1();
    float func2();
    virtual int func3();
}
// the declaration above is equivalent to the declaration below
class foo {
    __declspec(tm_safe) int func1();
    __declspec(tm_safe) float func2();
    __declspec(tm_safe) virtual int func3();
}
```

**Example 4.7: Legal use of class level **tm_safe** inheritance:**

```cpp
__declspec(tm_safe) class aaa { ... }
__declspec(tm_callable) class bbb { ... }
class ccc: aaa, bbb { ... }
// the declaration above is equivalent to the declaration below
__declspec(tm_safe) class aaa { ... }
__declspec(tm_callable) class bbb { ... }
__declspec(tm_safe) class ccc: aaa, bbb { ... }
```

### 4.2 Virtual Function Overriding Rules

Since derived class virtual functions can be invoked using a base class virtual function with the same name but different tm_attributes, we must define a set of rules showing what combinations are permitted to ensure correctness at the point of invocation. The table below captures these rules.

**Table 4 – Virtual function overriding rules**

<table>
<thead>
<tr>
<th>Derived class</th>
<th>tm_callable</th>
<th>tm_safe</th>
<th>tm_pure</th>
<th>tm_unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base class</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tm_callable</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>tm_safe</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>tm_pure</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>tm_unknown</td>
<td>yes</td>
<td>yes</td>
<td>warning and yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

In case of multiple inheritance, a function annotated with a TM attribute in the derived class may override virtual functions in the base classes only if the rules specified in Table 5 hold for every pair of functions from the base class and the derived class separately. See an example below:

```cpp
__declspec(tm_callable) class A {
```
In the example above class C inherits the \texttt{tm\_callable} attribute from class A and, as a result, C::\texttt{foo()} gets \texttt{tm\_callable} attribute as well. According to the rules specified in Table 5, it is legal for a \texttt{tm\_callable} function C::\texttt{foo()} to override both \texttt{tm\_unknown} A::\texttt{foo()} and \texttt{tm\_callable} B::\texttt{foo()}. At the same time, D::\texttt{foo()} overrides its own class-level \texttt{tm\_callable} attribute with \texttt{tm\_pure}, the marking of D::\texttt{foo()} with \texttt{tm\_pure} is illegal as rule shown in the table 5.

5 Failure Atomicity

5.1 The \texttt{__tm\_abort} Statement

Another TM construct, \texttt{tm\_abort}, is a statement that rolls back the state to what it was on entry to the innermost transaction enclosing the \texttt{tm\_abort} statement. The \texttt{tm\_abort} construct must be lexically bound to a TM region.

Semantics:

The \texttt{__tm\_abort} statement causes a control flow change and execution proceeds from the lexical statement that follows the transaction block. All effects of computation in the transaction up to the point of \texttt{__tm\_abort} statement are discarded.

Example 5.1: Legal use of \texttt{__tm\_abort} in a transaction

\begin{verbatim}
void foo(void)
{
    __tm_atomic {
        stmt1;
        __tm_atomic {
            stmt2;
            if (cond) __tm_abort;
        }
        stmt3;
    }
    stmt4;
}
\end{verbatim}

In above example, \texttt{__tm\_abort} rolls back to the starting state of transaction B, aborts iff the transaction is not irrevocable, and continues execution with stmt3. If transaction B is irrevocable a fatal runtime error will occur. Another example below shows an illegal use of \texttt{__tm\_abort}.

Example 5.2: Legal use of \texttt{__tm\_abort} in a transaction

\begin{verbatim}
void foo(void)
{
    stmt1;
    if (c) __tm_abort // illegal use as it is not lexically bound to a TM region
\end{verbatim}
6 Usage Examples

Wherever possible we want to place limited restrictions on the TM usage models, so that the language extensions are as general as possible with good portability and flexibility for different TM execution models. The following examples show legal and illegal uses of TM extensions in programs.

Example 6.1: Nested transactional memory code-block usage

```c
void foo(int *x, int *y)
{
    __tm_atomic {
        *x = *x + 1;
        __tm_atomic {
            y = *y + *x;
            if (*y == -1) *y = 10;
        }
    }
}
```

Example 6.2: Single-entry and multiple exits usage

```c
int foo(int *x)
{
    __tm_atomic {
        while (1) {
            if (x == NULL) break;
            else if (x->data == 100 || x->next == NULL) return (x->data);
            x = x->next;
        }
        return (1);
    }
    if (y == NULL) goto L1;
    return (1);
}
```

Example 6.3: Illegal usage of multiple-entry multiple-exit

```c
int foo(int *x, int *y)
{
    __tm_atomic {
        L1:
        while (1) {
            if (x == NULL) return 0;
            else if (x->data == 100 || x->next == NULL) return (x->data);
            x = x->next;
        }
        if (y == NULL) goto L1;
        return (1);
    }
}
```

Example 6.4: Calling a function containing __tm_atomic section and also marked as tm_callable in the __tm_atomic region

```c
__declspec(tm_callable)
static int gooo(int *x)
{
    int m = 0;
    __tm_atomic {
```
{
    while (m < 10) {
        \*x = \*x + 10;
        m = m + 1;
    }
}
return (m);

void func(int \*y)
{
    int m;
    __tm_atomic {
        m = goo(y);
    }
    printf("m = %d, y = %d\n", m, y);
}

Example 6.5: Calling a \texttt{tm callable} function in a TM region

\begin{verbatim}
_declspec(tm_callable)
static int goo(int \*x)
{
    int m = 0;
    while (m < 10) {
        \*x = \*x + 10;
        m = m + 1;
    }
    return (m);
}

void func(int \*y)
{
    int m;
    __tm_atomic {
        m = goo(y);
    }
    printf("m = %d, y = %d\n", m, y);
}
\end{verbatim}

Example 6.6: Calling a TM callable function inside and outside the TM region

\begin{verbatim}
_declspec(tm_callable)
static int goo(int x[10])
{
    int m;
    for (m=0; m<100; m++) {
        x[m] = x[m] + 10;
    }
    return (m);
}

int func(int y[10])
{
    int m, n;
    __tm_atomic {
        m = goo(y);
    }
    n = goo(y);
    return (m+n);
}
\end{verbatim}

Example 6.7: An example of using \texttt{\_\_tm_abort} in a parallel loop for failure atomicity

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define FAILURE_CODE1   288
#define FAILURE_CODE2   800

```c
```
int succTrans = 0;

__declspec(tm_callable)  // Linux* syntax is __attribute__((tm_callable))
static int function_checker(int code)
{
  if (code == FAILURE_CODE1 || code == FAILURE_CODE2) {
    return 1;
  }

  __tm_atomic {
    succTrans += 1;
  }
  return 0;
}

int main(int argc, char*argv[])
{
  int i, x = 0;
  int failure = 0;
  int checkpoints = 2000;

  srand(0);
  #pragma omp parallel for private(x) firstprivate(failure)
  for (i = 0; i < checkpoints; i++) {
    x = rand();
    __tm_atomic {
      failure = function_checker(x);
      if (failure>0) {
        __tm_abort;
      }
    }
  }

  if (succTrans == checkpoints) {
    printf("Launch the aircraft!\n");
  }
  else {
    printf("%d failures reported and can not launch the aircraft!\n", checkpoints-succTrans);
  }
  return 0;
}

Example 6.8: A simple pedagogic example of using __tm_abort for failure atomicity

// This introduces the classic pattern for detecting transaction failure.
#include <stdio.h>
static short value = 3;
static int calc_value(void)
{
  __tm_atomic {
    value = value*3;
    if (value < 0) __tm_abort;
  }
  else {
    printf("Value overflowed\n");
    return 0; // Failed
  }
  return 1; // Succeeded
}
int main(int argc, char *argv[])
{ int i;
for (i=1; i < (1<<8*sizeof(short)); i++) {
    if (!calc_value()) break;
}
printf('Max power of 3 which fits in a short is %d, (%d)\n',
    i, (int)value);

7 C/C++ Exceptions

The behavior of the TM constructs in the presence of C++ exceptions has not been discussed, though this is an important issue related to TM language extensions being worked on. The semantics for exceptions inside transaction decided upon may provide implications of what is needed for support in the compiler, runtime library and hardware to make C++ exceptions work correctly and effectively with TM constructs.

Every TM region will have an implicit exception handler associated with the TM region. When the code within the TM regions is executed and no exception occurs before reaching the commit phase of the TM region the TM handler is not invoked.

If an exception occurs while executing an instruction inside the TM region, the OS/runtime system takes control and calls the TM exception handler if no other handler handles the exception inside the TM region. The TM handler by default either the current transaction by committing the transaction and rethrows the exception so the exception can be propagated outside the TM region to enable other handlers to catch it. A transaction will be aborted and the exception will be rethrown if the exception was raised with a __tm_abort throw. See Section 7.3 for details.

7.1 Commit on Exceptions

The TM handler could either commit or abort the current transaction when an exception is raised. The justification for committing when an exception occurs is that the memory retains its values as seen at the time of exception. If the TM handler aborts the TM region, then the memory may be inconsistent as some of the values may be rolled back. If an object in C++ is thrown and this object is undone during an abort, then the object may be inconsistent when a handler outside the TM region is examining this object.

During commit of the transaction if memory conflicts are detected, the TM region is aborted and retried similar to a conflict occurring when committing a TM region outside an exception handling mechanism.

7.2 Examples

Below are few examples of various exception handling scenarios in the presence of the TM region.

In the following examples the TM_HANDLER as shown in the code is not the user code, but is shown for explanation.

Example 7.1: Simple TM region

```c
void func(void)
{
    __tm_atomic {
        stmt1;
        exception
        stmt2;
    } TM_HANDLER // generated by the compiler
}
```

If an exception is raised after statement 1 then the TM_HANDLER commits stmt1 and rethrows. Stmt2 is not executed.
Example 7.2: Nested TM region

```c
void func(void) {
    __tm_atomic {
        stmt1;
        __tm_atomic {
            stmt2;
            exception stmt3;
        } TM_HANDLER_2
        stmt4;
    } TM_HANDLER_1
}
```

If an exception is raised after stmt2, **TM_HANDLER_2** is called by the OS/runtime as part of the unwinding process. **TM_HANDLER_2** commits stmt2 and rethrows the exception so the runtime can now invoke **TM_HANDLER_1**. Stmt3 and stmt4 are not executed. **TM_HANDLER_1** commits stmt1 and rethrows the exception. Stmt 4 is not executed. The compiler could optimize and install only one handler for the outer TM region in this example.

Example 7.3: Simple catch in TM region

```c
void func(void) {
    __tm_atomic {
        try {
            stmt1;
            exception stmt2;
        } catch {
            ........
        }
        stmt3;
    } TM_HANDLER
}
```

If an exception is raised after stmt1, the catch handler will be called. If the handler handles the exception and subsequently the execution resumes at stmt3, the **TM_HANDLER** will not be called. In the event the **catch/except** does not handle the exception then the **TM_HANDLER** will be invoked, once the appropriate action is taken, and the exception will be rethrown to propagate to other handlers outside the TM region.

Example 7.4: Simple TM region nested in try-catch

```c
void func(void) {
    try {
        __tm_atomic {
            stmt1;
            exception stmt2;
        } TM_HANDLER
        } catch {
            ........
        }
    }
```

If an exception is raised after stmt1, the **TM_HANDLER** will be invoked; once the appropriate action is taken the exception will be re-thrown so that the catch handler can now have the option to handle the exception.

### 7.3 Abort on Exceptions

While choosing the default behavior for a thrown C++ exception in a TM region to be a commit o this region, we allow programmers to use an abort-and-throw syntax to specify that a given exception should abort the TM region it escapes:
__tm_abort throw <exception>;

This abort-and-throw syntax combines the __tm_abort statement with the throw statement. It simultaneously specifies that the block should be aborted, and that a given exception should be thrown from the aborted block. In some cases the exception thrown may be one that has been thrown by some code in (or called from) the TM region, and in other cases it may be more appropriate to construct a different exception based on state observed from within the TM region before it is aborted. The abort-and-throw statement may optionally leave out the exception expression, in which case it specifies an abort and rethrow of the current exception. Our current support does not allow re-throwing exceptions with __tm_abort, as we can not maintain state of the thrown object in this case. In addition, we don’t support exceptions thrown via a pointer, e.g. __tm_abort throw new Exception() is not supported.

The following code fragment specifies an idiom where an exception aborts the __tm_atomic block it escapes and propagates to a catch block higher up the stack.

```
__tm_atomic {
  try {
    __tm_abort throw obj
  }
}
```

Note that an abort and throw cannot be caught by any try-catch block nested within the TM region that it aborts; for example, in the following code, the first catch block never catches the exception thrown by the abort and throw:

```
try {
  __tm_atomic {
    try {
      __tm_abort throw X();    // the compiler will issue an error.
    } catch (X& x) {
      assert(0);    // never reached!
      __tm_abort throw X();    // the compiler will issue an error.
    }
    cout << "Caught X!" << endl;
  }
} catch (X& x) {
  assert(0);    // never reached!
}
```

If the exception was allocated or modified within the TM region, then it may contain or refer to state that will not be meaningful after the TM region is aborted. We describe a mechanism that gives natural default behavior for many simple cases, but can be overridden by the programmer when appropriate.

The first concern is that aborting the TM region may destroy the exception object or roll back changes made to it within the TM region. However, this issue exists already in C++ for exceptions that are allocated on the stack (which is about to be unwound), and the problem is already addressed. Specifically, when an exception is thrown, the C++ runtime typically clones it into a special “exception area” (using the exception’s copy constructor). By preventing the rollback from undoing the copies made in the special area, we ensure that there is no issue for simple exceptions. Programmers can provide copy constructors for more complicated ones that refer to additional state that may be rolled back. Programmers may also opt to replace the thrown exception with a new one, perhaps of a different type, indicating that an exception was thrown from an aborted TM region.

We note that, while our approach is convenient and flexible, it does not prevent all possible mistakes. In particular, a programmer may overlook the fact that an object whose pointer is copied by a complicated copy constructor is destroyed as a result of a transaction being aborted, which would result in a dangling pointer. Consequently, it is the programmer’s responsibility to ensure that the exception object is meaningful after the TM region is aborted; otherwise unexpected behavior is possible.
8 Application Programming Interface

TM runtime offers the programmers more flexibility and allows them to write more expressive code by making some utility functions available for a direct use. The declarations of the available functions can be found in the itmuser.h header file available in the distribution.

8.1 User registered commit and undo actions

To allow higher level libraries or user code to be made usable inside transactions, we provide an interface that allows higher level code to register callback functions to be executed on transaction abort and commit. One possible usage scenario for this interface is related to using the tm_warping annotation. The tm_warping annotation allows user functions to be executed non-transactionally from inside a transaction. Because such functions are uninstrumented, we need a mechanism allowing effects of operations executed by such functions to be reverted on transaction abort and committed on transaction commit.

8.1.1 Commit function

The addUserCommitAction function registers a commit callback:

```c
void _ITM_addUserCommitAction(_ITM_transaction *,
    _ITM_userCommitFunction,
    _ITM_transactionId_t resumingTransactionId,
    void * arg);
```

The user commit function is a pointer to a function that will be called when the appropriate transaction commits (see the next paragraph for how this is controlled). The resumingTransactionId determines at which commit the function will be invoked (transaction ids are assigned automatically by the runtime and can be retrieved using _ITM_getTransactionId function). The following example illustrates this:

```c
userInfo_t * outer = userInfoFactory();
__tm_atomic {  // Transaction 1
    userInfo_t * t = userInfoFactory();
    __tm_atomic {   // Transaction 2
        __tm_atomic {  // Transaction 3
            t->destroy();
            outer->destroy();
        }
    }
}
```

We assume that userInfoFactory and userInfo_t::destroy manage their own transactional state. It’s clear that you can’t really execute the destroy method until it is impossible to retry any transaction that would cause you to execute anywhere where the object was alive. Otherwise, as a result of transaction re-execution, the destroy operation could be executed more than once which would result in incorrect behavior.

Consider what this means for the two destroy operations in the code above. The first destroy operation can commit only when transaction two commits (since we can then only retry transaction one, and the object pointed to by t was not allocated at the start of that transaction), whereas the second destroy can only be performed when transaction one commits since only then can no retry take us back to a point where the object pointed do by outer was alive.

This can all be achieved by having the allocation functions remember the transactionId in which the allocation occurred (using _ITM_NoTransactionId for allocations outside the current transaction
nest), and then passing that as the resumingTransactionId to the addUserCommitAction function.

The user commit function is called when a commit occurs that returns to a transaction whose id is less than or equal to the resumingTransactionId. Passing a transaction id that is larger than that of the current transaction’s as an argument to addUserCommitAction function results in undefined behavior.

The final “void * arg” is an argument that will be passed to the commit function when it is called.

8.1.2 Undo action

A user undo action is registered by calling the function

```c
void _ITM_addUserUndoAction(_ITM_transaction *,
    _ITM_userUndoFunction, void *arg);
```

This function is called if a transaction is aborted or retried to allow the user code to undo any operations and restore its internal state to that before the transaction started.

8.2 Thread Id function

Since the TM runtime can generate statistical information that is indexed by a numeric thread id, it is useful to provide that id to the user, so that she can print it in relevant messages, or provide a key to help understand which thread in the library's report corresponds to which part of her code.

We therefore provide the function

```c
int _ITM_getThreadnum(void);
```

8.3 Internal library references removal

If the user library wants to release store inside a transaction that may have been seen by the TM runtime, it must ask the runtime to remove any internal references it may have into the store being released. Otherwise in an undo-log based runtime implementation there might be pointers in the undo-log that would be written through in the event of a later abort or retry. Since the user library has freed that store, such writes would be fatal.

```c
void _ITM_dropReferences (_ITM_transaction *,
    void * __start,
    size_t __size);
```

9 Generating Runtime Statistics

The STM runtime library supports the generation of simple transactional statistics that may help a user to understand the performance characteristics of their application and to locate performance bottlenecks. Statistics are generated per lexical transaction, per thread, and per program. All statistics are simple counters. Two modes of statistics are provided, simple, and verbose. Below is a list of all of the statistics generated:

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Description</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transactions</td>
<td>The number of times a transaction is executed.</td>
<td>simple</td>
</tr>
<tr>
<td>SerialTransitions</td>
<td>The number of times a transaction transitioned to serial irrevocable mode, disallowing other transactions from executing in parallel. This</td>
<td>simple</td>
</tr>
</tbody>
</table>
happens when code is compiled without the required instrumentation.

<table>
<thead>
<tr>
<th>Instrumentation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No statistics are generated. This is the default mode.</td>
</tr>
<tr>
<td>Simple</td>
<td>Only statistics for simple mode are generated.</td>
</tr>
<tr>
<td>Verbese</td>
<td>All statistics are generated.</td>
</tr>
</tbody>
</table>

When statistics are enabled in either simple or verbose modes, all of the generated statistics will appear in a file named itm.log. The output in the log file is broken down into 3 sections: the totals for each thread over all transactions and for each individual transaction, the totals for the whole program for each lexical transaction and the totals for the whole program over all transactions. Lexical transactions are identified by the code location where they occur. Totals for each statistic (aside from Transactions) show the Min, Max, and Mean value over all occurrences of that transaction.

9.2 An Example Program
Consider the following simple program

```c
foo.c:
1: #include <stdio.h>
2: int a = 0;
3: int main() {
4:     #pragma omp parallel
5:     { int i;
6:         for (i=0; i<100000; i++)
7:             __tm_atomic {
8:                 a++;
9:             }
10:     }
11:     printf("a = %i\n", a);
12:     return 0;
13: }
```

When this program was run with two threads in verbose mode, it generated the following itm.log file:

```
STATS REPORT
THREAD TOTALS
Thread 0          Min  Mean  Max  Total
Transactions      100000
Retries           0    0.14  2    14094
BytesRead        4    4.56  12   456376
BytesWritten     4    4.00  4    400000
```
10 Transactional Memory Semantics

In correctly synchronized programs transaction execute as if all operations of one transaction are completed before, or after, all operations in another transactions. A program is correctly synchronized if it contains no data races, that is, if two threads cannot simultaneously access the same shared data. A programmer can ensure that a program is correctly synchronized by enclosing all shared memory accesses in transactions.

Example 10.1: A correctly synchronized program

```c
// This program is correctly synchronized because shared variable p
// is read and written only in a transaction.

Thread 1

__tm_atomic {    __tm_atomic {
    p = new Foo();   if (p != null)
    f();         t = p->x;
}           }
```

In the presence of data races between transactional code in one thread and non-transactional code in another thread the execution can produce arbitrary results. This behavior is consistent with the upcoming revision of C++ standard that allows programs with data races to behave in an arbitrary way.

Example 10.2: An incorrectly synchronized program
// This program is incorrectly synchronized. The read of p in Thread 2 is
// not enclosed within transaction and can execute simultaneously with the
// write of p in Thread 1. This program can behave in arbitrary
// unpredictable ways. For example, dereferencing p in Thread 2 can result
// in segmentation fault violation. Practically this can happen if
// initially p==null and transaction in Thread 1 aborts between null check
// and dereferencing of p in Thread 2.

Thread 1                                  Thread 2
  __tm_atomic {
    p = new Foo();   if (p != null)
    f();        t = p->x;
  }

A program can access data outside of transaction without causing a data race if at the time of the access
the data cannot be accessed from other threads.

Example 10.3: Privatization pattern

// This program is correctly synchronized. Thread 2 removes the element
// for a shared queue. After this it cannot be accessed from other threads
// so Thread 2 can read and write that element outside of a transaction.

Thread 1                                  Thread 2
  __tm_atomic {
    foo(queue);
  }                                  __tm_atomic {
    p = queue->dequeue();
  }
  bar(p);

Example 10.4: Publication pattern

// This program is correctly synchronized. Thread 2 accesses p outside of
// transaction before it is accessible to other threads.

Thread 1                                  Thread 2
  __tm_atomic {
    foo(queue);
  }                                  __tm_atomic {
    queue->enqueue(p);
  }

Transactions may contain locks. However, locks are legacy functions. Consequently they are irrevocable
operations and will cause serialized execution of the transaction. Using locks to control concurrent access
between transactions and non-transactional code is, thus, correct but inefficient.

Example 10.5: Transactions inter-operate with locks.

// This program is correctly synchronized. However, transaction in
// Thread 1 will be executed serially with respect to other transactions
// because it contains locks.

Thread 1                                  Thread 2
  __tm_atomic {
    lock (L)    lock (L)
    p = new Foo();   if (p != null)
    f();        t = p->x;
    unlock (L)
  }

Lock operations break the atomicity of transactions. Transactions may use locks to safely communicate with
non-transactional code.
Example 10.6: Communication via locks

// Locks allow to break atomicity of transaction without creating a data
// race. This program terminates only if actions in Thread 2 execute in
// between of unlock (L) and lock (L) in Thread 1. TM semantics allows
// such an execution, but does not guarantee that it will be produced,
// as this partially depends on the thread scheduler.

Thread 1

```
__tm_atomic {
    lock (L);
    sendMessage();
    unlock (L);
}
```

Thread 2

```
lock (L);
recvMessage();
sendReply();
unlock(L);

lock (L);
recvReply();
```