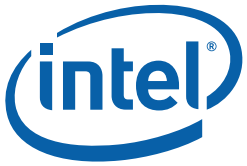

Intel® Transactional Memory Compiler and Runtime Application Binary Interface

Intel® Corporation

Document Number: 318523-002US

Revision: 1.0.1

World Wide Web: http://www.intel.com

Date: November 12, 2008

Disclaimer and Legal Information

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.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

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

Intel TM ABI (1.0.1)

Intel Corporation

Contents

1 Objectives

This document aims to define an application binary interface between IA32 and Intel(R)64 assembler level code, and the transactional memory library. The code may either have been written by an expert user in assembler, or may be the output of a native code compiler, binary translation or a JIT (though binary translation or a JIT may have more options for inlining than we contemplate here, and they are not expected to be the primary users of this specification).

1.1 Independence from TM implementation

We want to enable the same user code to be linked against different implementations of the transactional memory system without requiring recompilation. Ideally we would like to be able to change the TM implementation used by a program at load time by choice of dynamic library, though that may prove to have too large a performance overhead to be feasible.

A strong implication of this objective is that inlining code from the TM library into the user's program can only be accepted if either the inlined code is demonstrably useful to all conceivable TM library implementations (unlikely!), or we can defer the choice of code to inline to the time when we know that TM implementation is going to be used (i.e., link or load time).

Where a user is prepared explicitly to bind a particular TM library implementation to their code we may be able allow a more optimized implementation, though that is beyond the current scope of this interface.

1.2 TM implementation restrictions

Wherever possible we want to place no restrictions on the TM library implementation, so the ABI is intended to be as general as possible.

1.3 Overhead

We want to achieve the highest possible performance for the TM implementations. We therefore require the lowest possible overhead from the ABI. This implies that in future revisions we may choose to use non-standard calling conventions, use faster thread-local access, or suggest load-time code modification where this can reduce overhead. However since we follow Hoare in believing that "Premature optimization is the root of all evil" such optimizations are not addressed in this initial version of the specification.

1.4 Target environment

1.4.1 Operating system

We target both Windows and Linux operating systems running on both IA32 and Intel(R)64 architectures. Future versions of this interface may provide tailored versions of the ABI functions optimized for a particular OS.

1.4.2 Target processors

Although we are providing an interface for code to run on legacy processors we assume that we are dealing with at least Pentium II processors, i.e. that the processors have MMX technology and that we

need not support loads and stores to/from the 8087 stack. Future versions of this interface may provide tailored versions of the ABI functions optimized for a particular target processor.

1.5 Languages

We are considering C/C++ as the main target languages, though Fortran or other natively compiled languages could also use the interface. Since a standard conforming compiler cannot change the C/C++ object layout, we assume a TM coherence protocol based on memory addresses, rather than language level objects.

2 Non-objectives

2.1 General end-user library interface

It is not our objective to provide a general user level interface, but rather an optimized run-time interface for a compiler. We no more expect users to call these routines directly than we expect them directly to call the run time implementation of Fortran I/O, or C++ exception handling. Any use of the ABI directly by the programmer may violate assumptions made by the compiler and result in undefined behavior.

2.2 Interoperability

This interface intentionally does not address the issues of multiple TM systems inter-operating in the same address space, or of the inter-operability of a single TM system and lock based code. These issues, while important, are more concerned with the internal implementation of TM libraries than the interface between the library and a compiler.

3 Library design principles

3.1 Calling conventions

In the future we may choose to use non-standard calling conventions; however in this initial version of the library we use the `__fastcall` convention in 32 bit code and the default calling convention in 64 bit code. For 32 bit code the __fastcall convention on Linux and Windows appears to be identical, for 64 bit code the standards differ.

In gcc (and the Intel compiler on Linux) the `__fastcall` calling convention is written using `__attribute__(((regparm(2)))`, rather than `__fastcall`. We use `regparm(2)`, rather than `regparm(3)` on Linux to allow more code sharing between the Linux and Windows implementations, since the `regparm(2)` calling convention is the nearest to Window's __fastcall convention (though it uses different registers for arguments). If performance on 32 bit Linux ever becomes a significant issue we should consider using the `regparm(3)` convention.

3.2 TM library algorithms

We would like the interface to support libraries that implement any style of TM; that is, we want the same code to be linked against any style of TM library. We do not want to mandate the choice of a direct write or write buffering (eager or lazy version management) STM algorithm in the interface to the TM library. Similarly, we do not want to mandate the choice of either optimistic or pessimistic concurrency control in the STM algorithm. We even want to support efficiently libraries that implement transactions using simply a single global lock.

It would greatly simplify the interface to the TM library if we could structure it so that the actual load and store operations continued to be in the non-library code. Unfortunately that does not seem to be possible, since many implementations of the TM read and write barrier operations require either

operations both before and after the actual data transfer instruction, or the ability to repeat the data transfer instruction, or both.

This conclusion implies that

1. the write barrier routine must be passed both the address and value to be stored.
2. the read barrier routine must return the result of the load operation.

As well as causing a large explosion in the number of routines required, this also has an efficiency implication, since it will often force additional move operations in the code after the read barrier to put the result into the required target register, or before the write to move the source register to the argument register. That is unpleasant additional overhead, but appears unavoidable.

In the general case we have to handle at least the following loads and stores:

Datum	**Operation**	**Source or Target**	**Routines per Direction**
1,2,4 and 8 byte values	`mov`	GPR	4
4 and 8 byte values	`mov`	Mmx register	2
16 byte pair of doubles	`movapd, movupd`	Xmm register	2
8 byte double	`movsd, movhpd, movlpd`	Xmm register	3
8 byte pair of floats	`movhps, movlps`	Xmm register	2
4 byte float	`movss`	Xmm register	1

There are also additional specialized SSE*n* loads/stores that may require support (`movshdup, movsldup, movddup, maskmovdqu, movntpd, movntdq, movnti, lddqu`); we ignore these in this initial version of the interface. Similarly we expect that we will not need to support the X87 state save and restore operations (`fststw, fnstw, fstenv, fnstenv, fsave, fnsave, fxsave, fldenv, frstor, fxrstor`), to or from transactionally managed memory.

This gives us a total of 14 load routines and 14 store routines. However until we use non-standard calling conventions some of these routines are unnecessary, since the act of calling the barrier routine will force the argument into a different register or to memory. In particular none of the calling conventions uses MMX registers, and in 32 bit mode none uses xmm registers. We therefore require only the following loads/stores in this initial specification.

Datum	**Routines per Direction**
1,2,4 and 8 byte values	4
float	1
double	1
long double	1
__m64	1
__m128	1
__m256	1
complex*8, complex*16,	3

complex*32	

This gives us twelve load and nine store routines. By moving up to this higher (language data type rather than machine register) level and relying on the normal compiler calling convention we lose significant performance for floating point operands (particularly on IA-32, where floating point operands are passed in the stack), and also make it harder to use this interface from dynamic instrumentation where individual load and store instructions are being replaced.

The routines to support complex data types are provided as a convenience to the compiler, in the library they may well end up as additional entrypoints to pre-existing routines.

3.3 Optimized load and store routines

3.3.1 "After" operations

In addition to the default versions of the load and store operations we provide two more versions for each data type that the compiler can call when it can guarantee that the specific load or store being executed is dominated by another load or store to the same address with the same width that has occurred in the same transaction, or one nested inside it. Note that this definition is somewhat subtle; it means that operations performed in nested transactions can influence the choice of routine in the outer transaction, but that operations in the outer transaction cannot influence the choice of routine in the nested transaction. This is the behavior that is required because to support partial rollback the nested transaction may need to take special actions on its first access to a location whether or not the location was already accessed in the outer transaction.

In the case where the compiler can guarantee that both a load and a store dominate the operation (in either order), then the "after Write" version of the routine should be used in preference to the "after Read" version.

These versions are solely provided for optimization. Code emitted by the compiler would still be semantically correct if it never called these versions of the routines. The "after" version of a routine, however, should be significantly faster than the general routine. Note that calling these routines in cases where the compiler cannot guarantee that the required condition is met is an undetectable error, and may cause hard to find bugs at runtime.

3.3.2 "Read for Write" operations

There are occasions when the compiler knows that a read operation on a shared variable will always be followed by a write. Expressing this information to the runtime is useful to runtimes that perform eager write locking because it allows the runtime to take a write lock on the variable immediately, and thus avoid the need to promote a read lock (in a pessimistic readers implementation), or perform additional validations (in an optimistic readers implementation). To support this we provide "read for write" functions.

Given a read that will always be followed by a write, there are two possible cases that can occur:

1. no read dominates this read;
2. this read is dominated by a preceding read, but this read does not post-dominate the previous one, for example:

   ```
   y = x;
   if (…) x++;
   ```

   Here in the "x++", we know that x has already been read, and that we're about to write x, but the previous read is not guaranteed to be succeeded by the write implicit in x++, so it is sub-optimal

to generate it as a read for write, since it expresses a write that may not happen, and could therefore cause an unnecessary conflict.

This might lead one to introduce both “read for write” and “read after read for write” operations, however the benefit of “read after read for write” appears to be very small, so we choose only to provide “Read for Write”.

From the point of view of the library after a call to the Read for Write function its argument is in the written state, so subsequent calls generated by the compiler for the same address will optimally use the “after Write” versions of the data-transfer functions..

As an example, consider code like this inside a transaction, with the optimal functions provided as comments.

```
 = x;  // RfW
 = x;  // RaW
x=  ;  // WaW
```

The use of RfW functions is entirely optional, code will function correctly without them, so their use is only an optimization. Similarly using RfW when a subsequent write does not occur is only a performance issue.

3.4 Aligned load and store routines

Since the TM library is locking memory at a larger granularity than single bytes, if it is dealing with accesses that are not occurring on their natural alignment it will have to take a different code path to lock more than one slice. As the compiler often knows which accesses cannot be misaligned there is a possible optimization in which we have two different access routines, one for known naturally aligned operations (which can avoid the test for access to two slices) and one for accesses that might be misaligned.

However, evidence suggests that this optimization has only a small effect; therefore we avoid the extra (close to) doubling in the number of data transfer routines and do not provide support for it in this version of the library.

If we do later decide to implement this optimization it will be a relatively compatible upgrade, since the new library will have to maintain all the existing entry-points, so it will remain binary compatible with code compiled by an older compiler.

3.5 Data logging functions

For scalar local variables that are live-in to a transaction, the compiler can generate code that explicitly saves and restores their values, thus allowing it not to have to instrument accesses to such variables, while allowing them to be restored correctly on transaction retry or abort. However for accesses to components of larger local objects (such as arrays), the compiler cannot reasonably save and restore the whole array, particularly if only a few elements are accessed. Similarly for thread local variables used inside a tm_callable function the compiler cannot save their values locally since the abort/retry point is not in scope. Therefore there is benefit to having a set of calls that allow the compiler to request the runtime library to save values and restore them on abort or retry.

While it would be possible for the compiler to use existing read and write functions to access such variables, that would be more costly than is required (since such accesses may require read or write locks, which are unnecessary for these private variables, and which could introduce false conflicts).

If the library implementation uses direct, in place, updates, then it must already have an undo-log scheme internally that can be extended to implement these functions. For a buffered update library the implementation of these functions requires an undo-log that would not otherwise be present; however the optimization should still be worthwhile.

There is no need for either

1. these functions to return a value

or

2. read, write, or any of the “after” versions of these functions.

The second argument to the log functions is const qualified, even though the target will be written in the abort or retry path, since the const qualification expresses to the compiler the local behavior of the function that does not write the target of the pointer inside the call.

The compiler may generate a call to the log function on any write to each updated address inside each transaction, but need not generate any calls on subsequent accesses to the same address in the same transaction (though additional calls are not erroneous, merely sub-optimal).

Entering a nested transaction starts a new scope within which an abort or retry may happen, therefore the first update inside a nested transaction does require a logging operation even if a variable had already been logged in the outer transaction.

3.5.1 Example

Consider code like this

```
static void foo(void)
{
      int values[100];
      extern int which();

      initializeValues(&values, 100);
      __tm_atomic {
            int idx = which();
            values[idx] = 27;
            …Code which could cause a conflict and retry…
            values[idx]++;
      }
}
```

We expect it to generate code something like this (where all of the transaction setup is elided since it merely distracts from the point of this example).

```
static void foo(void)
{
      int values[100];
      extern int which(void);

      initializeValues(&values, 100);
      __tm_atomic {
            int idx = which();
            _ITM_LU4 (td, (uint32 *)&values[idx]);

            values[idx] = 27;
            …Code which could cause a conflict and retry…
            /* N.B. no need to log values[idx] again, since this update
             * is dominated by a logging call for the same address and
             * size.
             */
            values[idx]++;
      }
}
```

3.6 Scatter/gather calls

If we need to be able to make system calls from within a transaction, and we want to support a write-buffering TM library, then we will have to provide scatter and gather routines, since the combination of the "slice" based nature of the coherency implementation and the write buffering can lead to arrays that the user code thinks are contiguous in memory actually being non-contiguous at run time. If such an array is passed via a pointer to a system call (e.g. read or write), the kernel will access only the contiguous buffer, which would be incorrect.

An example would be code like this

```
char buffer[256];   // Assumed initialized elsewhere.
atomic {
    buffer[0] = 'a';
    write (1, &buffer[0], 256);
}
```

In a write buffering scheme the write to buffer[0] will cause the first "slice" of buffer to be copied into a write buffer, where the first byte will then be overwritten. However the tail of buffer will not be copied. On the write call the address passed to the write will be that of the original buffer, which will not contain the update of element 0 (since that's still in the uncommitted write-buffer).

In effect the problem is that the kernel's accesses to the user address space are not obeying the read/write barrier semantics that are required to see the same apparent image of memory as would be seen by the transactionally executing thread.

In an STM system we will likely have to provide STM compiled versions of any system calls that we want to allow from within a transaction. Those wrappers would then contain the necessary calls to the scatter/gather routines we would provide here.

Since the issue of system calls within transactions is still the subject of active research we merely note this issue here, and do not provide scatter/gather routines in this version of the interface.

3.7 Serial and irrevocable mode

To support performing unrestricted I/O operations and calling un-instrumented functions, the ABI requires the library to support *serial irrevocable* transactions. The ABI defines a function (`changeTransactionMode`) that explicitly switches to the serial irrevocable mode. A *serial* transaction precludes any other transaction from executing at the same time, implemented via a global lock that serializes execution of all transactions. An *irrevocable* transaction cannot be rolled back. It is a runtime error, therefore, to abort or retry an irrevocable transaction via the `abortTransaction` or `rollbackTransaction` ABI functions (described later). A *serial irrevocable* mode transaction is both serial mode and irrevocable. Serial irrevocable transactions require no read or write barriers.

The ABI allows serial but not irrevocable transactions; that is, serial transactions that can be rolled back. Such transactions require write barriers for undo logging.

The ABI allows the runtime to execute a transaction in either an STM mode (requiring read and write barrier instrumentation) or in a serial irrevocable mode (requiring no instrumentation), though the runtime should not select irrevocable mode for transactions that may have user-level abort or retry statements as this may result in a runtime error. This requires that the compiler generate two versions of the transactional code, one with the data access barrier functions, and one without. To support this we require that the compiler inform the runtime which versions of code have been generated as an argument to `startTransaction`, and have `startTransaction` return to the compiled code a result that tells it which version of the code should be executed. In this way we can maintain compatibility between compilers that don't generate both versions of the code and libraries that can't use both versions.

Although this interface allows the runtime to execute serial irrevocable transactions without instrumentation, it also does not preclude it from running such transaction with instrumentation (e.g., if

the compiler did not generate an un-instrumented version or if the runtime wants to profile memory accesses).

Although the choice of execution mode can only be made at the outermost transaction, this argument also passes other information into the library that may still be useful for nested transactions.

Most of the examples in this document will not include both versions of the code, since that simply makes them bigger with no worthwhile increase in pedagogic value.

3.8 Transaction descriptor

The interface assumes that most of the routines called will access a transaction descriptor that is opaque to the interface. This descriptor is returned by the `getTransaction` function and maintained by the compiled code to be passed back into subsequent function calls as an optimization. (Since it allows the relatively expensive lookup of the transaction descriptor to be performed only once per function, rather than once per TM library call).

It would clearly be possible to provide versions of each of the routines that did not take a transaction descriptor, but rather found it themselves and then called the underlying routine. This would save some generated code in the user program for routines that make few transactional memory accesses at the cost of approximately doubling the number of routines in the interface unless we can switch completely to a version without a visible transaction descriptor. Routines without a descriptor are not provided in this initial specification. Future versions of this ABI may remove the transaction descriptor argument if the lookup of the transaction descriptor becomes cheap enough (e.g., if we have dedicated slots in thread-local storage).

The difference between the two interfaces is shown in the two transformations of this example.[1]

```
int s;
int f(int);

void foo(void)
{
    atomic {
       int i;
       for (i=0;i<10;i++)
           s += f(i);
    }
}
```

Transformed code with transaction descriptor

```
int s;
int f(int);

void foo(void)
{
    void * td=getTransaction();
    initializeTransaction(td);
    startTransaction(td);
    {
       int i;
       for (i=0;i<10;i++)
       {
           int sval=
txnRead(td,&s);
           sval += f(i);
           txnWrite(td,&s,sval);
```

Transformed code without transaction descriptor

```
int s;
int f(int);

void foo(void)
{
    saveState();
    startTransaction();
    {
       int i;
       for (i=0;i<10;i++)
       {
           int sval= txnRead(&s);
           sval += f(i);
           txnWrite(&s,sval);
       }
    }
```

[1] The TM library function names in this example are merely illustrative. The actual names and interfaces to the routines will be detailed below. This example is merely illustrative of the two styles of interface.

```
        }
    }
    commitTransaction(td);
}
```

```
    commitTransaction();
}
```

The ABI specifies few data structures; however, there are various data structures pointers that are returned by one library call and passed to another. Rather than treating all such pointers as of type `void *` we use specific undefined `structs` (e.g. `struct foo;`) for each of the different types. This provides more type security across the interface as well as making the dataflow of these objects clearer in the specification.

3.9 Store allocation

The library internally allocates a single full transaction descriptor per thread at thread initialization time. This is the transaction descriptor returned by `getTransaction`. The first call to `getTransaction` will initialize the thread and allocate the transaction descriptor if the thread has not been initialized yet.

3.10 Naming conventions

We assume that the names of these functions must not conflict with the user symbol namespace. All of the library function names therefore start with the prefix `_ITM_` a single '_', followed by 'ITM' and a single '_') that moves these names out of the set names that user code is allowed to use according to the C standard. The `_ITM_` prefix may be omitted in discussions in this document for the sake of brevity, but should be implicitly read in. Note that the names given here are those used in C code, those may differ from the symbol names seen by the linker if the compiler adds additional prefixes or suffixes, or other wise mangles the name.

3.10.1 Read and write function names

We construct the names of the read and write functions according to the pseudo regexp

ITM[RW]{a[RW]} *type* | _ITM_RfW*type*

Name component	Interpretation
ITM	The fixed prefix '_ITM_' used on all our names.
[RW]	The character 'R' for a read and 'W' for a write.
RfW	The string "RfW" denoting a Read for Write operation
{a[RW]}	The optional strings 'aR' for the "after read" or 'aW' for the "after write" version of the function.
Type	Is a string to represent the type of the argument according to the table below.

Since in this initial version we are not supporting operations other than `mov` the `mov` suffix will be empty for all of the current routines.

Type encoding

Name component	Interpretation
U[1248]	Unsigned integer of size 1,2,4,8 bytes
[FDE]	Floating point value of size 4 (float), 8 (double) , or 16 (long double)
M64, M128, M256	__m64, __m128, or __m256 values

CF, CD,CE	complex composed of floats, complex composed of doubles, complex composed of long doubles

4 Types and macros list

```
/* Values used as arguments to abort. */
typedef enum {
    userAbort = 1,
    userRetry = 2,
    TMConflict= 4,
    exceptionBlockAbort = 8
} _ITM_abortReason;

/* Arguments to changeTransactionMode */
typedef enum
{
    modeSerialIrrevocable,
} _ITM_transactionState;

/* Results from inTransaction */
typedef enum
{
    outsideTransaction = 0,   /* So “if (inTransaction(td))” works */
    inRetryableTransaction,
    inIrrevocableTransaction
} _ITM_howExecuting;

// Values to describe properties of code, passed in to startTransaction
typedef enum
{
   pr_instrumentedCode   = 0x0001,
   pr_uninstrumentedCode = 0x0002,
   pr_multiwayCode = pr_instrumentedCode | pr_uninstrumentedCode,
   pr_hasNoXMMUpdate     = 0x0004,
   pr_hasNoAbort         = 0x0008,
   pr_hasNoRetry         = 0x0010,
   pr_hasNoIrrevocable   = 0x0020,
   pr_doesGoIrrevocable  = 0x0040,
   pr_hasNoSimpleReads   = 0x0080,
   pr_aWBarriersOmitted  = 0x0100,
   pr_RaRBarriersOmitted = 0x0200,
   pr_undoLogCode        = 0x0400,
   pr_preferUninstrumented=0x0800,
   pr_exceptionBlock     = 0x1000,
   pr_hasElse            = 0x2000
} _ITM_codeProperties;

// Result from startTransaction that describes what actions to take.
typedef enum
{
   a_runInstrumentedCode      = 0x01,
   a_runUninstrumentedCode    = 0x02,
   a_saveLiveVariables        = 0x04,
   a_restoreLiveVariables     = 0x08,
   a_abortTransaction         = 0x10,
} _ITM_actions;

typedef struct {} _ITM_transaction;  // Transaction descriptor
typedef uint32 _ITM_transactionId_t; // Transaction identifier
typedef struct
{
```

```
    uint32 reserved_1;
    uint32 flags;
    uint32 reserved_2;
    uint32 reserved_3;
    const char *psource;
} _ITM_srcLocation;
```

The _ITM_srcLocation structure is identical to the structure used for source location description in other places in the compiler (e.g. for OpenMP). Details can be found elsewhere. The psource field points to a string containing the source location generated in the compiler by code like this. This source location argument of all ABI functions that take it can be NULL.

```
sprintf(loc_name,";%s;%s;%d;%d;;", path_file_name, routine_name, sline, eline);
```

where sline is the first line of the construct and eline is the last line. If sline and eline are the same the construct is all on one line.

Typedefs for pointers to user functions that are used in the user interface to commit and abort/retry.

```
typedef void (* _ITM_userUndoFunction)(void *);
typedef _ITM_userUndoFunction _ITM_userCommitFunction;
```

The value used to identify being outside a transaction in the user undo logging code.

```
#define _ITM_noTransactionId 1  // Id for non-transactional code.
```

5 Function list

We use the `<stdint.h>` names for types in our prototypes.

5.1 Initialization and finalization functions

We document the initialization and finalization functions here since they need to exist in the library and need to be called by some part of the runtime. We do not expect the compiler to insert calls to any of these functions. The `getTransaction` function must initialize the thread if necessary.

We provide two sets of initialization and finalization functions. One set to initialize the library within a process, and a second set to initialize an individual thread. Since we're not expecting to replace the operating system's thread creation mechanisms, but cannot afford to pay the cost of testing whether the calling thread has been initialized on each library call we're driven to have a thread initialization call.

```
int  _ITM_initializeProcess(void);        /* Idempotent */
void _ITM_finalizeProcess(void);
int  _ITM_initializeThread(void);         /* Idempotent */
void _ITM_finalizeThread(void);
```

Process and thread initialization are idempotent, so redundant calls to these routines are merely a performance issue, not one of correctness.

The call to `initializeProcess` also initializes the thread in which it was made, so there is no need also to call `initializeThread` in that thread.

The return value for the initialization routines is zero for success, non-zero (an error code) for failure.

Before a thread can make other calls to the library it must have called the thread initialization routine at least once. Before a thread terminates it should call the thread finalization routine to allow the library to release any store associated with the thread.

How these routines get called depends on the way in which the threads are started, which is outside the scope of the TM ABI. It may be that they are called by wrapper routines in an adaptor library so that (for instance) user pthread creation calls are intercepted and modified to include the thread initialization call.

Since it is called just before a process exits the `finalizeProcess` function may appear redundant. However it is useful for the library to be called at this point to allow it to dump final performance statistics.

5.2 Version checking

We provide a macro that expands to the version of the header file with which the code was compiled, and a function in the library to check the compatibility of the version string. This allows us to detect dynamic linkage mismatches, and isolates knowledge of compatibility inside the library.

```
#define _ITM_VERSION "0.90 (Feb 29 2008)"
int _ITM_versionCompatible (int);
```

In addition we provide a macro that can be used for conditional compilation depending on the version of the interface being supported, which contains the version number multiplied by 100.

```
#define _ITM_VERSION_NO 90
```

We also provide a function to return the version string from the library. This allows the user code to print an error message of some sort if the version check fails, or just to know what library version it is using.

```
const char * _ITM_libraryVersion (void);
```

5.3 Error reporting

Under some circumstances the compiler can detect run time errors. We therefore provide an error reporting routine.

```
void _ITM_NORETURN _ITM_error (const _ITM_srcLocation *, int
errorCode);
```

We expect all errors detected by the compiler to be fatal, so this routine never returns. The values of `errorCode` are still to be determined.

5.4 inTransaction call

To support code that may run either inside or outside a transaction we provide the call

```
_ITM_howExecuting _ITM_inTransaction (_ITM_transaction *);
```

This returns a value from the _ITM_howExecuting enum. However the result can also be treated as a boolean, since the value zero in that enum corresponds to outsideTransaction. Therefore code like this

```
if (inTransaction(td)) {…}
```

executes as expected,

The additional information in the enum can be used by wrapper functions to determine whether they need to be concerned with the possibility of abort and retry, or not.

5.5 State manipulation functions

Most of the ABI functions require a transaction descriptor, an internal data structure of type `_ITM_transaction` that tracks the state of a thread's transaction. The ABI provides a routine for getting the transaction descriptor:

```
_ITM_transaction * _ITM_getTransaction(void);
```

A transaction descriptor is allocated for each thread by the thread initialization function or by the first call to the `getTransaction` function.

Each newly started transaction (including nested ones) gets a unique transaction identifier of type _ITM_transactionId_t. The ABI provides a routine for getting the current transaction's identifier:

```
_ITM_transactionId_t _ITM_getTransactionId(_ITM_transaction *);
```

The `getTransactionId` function returns a sequence number for the current transaction. Within a transaction, nested transactions are numbered sequentially in the order in which they start, with the outermost transaction getting the lowest number, and non-transactional code the value `_ITM_NoTransactionId`, which is less than any transaction id for transactional code.

5.6 Source locations

Several of the ABI functions take a source location argument (type `_ITM_srcLocation*`). This is useful for profiling versions of the library, where knowing the source location of the ABI call allows more user-friendly information to be produced. Source location arguments are optional, so the compiler may pass NULL as a source location. We could consistently omit the source location argument if we favor performance at all costs.

5.7 Starting a transaction

A transaction is started by calling

```
uint32 _ITM_beginTransaction(_ITM_transaction*,
                             uint32, _ITM_srcLocation*);
```

This function saves the machine state (callee saves registers, stack pointer) if necessary, and starts the transaction. The second (properties) argument is constructed by or-ing together a description of properties of the transactional code generated by the compiler and properties that the compiler has been able to deduce. The third argument is the source location where the atomic block begins.

This function behaves like setjmp in that it can return more than once, since every time that the transaction aborts internally (e.g., as a result of an explicit user abort or retry, or as a result of contention) the function will return. The result of the function is constructed by or-ing together a set of bit values that give the compiler information about what code should be executed next.

5.7.1 Transaction code properties

The input attributes describe information that the compiler has available and the kind of code that it has generated.

Attribute	Value	Meaning
`instrumentedCode`	0x0001	The compiler has generated code with read and write barriers for this transaction.
`uninstrumentedCode`	0x0002	The compiler has generated un-instrumented code (i.e., code with no read and write barriers) for this transaction.
`multiwayCode`	0x0003	The compiler has generated both instrumented and un-instrumented code for this transaction (convenience name).
`hasNoXMMUpdate`	0x0004	This transaction does not update the callee-saved XMM registers in its dynamic scope.
`hasNoAbort`	0x0008	This transaction has no __tm_abort in its lexical scope
`hasNoRetry`	0x0010	This transaction has no __tm_retry in its dynamic scope
`hasNoIrrevocable`	0x0020	This transaction does not enter serialIrrevocable mode in its dynamic scope.

doesGoIrrevocable	0x0040	This transaction enters serialIrrevocable mode on all its execution paths.
hasNoSimpleReads	0x0080	This transaction contains no “simple” read functions (_ITM_R<type>) in its dynamic scope.
awBarriersOmitted	0x0100	The compiler has omitted some "after write" barriers in the dynamic scope of this transaction.
RaRBarriersOmitted	0x0200	The compiler has omitted some “read after read” barriers in the dynamic scope of this transaction.
undoLogCode	0x0400	The compiler has generated only undo-logging barriers with no other barriers in the dynamic scope of this transaction.
preferUninstrumented	0x0800	The compiler thinks this transaction is best run with un-instrumented code.
exceptionBlock	0x1000	The compiler synthesized this nested transaction for a catch block or a destructor call during exception unwinding.
hasElse	0x2000	This atomic block has an else clause associated with it.

The compiler communicates properties of a transaction’s generated code to the runtime via these flags. The flags pass to the runtime information that the compiler has collected about the transaction being started. Some of the flags pertain to the lexical scope of the transaction (e.g., `hasNoAbort` and `hasElse`), which the compiler can deduce by analyzing the atomic block intra-procedurally. Other properties pertain to the dynamic scope of the transaction, which the compiler must deduce either inter-procedurally or intra-procedurally for transactions that have no function calls. The compiler may be able to deduce some of the inter-procedural properties using function annotations (such as annotations declaring that a function does not execute irrevocable actions). With the exception of the `preferUninstrumented` property (which is a compiler hint) the compiler should treat these properties as assertions on the generated code and should set these properties correctly; otherwise execution behavior is undefined.

At least one of `instrumentedCode` and `uninstrumentedCode` must be present (if only un-instrumented code is available the transaction will have to run irrevocably).

`hasNoXMMUpdate` should be set by the compiler if it can prove that no modifications of callee-saved XMM registers can occur in the dynamic scope of the transaction. This knowledge allows the runtime to avoid saving callee-saved XMM registers on operating systems where such registers exist.

`hasNoAbort` and `hasNoRetry` allow an optimization in some runtimes, since if neither of these statements is present undo-logging may not be required. The compiler can easily determine the `hasNoAbort` property because the `__tm_abort` statement is restricted to the lexical scope of a transaction. Determining the `hasNoRetry` property requires inter-procedural analysis as this property pertains to the dynamic scope of a transaction.

The compiler should assert `hasNoIrrevocable` if it can prove that the transaction cannot call `changeTransactionMode(…, modeSerialIrrevocable, …)` in its dynamic scope.

The compiler should assert `doesGoIrrevocable` if it can prove that every execution path through the transaction will require a switch to serialIrrevocable mode (e.g., a simple transaction with no conditional behavior that calls an un-annotated function will always switch to irrevocable).

The compiler should assert `hasNoSimpleReads` if it has generated no calls to “simple” read functions inside the dynamic scope of the transaction. (Calls to RfW do not count as simple reads, only R functions matter.) Knowing that the function has no reads allows optimization of some runtime algorithms.

The additional flags (`aWBarriersOmitted, RaRBarriersOmitted, undoLogCode`) are used if the compiler generates code that omits some barriers. Code generated without some barriers has specific requirements of the runtime (e.g., code without "after write" barriers requires an in-place update STM, and code without "read after read" barriers a pessimistic readers STM), therefore the runtime needs to know what barriers have been omitted.

The `preferUninstrumented` flag is a hint from the compiler that this transaction might be best run without instrumentation (i.e., in serial irrevocable model).

The `exceptionBlock` flag asserts that the transaction is one that the compiler synthesized for exception blocks. Aborts inside of exceptionBlock transactions are handled differently than non-exceptionBlocks as we describe later. If the outermost or innermost nested transaction is in an aborted state (as described in Section 5.8) then any call to a nested `beginTransaction` for an ExceptionBlock transaction immediately returns `a_abortTransaction`.

The `hasElse` flag asserts that the atomic block for the current transaction has an else block associated with it. This flag currently does not affect the behavior of the runtime but future versions of this ABI may define special semantics for transactions with else blocks (e.g., a user retry may cause control to transfer to the else block). The compiler may eliminate an else block if it can prove the `hasNoAbort`, `hasNoRetry`, and `hasNoIrrevocable` properties.

The remaining bits are reserved and should not be set by the compiler.

It is clearly possible to add more properties here if the compiler can provide additional useful information to the runtime (for instance it may be useful to know that the compiler cannot bound the number of reads/writes executed by the un-instrumented code because it contains calls or loops). Future versions of this ABI may define additional properties.

5.7.2 Windows exception state

On the Windows 32 bit platform (and possibly also in 64 bits), it is necessary to save and restore the value at `fs:__Except_list`. This should **always** be done, since the library (and, in general the compiler) cannot determine whether exception handling occurs inside the transaction. Doing this is always harmless, and testing whether to do it is more expensive than just doing it.

If the compiler is using `[ebp-4]` to hold the exception index for the current scope, then it must ensure that it always restores this value on return from any of the `beginTransaction` functions. It is more efficient to have the compiler generate this code than to have the runtime save and restore the value, since it is not safe for the runtime always to do so, and therefore the compiler would have to pass in a flag, and the runtime would have to test it, all of which is expensive compared with the single instruction that the compiler can generate in place.

Saving and restoring these values is necessary to handle the case where a transaction aborts and retries while executing inside an exception handler.

5.7.3 Other machine state

`beginTransaction` must save all callee saved registers the first time that it is called, and restore them in the abort/retry path. The `hasNoXMMUpdate` code property allows the compiler to optimize this save/restore.

5.7.4 Results from beginTransaction

The possible results of `beginTransaction` direct the compiled code to the variant of the code to execute; the possible values are

Mode	Value	Meaning

`runInstrumentedCode`	0x01	Execute the instrumented code
`runUninstrumentedCode`	0x02	Execute the un-instrumented code
`saveLiveVariables`	0x04	Save live in un-logged memory variables
`restoreLiveVariables`	0x08	Restore live in un-logged memory variables
`abortTransaction`	0x10	Abort the transaction

Although these flags allow many possible return values, the combinations detailed below are the **only ones that are supported**; if the library returns other values behavior is unspecified.

Transaction entry actions	
Action	**Result**
`Start a serial irrevocable transaction`	`runUninstrumentedCode`
`Start a transaction using instrumented code`	`saveLiveVariables | runInstrumentedCode`
`Restart a transaction using instrumented code`	`restoreLiveVariables | runInstrumentedCode`
`Restart a transaction as a serial irrevocable transaction (mode switch)`	`restoreLiveVariables | runUninstrumentedCode`
`Start an irrevocable transaction using instrumented code`	`runInstrumentedCode`

When `beginTransaction` returns one of the values above the new transaction has been entered, so conflicts will be reported inside that transaction.

Transaction exit actions	
Action	**Result**
`Abort transaction`	`restoreLiveVariables | abortTransaction`

When `beginTransaction` returns the value above the transaction has been exited, so conflicts will no longer be reported inside that transaction, and if it was the outermost transaction, the program is executing outside any transaction.

The saveLiveVariables and restoreLiveVariables flags allow the compiler to optimize the instrumented code generated for local variables. The observation here is that local variables that do not have their address taken cannot be visible to other threads, and therefore accesses to them do not require instrumentation (since only this thread can be accessing them). Their values must still be restored if the transaction aborts or retries, so the compiler must save copies of their values before the transaction starts and restore them when requested to by the runtime. Note that this optimization applies only to instrumented code (since the whole aim is to avoid generating instrumentation for accesses to these variables).

5.7.5 Nested transactions

The current ABI supports closed nested transactions; it does not support open nested transactions. A nested transaction is started by a call to `beginTransaction` from code that is already running in a transactional context. The compiler need generate only instrumented code if the nested transaction is being generated in instrumented code or if the nested transaction has either an abort statement or an else block. If a closed nested transaction has no lexically scoped user aborts and no else block, then the compiler may flatten it and remove the `beginTransaction` call.

5.7.6 Example: Generating code to start a transaction

The following illustrates code that a compiler could generate to start a transaction:

```
static _ITM_srcLocation start_loc = {…};

...

_ITM_transaction * td = _ITM_getTransaction();
Uint32 properties = pr_multiwayCode | ... ;
int doWhat = _ITM_beginTransaction (td,properties,&start_loc);
if (doWhat & a_restoreLiveVariables) {
    /* code to restore live local variables that aren't instrumented */
    ...
}
if (doWhat & a_abortTransaction) goto txn_abort_label;
if (doWhat & a_instrumentedCode) {
    if ((doWhat & a_saveLiveVariables)) {
    /* code to save live local variables that aren't instrumented. */
    ...
    }
    /* instrumented body of the transaction goes here */
    ...
} else {
    /* un-instrumented body of the transaction goes here */
    ...
}
txn_abort_label:
```

5.8 Aborting a transaction

A transaction can be rolled back by calling the abort or rollback transaction functions:

```
void _ITM_NORETURN _ITM_abortTransaction(_ITM_transaction *,
                                         _ITM_abortReason,
                                         const _ITM_srcLocation *);

void _ITM_rollbackTransaction(_ITM_transaction *,
                              const _ITM_srcLocation *);
```

The `rollbackTransaction` function rolls back the innermost transaction that is not an `exceptionBlock` transaction and puts that transaction into the aborted state. It does not end that transaction until that transaction is committed. If the innermost nested transaction is in an aborted state, then subsequent calls to `beginTransaction` will return immediately with the value `a_abortTransaction`; calls to other interface functions except for commit will result in undefined behavior.

The `abortTransaction` function long-jumps to a `beginTransaction` function, causing that `beginTransaction` call to return for a second time. The abort reason parameter determines both the target `beginTransaction` of the long-jump and the value returned by that `beginTransaction`:

1. If the innermost nested transaction has the `exceptionBlock` property set or if the reason is `userAbort`, then abort long jumps to the innermost `beginTransaction` call and that `beginTransaction` call returns the value (`a_abortTransaction | a_restoreLiveVariables`).
2. If the reason is `userRetry` or `TMConflict`, and there are no nested `exceptionBlock` transactions, then abort long jumps to the `beginTransaction` call of the outermost dynamic transaction. The `beginTransaction` call returns the value (`a_restoreLiveVariables | a_runInstrumentedCode`). On a `TMConflict`, the runtime may also return the value (`a_restoreLiveVariables | a_runUninstrumentedCode`) if it decides to restart the transaction in serial irrevocable mode (as a contention resolution policy).

3. If the reason is `userRetry` or `TMConflict`, and there are nested `exceptionBlock` transactions, then abort long jumps to the innermost `beginTransaction` call that has the `exceptionBlock` property set, and that `beginTransaction` call returns the value (`a_abortTransaction | a_restoreLiveVariables`).
4. If the reason is `exceptionBlockAbort`, then abort will behave as if the reason is the same as the last abort that did not have reason `exceptionBlockAbort` or as if the reason is `TMConflict` if the runtime aborted an exception block due to a conflict.

An abort reason of `userRetry` or `TMConflict` puts the outermost transaction into an aborted state until the `beginTransaction` of the outermost transaction returns (case 2 in the list above). If the outermost transaction is in an aborted state, then calls to `beginTransaction` will return immediately with the value `a_abortTransaction`, and calls to any subsequent interface functions other than commit will result in undefined behavior.

If an exception object has been registered for the transaction (via the `registerThrownObject` function), then abort and rollback do not roll back updates to the registered exception object.

If the target `beginTransaction` call belongs to an irrevocable transaction then this is viewed as an error and the program is aborted (presumably with some suitable error message).

5.9 Committing a transaction

A transaction can be committed by calling one of three commit functions:

```
void _ITM_commitTransaction (_ITM_transaction *,
                             const _ITM_srcLocation *);
bool _ITM_tryCommitTransaction (_ITM_transaction *,
                                const _ITM_srcLocation *);
void _ITM_commitTransactionToId (_ITM_transaction *,
                                 const _ITM_transactionId tid,
                                 const _ITM_srcLocation *);
```

The `commitTransaction` function may or may not return because it may decide that the transaction has conflicts and must retry. If the function returns, then the transaction has committed and execution continues outside the scope of the transaction. (Though for a nested transaction it is still possible that the containing function may abort or retry and undo the effects of the inner transaction, since we're dealing with closed nested transactions). If it decides to retry, it follows the same rules as defined in Section 5.8 (with reason `TMConflict`) for determining the target `beginTransaction` of the long jump and its return value. The `commitTransaction` function always returns without checking for conflicts if the innermost nested transaction is of type exceptionBlock.

The `tryCommitTransaction` function returns true if it successfully commits the transaction and false if it fails to commit (e.g., it detects a conflict). If the `tryCommitTransaction` function returns false, the transaction is in an inconsistent state and must be aborted via a call to `abortTransaction`.

The `commitTransactionToId` function commits all inner transactions nested within the transaction specified by the transaction id parameter (it does not commit the transaction specified by the transaction id). This function always returns to the caller without checking for conflicts. Its main purpose is to reset the current transaction id when an exception may have escaped the scope of a nested transaction without properly committing that transaction (e.g., if an exception propagates out of a nested atomic block in C code). The transaction id parameter must belong to an active transaction.

5.9.1 Example: Generating code to start and commit a transaction for C programs

The following illustrates code that a compiler could generate to start and commit a transaction in a C program:

```
static _ITM_srcLocation start_loc = {…};
static _ITM_srcLocation commit_loc = {…};
```

```
...

_ITM_transaction * td = _ITM_getTransaction();
Uint32 properties = pr_multiwayCode | ... ;
int doWhat = _ITM_beginTransaction (td,properties,&start_loc);
if (doWhat & a_restoreLiveVariables) {
    /* code to restore live local variables that aren't instrumented */
    ...
}
if (doWhat & a_abortTransaction) goto txn_abort_label;
if (doWhat & a_instrumentedCode) {
    if ((doWhat & a_saveLiveVariables)) {
    /* code to save live local variables that aren't instrumented. */
    ...
    }
    /* instrumented body of the transaction goes here */
    ...
} else {
    /* un-instrumented body of the transaction goes here */
    ...
}
_ITM_commitTransaction(td,&commit_loc);

txn_abort_label:
```

5.9.2 Example: Generating code to start and commit a transaction for C++ programs

The code sequence for starting and committing transactions C++ programs must also consider exceptions that propagate out of a transaction. The following illustrates code that a compiler could generate to start and commit a transaction in a C++ program:

```
static _ITM_srcLocation txn_start_loc = {…};
static _ITM_srcLocation txn_end_loc = {…};

...

_ITM_transaction * td = _ITM_getTransaction();
uint32 properties = pr_multiwayCode | ... ;
int doWhat = _ITM_beginTransaction (td,properties,&txn_start_loc);
if (doWhat & a_restoreLiveVariables) {
    /* code to restore live local variables that aren't instrumented */
    ...
}
if (doWhat & a_abortTransaction) goto txn_abort_label;
if ((doWhat & a_saveLiveVariables)) {
    /* code to save live local variables that aren't instrumented. */
    ...
}
_ITM_transactionId cur_id = _ITM_getTransactionId();
bool catch_aborted = false;
try {
    if (doWhat & a_instrumentedCode) {
        /* instrumented body of the transaction goes here */
        ...
    } else {
        /* un-instrumented body of the transaction goes here */
        ...
    }
} catch (...) {
    /* uncaught exception in transaction */
    /* reset the current transaction id in case the exception did not
       commit a nested transaction */
    _ITM_commitTransactionToId(td,cur_id,&txn_end_loc);
    if (_ITM_tryCommitTransaction(td,&txn_end_loc))
        throw; /* rethrow exception */
    /* commit failed due to conflict */
    catch_aborted = true;
}
if (catch_aborted)
```

```
    _ITM_abortTransaction(td,TMConflict,&txn_end_loc);
_ITM_commitTransaction(td,&txn_end_loc);
txn_abort_label:
```

5.9.3 Example: Generating code for instrumented try-catch blocks

Consider a try-catch block inside a transaction:

```
try {
    S1
} catch (…) { /* could also be catch of a specific exception type */
    S2
}
```

The following illustrates code that the compiler could generate for this try-catch block:

```
_ITM_transaction * td = _ITM_getTransaction();
bool catch_aborted = false;
bool catch_executed = false;
_ITM_transactionId cur_id = _ITM_getTransactionId();
try {
    /* instrumented code for S1 goes here */
    ...
} catch (…) {
    _ITM_commitTransactionToId(td,cur_id,&catch_start_loc);
    catch_executed = true;
    uint32 properties = pr_instrumentedCode | pr_exceptionBlock | ... ;
    int doWhat = _ITM_beginTransaction (td,properties,&catch_start_loc);
    if (doWhat & a_restoreLiveVariables) {
        /* code to restore live local variables that aren't instrumented */
        ...
    }
    if (!(doWhat & a_abortTransaction)) {
        if (doWhat & a_saveLiveVariables) {
            /* code to save live local variables that aren't instrumented */
            ...
        }
        /* instrumented code for S2 goes here */
        ...
    } else {
        catch_aborted = true;
    }
}
if (catch_aborted)
    _ITM_abortTransaction(td,exceptionBlockAbort,&catch_end_loc);
if (catch_executed)
    _ITM_commitTransaction(td,&catch_end_loc);
```

Note that the transaction synthesized for the catch block is committed after the catch block (i.e., after the C++ runtime has finished handling the exception). In practice, control flow may exit the catch block via other control flow statements, namely a return, break, continue, or goto. The compiler must generate the commit transaction call at the landing pads of such statements. Note also that any exception thrown by S2 will propagate out of the catch block without committing the catch block's transaction; instead, the try block that catches the exception or the catch block generated for the outermost transaction will properly commit the catch block transaction using `commitTransactionToId`.

We introduce the `catch_aborted, catch_executed` , and similar variables in our examples only to illustrate the logic in plain C++ source code. In practice, we expect the compiler to generate optimized control flow without the use of these variables.

5.10 Exception handling support

To support abort-on-exception semantics, the compiler needs to register thrown objects with the STM runtime so that the runtime can then avoid rolling back their values when they propagate out of an abort transaction.

```
void _ITM_registerThrownObject(_ITM_transaction *,
                               const void *exception_object, size_t);
```

The compiler calls the register function every time a new object that is about to be thrown is constructed. The second parameter points to the about-to-be-thrown exception object, and the third parameter specifies the size of that object. This function registers the object with the innermost non-exceptionBlock transaction. A rollback of that transaction will not roll back the registered object.

5.10.1 Example: Generating code for abort and throw

Consider the following atomic block that aborts and throws an exception:

```
__tm_atomic {
   X x();
   S1
   if (...) __tm_abort throw x;
}
```

The compiler could generate the following code:

```
_ITM_transaction * td = _ITM_getTransaction();
uint32 properties = pr_instrumentedCode | ... ;
int doWhat = _ITM_beginTransaction (td,properties,&start_loc);
if (doWhat & a_restoreLiveVariables) {
    /* code to restore live local variables that aren't instrumented */
    ...
}
if (doWhat & a_abortTransaction) goto txn_abort_label;
if ((doWhat & a_saveLiveVariables)) {
    /* code to save live local variables that aren't instrumented. */
    ...
}
_ITM_transactionId cur_id = _ITM_getTransactionId();
bool catch_aborted = false;
try {
    X x(); /* instrumented version of constructor */
    /* instrumented version of S1 goes here */
    . . .

    if (...) {
       /* code to restore live local variables that aren't instrumented */
        ...
        _ITM_registerThrownObject(td,&x,sizeof(X));
        _ITM_rollbackTransaction(td,&abort_loc);
       throw x;
    }
} catch (...) {
    /* uncaught exception in transaction */
    /* reset the current transaction id in case the exception did not
       commit a nested transaction */
    _ITM_commitTransactionToId(td,cur_id,&commit_loc);
    if (_ITM_tryCommitTransaction(td,&commit_loc))
        throw; /* rethrow exception */
    /* commit failed due to conflict */
    catch_aborted = true;
}
if (catch_aborted)
    _ITM_abortTransaction(td,TMConflict,NULL);
_ITM_commitTransaction(td,&commit_loc);

txn_abort_label:
```

5.11 Transition to serial irrevocable mode

In serial irrevocable mode the runtime maintains no undo logs, so it is impossible to abort or retry, and no other transaction can execute at the same time as the serial irrevocable transaction. Because of these restrictions un-instrumented code can be executed in this mode as can system calls whose effects cannot be undone, and calls to non-transactional functions whose side-effects are unknown.

Support for this mode is mandatory, since it is required to execute arbitrary un-instrumented code that may contain non-transaction safe function calls.

Transition to an irrevocable mode is made by calling

```
void _ITM_changeTransactionMode (_ITM_transaction *,
                                 _ITM_transactionState,
                                 const _ITM_srcLocation *);
```

`changeTransactionMode` may or may not return, since to make the mode transition it may be necessary to abort the transaction and restart it in the new mode. Since changing transaction mode may be a costly operation (requiring an abort and retry of the transaction in the new mode), we pass a source location so that the runtime has the opportunity to help the user find the places where the compiler has forced the transaction to change mode.

Individual library implementations can use additional modes internally without needing them to be in this ABI. In the future, it may be useful to add more options to this enumeration that allow codes (at least test codes) to request explicit transitions (e.g., to different STM modes).

5.12 Data transfer functions

For completeness, the full set of data transfer functions implied above are enumerated below. The fastcall attribute is omitted here for simplicity, but is present in the header file when compiling for an appropriate architecture.

In the header file these prototypes are generated using a macro, so the whole set is defined in 23 lines of code, and it is easy to add new types, or the special SSE transfer functions.

```
uint8 _ITM_RU1 (_ITM_transaction *, const uint8 *);
uint8 _ITM_RaRU1 (_ITM_transaction *, const uint8 *);
uint8 _ITM_RaWU1 (_ITM_transaction *, const uint8 *);
uint8  _ITM_RfWU1 (_ITM_transaction *, const uint8 *);
void _ITM_WU1 (_ITM_transaction *, uint8 *, uint8);
void _ITM_WaRU1 (_ITM_transaction *, uint8 *, uint8);
void _ITM_WaWU1 (_ITM_transaction *, uint8 *, uint8);

uint16 _ITM_RU2 (_ITM_transaction *, const uint16 *);
uint16 _ITM_RaRU2 (_ITM_transaction *, const uint16 *);
uint16 _ITM_RaWU2 (_ITM_transaction *, const uint16 *);
uint16  _ITM_RfWU2 (_ITM_transaction *, const uint16 *);
void _ITM_WU2 (_ITM_transaction *, uint16 *, uint16);
void _ITM_WaRU2 (_ITM_transaction *, uint16 *, uint16);
void _ITM_WaWU2 (_ITM_transaction *, uint16 *, uint16);

uint32 _ITM_RU4 (_ITM_transaction *, const uint32 *);
uint32 _ITM_RaRU4 (_ITM_transaction *, const uint32 *);
uint32 _ITM_RaWU4 (_ITM_transaction *, const uint32 *);
uint32  _ITM_RfWU4 (_ITM_transaction *, const uint32 *);
void _ITM_WU4 (_ITM_transaction *, uint32 *, uint32);
void _ITM_WaRU4 (_ITM_transaction *, uint32 *, uint32);
void _ITM_WaWU4 (_ITM_transaction *, uint32 *, uint32);

uint64 _ITM_RU8 (_ITM_transaction *, const uint64 *);
uint64 _ITM_RaRU8 (_ITM_transaction *, const uint64 *);
uint64 _ITM_RaWU8 (_ITM_transaction *, const uint64 *);
uint64  _ITM_RfWU8 (_ITM_transaction *, const uint64 *);
void _ITM_WU8 (_ITM_transaction *, uint64 *, uint64);
void _ITM_WaRU8 (_ITM_transaction *, uint64 *, uint64);
void _ITM_WaWU8 (_ITM_transaction *, uint64 *, uint64);

float _ITM_RF (_ITM_transaction *, const float *);
float _ITM_RaRF (_ITM_transaction *, const float *);
float _ITM_RaWF (_ITM_transaction *, const float *);
float  _ITM_RfWF (_ITM_transaction *, const float *);
void _ITM_WF (_ITM_transaction *, float *, float);
void _ITM_WaRF (_ITM_transaction *, float *, float);
void _ITM_WaWF (_ITM_transaction *, float *, float);
```

```
double _ITM_RD (_ITM_transaction *, const double *);
double _ITM_RaRD (_ITM_transaction *, const double *);
double _ITM_RaWD (_ITM_transaction *, const double *);
double  _ITM_RfWD (_ITM_transaction *, const double *);
void _ITM_WD (_ITM_transaction *, double *, double);
void _ITM_WaRD (_ITM_transaction *, double *, double);
void _ITM_WaWD (_ITM_transaction *, double *, double);

long double _ITM_RE (_ITM_transaction *, const long double *);
long double _ITM_RaRE (_ITM_transaction *, const long double *);
long double _ITM_RaWE (_ITM_transaction *, const long double *);
long double  _ITM_RfWE (_ITM_transaction *, const long double *);
void _ITM_WE (_ITM_transaction *, long double *, long double);
void _ITM_WaRE (_ITM_transaction *, long double *, long double);
void _ITM_WaWE (_ITM_transaction *, long double *, long double);

__m64 _ITM_RM64 (_ITM_transaction *, const __m64 *);
__m64 _ITM_RaRM64 (_ITM_transaction *, const __m64 *);
__m64 _ITM_RaWM64 (_ITM_transaction *, const __m64 *);
__m64  _ITM_RfWM64 (_ITM_transaction *, const __m64 *);
void _ITM_WM64 (_ITM_transaction *, __m64 *, __m64);
void _ITM_WaRM64 (_ITM_transaction *, __m64 *, __m64);
void _ITM_WaWM64 (_ITM_transaction *, __m64 *, __m64);

__m128 _ITM_RM128 (_ITM_transaction *, const __m128 *);
__m128 _ITM_RaRM128 (_ITM_transaction *, const __m128 *);
__m128 _ITM_RaWM128 (_ITM_transaction *, const __m128 *);
__m128  _ITM_RfWM128 (_ITM_transaction *, const __m128 *);
void _ITM_WM128 (_ITM_transaction *, __m128 *, __m128);
void _ITM_WaRM128 (_ITM_transaction *, __m128 *, __m128);
void _ITM_WaWM128 (_ITM_transaction *, __m128 *, __m128);

__m256 _ITM_RM256 (_ITM_transaction *, const __m256 *);
__m256 _ITM_RaRM256 (_ITM_transaction *, const __m256 *);
__m256 _ITM_RaWM256 (_ITM_transaction *, const __m256 *);
__m256  _ITM_RfWM256 (_ITM_transaction *, const __m256 *);
void _ITM_WM256 (_ITM_transaction *, __m256 *, __m256);
void _ITM_WaRM256 (_ITM_transaction *, __m256 *, __m256);
void _ITM_WaWM256 (_ITM_transaction *, __m256 *, __m256);

float _Complex _ITM_RCF (_ITM_transaction *, const float _Complex *);
float _Complex _ITM_RaRCF (_ITM_transaction *, const float _Complex *);
float _Complex _ITM_RaWCF (_ITM_transaction *, const float _Complex *);
float _Complex  _ITM_RfWCF (_ITM_transaction *, const float _Complex *);
void _ITM_WCF (_ITM_transaction *, float _Complex *, float _Complex);
void _ITM_WaRCF (_ITM_transaction *, float _Complex *, float _Complex);
void _ITM_WaWCF (_ITM_transaction *, float _Complex *, float _Complex);

double _Complex _ITM_RCD (_ITM_transaction *, const double _Complex *);
double _Complex _ITM_RaRCD (_ITM_transaction *, const double _Complex *);
double _Complex _ITM_RaWCD (_ITM_transaction *, const double _Complex *);
double _Complex  _ITM_RfWCD (_ITM_transaction *, const double _Complex *);
void _ITM_WCD (_ITM_transaction *, double _Complex *, double _Complex);
void _ITM_WaRCD (_ITM_transaction *, double _Complex *, double _Complex);
void _ITM_WaWCD (_ITM_transaction *, double _Complex *, double _Complex);

long double _Complex _ITM_RCE (_ITM_transaction *,
    const long double _Complex *);
long double _Complex _ITM_RaRCE (_ITM_transaction *,
      const long double _Complex *);
long double _Complex _ITM_RaWCE (_ITM_transaction *,
                                 const long double _Complex *);
long double _Complex  _ITM_RfWCE (_ITM_transaction *,
                                  const long double_Complex *);

void _ITM_WCE (_ITM_transaction *, long double _Complex *,
  long double _Complex);
void _ITM_WaRCE (_ITM_transaction *, long double _Complex *,
          long double _Complex);
void _ITM_WaWCE (_ITM_transaction *, long double _Complex *,
                 long double _Complex);
```

5.13 Transactional memory copies

While it would be possible to add transactional versions of all of the functions in `<string.h>` to the library (and, indeed it is likely that user codes will need such functions), our intention in this specification is to provide the functions required by the compiler. We hope that we can design an interface at this level that allows another library to provide functions such as `memset`, without requiring that they be implemented in this library.

For aggregate assignment the compiler may need versions of `memcpy` that treat either or both of the arguments transactionally. When coupled with the "after read" and "after write" properties this leads to fifteen new functions, generated according to the regexp

```
Void _ITM_memcpyR[nt]{,aR,aW}W[nt]{,aR,aW} (_ITM_transaction *td,
                                            void * target,
                                            const void * source,
                                            size_t count);
```

where

'n' means non-transactional

't' means transactional

'aR,'aW' mean after read and after write (as usual), and only used on transactional arguments.

Here is the full list of functions

```
void _ITM_memcpyRnWt   (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memcpyRnWtaR (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memcpyRnWtaW (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memcpyRtWn   (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memcpyRtWt   (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memcpyRtWtaR (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memcpyRtWtaW (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memcpyRtaRWn (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memcpyRtaRWt (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memcpyRtaRWtaR (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memcpyRtaRWtaW (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memcpyRtaWWn (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memcpyRtaWWt (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memcpyRtaWWtaR (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memcpyRtaWWtaW (_ITM_transaction *, void *, const void *, size_t);
```

We do not proved RfW versions of these functions, the increase in library complexity is not justified by the rarity of usage and small performance gains that might be possible,

5.14 Transactional versions of memmove

Similar to `memcpy`, we provide transactional variants of `memmove` for aggregate assignment where the destination and source locations overlap.

```
void _ITM_memmoveRnWt   (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memmoveRnWtaR (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memmoveRnWtaW (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memmoveRtWn   (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memmoveRtWt   (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memmoveRtWtaR (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memmoveRtWtaW (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memmoveRtaRWn (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memmoveRtaRWt (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memmoveRtaRWtaR (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memmoveRtaRWtaW (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memmoveRtaWWn (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memmoveRtaWWt (_ITM_transaction *, void *, const void *, size_t);
```

```
void _ITM_memmoveRtaWWtaR (_ITM_transaction *, void *, const void *, size_t);
void _ITM_memmoveRtaWWtaW (_ITM_transaction *, void *, const void *, size_t);
```

5.15 Transactional versions of memset

To support the use of `memset` in transactions we provide transactional versions of `memset`,.

```
void _ITM_memsetW (_ITM_transaction *td,
                               void * target,
                               int src,
                               size_t count);
void _ITM_memsetWaR (_ITM_transaction *td,
                               void * target,
                               int src,
                               size_t count);
void _ITM_memsetWaW (_ITM_transaction *td,
                               void * target,
                               int src,
                               size_t count);
```

The qualification (W,WaR,WaW) should be taken as the least restrictive qualification of any of the bytes that may be written. So, for example it we have code like

```
_ITM_memsetW(td,foo,0,10);
_ITM_memsetW(td,foo,1,20);
```

the second call is correctly `memsetW`, not `memsetWaW`, since some of the bytes written in the second call were not written by the first one.

5.16 Logging functions

The ABI provides a set of logging functions of the form

```
void _ITM_L type (_ITM_transaction *, const type *);
```

where *type* is any of the normal basic types.

The ABI also provides

```
void _ITM_LB (_ITM_transaction *,  const void *, size_t);
```

to allow the compiler to log arbitrarily sized chunks of store.

```
void _ITM_LU1 (_ITM_transaction *, const uint8 *);
void _ITM_LU2 (_ITM_transaction *, const uint16 *);
void _ITM_LU4 (_ITM_transaction *, const uint32 *);
void _ITM_LU8 (_ITM_transaction *, const uint64 *);
void _ITM_LF (_ITM_transaction *, const float *);
void _ITM_LD (_ITM_transaction *, const double *);
void _ITM_LE (_ITM_transaction *, const long double *);
void _ITM_LM64 (_ITM_transaction *, const __m64 *);
void _ITM_LM128 (_ITM_transaction *, const __m128 *);
void _ITM_LM256 (_ITM_transaction *, const __m256 *);
void _ITM_LCF (_ITM_transaction *, const float _Complex *);
void _ITM_LCD (_ITM_transaction *, const double _Complex *);
void _ITM_LCE (_ITM_transaction *, const long double _Complex *);
void _ITM_LB (_ITM_transaction *, const void*, size_t);
```

5.17 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 operations to be performed on transaction abort/retry and commit. These functions are not expected to be used by the compiler, so changes to this interface should not require corresponding compiler changes.

The idea here is that the `tm_wrapping` annotation allows user functions to be executed non-transactionally from inside a transaction. Such functions may very well be part of a subsystem that has internal state that (since the code is non-transactional) is not being tracked by the TM system. Therefore we need a mechanism to allow such a subsystem to ensure that these transactional state changes are reverted on transaction abort/retry and committed on transaction commit.

5.17.1 User commit and undo functions

To allow user code to be called at commit time to finalize ephemeral state, and at the time a transaction is being aborted to purge ephemeral state, we provide a way to register callback functions that will be called by the STM runtime. The `transactionId` of a transaction is used to determine when user registered commit and validate actions are invoked (see the discussion below).

5.17.1.1 Commit function

The addUserCommitAction function registers a commit callback:

```
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. The following example illustrates this:

```
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.

5.17.1.2 Undo action

A user undo action is registered by calling the function

```
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.

5.17.2 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

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

5.17.3 Remove internal library references

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.

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

6 Future enhancements to the ABI

Future versions of the ABI may make the following enhancements:

- **Define name mangling and indirect function call compiler conventions**. The Intel compiler uses a name mangling convention for the transactional (i.e., instrumented) clone of a function and uses a runtime technique to find the clone on an indirect call inside of a transaction. To allow compilers to generate inter-operable binaries, future versions of this ABI may define these conventions or provide an interface to lookup the transactional clone of a function.
- **Eliminate the transaction descriptor parameter**. The current ABI defines an explicit transaction descriptor parameter in most of the functions to avoid repeating expensive accesses to get the descriptor. The runtime stores the descriptor in thread local storage (TLS). Accessing TLS takes several memory accesses in the current versions of our target operating systems. We expect the STM runtime to be deployed as a dynamically linked library, which makes access to its TLS variables slower than those in statically linked libraries. Dedicated TLS slots accessible directly (e.g., via the segment registers) would eliminate this overhead. Future versions of the ABI may eliminate most if not all of the transaction descriptor parameters or allow the compiler to inline the `getTransaction` function call if such dedicated TLS slots were made available for TM (and thus defined as part of the OS ABI). Eliminating the transaction descriptor may also require support from the C++ runtime system

to initialize a thread's transaction descriptor to eliminate the need to lazily initialize the runtime on the first call to `getTransaction`.

- **Eliminate the call to getTransactionId at the start of a try block**. Future versions of this ABI may instead use the stack or frame pointer of the try-catch block to set the current transaction, eliminating the need for any call into the runtime when starting a try block.
- **Provide compiler helper functions for generating exception handling code**. Much of the code generated to handle exceptions inside transactions – such as code to start, commit, and abort transactions synthesized for catch blocks (see the examples in Sections 5.9.2 and 5.9.3) – can be factored out into compiler helper functions or even made part of the C++ runtime's exception handling code. This would greatly cleanup the code generated for handling exceptions inside transactions. Future versions of this ABI may provide such helper functions or may provide better integration with the C++ runtime.
- **Use lambdas to eliminate boilerplate code in C++ atomic blocks**. The code to start, commit, and handle uncaught exceptions in a C++ atomic block (see the example in Sections 5.9.2) can be easily hidden inside the STM runtime by using lambdas in the upcoming C++ standard. The idea would be to define a new ABI function that executes a lambda expression inside a transaction. The lambda expression would contain the body of the transaction:

```
[&] (int doWhat) {
    if (doWhat & a_restoreLiveVariables) {
        /* code to restore live local variables that aren't instrumented */
        ...
    }
    if (doWhat & a_abortTransaction)
        return;
    if (doWhat & a_instrumentedCode) {
        if ((doWhat & a_saveLiveVariables)) {
        /* code to save live local variables that aren't instrumented. */
            ...
        }
        /* instrumented body of the transaction goes here */
        ...
    } else {
        /* un-instrumented body of the transaction goes here */
        ...
    }
}
```

7 Sample code

These (handwritten, and therefore likely wrong) examples demonstrates the expected usage of the routines. These are for illustrative purposes only; we expect the compiler to generate code more optimized that these sequences (e.g., use control flow rather than set flags).

7.1 Initialization

Simple library initialization and version checking will look something like this.

```
if (!_ITM_versionCompatible (_ITM_VERSION_NO))
{
      fprintf(stderr,"TM library versions incompatible, compiled with
      %s"
                    "linked with %s\n",
                    __ITM_VERSION, _ITM_libraryVersion());
      exit (-1);
}

if (!_ITM_initalizeProcess())
{
```

```
      fprintf (stderr,"TM library initialization failed\n");
      exit (-1);
}
```

The compiler would no doubt embed such code in its normal runtime startup.

7.2 Atomic block (C code)

Consider a (probably poorly written) example like this.

```
int s;
int __declspec (tm_callable) f(int);

void foo(void)
{
    int i;
    for (i=0;i<10;i++)
    {
      atomic {
           s += f(i);
           if (s > 1000) __tm_abort;
      }
    }
}
```

We expect it to be transformed to something like this,

```
int s;
int f(int);
static _ITM_srcLocation start_outer_loc = {…};
static _ITM_srcLocation commit_outer_loc = {…};
static _ITM_srcLocation abort_loc = {…};

void foo(void)
{
_ITM_transaction * td = _ITM_getTransaction();
 for (i=0;i<10;i++) {
    int doWhat = _ITM_beginTransaction (td,pr_instrumentedCode |
                                        &start_outer_loc);
    if (doWhat & a_restoreLiveVariables) {
    /* Compiler restores live local variables that aren't instrumented */
    }
    if (doWhat & a_abortTransaction) goto txn1_abort_label;
    if ((doWhat & a_saveLiveVariables)) {
    /* Compiler saves live local variables that won't be
     * instrumented.
     */
    }
    int sval = (int)_ITM_RU4 (td, (uint32 *)&s);
    sval += f_@TXN(i);    // Calls the instrumented "f"
    _ITM_WaRU4 (td, (uint32 *)&s, (unit32_t)sval);
    if (sval > 1000)
         _ITM_abortTransaction(td,userAbort,&abort_loc);
    _ITM_commitTransaction(td,commit_outer_loc);
    txn1_abort_label:
  }
  return;
}
```

Note that

- The write routine called is the write after read version since the compiler knows that the write is dominated by a read, however the read is the general routine because there has been no previous read of this address inside the transaction.

7.3 Nested transaction (C code)

```
extern int a,b;

int foo(int arg)
{
     atomic
     {
        a = a + 5;
        atomic
        {
             b = b + arg;
        }
        a = b + 5;
     }
     return(b);
}
```

Can be translated into something like this

```
static _ITM_srcLocation outer_start = {…};
static _ITM_srcLocation outer_commit = {…};
static _ITM_srcLocation inner_start = {…};
static _ITM_srcLocation inner_commit = {…};

int foo (int arg)
{
  _ITM_transaction * td = _ITM_getTransaction();
    int doWhat = beginTransaction (td,pr_multiwayCode | pr_hasNoAbort |
                                   pr_hasNoRetry | pr_hasNoXMMUpdate,
                                   &outer_start);
    if (doWhat & a_restoreLiveVariables) {
    /* Compiler restores live local variables that aren't instrumented */
    }
    /* No abort test since there is no user abort in the transaction */
    if (doWhat & a_runUninstrumentedCode) {   /* Uninstrumented code */
        a = a + 5;
        /* Inner transaction flattened in place in the uninstrumented code. */
        b = b + arg;
        a = b + 5;
    }
    else if (doWhat & a_runInstrumentedCode) {
      if ((doWhat & a_saveLiveVariables)) {
      /* Compiler saves live local variables that aren't instrumented */
      }
      /* a = a + 5; */
      int a_tmp = (int)_ITM_RfWU4 (td, (uint32 *)&a);
      a_tmp = a_tmp + 5;
      _ITM_WaWU4 (td, (uint32 *)&a), (uint32)a_tmp);
      /* Only instrumented code generated here, since the uninstrumented
       * version was generated inline in the outer transaction.
       */
      int doWhat = _ITM_beginTransaction (td,pr_instrumentedCode |
                                          pr_hasNoAbort | pr_hasNoRetry |
                                          pr_hasNoXMMUpdate,
                                          &inner_start);
      if (doWhat & a_restoreLiveVariables) {
      /* Compiler restores live local variables that aren't instrumented */
      }
      // No abort test since there is no user abort in the transaction
      // No test of what to execute since there's only one thing that
      // can be executed.
      if ((doWhat & a_saveLiveVariables)) {
      /* Compiler saves live local variables that aren't instrumented */
      }
      /* b = b + arg */
      int b_tmp = (int) _ITM_RfWU4 (td, (uint32 *)&b);
```

```
      b_tmp = b_tmp + arg;
      _ITM_WaWU4 (td, (uint32 *)&b), (uint32)b_tmp);
      _ITM_commitTransaction (td,&inner_commit);
 txn2_end:
      /* a = b + 5 */
      a_tmp = _ITM_RaWU4 (outer_td, (uint32 *)&b);
      a_tmp = a_tmp + 5;
      _ITM_WaWU4 (outer_td, (uint32 *)&a, (uint32)a_tmp);
    }
    _ITM_commitTransaction(td,&outer_commit);
    return b;
}
```

7.4 C++ exception handling

7.4.1 Simple atomic block

// foo() is a c++ function

```
int foo() {
    atomic {
        f(); // assume f() is not no-throw
             // (i.e., assume it may throw exception)
    }
}
```

We expect it to be transformed to something like this,

```
static _ITM_srcLocation txn_start_loc = {…};
static _ITM_srcLocation txn_end_loc = {…};

void foo(void)
{
    _ITM_transaction * td = _ITM_getTransaction();
    int doWhat = _ITM_beginTransaction(td,pr_instrumentedCode,&txn_start_loc);
    if (doWhat & a_restoreLiveVariables) {
    /* Compiler restores un-instrumented live local variables */
    }
    if (doWhat & a_abortTransaction) goto txn_abort_label;
    if ((doWhat & a_saveLiveVariables)) {
    /* Compiler saves un-instrumented live local variables */
    }
    _ITM_transactionId cur_id = _ITM_getTransactionId();
    bool catch_aborted = false;
    try {
       f_@TXN();    // Calls the instrumented “f”
    } catch (...) {
       _ITM_commitTransactionToId(td,cur_id,&txn_end_loc);
       if (_ITM_tryCommit(td,&txn_end_loc))
          throw; // rethrow the object
       catch_aborted = true;
    }
    if (catch_aborted)
       _ITM_abortTransaction(td,TMConflict,&txn_end_loc);
    _ITM_commitTransaction(td,&txn_end_loc);
  txn_abort_label:
    return;
}
```

7.4.2 Simple nested atomic block

// foo() is a c++ function

```
int foo() {
    atomic {
        f();
        atomic {
```

```
            g();
            if (cond1) __tm_abort;
            if (cond2) __tm_retry;
        }
    }
}
```

We expect it to be transformed to something like this,

```
static _ITM_srcLocation txn1_start_loc = {…};
static _ITM_srcLocation txn1_end_loc = {…};
static _ITM_srcLocation tnx2_start_loc = {…};
static _ITM_srcLocation txn2_end_loc = {…};
static _ITM_srcLocation abort_loc = {…};
static _ITM_srcLocation retry_loc = {…};

void foo(void) {
    _ITM_transaction* td = _ITM_getTransaction();
     int doWhat = _ITM_beginTransaction(td,pr_instrumentedCode,
                                        &txn1_start_loc);
     if (doWhat & a_restoreLiveVariables) {
         /* Compiler restores un-instrumented live local variables */
     }
     if (doWhat & a_abortTransaction) goto txn1_abort_label;
     if ((doWhat & a_saveLiveVariables)){
         /* Compiler saves un-instrumented live local variables */
     }
     bool catch_aborted = false;
     _ITM_transactionId txn1_id = _ITM_getTransactionId();
     try {
       f_@TXN();    // Calls the instrumented “f”
       int doWhat = _ITM_beginTransaction(td,pr_instrumentedCode,
                                          &txn1_start_loc);
        if (doWhat & a_restoreLiveVariables)  {
        /* Compiler restores un-instrumented live local variables */
        }
       if (doWhat & a_abortTransaction) goto txn2_abort_label;
       if (doWhat & a_saveLiveVariables) {
        /* Compiler saves un-instrumented live local variables */
       }
       g_@TXN();    // Calls the instrumented “g”
       if (cond1) _ITM_abortTransaction(td,userAbort,&abort_loc);
       if (cond2) _ITM_abortTransaction(td,userRetry,&retry_loc);
       _ITM_commitTransaction(td,&txn2_commit_loc) ;
   txn2_abort_label:
    } catch (...) {
       _ITM_commitTransactionToId(td,txn1_id,&txn_end_loc);
       if (_ITM_tryCommitTransaction(td,NULL))
          throw;
       catch_aborted=true;
    }
    if (catch_aborted)
       _ITM_abortTransaction(td,TMConflict,NULL);
    _ITM_commitTransaction(td,&txn1_commit_loc);
txn1_abort_label:
    return;
}
```

Note that the compiler has optimized away the try-catch block of the inner nested transaction.

7.4.3 Try-catch block

```
int foo() {
    try {
        f();
    } catch (...) {
        g();
    }
}
```

We expect it to be transformed to something like this,

```
static _ITM_srcLocation catch_start_loc = {…};
static _ITM_srcLocation catch_end_loc = {…};
static _ITM_srcLocation txn2_start_loc = {…};
static _ITM_srcLocation tnx2_commit_loc = {…};
static _ITM_srcLocation abort_loc = {…};

struct

// instrumented version of foo
void foo_@TXN(void)
{
    int i;
    bool catch_aborted = false;
    bool catch_executed = false;
    _ITM_transaction* td = _ITM_getTransaction();
    _ITM_transactionId cur_id = _ITM_getTransactionId();
    try {
        f_@TXN();    // Calls the instrumented “f”
    } catch (...) {
        _ITM_commitTransactionToId(td,cur_id,&commit_loc);
        catch_executed = true;
        int doWhat = _ITM_beginTransaction(td,pr_instrumentedCode |
                                           exceptionBlock,
                                           &catch_start_loc);
        if (doWhat & a_abortTransaction) {
             catch_aborted = true;
        } else {
            g_@TXN();
        }
    }
    if (catch_aborted)
        _ITM_abortTransaction(td,exceptionBlockAbort,&catch_end_loc);
    if (catch_executed)
       _ITM_commitTransaction(td,&catch_end_loc);
    return;
}
```

7.4.4 Atomic block with abort and retry

```
int foo() {
    atomic {
        f();
        if (cond1) __tm_abort;
        if (cond2) __tm_retry;
    }
}
```

We expect it to be transformed to something like this,

```
static _ITM_srcLocation txn_start_loc = {…};
static _ITM_srcLocation txn_end_loc = {…};
static _ITM_srcLocation abort_loc = {…};
static _ITM_srcLocation retry_loc = {…};

void foo(void)
{
    _ITM_transaction* td = _ITM_getTransaction();
    int doWhat = _ITM_beginTransaction(td,pr_instrumentedCode,txn_start_loc);
    if (doWhat & a_restoreLiveVariables) {
        /* Compiler restores un-instrumented live local variables */
    }
    if (doWhat & a_abortTransaction) goto txn_abort_label;
    if ((doWhat & a_saveLiveVariables)) {
        /* Compiler saves un-instrumented live local variables */
    }
    _ITM_transactionId cur_id = _ITM_getTransactionId();
    bool catch_aborted = false;
    try {
       _ITM_idSetter(td);
        f_@TXN();    // Calls the instrumented “f”
```

```
        if (cond1)
            _ITM_abortTransaction(td,userAbort,&abort_loc);
        if (cond2)
            _ITM_abortTransaction(td,userRetry,&retry_loc);
    } catch (...) {
       _ITM_commitTransactionToId(td,cur_id,&txn_end_loc);
       if (_ITM_tryCommitTransaction(td,NULL))
            throw;
       catch_aborted = true;
    }
    if (catch_aborted)
        _ITM_abortTransaction(td,TMConflict,&txn_end_loc);
    _ITM_commitTransaction(td,&txn_end_loc);
  txn_abort_label:
    return;
}
```

7.4.5 Catch block with abort

```
int foo() {
    atomic {
        f();
        try {
            g();
        } catch (...) {
            h();
            if (cond1) __tm_abort;
            if (cond2) __tm_retry;
            if (cond3) throw X();
        }
    }
}
```

We expect it to be transformed to something like this,

```
static _ITM_srcLocation txn_start_loc = {…};
static _ITM_srcLocation txn_end_loc = {…};
static _ITM_srcLocation catch_start_loc = {…};
static _ITM_srcLocation catch_end_loc = {…};
static _ITM_srcLocation abort_loc = {…};
static _ITM_srcLocation retry_loc = {…}

void foo(void)
{
    _ITM_transaction* td = _ITM_getTransaction();
    int doWhat = _ITM_beginTransaction(td,pr_instrumentedCode,&txn_start_loc);
    if (doWhat & a_restoreLiveVariables) {
        /* Compiler restores un-instrumented live local variables */
    }
    if (doWhat & a_abortTransaction) goto txn_abort_label;
    if (doWhat & a_saveLiveVariables) {
        /* Compiler saves un-instrumented live local variables */
    }
    _ITM_transactionId txn_id = _ITM_getTransactionId();
    bool catch1_aborted = false;
    try {
       f_@TXN();    // Calls the instrumented “f”
       bool catch2_aborted = false;
       bool catch2_executed = false;
       try {
          g_@TXN(); // calls the instrumented “g”
       } catch (...) {
          _ITM_commitTransactionToId(td,txn_id,&catch_start_loc);
          catch2_executed = true;
          int doWhat = _ITM_beginTransaction(td,pr_instrumentedCode |
                                             pr_exceptionBlock,
```

```
                                                &catch_start_loc);
          if (doWhat & a_abortTransaction) {
            catch2_aborted = true;
          } else {
            h_@TXN();
            if (cond1) _ITM_abortTransaction(td,userAbort,&abort_loc);
            if (cond2) _ITM_abortTransaction(td,userRetry,&retry_loc);
            if (cond3) throw x;
          }
      }
      if (catch2_aborted)  _ITM_abortTransaction(td,exceptionBlockAbort,NULL);
      if (catch2_executed) _ITM_commitTransaction(td,&catch_end_loc);
    } catch (...) {
         _ITM_commitTransactionToId(td,txn_id,&txn_end_loc);
         if (_ITM_tryCommitTransaction(td,&txn_end_loc))
             throw;
         catch1_aborted = true;
    }
    if (catch1_aborted)
        _ITM_abortTransaction(td,TMConflict,&txn_end_loc);
    _ITM_commitTransaction(td,&txn_end_loc);
 txn_abort_label:
    return;
}
```

7.4.6 Catch block with abort and throw

```
int foo() {
    atomic {
        try {
            f();
        } catch (...) {
            g();
            if (cond1) __tm_abort;
            if (cond2) __tm_retry;
            if (cond3) __tm_abort throw X();
            if (cond4) throw Y();
        }
    }
}
```

We expect it to be transformed to something like this,

```
static _ITM_srcLocation txn_start_loc = {…};
static _ITM_srcLocation txn_end_loc = {…};
static _ITM_srcLocation catch_start_loc = {…};
static _ITM_srcLocation catch_end_loc = {…};
static _ITM_srcLocation abort_loc = {…};
static _ITM_srcLocation retry_loc = {…};

void foo(void)
{
    int i;

    int doWhat = _ITM_beginTransaction(td,pr_instrumentedCode,&txn_start_loc);
    if (doWhat & a_restoreLiveVariables) {
            /* Compiler restores un-instrumented live local variables */
    }
    if (doWhat & a_abortTransaction) goto txn1_abort_label;
    if ((doWhat & a_saveLiveVariables)) {
            /* Compiler saves un-instrumented live local variables */
    }
    _ITM_transactionId txn_id = _ITM_getTransactionId();
    bool catch1_aborted = false;
    try {
       bool catch2_aborted = false;
       bool catch2_executed = false;
       try {
         f_@TXN();    // Calls the instrumented “f”
       } catch (...) {
```

```
            _ITM_commitTransactionToId(td,txn_id,&commit_loc);
            catch2_executed = true;
            int doWhat = _ITM_beginTransaction(td,pr_instrumentedCode |
                                               pr_exceptionBlock,
                                               &catch_start_loc);
            if (doWhat & a_abortTransaction) {
               catch2_aborted = true;
             } else {
                g_@TXN(); // This could throw!
               if (cond1) _ITM_abortTransaction(td,userAbort,&abort_loc);
              if (cond2) _ITM_abortTransaction(td,userRetry,&retry_loc);
               if (cond3) {
                 X x(); // create the object to throw
                 _ITM_registerThrownObject(td,&thrownObject,sizeof(X));
                 _ITM_rollbackTransaction(td,&abort_loc);
                  /* Compiler restores un-instrumented live local variables */
                 throw x;
               }
               if (cond4) throw Y();
             }
         }
         if (catch2_aborted) _ITM_abortTransaction(td,exceptionBlockAbort,NULL);
         if (catch2_executed) _ITM_commitTransaction(td,&catch_end_loc);
   } catch(…) {
         _ITM_commitTransactionToId(td,txn_id,&commit_loc);
         if (_ITM_tryCommitTransaction(td,NULL))
             throw;
         catch1_aborted = true;
   }
   if (catch1_aborted) _ITM_abortTransaction(td,TMConflict,NULL);
   _ITM_commitTransaction(td,&txn_end_loc);
txn1_abort_label:
    return;
}
```

7.4.7 Atomic block nested in catch

We provide the following example as an exercise to the reader:

```
int foo() {
    atomic {
        try {
            f();
        } catch (...) {
            atomic {
                try {
                    g();
                } catch (...) {
                    if (cond1) __tm_abort;
                    if (cond2) __tm_retry;
                   if (cond3) __tm_abort throw X();
                }
                if (cond4) __tm_abort throw Y();
                if (cond5) throw X();
            }
        }
    }
}
```

7.4.8 C++ destructors called during exception unwinding

A C++ exception throw invokes all the destructors for blocks that go out of scope. An abort while executing these destructors cannot long-jump to the `beginTransaction` because it leaves exception handling in an undefined state. These destructors, therefore, must be executed inside their own `exceptionBlock` transaction so that aborts are properly intercepted and propagated. Consider the following example:

```
int foo() {
    X x;
```

```
    f();
}
```
The destructor for the local variable x must be called if `f` throws an exception.

Compilers typically implement this by registering a handler that calls the destructor for x. Inside a transaction, this handler must start a nested `exceptionBlock` transaction before calling the transactional version of the destructor:

```
_X_dtor_exception_handler:
int doWhat = _ITM_beginTransaction(td,ExceptionBlock,NULL);
if (doWhat & a_abortTransaction)
   return; // the destructor aborted because of a conflict
x.~X.$TXN(); // call transactional destructor for X
_ITM_commitTransaction(td,NULL);
return;
```

8 References

http://www.agner.org/optimize/calling_conventions.pdf A good comparison and summary of IA32 and Intel 64 calling conventions for Windows and Linux.

Y. Ni, A. Welc, A. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R.Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *OOPSLA*, 2008. A description of the Intel STM library implementation and C++ language extensions.