This work was generated in the framework of the VELOX Integrated Project, which is co-financed by the European Commission through the contract no. 216852.

An Integrated Approach to Transactional Memory on Multi- and Many-core Computers

*EU FP7 project 216852 (1 Jan 2008 – 31 Dec 2010)*

SD4 Formal Specification of Language Extensions And Language Integration (D5.1, D5.2)

Derin Harmanci (UniNE)
## Document Information Page

<table>
<thead>
<tr>
<th><strong>Contract Number</strong></th>
<th>216852</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project Name</strong></td>
<td>An Integrated Approach to Transactional Memory on Multi- and Many-core Computers</td>
</tr>
<tr>
<td><strong>Project Acronym</strong></td>
<td>VELOX</td>
</tr>
<tr>
<td><strong>SUPER Deliverable Name</strong></td>
<td>Formal Specification of Language Extensions and Language Integration</td>
</tr>
<tr>
<td><strong>Reference Deliverable Numbers</strong></td>
<td>D5.1, D5.2</td>
</tr>
<tr>
<td><strong>Contractual Deadline</strong></td>
<td>M36</td>
</tr>
<tr>
<td><strong>Version</strong></td>
<td>v2.0</td>
</tr>
<tr>
<td><strong>Dissemination Level</strong></td>
<td>PU</td>
</tr>
<tr>
<td><strong>Nature</strong></td>
<td>R</td>
</tr>
<tr>
<td><strong>Responsible Author (Beneficiary)</strong></td>
<td>Derin Harmanci (UniNE)</td>
</tr>
<tr>
<td><strong>All Authors (Beneficiaries)</strong></td>
<td>Patrick Marlier (UniNE), Vincent Gramoli (UniNE), Pascal Felber (UniNE)</td>
</tr>
<tr>
<td><strong>Review and Approved by (Beneficiaries)</strong></td>
<td>Aldy Herndandez (RHAT), Jons-Tobias Wamhoff (TUD)</td>
</tr>
<tr>
<td><strong>Keywords</strong></td>
<td>(S)TM – (Software) Transactional Memory, ABI – Application Binary Interface, API – Application Programming Interface</td>
</tr>
</tbody>
</table>

The information in this document is provided as is. No guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability. Its owner is not liable for damages resulting from the use of erroneous or incomplete information.
**Version Log**

**External Version Log**

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>21 Feb 2011</td>
<td>Derin Harmanci (UniNE)</td>
<td>Initial version sent to the European Commission.</td>
</tr>
</tbody>
</table>
# Table of Contents

1 Executive Summary.......................................................................................... 7

2 The VELOX Stack at a Glance........................................................................ 7

3 Motivation ......................................................................................................... 8
   3.1 The Problem............................................................................................. 8
   3.2 The Solution............................................................................................ 9

4 Standardization of the Transactional Memory Library for C/C++............. 10
   4.1 Introduction............................................................................................. 10
   4.2 From Transaction Theory to Code Engineering .................................... 10
   4.3 Main ABI Functions............................................................................... 12
      4.3.1 Start a transaction ....................................................................... 12
      4.3.2 Commit a transaction .................................................................. 13
      4.3.3 Read memory barriers .................................................................. 13
      4.3.4 Write memory barriers .................................................................. 14
      4.3.5 Abort a transaction ....................................................................... 14
      4.3.6 Change execution mode .................................................................. 15
   4.4 Extended ABI Functions................................................................. 15
      4.4.1 Local variables accesses ............................................................. 15
      4.4.2 Optimized loads and stores using compiler hints ......................... 16
      4.4.3 Optimized block accesses ............................................................ 18
      4.4.4 Memory management .................................................................... 19
      4.4.5 Exception handling ....................................................................... 19
      4.4.6 Transaction descriptor extension ............................................... 19
   4.5 ABI Functions Available to the User.................................................. 20

5 Fundamental Transactional Semantics...................................................... 21
   5.1 Terms and Definitions............................................................................ 21
   5.2 Basic Transactional Behaviour ............................................................ 21
   5.3 Irrevocable Transactional Behavior ...................................................... 22
   5.4 Elastic Transactional Behavior............................................................ 22
      5.4.1 Why different semantics for transactions? ..................................... 22
      5.4.2 Elastic transactions ....................................................................... 23
      5.4.3 Evaluation of elastic transactions ................................................. 24
   5.5 Memory Model Associated to Transactional Behavior ......................... 27
      5.5.1 The C++/Java memory model at a glance ..................................... 27
      5.5.2 Memory model extensions for transactions .................................... 27

6 C++ Language Extension Specification.................................................. 28
   6.1 Fundamental Transactional Constructs .............................................. 28
6.2 Types of Transactional Guarantees ................................................................. 28
6.3 Function Usage in Transactional Code .............................................................. 30
6.4 Constructs for Control Flow in Transactions .................................................... 31
   6.4.1 Regular control flow constructs ................................................................. 31
   6.4.2 Transactional control flow constructs ....................................................... 31
   6.4.3 Exceptional control flow constructs .......................................................... 32
6.5 Transactional Nesting Support ............................................................... 33
   6.5.1 Basic nesting .................................................................................. 33
   6.5.2 Nesting and control flow ........................................................................ 33
   6.5.3 Nesting and exception handling ............................................................. 35

7 Java Language Extension Specification .................................................. 35
   7.1 Fundamental Transactional Constructs ....................................................... 35
   7.2 Types of Transactional Guarantees .............................................................. 36
   7.3 Function Usage in Transactional Code .......................................................... 36
   7.4 Constructs for Control Flow in Transactions ................................................ 37
      7.4.1 Regular control flow constructs ............................................................. 37
      7.4.2 Transactional control flow constructs .................................................. 37
      7.4.3 Exceptional control flow constructs .................................................... 40
   7.5 Transactional Nesting Support ................................................................. 40
      7.5.1 Basic nesting .................................................................................. 40
      7.5.2 Nesting and control flow ........................................................................ 40
      7.5.3 Nesting and exception handling .......................................................... 40

8 Implementation of C++ Language Extensions ........................................... 41
   8.1 Introduction ......................................................................................... 41
   8.2 gcc-tm ........................................................................................................ 41
      8.2.1 Objectives .................................................................................. 42
      8.2.2 Status ............................................................................................. 42
      8.2.3 Optimizations .................................................................................... 42
   8.3 DTMC ......................................................................................................... 43
   8.4 TinySTM and ABI Compatibility ............................................................... 44
   8.5 Summary .................................................................................................. 44

9 Implementation of Java Language Extensions ......................................... 44
   9.1 Introduction ............................................................................................. 44
   9.2 Deuce ....................................................................................................... 45
   9.3 TMJava ..................................................................................................... 45
   9.4 Summary .................................................................................................. 47
10 Software Prototypes................................................................. 47
11 References.............................................................................. 47
1 Executive Summary

The objective of this deliverable is to report the achievements in the integration of transactions to programming languages and to present the associated language extension specifications. As such, the document incorporates the following:

- explanation of the transactional behaviour that will be provided to the programmer,
- proposition of a standardized interface for TM libraries in providing this transactional behavior,
- specification of the language constructs introduced to integrate transactions into the C++ and Java programming languages,
- discussion of the implementation of the specified language constructs, and
- demonstration of the simplicity of programming with transactional memory

The Super Deliverable 4 Formal Specification of Language Extensions subsumes the following deliverables:

- D5.1 Java and C/C++ Language Integration
- D5.2 Formal Specification of the Language Extensions

The latest versions of all products mentioned in this document are available at: www.velox-project.eu/releases.

2 The VELOX Stack at a Glance

![The VELOX Stack, API and ABI Diagram](image-url)
3 Motivation

3.1 The Problem

TM-based programming has been proposed as a promising alternative to lock-based programming, introducing the abstraction of transaction into programming languages. However, up to recently the transaction support has been mostly provided in terms of software libraries and not as a programming language construct. Coding transactions using TM libraries hampers software development since:

- it is cumbersome and time consuming (the programmer needs to specify all the transactional accesses through library calls),
- the readability of the code is low (simple memory accesses that needs to be transactional become all library calls),
- it is not portable (the written software stays specific to the TM library for which it is written).

Especially the fact that a program using a specific TM library is not portable is an important issue for the integration of TM into programming languages. Different TM libraries perform better in different workloads and programmers may like to compile the same code with different TMs to find out the best performing TM for their application. Hence, it is desirable for a compiler to be able to choose different TM libraries for compilation.

The ideal would be to have a standard TM library interface so that any high level language providing higher-level transactional language constructs could use a single TM library interface. Providing TM library programmer such an interface allows the easy integration of any TM library implementation with a programming language.

The integration of TM libraries to the programming language using a standard interface is unfortunately not enough to integrate transactional behavior into programming languages. To complete the integration, it is required to specify a clear syntax and associated semantic for the application programmer. The ideal syntax to express a transaction is a block of code (we call such a block transaction block) in which the contents execute according to a specified transactional behaviour. The syntax of such an ideal transaction block is simple: it encloses code as would any other traditional block (e.g., function body, if statement body, loop body etc.). Although the syntax of the ideal transaction block is that simple, its semantics does not always fit with the semantics of the statements that could be enclosed in a transaction block, introducing difficulties of interoperability of the ideal transaction block with other language constructs.

Currently, TM is the major candidate for providing transactional behaviour, but the range of language level constructs the TM can control is limited. This is partly an implementation issue but the main problem is that several language constructs (such as system calls, I/O calls, synchronization constructs, exceptions etc.) cannot be rolled back and/or repeated (as transaction semantics would require). Hence, their interoperability with a transaction block needs to be solved in the language level rather than in the TM implementation level.
Specification of a standard TM library interface and a set of transactional language constructs sets the goals to reach for the complete integration of TM to programming languages. To achieve integration, these goals should be fulfilled with appropriate compilation and runtime tools. Hence, the integration is to be achieved by the extension of existing compiler and runtime tools.

### 3.2 The Solution

This document explains the different aspects of TM integration to programming languages and provides working solutions to each of these aspects. Part of the solution requires the specification of different interfaces: an interface at the TM library level (denoted as VELOX ABI in Figure 1) and a second at the programming language level (denoted as VELOX API in Figure 1). The interface at the TM library level aims at standardizing the interface for TM library calls (to be used by TM library programmers) while the programming language interface aims at describing the syntax and associated semantics of language extensions for transactional constructs (to be used by application programmers). The rest of the integration solutions lie in the implementation required to realize both interfaces as extensions to compilation and runtime tools. The document thus targets specifying the above mentioned interfaces and presents the required implementation associated to these interfaces.

This document first describes, in Section 4, the standard TM library interface named Application Binary interface (the VELOX ABI in Figure 1). This is followed by the specification of programming language interfaces for transactional execution which is denoted as VELOX API in Figure 1. As part of programming language interfaces, the fundamental semantics associated to a transaction block, which is common to any programming language, is described in Section 5. In the same section novel transaction block semantics, proposed within the VELOX Project is also detailed and numerical results are provided to prove its utility as transaction block semantics.

The specification of semantics is followed by the presentation of the syntax and semantics of language constructs proposed for C++ and Java languages in Sections 6 and 7, respectively. Each of these sections is organized in the same manner for simplifying the comparison of the language constructs introduced in the two languages. We detail the fundamental transactional constructs in the first subsection. We then explain the types of transactional guarantees proposed by each of the language specifications in the second subsection. In the third subsection, we illustrate the requirements on functions for using them within transactional code. We finally describe the support of control flow and nesting for programs including transactional constructs in rest of the subsections.

Sections 8 and 9 detail the implementation required for making the transactional language constructs explained in Section 6 and 7 operational. Section 8 elaborates on implementation details in C++ while Section 9 presents similar details for the Java language. The document is concluded with Section 10 where the prototype tools that realize the implementations described in Sections 8 and 9 are listed.
4 Standardization of the Transactional Memory Library for C/C++

4.1 Introduction
The input of the TM compiler is a C/C++ program, which can be enhanced by the Transactional C/C++ API. This program will be transformed to machine instructions and all transactional operations will be matched to library function calls. The reason to have distinct libraries is to support the exchange of the underlying TM implementation, for example if a new algorithm or a hardware TM is released. This way, all the management of transactions and concurrency is performed by an external library: the transactional memory runtime library (TM runtime).

Historically, the first TM runtime implementations for C/C++ have provided APIs to programmatically declare, start, commit, and abort transactions, as well as read from and write to transactional (shared) memory. This is the case, for instance, for TL2 (developed by TAU) and TinySTM (developed by UniNE/TUD). An essential aspect of C/C++ TM support is the standardization of the TM runtime library interface that specifies how the compiler maps transactional operations to the underlying TM runtime. Having a standardized interface allows us to use any complying compiler with any complying TM runtime.

Such convention between the compiler and the TM runtime has been named as the Application Binary Interface (ABI) and has been defined between several partners (VELOX partner as well as external partners) and companies. This section describes the specification of this interface in detail.

4.2 From Transaction Theory to Code Engineering
As mentioned in Section 4.1, the main transaction operations are start, commit, load and store, but these have to be converted to real machine code. The conversion process can be illustrated as follows:
An example of this transformation process using a Transactional C/C++ API is as follows:

```c
int a, b, c;
__transaction {
    a = b;
    if (c == 1)
        __transaction_abort;
}
```

```c
int a, b, c;
ret = _ITM_beginTransaction(pr_instrumentedCode);
if (ret != a_abortTransaction) {
    int tmp = _ITM_RU4 (&a);
    _ITM_WU4 (&b, tmp);
    tmp = _ITM_RU4 (&c);
    if (tmp == 1) {
        _ITM_abortTransaction ();
    }
    _ITM_commitTransaction ();
}
```

Note that this transformation is hidden to the developer and happens internally inside the compiler. The result is a compiled binary code (object code in .o files).

All functions names are prefixed with `_ITM_` to avoid clashes with any of the other library or user defined function names.

In the rest of the ABI specification, first main ABI functions are described and then other ABI functions are introduced accompanied with an explanation for their existence.
4.3 Main ABI Functions

This section describes all required functions to have a minimal transactional system.

4.3.1 Start a transaction

Starting a transaction is one of the most important steps because it drives how the transaction has to behave. To that end, the properties that the compiler detected are passed as arguments to the `start` function and the `start` returns some attributes to indicate how the transaction behaves.

```c
uint32_t _ITM_beginTransaction(uint32_t, ...);
```

The compiler can give properties for the transaction by passing them in to `beginTransaction`:

- `pr_instrumentedCode`, `pr_uninstrumentedCode` indicates if the instrumented or the uninstrumented code path is available. `pr_multiwayCode` is defined for convenience purposes.
- `pr_hasNoXMMUpdate` (also called `pr_hasNoVectorUpdate`) indicates that the transaction is not using vector registers (i.e., MMX/SSE for x86 CPU).
- `pr_hasNoAbort` indicates that the transaction has no user abort.
- `pr_hasNoRetry` indicates that the transaction has no user retry.
- `pr_hasNoIrrevocable` indicates that the transaction does not become irrevocable.
- `pr_doesGoIrrevocable` indicates that the transaction has to switch the serial irrevocable mode.
- `pr_aWBarriersOmitted` and `pr_RaRBarriersOmitted` indicates that the compiler has omitted “after write” or “Read after Read” barriers.
- `pr_undoLogCode` indicates that the transaction has only undo logging and no other barriers.
- `pr_preferUninstrumented` indicates that the uninstrumented code path is the best choice.
- `pr_exceptionBlock` indicates that the transaction has an exception block.
- `pr_readOnly` (GCC only) indicates that the transaction does transactional reads only.

```c
typedef enum
{
    pr_instrumentedCode       = 0x0001,
    pr_uninstrumentedCode     = 0x0002,
    pr_multiwayCode           = pr_instrumentedCode |
    pr_uninstrumentedCode,    
    pr_hasNoVectorUpdate      = 0x0004,
    pr_hasNoAbort             = 0x0008,
    pr_hasNoRetry             = 0x0010,
}
```

The result of startTransaction describes what actions to take:

- `a_runInstrumentedCode` and `a_runUninstrumentedCode` indicates either to run the instrumented code path or the uninstrumented one.
- `a_saveLiveVariables` and `a_restoreLiveVariables` indicates either to save or restore local variables.
- `a_abortTransaction` indicates to leave completely the transaction.

### 4.3.2 Commit a transaction

There are several functions for committing a transaction to deal with different cases. `_ITM_commitTransaction` is the regular commit function but for example, `_ITM_commitTransactionToId` is used to commit different levels of nested transactions. `_ITM_tryCommitTransaction` is used in the case of C++ program to commit transaction when there is a try/catch block.

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

### 4.3.3 Read memory barriers

Since the TM runtime library needs to know when the memory is read to manage conflicts, the ABI also incorporates the read function. Unfortunately, it is not enough to have only one read function because many types exist in C, so to cover all kind of type sizes, many functions are defined for sizes from 1 byte to 8 bytes and for different other types (floating point and complex values: `float`, `double`, `long double`).

```c
uint8_t _ITM_RU1(const uint8_t *);
uint16_t _ITM_RU2(const uint16_t *);
```
4.3.4 Write memory barriers
Like for read barriers, the transactional library needs to know all writes to the shared memory to manage concurrent accesses. Again as for reads, a collection of write barrier functions are needed for accesses that correspond to different size and types. Write barrier functions have 2 parameters: the address to be written and the value to write in it.

void _ITM_WU1 (uint8 *, uint8);
void _ITM_WU2 (uint16 *, uint16);
void _ITM_WU4 (uint32 *, uint32);
void _ITM_WU8 (uint64 *, uint64);
void _ITM_WF (float *, float);
void _ITM_WD (double *, double);
void _ITM_WE (long double *, long double);
void _ITM_WM64 (__m64 *, __m64);
void _ITM_WM128 (__m128 *, __m128);
void _ITM_WCF (float _Complex *, float _Complex);
void _ITM_WCD (double _Complex *, double _Complex);
void _ITM_WCE (long double _Complex *, long double _Complex);

4.3.5 Abort a transaction
The ABI function _ITM_abortTransaction enables a transaction to abort explicitly. Different parameters indicate the type of abort. Note that this function never returns.

typedef enum {
    userAbort = 1,
    userRetry = 2,
    TMConflict= 4,
    exceptionBlockAbort = 8,
    outerAbort = 16
}
On the contrary, `_ITM_rollbackTransaction` returns (no `longjmp`) and it rolls back a transaction to the innermost nesting level.

```c
void _ITM_rollbackTransaction(void);
```

### 4.3.6 Change execution mode

Unfortunately, a transaction may have to run in a different execution mode to meet its properties. For example, a transaction has to run in serial and irrevocable mode if an operation is undoable.

```c
typedef enum {
    modeSerialIrrevocable,
} _ITM_transactionState;
void _ITM_changeTransactionMode(_ITM_transactionState);
```

The argument to `_ITM_changeTransactionMode` indicates which mode to run. It can be extended to allow new modes of execution.

### 4.4 Extended ABI Functions

To allow advanced functions like memory allocation in transactions and also to increase the speed of transactional accesses, basic ABI functions have been extended.

#### 4.4.1 Local variables accesses

In a transaction, local variables which are outside of the transaction block can be accessed and also modified but if the transaction conflicts and must roll back, these local variables must be restored. The purpose of these `ITM_L*` functions is to save the variable before it is modified by the transaction. Of course, all accesses of local variables could be done with `ITM_RU/ITM_WU` but using this specific function saves a call to `ITM_RU` and also is less costly compared to a regular store.

```c
void _ITM_LU1(const uint8 *);
void _ITM_LU2(const uint16 *);
void _ITM_LU4(const uint32 *);
void _ITM_LU8(const uint64 *);
void _ITM_LF(const float *);
void _ITM_LD(const double *);
void _ITM_LE(const long double *);
void _ITM_LM64(const __m64 *);
void _ITM_LM128(const __m128 *);
void _ITM_LCF(const float _Complex *);
void _ITM_LCD(const double _Complex *);
void _ITM_LCE(const long double _Complex *);
```
_ITM_LB permits to log an arbitrary size of memory.

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

### 4.4.2 Optimized loads and stores using compiler hints

In order to alleviate performance issues encountered by Transactional Memory, the ABI incorporates some specific functions for accesses to the same address. The compiler can easily detect if the same address is accessed and then adds these specific calls.

Specific prefixes have been added:

- “aR”: after Read
- “aW”: after Write
- “RfW”: Read for Write

Examples of improvements achieved with optimized load and store functions using an experimental TinySTM with early conflict detection and write back strategy on a 64-bit machine is presented below.

<table>
<thead>
<tr>
<th>Improvements</th>
<th>Code size</th>
<th>Speed improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITM_RU8</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>(486 bytes)</td>
<td></td>
</tr>
<tr>
<td>ITM_RaRU8</td>
<td>58.8 %</td>
<td>8.3 %</td>
</tr>
<tr>
<td></td>
<td>(286 bytes)</td>
<td>(77 cycles)</td>
</tr>
<tr>
<td>ITM_RaWU8</td>
<td>16.4 %</td>
<td>15.4 %</td>
</tr>
<tr>
<td></td>
<td>(80 bytes)</td>
<td>(77 cycles)</td>
</tr>
<tr>
<td>ITM_RfWU8</td>
<td>69.8 %</td>
<td>9.10 %</td>
</tr>
<tr>
<td></td>
<td>(884 bytes)*</td>
<td>(140 cycles)*</td>
</tr>
<tr>
<td>ITM_WU8</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>(780 bytes)</td>
<td></td>
</tr>
<tr>
<td>ITM_WaRU8</td>
<td>0 %</td>
<td>0 %</td>
</tr>
<tr>
<td></td>
<td>(780 bytes)</td>
<td>(147 cycles)</td>
</tr>
<tr>
<td>ITM_WaWU8</td>
<td>13.3 %</td>
<td>15.4 %</td>
</tr>
<tr>
<td></td>
<td>(104 bytes)</td>
<td>(77 cycles)</td>
</tr>
</tbody>
</table>

* For the read plus the write operation, compared to regular read plus write.

The complete list of optimized load and store functions for different data sizes and types is given below.

```c
uint8 _ITM_RaRU1 (const uint8 *);
uint8 _ITM_RaWU1 (const uint8 *);
uint8 _ITM_RfWU1 (const uint8 *);
uint16 _ITM_RaRU2 (const uint16 *);
uint16 _ITM_RaWU2 (const uint16 *);
uint16 _ITM_RfWU2 (const uint16 *);
```


uint32 _ITM_RaRU4 (const uint32 *);
uint32 _ITM_RaWU4 (const uint32 *);
uint32 _ITM_RfWU4 (const uint32 *);
uint64 _ITM_RaRU8 (const uint64 *);
uint64 _ITM_RaWU8 (const uint64 *);
uint64 _ITM_RfWU8 (const uint64 *);
float _ITM_RaRF (const float *);
float _ITM_RaWF (const float *);
float _ITM_RfWF (const float *);
double _ITM_RaRD (const double *);
double _ITM_RaWD (const double *);
double _ITM_RfWD (const double *);
long double _ITM_RaRE (const long double *);
long double _ITM_RaWE (const long double *);
long double _ITM_RfWE (const long double *);
__m64 _ITM_RaRM64 (const __m64 *);
__m64 _ITM_RaWM64 (const __m64 *);
__m64 _ITM_RfWM64 (const __m64 *);
__m128 _ITM_RaRM128 (const __m128 *);
__m128 _ITM_RaWM128 (const __m128 *);
__m128 _ITM_RfWM128 (const __m128 *);
float _Complex _ITM_RaRCF (const float _Complex *);
float _Complex _ITM_RaWCF (const float _Complex *);
float _Complex _ITM_RfWCF (const float _Complex *);
double _Complex _ITM_RaRCD (const double _Complex *);
double _Complex _ITM_RaWCD (const double _Complex *);
double _Complex _ITM_RfWCD (const double _Complex *);
long double _Complex _ITM_RaRCE (const long double _Complex *);
long double _Complex _ITM_RaWCE (const long double _Complex *);
long double _Complex _ITM_RfWCE (const long double _Complex *);

void _ITM_WaRU1 (uint8 *, uint8);
void _ITM_WaWU1 (uint8 *, uint8);
void _ITM_WaRU2 (uint16 *, uint16);
void _ITM_WaWU2 (uint16 *, uint16);
void _ITM_WaRU4 (uint32 *, uint32);
void _ITM_WaWU4 (uint32 *, uint32);
void _ITM_WaRU8 (uint64 *, uint64);
void _ITM_WaWU8 (uint64 *, uint64);
void _ITM_WaRF (float *, float);
void _ITM_WaWF (float *, float);
void _ITM_WaRD (double *, double);
void _ITM_WaWD (double *, double);
void _ITM_WaRE (long double *, long double);
void _ITM_WaWE (long double *, long double);
void _ITM_WaRM64 (__m64 *, __m64);
void _ITM_WaWM64 (__m64 *, __m64);
void _ITM_WaRM128 (__m128 *, __m128);
void _ITM_WaWM128 (__m128 *, __m128);
void _ITM_WaRCF (float _Complex *, float _Complex);
void _ITM_WaWCF (float _Complex *, float _Complex);
void _ITM_WaRCD (double _Complex *, double _Complex);
void _ITM_WaWCD (double _Complex *, double _Complex);
void _ITM_WaRCE (long double _Complex *, long double _Complex);
void _ITM_WaWCE (long double _Complex *, long double _Complex);

4.4.3 Optimized block accesses
Accessing consecutive addresses using the regular read function can be costly because it requires using many calls to this function. In order to improve this situation, transactional versions of memset, memcpy and memmove have been defined.

memcpy and memmove have the same optimized barriers as loads and stores, but different versions of these functions follow the abbreviations stated below as a naming convention that determines whether the source or the destination can be accessed in a non-transactional manner:

- ‘R’ indicates read.
- ‘W’ indicates write.
- ‘n’ indicates non-transactional region.
- ‘t’ indicates transactional region.
- ‘aR’ indicates after read access.
- ‘aW’ indicates after write access.

The complete list of different transactional versions of memcpy function is as follows:

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

The memmove function has the same transactional versions as memcpy which are listed below:

void _ITM_memmoveRnWt(void *, const void *, size_t);
void _ITM_memmoveRnWtaR(void *, const void *, size_t);
The `memset` function is simpler because it has only the destination parameter. The different transactional versions available for `memset` are:

```c
void _ITM_memsetW(void *, int, size_t);
void _ITM_memsetWaR(void *, int, size_t);
void _ITM_memsetWaW(void *, int, size_t);
```

### 4.4.4 Memory management

In a transaction, the memory is managed differently because (i) if a transaction aborts the memory allocated during transaction execution has to be freed, and (ii) memory freed during transaction execution has to be protected until commit against concurrent accesses and the actual free operation should be performed only when the transaction commits.

```c
void *__ITM_malloc (size_t);
void *__ITM_calloc (size_t, size_t) ;
void *__ITM_free (void *);
```

The default `new` and `delete` operators of the C++ language can be wrapped by using transactional `malloc` or transactional `free` or having its own transactional calls but this is not yet standardized.

### 4.4.5 Exception handling

The exception management needs some help from the TM runtime library to write the thrown object to memory, otherwise the transaction mechanism will undo this object. To that end, the function `_ITM_registerThrownObject` have been added.

```c
void _ITM_registerThrownObject (const void *, size_t);
```

### 4.4.6 Transaction descriptor extension

All ABI functions require locating the transaction descriptor of the current transaction using Thread Local Storage (TLS), which can be costly in some cases. So in order to improve performance, another ABI version with the transaction descriptor as extra argument exists.

```c
typedef struct {} _ITM_transaction;
```
4.5 ABI Functions Available to the User

The user of transaction blocks may need to do specific actions while in a transaction, so some functions are available directly from the ABI.

The application may need to perform specific actions upon transaction commit or rollback. _ITM_addUserCommitAction adds an action to the commit log.

```c
typedef void (* _ITM_userCommitFunction) (void *);
void _ITM_addUserCommitAction(_ITM_userCommitFunction, _ITM_transactionId_t, void *)
```

_ITM_addUserUndoAction adds an action to the undo log that will be executed if the transaction aborts.

```c
typedef void (* _ITM_userUndoFunction)(void *);
void _ITM_addUserUndoAction(_ITM_userUndoFunction, void *)
```

The user can get the unique thread number that the transactional library generates. This identifier is used to match transactional statistics.

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

In some particular case, the application may need to unprotect previous transactional accesses, e.g., for a weaker transactional memory model.

```c
void _ITM_dropReferences (void *, size_t)
```

From a user perspective, it is important to check that the current version of the transactional library is compatible with its application. Two functions are defined to check the compatibility and to get the name of transactional library.

```c
int _ITM_versionCompatible (int);
const char * _ITM_libraryVersion (void);
```

In order to raise a fatal error while in transaction, the function _ITM_error has been defined:

```c
void _ITM_error(const _ITM_srcLocation *, int errorCode);
```
The user may propose alternative code if the transaction is in regular or irrevocable mode. To that end, _ITM_inTransaction returns the status of the current transaction.

```
_ITM_inTransaction
```

```c
typedef enum
{
    outsideTransaction = 0,
    inRetryableTransaction,
    inIrrevocableTransaction
} _ITM_howExecuting;
```

The application may need to get a transaction identifier, e.g., for debugging purposes.

```
_getTransactionId
```

## 5 Fundamental Transactional Semantics

Although the specification details the semantics of the various TM language constructs, all of these constructs share fundamental semantic characteristics, which we denote as the transactional behavior. In this section, we describe different possibilities for transactional behavior applicable for programming languages as well as the memory model associated to the transactional model.

### 5.1 Terms and Definitions

The transactional behavior is based on the definition of a transaction, which is a set of actions delimited, and distinguished from regular code, by a starting action (generally called start) and a terminating action (generally called commit). The code that appears between start and commit actions is defined as transactional code. Any code that is not transactional is called non-transactional code. Any memory access that takes place in transactional code is called transactional access, while a memory access performed in non-transactional code is called non-transactional access.

### 5.2 Basic Transactional Behaviour

The transactional behavior defines the guarantees required for the execution of the transactional code. A transaction is said to guarantee basic transactional behavior if the following properties are enforced for transactional code [2]:

- **atomicity**: either all the program state modifications performed by a transaction are visible at once to the other concurrent transactions (i.e., the transaction commits), or not visible at all (i.e., the transaction aborts).
- **isolation**: A transaction executes as if there is no other concurrent transaction in the system. Since the property is transitive it implies the following statements: (i) a transaction T does not observe the execution of other concurrent transactions (ii) concurrent transactions do not observe the execution of transaction T.
• **consistency:** A transaction always acts on a consistent state (even if it would abort in the end). In other words, a transaction is capable of capturing a consistent snapshot for the state of the data it accesses and perform modifications based on this consistent snapshot. The atomicity and isolation properties described above effectively ensure consistency; isolation allows a transaction to start from a consistent snapshot and atomicity allows applying modifications as a single indivisible operation with respect to other transactions. This results in a transaction to start and commit in consistent states.

The semantics provided by the basic transactional behaviour is also known as opacity in the literature [7]. Both C++ and Java language extension specifications aim to provide at least this basic transactional behaviour to the application programmer. Note that, however, the basic transactional behaviour is enforced only among transactions and the concurrent interaction of transactional and non-transactional code on the same data is left unspecified.

### 5.3 Irrevocable Transactional Behavior

The **basic transactional behavior**, as defined above, is not always enough for programming languages because some programming statements (such as I/O accesses, system calls or synchronization actions) cannot be rolled back. Such statements are called **irrevocable**, code including irrevocable statements is defined as **irrevocable code**. The use of irrevocable statements in transactional code requires a different semantics, which is called **irrevocable transactional behavior**. This semantics enforces the atomicity, isolation and consistency properties with respect to other transactions (as in basic transactional behavior) as well as the requirement of a transaction to be executed only once (for the correct execution of statements that can not be rolled back).

### 5.4 Elastic Transactional Behavior

Beyond basic and irrevocable transactional behaviour semantics our language extension specification introduces a novel transactional semantics (developed as part of the VELOX Project): **elastic transactional behaviour**.

#### 5.4.1 Why different semantics for transactions?

Despite having the appealing property of composition, implementations ensuring basic transactional behavior are known to execute generally slower than lock-based and lock-free alternatives. Conceptually, elastic transactional behaviour proposes to improve performance of transactional code based on following observation explained below.

In a concurrent environment, two operations may look very similar even though they do not share the same semantics. For example, this is the case for a `contains(z)` operation parsing a linked list data structure and failing in finding element `z` and another operation `size()` capturing an atomic snapshot of the number of elements of this data structure. Both operations have the same sequence of read/write accesses, yet they have distinct semantics. The following figure depicts the reads `r(*)` and write `w(*)` of these operations.
Figure 2. Two operations with same accesses and differing semantics.

The `contains(z)` is consistent even though `y` is concurrently inserted after `r(u)` occurs. Conversely, the `size()` requires for example that `u` and `y`, which are both counted, were both present in the linked list at the same time. Hence, `contains(z)` enables theoretically more concurrency than `size()` as it tolerates concurrent updates—and a fine-grained locking technique (e.g., hand-over-hand locking) will naturally benefit from this additional concurrency.

According to this observation, if `contains(z)` is executed using the basic transactional behaviour, the concurrent updates that are theoretically possible are forbidden. Conversely, using the elastic transaction behaviour for `contains(z)`, basic transactional behaviour is only enforced for consecutive pairs of read and write operations (as shown in Figure 2 for `contains(z)`) thus allows more updates to be performed, improving performance.

Another advantage of elastic transactions is that they can execute concurrently to transactions executing according to basic transactional behavior. As shown in Figure 2, while `size()` executes according to basic transactional behaviour, `contains(z)` can execute according to elastic transactional behaviour. The only condition that enables the use of elastic transactional behaviour is that the correctness of the code still holds under elastic transactional behaviour (as it is the case for `contains(z)`).

5.4.2 Elastic transactions

We propose a new semantics, *elastic opacity*, associated with transactions that allow transactional memory to compete (in performance) with fine-grained locking synchronization techniques. Elastic opacity builds upon the opacity consistency criterion (which is explained in Section 5.2). We call transactions that capture elastic opacity as *elastic transactions*. This transactional behaviour model is compatible with the basic transactional behavior model, as opposed to lock-free and lock-based counterparts. It relies on transactions, hence it is very simple to program with and allows more concurrency as it relaxes low-level atomicity while ensuring application-level atomicity.

5.4.2.1 Definition

A system is *elastic opaque* if there exist some consistent cuts of its elastic transactions such that: (a) the transactions resulting from these cuts and the regular transactions always access a consistent state of the system (even if they are pending or aborted), (b) they look like they were executed sequentially, and (c) this sequential
execution satisfies the real-time precedence of non-concurrent transactions and is legal.

The programmer can label a transaction as elastic if the transaction does not need to appear as atomic, but still requires that all couples of consecutive operations or the operations enclosed by write operations in this transaction appear as atomic. Basically, a cut of $H$ is consistent if there are no writes separating two of its sub-histories each accessing one of the objects written by these writes.

For example, consider the history $H_2$ depicted above in Figure 3 where $e$ is an elastic transaction and $n$ is a regular transaction, and where $r(x)^e$ and $w(x)^e$ refer to a read and a write operation on $x$ in transaction $t$. Two consistent cuts of $H_2|_e$ are possible. One contains two sub-histories $r(x)^e$, $r(y)^e$; $r(z)^e$, $w(u)^e$ while the other contains one sub-history $r(x)^e$, $r(y)^e$, $r(z)^e$, $w(u)^e$.

In contrast, consider history $H_3$, depicted below, where $e$ is elastic and $n$ is regular. There is no consistent cut of $H_3|_e$ because $n$ writes $y$ and $z$ between the times $e$ reads each of them.

### Figure 3

![Figure 3](image)

5.4.3 Evaluation of elastic transactions

The elastic transactional model allows, perhaps for the first time, STMs to compete performance-wise with the most efficient traditional synchronization techniques, be they lock-based or lock-free. As performance has always been a major issue limiting the widespread adoption of TM, we strongly believe that elastic transactions will foster the dissemination of the TM paradigm to the masses. We demonstrate below, how elastic transactions, already presented in D5.2 and published in [3], achieve such
performance. For evaluation we use ESTM, an STM we have developed to perform measurements [3].

5.4.3.1 Benchmark
We have implemented a simple Java Collection benchmark with size (10%), contains (80%), remove (5%), insert (5%) operations to compare the performance of our elastic transactions against lock-based and lock-free alternatives (although we only present the results for 10% update here, we have observed comparable behaviors with alternated configurations by varying the update ratio from 5% to 20%). We have used a simple linked list data structure with 1024 elements as depicted in Figure 2. The results obtained on a Sun UltraSPARC T2 (Niagara 2) are given in Figure 5, where each point of the graph is the throughput, given as the number of operations per millisecond, averaged over three runs of 3 seconds each.

1. For the lock-free evaluations, we have used the copyOnWriteArraySet from the java.util.concurrent (j.u.c) package. Initially, we tested the j.u.c.ConcurrentLinkedQueue algorithm and we discovered a bug for the size() operation. The size() could return an error margin that was growing linearly with the number of threads concurrently updating the data structure (we reported this issue on the JSR166 expert group on May, 20th 2010). The copyOnWriteArraySet was known as an alternative to a similar issue in ConcurrentSkipListMap [4], and we adopted the same solution for our evaluation. As the copyOnWriteArraySet technique may suffer from frequent updates as each update copies the entire array, we tested it on reasonably sized data structure (1024 elements) with realistic update ratios (10%).

2. For the lock-based evaluations, we used a Collections.synchronizedSortedSet technique to lock the data structure appropriately for all tested operations. Finer-grained synchronization would have been deadlock-prone in a Collection benchmark, as executing updates on multiple locations in different order would have deadlocked. Additionally, a deadlock would also arise when sizing two linked lists in opposite order.

3. For the transaction-based approach, we have developed a Java version of ESTM and integrated field-based elastic transactions in Deuce [5]. Elastic transactions are differentiated from regular transactions using the elastic parameter of the __transaction keyword (see Section 7.2). We have evaluated four additional field-based STMs with Deuce featuring only regular transactions (i.e., LSA, NOrec, TL2, SwissTM).

5.4.3.2 Observations
Overall elastic transactions (ESTM) speeds up the fastest regular STM, lock-based and lock-free techniques by factors of 2.5x, 4x, and 1.6x at the maximum level of parallelism we could test (64 hardware threads). More precisely, we observe that ESTM suffers from the overhead common to other STM implementations at low levels of parallelism; hence lock-based and lock-free alternatives are more efficient
up to 4 threads. This comes from the redirection of memory accesses imposed by transactions and by the costly metadata management associated with them.

![Graph](image)

**Figure 5.** Throughput (ops/ms) of elastic transactions as the number of threads grows. (Sequential throughput is lower than 40 ops/ms.)

As the number of threads becomes larger, the experiment gets more affected by contention, which lets STM outperform JDK lock-based alternatives. JDK lock-free performance remains, however, above regular STMs. Finally, at high levels of parallelism, ESTM compensates the overhead and copes with the additional contention because of the highly concurrent elastic transactions that enable higher concurrency. We can see that the JDK lock-free alternative that performs much faster than regular transactions, stops scaling at 8 threads, due to the overhead associated with the increasing number of data structure copy. The scalability of ESTM allows better performance than lock-free and purely regular STM alternatives starting at 16 and up to 64 threads.

5.4.3.3 Distribution

For M36, we have released the code of ESTM Java that has been successfully integrated into Deuce. The code is freely available on [http://lpd.epfl.ch/gramoli/php/estm.php](http://lpd.epfl.ch/gramoli/php/estm.php) and is part of the current Deuce distribution.

 Additionally, our recent results on the composition capabilities of elastic transactions appeared at the 2\textsuperscript{nd} Workshop on the Theory of Transactional Memory [6].
5.5 Memory Model Associated to Transactional Behavior

5.5.1 The C++/Java memory model at a glance

A memory model basically defines the ordering constraints on program execution. Writing a program that can be explained by the memory model results into data-race-free programs (i.e., a program with no data races). In a data-race-free program each read from a memory location sees the value written by the last write ordered by the happens-before relation [1].

Fortunately, all the ordering constraints of a memory model can be described in terms of the happens-before relation. The memory model of a C++/Java program can be summarized using the happens-before relation as follows [1]:

- If a statement A appears before another statement B in the code of the same thread, we say that A happens before B (this is also known as program order).
- If A and B are synchronization actions (known to C++/Java) appearing in different threads, these synchronization actions are ordered according to their specific ordering rules (e.g., a lock acquisition happens before a lock release). If due to the nature of synchronization actions A happens before B then by transitive closure the following is true:
  - for any statement C that appears before A in the code of same thread, we say that C happens before B (even if C and B are actions belonging to different threads).
  - for any statement D that appears after B in the code, we say that A happens before D (even if A and D are actions belonging to different threads).

5.5.2 Memory model extensions for transactions

Transactional behavior has important implications on the memory model. Introducing transactions into C++/Java language actually introduces two more synchronization actions. We will call them \texttt{starttx} and \texttt{endtx}. Each transaction should start with a \texttt{starttx} and terminate with an \texttt{endtx}, i.e., a transaction can be described only by its corresponding \texttt{starttx} and \texttt{endtx} pair. \texttt{starttx} and \texttt{endtx} extends the C++ memory model with the following additional ordering constraints:

- All statements between the \texttt{starttx} and \texttt{endtx} of the same transaction happen before the \texttt{endtx}, and the \texttt{starttx} happens before all the statements between the \texttt{starttx} and \texttt{endtx}.
- A \texttt{starttx} on thread T1 is either the first \texttt{starttx} or an \texttt{endtx} on another thread T2 happens before the \texttt{starttx} of T1. In other words, if the \texttt{starttx} of T2 happens before the \texttt{starttx} of T1, the \texttt{endtx} of T2 also happens before \texttt{starttx} of T1.
With the above constraints the happens-before relation implicitly defines an order where an operation O of a transaction T1 executed by one thread cannot be ordered in between starttx and endtx of another transaction T2 on another thread, thus O cannot appear to interleave operations of T2 (note that the constraint does not exclude nesting since the rules apply on starttx and endtx operations of concurrent threads). However, all the above ordering constraints do not explain the ordering between transactional and non-transactional accesses. Hence, there is no happens-before relation between concurrent transactional and non-transactional accesses to the same data and the result of such concurrent accesses is left unspecified (as mentioned at the end of Section 5.2).

6 C++ Language Extension Specification

6.1 Fundamental Transactional Constructs

The specification allows three language constructs to be executed in a transaction: compound statements, expressions and functions. Transactional compound statements are called transaction statements, transactional expressions are called transaction expressions and transactional functions are called function transaction blocks. The keyword that allows performing the execution of these three language constructs in a transaction is __transaction. The syntax for each of the transactional constructs is as follows:

- transaction statement: __transaction compound-statement
- transaction expression: __transaction ( expression )
- function transaction block: function-signature __transaction { function-body }

In general, it is common to associate a transaction to a transaction block, where the beginning and the end of the block are the start and commit actions and the content is the set of actions for which the transaction guarantees are ensured. Among the three constructs described above, the transaction statement corresponds to a transaction block while the transaction expression and function transaction blocks can be seen as derivations of the transaction statement. A function transaction block is merely a reusable transaction statement, while a transaction expression can be considered as the transactional computation of an expression that is equivalent to the following transaction statement:

__transaction { T temp = expression }

Hence, a transaction expression computes an expression in a transaction and stores the result of the expression in a temporary object.

6.2 Types of Transactional Guarantees

The __transaction keyword can be followed by one of two transactional attributes while declaring a construct transactional: [[atomic]] or
[[relaxed]]. These attributes specify that the following guarantee is ensured for the described transactional construct:

- The [[atomic]] attribute requires that the described transaction enforces the basic transactional behavior (e.g., it enforces only atomicity, isolation and consistency). This attribute forbids the use of statements that cannot be rolled back.
- The [[relaxed]] attribute suggests that the described transaction executes according to irrevocable transactional behavior if it includes irrevocable code, otherwise it executes according to basic transactional behavior.

A transaction that is assigned an [[atomic]] attribute is called an **atomic transaction**, while a transaction assigned a [[relaxed]] attribute is called a **relaxed transaction**. Syntactically the attributes just need to follow the `__transaction` keyword to make the distinction between the different transactional guarantees as follows (although below the syntax is given for transaction statement the same construction applies to transaction expression and function transaction blocks):

- **atomic transaction:** `__transaction [[atomic]] compound-statement`
- **relaxed transaction:** `__transaction [[relaxed]] compound-statement`

The explanations of [[atomic]] and [[relaxed]] attributes imply that atomic transactions enforce stricter guarantees. It is desirable to control that this stricter guarantee is respected at compile time. The specification calls any code that can be enclosed inside a relaxed transaction but not inside an atomic transaction as **unsafe**. More specifically a statement that is used in a transaction is deemed unsafe if any of the following applies¹:

- The statement is a transaction statement with the [[outer]] or [[relaxed]] attribute.
- The statement performs any use of a volatile object (initialization, assignment or a read).
- The statement is a function call that is
  - either not explicitly declared safe with the [[transaction_safe]] or the [[transaction_may_cancel_outer]] attributes (see Section 6.3 for these attributes),

---

¹ There is one more condition mentioned in the specification which is not presented here since it is a restriction on assembly code that can be used in a transaction and is out of the scope of this document.
² The outer attribute only applies to transaction statements and is a nesting related attribute which is explained in Section 4.5.2.
A statement is called safe if none of the above applies. According to this definition of safety, atomic transactions are transactions that contain only safe code. Since only relaxed transactions can contain irrevocable code, irrevocability mechanisms for transactional memory can only be used for implementations of relaxed transactions. Note that relaxed transactions provide basic transactional behavior if their content does not include irrevocable actions. If a relaxed transaction encloses irrevocable actions, it is only then the relaxed transaction provides irrevocable transactional behavior (where the basic transactional behavior guarantees are still possible while it is only the concurrency of the execution that is hampered).

### 6.3 Function Usage in Transactional Code

The usage of functions in atomic or relaxed transactions depends on whether the code they contain is safe. If a function contains only safe code it is said to be a safe function. A function can be declared safe using two attributes: `[[transaction_safe]]` and `[[transaction_may_cancel_outer]]`. It is also possible to explicitly declare a function unsafe using the `[[transaction_unsafe]]` attribute. Function safety attributes are syntactically located in front of the function signature, e.g., a function `f()` returning `void` can be annotated with one of the above safety attributes as follows:

```
[[transaction_safe]] void f()
[[transaction_unsafe]] void f()
[[transaction_may_cancel_outer]] void f()
```

Function pointers can also be declared safe or unsafe with the same attributes. This is useful to forbid the assignment of an unsafe function or function pointer to a safe function pointer.

In case of class inheritance, member functions preserve safety attributes declared for their base class. For virtual functions the overriding restrictions are as follows: A virtual function explicitly declared safe could only be overridden by a virtual function also explicitly declared safe (except that if the base class function is declared `[[transaction_safe]]`, the derived class function cannot be declared `[[transaction_may_cancel_outer]]`, while the reverse is possible).

For simplicity of assigning attributes to functions, the specification allows classes to be annotated with a function safety attribute (except the

---

3 The `[[transaction_may_cancel_outer]]` attribute is a control flow and nesting related attribute and is be explained in the corresponding sections (Sections 6.4.2 and 6.5.2).
Such a class attribute acts as the default attribute for all the member functions declared in the class unless overridden explicitly by the member function declaration. Also class attributes do not apply to functions inherited from a base class or functions included in the class via the using declaration.

If a function is not explicitly declared safe, it can still be inferred to be safe from its definition if it contains only safe statements. This is useful especially for template functions that do not take function safety attributes, because their safety can only be determined at compile time when the template parameters are known. In such a case, the compiler can decide the safety of the function analyzing the body of the function.

A final function attribute is the [[transaction_callable]] attribute but it has no safety implications. It is just a hint for the compiler to indicate that the function is intended to be called in relaxed transactions. Such an attribute can be used if the function is specifically written and optimized for use in relaxed transactions.

### 6.4 Constructs for Control Flow in Transactions

We analyze the three types of control flow constructs related to transactions: regular, transactional and exceptional control flow. The regular and transactional control flow constructs are related to the normal execution while exceptional control flow constructs are related to constructs to be used upon exceptions.

#### 6.4.1 Regular control flow constructs

The regular control flow statements `goto`, `break`, `return` and `continue` can also be used to transfer control out of a transaction. A limitation in normal execution control flow including transactions is that `goto` or `switch` statements must not be used to transfer control into a transaction statement.

#### 6.4.2 Transactional control flow constructs

Apart from the usual control flow statements, transactions have an additional control flow mechanism: aborting. If the code requires, a transaction can abort, i.e., can roll back all its modifications and transfer the control to the statement that follows the transaction. This can be done using a so-called cancel statement: the `__transaction_cancel` keyword (the keyword corresponds to the cancel statement by itself). A cancel statement is only allowed inside atomic transactions and only for transaction statements (and not for transaction expressions or function transaction blocks). The cancel statement should appear inside the transaction statement for which it would perform its rollback. A programmer can use a cancel statement within a function only if the function is annotated with a [[transaction_may_cancel_outer]] attribute. Details on this kind of usage are explained in Section 6.5.2.

It should be noted that an aborted transaction, although not having an effect on the memory once aborted, is part of the program execution and is subject to data races, i.e. a transaction that is aborted using `__transaction_cancel` could still be a reason for a data race occurring in the application.
6.4.3 Exceptional control flow constructs

The usual mechanism to control the flow of an application that raised an exception is to use a try-catch block. With the use of transactions this mechanism does not change. The specification, however, defines the way a transaction throws an exception and proposes two types of exceptional throw behavior in transactions: commit-and-throw and abort-and-throw. Both behaviors cause a transaction to throw the desired exception. However, the behaviors differ in the visibility of the effects of the transaction when the exception is thrown out of the transaction. With commit-and-throw behavior the effects of the transaction up to the point where the exception is raised are made visible to other threads with a commit. In other words, commit-and-throw allows partial execution of transactions. The abort-and-throw behavior, however, rolls back all the effects of the transaction (also mentioned as cancellation in the specification) and only then throws the exception.

The commit-and-throw behavior is provided by the existing C++ throw statement. To specify the abort-and-throw behavior it is enough to prepend the __transaction_cancel keyword to the existing C++ throw statement. More explicitly, the syntax for each of the behaviors are as follows:

- commit-and-throw: throw throw-expression
- abort-and-throw: __transaction_cancel throw throw-expression

If an exception is to be raised inside a transaction, it requires special care to preserve application consistency because a transaction should guarantee atomicity. If the partial effects of a transaction are visible to other threads at the time it raises an exception, the atomicity guarantee of the transaction will be violated. The commit-and-throw behavior allows this while the abort-and-throw behavior avoids such a problem by throwing the exception only after aborting the transaction. However, for abort-and-throw, since the exception is thrown after the transaction is aborted, the exception object cannot carry information about the actions performed in the transaction and hence objects that can be thrown inside a transaction are limited to integral and enumerated types.

The specification also extends the exception specification of traditional functions to transaction statements and transaction expressions (the function transaction blocks already have the possibility to express throwable exceptions through the exception specification syntax allowed for traditional functions). The syntax for transaction statements and transaction expressions with an exception specification is as follows:

```plaintext
__transaction tx-attrib throw( type-id-list ) compound-statement
__transaction tx-attrib throw( type-id-list ) ( expression )
```

---

4 Here, by transaction we mean a transaction statement or a function transaction block. By its nature a transaction expression cannot include a separate throw statement, so it can only support one of the behaviors; namely commit-and-throw)
In the above syntax the \textit{tx-attrib} is one of the \texttt{[[atomic]]}, \texttt{[[relaxed]]} attributes (for a transaction statement \textit{tx-attrib} can also be \texttt{[[outer]]}). The part that defines the exception specification is \texttt{throw(type-id-list)}. The \texttt{type-id-list} can be one of the following:

- a list of exception object types, e.g., \texttt{throw(int, char)}
- empty, i.e., \texttt{throw()}. Such syntax means that the transaction throws no exceptions.
- an ellipsis, i.e. \texttt{throw(...)}. This syntax means that the transaction can throw \textit{any} exception.

Not specifying any exception specification is accepted to be equivalent to \texttt{throw(...)}. If a transaction that throws an exception other than in the exception specification list, this results in a call to \texttt{std::unexpected()}.

### 6.5 Transactional Nesting Support

#### 6.5.1 Basic nesting

Syntactically, the nesting support in the specification is of two types: \textit{lexical} and \textit{dynamic}. Lexical nesting is where a transaction statement takes place inside transaction statements as code. Dynamic nesting is the one where the nested transaction statement is not directly visible inside the outer transaction statement, however, one of the statements inside the outer transaction statement actually causes another transaction to execute. An example of dynamic nesting is a transaction statement that contains a call to a function that further encloses a transaction statement in its body; the nesting is not observable from the transaction statement code that contains the function call, the compiler needs to check further in the code of the function to find out if there is a transactional nesting or not.

The specification allows both lexical and dynamic nesting of an atomic transaction inside atomic and relaxed transactions. Nesting of relaxed transactions in other relaxed transactions is also allowed. It is only the nesting of relaxed transactions in atomic transactions that is forbidden since it would violate safety of atomic transactions.

Semantically, the C++ language extension supports closed nesting, i.e., the abort of an inner transaction does not result in the abort of an outer transaction. The abort of an outer transaction should be specified explicitly.

#### 6.5.2 Nesting and control flow

The control flow also has support for nesting of transactions. The \texttt{__transaction_cancel} keyword by itself allows only aborting the innermost transaction, i.e., if the transaction is nested in other transactions the outer transactions would not be aborted. The specification defines two attributes to be able to abort also outer transactions: the \texttt{[[outer]]} attribute (for transaction statements and for the
__transaction_cancel statement), and the [[transaction_may_cancel_outer]] attribute (for functions).

When a group of nested transactions are aborted by the innermost transaction, a chain of aborts occurs to abort all the nested transactions in the reverse order to the nesting order (if the abort was explicitly requested to propagate out of the innermost transaction). The [[outer]] attribute is used to mark a transaction statement as the outermost transaction where the chain of aborts stops. In other words, it specifies that an abort cannot be propagated further out of such transaction. This attribute can only be used by transaction statements and such statements are called outer atomic transactions. An outer atomic transaction is accepted to be unsafe by the specification. The syntax illustrating the declaration of an outer atomic transaction is as follows:

__transaction [[outer]] compound-statement

The [[outer]] attribute is also used together with the __transaction_cancel statement to form a cancel-outer statement as follows:

__transaction_cancel [[outer]]

A cancel-outer statement aborts the outermost atomic transaction of a group of nested transactions and passes the control to the statement following the outermost atomic transaction. The outermost atomic transaction is the outer atomic transaction that lexically or dynamically contains the transaction where the cancel-outer statement is executed. A cancel-outer statement is allowed to be contained either lexically inside an outer atomic transaction or inside a function that has the [[transaction_may_cancel_outer]] attribute.

The [[transaction_may_cancel_outer]] attribute is the only way for a function to use cancel-outer statement. It also allows a convenient way for a cancel-outer statement to cancel an outermost atomic transaction from anywhere (rather than being limited with the lexical scope of the outer atomic transaction where it resides in). A function specified with the [[transaction_may_cancel_outer]] attribute is called a cancel-outer function. A cancel-outer function is accepted to be safe and should not contain unsafe statements.

The location of the [[transaction_may_cancel_outer]] attribute in the declaration syntax of a function is the same as the location of the [[transaction_safe]] attribute. Since [[transaction_may_cancel_outer]] attribute annotates a safe function it can neither be used with [[transactionUnsafe]] attribute (this will be a contradiction in the function safety) nor with the [[transaction_safe]] attribute (this will be redundant).
6.5.3 Nesting and exception handling
As the cancel statement can precede a throw statement to first abort a transaction and then propagate an exception, a cancel-outer statement can also precede a throw statement:

```__transaction_cancel [[outer]] throw throw-expression```

The semantic associated with adding the `[[outer]]` keyword is that instead of the transaction that encloses the throw, it is the outermost atomic transaction marked with the `[[outer]]` attribute that is aborted and that raises the specified exception. As with cancel-outer statement, such a throw statement is only allowed either to be lexically contained inside an outer atomic transaction or to be inside a function that has the `[[transaction_may_cancel_outer]]` attribute.

7 Java Language Extension Specification
7.1 Fundamental Transactional Constructs
Our Java language extension specification allows only a single language constructs to be executed in a transaction: compound statements. We call compound statements that exhibit transactional behaviour as transaction statements (as in C++ language extension specification). We do not specifically provide a transactional behaviour for expressions or for functions as in C++ language (see Section 6.1 for such language constructs). The reasons for such decision are as follows:

- Supporting transactional behaviour for mere expressions is nothing but a syntactic sugar for transaction statements enclosing simple language statements (see Section 6.1 to observe how a transaction expression can be written in terms of a transaction statement).
- Functions can be used both in transactional and non-transactional code even in the same program. The fact that a function is to be executed in a transaction is mainly decided by the caller of the function and the caller code does this performing the call in a transaction statement (which may enclose also other statements). Thus, a programmer does not need to indicate any function transactional: it is perfectly possible for a compiler to decide whether a function should support transactional behaviour or not. This is the approach taken by the Java language extension specification.

The keyword that allows performing the execution of a compound statement as a transaction statement is the same as in the C++ language (see Section 6.1 for the corresponding syntax in C++), i.e., it is the keyword `__transaction`. Hence, the syntax for a transaction statement is:

```__transaction compound-statement```
7.2 Types of Transactional Guarantees

The syntax for expressing of transactional guarantees for Java language extension is quite different from that of C++. Different transactional behaviors required by the programmer are given as a parameter to the __transaction keyword rather than an additional attribute. The three different transactional guarantees provided to the programmer are as follows:

- **atomic transaction**: An atomic transaction executes according to irrevocable transactional behavior if it includes irrevocable code, otherwise it executes according to basic transactional behavior. In that sense, an atomic transaction is equivalent to the relaxed transaction defined for C++ language extension specification (see Section 6.2). This is the default transactional guarantee specified for the Java language extension. The syntax for declaring an atomic transaction is:

  __transaction compound-statement, or
  __transaction(atomic) compound-statement

- **irrevocable transaction**: An irrevocable transaction executes only according to irrevocable transactional behaviour. The programmer may like to use this guarantee mainly for two purposes: (i) if the defined transaction statement encloses statements that are not known to be irrevocable by the compiler (e.g., user defined classes or class functions that needs to support irrevocable semantics), or (ii) for speeding up transaction execution by giving a hint to the compiler right at the beginning of the transaction (it may take a while for an atomic transaction to switch from basic transactional behaviour to irrevocable transactional behaviour when the necessity occurs). The syntax for declaring an irrevocable transaction is:

  __transaction(irrevocable) compound-statement

- **elastic transaction**: An elastic transaction executes according to elastic transactional behaviour. The syntax for declaring an elastic transaction is:

  __transaction(elastic) compound-statement

7.3 Function Usage in Transactional Code

The usage of functions in transaction statements is simplified for Java language extension specification (compared to C++ language extension). The programmer is exempted from the task of explicitly assigning attributes to functions to express whether the function is to be used in transactional code or not. This task is delegated to the language extension support (either to the compiler and/or the runtime system) as explained in Section 7.1. However, if a user defined class function includes irrevocable code this should be indicated to the compiler for correctness of a potential call to the function from transactional code. The syntax for indicating that a function
contains irrevocable code is by adding the @Irrevocable annotation to the function signature as follows:

```java
@Irrevocable void f()
```

In case of class inheritance, member functions preserve their irrevocable semantics except for overridden functions or interface member function implementations (this is because the overridden or implemented functions do not need to perform the required functionality using irrevocable code).

### 7.4 Constructs for Control Flow in Transactions

As for C++ language extension specification in Section 6.4, we present three types of control flow constructs related to transactions: regular, transactional and exceptional control flow.

#### 7.4.1 Regular control flow constructs

The semantics for usual control flow statements are kept exactly as in C++ language extension specification explained in Section 6.4.1 except that the `goto` statement does not exist in Java (i.e., `break`, `return` and `continue` can be used to transfer control out of a transaction while `switch` statements must not be used to transfer control into a transaction statement).

#### 7.4.2 Transactional control flow constructs

The transactional control flow constructs for the C++ language extension are mostly limited to provide the abort of a transaction statement where the transaction statement is mostly considered as merely a synchronization construct. The Java language extension tries to take advantage of transactional behaviour for the use of programmer in different possible ways. With this objective it proposes two types of control flow constructs: alternative execution paths, and novel transactional control flow keywords.

##### 7.4.2.1 Alternative execution paths

The Java language extension specification introduces a language construct (that does not exist in C++ language extension specification) to enrich transactional control flow: alternative execution paths. An alternative execution path can be considered to correspond to a `case` statement of a `switch-case` statement that resides in a transaction statement. The main difference is that an alternative execution path comes with an additional transactional control flow semantics: at any point inside an alternative execution path the modifications performed by the enclosing transaction statement can be aborted and the transaction statement can be re-executed using another alternative execution path.

Alternative execution paths are useful mainly in two kinds of situations. The first situation is when there are alternative ways to perform some actions provided in the transaction statement (as in graceful degradation). In such a case, each of the alternatives will correspond to a different alternative execution path in the transaction statement. If one alternative cannot be completed (e.g., due to some error, exception...
or a nonfulfillment of a programmer defined condition) re-execution of the transaction statement using another alternative can be performed by rolling back the effects of the uncompleted alternative. The second situation where alternative execution paths are useful is speculative execution. If a thread needs to execute one task among a bunch of speculative tasks and need to switch between one task to the other upon misspeculation, alternative execution paths propose a natural way to provide this behavior. The transactional nature of alternative execution paths matches the requirements speculative task execution upon misspeculation detection: when a misspeculation is detected the effect of the current speculative task is to be rolled back before executing another speculative task.

The syntax for alternative execution paths introduces three new keywords to the language; either, or and otherwise. Each of the keywords express different code blocks in which an alternative is specified. The block starting with the either and otherwise keywords correspond to the very first and the very last alternatives respectively. Each of the code blocks starting with an or keyword corresponds to one of the alternatives that are between the first and last alternatives (if there are only two alternatives no blocks starting with or exists). The syntax of alternative execution path language construct is presented in Figure 6.

```
// Statements preceding the transaction statement
...
__transaction{
  // transaction statement
  ///** Code Region A ***/
  ...
  // Beginning of code representing alternative execution paths
  either{
    // Code for the 1st alternative
    ///** Code Region B ***/
  }
  or{
    // Code for the 2nd alternative
    ///** Code Region C ***/
  }
  ...
  or{
    // Code for the (n-1)th alternative
  }
  otherwise{
    // Code for the (n)th alternative
  }
  ...
// Statements following the transaction statement
...  ///** Code Region D ***/
```

**Figure 6.** The syntax of the alternative execution path construct. The construct is composed of the either, or and otherwise blocks.
7.4.2.2 Transactional control keywords

The Java language extensions introduce transactional control keywords to give the programmer control of transaction execution. Three new keywords are introduced for this purpose: __transaction_retry, __transaction_next, and __transaction_cancel.

As in the C++ language extension, the __transaction_cancel keyword (i.e., the cancel statement) is used to abort a transaction statement (i.e., cancel all the effects of the transaction statement) upon a serious error where the software should stop execution immediately. Other keywords give more control to the programmer in determining the control flow of a transaction. __transaction_retry is useful in handling temporarily raised exceptions or situations that are generated due to temporary conditions such as data races, while __transaction_next is mainly useful within alternative execution paths especially if used for the purpose of graceful degradation or speculative execution.

According to the scope where the keywords are used, their semantics have slight differences. The semantics of each of the keywords is described using the following table:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Only in transaction statement</th>
<th>In alternative execution path</th>
</tr>
</thead>
<tbody>
<tr>
<td>__transaction_retry</td>
<td>Rollback and re-execution of the transaction statement</td>
<td>Rollback and re-execution of the transaction statement using the same alternative execution path</td>
</tr>
<tr>
<td>__transaction_next</td>
<td>Rollback and passing of the control to code following the transaction statement</td>
<td>Rollback and re-execution of the transaction statement using the next alternative execution path</td>
</tr>
<tr>
<td>__transaction_cancel</td>
<td>Rollback and passing of the control to code following the transaction statement</td>
<td>Rollback and passing of the control to code following the transaction statement</td>
</tr>
</tbody>
</table>

To concretize the behaviour associated to each the keyword example usage of the keywords over the code regions marked on Figure 6 is given as a table below:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Behavior upon keyword execution in region A</th>
<th>Behavior upon keyword execution in region B</th>
</tr>
</thead>
<tbody>
<tr>
<td>__transaction_retry</td>
<td>Rollback and re-execute region A</td>
<td>Rollback and execute code in region A and then B</td>
</tr>
<tr>
<td>__transaction_next</td>
<td>Rollback and execute code in region D</td>
<td>Rollback and execute code in region A and then C</td>
</tr>
<tr>
<td>__transaction_cancel</td>
<td>Rollback and execute code in region D</td>
<td>Rollback and execute code in region D</td>
</tr>
</tbody>
</table>
7.4.3 Exceptional control flow constructs

The two types of exceptional throw behavior in transactions, commit-and-throw and abort-and-throw also exist for the Java language extension. Transactional commit-and-throw and abort-and-throw statements are adopted as they are defined in C++ language extension in Section 6.4.3. So the syntax and semantics of commit-and-throw and abort-and-throw statements are the same as in C++. We repeat here the syntax of these statements for convenience:

- **commit-and-throw:** throw throw-expression
- **abort-and-throw:** __transaction_cancel throw throw-expression

As in the C++ language specification, the exception types that can be thrown with abort-and-throw statement are limited, however the limitation in Java language is different: only a single type of exception, CancelException, can be raised for abort-and-throw semantics. The programmer, however, can give the class of a valid exception class as a parameter to the CancelException as a means to convey the reason of the exception outside the transaction. The corresponding syntax is hence:

```
__transaction_cancel throw CancelException(ExceptionClass)
```

Apart from the exception throw behavior, transaction statements do not possess any exception specification in the Java language specification as opposed to C++ specification.

7.5 Transactional Nesting Support

7.5.1 Basic nesting

The Java language extension specification allows syntactically both lexical and dynamic nesting of atomic transaction inside atomic transactions. No nesting is allowed in irrevocable or elastic transactions. Semantically, only flat nesting of an atomic transaction in another atomic transaction is supported in the Java language extension, i.e., the abort of the inner transaction aborts also the outer transaction.

7.5.2 Nesting and control flow

As only flat nesting of transactions is supported, __transaction_cancel keyword by itself causes the automatic abort of the outermost transaction, so there is no need for a special keyword to indicate the aborting of an outermost transaction.

7.5.3 Nesting and exception handling

As for the control flow, since only flat nesting is supported by the specification, an exception thrown and propagated using a throw statement preceded by __transaction_cancel keyword causes the outermost atomic transaction to abort and raise the specified exception for further propagation.
8 Implementation of C++ Language Extensions

8.1 Introduction

A major objective of VELOX is to define higher-level language constructs to seamlessly support TM in the C/C++ common programming languages and add these constructs to “transactifying” compilers. This essentially requires handling the following tasks:

1. Transaction demarcation: where to start/end transactions, and how to handle unsuccessful executions (aborts).
2. Accesses to shared data: regular accesses to transactional memory should be automatically identified and transformed by the compiler.
3. Support transaction nesting and execution of external (irrevocable) actions or functions of legacy libraries in transactions.

Part of this tasks is covered by the specification of language extensions for C++ described in Section 6.

In addition to the language constructs, an essential aspect of C/C++ TM support is to support the ABI, which specifies how the compiler maps transactional operations to the underlying TM library calls. The ABI specification has been described in Section 4.

The language extensions and ABI are being standardized by working groups to which VELOX members participate (RHAT, TUD, UniNE). Discussion on the standardization processes are held on:

- The tm-abi mailing list hosted by RHAT (https://www.redhat.com/mailman/listinfo/tm-abi)
- The tm-languages Google group (http://groups.google.com/group/tm-languages)

Our TM stack follows the specifications being defined as part of the standardization process and provides implementations respecting the interfaces (both the ABI and the programming language constructs) defined by this process. Below, we present the tools that perform these implementations for C++ language.

8.2 gcc-tm

The gcc-tm Compiler has been developed by Red Hat in the context of the VELOX Project. It supports roughly the same language constructs as Intel’s C/C++ STM compiler [8] and Sun C++ Compiler with TM Extensions [9]. Being based on gcc, the most popular C/C++ compiler, gcc-tm is likely to reach a much greater developer base than Intel and Sun prototype compilers.

The language integration and ABI used by gcc-tm are based on the draft specifications of the working groups, which are summarized in Sections 4 and 6. Refer to the latest draft specifications [1] for details. Information about the development status of gcc-tm by Red Hat is given below.
8.2.1 Objectives

Red Hat's goal is to add extensions to allow TM programming using the industry-standard gcc compiler. Due to the nature of TM, which requires a supporting runtime environment to implement software or hybrid TM, and the tight integration of TM into the OS's ABI it is essential to coordinate the efforts among the parties working on TM for a platform.

In addition, ease of use of TM requires programming language extensions as opposed to library-approaches. This requires collaboration at that level even among platforms.

Red Hat has therefore entered into collaborations on TM with parties outside the VELOX Project. This will foster the goal of the EC to make TM viable and spread its use. Specially, Red Hat is collaborating closely with Intel on the extended Linux ABI to handle TM. Furthermore, collaboration with Intel, IBM, and Sun Microsystems exist to develop the extensions for the C++ programming language (see [1]).

The following is a description of the state of development of gcc (the GNU compiler collection) as performed by Red Hat and the API/ABI and language extensions as developed with partners inside and outside the VELOX Project.

8.2.2 Status

Red Hat develops the gcc-tm extension out in the public on a branch in the gcc repository named 'transactional-memory'. Everybody can follow the development and start using it although it is in the moment only really tested on Linux using the x86-64 and x86 architectures.

The implementation follows the results of the discussions of the TM ABI for Linux, as summarized in Section 4. The discussions are still going on and the draft release is lagging behind. gcc-tm usually implements the latest state of the discussions.

The main effort in the recent discussions centered around improving the performance of the software TM approach and to make the hybrid TM approach (using hardware TM when possible) flexible enough so that hardware support can be exploited easily when it is available and that, if necessary, alternative implementations can be handled in the same binary.

Furthermore, the use of TM in existing environments has received attention. Existing environments, practically all OSes, are not designed with TM in mind and have myriads of existing interfaces.

Enabling the use of these interfaces without undermining the advantages of TM is a big challenge. Various extensions to the language extensions and the implementation of the OS runtime have been discussed and implemented.

8.2.3 Optimizations

Instrumenting code to implement the transaction semantics means adding significant overhead. All memory accesses have to be investigated and possibly made known to
the TM runtime. In the first version of the gcc-tm compiler this happened unconditionally for all memory accesses.

There are several types of memory accesses for which no instrumentation is necessary because there can be no conflict with another thread.

Accesses to thread-local storage (TLS, see http://people.redhat.com/drepper/tls.pdf) don't have to be annotated because other threads are guaranteed to not have access. Only undo operations have to be performed upon cancellation. Even this is not necessary for transaction-local memory locations.

gcc-tm can also optimize accesses to newly allocated memory (e.g., using malloc). As long as the compiler can prove that the memory has not escaped to other threads only minimal instrumentation is necessary for the undo operation.

A second optimization is to instrument code so to avoid unnecessary calls for the same memory location. An expression like

\[ v = v + 42 \]

in a trivial implementation would add 'v' first to the read set and then later to the write set. In an optimized implementation with a sophisticated TM runtime the compiler could add code to announce that the variable is now read but later written to. The runtime can then make appropriate decisions right away (like aborting the transaction).

gcc-tm optimizes the TM instrumentation by injecting four of these combined notification of the TM runtime:

- Write-after-read (WaR): as described above.
- Read-after-write (RaW): the other way round. The variable also gets added to the read set.
- Read-after-read (RaR): a second read dominates the second.
- Read-for-write (RfW): the value is read to be stored.

Recognizing the above notifications allows the compiler to call special versions of the TM library. The ABI definition defines for many of the variants the interfaces to inform the runtime about the read and write sets. E.g., the _ITM_Ru4 function to add a 4-byte integer to the read set has an optimized _ITM_RfWu4 companion.

### 8.3 DTMC

DTMC (Tanger) [10] is a compilation tool that supports transactions in C and C++ programs. It uses the LLVM open compiler framework for automatically transactifying applications. DTMC performs transformations of the intermediary code (bytecode) produced by LLVM and maps transactional constructs to TinySTM (or its TinySTM++ variant in C++).

The early versions of DTMC used a minimal application-level API: instead of using explicit language constructs, the programmer had to insert explicit function calls to demarcate transactions and for various other tasks. The latest releases of DTMC
support the same explicit language constructs (transaction statements, function attributes; see Section 6 for all the constructs) as gcc-tm on the language level, and maps transactional operations to the same ABI (with some minor differences).

8.4 TinySTM and ABI Compatibility
TinySTM has been in development since the beginning of the VELOX Project. Recently, most of the development efforts have been made on the following aspects:

- ABI compatibility: TinySTM can be used with gcc-tm and Intel’s compiler.
- Support for several features required by the ABI specification, such as irrevocability (both serial and concurrent).
- Support for modular contention management, which allows developers to define conflict resolution policies and improve the performance and/or liveness of the application.
- Refactoring and performance optimizations.
- Support for delayed commit, as required by the event stream processing application.

8.5 Summary
On the C/C++ side, we now have two operational compilers (gcc-tm and DTMC), and an STM runtime that can be used with any transactional compiler that complies with the TM ABI specification. Some features are missing and more work is needed to enhance the robustness of the software stack, but the current prototypes are operational and can compile/execute non-trivial applications.

9 Implementation of Java Language Extensions

9.1 Introduction
Many STM prototypes for Java have been proposed over the last few years. Java is generally easier to support than C/C++, because it is a managed language and it provides higher-level, portable mechanisms for threading and concurrent programming.

We can classify Java STM implementations into three classes:

1) STM libraries with an API for transaction demarcation and explicit identification of transactional reads/writes. Examples include CCR [11], DSTM [12], DSTM2 [13], LSA-STM [15].
2) STM instrumentation frameworks that transform Java bytecode at load time (using BCEL/ASM-like libraries, or using AOP) using indications given by the programmer, typically annotations. Examples include LSA-STM [15] (also provides an explicit API), Deuce [5].
3) Transactional extensions to the language, with support for new constructs. Examples include AtomJava [16].
In VELOX, our objective is to support the second (for its portability and ease of implementation) and third (for its convenience and flexibility for the programmer) classes. To that end, we have developed the following software:

- The Deuce instrumentation framework [5].
- The TMJava (pre-)compiler that transforms Java source code with TM constructs into code that can be instrumented by Deuce.

Both are operational prototypes. We briefly describe below their main features.

### 9.2 Deuce

Deuce is a novel open-source Java framework for transactional memory. It has several desired features not found in earlier Java STM frameworks. It is non-intrusive in the sense that no modifications to the Java virtual machine (JVM) or extensions to the language are necessary. It uses, by default, an original locking design that detects conflicts at the level of individual fields without a significant increase in the memory footprint (no extra data is added to any of the classes) and therefore there is no GC overhead. This locking scheme provides finer granularity and better parallelism than former object-based lock designs. It also supports a pluggable STM back-end and an open easily extendable architecture, allowing researchers to integrate and test their own STM algorithms. In particular, it includes back-ends with the TL2 [14] and LSA [15] algorithms.

As Deuce does not have compiler support, it provides transactional behaviour at the level of methods. It uses Java annotations to indicate which methods should execute in the context of a transaction. It further performs bytecode instrumentation on the Java code, to transactify annotated methods.

Deuce has been heavily optimized for efficiency and, while there is still room for improvements, our performance evaluations on several high-end machines (up to 128 hardware threads on a 16-core Sun Niagara-based machine and a 96 core Azul machine) demonstrate that it scales well. Our benchmarks show that it outperforms the main competing compiler and JVM independent Java STM, the DSTM2 framework [5], in many cases by two orders of magnitude, and in general scales well on many workloads.

### 9.3 TMJava

Transactional extensions to Java augment the language with new constructs, typically by adding a new atomic keyword for transaction blocks (see Section 6.1 for transaction blocks). Besides a more convenient way to specify transaction blocks satisfying atomicity semantics, this approach provides finer transaction granularity and can support more sophisticated constructs to control recovery, retries, etc. [17] so that transactions can be used in more powerful ways. Transactional extensions do, however, require either using a pre-processor, or modifying a Java compiler to support the new constructs.

As previously discussed, Deuce provides the integration of an STM implementation into Java programs using bytecode instrumentation as long as the Java program has a structure suitable for Deuce (method-level transaction granularity, with transactions
declared using annotations). Therefore, we use the TMJava transformation front-end tool (pre-compiler) to transform the source code of a program written according to the Java language extension specification given in Section 7 into a program that is suitable to be instrumented by Deuce.

The transformation performed by the front-end tool maps the transaction blocks in the extended language into method bodies. In other words, the front-end tool analyzes the code to find the transaction blocks (__transaction keyword) inside class methods; it then creates new methods whose bodies are the content of the transaction blocks and replaces the blocks with calls to these new methods. However, moving an transaction block into a method body is not trivial, as we need to take into account several issues, notably:

- Variables and objects that are accessible inside the scope of the transaction block should also be available inside the scope of the method that corresponds to the transaction block.
- Modifications inside a transaction block on variables and objects that are defined outside the scope of transaction block should be visible outside the transaction block.

We call the variables and objects that are defined outside the transaction block but used and/or modified inside the transaction blocks as transaction-parameter-scope. To handle this issue, we pass the transaction-parameter-scope elements as parameters to the annotated method replacing the transaction block. Since the parameter names differ from original variable and object names, the naming inside the annotated method is modified accordingly. For the modifications on transaction-parameter-scope to be visible after the method call replacing the transaction block, we pass the variables and objects that require this visibility as elements of arrays to the annotated method and upon return we copy values of those array elements back to those variables and objects.

The transformation that has been explained up to this point only provides the extension of Java language to use a transaction block. The support for transactional and exceptional control flow requires the handling of new keywords __transaction_retry, __transaction_next and __transaction_cancel, and new blocks either/or/otherwise (for providing alternatives). Further exception handling features can also be added to with an on failure block as stated by Fetzer et al. [17], however this construct is not yet supported in the current prototype.

It is worth mentioning that TMJava transforms the Java code with TM extensions into “pure” Java source code that can be compiled using a standard compiler, but with the structure and annotations expected by Deuce for transactifying the application.

One should finally stress that there is no standardization process for Java TM extensions like for C/C++. Therefore, our language constructs are designed to loosely follow those proposed in C/C++ and in related work by other research groups.
9.4 Summary
Our efforts toward supporting STM in Java use a combination of bytecode instrumentation (easier to prototype and maintain in the long term) and the addition of new language constructs (easier to use). The Deuce and TMJava software prototypes cover respectively these two aspects.

10 Software Prototypes
Besides being available from the VELOX Web site, the software prototypes can also be downloaded from the following pages.

C/C++:
- **DTMC (Tanger)** is available from: [http://tm.inf.tu-dresden.de/](http://tm.inf.tu-dresden.de/)
- **TinySTM** is available from: [http://www.tmware.org/tinystm](http://www.tmware.org/tinystm)

Java:
- **TMJava** is available from: [http://www.tmware.org/tmjava](http://www.tmware.org/tmjava)
- **Deuce** is available from: [http://code.google.com/p/deuce](http://code.google.com/p/deuce)

11 References


