.NET Framework 4 Beta 1
enabled to use
Software Transactional Memory
(STM.NET Version 1.0)

Programmers' Guide

Published July 24, 2009

STM

This document supports a preliminary release of a software product that may be changed substantially prior to final commercial release. This document is provided for informational purposes only and Microsoft makes no warranties, either express or implied, in this document. Information in this document, including URL and other Internet Web site references, is subject to change without notice. The entire risk of the use or the results from the use of this document remains with the user. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Software Transactional Memory (STM) is an emerging technology for protecting shared state in concurrent programs, based on automatic control of concurrent accesses. Its goal is to provide a more user-friendly and scalable alternative to locks by promoting the notion of transactions as a first-class citizen in .NET. Microsoft is experimenting with the STM technology but at this point has no commitment to productize the technology.

Please install the software package ”.NET Framework Beta 1 enabled to use Software Transactional Memory” which will allow you to experiment with a preliminary release of the .NET runtime and framework that has transactional memory support. The following guide explains how to use the package, how to write programs with STM and how to contact us for help and feedback. We hope you enjoy working with this offering and would help us drive improvements into future releases.

There are two installations packages. In addition to the STM-enabled version of the .NET Framework, the second installation package contains a set of samples, documentation, and a VS2008 project template for creating STM application. This will get you going with STM!

We would very much like to hear from you about your experience with STM. Please feel free to comment and ask questions on the STM Devlab MSDN forum.

The latest version of this document is available at:

http://go.microsoft.com/fwlink/?LinkID=158427

Thank you for evaluating STM.NET.

—The STM.NET Team

Contents

1 Introduction 7

1.1 Caveats 7

2 Hello, World! 8

2.1 Known Issues with Debugging 12

3 What Problems Does STM Address? 14

3.1 The Crisis 14

3.2 What Are Transactions? Why Are They Useful? 15

3.3 Problems with Locking 15

3.3.1 Lock Accounting 15

3.3.2 Deadlock 16

3.3.3 Lack of Composition 17

4 Atomic Block to the Rescue: Basic Concepts 17

4.1 Failure Atomicity 18

4.2 Transaction Nesting 19

4.3 The System.TransactionalMemory Namespace 20

4.4 How Is STM implemented? What Does It Cost? 21

4.5 Putting the Basics Together: BankAccount Sample 22

5 Writing Correctly Synchronized Code 23

5.1 Ensuring Race Freedom 24

5.1.1 Follow Standard Data Hiding Practices 24

5.1.2 Synchronize All Accesses to Shared Data 25

5.1.3 Use Structured Parallelism 25

5.1.4 Publishing and Privatizing Data 25

5.2 Granularity 27

5.2.1 Bad Effects of Races due to Granularity Violations 29

5.2.2 Avoiding Granularity Problems 29

6 Atomic Compatibility Contracts 29

6.1 Contract Assignment to Metadata Items 30

6.1.1 Assembly Level Contract 31

6.1.2 Method Level Contract 31

6.1.3 Field Level Contract 31
6.1.4 Delegate Type Contract 31

6.2 Atomic Context....31

6.3 Putting Contracts in Context 32
6.3.1 Method Invocation 32
6.3.2 Field Access....33
6.3.3 Delegate Invocation 33
6.3.4 Delegate Instance Construction....33

6.4 Polymorphism 34

6.5 Contracts Sample 34

6.6 Static Checking for Contract Compatibility 34
6.6.1 Static Checking Errors 35
6.6.2 Using the static checker....37

6.7 Dynamic Contract Checking 40
6.7.1 Strictness Level of Dynamic Checking....40
6.7.2 Reporting Violations 41
6.7.3 Configuring the Runtime Checker....42
6.7.4 Example 42

7 Coordinating Threads with Transactions 44

7.1 Retry and Nesting....46

8 Transaction Abort in Greater Detail: Consistency and Exception Marshalling.... 46

8.1 Consistency of Aborted Transactions 46

8.2 Transactional Exception Marshaling....49

8.3 Re-execution Terminology Recap....50

9 Transactions at Large: Integration with System.Transactions 50

10 Taming Side Effects inside Transactions 53

10.1 Atomic Suppress....54
10.1.1 AtomicSuppress and Inconsistent State 56
10.1.2 Current Limitations of AtomicSuppress....57

10.2 AtomicRedirect....57

10.3 Using the Monitor Class inside Atomic Blocks....59

10.4 Deferred and Compensated Actions....60
10.4.1 Deferred Actions 60
10.4.2 Compensated Actions....61

10.5 Implementing a Resource Manager for Your Operation 62

10.6 Summary ... 63

11 Performance Optimization and Troubleshooting ... 64

11.1 Sequential Performance ... 64

11.2 Scalability ... 65

11.2.1 Contention Management ... 66

11.2.2 Array concurrency ... 67

11.2.3 Scalability Bottlenecks in this Release ... 68

11.3 Benchmarking Results ... 70

11.4 STM vs. Reader-Writer Lock vs. Concurrency Data Structures (CDS) ... 72

11.5 Performance Tips ... 76

11.5.1 “readonly” Field Modifier ... 76

11.5.2 x = value; → if (x != value) x = value; ... 77

11.5.3 Avoid Contention on an Object: the Deque Example ... 77

11.5.4 Benign Conflicts: the Linked-List Example ... 78

11.5.5 ByRef ... 78

11.6 Using Event Tracing to Diagnose STM Application ... 79

12 References ... 82

13 Index ... 83

1 Introduction

The .NET Framework version 4 Beta1 enabled for Software Transactional Memory (STM.NET version 1.0) provides experimental language and runtime features that allow the programmer to declaratively define regions of code that run in isolation. This specific implementation of software transactional memory (STM) is implemented using a modified version of the Common Language Runtime (CLR) version 4.

The goal of this document is to explain what STM is, why we chose this specific implementation, how to run the provided samples, and how to write your own applications using STM.

This is both a programming guide and an introduction to transactional memory. If you want to jump right into using STM, you will want to read the following sections first:

- **Section** 2 **"Hello, World!"** This section demonstrates how to compile and run a basic STM sample.
- **Section** 4 **"Atomic Block to the Rescue: Basic Concepts"**. This section explains basic concepts of STM.
- **Section** 5 **"Writing Correctly Synchronized Code"**. This section explains how to write your code such that it is correctly synchronized and doesn't have race conditions.

For a deeper understanding of both STM in general and our implementation, you may want to read through this manual in order. We have arranged it to present:

- Writing "Hello World"
- Introduction to Transactional Memory Concepts
- Writing Correctly Synchronized Code
- Atomic Compatibility Contracts
- Coordination Between Threads When Using Transactional Memory
- Transaction Abort in Greater Detail
- Integration with Traditional Transactions
- Dealing with I/O and other Side Effects
- Performance Considerations and Troubleshooting with ETW (Event Tracing for Windows)

1.1 Caveats

1. **This is a MSDN DevLab incubation release**. There are no confirmed plans to ship STM as part of any version of the .NET Framework. The license to use this software specifically prohibits developing and deploying production software on this framework.
2. **There are no performance optimizations.** We have not spent much time optimizing the performance of our changes. Further, NGEN is disabled. You cannot make meaningful performance comparisons between programs executing on this framework to the same code running on another version of the framework.
3. **Alpha-level quality**. We are a small team tackling a big problem. There are many known and many unknown bugs and limitations. We have done the best that we can to ensure that all samples work

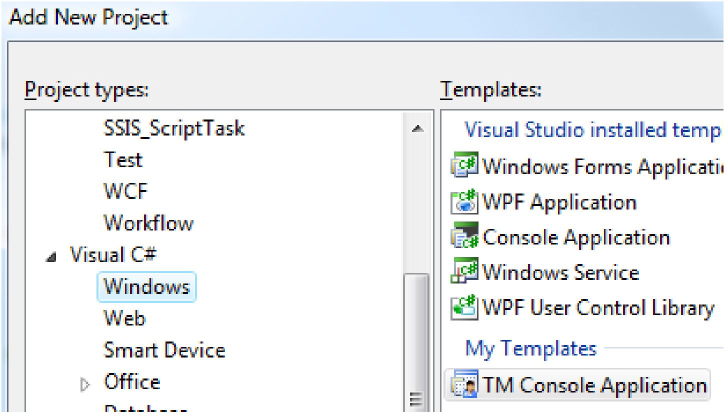
and that all features covered in this document work in their basic usage. Please report problems through the MSDN forum for STM.NET. Although we will make an effort to answer users' concerns and issues, this software offering is not supported by Microsoft and its use is restricted as per the terms in the End User License Agreement (EULA).

4. **Modified .Net Framework 4 Beta 1 base**. The current STM release is based off of the .NET Framework 4 Beta 1 but has been modified to provide STM and work in a Visual Studio 2008 environment.

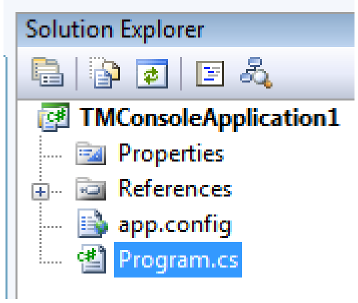
2 Hello, World!

Before we go into the fine details of STM, let's go through a simple "Hello World" sample, STM style.

Start *Visual Studio 2008*. Choose *File* | *New* | *Project*; choose *Visual C#* project type, and *My Templates* group. You should see there "*TM Console Application*". This is a custom template that was installed on your computer that has a combination of settings that are required for the building and debugging of STM applications using STM.NET. Select it, specify any name/location of your liking, and click OK.



When your new project opens, take a look at the Solution Explorer on the right side:



There are two details to note:

1. **app.config tweaks the VS environment to use STM.NET**. In case you're curious, this is what the file contains. We will talk more about the dynamic checker in section 6.6.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <requiredRuntime version="v4.0.20506"/>
  </startup>

  <runtime>
    <ZapDisable enabled="1" />
    <gcServer enabled="true" />

    <STMLogOnViolation enabled="0" />
    <STMExceptionOnViolation enabled="1" />
    <STMRuntimeCheckLevel enabled="strict" />
    <!--
      <STMRuntimeCheckLevel enabled="minimal" />
      <STMRuntimeCheckLevel enabled="relaxed" />
      <STMRuntimeCheckLevel enabled="strict" />
    -->

  </runtime>
</configuration>
```

2. The assembly references in your project contain versions of mscorlib.dll and System.Transactions.dll which enable the use of STM. Use these references when developing STM-enabled applications.

The default Program.cs contains an empty Main method[1]:

```
static void Main(string[] args)    {    }.
```

We are now ready to code up our STM Hello World program. STM is about providing thread-safety. Therefore our scenario will contain threads that are performing some tasks in a thread-unsafe manner. We will demonstrate how to provide safety using STM.

In our scenario, there will be two string fields. The program tries to maintain the invariant that these two fields do not reference strings with the same string value. i.e., if field1 equals field2, then we have a violation of the program's expectation.

Two threads will be constantly assigning string references to the fields and a third thread will be periodically verifying that the program's invariant is upheld.

Let's start with a naïve and wrong program without locks or any other form of synchronization to provide thread-safety. Copy the following code instead of Main, then build and run the program. You should see a bunch of lines with the phrase "Violation!" in the output: this is the "watchdog" thread noticing violations of the program's invariant.

[1] Main also has a mysterious looking attribute assigned to it [AtomicNotSupported]. You can safely ignore this for now. We will discuss this attribute and other Atomic contract attributes in section 6.

```
private class MyObject
{
    private string m_string1 = "1";
    private string m_string2 = "2";

    public MyObject() { }

    public bool Validate() {
        bool result;
        result = (m_string1.Equals(m_string2) == false);
        return result;
    }
    public void SetStrings(string s1, string s2){
        m_string1 = s1;
        Thread.Sleep(1);   // simulates some work
        m_string2 = s2;
    }
}

[AtomicNotSupported]
static void Main(string[] args) {
    MyObject obj = new MyObject();
    int completionCounter = 0, iterations = 1000;
    bool violations = false;

    Thread t1 = new Thread(new ThreadStart( delegate {
        for (int i = 0; i < iterations; i++)
        {
            obj.SetStrings("Hello", "World");
        }
        completionCounter++;
    }));
    Thread t2 = new Thread(new ThreadStart( delegate {
        for (int i = 0; i < iterations; i++)
        {
            obj.SetStrings("World", "Hello");
        }
        completionCounter++;
    }));
    Thread t3 = new Thread(new ThreadStart( delegate {
        while (completionCounter < 2)
        {
            if (!obj.Validate())
            {
                Console.WriteLine("Violation!");
                violations = true;
            }
        }
    }));
    t1.Start();
    t2.Start();
    t3.Start();
    while (completionCounter < 2)
        Thread.Sleep(1000);

    Console.WriteLine("Violations: " + violations);
}
```

Did you get violations? Great, that was expected. By the way: how many statements in the program above need thread-safety which isn't provided? (i.e., how many *races* can you spot?)

We could add locks or mutexes to provide thread safety and if you want you can take a moment to do that. Now let's try to use STM instead. STM will provide the needed isolation, without having to worry about what locks to use and how to avoid deadlocks.

To add the needed isolation in the Hello World sample, please replace the Validate and SetString methods of MyObject with the following code:

```
public bool Validate()
{
    bool result = false;
    Atomic.Do(()=>
    {
        result = (m_string1.Equals(m_string2) == false);
    });
    return result;
}

public void SetStrings(string s1, string s2)
{
    Atomic.Do(()=>
    {
        m_string1 = s1;
        Thread.Sleep(1);
        m_string2 = s2;
    });
}
```

Instead of using locks, we instead used "Atomic.Do" to wrap access to shared state in what we refer to as an "atomic-block". When you see:

```
Atomic.Do(()=> { <statememts> });
```

Try to squint at it and imagine that you are actually seeing this:

```
atomic { <statememts> }
```

In order to provide the "atomic" keyword we had to change multiple language compilers and our work so far has concentrated on libraries and runtime. For this experimental release, we could not provide the "atomic" keyword in the language, so we actually demarcate atomic-blocks using a try/catch(AtomicMarker). AtomicMarker is not a real exception that we ever throw and the catch body will never get executed[2]. We provide the simple "Atomic.Do" construct to hide the implementation details.

Build and run the sample now, there should be no violations anymore (although, one race condition remained in the code, have you noticed it?!)[3].

In this sample case, you could modify the code by adding a lock (this) statement around the racy code. This is not an example where STM is particularly interesting with the exception that you didn't have to specify what to lock. It was inferred automatically for you by the system. We will talk more about the traits of STM and how it's different from locks in the next section: introduction to STM.

2.1 Known Issues with Debugging

The .NET Framework enabled to use STM has rudimentary support for debugging STM-enabled applications under Visual Studio 2008. There are some things which will not work correctly when stepping through the code, inside "atomic" blocks. In order to get this rudimentary debugging experience the setup package for STM.NET samples and documentations modifies your Visual Studio environment in the following manner:

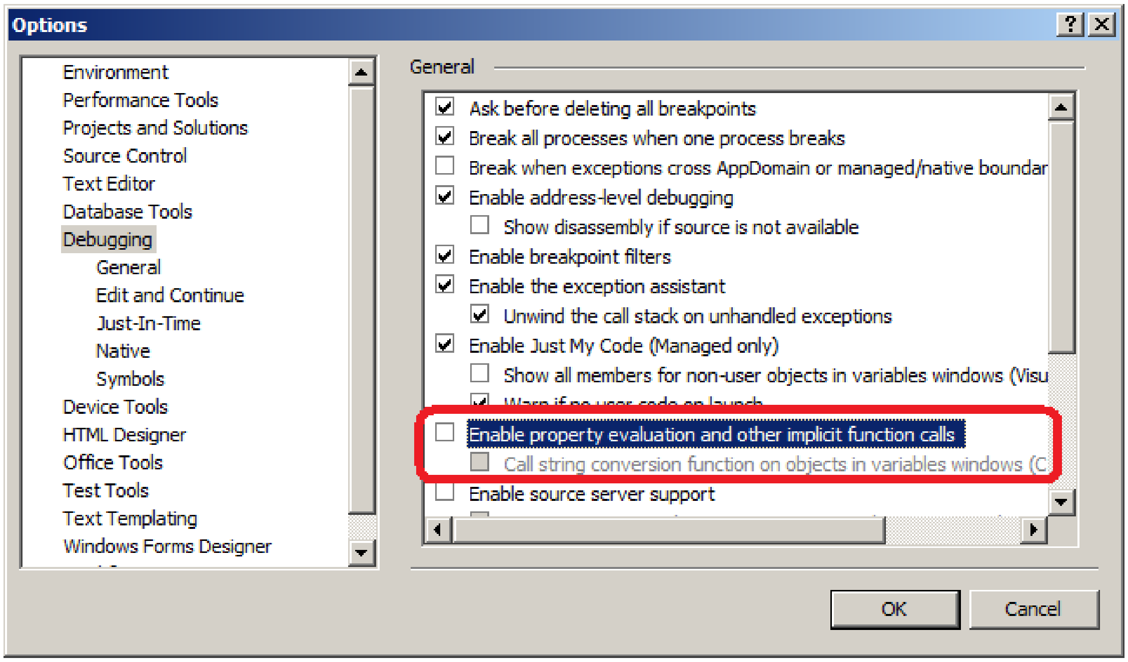
1. The "Call string conversion function on objects in variables windows" option is disabled.
2. The "Enable property evaluation and other implicit function calls" option is disabled.
3. The "Edit and Continue" feature is disabled as well.

   Please make sure that these options remain disabled whenever you are debugging STM-enabled programs. The below screen shots illustrate how to make sure these options are disabled.

   Select Menu | Tools | Options | Debugging, and then inspect the general options and the edit-and-continue sub-options, as illustrated below.

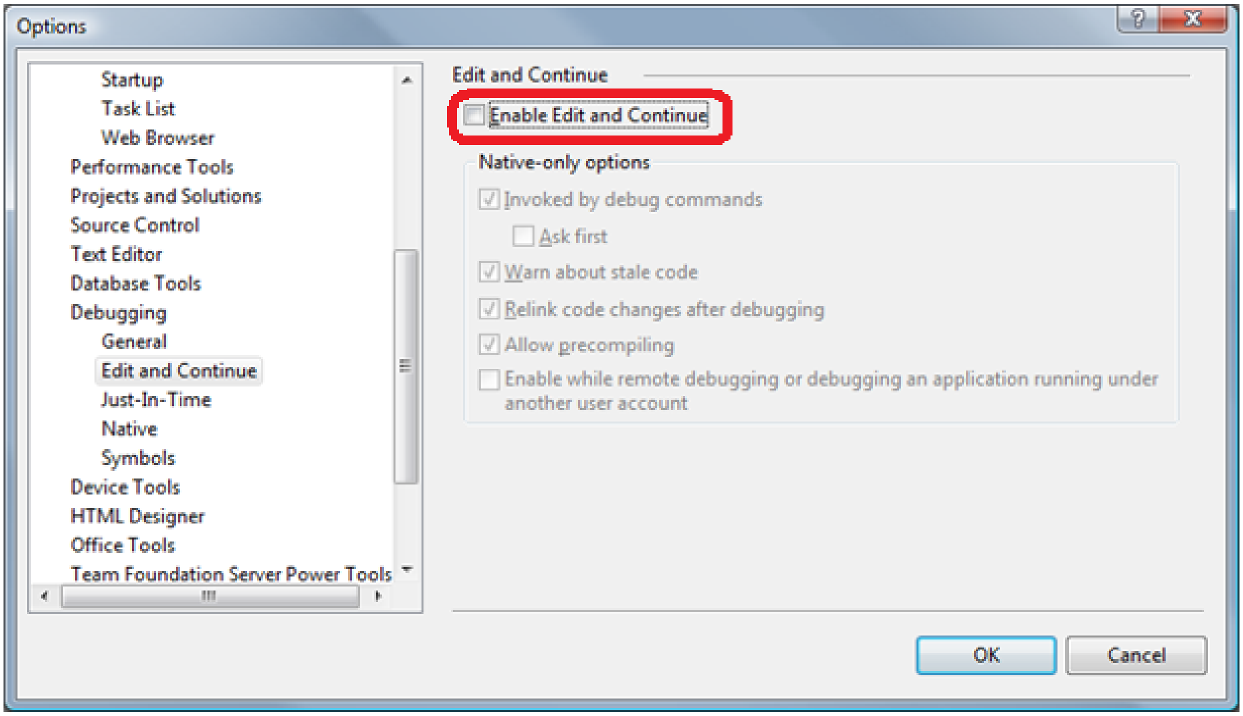
[2] Sometimes you would have to put a 'throw' in the catch handler of AtomicMarker, to keep the C# or VB.Net compiler happy, since they are not aware that control flow never reaches those points. The 'throw', like any other code you put there will never get executed.

[3] As you run the Hello World sample and other samples you may note that exceptions of type AbortReexecException are reported as thrown in the debugger output windows. These exceptions are internal to the STM system. They are expected and do not indicate any problem in your environment.



General

☑ Ask before deleting all breakpoints
☑ Break all processes when one process breaks
☐ Break when exceptions cross AppDomain or managed/native boundar
☑ Enable address-level debugging
☐ Show disassembly if source is not available
☑ Enable breakpoint filters
☑ Enable the exception assistant
☑ Unwind the call stack on unhandled exceptions
☑ Enable Just My Code (Managed only)
☐ Show all members for non-user objects in variables windows (Visu
☑ Warn if no user code on launch
☐ Enable property evaluation and other implicit function calls
☐ Call string conversion function on objects in variables windows (C
☐ Enable source server support



Edit and Continue

☐ Enable Edit and Continue

Native-only options

☑ Invoked by debug commands
☐ Ask first
☑ Warn about stale code
☑ Relink code changes after debugging
☑ Allow precompiling
☐ Enable while remote debugging or debugging an application running under another user account

3 What Problems Does STM Address?

We are now taking a break from working with samples in VS in order to present the basic premises and value proposition of STM. Once the marketing pitch is over, we will resume with coverage of features and samples. If you are familiar with STM you may wish to skip to the next section.

3.1 The Crisis

By now you'd be hard-pressed to find anyone in the industry who hasn't heard about the coming crisis in software performance. The rate of increase in processor speed has slowed dramatically and is nowhere near the doubling of speed that we saw every two years during the last twenty years.

That's the bad news; the good news is that Moore's law is by no means dead. The hardware industry is still able to follow the path that Moore predicted. The density of transistors, in an integrated circuit, still doubles every two years. The industry has, however, run out of effective ideas for turning those transistors into faster processors. Therefore, they are using the extra transistors to create *more* processors, building *chip multiprocessors (CMP)* with many processors on a single chip. Those chips are also known as multi-cores, or many-cores.

This is difficult news for the software industry. We can no longer add new features to our applications and rely on increasing hardware capacity to make evermore complex software perform acceptably. To state things more positively, we can also no longer rely just on better hardware to bring applications that were too expensive in the past to become feasible today (as was the case with, for example, speech recognition). Now, to take advantage of future hardware capacity, we have to fundamentally change the way we program, to exploit the parallel capacity of this new hardware.

Parallel programming is notoriously difficult. Researchers have been working for decades on systems to make parallel programming easier, yet despite these efforts, parallel programming has not yet become popular with all developers. Some amount of skepticism about whether we'll succeed now is therefore justified – but note that in the past parallel programming was a niche market, if only because of the expense of multiprocessor computers. The hardware industry has the capability to produce chip multiprocessors whose core counts double in time-frames similar to those we have been used to for processor speeds – whether they will do so depends on whether anyone will buy them which, in turn, depends on whether there is any software that utilizes their power to do interesting things that couldn't be reasonably done on a single-processor machine. If we succeed in creating such software, the virtuous cycle of new hardware and software features and applications will continue.

In order to enable this virtuous cycle Microsoft is making an effort to make effective parallel programming easier. There are many competing ideas on how this might be accomplished. Some are fairly radical: "get everyone to use pure functional languages," for example. Here we focus on one of these ideas, Transactional Memory. Compared to some other ideas in this space, it is evolutionary rather than revolutionary; programming with transactions will feel fairly similar to programming today, just simplified.

This section of the guide is intended to give an introduction to transactional memory in general. After introducing the basics we'll move to discuss the design and use of Software Transactional Memory (STM) in the .NET Framework in particular.

3.2 What Are Transactions? Why Are They Useful?

Transactions (like many of the best ideas in computing) originated in the data management world – in fact, they're one of the foundations of database technology. Full-scale transactions are defined by the *ACID* properties*,* of which memory transactions use two: they are *atomic* (they either happen entirely or not at all) and *isolated* (if transactions execute concurrently, their composite effects are as if the individual transactions ran serially in some order).

We should be very clear that this is a different problem from either (a) designing a parallel algorithm, or (b) coding that algorithm in a way that exposes parallelism. Task (a) is a conceptual task – given the problem at hand, you must decide what subtasks can be run at the same time, and which may only be run after others. Task (b) is a programming language issue – if you know that some set of subtasks should be run in parallel, how do you express it in your programming language or libraries? This is the problem that in .NET Framework 4 the new *Task Parallel Library* addresses – the TPL provides parallel iteration constructs, where loop iterations are performed in parallel. The bodies of these iterations may introduce new, nested parallelism.

When you've created parallel tasks, you have no problem until the tasks exhibit *interference*, where two tasks access some shared object, and at least one of them writes to it. Perhaps the shared object has some associated invariant (as in our Hello World example), and the writing transaction takes it from one state satisfying this invariant to another – but via some intermediate state in which the invariant is violated. Without some mechanism for preventing or handling interference, a reader might observe the object in this invariant-violating intermediate state, with potentially disastrous results.

3.3 Problems with Locking

The traditional mechanism for preventing interference is *locking;* we associate a lock with the object, and specify a protocol in which threads acquire the lock before accessing the object. In the Hello World example above, the watchdog thread would acquire the lock to prevent the object from being observed in the intermediate state; the "setter" threads would hold the lock while they are modifying the object. They would have to complete their updates to restore the invariant before releasing the lock. The reader "watchdog" thread would obey a similar protocol.

This is the current state of the art; this is what is available and recommended in languages like C# or Visual Basic.NET. But there are many problems with using locks.

3.3.1 Lock Accounting

The first major problem associated with locking is hidden in the description above: "we <u>associate a lock</u>…and *specify a protocol…"*   The association between data and the lock that protects access to it is a matter of program convention. To the extent that these conventions are documented by good programmers, they are done as unchecked comments: systems for specifying and checking locking conventions are still very much programming language research.

There are many possible locking conventions. Some objects may be protected by locking "this;" in other cases, locking some single shared object may provide access to a large set of objects. In previous versions of the .NET Framework it was considered a best practice to export a "SyncRoot" method, and require callers to lock the resulting object rather than locking "this." Sometimes different locks protect different fields in the same object. A programmer striving to be careful in today's world faces this question every time he or she accesses a field of a shared object in a parallel program: does the current thread hold a lock that allows me to access that field? Failure to properly follow the locking protocol could allow a *race condition,* which is the worst kind of bug: intermittent (and possibly extremely rare), with consequences that may only be manifest millions of instructions after it occurs, and in some cases costs millions of dollars to fix if not caught on time, before software is deployed.

3.3.2 Deadlock

Deadlock is a fundamental problem in lock based programming. If thread 1 attempts to obtain lock A then lock B, but thread 2 attempts to obtain B then A, then they can fall into a "deadly embrace", in which each holds the lock that the other is attempting to acquire. The solution to this problem is well-known, too: have all threads acquire locks in the same order.

Often this dictum is easier stated than followed. It is common that large multi-threaded applications home-grow "lock-leveling" systems that try to enforce lock acquisition order: each lock is given a level, and a dynamic checking mode ensures that every thread acquires locks in (say) ascending level order. Such systems can be fairly effective in finding violations (at least to the extent that the system's test suite exercises all the possible orders of lock acquisition in parallel executions). And in many cases this checking could even be done statically, with sufficient annotations and checking tools. But they don't really give the programmer much help when a violation is detected! Lock X protects data D, and I change some method BB to access data D, and therefore acquire lock X to do so – but the method BB I'm modifying is called by method FF, which holds lock Y, which we'd previously decided must be acquired *after* X – what should I do? One common, sometimes disastrous error in such situations is to modify FF to temporarily release lock Y, just for the duration of the call to BB. If FF held Y in order to modify some object, this may allow that object to be visible in an inconsistent state in which its invariants are violated.

In short, specifying a lock order is a difficult global problem, requiring the programmer to understand interactions between completely separate parts of a program, including implementation details of libraries that ought to be invisible to the caller. (Consider a container class used in a concurrent setting, which exports "Map" function that applies a delegate to all elements of the container. How does locking that protects access to the container class interact with locking that might be required by the delegate?)

Further, lock order schemes have generally been applied to systems with *static* locks, in which a lock may be named statically. This works in some cases. However object-oriented methodologies lead naturally to more dynamic locking, in which there may be many instances of a class, each protected by its own lock. If an instance of class A may point to an instance of B, and both of these have associated locks, the lock ordering has to match with “what points to what”. If pointer cycles are possible, or if “what points to what” changes over time, this may not work.

In conclusion, avoiding deadlocks in a parallel system is subtle and vexing.

3.3.3 Lack of Composition

The deadlock and locking discipline problems illustrate how in order to compose separately authored thread-safe components, a developer needs to understand these components’ *locking implementation details*, in a manner that violates good software abstraction practices. For example, consider a developer who is provided with a thread-safe dictionary class, and would like to build a bi-directional dictionary using two dictionaries. In order to make the bi-directional dictionary thread safe, the developer will need to either coarsen the locks used (and in the process lose scalability) or be given access to the implementation details of the dictionary, so that she’ll be able to reuse those locks in order to provide the higher order operations on the composite data structure. Even if the locking details are externalized, it is still not guaranteed that the original locking discipline could be extended in a manner that is deadlock free.

In all likelihood, composing the data structures in a safe and scalable manner is going to as complicated as building the composite data structure from scratch, thus losing all the advantages of software reuse and componentization.

4 Atomic Block to the Rescue: Basic Concepts

In the previous section we have introduced the problems associated with the current solution to *shared state management*. In this section we will go into the concept of the *atomic block* in detail, which is the mechanism we use for providing transactional memory support in .NET. We will explain how to use it and how it addresses the problems that were raised in the previous section.

The basic idea for adding transactions to a language is the *atomic block* construct*:*

```
atomic { body }
```

The system would promise to execute *body* atomically and in isolation – or at least provide those semantics. To a first approximation, programmers can think about an atomic block as if it acquires a single unnamed lock shared by all atomic blocks – the result will be as if no other atomic block was running while the current one was executing. Of course, the “as if” was very important in that statement – the underlying implementation will provide significantly more concurrency.

It is important to note that “execute *body* atomically” means that everything (transitively) executed within *body* is part of the transaction; if *body* makes calls, the called methods become part of the transaction. Transactions are lexically scoped but their execution proceeds dynamically while in scope.

The atomic block is considerably more abstract than locking, and immediately solves both of the problems with locking described above. Lock accounting is no longer an issue – in a parallel program using transactions ubiquitously, there is, conceptually, only a single lock that protects *all* data. Instead of worrying about what lock needs to be held to access a shared field, we need only worry about *whether* the field is shared, and, if so, whether we're currently within a transaction. We still have to worry about race conditions, but this now means concurrent transactional/non-transactional access to code. Dynamic race-detection tools become simpler and more effective, since there's only a single lock for them to worry about.

Similarly, deadlock becomes a non-issue: since there is only a single "lock" to acquire (when we think of an atomic block as abstractly taking a single big lock), there is no question of lock order. (If you're worried about what happens when we attempt to start a transaction within a transaction, see the explanation of *transaction nesting* below – the short answer is that only the outermost atomic block has any effect, and the inner ones are, for most purposes, semantic no-ops.)

Getting rid of the accounting and deadlock issues makes parallel programming considerably simpler. The only issues become:

1. Am I accessing possibly-shared data?
2. Does that data have some invariant that might be violated?
3. If so, enclose my access in an atomic block – if I'm a writer, make sure the invariant holds before exit from the atomic block.

4.1 Failure Atomicity

In the database world, the programmer has a way to specify that the current work being done against the database (the transaction) should be cancelled and its effects *rolled back*. When designing the "atomic" block we had either the option to provide similar behavior to that of database transactions, i.e., allow the user to abort them, or alternatively we could have kept with the "single lock" semantics, which don't offer any sort of rollback.

Our current "atomic" block implementation does allow the user to abort a transaction. This is done by letting an exception escape the boundaries of an atomic block. Thus, the following two sequences are not equivalent:

```
atomic {
    m_x++;
    m_y--;
    throw new MyException()
}
```

```
lock (GlobalStmLock) {
    m_x++;
    m_y--;
    throw new MyException()
}
```

While these two samples are equivalent from isolation perspective, the one on the left will result in no change to the variables modified while the one on the right will obviously externalize the side effects to the variables as soon as the lock is released.

Providing **database-style failure atomicity is an extremely powerful tool**. This cannot be stressed enough. It provides a much more robust mechanism for handling failures, instead of the fragile and untested “catch” handlers that developers use today with manual error recovery code. STM provides here a great deal of value which is not only available to concurrent code, but to any code that requires a high level of reliability.

4.2 Transaction Nesting

Since atomic blocks are just blocks in the code, they can be nested, either lexically within one method or dynamically when one method calls another one. What happens when transactions nest? Here’s a standard example that leads to this question. Let’s say we have a *BankAccount* abstraction, with a *Balance* property and a *ModifyBalance* operation (for simplicity, we’ll use this for both deposits and withdrawals). If *BankAccounts* may be accessed concurrently, we’d want to put the body of the *ModifyBalance* method inside an atomic block (just as we would use a lock to synchronize such access in a lock-based program):

```
class BankAccount
{
    private int m_balance;
    public void ModifyBalance(int amount) {
       atomic {
           m_balance = m_balance + amount;
           if (m_balance < 0)
              throw new OverdraftException(m_balance, amount);
       }
   }
}
```

Note that if there’s an overdraft the balance modification is undone. Yes, this was deliberately and cutely written to illustrate the failure atomicity feature – the consistency check is done *after* the balance modification and then the modification is undone when the transaction is aborted due the exception thrown. It’s convenient to be able to notice an error at any point, and know that your tentative changes will be discarded.

Now consider another operation we might find useful:

```
public static void Transfer(BankAccount from,
                            BankAccount to,
                            int amount)
```

Obviously, we would like the balance transfer to be atomic: we want it to happen “all-or-nothing,” and we don’t want a concurrent operation to “steal” the money from the “from” account.
We can conveniently achieve this with another, higher-level, transaction.

```
public static void Transfer(BankAccount from,
                            BankAccount to,
                            int amount) {
    atomic {
        from.ModifyBalance(-amount);
        to.ModifyBalance(amount);
    }
}
```

Now our question is what does it mean to execute the nested transaction inside ModifyBalance, when it's being called from within the enclosing transaction in Transfer? We definitely do not want the effects of the withdrawal to be globally visible as long as the entire transfer is not complete, so we get the following rule for nested transactions:

From isolation perspective, nested transactions are flattened into their parent.

On the other hand, failure due to an over-draft in ModifyBalance doesn't necessarily mean that the entire Transfer operation needs to be fail. Perhaps Transfer can be coded such that it handles the failure. For example:

```
public static void Transfer(BankAccount from,
                            BankAccount fromBackup,
                            BankAccount to,
                            int amount) {
    atomic {
        try {
            from.ModifyBalance(-amount);
        }
        catch (OverdraftException) {
            fromBackup.ModifyBalance(-amount);
        }
        to.ModifyBalance(amount);
    }
}
```

Thus, we arrive at the following additional rule for nested transactions:

Nested transactions fail independently of their parents.
When they abort, only the side effects that they have affected are rolled back.

This property is commonly referred to: *partial rollback*.

4.3 The System.TransactionalMemory Namespace

STM.NET adds the System.TransactionalMemory namespace to the .NET Base Class Library (BCL). In this namespace you would find all the API's that are necessary for using STM. In the fullest of time, most uses of STM will not require invoking any API—the "atomic" block is all that is typically required.

However as we have noted before, we currently do not have C# or Visual Basic.Net integration and thus instead of writing

```
atomic { <body> }
```

One uses delegates:

```
Atomic.Do( () => { <body> } );
```

We feel this is a user-friendlier version of the following; albeit at a minor loss of performance; both syntaxes prove the atomic-block functionality

```
try { <body> } catch {AtomicMarker} {}
```

The class Atomic resides in the System.TransactionalMemory namespace and it allows the usage of STM with delegates, anonymous methods, closures, etc. Given what you know already about STM, the implementation of Atomic.Do should appear trivial by now:

```
namespace System.TransactionalMemory {
    ...
    public static class Atomic {
        ...
        public static void Do(AtomicAction action) {
            try {
                action();
            }
            catch (AtomicMarker) {}
        }
    }
}
```

4.4 How Is STM implemented? What Does It Cost?

STM can be implemented in a myriad of ways, including using special hardware support that some hardware vendors have been considering to accelerate a pure software implementation. STM can be almost trivially implemented using a global lock however this would provide very bad scaling and will not supply failure atomicity.

In general, STM implementations work by instrumenting the program's access to shared state. In our implementation of STM, this instrumentation is done by the *Just-in-Time compiler* (CLR JIT). When a piece of code executes inside a transaction, the JIT generates a special version of the code that does the right thing when objects are read or written into.

Isolation is achieved by associating thin locks with objects, statics, etc. at runtime. Transactional locks support optimistic concurrency control. i.e., instead of acquiring locks pessimistically a transaction may inspect a lock, proceed with its computation and then recheck that the lock has not changed state before commit time.

Finally transactions have the ability to transparently *rollback* and *re-execute*, most notably due to contention on transactional locks. In the “atomic” block programming model, this event is mostly transparent to the programmer (although it can be debugged and profiled as an interesting event from a performance perspective).

As you can imagine the cost of all the instrumentation and fine-grained locking is significant. We are currently measuring anywhere between 2x and 7x serial slowdown compared to lock-based code. There are additional details in section 0 that discusses performance. It should be noted though that our current STM implementation hasn’t undergone a significant amount of optimization. There is still a lot of room for improvement and of course hardware support, if and when it becomes available, will dramatically change the picture.

4.5 Putting the Basics Together: BankAccount Sample

In your installation’s directory go to Samples\Features and open the Features.csproj VS project. This solution contains small feature samples that demonstrate the material discussed in this guide. Each feature sample corresponds to a method in the Features class in Features.cs so you should be able to quickly locate the source code for each feature. You select which feature to run using the command line interface.

The next sample to “HelloWorld” is “BankAccount” which puts into code the discussion from the previous section regarding the Atomic.Do API, transaction nesting and failure atomicity. This is how the Transfer method is defined in the sample:

```
public static void Transfer(BankAccount from,
                            BankAccount backup,
                            BankAccount to,
                            int amount)
{
    Atomic.Do(() =>
    {
        // Be optimistic, credit the beneficiary first
        to.ModifyBalance(amount);
        // Find the appropriate funds in source accounts
        try
        {
            from.ModifyBalance(-amount);
        }
        catch (OverdraftException)
        {
            backup.ModifyBalance(-amount);
        }
    });
}
```

Recall that each ModifyBalance operation is in itself a small transaction, thus if it fails, it leaves no modifications behind. To further illustrate how failure atomicity works, we first credit the “to” account, even though the other accounts may not have sufficient funds for the transfer. We then try to withdraw from the “from” account. If this fails, we try to withdraw from the “backup” account. If that fails too, the OverdraftException will escape the top-level transaction boundary and the crediting of the “to” account will be rolled back. The entire transaction will be aborted and the exception will surface to the surrounding code.

5 Writing Correctly Synchronized Code

We have briefly discussed how “atomic” blocks resemble taking a global lock. Well, even with a global lock it is still possible to make the mistake of not using the lock at all! A failure to synchronize, results in a *race condition*.

Very informally, a race condition within the context of STM is a situation where the same data can be accessed in a conflicting manner both inside and outside transactions at the same time. “Conflicting manner” means that at least one of the accesses is a write (reading a piece of data simultaneously inside and outside a transaction is not considered a conflict).

Let’s consider an example:

```
Initially:
static int X = 0;
static bool X_Shared = false;
```

```
1. // “Publisher” thread

2. X = 42;
3. atomic {
4.     X_Shared = true;
5. }
```

```
1. // “Consumer” thread

2. int fetchedX;
3. bool fetchedXValid;

4. atomic {
5.     fetchedX = X;
6.     fetchedXValid = X_Shared;
7. }
```

This program contains two pieces of shared data, or potentially shared data: the variables X and X_Shared. It is very easy to see that X_Shared is correctly protected throughout the program; it is always accessed within a transaction.

However when we examine accesses to X we see that the “publisher” thread writes to it outside of a transaction while the “consumer” thread reads it inside a transaction. This still doesn’t necessarily mean that the program is racy. For the program to be racy there needs to be a condition under which those two conflicting accesses are *concurrent*.

How do we determine whether two accesses could be concurrent? We pretend that we are the OS scheduler and the computer executing these pieces of code, and we observe whether at any given moment we can get to a situation where both instructions are *enabled for execution*. If that’s the case, then we have a race. In our concrete example, suppose the scheduler executes the publisher up till instruction #2, where the program wishes to write into X. Then it switches to the consumer thread, takes the "global lock" associated with the atomic block in instruction number 4, and then *voila!* we arrive at instruction #5 that wishes to read the value of X. Where we can see that the:

1. Next instruction for the "publisher" is "write X".
2. Next instruction for the "consumer" is "read X".

Thus, we reach the conclusion that this program has a race condition on X since both threads have conflicting operations on shared data. So this program is racy and may result in unexpected results. Note that this observation has nothing to do with the hardware used, how many cores it has, how fast it is, etc.

Having or not having a race in a program is an intrinsic property of the program!

Many races are benign and never materialize. Some programmers deliberately code races into to their programs as a means to optimize them. For example, the Double-Checked Locking (DCL) pattern is racy under the above definition, but it may still work for locks and a given *memory model*. However, patterns such as DCL are not supported under STM.NET.

Whether programs that are racy under the above definition of races work as expected or not depend on a specification of the system referred to as the system's *memory model*. Memory models can be quite non-portable, arcane and difficult to reason about and therefore we encourage our users to stick with the definition of race freedom that was brought here. This will save you a lot of nasty and hard to find concurrency bugs in your code.

In addition to the general benefits of race-free design, when you're working with STM, it is even more important since:

STM provides single lock semantics only to race free programs

Previously in this guide you were told that an "atomic" block is very similar to a locked region with a global lock. Well, that wasn't the entire truth. This is true only for race free programs.

What if your program contains races? In that case the semantics of your program as still well defined, but are much harder to explain and understand so we will not do so now. Further information on the topic can be found in [11][12]. But let us repeat our advice: keep your program race free!

5.1 Ensuring Race Freedom

Races happen when a piece of data is accessed concurrently inside and outside of transactions. In this section we will describe successful patterns for avoiding such races.

5.1.1 Follow Standard Data Hiding Practices

First off, following standard data hiding and abstraction practices will make your design problem easier. The important thing to get right is to protect all access paths to shared data. It should be a design goal to "firewall" all access paths to shared data that is managed by a type such that it is sufficient to reason about the type in order to ensure race freedom for the data it encapsulates. Race freedom is just another invariant that the type enforces over the data that is encapsulated inside it and is "owned" by it.

5.1.2 Synchronize All Accesses to Shared Data

One conservative approach to providing correct synchronization is to make sure all accesses to the data are always synchronized. For example, in the Bank Account sample, the BankAccount type introduces transactions around all of its public methods and thus the underlying data (the m_balance) field is never accessed outside of transactions.

This is a safe practice but it doesn't allow taking advantage of thread locality. i.e., perhaps only a single thread has access to a BankAccount object sometimes and then it doesn't need to pay the cost of transactions to access the object. Introducing a transaction in such situation is wasteful, but safe.

5.1.3 Use Structured Parallelism

Structured parallelism makes concurrency both explicit and scoped and as such provides a great framework for reasoning about the "sharedness" of data. The .NET Framework 4 currently in Beta1 contains the Parallel Extensions to .NET which is a set of libraries offering both structured and unstructured API's for parallelism.

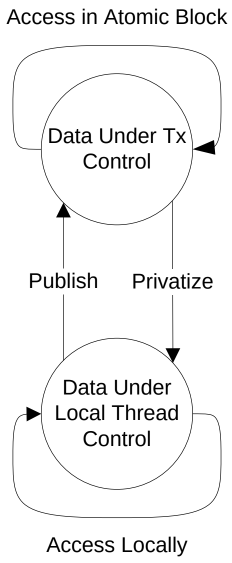
The best example of unstructured parallelism is the Thread API. Once you create a thread, it is executing independently of the code that has created it. You have to explicitly wait for the thread's completion in order to join the current activity with the parallel activity that was carried out by the thread.

The best example of structured parallelism is the Parallel.For API which takes care of spawning and joining the parallel activities such that when the Parallel.For API returns, the programmer knows that all parallel tasks have completed.

With structured parallelism, you get very strong guarantees on task completion and can thus reason about "publishing" data as you enter the API and "privatizing" the data as you exit it, all due to the internal coordination and scoping of the API.

5.1.4 Publishing and Privatizing Data

If it is important for you to take advantage of thread locality of objects, then the patterns of "publication" and "privatization" may be useful. You can code the publication and privatization patterns in a race-free and safe manner using STM. The pattern is depicted graphically in the next figure:



The act of publishing or privatizing is done through modification of some state that is always accessed in a synchronized manner, and it indicates the state of the data that is subject to moving between local and shard states.

Consider for example a data structure that contains many records. Usually the data structure is “online” and threads inspect it using transactions. However once in a while there is a lengthy operation that needs to take place that scans the entire data structure. The data structure is then brought “offline” (privatized), it is then accessed in a local manner, and when this is done, it is published again. The PrivatizePublish feature sample shows this pattern in code. Here is the piece of code that demonstrates both privatization and publication:

```
// Class declarations
static int[] items = new int[DataStructureSize];
```

```
// Privatize
int[] localItems = null;
Atomic.Do(() =>
{
    localItems = items;
    items = null;
});

// Now can work on this locally.
Array.Sort(localItems);

// Local work finished, let's publish
Atomic.Do(() => items = localItems);
```

The piece of information that is always shared is the static reference “items”. However the array this static reference points to transitions from shared to local, using the transactions in the above code snippet.

Of course, this means that transactions that wish to inspect the array need to first confirm that it is indeed shared. This is demonstrated in the following code snippet:

```
// Transactional modification, conditioned on the existence
// of a published items array.
//
// The Retry sample that we will show later demonstrates
// how to wait for the availability of such data.
Atomic.Do(() =>
{
    int[] items = SampleDriver.items;
    if (items != null)
    {
        int temp = items[toIdx];
        items[toIdx] = items[fromIdx];
        items[fromIdx] = temp;
    }
});
```

In this example, if the data is not available, nothing is done. Sometimes you must wait until the data is available and then apply the operation. This is done in transactions using the “retry” operation, which we will discuss in section 7.

5.2 Granularity

In the definition of race conditions we have introduced in this section, we said that it is illegal to access the same data concurrently in a conflicting manner inside and outside of transactions. We spend a good deal of time explaining what “concurrently” means, but we still have to define what “same data” means here. This is where the question of *data granularity* comes into consideration.

The CLR memory model defines some rules regarding the atomicity of reading and writing data (load and stores), as a function of the data types. For example a load or a store of an integer or a reference is always atomic. You’d never get an odd mixture of bits. However with double precision floating point values, or other “big” data types, atomicity is not guaranteed and you may be able to notice a value that is the result of an odd mixing of concurrent stores.

Similarly, STM.NET, too, defines rules of granularity of accesses that are supported versus those that are *racy*. Unsafe code and pointers can really muddy the water. Suppose you have a static variable of type int, and you obtain two byte* pointers into the integer, one pointing into the first byte of the variable and the other pointing into the second byte of the integer. Then suppose the first byte is modified within transactions, and the second byte is modified outside of transactions. Would this be a race? Do the concurrent activities in this example modify the same piece of data?

The answer to this question depends on the (hopefully precise) definition of granularity that we provide here[4]. There are basically two classes of shared storage in the CLR, one is static fields[5] and the other is heap objects. So what are the rules for these two classes?

> ***Objects****: conflicting concurrent access to any part of the same object is considered racy.*
>
> ***Static fields****: conflicting concurrent access to any part of the same static field is considered racy.*

Let's consider some examples. Suppose we have the following two types:

```
public struct S {
   public int fs1;
   public int fs2;

   public static int si;
}
```

```
public class  C {
    public int fc1;
    public int fc2;

    public S sdc1;
    public S sdc2;

    static public S sfc1;
    static public S sfc2;
}
```

Also assume that we have created an instance of class C and that it's referenced through the variable myC. Now let's consider pairs of accesses where at least one of the accesses is a write and furthermore where one is transactional and the other isn't and see whether they are considered racy or not:

myC.fc1	myC.fc2	**Racy**. Accesses the same object concurrently.
myC.sdc1.fs1	myC.sdc2.fs2	**Racy.** The accesses are to the same heap object (referenced by myC). In .NET, structs are "inlined" into their containers.
C.sfc1	C.sfc2	**Not racy**. Accesses two distinct static fields.
C.sfc1.fs1	C.sfc1.fs2	**Racy**. Accesses the same static field.
byte * p1 = (byte*)&S.si // Dereference p1	Byte *p2 = 1 + (byte*)&S.si // Dereference p2	**Racy**. p1 and p2 point into the same static field.
byte * p1 = (byte*)&myC.fc1 // Dereference p1	Byte *p2 = (byte*)&myC.fc2 // Dereference p2	**Racy**. p1 and p2 point into the same heap object.

[4] It is important to note that the definitions that we have chosen have far reaching ramifications for performance. In an ideal world we may have chosen the unit of granularity to be a single bit, or short of that, a single byte, but that would make bookkeeping extremely costly.

[5] But not *thread static*! These variables are local to each thread.

5.2.1 Bad Effects of Races due to Granularity Violations

If your program contains some violations of the granularity rules just explained in the previous subsection, this can result in the transactional code over-writing modifications done by non-transactional code. The reason for this effect is that the STM.NET implementation maintains *shadow copies* that contain tentative changes made to static fields and objects. The granularity of shadow copies is generally the one that is described in the rules above, meaning one for each object modified and one for each static field modified. When the transaction commits, the contents of shadow copies are written out. However, not only the changed pieces of data inside the shadow copy are copied out but rather the entire shadow copy is copied out indiscriminately onto the master location. This causes no problems if all writes are protected by transactions. However when a write is not protected by a transaction, it can be overwritten.

5.2.2 Avoiding Granularity Problems

Aside from unsafe code and managed unions, which are not recommended practices anyway, the main problems due to granularity will ensue when some fields of an object are modified within transactions while other fields of the same object are modified outside of transactions, presumably by a distinguished thread that has knowledge that it can access said fields “privately”. The best way to eliminate the problem in such case is to break-up the object into two sub-objects. This will not only provide protection from granularity hazards but also better capture the program’s intent and design.

6 Atomic Compatibility Contracts

In previous sections we have described how atomic blocks provide automatic isolation and how they sometimes need to roll back due to various reasons. To provide these properties code that runs inside atomic blocks needs to be able to be instrumented by the runtime to provide both (a) isolation and (b) failure atomicity.

Verifiable managed code lends itself to instrumentation in a straight forward manner such that the JIT in the CLR version of STM.NET is capable of automatically generating an atomic version of the code. However (some) unsafe code, p/invoke and other forms of native interop are not “visible” to the JIT and thus atomic behavior cannot be automatically inserted in such cases. We will describe in section 10 the mechanisms that are available for “taming” such code such that it could be used inside atomic blocks.

However, there are always going to be cases where isolation and failure atomicity cannot be provided, or it would be prohibitively complex or costly to do so. One must also be cognizant about adding atomic support to public API’s as this is adding another dimension of supported usage that becomes part of the API’s contract and thus cannot be retracted easily in the future.

Thus, some services will be supported in atomic blocks, while others won’t be. Some other services will be available *only* inside atomic blocks. How do we codify that such that we can check for errors? We do so by using *atomic compatibility attributes*:

- **AtomicSupportedAttribute**. Signifies that the given item may be accessed both inside and outside of atomic blocks[6].
- **AtomicNotSupportedAttribute**. Signifies that the given item can be accessed only outside of atomic blocks.
- **AtomicRequiredAttribute**. Signifies that the given item can be accessed only within atomic blocks.

There is also a fourth "contract" which is basically an escape hatch:

- **AtomicUncheckedAttribute**. Signifies that no static checking of compatibility is attempted for the given item.

These contracts can be applied to assemblies, methods and fields, with the restrictions captured in the following table:

Item\Mode	AtomicSupported	AtomicNotSupported	AtomicRequired	AtomicUnchecked
Assembly	X	X	X	-
Method	X	X	X	-
Field	X	-	X	-
Delegate Type	X	X	X	X

We will now describe what each contract means on each item.

6.1 Contract Assignment to Metadata Items

Each method, field, and delegate type has a unique and unambiguous atomic compatibility value assigned to it. We will refer to this value as the *effective atomic compatibility value* of the item. This value is determined by a set of rules that take into account the nature of the item (e.g., whether the method is extern or not) and the set of explicit compatibility annotations that exist on the item and on the assembly the item is defined within.

Once we have defined how contracts are assigned to these code elements we can start considering what actions are legal or illegal in the code that you write. Our overall goal is to devise a static annotations system that warns the user at build time of errors such as calling a method that has the AtomicNotSupported contract inside an atomic block or calling a method that requires an atomic block, outside of an atomic block.

[6] In this section, when we say inside or outside of atomic blocks we mean dynamically so. For example if method F1 has an atomic block in which it calls method F2 and F2 invokes method F3 then F3 is "invoked inside an atomic block"—the one introduced by F1.

6.1.1 Assembly Level Contract

The assembly level contract controls a single thing: the default contract for methods that don't have other rules apply to them. The default can be set to either AtomicSupported, or AtomicNotSupported. The default-default that applies in the case the assembly doesn't have a compatibility attribute is AtomicNotSupported.

6.1.2 Method Level Contract

The compatibility value of a method controls which contexts (see Section 6.2) the method may be invoked from and which delegate types can be constructed from the method.

An explicit compatibility contract may be applied to methods.

Non extern methods that are not explicitly annotated with a compatibility contract follow the assembly level compatibility value.

Extern methods are by default AtomicNotSupported, regardless of the assembly level contract. We will discuss in section 10.2 methods of making external methods supported, but let's assume for now that they are all AtomicNotSupported. It is illegal to just mark an extern method with AtomicSupported, without using one of the advanced attributes described in section 10.

6.1.3 Field Level Contract

The compatibility value of a field controls which code contexts could access the field.

Fields do not follow assembly level contract. The default contract for fields is always AtomicSupported.

Only fields of classes may have an explicit contract assigned to them. Fields of structs are always considered AtomicSupported and it is illegal to annotate them with a compatibility contract.

6.1.4 Delegate Type Contract

The compatibility value of a delegate type controls both what methods could be used to construct an instance of the given delegate type and which code contexts may invoke an instance of the given delegate type.

A compatibility attribute may be applied to a delegate type. The default for delegate types is AtomicUnchecked.

6.2 Atomic Context

At runtime, a thread can be either inside an atomic block, or outside of all atomic blocks. At build time, when we look at a particular piece of code, we may know exactly the circumstances that it will be invoked under at runtime, or we may know that several options are available. For example, when we author a method M and decorate it with AtomicRequired we know that no matter what, control will only ever enter M when the current thread has already entered an atomic block. This means that when we enter M we have a very strong guarantee on the runtime state, which allows us to further reason that we can call other[AtomicRequired methods, because we will be still maintaining the invariants that they require—that the thread is within a transaction.

At other occasions we know less about the state of the thread. When a method M has a contract of AtomicSupported we assert that M can be called either within transactions, or outside of transactions. Thus when we author the method, we can only safely do things that are allowed under both conditions. We may not call AtomicNotSupported methods, since M may be invoked inside a transaction, and we also may not call methods with an AtomicRequired contract, since M may be invoked outside of a transaction.

So let's formalize the concept of "what is known about the compatibility of a piece of code" as *atomic context*. Any IL instruction in any IL method has a unique and unambiguous atomic context value assigned to it from the following set:

- **AtomicContext.Always**. Control flow will always reach the instruction while the thread is in an atomic block.
- **AtomicContext.Never**. Control flow will never reach the instruction while the thread is in an atomic block.
- **AtomicContext.Unknown.** Control may reach the instruction while the thread is in an atomic block, but it may also reach the instruction when the thread is not within an atomic block.

In order to decide the atomic value of instructions we need to look at the method body that they're a part of, and at the effective compatibility value of the method. No other information is required.

The effective atomic compatibility value of the method determines the initial atomic context for the method, in a pretty straightforward manner:

- **AtomicRequired** methods start off with an **Always** atomic context.
- **AtomicNotSupported** methods start off with a **Never** atomic context.
- **AtomicSupported** methods start off with an **Unknown** atomic context.

At this point in our guide we have introduced a single point where the context changes:

- atomic blocks (encoded as try/catch(AtomicMarker)) induce an **Always** atomic context inside the try body.

In section 10 we will see other mechanisms that introduce context transitions.

6.3 Putting Contracts in Context

We are now finally ready to specify what operations are allowed inside what context.

6.3.1 Method Invocation

A method (including a constructor) can only be invoked from a context that is compatible with the method's effective compatibility value. This is captured in the following table:

Context\Contract	AtomicRequired	AtomicNotSupported	AtomicSupported
Always	X	-	X
Never	-	X	X
Unknown	-	-	X

An ‘X’ in the cell indicates that a method with the contract associated with the column can be called inside the context associated with the row. So AtomicRequired can be called inside an Always context, but not inside Never and Unknown contexts.

6.3.2 Field Access

Field access follows the same matrix that we have specified above for method invocation, except AtomicNotSupported is not allowed on fields.

6.3.3 Delegate Invocation

Recall that delegate types have four possible contracts assigned to them: AtomicSupported, AtomicNotSupported, AtomicRequired or AtomicUnchecked. The default for delegate types is AtomicUnchecked.

The rules for invoking a delegate are identical to those of method invocation and field access, except that it is always valid to invoke a delegate with a compatibility value of AtomicUnchecked. In case the method invoked doesn’t support the context it was invoked in, through the delegate, a dynamic exception will be raised. We will talk more about dynamic contract checking in section 6.6.

6.3.4 Delegate Instance Construction

When a delegate type that has a compatibility value other than AtomicUncheckedis constructed from a method we make sure that the assignment is valid such that the delegate could be invoked safely without breaking the assumptions of both the caller and callee. Here is an example demonstrating that:

```
[AtomicRequired]
public delegate void MyAction();

public class DelegateSample
{
    [AtomicRequired]
    public void M2(MyAction action)
    {
        action(); // Since MyAction is AtomicRequired, it is
                  // always safe to invoke it from Always context
    }

    [AtomicSupported]
    public void M1()
    {
    }

    public void Test()
    {
        MyAction action = M1; // OK. M1 can be invoked from
                              // whereever a AtomicRequired method
                              // is valid.
        try
        {
            M2(action); // OK. AtomicRequired in Always context
        }
        catch (AtomicMarker) { }
    }
}
```

Following the same principle for all contexts and compatibility values we arrive at the following construction validity rules for delegates.

Delegate Contract	Method Contract		
	AtomicRequired	AtomicNotSupported	AtomicSupported
AtomicUnchecked	X	X	X
AtomicRequired	X	-	X
AtomicNotSupported	-	X	X
AtomicSupported	-	-	X

To intuit the above table:

- Assignment when the contract as identical is always legal
- AtomicSupported methods may be invoked in any context, and thus can be assigned to the delegate regardless of the delegate contract.
- Delegate types with unchecked contracts substitute static checking with runtime checking, so any method can be assigned to them, regardless of the method’s compatibility contract.

6.4 Polymorphism

Whenever virtual or interface methods bind themselves to a particular contract, derivations and implementations of the given method must provide the contract that the base class or interface has committed to. If a derived method is unable to provide the contract that the base obliged it to, then it can choose to throw NotSupportedException in case it is invoked in an atomic context it doesn’t support. AtomicRedirect, described in section 10.2, provides a good way of doing that in a statically verifiable manner.

6.5 Contracts Sample

Contracts.cs in the Features project contains an annotated sample demonstrating the correct use of atomic contract annotations.

6.6 Static Checking for Contract Compatibility

All the contract compatibility rules stem from three following basic rules:

- Code pieces that cannot be correctly transacted by the STM system should never be executed inside a transaction. This includes calling external methods and some forms of unsafe code.
- A language element that requires a transactional context of **Always** (e.g. fields with the AtomicRequired contract) should not be accessed outside of a transaction.
- A language element that requires a transactional context of **Never** (e.g. an AtomicNotSupported method) should not be accessed inside a transaction.

Any program that uses our STM system should adhere to these rules in order for it to work correctly.

If we have these three basic rules, then why did we feel the need to invent a more elaborate contract checking scheme? These three rules can be verified at runtime since we always know the AtomicContext in which the thread is currently executing and the AtomicContract of the field or method that is currently being accessed by the thread. However, due to the language features such as polymorphism and delegates, at compile time we do not always know these two pieces of information.

Our contract compatibility scheme helps fill this gap. It allows you to formally specify contracts on the different language elements. Once these contracts are in place our static checking tool, TxCop, can verify whether your assembly is correctly using transactions.

6.6.1 Static Checking Errors

This section describes all the possible errors that can be generated during static checking.

Error Code	Error name	Explanation
TX0001	UncheckedContractOnMethod	AtomicUnchecked contract has been placed on a method. This is currently not supported by our system.
TX0002	MismatchFromInterface	The contract on an implementing method is not compatible with the contract on the interface method it is implementing. This violates the polymorphism rules we described in an earlier section.
TX0003	MismatchFromBaseMethod	The contract on an overriding method is not compatible with the contract on the virtual method it overrides. This again violates our polymorphism rules.
TX0004	MismatchOnInvoke	The contract on a method is incompatible with the context in which it is invoked.
TX0005	MismatchOnFieldAccess	The contract on a field is incompatible with the context in which it is accessed
TX0006	MismatchOnDelegateInvoke	The contract on a delegate type is incompatible with the context in which it is invoked
TX0007	MismatchOnDelegateAssignment	The contract of a method passed to a delegate constructor is incompatible with the contract of that delegate type. For more details please

Error Code	Error name	Explanation
		see the section that discusses delegate instance construction rules
TX1001	IncompatibleContractWithRedirect	A method with the AtomicRedirect attribute has an AtomicNotSupported contract on it. This is not allowed.
TX1002	IncompatibleContractWithSuppress	A method with the AtomicSuppress attribute has an AtomicNotSupported contract on it. This is not allowed.
TX1003	IncompatibleRedirectTarget	The target method specified in the AtomicRedirect attribute either does not exist or it has a signature (return type and formal parameters) that is different from the method on which the AtomicRedirect attribute is placed
TX1004	IncompatibleContractOnRedirectTarget	The target method specified in the AtomicRedirect attribute has the AtomicNotSupported contract, which makes it un-callable inside transactions.
TX1005	RedirectGenericNotImplemented	We have not implemented redirect support for generic methods. So whenever an AtomicRedirect contract is placed on a generic method this error will be generated.
TX1006	SuppressOnNativeMethod	AtomicSuppress attribute is placed on a native method. We have not implemented support for this capability. AtomicSuppress can only be used with managed methods. A workaround to using AtomicSuppress on native methods is given in section 10.
TX1007	AtomicMarshallOnIncompatibleParameter	AtomicMarshalReadonly attribute is placed on a ref, out, or pointer parameter. Since adding this attribute creates a deep copy of the parameter, usage with pointer/ref/out parameters is incorrect.
TX2001	IncompatibleContractOnStaticInitializer	A type constructor has an AtomicRequired contract on it.
TX2002	IncompatibleContractOnFinalizer	A finalize method has an AtomicRequired contract on it.
TX2003	IncompatibleAttributeOnAtomicDo	This is only applicable to mscorlib authors. It is raised when an AtomicSuppress or AtomicRedirect contract is placed on Atomic.Do
TX2004	SpecialMethodAccessesConflictingElements	An anonymous method, closure or another compiler generated construct accesses methods or fields with AtomicRequired and

Error Code	Error name	Explanation
		AtomicNotSupported contracts
TX2005	InvocationThroughUnmgedFnPtr	A method is invoked using the calli instruction i.e. through an unmanaged function pointer.
TX3001	MultipleContracts	A field, method or delegate type has multiple contracts on it.
TX3002	RequiredContractWithRedirect	AtomicRedirect and AtomicRequired are placed on the same method. The method is possibly dead code

Our static checking scheme is conservative so there is room for you to ignore some of the errors that are generated. For instance, let’s say you've implemented an interface method. The interface method has a contract of AtomicSupported. Now according to the polymorphism contract checking rules the implementing method that you’ve written should also have the AtomicSupported contract, so that it can be invoked in all the contexts that the interface method is invoked. However, you annotate the method as AtomicRequired since you expect it to be always be invoked inside a transaction.

TxCop, our static checking tool, will flag the AtomicRequired contract on your method as an error. However, given your understanding that this method will never be used outside of a transaction, it is safe for you to execute your program.

Broadly, our polymorphism errors serve as warnings and can be ignored when the programmer thinks it is safe to do so. However, ignoring any of the other errors generated by our static checker is likely to lead to runtime exception.

6.6.2 Using the static checker

Our static checking tool is called TxCop. There are two ways in which you can use it: via the command line or as a post-build step in visual studio. I’ll describe both ways of using the tool in this section.

6.6.2.1 Command-line usage

After you install STM .Net TxCop will be placed in your Program Files folder under Microsoft.Net\STM. To get the various options just type TxCop.exe /? on the command line. Doing so would give you the following output.

```
C:\Program Files\Microsoft.Net\STM>TxCop.exe /?
TxCop.exe
    /assembly <assembly>
    /pdb <symbols> (not required if pdb is colocated with the assembly)
    /reference <reference or comma seperated list of reference> (short form /r)
    /verbose specifies whether informational messages should be displayed
    /exclude <warning type>
```

The /assembly option is used to specify the assembly or executable you want to run the static checker on. The /pdb option is used to specify the location of symbols for the assembly you are checking.

/reference or /r is used to specify the assemblies that are referenced by the assembly you are checking. It is necessary to specify all the references. The /exclude options lets you turn-off static checking for a given error or warning type.

After running TxCop it will generate a list of errors and print them out on the console.

An example command line usage will look as follows.

```
C:\>TxCop.exe /assembly "C:\stm\samples\TraditionalTransactions.exe"   /r
"C:\mscorlib.dll" /r "C:\system.data.dll","C:\system.data.datasetextensions.dll"
/exclude MismatchFromInterface /exclude MismatchFromBaseMethod
```

An example output of running TxCop from the command line on assembly will look as shown below. The output gives two different categories of information.

The first two lines are informational messages (only displayed if /verbose is specified). These lines tell us that TxCop inferred a certain contract for a delegate or closure -- since delegates and closures cannot be annotated by our contracts, which are nothing but .Net attributes, we infer their contract based on the contract of the methods they invoke and the fields they access.

The last line gives an example of how a static checking error will look if TxCop is run from the command line. This specific error tells us that the contract of a method or closure passed to a delegate type's constructor is not compatible with the contract of the delegate type.

```
C:\STM.NET\TradTx.cs(189,17): Inferred contract [AtomicSupported] for [STMSamples.
TradTx.<Test8>].
C:\STM.NET\TradTx.cs(215,17): Inferred contract [AtomicNotSupported] for [STMSamples.
TradTx.<Test9>].

STM Static Checking Report:

C:\STM.NET\TradTx.cs(214,17): error TX0006: Incompatible contracts in delegate
assignment.[AtomicNotSupported]STMSamples.TradTx.<Test9> assigned to delegate
[AtomicRequired]System.TransactionalMemory.AtomicAction.
[MismatchOnDelegateAssignment]
```

6.6.2.2 Visual Studio integration

TxCop can also be added as a post-build step to Visual Studio. TxCop capabilities should ideally be a part of the compiler. Adding TxCop as a post-build step comes close to mimicking this behavior.

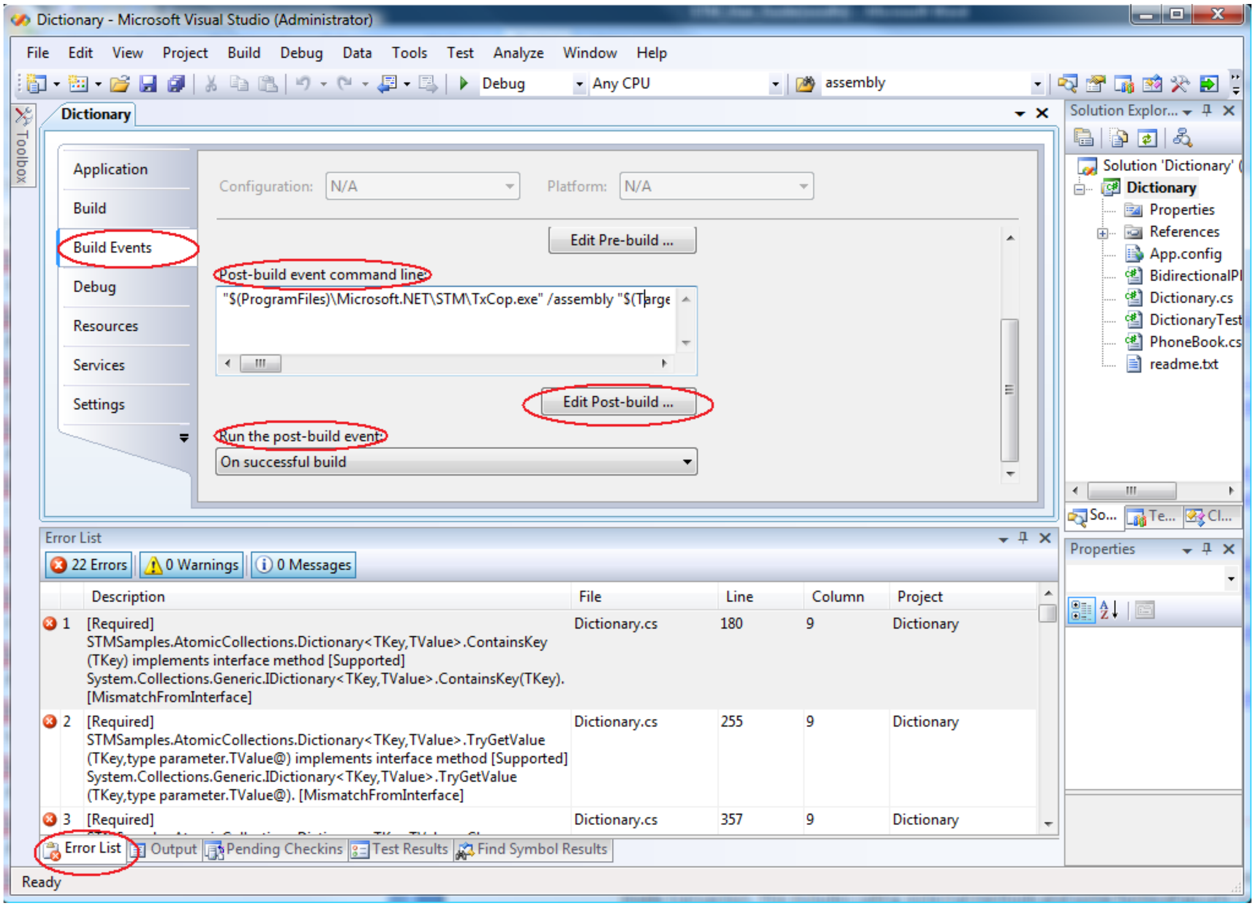
To add a post-build event open Visual studio, load your solution, go to the Project menu and click on Properties. This will bring up the properties pane. In the properties pane, click on the 'Build Events' option. At this point you will see two text boxes.

Enter the command to execute TxCop in the textbox that says 'Post-build event command line' or click on 'Edit Post-build' and then enter the command in the pop-up window and then click on OK.

Make sure that for the 'Run the post-build event' drop-down (in the build events pane) you pick 'On successful build'.

Now when you compile your solution, you will see a list of errors in the 'Error List' window in Visual Studio.

We have reproduced a screen shot of an example of using TxCop with VS integration. The screen shot highlights several things we have mentioned earlier in this section and shows the errors displayed in the error list window as well. Clicking on the error will take you to the offending file and line. The underlying framework on which TxCop is built is still not completely mature, which is why you may notice that some of the line numbers are a little off at times.



Configuration: N/A Platform: N/A

Post-build event command line

"$(ProgramFiles)\Microsoft.NET\STM\TxCop.exe" /assembly "$(Targe

Run the post-build event

On successful build

	Description	File	Line	Column	Project
1	[Required] STMSamples.AtomicCollections.Dictionary<TKey,TValue>.ContainsKey (TKey) implements interface method [Supported] System.Collections.Generic.IDictionary<TKey,TValue>.ContainsKey(TKey). [MismatchFromInterface]	Dictionary.cs	180	9	Dictionary
2	[Required] STMSamples.AtomicCollections.Dictionary<TKey,TValue>.TryGetValue (TKey,type parameter.TValue@) implements interface method [Supported] System.Collections.Generic.IDictionary<TKey,TValue>.TryGetValue (TKey,type parameter.TValue@). [MismatchFromInterface]	Dictionary.cs	255	9	Dictionary
3	[Required]	Dictionary.cs	357	9	Dictionary

The post-build event command line that is used for the Dictionary sample, which was used to generate the screen shot is given below. Please use it as a reference when adding your own post-build event.

```
"$(ProgramFiles)\Microsoft.NET\STM\TxCop.exe" /assembly"$(TargetDir)$(TargetFileName)"
/r "$(ProgramFiles)\Microsoft.NET\STM\mscorlib.dll" /r
"$(Windir)\Microsoft.NET\Framework\v4.0.STMret\system.dll" /r
"$(Windir)\Microsoft.NET\Framework\v4.0.STMret\system.core.dll" /r
"$(Windir)\Microsoft.NET\Framework\v4.0.STMret\system.data.dll","$(Windir)\Microsoft.N
ET\Framework\v4.0.STMret\system.data.datasetextensions.dll" /r
```

```
"$(Windir)\Microsoft.NET\Framework\v4.0.STMret\system.xml.dll" /r
"$(Windir)\Microsoft.NET\Framework\v4.0.STMret\system.xml.linq.dll"  /r
"$(ProgramFiles)\Microsoft.NET\STM\system.transactions.dll"
```

6.7 Dynamic Contract Checking

As part our STM .Net system we provide a runtime checker that complements our static checking solution. The runtime checker is part of the CLR execution engine, and can be used to help users identify code that might break the three basic rules of the STM system (described in the previous section) and help them fix these problems so that their applications can safely take advantage of the STM technology.

6.7.1 Strictness Level of Dynamic Checking

There are four different strictness levels of the runtime checker: minimal, relaxed, strict, and highest. The following table defines the violation types caught by the runtime checker at different strictness levels:

Strictness level	STM violations caught by the runtime checker
Minimal	The thread is in a transaction and • Non-transactable code feature is encountered (e.g., certain forms of unsafe code) —or— • A native method without the **AtomicRequired** or **AtomicSupported** contract is invoked[7].
Relaxed	Reports all violations of the **Minimal** level and these additional ones: • The thread is in a transaction and a field or a method with an explicit **AtomicNotSupported** contract is accessed. • The thread is outside of a transaction and an **AtomicRequired** method or field is accessed. (Note that if neither the assembly nor the method has an annotation, then the method is *implicitly* **AtomicNotSupported**. If such a method is encountered inside a transaction in relaxed mode, then this is not reported as a violation.)

[7] Note that user's code is not allowed to add AtomicRequired or AtomicSupported to a native method, unless AtomicSuppress is added too. Only native methods that are implemented by the CLR can have an AtomicSupported or AtomicRequired annotation without an AtomicSuppress. Thus, any p/invoke in your code MUST have an effective atomic compatibility value of AtomicNotSupported OR have an AtomicSuppress attribute.

Strict	Reports all violations of the **Relaxed** level and these additional ones: • The thread is in a transaction and an **AtomicNotSupported** method or field is accessed. (Strict mode doesn't distinguish between explicit or implicit contracts on methods. All methods that have an effective compatibility value of **AtomicNotSupported** and are accessed within transactions will be reported as violations.)
Highest	Reports all violations of the **Strict** level and these additional non-critical ones: • An **AtomicMarshalReadonly** attribute is put on a non-reference type parameter. • An **AtomicRequired** attributes is put on an instance field of a struct type.

One reason that we provide four strictness levels is that we want to accommodate different user scenarios. Eventually, for products using the STM system, the runtime checker should run at the strict level to enforce the three basic rules of the STM system as much as possible for safety concerns. But during the early adoption stages of the STM system, it's very likely that a programmer would like to experimentally run her program inside transactions to see if it's possible to transition it to transaction-safe code. Or she may have transitioned all of her code to the transactional version, and marked them with appropriate contracts. But she also needs to use a third party library, which does not have any STM contracts. The programmer may want to run the third party library, but still have contracts in the code checked, which is one of the scenarios making the relaxed level useful, since it allows strict checking of assemblies that are "STM aware" and at the same time more lax checking on legacy assemblies.

6.7.2 Reporting Violations

Based on the severity of violations, we divide all violations the runtime checker catches into three categories:

- **Critical violations**: violations caught at the minimal level.
- **Contract violations**: violations caught only at the relaxed level and the strict level.
- **Harmless violations**: violations caught only at the highest level (e.g. placing an AtomicMarshallReadonly attribute on parameters of value type. See section 10.1 for a discussion on the AtomicMarshalReadonly attribute).

Users can configure how the runtime checker reports a violation. You may choose to throw an exception (of type AtomicContractViolationException) whenever the runtime checker detects a violation. But sometime, you may want to run the program once, gather all error messages, fix them one by one, and then try again. Throwing exceptions on every violation makes the diagnostic process inconvenient. Therefore, we allow you to choose how to handle violations:

- They can be surfaced as exceptions on violations;

- You can also choose to just log non-critical violations (but critical violations will still be reported as exceptions);
- Finally you can choose to have violations first logged, and then surfaced as exceptions.

If the runtime checker detects a critical violation, it will always throw an exception since such a violation may result in incorrect program execution or a system crash. For harmless violations, the runtime checker won't throw exceptions since they are harmless unless at the Highest level.

6.7.3 Configuring the Runtime Checker

In summary, there are four configuration file variables that you could use to configure the runtime checker:

Configuration file variable	Purpose	Allowed values
STMRuntimeCheckLevel	Controls the strictness level of the runtime checker	minimal, relaxed, strict, highest
STMExceptionOnViolation	Throw exception if a violation is detected. This variable only affects violations in the contract violation category. At the highest level, it also affects the harmless type of violations.	0 (disable), 1 (enable)
STMLogOnViolation	Log violations to a log file if the log is specified, otherwise dump to the standard error output. This variable affects violations in all categories.	0 (disable), 1 (enable)
STMViolationLogFile	Specify the file where to log violations	File name

The app.config file in each of the samples that are provided contains an example setting for these variables.

6.7.4 Example

Let's say we have the following program[8]:

```
using System;
using System.TransactionalMemory;
using System.Runtime.CompilerServices;

class TestObject
{
    public TestObject()
```

[8] The NoInlining directives that are present in this example prohibit the JIT from inlining the given methods into their callers. When a method is inlined, method level compatibility checks will not be performed.

```
    {
        m_privateField = 1;
    }

    private int m_privateField;

    [AtomicRequired]
    [MethodImpl(MethodImplOptions.NoInlining)]
    public void AtomicRequiredMethod()
    {
        m_privateField++;
    }

    [AtomicNotSupported]
    [MethodImpl(MethodImplOptions.NoInlining)]
    public void AtomicNotSupportedMethod()
    {
        Console.WriteLine("This method is not supported in transaction.");
    }

    [AtomicNotSupported]
    public static void Main(string[] args)
    {
        TestObject obj = new TestObject();

        try
        {
            // Start a transaction.
            Atomic.Do(delegate
            {
                // Call a NotSupported method within the transaction.
                obj.AtomicNotSupportedMethod();
            });
        }
        catch (AtomicContractViolationException e)
        {
            Console.WriteLine(e.Message);
        }

        try
        {
            // Call a Required method outside the transaction.
            obj.AtomicRequiredMethod();
        }
        catch (AtomicContractViolationException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

Further let’s assume we run the runtime checker at the strict level with STMExceptionOnViolation enabled, and STMLogOnViolation disabled. We will get the following exception message:

```
'TestObject+<>c__DisplayClass2.<Main>b__0()' is accessed by
'System.TransactionalMemory.Atomic.Do(System.TransactionalMemory.AtomicAction)' inside a transaction, but it does not
have any atomic contract, neither does its assembly

'TestObject.AtomicRequiredMethod()' is accessed by 'TestObject.Main(System.String[])' outside a transaction, but it has
[AtomicRequired] contract.
```

Not the exact messages you would expect, right? This is because there are many compiler generated methods (to support the C# closure language feature in this case), on which we can't put atomic compatibility contracts. To solve this problem, you can run the runtime checker in the relaxed mode, which will ignore all elements without annotations, or you can add an assembly level AtomicSupported attribute such as:

```
        [assembly: AtomicSupported]
```

Now, we will get the following exception message as we expected:

```
'TestObject.AtomicNotSupportedMethod()' is accessed by 'TestObject+<>c__DisplayClass2.<Main>b__0()' inside a transaction,
but it has [AtomicNotSupported] contract.

'TestObject.AtomicRequiredMethod()' is accessed by 'TestObject.Main(System.String[])' outside a transaction, but it has
[AtomicRequired] contract.
```

7 Coordinating Threads with Transactions

If you are familiar with locks, critical sections and events, then you might be wondering by now how you are supposed to wait for a certain event when programming with transactions. Waiting for a condition to become true is a pattern that becomes more and more unnecessary as you move to using the .NET Parallel Extensions library since much of the coordination that you needed to explicitly code in the past is now taken care of for you by the system. For example, if in the past you needed to chunk some task between multiple processors you'd have to create some worker threads, feed the work to them and then wait for their completion. With Parallel.For the task becomes trivial and in particular there isn't any need to *wait* for the worker threads to finish.

In general, the Parallel Extensions for .NET promote a programming style that is centered on non-blocking tasks. When the tasks obey the non-blocking style, the system can schedule them optimally on just the right number of OS threads and thus save a lot of memory dedicated to threads' stack and eliminate unnecessary context switches.

Given all of the above, our recommendation for when you think you need to block inside a transaction is: *don't!* Take a step back and think how you can refactor the problem differently, using non-blocking tasks and structured parallelism.

One example that is often used to explain blocking patterns is the consumer/producer pattern. In this pattern there is a blocking queue. A producer wants to add elements to the queue, but needs to wait while space is not available. A consumer has to wait until elements are available, then they can be removed from the queue.

How would such a pattern be coded with non-blocking tasks? You'd do that by scheduling the consumer code of processing a removed element as a *continuation* to the production of the given element. For example, using Microsoft's Task Parallel Library, a continuation of a Task may be specified using the ContinueWith API.

However, we recognize that the world cannot switch instantly to this style of coding and therefore we do provide a mechanism for coordinating transactions, although we are unsure whether we will continue to support it in the future. You feedback on the question is highly appreciated.

STM.NET implements the "retry" coordination mechanism first described in 7.1. The semantics are dead-simple: if you hit a *retry* statement then the transaction rolls back and re-executes. e.g., consider this example:

```
public class SingleCellQueue<T> : where T : class {
      T m_item;

      public void T Get() {
            atomic {
                  T temp = m_item;
                  if (temp == null) retry;
                  m_item = null;
                  return temp;
            }
      }

      public void T Put(T item) {
            atomic {
                  if (m_item != null) retry;
                  m_item = item;
            }
      }
}
```

Suppose a thread calls `Get()` and `m_item` is null. In this case the retry statement would be encountered, the transaction will roll-back and re-execute. At this point we can imagine the reader is a little bit anxious: is it just going to spin and re-execute, eating up my CPU and killing my battery?

While in general there are many ways to implement "retry", our implementation is actually pretty precise, meaning it will only wake up a transaction once some object or a static that it has examined before retrying has actually changed by other transactions. This still doesn't mean that the condition that caused it to retry will be satisfied when it is finally woken-up, but it does reduce by a large factor the amount of unnecessary spinning that is associated with "retry".

One central property of *retry* is that it rolls back anything done so far in the transaction. This means it is not possible to externalize to the outside world why you're waiting and what it is that you wish be done by parallel activities, which is a pattern common with condition variables.

Let's move now to a sample that exercises "retry". As you may recall in the PrivatizePublish sample the workers simply did nothing when the data was not available to them (it could be unavailable if it was privatized temporarily by the "Sweeper" privatizing and publishing activity). We will now use a similar sample that introduces a wait as long as the data is not available. Retry.cs in the *Features* solution contains the source code. Here is the relevant piece of code:

```
// Transactional modification, conditioned on the existence
// of a published items array. We wait using retry for
// the items's array to get published.
Atomic.Do(() =>
{
    int[] items = SampleDriver.items;
    if (items == null) Atomic.Retry();

    int temp = items[toIdx];
    items[toIdx] = items[fromIdx];
    items[fromIdx] = temp;
});
```

If you are familiar with API's like Monitor.Wait/Pulse you'll notice the conspicuous absence of an API to "pulse" waiting transactions. This is the case since with "retry", the system automatically detects, as transactions commit, whether "interesting" state changes have occurred and what waiting transactions should be woken up and re-executed.

7.1 Retry and Nesting

If a retry occurs within a nested transaction, then, semantically, the entire *transaction nest* has to be retried. Again, this is in keeping with the understanding that nesting is only relevant to failure atomicity, not to isolation.

However the implementation has some optimization tricks that allow it to first retry just inner transactions and then progressively roll back all the way to the top, if necessary.

8 Transaction Abort in Greater Detail: Consistency and Exception Marshalling

In section 4.1 we explained the basics of aborting a transaction. To recap we explained that a transaction is aborted when an exception escapes the boundaries of an atomic block. The effects of the transaction are undone and this mechanism obeys nesting. There are two questions that we have left open and we will discuss them now after we have overall better understanding of the system.

8.1 Consistency of Aborted Transactions

The first question is about the consistency of aborted transactions. It is common to implement STM using optimistic concurrency control where locks are not physically taken, but are just probed for their version number and then the transaction proceeds, until it wishes to commit, and at this point the version numbers are checked to be unchanged from the first time they were read. This provides a great deal of scalability to TM but it also means that the code may run into inconsistent states temporarily. Such states will be ultimately detected at commit time. Here is an example:

```
Initially:
static int X = 0;
static int Y = 0;
```

```
// Thread 1
atomic {
    X++;
    Y++;
}
```

```
// Thread 2
int lx, ly;
atomic {
    lx = X;
    ly = Y;
}
Debug.Assert(lx==ly);
```

If we think of “atomic” as implemented in terms of a global lock, it’s clear that the assert on thread #2 should never fire. However in our implementation it is possible that thread 2 first fetches the value of X and then thread 1 executes its transaction. Thread 2 is now “doomed” but it still doesn’t know that. It proceeds to load the value of Y which is one bigger than the value it fetched for X. Now thread #2 reaches the end of the atomic block and it would *validate* its reads. It will discover that the version of the lock associated with X has changed since it first read X and therefore it will roll back and *re-execute*.

Thus thread #2 will only exit the atomic block once it has consistently read both the values of X and Y and therefore the assert will never fire. However, there will be intermediate states in which lx != ly. These states are in general not visible to the program. The system maintains the illusion as if the atomic block is really executing under a single global lock[9].

Now suppose that we modify the program to look like this:

```
// Thread 1
atomic {
    X++;
    Y++;
}
```

```
// Thread 2
int lx, ly;
atomic {
    lx = X;
    ly = Y;
    if (lx != ly) {
      // Can we get here?
      throw new AssertException();
    }
}
```

Again under global lock we would never see an exception thrown and therefore we arrive at the conclusion that the transaction can only abort if it is consistent. If it is not consistent it will re-execute. What does that mean in our single global lock model? Previously we said that the atomic block is equivalent to a global lock under the following transformation:

[9] But remember: this illusion is only provided for race-free programs.

```
atomic {
    <statements>
}
```

```
lock (globalStmLock){
    <statements>
}
```

We will now refine this transformation such that it more accurately depicts the fact that a transaction abort is a consistent event[10]:

```
atomic {
    <statements>
}
```

```
lock (globalStmLock){
    try {
        <statements>
    } catch {
        <undo-side-effects>
        throw;
    }
}
```

So essentially, an aborting transaction is identical to a read-only transaction reading the same values as the aborting transaction and then successfully committing. The values that are read by an aborted transaction can be communicated reliably to outside of the atomic block by capturing them in the exception that is thrown out. Here is yet another example demonstrating this:

```
public class MyException : Exception {
    public readonly int XValue;
    public readonly int Yvalue;

    public Exception(int x, int y) {
        XValue = x;
        YValue = y;
    }
}

// ...
static int X = 0;
static int Y = 0;
static int Z = 0;
```

```
// Transaction #1:
// readonly-commit
int lx, ly;
atomic {
    lx = x;
    ly = y;
}
Debug.Assert(lx == ly);
Debug.Assert(Z == 0);
```

```
// Transaction #2:
// read-write-abort
MyException myEx;
try {
   atomic {
      Z++;
      throw new MyException(X, Y);
   }
}
catch (MyException e)
{
```

[10] While this is an accurate model, our system of course doesn't implement the model using a single global lock. The section on performance tuning (section 0) contains a great deal of detail on how this model is actually implemented in STM.NET.

```
    myEx = e;
}
Debug.Assert(myEx != null);
Debug.Assert(myEx.XValue == myEx.YValue);
Debug.Assert(Z == 0);
```

Transactions #1 and #2 are equivalent. They both consistently read the values of the shared variables X and Y and they result in no modification to any other variables.

8.2 Transactional Exception Marshaling

The astute reader may notice that there is a problem with the atomic transformation we have just postulated:

```
atomic {
    <statements>
}
```

```
1. lock (globalStmLock){
2.     try {
3.         <statements>
4.     } catch {
5.         <undo-side-effects>
6.         throw;
7.     }
8. }
```

If before re-throwing the exception on line #6 we first undo all side effects in line #5, then that would include, potentially, the construction of the exception object that is thrown by the transaction. i.e., part of the "undoing" would be reversing the effects of the initialization of the exception object, in the likely case that it was constructed inside the transaction.

Thus, the above transformation can result in throwing a "weird" exception object. e.g., if we consider MyException object, the XValue and YValue fields would have their value reset to zero before the exception is re-thrown.

We need a solution that would "freeze" the exception object (and everything that it refers to) in a manner that will allow resurfacing the "frozen" exception object in its consistent state outside of the atomic block. Luckily such as abstraction already exists in .NET when exceptions have to be marshaled across isolation boundaries, e.g. in cross app-domain calls. What happens is that (at least conceptually) the exception is serialized into a buffer that is passed from inside transaction to the outside world. The reading of the object graph for serialization is done transactionally and thus it becomes part of the validation criteria of the transaction. In other words, the serialized state of exceptions is always consistent. The writes into the buffer are done through some "magic" such that they are not rolled back when the rest of the side effects of the transactions are rolled back. Finally, when the transaction has rolled back the serialized exception state is deserialized and the exception object is re-thrown.

In our implementation we have some optimizations over the serialization/deserialization pass. In particular we clone objects-to-objects without going through an intermediary buffer.

From the programmer’s perspective, the transactional exception mechanism means the following:

1. The exception object that you catch outside of the atomic block is not the same one that is thrown inside the block. It is the result of serialization/deserialization pass.
2. At the same time, the system reserves the right to introduce optimizations that will allow surfacing the same exception object. So the exception that you get out may or may not be the same object you throw inside the block, but it is functionally identical (under serialization/deserialization rules).
3. You thus must ensure that the exceptions you throw out of atomic blocks are serializable.
4. Be careful not to point to big amounts of data from your exception object, as this data will be copied in the marshalling.

In summary the best way to think about surfacing exceptions from atomic blocks is that it’s extremely similar to throwing exceptions across app-domain calls.

8.3 Re-execution Terminology Recap

By now you have learned about multiple conditions due to which a transaction may rollback and re-execute and it’s typical at this point to be a little bit confused, so let’s recap the definition of some terms:

- **Rolling back a transaction**. Transaction roll back is a system mechanism that undoes all the effects of the current transaction. It is employed in various situations such as when a transaction aborts, retries, or sometimes when it encounters contention on a lock and re-executes.
- **Aborting a transaction**. A transaction is aborted when a thrown exception escapes the boundaries of the atomic block and the transaction is in a consistent state.
- **Retrying a transaction**. A transaction is retried when it encounters a retry statement. This results in the rolling back of the transaction, waiting for changes and then re-executing the transaction.
- **Re-executing a transaction.** A transaction may roll back and re-execute multiple times, until it results in the transaction being aborted or committed. These are the only two terminal states of a transaction.

9 Transactions at Large: Integration with System.Transactions

When we spoke about failure atomicity in this document, we always meant that it applied to .NET memory only. But programmers deal with more entities than just memory – there are files, databases, networking, etc., and there are well-established ways to achieve failure atomicity for these entities. Surprisingly, real applications typically excluded memory from the failure atomicity and isolation equation, even though there are technologies to do so today. Instead, programmers manually code complicated compensation and isolation algorithms for in-memory state, probably because currently deployed technologies for providing failure atomicity and isolation for in-memory state are either not established or not convenient enough. STM offers to change this situation and make in-memory state yet another resource, similar to a database, which provides failure atomicity and isolation.

In the traditional transactions' world, e.g. in Microsoft's Distributed Transaction Coordinator (DTC) there exists the notion of *transactional resource managers*. Transactional resource managers or simply RM's manage a resource that participates in a transaction. As work proceeds against resources, the resource managers keep tabs on what in particular has been done (such that it can be undone if the transaction aborts) and what locks are necessary and have been taken to ensure isolation (such that they are released when the transaction completes etc.). The transaction coordinator orchestrates all the resource managers that are involved in a transaction such that they reach agreement on the *outcome* of the transaction. More information about the topic of distributed transactions can be found in Jim Gray's book on transaction processing [14] and other sources.

In this implementation of STM you can see that it is both a resource manager, managing the state of .NET objects that are accessed during the transaction, and a transaction coordinator that controls the lifetime and outcome of the transaction, with STM being the only resource manager participating in the transaction. When we set out to control entities beyond .NET memory we once again reach the DTC/RM design.

Remember the BankAccount sample of section 0? It was handy to get failure atomicity from STM. But that sample dealt only with .NET memory; it is not what happens with real bank accounts – those live in durable databases. Database programmers have perfected over the decades the art of making the operations in their code ACID (Atomic, Consistent, Isolated, and Durable). It is achieved by using transaction coordinators and resource managers.

Windows has a pretty compelling story here: three transaction managers (distributed: MSDTC; kernel: KTM; and managed LTM in System.Transactions namespace) work with a group of resource managers (e.g. SQL, MSMQ, Transactional NTFS (TxF), and the transactional registry). With these, the user can get failure atomicity from any combination of database, networking, file, registry operations—either all of the operations happen, or none do. It makes programming much easier, since it eliminates extremely error-prone processing of all possible outcomes.

It would be nice to add memory to this list. Actually, it is already achievable today by writing a custom Volatile Resource Manager, a concept in LTM (Lightweight Transaction Manager); but it requires you to manually write a resource manager for every custom data structure or variable that you need to access in a transaction—you need to provide isolation, failure atomicity, and provide mechanisms to avoid deadlocks etc. These are exactly the things that STM.NET provides, so it seems that STM would be a great way for handling "volatile resource management" or, in other words, transactional management of in-memory state.

There are many applications that manage both durable resources and in-memory data. The memory can represent not only transient data, but a longer-used model, reference data, or a cache. It is important to be sure that in-memory data correspond to the durable "truth". STM together with system transaction provide this for free (in-memory and durable state sync), because STM transactions are fully integrated with LTM/DTC transactions. In the example below, failure atomicity is guaranteed for all participants:

```
Atomic.Do(() =>
{
    << memory  operation >>
    << database operation >>
    << any other transacted operations, e.g. networking via MSMQ >>
});
```

Let us illustrate these capabilities using the BankAccount sample we covered in section 0: the accounts will now live both in a database and in memory, and they need to be kept in sync. We would love to show you the program with SQL, but we cannot do that yet. Though we have fully prototyped STM integration with LTM/DTC, each resource manager library (in SQL's case, ADO.NET) needs some modifications before it could be used, and in our incubation effort we chose to first enable MSMQ–Microsoft Message Queuing—for use in atomic blocks. With respect to transactional behavior, MSMQ is not much different from SQL; some customers actually use MSMQ for storing durable data and accessing it in a transactional manner. So here is the real code that you could run on STM.NET today:

```
private MessageQueue qAccount;

public void ModifyBalance(int amount)
{
    Atomic.Do(() =>
    {
       m_balance = m_balance + amount;
       qAccount.Send(m_balance, MessageQueueTransactionType.Automatic);
       if (m_balance < 0)
           throw new OverdraftException();
    });
}
```

You are guaranteed that the balance amount is always the same in memory and in the queue, and you didn't have to write any additional code for that. *Atomic* here starts as a usual STM transaction (relatively cheap). Then, when the MSMQ *send* operation is invoked, the STM transaction is *promoted* into a LTM transaction (LTM, in its turn, may promote it to MSDTC). From that point, LTM or DTC manages the coordination of transaction, and STM becomes a Resource Manager, responsible for memory only—the same way as the sibling MSMQ resource manager is responsible for durable data.

The installation contains two samples[11] demonstrating the use of traditional transactions with STM: *TicTacToe* and *TraditionalTransactions*. It is an interesting question; "how do *Atomic* blocks compose

[11] Note: while running samples in the debugger, you may notice two exceptions that we handle internally – HoistingException, which signals the need for promotion from STM to LTM, and SystemReexecException, that is used on the slow path for combined STM-system transaction. User should never see these exceptions.

with traditional transactions (introduced typically with the syntax “*using new TransactionScope()*”)? Whether and how can you use these two types of transactions inside one another?”

You cannot use an *Atomic* block inside of a top-level system transaction, if there is a chance that the code inside *Atomic* will call some other operations relying on System.Transactions. The reason for that restriction is that System.Transactions does not support true nesting with partial failure atomicity. System transactions are always flattened.

One more thing to note is that for integration with traditional transactions you have to use *Atomic.Do* (and not try/catch with *AtomicMarker*).

Since *Atomic* can play a façade role for *TransactionScope*, we also allow passing through *TransactionScope* options. Here is the API:

```
public delegate void  AtomicAction();

public enum SystemTransactionPromotion { Disallowed, Allowed,  Always };

public class Atomic
{
   public static void Do(AtomicAction action);
   public static void Do(SystemTransactionPromotion promotion, AtomicAction action);
   public static void Do(IsolationLevel isoLevel, AtomicAction action);
   public static void Do(TimeSpan timeout, AtomicAction action);
   public static void Do(IsolationLevel isoLevel, TimeSpan timeout, AtomicAction action);
}
```

One more question you might have now is how it all affects performance. The answer is that it is all “pay for play”: if you don’t use integration with traditional transactions, your pure memory transactions are not slowed down. On the other hand, if you do use durable transactional operations, their latency is going to dominate the performance of your code anyway, such that adding STM on top of your in-memory data structures and algorithms is hardly even measurable.

10 Taming Side Effects inside Transactions

We have talked about how the transactional model requires code components to provide two fundamental capabilities: (a) isolation and (b) failure atomicity. When you create and access managed objects inside transactions the JIT provides these two capabilities automatically by locking fine-grained transactional locks and writing the data into shadow copies, which are copied into their master locations, if and only if the transaction commits, and under the protection of the transactional locks.

For legacy code and for I/O operations however, we still have to present solutions that will allow their usage inside transactions. Well, in this section we are finally going to be exploring this aspect of STM.NET.

The solutions we will explore run the gamut from very simple and crude, such as “punching through” the transaction to emit debug output, to sophisticated and high fidelity, such as using a transaction to writing into a message queue through integration with System.Transactions. When dealing with legacy I/O there isn't one correct answer. The "correct" answer depends on the nature of the resource that one wishes to leverage inside transactions, whether it already has transactional semantics and to what degree of fidelity isolation and failure atomicity need to be provided.

We will start with primitive building blocks and work our way up the stack.

10.1 Atomic Suppress

Sometimes it is useful to *suppress* the transaction momentarily and carry out operations that are not transacted and then return back to the transaction. Some information may flow between the transaction and the "suppress region". The STM system provides the mechanisms to do this but it should be noted that there are quite a few pitfalls associated with this advanced feature. Let's start with a simple example:

```
[AtomicSupported]
[AtomicSuppress]
public static void PrintMe([AtomicMarshalReadonly]string msg)
{
    Console.WriteLine(msg);
}

public static void Test()
{
     Atomic.Do(() =>
     {
          m_balance++;
          string msg =  string.Format(
              "Suppressed, inside trasaction: {0} ", m_balance);
          PrintMe(msg);
          m_balance++;
     });
}
```

In the sample we would like to use console output as a debugging aid, so we would like the messages that we output to print regardless of whether the transaction commits or aborts, and no matter how many times it is re-executed. (Debugging is one of the main uses of [AtomicSuppress].)

In order to do that we wrap Console.WriteLine in a method called PrintMe which we decorate with [AtomicSupported] and [AtomicSuppress]. As you can imagine, Console.WriteLine has a contract of [AtomicNotSupported] as it really needs to drive output to the console, which is not at all a transactional activity. PrintMe, on the other hand, has an explicit contract of [AtomicSupported]. Isn't there a contract violation here with an [AtomicSupported] method calling an [AtomicNotSupported] method? The answer is no. The [AtomicSuppress] in PrintMe instructs the system to put the transaction on hold and therefore the body of PrintMe is running outside of the atomic block and can call any [AtomicNotSupported] method it wishes. The onus is on the programmer to make sure this indeed makes sense to do, and this is exactly why this feature is a little bit tricky.

So now we understand how to temporarily act as if you're not inside the transaction, but how do you get data in and out of the transaction? The most fundamental thing to understand about suppress is that, from a model perspective, it runs concurrently with the suppressed transaction and thus it is not allowed to access the same data that the suppressed transaction is accessing, nor is it allowed to access any state that any other transaction is accessing. This may not be intuitive to understand so let's explain why this is the case. When the transaction body executes, some changes that the transaction makes are not applied to the objects and statics that are accessed, but rather to shadow copies. Thus, changes that have been made by the transaction may be invisible to the suppress region. Similarly, if the suppress region changes objects that have been accessed already by the transaction, those changes too may not be visible to the transaction once it resumes, since it's working under the assumption that the most up-to-date information is available in the shadow copies it has created, not in the original locations that are modified by the non-transactional code in the suppress region.

Every piece of information that you want to pass to the suppressed function must be *marshaled out*. Does that sound familiar? You may recall that in section 8.2 we explained how exceptions were marshaled outside of atomic blocks. Suppress uses a similar mechanism.

In the sample above we have created a string (referred to by the string reference variable 'msg'). That string object having been constructed inside a transaction cannot be accessed directly inside the suppressed method; it must be marshaled to it first. This is achieved using the [AtomicMarshalReadonly] attribute on the 'msg' parameter of the PrintMe method. The semantics of this attribute are the following:

> *Clone as necessary (using standard serialization infrastructure) the given object and the entire object graph rooted at it such that the resulting object graph can be read in the suppress method.*

The reason for this definition of AtomicMarshalReadonly is three-fold:

- It provides reasonable semantics such that you could call PrintMe also *outside* of transactions. If you did that, we wouldn't want to introduce any marshaling. If we don't introduce any marshaling, it would still be legal to read the original object. So if you coded PrintMe such that it only reads the parameter, it would work both inside and outside of transactions.
- If you haven't really modified the object inside the transaction, the system may be able to give you the same object to read in the suppressed method, eschewing the costly cloning process.
- The system may be able to modify some objects in-place when they are modified inside transactions. In that case too the system will be able to pass the same object reference to the suppressed method without additional cloning, which is again a performance win.

Note that marshaling is only necessary when you need to pass objects that were modified or created within the transaction. If you have an object reference pointing to an object that is consistently only updated outside of transactions, then no marshaling is necessary.

Also note that value types are always copied by value when they are passed to methods, so no marshaling is necessary.

The [AtomicMarshalReadonly] attribute cannot be applied to parameters of type pointer or by-ref (ref or out parameters in C#), nor is it valid to pass a by-ref or a pointer to memory that has been updated by the transaction.

So far we have covered passing state from within the transaction and into the suppressed method. What about the other direction? Can state be created in the suppressed method and passed back to the transaction? The answer is yes, this is a variant of the publication pattern (section 5.1.3). Here is an example that demonstrates:

```
[[AtomicSupported, AtomicSuppress]
private static string LoadReasourceString(
     [AtomicMarshalReadonly] string resourceId)
{
     // Heavy lifting here. Load resource dll, cache strings etc.
}
```

This method will suppress the transaction, will load a resource assembly, lookup the asked-for resource using the string resourceId (which was marshaled in) and will finally return the result. The result will then be available for the transaction to inspect. In this case we are passing a string back into the transaction, which is an immutable type, but in other cases mutable objects may be passed into the transaction as well and the transaction will be free to modify them as long as they *cease to be available to non-transactional code*. i.e., it is valid to do so as long as the publication pattern is implemented correctly.

It is interesting to note that the operation LoadResourceString is *logically side-effect free*. It is true that at some layer of abstraction it is not at all side effect free—a resource assembly is loaded, caches are populated etc. But at a higher layer of abstraction, these details are just implementation details of a *functional interface* that maps resource ID's to resource strings in a fixed manner, like a mathematical function that always provides the same output for a given input. It is common to use suppress around such functional interfaces.

10.1.1 AtomicSuppress and Inconsistent State

We have explained briefly in section 5 how STM implementations use optimistic concurrency control in order to achieve great levels of scalability and we will discuss this topic in depth in our performance tuning guide, in section 0. One unfortunate result of optimistic concurrency control though is the possibility of inconsistent states. Usually the STM system hides these states by rolling back the transaction and re-executing. However, [AtomicSuppress], being a low-level tool allows observing inconsistencies and the onus is on the application programmer again (you!) to make sure that the code in the suppressed method can sustain inconsistencies and if not, to make sure that it *validates* the transaction before carrying out any unsafe operations.

For example, consider the above PrintMe example. Running into an inconsistent state may result in a string that contains information that doesn't make sense from an application perspective, but it will always be a valid .NET string object. Thus you have the choice of either validating the transaction before printing the message, in which case the method will run slower (and you won't be able to observe funny inconsistent state) or you can choose to always print the message, even if it contains inconsistent state, maybe because you are interested in the a more low-level view of the system.

In order to request validating you have to specify a SupressValidationOption in the AtomicSuppress Attribute, as is demonstrated in the next code snippet:

```
// When validation option is specified, ConsistentPrintMe
// will only be invoked with consistent data, but invoking
// it will require validating the transaction, which will
// make it slower.
[AtomicSupported]
[AtomicSuppress(SuppressValidationOption.Validate)]
public static void ConsistentPrintMe(
    [AtomicMarshalReadonly]string msg)
{
    Console.WriteLine(msg);
}

// When validation option is not specified, the method
// may be invoked with inconsistent data, but no validation
// is required before invoking the method.
[AtomicSupported]
[AtomicSuppress]
public static void InconsistentPrintMe(
    [AtomicMarshalReadonly]string msg)
{
    Console.WriteLine(msg);
}
```

10.1.2 Current Limitations of AtomicSuppress

Currently suppressing an extern method is not supported. You have to suppress a managed method, which in turn can call any method it wishes, including extern methods.

We also do not support starting new transactions from within suppressed methods since this may cause a deadlock in the STM implementation.

10.2 AtomicRedirect

Sometimes it is useful to do different things depending on whether or not you're in a transaction. AtomicRedirect provides a mechanism to redirect control flow to an alternative method. For example, suppose we have a class that doesn't want to support transactions at all. However the class overrides Object.GetHashCode and Object.GetHashCode has a contract of [AtomicSupported] that requires all derivations to be [AtomicSupported] too. The derived class cannot change the contract on Object.GetHashCode, this is out of its control. However, it can throw a NotSupportedException if the GetHashCode method is ever called inside a transaction. This is how you would code this:

```
public class RedirectSample {
    [AtomicSupported]
    [AtomicRedirect("TxGetHashCode")]
    public override int GetHashCode()
    {
        return SomeNotSupportedMethod();
    }

    [AtomicRequired]
    private int TxGetHashCode()
    {
        throw new NotSupportedException();
    }

    //...
```

The [AtomicRedirect] takes a single string parameter which is the name of a private method in the same class, with the exact same number of arguments, return values and it must also be static if the “redirected” method is static, or an instance method, if the redirected method is an instance method (i.e., they must also agree on the “this” parameter). The method that is the redirection target must have an effective atomic compatibility value of either [AtomicSupported] or [AtomicRequired].

Similarly to [AtomicSuppress], [AtomicRedirect] also introduces an *atomic context switch* (from AtomicContext.Unknown to AtomicContext.Always). When control enters GetHashCode, we know for sure that it wasn’t called within an atomic block, because it would have been called inside an atomic block, then control would have been diverted to TxGetHashCode. Thus, the initial atomic context in a redirected method is AtomicContext.Never, meaning you can call any [AtomicNotSupported] method you wish etc.

Here is another example:

```
public static bool IsInAtomicBlock
{
    [AtomicSupported]
    [AtomicRedirect("TxGetIsInAtomicBlock")]
    get
    {
        return false;
    }
}

[AtomicRequired]
private static bool TxGetIsInAtomicBlock()
{
    return true;
}
```

The “IsInAtomicBlock” static property simply returns true if it’s invoked inside an atomic block and false otherwise. (Note how the getter method of the property is hooked up to a method with the same signature.)

Now assume that we had provided IsInAtomicBlock as a static property in the Atomic class. Then we could have coded the GetHashCode in the following manner:

```
public class RedirectSample {

[AtomicSupported]
public override int GetHashCode()
{
    if (IsInAtomicBlock)
        throw new NotSupportedException();
    return SomeNotSupportedMethod ();
}

//...
```

Which version is preferable? The latter version is indeed more concise, but note that it is not easily verifiable as [AtomicSupported] since its body contains a call to SomeNotSupportedMethod. So in other words [AtomicRedirect] provides a statically verifiable atomic context transition that allows static checking of atomic contract compatibility.

10.3 Using the Monitor Class inside Atomic Blocks

One facility that existing code uses extensively that is commonly considered side-effecting is the acquisition and release of locks. In .NET it is common to use the Monitor class with its nice syntactic wrapping using the lock keyword. In order to allow the reuse of such existing code inside atomic block we allow using Monitor.Enter, TryEnter and Exit inside atomic blocks. In order to reason about lock usage inside transactions you have to take two considerations into account:

1. The single-global-lock abstraction of STM, and
2. The granularity rules for data races

Both of these considerations are described in detail in section 5.

One example of a problem that can ensue from using locks with atomic blocks is the possibility of a deadlock. Consider this program:

```
//Thread 1
atomic {
    lock (obj) {
        // ...
    }
}
```

```
//Thread 2
lock (obj) {
    atomic {
        // ...
    }
}
```

If we replace “atomic” with lock(globalStmLock) then readily we see that we have conflicting lock acquisition order and thus a deadlock is possible.

Generally, you need to consider “atomic” as a distinct lock in your application lock hierarchy in order to avoid deadlock. One simple way of ensuring that this principle is upheld is using this pattern:

Treat “atomic” as the top-most lock in your application

10.4 Deferred and Compensated Actions

Side-effecting actions, such as I/O, traditionally are not handled by transactional memory systems. These operations generally do not involve memory, so it’s easy for us to just say “not supported. We didn’t take the easy way out.

The problem with side-effecting actions is that optimistic concurrency control assumes potential re-execution and not every operation can be safely repeated. For example, if you print inside transaction, you don’t want to see the multiple printouts in case transaction gets re-executed. If you format the disk, you don’t want it to happen even if a transaction eventually aborts.

Since STM.NET integrates with traditional transactions we have a chance to provide a solution. We cannot offer magic, but we can offer a bunch of partial solutions, which might help in majority of situations. Please note that our I/O proposal is based on LTM integration which means you lose STM nesting benefits, specifically, partial abort of nested transactions. We provide multiple mechanisms:

- Deferred actions that are done immediately after commit
- Compensated actions, that have a defined “undo” operation
- Transactional building blocks, that allow you to create your own transactional resource

10.4.1 Deferred Actions

In many cases, it is possible just to postpone an operation: keep all necessary data, and execute the action only after committing transaction. Deferred actions will accumulate while execution progresses; on rollback, they are dropped; but on commit, all of them are executed in the accumulation order. These operations happen after the transaction completes, you are free to do anything you like. It is more convenient than physically coding after transaction because you get a formal way of “transporting” local data from inside transaction, and you can inline your code in same place where you draw the data from. Here is the API:

```
public class Atomic
{
    public static void DoAfterCommit(Action<Object> commitAction, Object context)
}
```

The context object in the API helps you by marshaling data from inside transaction at the time this API is called so that action could use it after transaction commits. Here is example how you might use it in the printing case:

```
Atomic.Do(  () =>
{
   m_balance = m_balance + 1;
   Atomic.DoAfterCommit( (Object o)=>
   {
        Console.WriteLine("m_balance= " + o);
   }, m_balance);
});
```

You can safely do anything inside of the delegate that does not require a transaction, since it is executed outside of the transaction, not more than once, after it had committed. But don't forget that all the data you might use, except of the specially transported context, will be in their post-transaction global state - it is as if you just coded your actions after the transaction, but used the saved context. For instance, the example above might give you simple tracing facility. Case 10 in TraditionalTransactions sample illustrates deferred actions.

10.4.2 Compensated Actions

Not in all cases is deferral possible, sometimes you need to do your action immediately, inside non-committed-yet transaction, but compensate it in the case of rollback. This breaks isolation (the world will see your side effects even if the transaction is doomed later), but maybe it is OK, so long as those side effects are compensated if the transaction fails. Here is the API:

```
public class Atomic
{
    public static void DoWithCompensation(
         Action<object> immediateAction,
         Action<object> rollbackAction,
         object context);
 }
```

You define separately:

- what to do immediately;
- how to compensate it, if needed;
- common context for both actions.

Both the **`immediateAction`** and **`rollbackAction`** will be executed under a suppressed context (first immediately, second – during rollback, if it happens), so you can use any operations inside that does not require a transaction; just beware of using any data other than that which was explicitly passed via the context parameter.

And here is an example (see case 11 in TraditionalTransaction sample):

```
Atomic.Do(  () =>
{
   m_balance = m_balance + 1;

   Atomic.DoWithCompensation(
    (Object o) => {Console.WriteLine("Immediate action : " + o); },
    (Object o) => {Console.WriteLine("Compensating action: " + o); },
    m_balance);

   m_balance = m_balance + 1;
});
```

Case 11 in TraditionalTransactions sample illustrates compensated actions.

10.5 Implementing a Resource Manager for Your Operation

There certainly will be cases when you don't want peculiarities like above, but have more precise requirements and you need to implement for your operations a complete resource manager. We may be able to help you to do it more easily through the new *TransactionalOperation* helper class from the *System.TransactionalMemory* `namespace`. This class hides almost all knowledge of LTM and STM interaction; all you need to do is to inherit from it and implement proper operations. Of course, the nature of your operation may make it hard, but at least you stay in the realm of your subject.

Here is the programming model:

```
public class TransactionalOperation   // you inherit from this
{
    public void OnOperation();        // you call this at start of operation
    public void FailOperation();      // you call this if operation fails

    public virtual void OnCommit();   // you implement commit
    public virtual void OnAbort();    // you implement abort
}
```

The exact rules for using `TransactionalOperation` are:

1. Derive your class from TransactionalOperation
2. In each operation method, call OnOperation() at start, and FailOperation() at failure
3. Override void OnCommit() and implement proper 2nd phase commit actions
4. Override void OnAbort() and implement proper 2nd phase abort actions

Here is an example (see case 13 in TraditionalTransactions sample): we implement a text file appending operation. Please note how we abstracted away most of STM/LTM details:

```
public class TxAppender : TransactionalOperation
{
     private TextWriter m_tw;
     private List<string> m_lines;

     public TxAppender(TextWriter tw) : base() {
         m_tw = tw;
         m_lines = new List<string>();
     }
     // This is the only supported public method
     public void Append(string line) {
         OnOperation();

         try {
             m_lines.Add(line);
         }
         catch (Exception e) {
             FailOperation();
             throw e;
         }
     }
     // Implementing virtual methods of TransactionalOperation
     public override void OnCommit() {
         foreach (string line in m_lines) {
             m_tw.WriteLine(line);
         }
         m_lines = new List<string>(); // Prepares for the next Tx
     }
     public override void OnAbort() {
         m_lines.Clear();
     }
}
```

And here is how you can use TxAppender inside an atomic block:

```
public static void Test()
{
    TxAppender tracer = new TxAppender(Console.Out);
    Atomic.Do(
        delegate()
        {
            tracer.Append("Append 1:  " + m_balance);
            m_balance = m_balance + 1;
            tracer.Append("Append 2:  " + m_balance);
        });
}
```

10.6 Summary

Introducing optimistic concurrency control into an existing program will never be seamless, since some parts of it suddenly start to re-execute. But you can prepare resource managers for popular operations and use deferred, compensated, or suppressed operations for ad-hoc situations. We hope that this set of options will cover the striking majority of situations you need to handle.

11 Performance Optimization and Troubleshooting

The purpose of the release of STM.NET is to present this emerging technology and seek feedback about its programming model, potential usage, and user experiences. Though performance is not the focus of the release, we still want to help early adopters understand the performance characteristics of the STM system, as they are in this point. When considering performance, we care about both sequential performance and parallel scaling.

This section walks through the state of performance in the current release. We reveal some design and implementation details to help the reader gain better understanding of the performance behavior. We also outline some performance optimizations and tuning work can be helpful in the future, and demonstrate the potential performance gains we are likely to obtain by doing so.

All the performance data reported in this document is collected on a 24-way machine.

- Processor: 4 x 6-core Intel Core 2 processors. 2.66GHz.
- Cache: L1 Data: 8 x 32 Kbytes, L1 instruction: 8 x 32 Kbytes, L2: 4 x 3072 Kbytes, L3: 16 Mbytes.
- Memory: 16,378 Mbytes.
- Operating System: Windows Server 2008 64-bit. We run STM.NET in Wow64 mode.

11.1 Sequential Performance

STM is not for free. It comes with sequential overheads because of the runtime instrumentation and bookkeeping. In this release, we have not heavily optimized the runtime overheads yet. The only substantial optimizations we have done are:

- Path optimizations in object read/write scenarios
- Common sub-expression elimination of transactional barriers

For the future, we have identified a series of compiler and runtime optimizations as well as code tuning that we believe will reduce the sequential slowdown. Meanwhile, we also keep a close eye on the relevant techniques exploited by the research community.

The following table shows the sequential slowdown of one benchmark on the current release. The benchmark we use here is a thread-safe red-black tree data structure. We can perform operations, such as lookup, insert, and remove, on the tree. This benchmark does not exploit many C# language features, so it is only used to give the readers a rough idea on the sequential cost of a common data structure on the current STM.Net release. For the measurement, each thread performs 10,000,000 operations. We report the average of 5 runs.

- With read-only operations (lookup):

	Execution Time (seconds)	STM Sequential Slowdown
STM	8.111	
Without any Synchronization	1.618	5.0x
Lock[12]	2.085	3.9x
RWLockSlim[13]	2.456	3.3x

- With 25% updating operations (insert, and remove):

	Execution Time (seconds)	STM Sequential Slowdown
STM	9.786	
Without any Synchronization	1.947	5.0x
Lock	2.392	4.1x
RWLockSlim	2.783	3.5x

- With 50% updating operations (insert, and remove):

	Execution Time (seconds)	STM Sequential Slowdown
STM	11.488	
Without any Synchronization	2.276	5.0x
Lock	2.700	4.2x
RWLockSlim	3.115	3.7x

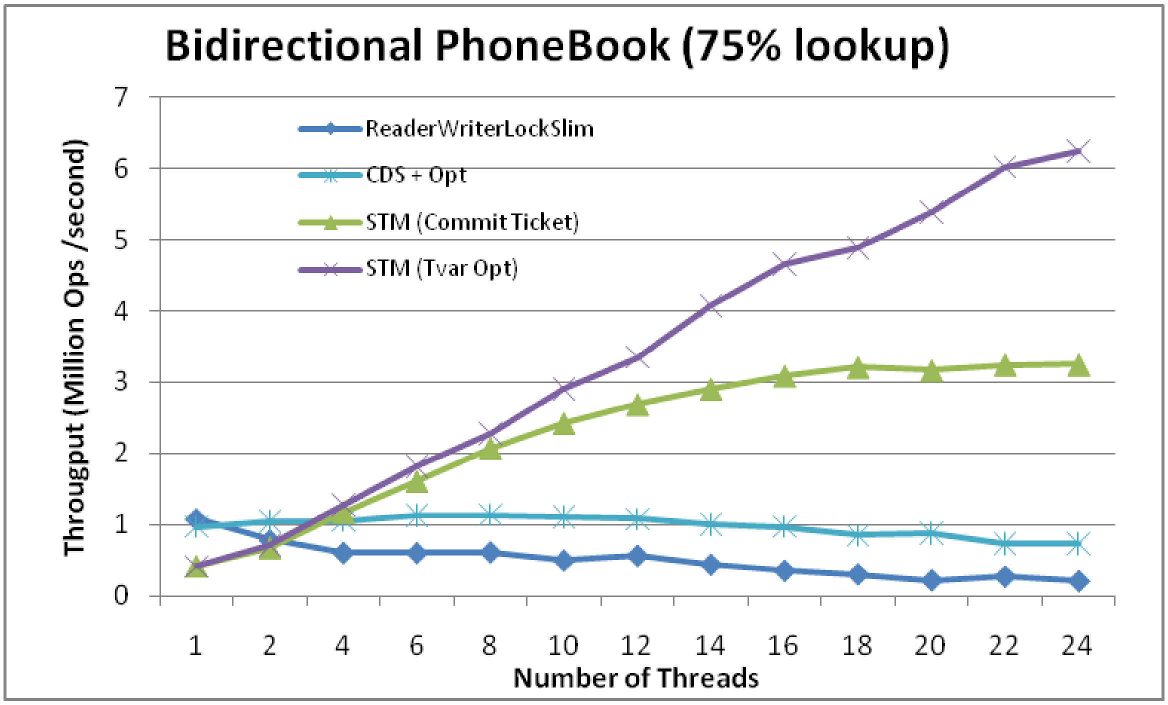
As shown by the above tables, for the read-only cases, the STM version shows a 3.9x and 3.3x sequential slowdowns over the lock based and ReaderWriterLockSlim-based versions, respectively. The updating operations may contain writes, though they are still dominated by reads. It is clear we pay a higher cost for writes than for reads. The sequential slowdown is exacerbated when there are more writes. Note that in this microbenchmark, all threads are essentially always executing transactions. Obviously, many programs spend only a small fraction of their execution in synchronized regions, and such programs would experience correspondingly smaller sequential slowdowns.

11.2 Scalability

This section provides information on the scalability characteristics of STM.NET. Generally speaking, STM achieves great scalability, unrivaled by any other technology with the same level of usability and composability, to the point where it is able to offset the sequential slowdown at relatively low core counts. Section 11.4 provides some detailed measurements in two scenarios, but just to illustrate the point here, consider the below performance curve:

[12] Use lock(this) {}

[13] Use System.Threading.ReaderWriterLockSlim class supported since .Net 3.5.



We see that due to better scaling, compared with available alternatives, STM performance is the best available of all alternatives, starting from four cores.

11.2.1 Contention Management

STM.NET provides object-level granularity for transacting objects and field-level granularity for transacting static fields. Therefore, the read/write barriers that STM introduces on data access paths and conflict detection happen at the level of objects. A conflict occurs when two concurrent transactions access *the same object* (even if they access different fields), and at least one of the accesses is a write. The runtime is responsible for detecting the conflict and taking an appropriate action. Again, keep in mind that for objects, contention detection happens at the whole object level and not at the object field level. When two transactions conflict, they are *serialized* by the runtime. That is, they will be prevented from running concurrently by the runtime. Let's consider an example.

Thread 1	Thread 2

`atomic {`
`  o.a =  1;`
`}`
	`atomic {`
	`  localVar = o.b;`
	`}`

In the above example, thread 1 writes to field "a" of object "o" in a transaction. Concurrently, thread 2 reads field "b" of object "o", also in a transaction. Though you may feel there is no conflict between the two transactions, there is! Because they are accessing the same object "o" and thread 1 is writing to a field, the two transactions conflict, and thus they will be serialized by the runtime. If this is not the desired behavior, the programmer needs to consider redesigning the data structure, e.g., by breaking-up the object into two sub-objects, one containing the field 'a' and the other containing the field 'b'.

As noted above, conflict detection for static fields happens at the field level. That is, two threads can access two different static fields of the same class concurrently without experiencing contention.

STM.NET handles array objects differently from normal objects, as will be explained in the next section.

When there is any form of conflict in the system, we employ the use of a contention manager (CM) to help decide what action to take next. Contention arises from conflicts detected when attempting to acquire pessimistic locks (pessimistic reads and writes), and from conflicts detected when validating optimistic reads when the transaction prepares to commit.

The contention manager is responsible for a variety of chores: identifying the transactions involved in the conflict, determining what each transaction should do in response to the conflict, and helping each transaction respond (i.e., by providing functionality to wait/rollback/re-execute an operation, etc.). How the CM makes these decisions is referred to as the contention management policy. There are a large number of policies possible, and many studied in various research papers [1], [2]**.**

In this release, we provide an exponential backoff based contention manager. It is fairly simple—the transaction that detects conflict is the transaction that is re-executed, with exponential backoff before re-execution. This CM also performs dynamic read-mode switching when re-executing a transaction. When a transaction starts, it attempts to use optimistic read techniques for all read operations. Once it detects a conflict and rolls back for re-execution,

- if the transaction has a large read set, it is considered as a large transaction, it re-executes using pessimistic reads.
- if the transaction has re-executed many times (reaching a pre-defined threshold), it re-executes using pessimistic reads, too.

The dynamic read-mode switching increases the chance of commit for a large transaction or an old transaction.

11.2.2 Array concurrency

In the CLR, an array is regarded as one single object. In the previous section, we mentioned that contention detection happens at the object-level. As a result, concurrent accesses to different elements of the same array will be serialized by the runtime. However, one of the most common ways to exploit parallelism is to extract it from disjoint accesses to large arrays.

This release does support array concurrency in transactions. That is, our runtime allows multiple transactions to access different chunks of the same array concurrently. At runtime, if the size of an array is larger than a pre-defined threshold, it can be accessed concurrently by multiple transactions. Otherwise, it is still treated as a normal object. At this point, the runtime is responsible for dividing the array into multiple chunks, such that accesses to the different chunks can be concurrent, and accesses to the same chunk are still serialized. Also note that the runtime divides the array into chunks "logically". That is, a chunk consists of an integral number of elements. An element will never be split between two different chunks.

This feature, too, is not for free. It involves a sequential overhead, even when there are only read accesses to the array. For this reason, array concurrency is only introduced for large size arrays.

In this release, for a given array, the runtime uses array concurrency for it only if the size of the array (in bytes) exceeds a pre-defined threshold. The current default size in retail builds of STM.NET is 128 bytes. You can change it through a configuration parameter: **STMMinimumArraySizeForConcurrency**. For example, if your application only uses arrays as read-only inside transactions, you can set the threshold to a large threshold, so that array concurrency is not used, thus the overhead of the array concurrency is avoided.

We would like to stress that even though we provide contention management for arrays at a more granular level than the entire object, it is still considered a race to access any two elements of any array in a conflicting manner, concurrently inside and outside transactions. This was discussed in section 5.2.

11.2.3 Scalability Bottlenecks in this Release

This section will briefly describe two safety mechanisms we have developed. However, not surprisingly, it turns out that these mechanisms are also bottlenecks for achieving scalability. We will discuss the reason why these two mechanisms were introduced, why they appear to be scalability bottlenecks and therefore what we have done or what we are planning to do to solve the problems.

11.2.3.1 Global Versioning

The following table shows an interesting code pattern called ***publication***, which we have previously presented in section 5.1.4.

Initially, x_shared = false, x = 0.

Thread 1	Thread 2

`x   = 17;`
`atomic {`
`      x_shared = true;`
`}`
	`atomic {`
	`    if (x_shared==true){`
	`        localVar  = x;`
	`    }`
	`}`

In the above example, thread 1 initializes “x” to 17, then publishes it by setting “x_shared” to “true”. Thread 2 should read “x” as 17 if it sees “x_shared” is true. This is the publication pattern. Since it is impossible for the non-transactional write (x = 17) to run concurrently with the transactional read “localVar = x”, this program is race-free. As we mentioned in section 5 this pattern is supported by STM.NET, since we provide global-lock equivalence to race-free programs. This non-racy publication pattern works if an STM implementation validates reads and stamps writes using a version number per object (which is a de-centralized and thus scalable mechanism). However, an optimizing compiler or a programmer may change the above code to:

Initially, x_shared = false, x = 0.

Thread 1	Thread 2

x = 17;
atomic {
    x_shared = true;
}
	atomic {
	    tmp = x;
	    if (x_shared == true){
	        localVar = tmp;
	    }
	}

The above example is normally referred to as "***racy publication***". It is "racy" because the non-transactional write (x = 17) can happen concurrently with the transactional read (tmp = x). Since this is a racy program, STM.NET is not obligated to support it (more accurately, STM.NET is not obligated to provide single-lock equivalence in this case). However, it seems possible that an optimizing compiler might transform the non-racy pattern to the racy pattern. It was therefore argued by some that the STM system needs to provide the same semantic guarantee for both patterns.

In order to support the "racy publication" pattern, a global versioning mechanism is normally adopted. For more detail, please refer to [3]. However, global versioning is a centralized mechanism. As a result, it hurts scalability.

Recently, it has become clear that the CLR memory model does not and will not allow the above transformation [4][5]. The C++ memory model seems to be moving towards the same conclusion [6], [7]. As a result, the JIT compiler will not generate such a speculative read (tmp = x). On the other hand, our STM programming model does not provide strong guarantees for racy programs. Therefore, in this release, instead of the global versioning mechanism, we deploy a scalable algorithm—using a version number per object. Thus STM.Net does not suffer the scalability bottleneck of global versioning and still ensures the safety of the non-racy publication pattern.

11.2.3.2 Commit Ticket Barrier

It is well recognized and agreed that an STM system should support the ***privatization*** pattern, the dual of the publication pattern (which we also discussed in section 5.1.4).

Initially, x_shared = true, x = 0.

Thread 1	Thread 2

atomic {
    x_shared = false;
}
r1 = x;
r2 = x;
Debug.Assert(r1==r2);
	atomic {
	    if (x_shared==true){
	        x = 1;
	    }
	}

Basically, thread 1 privatizes "x" by setting "x_shared" to false. If thread 2 observes "x_shared" as false, it does not perform any writes to "x". This is not a racy pattern. The STM system should guarantee the expected semantics in accordance with single-lock equivalence.

However, in a system with optimistic reads, providing the above pattern could only be done by enforcing some sort of ordering on the order at which transactions commit. We provide this ordering using a *commit ticket protocol* that serializes the commit processing of transactions. All non-read-only transactions need to execute a *privatization barrier*, which executes the commit ticket protocol, in order to commit. This protocol essentially ensures that transactions are retired from commit processing in the same order they entered. As you can imagine, this protocol by its very serializing nature affects the scalability of the STM system.

Currently, we are working on an approach with both programming model support and compiler static analysis techniques to let many transactions skip the privatization barrier. It is expected to improve the scalability of the system significantly. This optimization partly will rely on the user taking advantage of AtomicRequired fields. Such fields cannot be "privatized" and thus transactions that exclusively used AtomicRequired fields could be exempted from the commit ticket protocol. We refer to this optimization as the "TVar optimization" ("tvars" or "transactional variables", are the Haskell STM [15] equivalent of AtomicRequired fields in STM.NET). By exploiting programming model support and compiler analysis to enable skipping of the commit ticket protocol for transactions that only access AtomicRequired fields [8], we expect that the scalability will be significantly improved, especially when the number of threads becomes large.

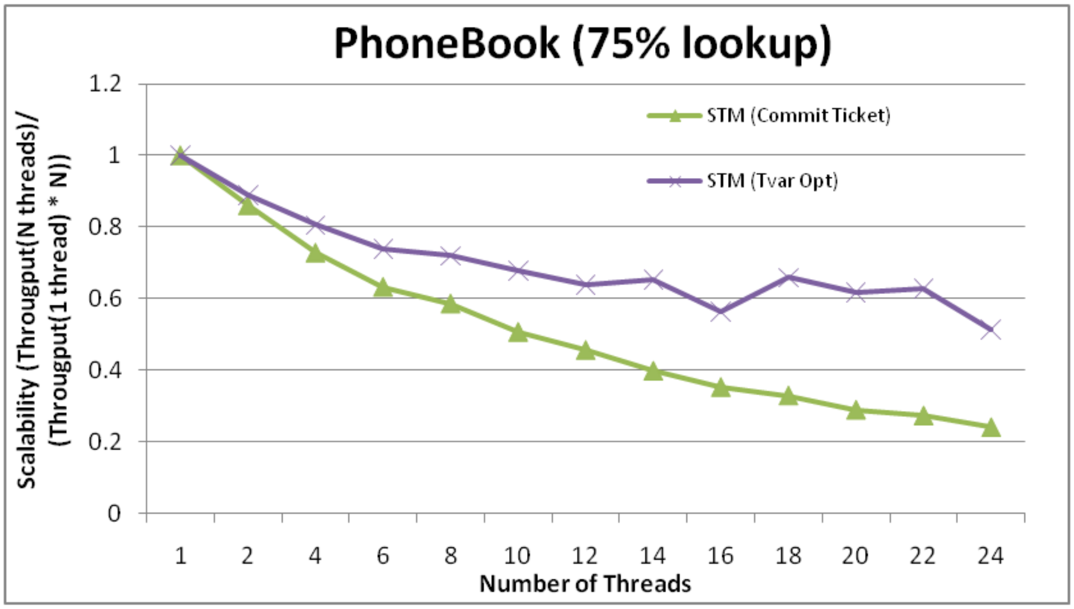
11.3 Benchmarking Results

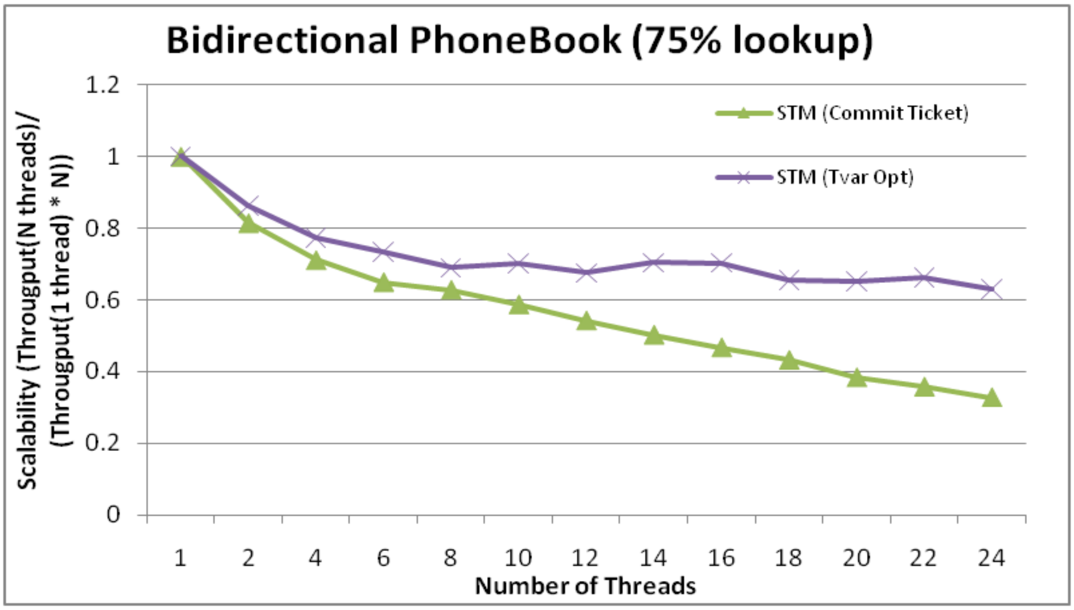
In this section we use two benchmarks to show the scalability of the current release and the scalability expected with the Tvar optimization. Both benchmarks are based on an STM-based concurrent dictionary, which is designed to take advantage of STM.NET system. The code for the STM-based dictionary is available in the Dictionary sample in your STM installation. The STM-based dictionary is implemented using an array of buckets, each of which is a linked-list. The automated fine-grain locking mechanism provided by STM ensures that accesses to different buckets in the hashtable could be executed in parallel. The two benchmarks are:

- A **Phone Book**, which maps a name to a phone number. Internally, it uses one concurrent dictionary.
- A **Bidirectional Phone Book**, which provides bidirectional mapping between a name and a phone number. Internally, it uses two concurrent dictionaries. Most of its public methods involve accesses to both dictionaries. STM is used to ensure the atomicity and isolation of such operations that access multiple concurrent data structures.

First, let us compare the scalability of the STM of this release (with the commit ticketing) to the STM with the TVar optimizations:

- Use the Tvar optimization to avoid the commit ticketing. (Referred to as "TVar Opt" in the figures below)





The above two figures show the scalability of the two benchmarks, using a workload with 75% lookup operations, which are ensured to be read-only. The remaining 25% operations are evenly split between insertion and deletion operations. An insertion is an updating operation if the key is not found in the dictionary (or dictionaries). A deletion is an updating operation if the key is found. In our experiment setup, in average half of the insertions and deletions end up as updating operations. Therefore, in both experiments, we expect 12.5% truly-updating transactions. The scalability is calculated as

$$\frac{Througput(N\ threads)}{Througput(1\ thread) * N}$$

Therefore, scalability of 1.0 represents linear scaling. The closer to 1.0, the better the scalability is. Note the scalability is a relative number. For each curve, the scalability is relative to the 1-thread performance of that version.

The above two figures show that the STM with the TVar optimization achieves better scalability when the number of threads is larger than two. If the percentage of updating transactions increases, the difference is becoming even more pronounced.

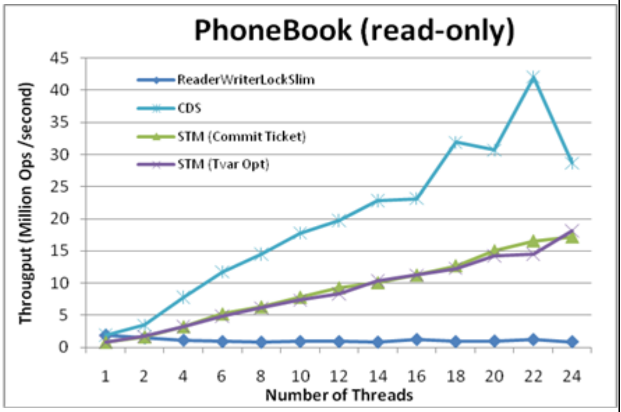
11.4 STM vs. Reader-Writer Lock vs. Concurrency Data Structures (CDS)

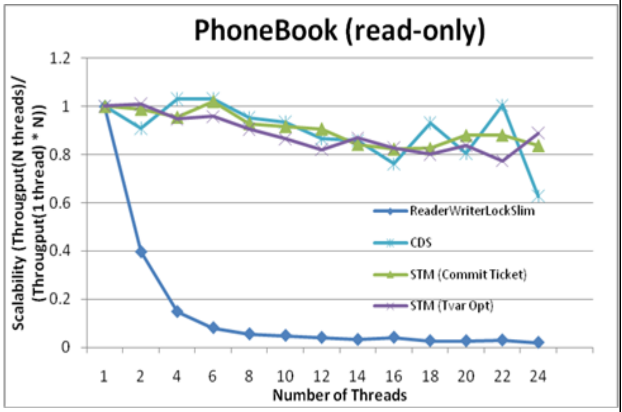
This section compares the scalability of concurrent data structures synchronized using STM to ones using other synchronization mechanisms.

We will continue to use the concurrent dictionary as the benchmark. We compared three different implementations

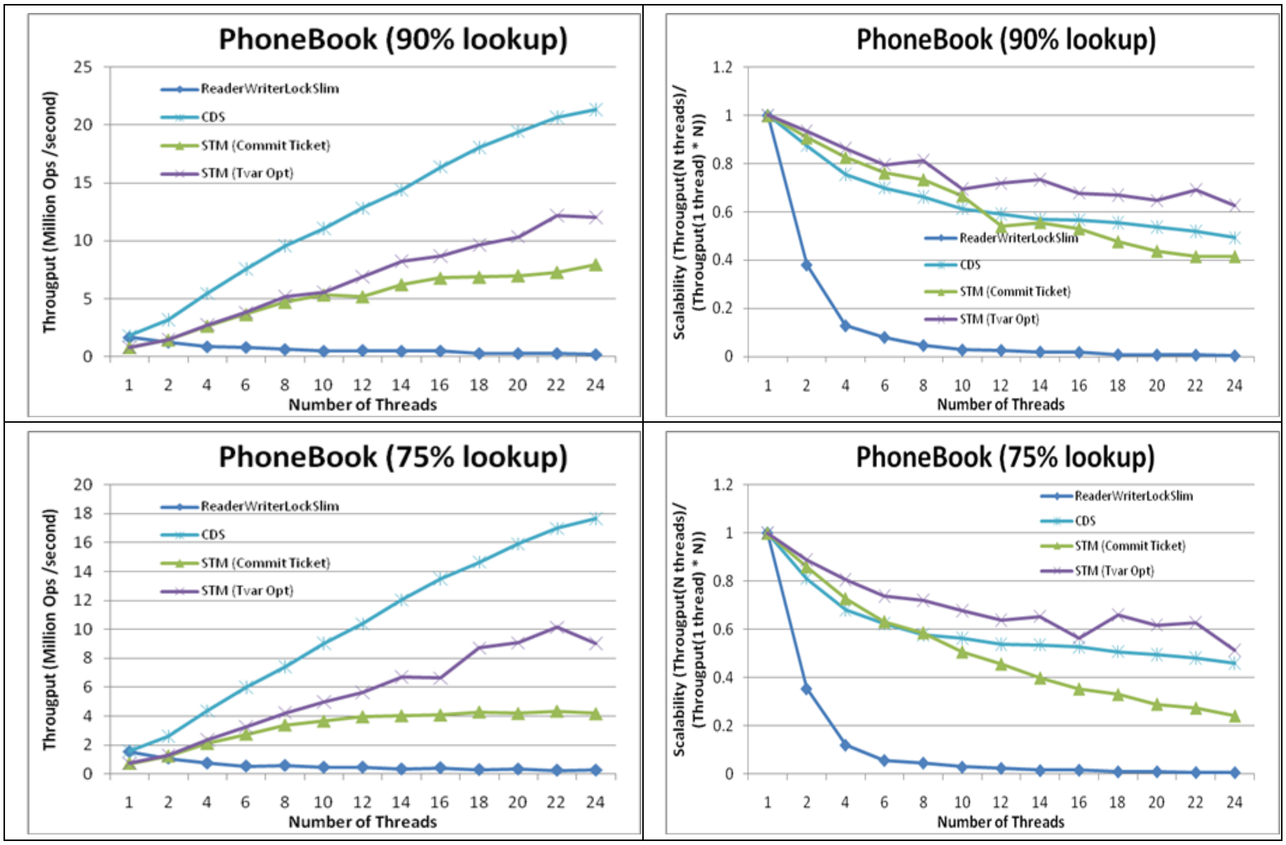
- Directly use the System.Collections.Generics.Dictionary class in .Net Framework 4.0. Use ReaderWriterLockSlim as the synchronization mechanism. For lookup operations, acquires the reader lock to perform the operations. For insertion and deletion, acquires the writer lock.
- Use the System.Collections.Concurrent.ConcurrentDictionary class in .Net Framework 4.0. This class is one of the coordination data structures offered by the Parallel Extensions for .NET. It uses fine-grain locking internally. It is implemented and fine tuned by programmers, who are experts on synchronization, memory model, and multithreading programming.
- The STM version in this release.
- The STM version with the Tvar optimization.

Let us first look at the performance results from the Phone Book benchmark

<table>
<tr><th>Throughput</th><th>Scalability</th></tr>
<tr><td>


</td><td>


</td></tr>
</table>



From the figures above we can see that:

- As expected, the expertly hand-tuned CDS version achieves the best throughput in all cases.
- The ReaderWriterLockSlim-based version does not scale at all, even in the read-only cases. This may be a little surprising to many readers. However, the implementation of ReaderWriterLockSlim, even when acquiring a read lock, requires an update to a shared location in the lock data structure (using interlocked hardware instructions). When many threads are attempting to acquire the read lock frequently, there can be contention on the cache line holding this shared location. When the critical section is not big enough to “space out” the cost of the lock acquisitions, the resulting cache contention prevents scaling.
- The STM based versions have lower throughput than the CDS version because of the sequential overhead. However, the STM version with TVar optimization can achieve similar or better scalability than the CDS version. It is worth noting that the development of the STM based-version of the benchmark was much easier than the development of the CDS version, and required much less expertise. This is because the STM system automates fine-grain locking and contention management for developers, who are not required to deal with the complexity of using fine-grain locks.

Let us also take a look at the Bidirectional Phone Book example. This example is interesting because it requires composition: two dictionaries must be accessed in a coordinated manner atomically[14]. The expert hand-tuned CDS dictionary can only provide atomicity for a single dictionary. Thus, it is not composable and cannot be directly used here. A simple solution is to acquire a reader-writer lock before the proceeding with operations on the two dictionaries. As discussed previously, this approach kills scalability even for the read-only case. Fortunately, we were able to consult our friend, the MSR researcher, Tim Harris, who is a world-class expert on synchronization, to seek advice on how to reuse the concurrent dictionary in this composition scenario. He proposed a solution that uses an optimistic concurrency control scheme on the read path. The code is provided below. We have used white colored font to mark "application logic" and yellow colored font to mark "thread-safety logic". You can see that the method is concerned more with thread-safety logic than with application logic.

```
bool ValidateMapping(string name, string phone)
{
  bool result1, result2;
  string name2, phone2;
  const int MaxOptReadOps = 16;
  int backoffIterations = 1;
  int v1, v2;  int optReadOps = 0;
  while (true)  {
    if (optReadOps <= maxOptReadOps) {
     // Use counting based optimistic scheme
     v1 = version;
     result1 = name2phone.TryGetValue(name,
                          out outPhone);
     result2 = phone2name.TryGetValue(phone,
                          out outName);
     v2 = version;
     if( (v1 != v2) || ((v1 & 1) != 0))  {
       // An update is in progress.
       Thread.SpinWait(backoffIterations <<
                       optReadOps);
       optReadOps++;
     } else { break; }

   } else {
    // We have tried the optimistic scheme
     // a few times, we switch to a
     // pessimistic mode by acquiring a lock
    rwls.EnterReadLock();
    try {
```

[14] In a straight-forward implementation one would use the first dictionary to map persons' names to their phone numbers, and the other dictionary to map persons' phone numbers to their names. The challenge would to keep those to data structures in-sync while maintaining good scalability for concurrent access.

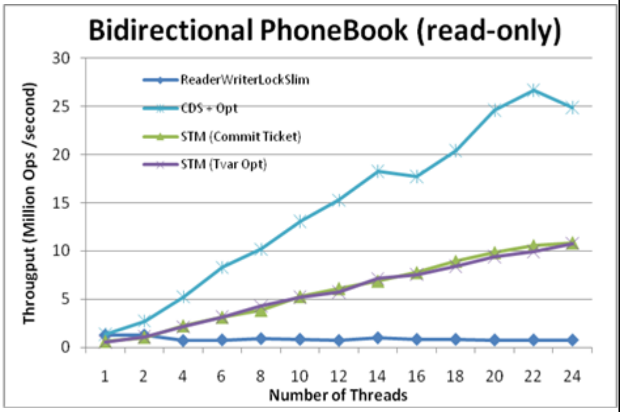
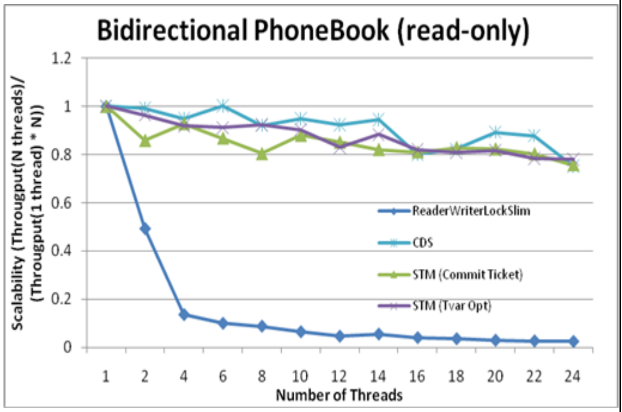
```
        result1 =  name2phone.TryGetValue(
        name, out outPhone);
        result2 =  phone2name.TryGetValue(  phone, out outName);
     } finally { rwls.ExitReadLock(); }
     break;
   } // while
   // Check results
   . . .
 }
```

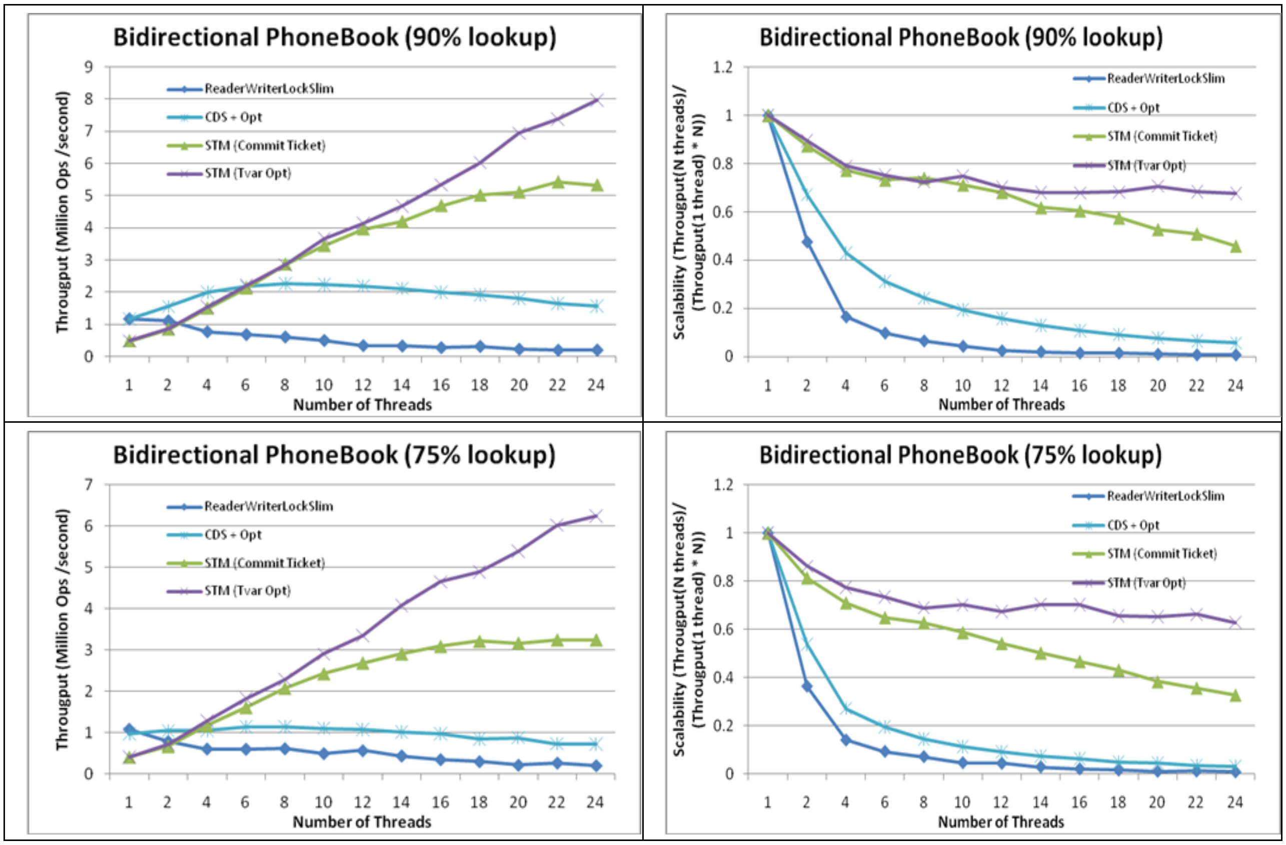
Obviously, this code is quite complicated, and is far beyond what most programmers would code-up, if they were given this task. Moreover, though it solves the composition-of-two-dictionaries case, it cannot be easily generalized to compose N dictionaries, where N is an arbitrary number, or generalized to support other lookup or insert semantics. Finally, this solution still only allowed concurrent lookups, but all potentially modifying operations have to serialize on a write-lock.

On the other hand, the STM-based version, below, is extremely simple and the thread-safety tax is limited to a single atomic block.

```
bool IBidirectionalPhoneBook.ValidateMapping(string name, string phone)
{
    string outPhone = null;
    string outName = null;

    atomic {
        outPhone = name2phone.TryGetValue(name);
        outName = phone2name.TryGetValue(phone);
    }
    // Check results...
}
```

Let us take a look at the measured throughput and scalability for the bidirectional phone book example.

Throughput	Scalability
	



The CDS and optimistic concurrency control based version only achieve best throughput on the read-only case. In all other cases, the STM version performs better than the CDS version as well as the reader-writer lock version significantly. So in other words STM allowed here the reuse of existing code in a very straight forward manner while retaining excellent scalability characteristics.

11.5 Performance Tips

This section provides advanced users with some useful tips on performance.

11.5.1 “readonly” Field Modifier

C# has a field modifier called “readonly”. When a field declaration includes a readonly modifier, assignments to the field introduced can only occur as part of the declaration or in a constructor in the same class. In other words, once constructed, the readonly field will not be written to any more. It is therefore safe for the transaction to read the value of the field directly without transacting it[15]. Therefore, no read barriers are needed for such reads. This optimization reduces sequential overheads of transactions that access readonly fields. Therefore, it is encouraged to declare fields as readonly whenever possible. This is a good practice in general as it promotes the notion of immutability which is instrumental not only for STM, but in general allows reasoning more easily about program invariants.

[15] Special care is taken for transactional accesses to readonly fields in the constructor.

11.5.2 x = value; → if (x != value) x = value;

When a transaction performs a write, such as:

x = value;

it dooms all other concurrent transactions that read the same “x”. However, in some applications, it is possible that “x” is equal to “value” most of the time. In such cases, you can write the assignment above as

if (x != value) x = value;

As a result, if “x” is equal to “value”, the STM only executes a transactional read, instead of a write. Therefore, the current transaction does not doom other transactions that read x. This optimization can sometimes reduce runtime contention and improve scalability.

11.5.3 Avoid Contention on an Object: the Deque Example

STM.NET, as a general rule, detects conflicts at the object level. Therefore, if two threads are transactionally accessing two different fields of the same object concurrently, the transactions are still in conflict with each other and thus are serialized. This is in many cases acceptable. However, in some scenarios, more attention needs to be paid.

For example, assume we want to develop a concurrent deque based on a doubly linked-list. The idea is that one or more threads always push/pop items from one end (let’s say “head”); other threads always push/pop items from the other end (let’s say “tail”). Ideally, if the queue contains enough items, a thread working on the head should not conflict with another thread working on the tail.

Such a deque data structure is normally abstracted with a class, as:

```
class Deque
{
    Node head;
    Node tail;
}
```

Unfortunately, since “head” and “tail” are the fields of the same Deque object, a write to one of them will conflict with accesses to the other field. This is definitely not what we want when we design the deque data structure.

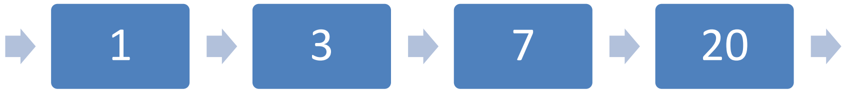
To avoid such conflicts, we can do the following instead:

```
class NodeHolder {  public Node node; }

class Deque
{
    private readonly NodeHolder head = new NodeHolder();
    private readonly NodeHolder tail = new NodeHolder();

    public Head { get { return head.node; } set { head.node = value; }
    public Tail { get { return tail.node; } set { tail.node = value; }
}
```

In this way, the head and tail are held by two different objects. Therefore, there is no contention when accessing them concurrently.

11.5.4 Benign Conflicts: the Linked-List Example

Assume we use an ordered linked-list to implement an integer set, which supports insertion, deletion, and lookup operations.



Assume there is a transaction executing a lookup operation to find integer 20. It needs to search from the head of the linked list. Therefore, when it reaches the node that contains 20, its read set has contained the nodes that hold 1, 3, 7, and 20.

Assume that at the same time there is another transaction that inserts integer 2 into the set and then commits before the lookup transaction commits. As a result, it needs to modify the “next” pointer of the node holding “1”, so it writes to node 1. Note that node 1 is in the read set of the lookup transaction, which is thus doomed and has to be rolled back to re-execute.

If we look at these two concurrent transactions at an abstract level, they do not conflict. However, they conflict at the concrete implementation level, and are serialized by the runtime.

Unless advanced techniques, such as open nesting [9], or abstract nested transactions [10], are used, STM doesn’t excel at such scenarios. However, in terms of scalability, it is worth noting that STM does not perform worse than a single-lock based-version even in such scenarios.

Our advice would be to prefer data structures that, unlike a singly linked list, are more distributed and contain less central “choke-points”. Trees and hashtables are good examples of such data structures.

11.5.5 ByRef

The CLR allows the use of *managed* and *unmanaged pointers.* Managed pointers are exposed in the C# programming language as **ref** and **out** parameters. Unmanaged pointers may only be used in **unsafe** code; the **fixed** statement *pins* an object to its current heap location and allows pointers into the pinned object to be created. The ability to create managed or unmanaged pointers creates a problem for an STM like ours with object-based locking:

1. if we access memory via such a pointer, how do we decide whether the pointer is to a local variable, a static variable, or a heap object? We need to know that since we "transact" accesses differently in each case, and,
2. if the pointer is into a heap object, how do find the start of the object in order to acquire the transactional lock that protects the object?

We solve these problems by doing extra static analysis to track information about the targets of pointers, and transmitting this information for pointer/**ref** arguments from callers to callees. We've worked hard to optimize this as much as possible, especially to ensure that this does not impose undue cost when you're *not* using transactions (as would be required if TM were ever to become part of a production CLR). But there's still some significant cost associated with transactional accesses via **ref** arguments and pointers. In one experiment we compared two different signatures for a **lookup** operation on a dictionary. In one case, the *Value* type in the *Key*→*Value* mapping is required to be an object type, and mappings to null are not allowed, so that **lookup** can return null to indicate failure. The other signature returns a **bool** to indicate success or failure of the query, and, on success, returns the *Value* found as an **out** parameter. The latter style is currently 40% slower because of overhead associated with the **ref** parameter .We may well decrease this overhead in future releases.

The bottom line is that if you can avoid the usage of ref and out parameters in perf-sensitive code, then do so. It will be more efficient to create and pass around "result objects" that can capture structures and multiple result values, rather than to use out and ref parametes.

Unsafe code is always discouraged, and with STM this is the case, too.

For more details on how we transact pointer-based accesses in STM, please refer to [16].

11.6 Using Event Tracing to Diagnose STM Application

Event Tracing for Windows (ETW) is a tracing and profiling technology from Microsoft that is used to profile and troubleshoot many Microsoft applications. For example, the .NET Framework provides ETW events that can be consumed using the standard ETW tools.

STM.NET provides additional events that will allow you to glean important information about the dynamics of your program execution, find sources of contention and more. In this section we describe the steps for using ETW tracing to get the trace of STM events. ETW is available only on Windows Vista and later releases of Windows; we have only tested our use of ETW on the Windows Vista 32-bit edition.

In order to capture, parse and view ETW events you will need to install the Windows Performance Tools Kit from this link. For details about this Toolkit, please read Windows Performance Tools Kit Documentation (MSDN). In particular, you need to use a command line tool called xperf. Please read Xperf Command Line Tool in Detail (MSDN) and Xperf Command-Line Reference.

In order to get the trace of CLR STM events, please follow the steps described below.

1. Launch a “cmd” command line shell. Please use “run as administrator”.
2. If it is your first time using ETW after the installation, execute “wevtutil im C:\Windows\Microsoft.Net\Framework\v4.0.20506\CLR-ETW.man”.
3. In the shell, execute “set COMPLUS_ZapDisable=1”, and “set COMPLUS_DefaultVersion=v4.0.20506”
4. In the shell, execute “set PATH=%PATH%;”%ProgramFiles%\Microsoft Windows Performance Toolkit””. As a result, you’ll now have “xperf” in your PATH.
5. Execute “set COMPLUS_STMEtwTraceLevel=3”, this environment variable controls what events are recorded in the trace. More detail will be given later.
6. Execute “xperf -start clr -on e13c0d23-ccbc-4e12-931b-d9cc2eee27e4 -f clr.etl”, {e13c0d23-ccbc-4e12-931b-d9cc2eee27e4} is the GUID of the CLR public events provider. STM events are generated by this provider.
7. Execute “xperf -on base+cswitch -f kernel.etl”. This command instructs xperf to also include base kernel events and the context switch events.
8. Run your application in the STM Shell.
9. Execute “xperf -stop". This dumps the kernel events into kernel.etl.
10. Execute “xperf -stop clr”. This dumps the CLR public events, including STM events, into clr.etl.
11. Execute “xperf -merge kernel.etl clr.etl clrevents.etl”. This command merges the kernel.etl and clr.etl, and outputs clrevents.etl.
12. Execute “xperf -i clrevents.etl -o clrevents.csv”. Now, you get a readable file – clrevents.csv.
13. Post process the csv file to make it easier to understand STM events. Execute “%ProgramFiles%\Microsoft.Net\STM\STMEtwTracePostProcessor.exe” clrevents.csv[16]. This step generates
    - A new “clrevents.csv” file, which contains all the kernel and CLR events, including STM events. The pre-post-processing file is backed up as “clrevents-bak.csv”
    - “clrevents-stm.csv”. This trace file only contains STM events.
    - “clrevents-stm-summary.txt”. This text file provides a summary of STM events.

In step 10, if you get any error/warning (such as loss of events) from xperf, please read Xperf Command Line Tool in Detail (MSDN) and Xperf Command-Line Reference on how to use the tool.

The following table summarizes the STM events provided now.

[16] On x64 systems, use %ProgramFiles(x86)% instead of %ProgramFiles%

Event	Description	User Data Layout
TXStart	A transaction starts	<TX pointer[17], TxSiteId[18], TimeStamp[19]>
TXCommit	A transaction commits	<TX pointer, TimeStamp, Read set count, Write set count>
TXRollback	A transaction rolls back	<TX pointer, Read set count, Write set count>
TXRetry	A transaction enters Retry	<TX pointer>
TXSleep	A transaction suspends	<TX pointer, Milliseconds to sleep, Read set count, write set count>
TXWakeup	A transaction wakes up	<TX pointer>
TXConflictOnWrite	A transaction detects a conflict upon a write	<TX pointer, Object address[20], Enemy TX pointer, CM Decision[21]>
TXConflictOnReadOpt	A transaction detects a conflict upon attempting an optimistic read	<TX pointer, Object address, Enemy TX pointer, CM Decision>
TXConflictOnReadPes	A transaction detects a conflict upon attempting a pessimistic read	<TX pointer, Object address, Enemy TX pointer, CM Decision>
TXDoomTx	A transaction dooms another transaction	<Requesting TX pointer, Doomed TX pointer>
TXFoundDoomed	A transaction detects that it's doomed	<TX pointer>

The environment variable COMPLUS_STMEtwTraceLevel used in Step 5 controls what types of events are traced.

COMPLUST_STMEtwTraceLevel	Events
1	TXStart, TXCommit, TXRollback, TXRetry
2	TXSleep, TXWakeup, and Level 1 events
3	TXConflictOnWrite, TXConclictOnReadOpt, TXConflictOnPes, TXDoomTx, TXFoundDoomed, and Level 1 and 2 events

[17] TX pointer is an address pointing to the runtime transaction object, which manages the transaction state.

[18] TxSiteId identifies a unique lexical transaction site.

[19] This field in all ETW events is unused right now. It is always 0. Timestamp is the global version number that the transaction reads when it starts.

[20] The address of the object that causes the conflict.

[21] The decision of the CM: 0 – try the read/write operation again, 1 – rollback the transaction, 2 – doom the enemy transaction.

12 References

[1] William N. Scherer III, and Michael L. Scott, Advanced Contention Management for Dynamic Software Transactional Memory, in Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing, 2005

[2] Rachid Guerraoui, Maurice Herlihy, Michal Kapalka, and Bastian Pochon, Towards a Theory of Transactional Contention Managers, in Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing, 2005.

[3] Dave Dice, Ori Shalev, and Nir Shavit, Transactional Locking II, in Proceedings of the 20th Internal Symposium on Distributed Computing, 2006.

[4] Chris Brumme, Memory Model, http://blogs.msdn.com/cbrumme/archive/2003/05/17/51445.aspx

[5] ECMA-335 4th Edition: Common Language Infrastructure $^{(CL}$I), http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf

[6] Hans-J$^{.}$ Boehm, and Sarita V. Adve, Foundation of the C++ Concurrency Memory Model, in Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2008.

[7] Herb Sutter, Prism: A Principle-Based Sequential Memory Model for M^{ic}rosoft Native Code Platforms, 2006. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2197.pdf

[8] STM Programming Model Document

[9] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman, Open Nesting in Software Transactional Memory, in Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2007.

[10]Tim Harris, Sjdjan Stipic, Abstract Nested Transactions, in Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing, 2007

[11]Sing^{le} Global Lock Semantics in a Weakly Atomic STM [slides] Vijay Menon (Intel Labs), Steven Balensiefer (University of Washington, Seattle), Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha and Adam Welc (Intel Labs)

[12]Martín Abadi, Andrew Birrell, Tim Harris, Michael Isard:
Semantics of transactional memory and automatic mutual exclusion. 63-74
Electronic Edition (ACM DL) BibTeX

[13]Tim Harris, Mark Plesko, Avraham Shinnar, David Tarditi. Optimizing Memory Transactions June 2006 *PLDI '06: ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*

[14] Jim Gray, Andreas Reuter. Transaction Processing: Concepts and Techniques

[15]Tim Harris, Simon Marlow, Simon Peyton-Jones, Maurice Herlihy. Composable memory transactions *PPoPP 2005*

[16]David Detlefs, and Lingli Zhang, Transacting Pointer-based Accesses in an Object-based Software Transactional Memory System. Transact 2009.

13 Index

API
- atomic 11, 17, 20
- **Atomic.Do** 20
- Atomic.Do(AtomicAction uow) 54
- **Atomic.DoAfterCommit** 61
- **Atomic.DoWithCompensation** 62
- Atomic.Retry 46
- AtomicMarker 21
- Parallel.For 25
- *Task Parallel Library (TPL)* 15

Atrtribute
- AtomicMarshalReadonlyAttribute 56
- **AtomicNotSupportedAttribute** 30
- AtomicRedirectAttribute 58
- **AtomicRequiredAttribute** 30
- **AtomicSupportedAttribute** 30
- AtomicSuppressAttribute 55, 57
- **AtomicUncheckedAttribute** 30

Concepts
- abort 51
- array concurrency 68
- *atomic block* 17
- atomic compatibility 30
- *atomic context* 32
- compensated actions 62
- Concurrency Data Structures (CDS) 73
- conflicting accesses 23
- contention management 67
- deferred actions 61
- *effective atomic compatibility value* 30
- failure atomicity 18
- *functional interface* 57
- *granularity of data access* 27
- I/O in transactions 54
- immutable types 57
- isolation 15
- nesting 19
- privatization 25
- publication 25
- *race condition* 23
- race freedom 24
- reader-writer lock 73
- readonly 77
- redirect 58
- re-execution 51
- retry 51
- rollback 51
- side-effecting actions 54
- static contract checking 60
- structured parallelism 25
- suppressing transactions 55
- system.transactions 52

Configuration
- **app.config** 8, 43
- COMPLUS_STMEtwTraceLevel 82
- Complus_STMExceptionOnViolation 43
- Complus_STMLogOnViolation 43
- Complus_STMRuntimeCheckLevel 42
- Complus_STMViolationLogFile 43
- COMPLUST_STMEtwTraceLevel 82

Delegate
- AtomicAction 54

Enumeration
- **AtomicContext** 32
- SupressValidationOption 57
- SystemTransactionPromotion 54

ETW Event
- TXCommit 82
- TXConflictOnReadOpt 82
- TXConflictOnReadPes 82
- TXConflictOnWrite 82
- TXDoomTx 82
- TXFoundDoomed 82
- TXRetry 82
- TXRollback 82
- TXSleep 82
- TXStart 81

Exception
- AtomicContractViolationException 42

Helper class
- *TransactionalOperation* 63

Internal Exception
- AbortReexecException 12
- HoistingException 53
- SystemReexecException 53

Library
- mscorlib.dll 9
- System.Transactions.dll 9

Namespace
System.TransactionalMemory 20
System.Transactions 52

VS2008 Template
TMConsoleApplication 8