Software Transactional Memory for .NET

Bachelor Thesis

Department of Computer Science
University of Applied Science Rapperswil

Fall Term 2015

Author(s):	Christoph Amrein, Timothy Markiewicz
Advisor:	Prof. Dr. Luc Bläser, HSR
Project Partner:	Institute for Software, Rapperswil
External Co-Examiner:	Dr. Felix Friedrich, ETH Zürich
Internal Co-Examiner:	Prof. Dr. Andreas Rinkel, HSR

Abstract

Classical multi-threaded programming requires explicit synchronization of shared resources, being a highly challenging task for many software developers: On the one hand, identifying shared resources is non-trivial, easily leading to race conditions in case of under-synchronization. On the other hand, synchronization naturally bears the risk of deadlocks and starvation.

Transactional memory is an alternative concept, significantly simplifying concurrent programming. It employs a descriptive programming model using the notion of *transactions,* inspired by database systems. A transaction constitutes an atomic sequential execution that is automatically isolated to other concurrent transactions. The runtime system guarantees the correct transactional execution, typically by using an optimistic concurrency control scheme.

We have developed a practical transactional memory programming model and runtime system for the .NET framework. Our experimental evaluation shows that the solution is superior to existing .NET transaction frameworks in terms of performance, correctness, and ease of use. Moreover, it employs Intel TSX hardware transactional memory to increase performance. Last but not least, a refactoring tool for Visual Studio C# assists programmers on migrating existing code to the transactional model.

Management Summary

Introduction

Today's programming languages support the concept of multi-threading, meaning that a single program can perform multiple tasks simultaneously. As soon as multiple threads access the same resources, these operations need to be synchronized, otherwise this leads to inconsistent data. Correctly synchronizing these memory accesses requires experienced programmers. A promising model to simplify such synchronization is *Software Transactional Memory (STM)*. It focuses on what needs to be synchronized rather than how.

Explicit Synchronization

Explicit synchronizations mix the domain logic with thread synchronization logic, making the code hard to maintain. There are different approaches to accomplish such synchronization, either by preventing other threads from accessing the same memory or by utilizing the limited number of available atomic instructions. This increases the risk of a thread is not being able to progress, or even worse, of multiple threads blocking each other completely.

Transaction Model

The vision of the transaction model is, is that the developers are only in charge of marking sections requiring synchronization as transactions. Synchronization is handled fully automatically by the desired runtime system. This runtime is in charge of resolving conflicts and guaranteeing that transactions eventually complete. There is even the possibility that a transaction waits for a specific condition to be fulfilled before it continues in progress.

Programming Model

With *[STM]4.NET*'s straightforward and seamless programming model, there is no training of developers needed. If a section needs to be executed as a transaction, one would simply need to enclose this section in the transaction-statement.

Implementation

The implementation consists of two parts. A runtime which ensures strict consistency during an application's lifetime. Under the hood is a lock-based algorithm. To improve performance, the implementation tries to hardware accelerate transactions using Intel TSX. The second part is the post-compilation instrumentation. This instrumentation enriches compiled code with instructions required to track access to shared memory.

Evaluation

The evaluation has shown that *[STM]4.NET*'s programming model is superior to the one of its competitors. Different to other solutions, with *[STM]4.NET* a developer can easily adopt his existing solution to transactions. The executed benchmarks illustrate that *[STM]4.NET* surpasses its competitors on the .NET platform in terms of overall performance.

Conclusion

[STM]4.NET proves to be a good alternative to conventional, explicit synchronization because of its simplicity, yet there is still room for improvement. First of all, it lacks simple collection types for use within transactions. Moreover, the performance can potentially be improved by using more sophisticated algorithms.

Content

1. Introduction

Software development involves concurrent programming to a substantial extent. For this purpose, object-oriented programming languages offer multi-threading, where threads run concurrently, i.e. in arbitrary parallel and/or with non-deterministic interleaving unless synchronization is applied. A thread thereby executes a sequence of statements operating on shared resources, represented as objects located in the heap. Higher parallel programming abstractions such as thread-pools, parallel statements, asynchronous and reactive models also map to the multi-threaded execution. It is in the nature of this programming model that several concurrency issues may arise. In order to prevent concurrency issues, accesses to shared resources must be *synchronized*. Prevalent synchronization strategies are flawed and introduce new concurrency issues. As they are implemented explicitly, it is a programmer's responsibility to ensure that all shared resources are synchronized adequately. Furthermore this requires that the association between data and synchronization must be documented or enforced by conventions [1]. It requires experienced and sophisticated programmers to correctly implement synchronization, especially in large systems.

Software Transactional Memory (STM) is a promising model to significantly simplify concurrent programming. By the usage of a transaction concept similar to database systems, STM implicitly controls synchronization across threads, therefore it prevents many concurrency issues by default. Moreover, it embraces a descriptive approach as opposed to explicit synchronization. STM enjoys a sound theoretical background and a lot of research has been conducted over the past decades. However, this has not yet yielded a practical STM implementation for a mainstream programming language. The reasons therefore are manifold but primarily concern performance, correctness, and ease of use.

1.1 Goals and Challenges

The goal of this thesis is to develop a practical STM implementation for one of the most popular programming environments, the .NET Framework [2]. The implementation must be reasonably efficient, correctly implemented and easy to adopt by programmers. The implementation should work for any CLR-compliant [3] language but is focused, especially regarding tool support, particularly on C#. Additionally we aim to provide hardware transactional memory support in order to increase performance.

The main challenge is to ensure strict correctness by obeying the properties of atomicity, consistency, and isolation with as little performance overhead as possible. Therefore our goal is to provide an overall solution which proves to be a superior alternative to existing ones, at least for the .NET Framework.

2. Explicit Synchronization

As mentioned in chapter 1, concurrent programming requires synchronization of shared resources. Executing threads concurrently potentially leads to several concurrency issues:

- Race Conditions
  A low-level race condition [4], also called data race, is an erroneous program behavior caused by insufficiently synchronized shared resource accesses by multiple threads. The notion of a high-level race condition refers to a situation where each access to shared resources is synchronized, yet still the program behaves erroneously due to conflicts on a higher semantic layer.
- Deadlocks
  A deadlock [5] is a constellation where multiple threads mutually block each other in a way that none of them can continue anymore. A livelock is a specialized form of deadlock where threads indefinitely loop because they are blocked by other threads.
- Starvation
  A starvation [6] is a situation where a thread indefinitely awaits a condition that is continuously granted to other threads whenever fulfilled.

Concurrency issues arise non-deterministically, meaning they are very hard to identify, reproduce, and to prove that they cannot occur in a program. Therefore concurrency issues are program errors and a constant threat to the integrity and consistency of a program.

Figure 1 shows a scenario where two threads concurrently deposit money on a shared object *BankAccount*.[1] Thread 1 deposits 100 and thread 2 deposits 50, thus yielding a final balance of 150.


```
1. class BankAccount {
2.   private int balance = 0;
3.
4.   public void Deposit(int amount)
5.     this.balance += amount;
6.   }
7. }
```

Figure 1 Race Condition

When a thread calls the *Deposit* method, the *balance* field is updated with the new value at line 5. It is worth noting that even a single-line statement, such as line 5 of Figure 1, can map to a sequence of multiple machine instructions, as outlined in Figure 2.

[1] The bank account example and related diagrams are gratefully adapted from the lecture „Parallel Programming“ by L. Bläser, HSR.

Figure 2 Low-level Instructions

In presence of multi-threading, these instructions can now be arbitrarily interleaved or executed in parallel, resulting in potential race conditions. Figure 3 sketches a possible scenario which results in an erroneous value of *balance*. Each row in Figure 3 represents a progress in time.

Thread 1	balance	Thread 2
read(balance) balance => 0	0	
	0	read(balance) balance => 0
add result => 100	0	
	0	add result => 50
write(balance)	100	
	50	write(balance)

Figure 3 Deposit Race Condition

In the sketched scenario, the two threads execute a single instruction after each other. The final *balance* is 50 since the deposit by thread 1 is overwritten. Thread 2 does not consider the deposit by thread 1 since it reads the balance field before thread 1 updates it.

By synchronizing accesses to shared objects, such low-level race conditions can be prevented. Prevalent synchronization strategies are implemented explicitly: Within the program code, each shared object needs to be explicitly synchronized. This leads to a mix of statements handling synchronization and statements that solve the actual domain problem. This may be undesired since it decreases the cohesion along with increasing complexity in any place where synchronization is introduced. Moreover, it remains a programmer's responsibility that all necessary resources are adequately synchronized. Figure 4 shows how the *Deposit* method can be synchronized by using a monitor lock in C# [7]. Note that in this example the shared object *BankAccount* is explicitly locked by the *lock(this)* statement, whereas *this* refers to the current instance of *BankAccount*.

```
class BankAccount {
  private int balance = 0;

  public void Desposit(int amount) {
    lock(this) {
      this.balance += amount;
    }
  }
}
```

Figure 4 Bank Account Locked

Locking realizes mutual exclusion [8] that introduces at the same time the risk of deadlocks. To demonstrate this, Figure 5 enhances the example with a *Transfer* method which transfers money from one *BankAccount* instance to another.

```
class BankAccount {
  private int balance = 0;

  public void Deposit(int amount) {
    lock(this) {
      this.balance += amount;
    }
  }

  public void Transfer(int amount, BankAccount target) {
    lock(this) {
      this.balance -= amount;
      target.Deposit(amount);
    }
  }
}
```

Figure 5 Bank Account Transfer

The *Transfer* method in Figure 5 acquires two locks in a nested way. First the instance transferring money is locked, followed by locking the target by calling its deposit method. Figure 6 sketches a multi-threaded scenario which leads to a deadlock. Thereby each row represents a progress in time.

Thread 1	Thread 2	Locked Objects
transfer => lock(Account 1)		Account 1
	transfer => lock(Account 2)	Account 1, Account 2
target.deposit => blocked		Account 1, Account 2
	target.deposit => blocked	Account 1, Account 2

Figure 6 Bank Account Dead Lock

Deadlocks occur due to a cyclic flow of information and the use of nested locking without any timeout handling when a lock cannot be immediately acquired. Figure 7 visualizes this by creating a Holt graph which is cyclic, if and only if a deadlock has occurred [9].

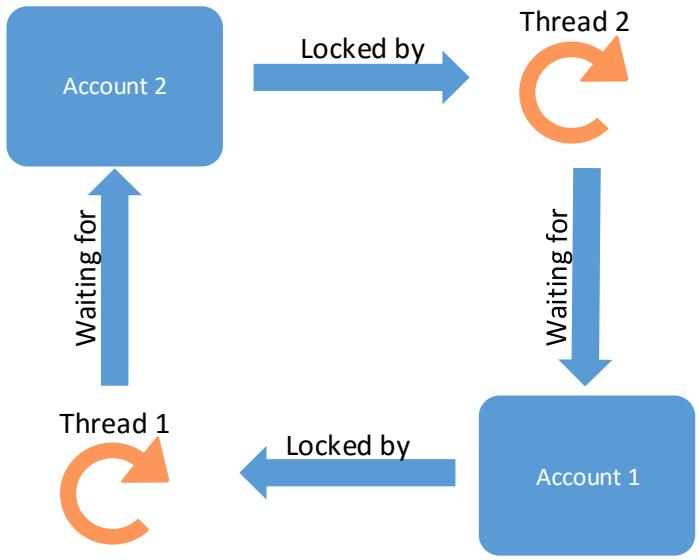


Figure 7 Dead Lock Graph

The usage of locks is called pessimistic concurrency control because locks are always acquired beforehand. However runtime cost for synchronization is expensive and thus undesired in circumstances where no conflicts would occur.

Another common way to implement explicit synchronization is to use atomic statements which are offered by almost any programming language. Thereby, synchronization is handled in hindsight and thus is an optimistic approach. Before shared resources are updated, atomic statements check whether a conflict has occurred. Figure 8 shows the usage of an atomic instruction to update the *balance* of the *BankAccount*.

```
public void Deposit(int amount) {
  bool success = false;
  int balance;
  while(!success) {
    balance = this.balance;
    int newBalance = balance + amount;
    success = atomicCompareExchange(balance, newBalance);
  }
}
```

Figure 8 Lock free explicit synchronization

This solution does not suffer from deadlocks, but can lead to starvation. It is possible that other threads keep outpacing the thread, thereby constantly changing the value of *balance*. This way the thread will loop indefinitely because *atomicCompareExchange* will always fail since *balance* has been changed.

3. Transaction Model

Transactional Memory is a programming model for the synchronization of shared resources. It was first introduced by Maurice Herlihy and Eliot Moss [10] and uses a concept of *transactions* similar to the ones provided by database management systems. In favor of explicit synchronization, transactional memory uses a descriptive approach. Thereby blocks of statements are enclosed in a *transactional block.* Everything in such a transactional block is then adequately synchronized at runtime by the underlying implementation. Transactions are implicitly synchronized and do not suffer from race conditions, deadlocks and starvation. Figure 9 shows how the bank account example is solved by using transactions.

```
class BankAccount {
  private int balance = 0;

  public void Deposit(int amount) {
    Transaction {
      this.balance += amount;
    }
  }

  public void Transfer(int amount, BankAccount target) {
    Transaction {
      this.balance -= amount;
      target.Deposit(amount);
    }
  }
}
```

Figure 9 Transaction

The statements in the *Deposit* and *Transfer* method are now enclosed within a transactional block. It is the responsibility of the underlying implementation to ensure that these transactions are executed with adequate synchronization. It is obvious that, due to the descriptive nature of transactional memory, using transactions is a much simpler way to introduce synchronization than with prevalent explicit synchronization strategies. Figure 9 also demonstrates another powerful feature of transactions – they can easily be composed by nesting transactions. Nested transactions are described in more detail in chapter 3.3.

3.1 Atomicity, Consistency and Isolation

Transactions have several properties, stemming from database systems, commonly abbreviated as ACID [11]:

- Atomicity
  When a transaction completes, it either commits, making all its changes visible to the rest of the system instantaneously, or it aborts, causing its changes to be discarded.
- Consistency
  Consistency requires that a transaction takes the system from one valid state to another. Transactions never cause the system to be in an invalid or temporary state.
- Isolation
  The concurrent execution of transactions must result in a system state that could also be obtained by a serial execution of transactions.

ACID also defines *durability* as the fourth property. However, durability is primarily involved in database transactions and does not bear any relevance since transactions discussed in this work operate on non-persistent main memory.

In order to provide correct isolation, the schedule of transactions must be *serializable*. A serializable schedule implies that the effects of that schedule have the same effect as a schedule of sequentially executed transactions. To verify whether a schedule is serializable, it is analyzed by identifying conflicting transactions. Formal representation of a schedule uses the following notation to describe events:[2]

- $r_1(x)$: Transaction T_1 reads variable x in shared memory.
- $w_1(x)$: Transaction T_1 writes variable x in shared memory.
- c_1: Transaction T_1 commits.

Two accesses (i.e. variable reads or writes) conflict with each other if at least one operation is a write operation and both access the same variable. Figure 10 shows the conflict matrix denoting when two accesses conflict with each other.

	$r_1(x)$	**$w_1(x)$**
$r_2(x)$	No conflict	Conflict
$w_2(x)$	Conflict	Conflict

Figure 10 Conflicts

Figure 11Figure 12 shows a non-serializable schedule of two transactions. The arrows indicate the conflicting operations between transactions. This schedule is then translated to a *serialization graph* [12]. Serialization graphs model transactions as nodes and conflicts as edges. The resulting serialization graph is cyclic, if and only if, this schedule is non-serializable. A non-serializable schedule results in a race condition.

Figure 11 Non-Serializable Schedule.

Figure 12 is a more complex example consisting of three transactions concurrently accessing the two shared objects x and y. However, even though this schedule is highly interleaved it results in an acyclic serialization graph. Hence, executing this schedule does not lead to race conditions and could be executed without any synchronization.

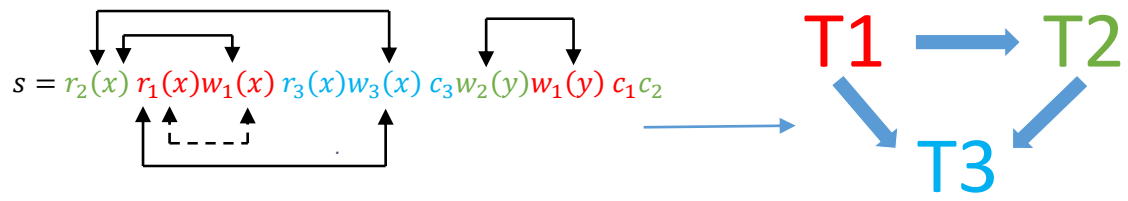


Figure 12 Serializable Schedule

[2] The schedules and related diagrams are gratefully adapted from the lecture „Databases 1“ by S. Keller, HSR.

3.2 Transactional Features

When a transaction starts, it *begins* its execution. At the end when all instructions within a transaction have been executed, it *commits*. By committing, a transaction states that it has completed its execution and that changes to shared memory shall be made visible atomically to the rest of the system. In case of a conflict, a transaction is *rolled back,* whereby all changes made so far, are discarded. The transaction subsequently restarts and makes another attempt to commit when it has completed. Transactions can also be aborted explicitly by the programmer. By aborting a transaction, all changes are discarded and the transaction does not restart. Aborting a transaction can be useful if it is already obvious that it will never be able to commit, or because the execution of the transaction is no longer desired. Figure 13 demonstrates the usage of the *Abort* feature. When withdrawing money from a bank account, a check is done to ensure that there is a sufficient balance to withdraw the amount of money. If this is not the case then the transaction aborts.

```
Transaction {
  if(this.balance < amount) {
    Abort;
  }
  this.balance -= amount;
}
```

Figure 13 Transaction abort

Transactions also support the concept of *Retry* as sketched by Figure 14. Retrying is a way to introduce wait dependencies among transactions. When a transaction requests a retrial, it is suspended until a continuation condition is satisfied. This way the transaction waits for a sufficient value of *balance* and then restarts its execution. In the scenario demonstrated by Figure 14, the transaction is expected to restart when *balance* is equal or higher than *amount*. However, when a retrying transaction is restarted, depends on the actual implementation.

```
Transaction {
  if(this.balance < amount) {
    Retry;
  }
  this.balance -= amount;
}
```

Figure 14 Retry concept

3.3 Nested Transactions

Transactions can be nested arbitrarily. This is another advantage of transactional memory over explicit synchronization because this way, composing synchronized sections can be easily achieved. To demonstrate this, a further method for the BankAccount is introduced: *Bulktransfer* sketched in Figure 15. The *Bulktransfer* method transfers money to an arbitrary amount of other BankAccounts, also as a transaction. Thereby it calls its *Transfer* method which is also implemented using transactions. *Transfer* then proceeds to call the *Deposit* method which is yet again a transaction.

```
public void BulkTransfer(int amount, BankAccount[] targets) {
  Transaction {
    foreach (BankAccount target in targets) {
      this.Transfer(amount, target)
    }
  }
}
```

Figure 15 Bulktransfer

The resulting nested hierarchy is outlined by Figure 16. It is important to note that changes are not published as long as the outermost transaction (*Bulktransfer* in this case) has not yet committed. When *Deposit* commits only the parent transaction can see the introduced changes. When *Transfer* commits *BulkTransfer* sees the changes introduced by *Deposit* and *Transfer*. However, as long as *BulkTransfer* has not yet committed, no effects are visible to the rest of the system.

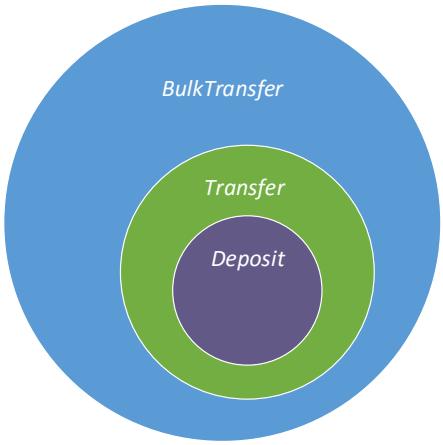


Figure 16 Nested Transactions.

Nested transactions inherit the changes introduced by parent transactions, meaning that *Deposit* is aware of all changes previously introduced by *Bulktransfer* and *Transfer* respectively. Another important aspect of nested transactions is that if a nested transaction aborts, no effects of the nested transaction are visible to the parent transaction. Consider the example outlined in Figure 17 where the *Parent* method assigns the value 5 to *X.* The *Child* method assigns the value 10 to *X* but aborts afterwards because *Y* has a value of 0 (introduced by the parent transaction). Since the nested transaction (*Child* method in this case) has not committed, the parent may not see any changes introduced, hence the value of *X* remains 5.

```
public int X { get; set; }
public int Y { get; set; }

public void Parent() {
  Transaction {
    X = 5;
    Y = 0;
    Child();
  }
}

public void Child() {
  Transaction {
    X = 10;
    if(Y == 0) {
      Abort;
    } else {
      X = X / Y
    }
  }
}
```

Figure 17 Nested Transactions with Abort

4. Programming Model

The *[STM]4.NET* runtime supports any CLR-compliant language but is specifically targeted at C#. This chapter describes the usage and application of the runtime system in C#. To do so, the bank account sample will be revised once more. First of all, an initial bank account which solely allows money withdrawal is sketched in Figure 18. As illustrated, rather than a simple *Transaction* block the usage of lambda-expressions [13] is necessary, apart from that it is very similar to the notation described in chapter 3.

```
class BankAccount {
  private int balance = 0;

  public void Withdraw(int amount) {
    Transaction.Execute(() => {
      balance -= amount;
    });
  }
}
```

Figure 18 Bank account with [STM]4.NET transactional money withdrawal

Transactions support manual retrials if desired. This can be accomplished by simply utilizing the method *Transaction.Retry()* as described in Figure 19. In this example, the client – the thread – shall wait until there is sufficient money on the given bank account. No additional code is required. The runtime will automatically ensure that no further code is executed and restarts the transaction when needed.

```
Transaction.Execute(() => {
  if(balance < amount) {
    Transaction.Retry();
  }
  balance -= amount;
}
```

Figure 19 Retrying transactions with [STM]4.NET if the balance is not sufficient

Another feature of transactions is the possibility of explicitly cancelling them during execution. To illustrate this, the bank account sample is enhanced with an additional field *locked*. If this field is set to true, no more money may be transferred from or to this account. In Figure 20, the transaction simply aborts if an account is locked. In order to inform the enclosing block about the cancellation of the transaction, the runtime simply throws an exception [14] of type *AbortException*. In order to have a meaningful result in such cases, the sample now returns a bool identifying whether the withdrawal has succeeded.

```
class BankAccount {
  private int balance = 0;
  private bool locked = false;

  public bool Withdraw(int amount) {
    try {
      Transaction.Execute(() => {
        if(locked) {
          Transaction.Abort();
        }
        if(balance < amount) {
          Transaction.Retry();
        }
```

```
        balance -= amount;
      }
    } catch(AbortException) {
       return false;
    }
    return true;
  }
}
```

Figure 20 Cancelling transactions with [STM]4.NET if the account is locked

The final and most outstanding feature of transactions is the composability, meaning that transactions may be nested into others. Of course this is also supported by *[STM]4.NET* and does not require any additional logic besides the composition itself. In the sample in Figure 21 an implementation of a *Transfer* method is shown. As seen in Figure 20, the *Withdraw* method already encloses a transaction and returns *true* if the withdrawal has succeeded. In this case, the given amount is added to the balance of the current bank account.

```
public bool Transfer(BankAccount from, int amount) {
  return Transaction.Execute(() => {
    if(!from.Withdraw(amount)) {
      return false;
    }
    balance += amount;
    return true;
  });
}
```

Figure 21 Transaction composition in [STM]4.NET

5. Implementation

This chapter describes the implementation and its architecture. Additionally, it describes how important aspects of the solution are achieved. This is embellished by a discussion and reasoning of the correctness of isolation and atomicity.

5.1 Runtime System

The runtime system handles the synchronization of shared memory among all transactions. It guarantees that transactions do not suffer from race conditions, starvation and deadlocks. By providing strict isolation in terms of serializability, it ensures that transactions observe a consistent state of the system at all time. Furthermore, it ensures that transactions appear to execute atomically, which closely relates to isolation and consistency.

[STM]4.NET is split into two assemblies: *Stm4Net.Runtime* and *Stm4Net.Htm*. This is needed as for the HTM feature, native compiled code is necessary. It is used to wrap all Intel TSX calls, provide an API for managed code and to identify if the Intel TSX features are available.

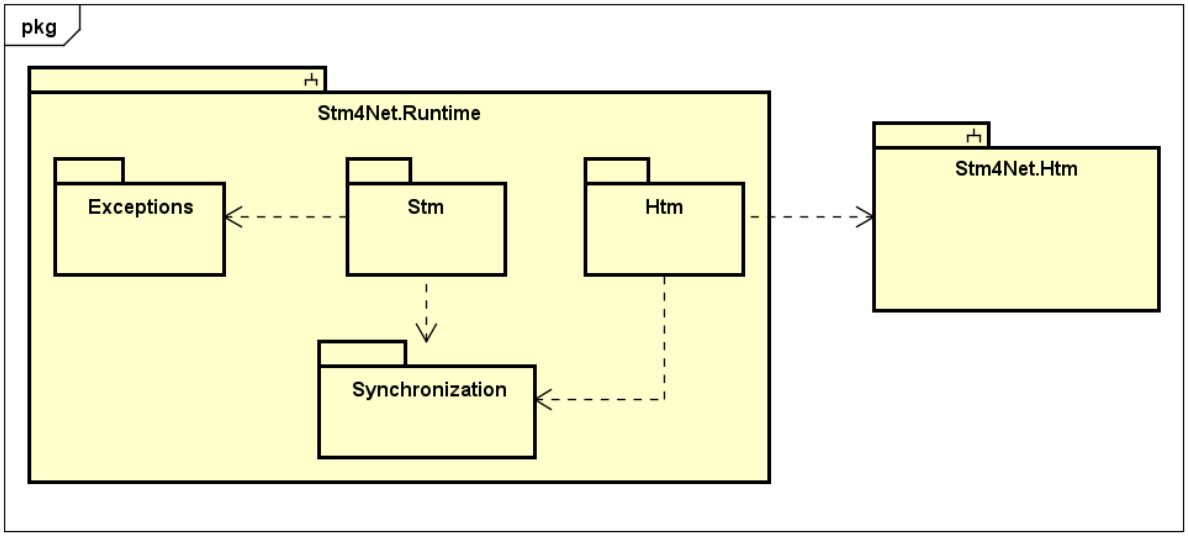


Figure 22 Package Diagram

As illustrated in Figure 22, the assembly *Stm4Net.Runtime* is internally split into multiple packages. The *Stm* package holds the logic used to execute software transactions, whereas the *Htm* package handles hardware transactions. The *Synchronization* package is used to structure all synchronization primitives. Moreover, all exceptions are located within the *Exceptions* package. The top level package *Stm4Net.Runtime* holds the API layer visible to the developers. The software transactional algorithm is based upon Maurice Herlihy and Nir Shavit's lock based TinyTM [15].

The implementation uses an optimistic concurrency control mechanism. Transactions traverse two phases: Firstly an active phase in which they run isolated from other transactions and without any synchronization (hence why it is an optimistic approach). Secondly a commit phase which re-validates the shared memory for consistency and makes the changes visible to other transactions.

Figure 23 sketches an overview of how transactions are executed. It also illustrates that hardware transactions eventually degrade if the retry conditions are not met.

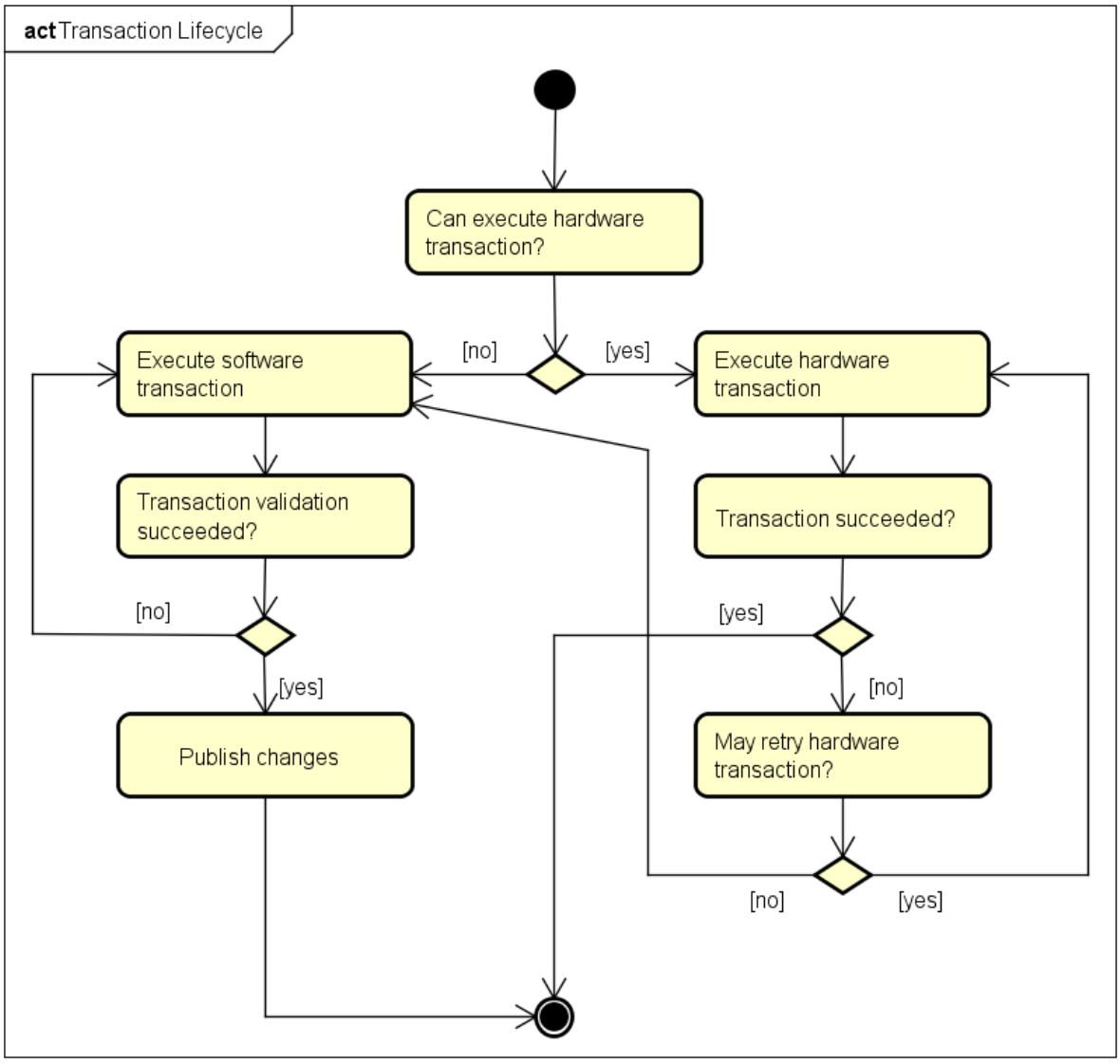


Figure 23 Transaction lifecycle overview

5.1.1 Isolation

In order to provide isolation, every access to shared objects is intercepted. Therefore a generic wrapper is provided. Shared objects must implement the wrapper type and can then be safely used from within transactions. Any shared object lacking the wrapper type is not isolated and enables race conditions.

Transactions have a transaction local registry. Upon every field access, may it be a reading or writing, this access is stored within the registry. Read accesses also ensure that the current value is consistent with the transaction itself. This is necessary due to optimistic concurrency since there is a high chance that values are not consistent with each other, leading to errors which would not happen if the code was executed sequentially. If such inconsistency is detected, an exception is thrown, leading to a retrial of the transaction. Moreover, write-accesses are only stored within the local registry and therefore are not made visible to other transactions as long as the transaction has not committed. Figure 24 outlines a transaction doing a read and a write access to a shared object and shows how this object interacts with the registry.

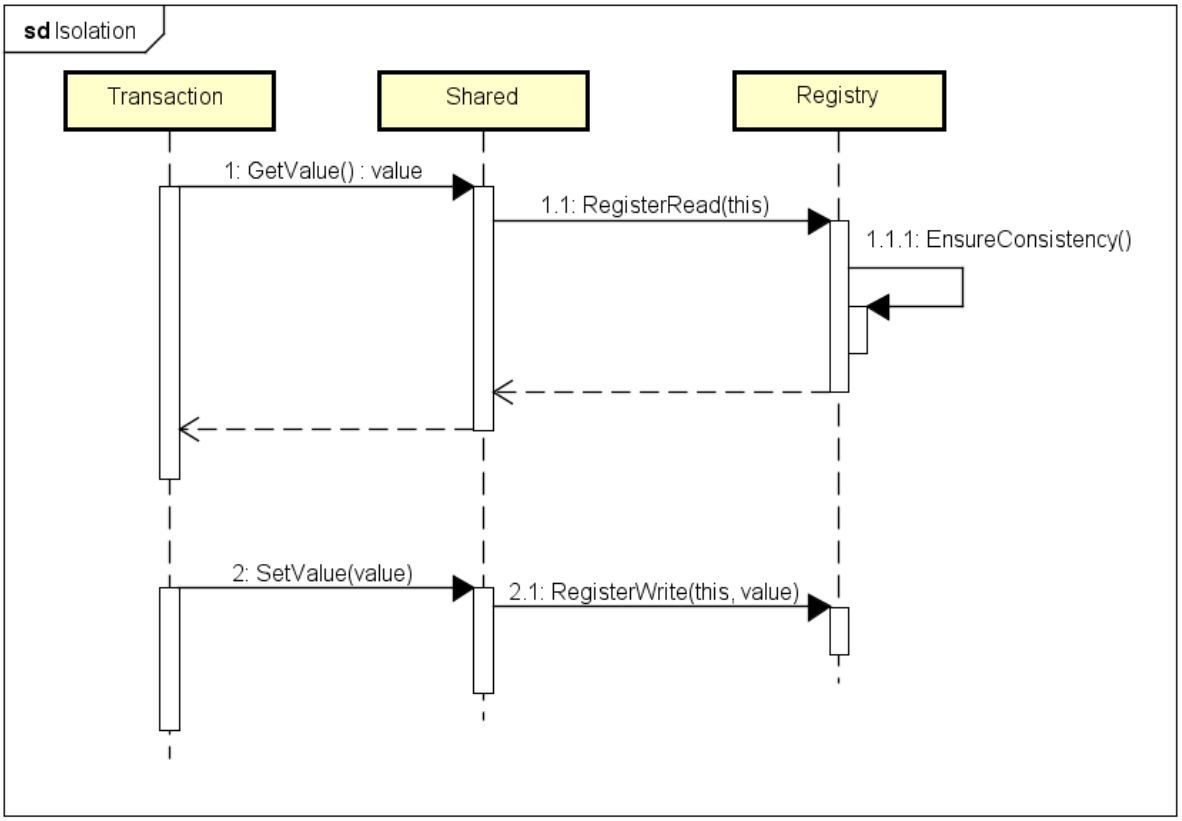


Figure 24 Read and write access to fields

Checking for value equality does not guarantee that the field has never changed during a transaction. In order to make transactions aware of concurrent changes, a version clock [16] is used. Upon start of a new transaction, the current global version – which is shared by all threads – is stored within the transaction. This means that every field having a version less or equal to this stamped version, is consistent with this transaction. This check is made upon each value retrieval. If a version is greater than the stamp of the transaction, a change has occurred concurrently – a conflict – and the transaction is restarted.

5.1.2 Atomic commit

As soon as a transaction completes, it tries to acquire all required locks to write the changes exclusively. In case a field was only read, only a read lock is necessary. By the time the locks have been acquired, the transaction re-validates the stamps of all variables within the read-set. If the stamp validation fails, a conflict is assumed and the transaction is restarted. If the validation succeeds, the global version clock is increased. Furthermore, the fields' values are updated with the values within the write-set and the fields' versions are updated to the one of the version clock.

There is a chance that transactions start right after the global clock has been increased, however the committing transaction has not yet written all changes. In this case, the new transaction may observe shared objects of an old state, currently being updated. Even worse would be reading a shared object which is being updated just at that time. To prevent such critical inconsistency, a transaction may not access fields which are part of a committing transaction's write-set.

To avoid transactions being blocked by transactions that do not have conflicts, it was decided to use per-object locking rather than one global lock. Otherwise scenarios with low contention would suffer from unnecessary synchronization overhead. These locks are acquired as soon as the transaction tries to commit.

In order to avoid deadlocks, a linear locking order is enforced. This guarantees that locks are always acquired in the same order and no cyclic dependencies occur. To each object, a unique index is assigned that is used to order the objects. To illustrate this strategy, Figure 25 shows two threads acquiring locks in a linear order. Each thread holds its own list containing its desired locks. Thread 1 is waiting to lock object B which is already acquired by thread 2. Because thread 2 does not require a lock of object A, thread 1 was already able to obtain the object's lock. As soon as thread 2 releases the lock on object C, thread 1 can obtain it and continue its progress.

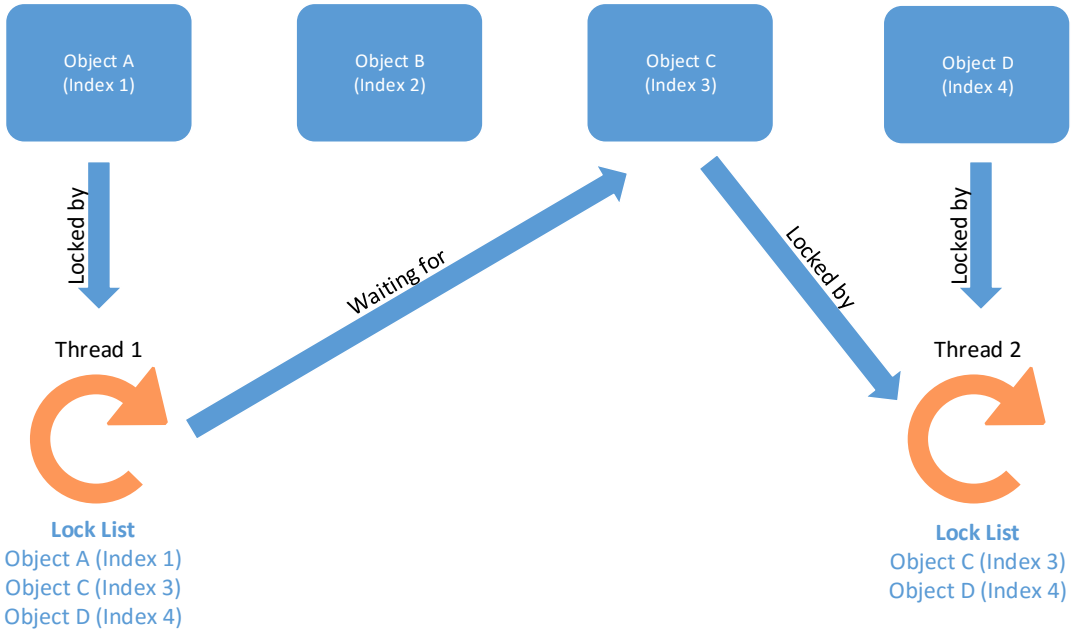


Figure 25 Linear locking order

5.1.3 Starvation avoidance

Because of the nature of optimistic concurrency, transactions may starve, hence why software transactions make use of a *Starvation Monitor* in *[STM]4.NET*. This unit counts the number of retries of each transaction. If one is exceeding a certain threshold, other transactions may not execute as long as the starving transaction has not completed.

A big drawback of this strategy is that the *Starvation Monitor* affects all transactions. This means that if one transactions is marked as starving, all other transactions have to wait. Furthermore, even the transactions which do not interfere with the starving one are blocked. A more promising is the conflict resolution directly between the conflicting transactions, using the age of the transaction [17].

There is no special handling for hardware transactions. But by limiting their maximum retries, hardware transactions eventually degrade to software transactions when retried too often.

5.1.4 Hybrid HTM and STM

[STM]4.NET supports a hybrid approach for HTM and STM with the use of Intel TSX. This requires a few things to be taken into account. Firstly the size of transactions is limited to the CPU's local cache, more specifically: The read-set and the write-set are stored within the L1 cache [18].

Secondly self-modifying code is most likely to fail with current CPUs [19]. As *[STM]4.NET* is .NET based and therefore will be JIT-compiled at runtime, transactions have a high chance of failure the first time they run under HTM. This means transactions need to run on the software layer at least once so that JIT-compilation has already taken place.

Finally it is hardly possible that the two execution models – software and hardware – will get to know of changes/conflicts caused by the other one. Therefore a mechanism is used which ensures that only one of the two execution models may run simultaneously. Figure 26 illustrates how the so-called *Transaction Monitor* works.

The waiting room represents the location software transactions stay in for as long as there are active hardware transactions. New transactions are immediately moved into this waiting room if there is at least one software transaction waiting for its execution. This fallback is used to avoid starvation of software transactions if there are always new hardware transactions starting.

If there is no software transaction either in the waiting room or being executed, the transaction is run as a hardware transaction. If the transaction fails to be executed on the hardware layer, it will be degraded to a software transaction and moved into the waiting room.

As soon as there are no more active hardware transactions, software transactions are executed. To summarize, *[STM]4.NET* ensures a strict switch between hardware and software transactions.

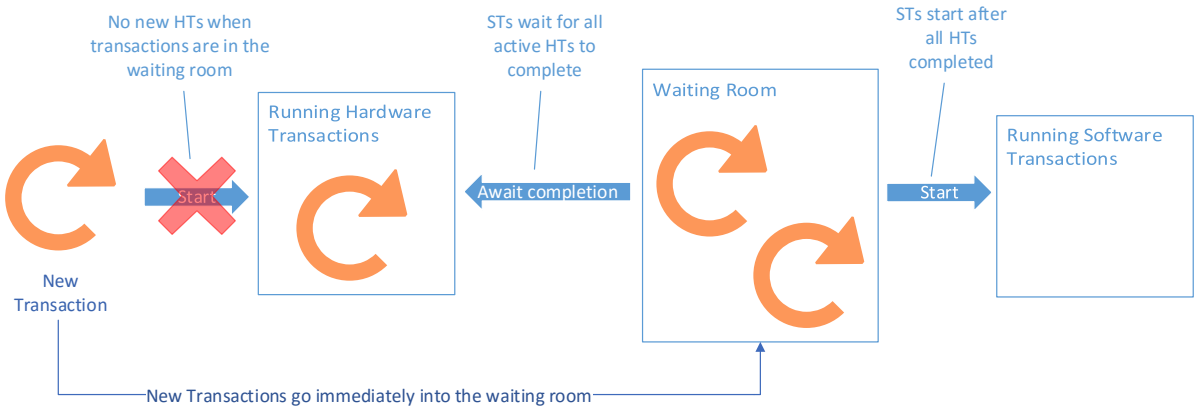


Figure 26 Transaction Monitor

During initialization *[STM]4.NET* also ensures that the hardware is actually capable of Intel TSX. This is required because executing HTM instructions on incompatible hardware will cause the application to crash.

5.1.5 Nested Transactions

As composability of transactions is a vital feature of transaction based synchronization, *[STM]4.NET* also supports nesting of transactions. But simply executing a nested transaction on the same level as its parent transaction enables dangerous consistency errors to arise. For example, a nested transaction may change a value but then decides to fail because some conditions are not met and throws an exception. Now the parent transaction happens to handle the exception raised by the nested transaction, thus the parent transaction can continue its execution. Of course, the parent transaction must not see any changes introduced by the nested transaction as this would lead to inconsistency.

Figure 27 depicts how the registry is handled if a nested transaction is being executed. In order to correctly handle nested transactions, *[STM]4.NET* clones its registry as soon as a nested transaction begins. In the case the nested transaction fails, the runtime simply discards the cloned registry and continues with the registry of the parent. If the nested transaction succeeds, the registry of the parent transaction is replaced with the nested's registry.

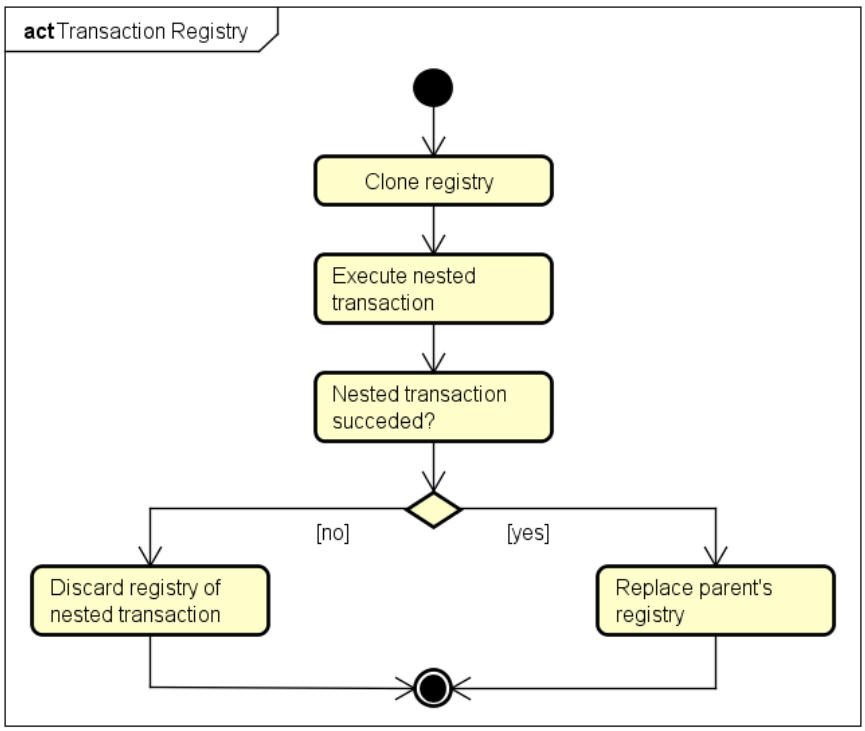


Figure 27 Registry handling of nested transactions

5.1.6 Retry

In *[STM]4.NET* there is a globally shared queue. As soon as a transaction requests a retrial, its execution is stopped and added to the waiting queue. The queue ensures that transactions are retried in a first-in-first-out manner. To stop transactions from consuming CPU time, they use a unique semaphore. As there is a slight possibility that another transaction commits at the same time as another one is requesting a retrial, the retrying transaction re-checks its consistency right after adding itself to the queue. If consistency is still given, it waits for its previously created semaphore to be released.

As soon as a transaction commits, it releases all semaphores currently in the queue. To prevent possible endless loops from occurring, the transaction will only release the amount of transactions that have been in the queue at the time it has committed. This is necessary because it may happen that one transaction releases while another transaction constantly retries.

It is noteworthy that if a transaction retries, it will always retry the outermost-transaction and not the possibly requesting nested transaction. The reason for this is, that the consistency of all nesting levels would require a check anyway. Moreover, Intel TSX does not support retrying. Therefore, hardware transaction will directly downgrade to software transactions if a retry is requested.

5.1.7 Correctness

Isolation

Active transactions store their changes in a local registry, no changes are visible to other transactions while the transaction is active, and thus the property of isolation is satisfied. The most critical part of isolation is the commit phase of transactions. To prevent access to only partially updated data, transactions may not access shared objects which are part of the write-set of committing transactions as mentioned in chapter 5.1.2.

Deadlocks

As mentioned in chapter 2, deadlocks would occur if locks are acquired in a cyclic way. Hence linear locking order is used, no cycles may occur and therefore no deadlocks.

Starvation

If a transaction retries up to a certain threshold, the *Starvation Monitor* ensures that it will eventually be run sequentially. As soon as the transaction is running sequentially, no conflicts can occur and thus the transaction can complete successfully. Furthermore, the locks used for per-object locking are guaranteed to be fair [20].

HTM and STM consistency

Because of the *Transaction Monitor*, software and hardware transactions may not run concurrently. Therefore no conflicts between these two execution models can occur.

5.2 Code Instrumentation

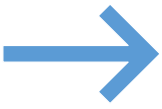
Since the programming model of *[STM]4.NET* uses wrappers to provide transactional correctness, the programmer is required to make sure that all necessary fields are wrapped. As this is error prone and cumbersome, automatic code instrumentation is provided in order to handle this tedious task. It is noteworthy that only code within the same solution [21] can be instrumented and not any second or third party libraries. Therefore, it is possible to make function calls within transactions that result in side effects which are not intercepted by the STM implementation. However, such calls are detected and reported as warnings but are not forbidden.

Fields may only have a private visibility. If the instrumentation were to change public fields all referencing types would need to be recompiled. The usage of fields with another visibility than private are detected and reported as errors, hence the instrumentation does not take place. However, it is possible to use properties with arbitrary visibility since they have private backing fields.

The .NET Framework compiles all source code into assemblies. Regardless in which programming language the assemblies were developed in, after compilation all assemblies are represented in an intermediate language called “Common Intermediate Language” (CIL) [22]. CIL is a stack-based, object-oriented, intermediate language which is compiled to native code at runtime. Figure 28 shows the body of the *Deposit* method represented in CIL. The first two instructions load the “this” pointer onto the stack (local-memory). Then the value of the field *balance* is loaded onto the stack followed by loading the argument (amount). The top two values on the stack (*balance* and *amount*) are added together removing the values from the stack and pushing the result of the addition onto it. The result is then stored back in shared memory by setting the field *balance* with the *stfld* instruction at line 6.

```
class BankAccount {
  private int balance = 0;

  public void Deposit(int amount)
    this.balance += newBalance;
  }
}
```



```
1. ldarg.0
2. ldfld int32 BankAccount::balance
3. ldarg.1
4. add
5. stfld int32 BankAccount::balance
6. ret
```

Figure 28 Common Intermediate Language Representation

During code instrumentation, assemblies are analyzed und manipulated in order to behave correctly within transactions. Subject to manipulation are fields, properties and their usages. Properties are an abstraction of fields and either a backing field is generated for them at compile time, or they are solely methods in the syntactical form of properties (getters and setters). In the first case the instrumentation must handle the backing field and in the latter no additional handling is required (in regards to properties).

Code instrumentation takes place in two steps. The first step inspects the C# source code and analyzes all transactions, and involved fields. After that, a second step manipulates the assemblies according to the analyzation results from step one.

5.2.1 Analyzation

For analyzing the C# source code the Roslyn API [23] is used. Roslyn provides a rich API to inspect and analyze source code. The algorithm takes the following steps:

- Detect transactions.
- Analyze the execution path for access to fields and properties.
- Aggregate the results to pass to the proceeding step.

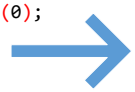
The first step analyzes C# source code rather than the CIL code in order to detect necessary manipulations. The reason therefore is that it is more convenient, since a full featured source code API already exists. Analyzing CIL code is more complex and would have resulted in too much effort dedicated to code instrumentation. However this means that, at least at the moment, code instrumentation is solely supported for C# and not for other languages supported by the .NET Framework.

5.2.2 Instrumentation

After the assembly has been compiled, assemblies are manipulated according to the analyzation results. Thereby all necessary fields – including backing fields – are wrapped. After instrumentation the *Deposit* method would appear as follows:

```
class BankAccount {
  private Shared<int> balance = new Shared<int>(0);

  public void Deposit(int amount)
    this.balance.Value += amount;
  }
}
```



```
1. ldarg.0
2. ldfld Shared`1<int32> BankAccount::balance
3. dup
4. callvirt Shared`1<int32>::get_Value()
5. ldarg.1
6. add
7. callvirt Shared`1<int32>::set_Value(!0)
8. ret
```

Figure 29 Instrumented CIL Code

The *balance* field has been enclosed with the *Shared* wrapper and the usages of the field have been updated. Since now the actual value of *balance* is encapsulated behind the *Value* property, additional calls need to be made in order to retrieve and set the value of balance (Line 4 and Line 7).

Rewriting assemblies imposes a high responsibility because errors can have a devastating effect at runtime. When adding, removing or changing CIL instructions correlating metadata also needs to be updated correctly. Offsets, for example used for jump instructions, need to be carefully updated otherwise errors occur at runtime. These errors might be very hard to relate to code instrumentation. Fortunately there is a well-accepted and popular framework for rewriting assemblies called *Mono.Cecil* [24]. The framework handles all the related tasks like updating metadata, offsets etc. efficiently and correctly.

5.2.3 Debugging

When an assembly is compiled, symbol files can also be emitted for debugging purposes. Symbol files draw the relation of CIL instructions to source code instructions. When an assembly is debugged, an IDE like visual studio can highlight the line in source code that is currently executing even though CIL instructions are being executed. Since the post-compile step takes place after the symbol files have been emitted, additional handling is required so that the symbol files and the actual source code are compliant. Not handling this issue would result in a degraded debugging experience for the developer

which is often not acceptable. Fortunately *Mono.Cecil* has integrated support for emitting updated symbol files along with the instrumented assembly.

6. Evaluation

To evaluate the performance and efficiency of the implementation, a detailed comparison with existing software transactional memory is done. The existing solutions are compared by performance, feature-set, and correctness properties.

To achieve meaningful results, all tests are executed on the computers which have the setups as listed in Table 1.

Component	HTM Setup	Many Cores Setup
Operating System	Windows 7 64 Bit	Windows 7 64 Bit
CPU	Intel Xeon CPU E3-1245 v3 @ 3.40 GHz, Haswell, 4 Cores	2x Intel Xeon CPU E5-2650 v3 @ 2.30 GHz, Sandy Bridge, 8 Cores
RAM	16 GB	32 GB
.NET Framework	4.6.2	4.6.2
Java JRE Version	1.8.0_65 (x64)	1.8.0_65 (x64)

Table 1 Evaluation Setup

For the purpose of evaluation different benchmarks are run, each simulating a specific scenario. The computers utilized were provided by HSR with the drawback of not having full control over the running software – antivirus, updates and so forth. Moreover, the benchmarks are run solely on the 64 bit virtual machines.

6.1 Frameworks

This section describes other publicly available software transactional memory frameworks. It lists the available features of each framework and also makes a short description of their usage.

6.1.1 Shielded

Shielded is an implementation of software transactional memory for .NET. Similar to other software transactional memory frameworks it achieves concurrency control by wrapping variables. It provides the most common features like nested transactions, retry mechanism. Shielded does not handle isolation for nested transactions correctly, therefore it does not fulfill all correctness properties. Table 2 gives an overview of the feature-set provided by *Shielded.*

Feature	Supported
Nested Transactions	Yes, but with incorrect isolation
Retry / Rollback	No
Abort	Yes (via Exceptions)
HTM Support	No
Transactional Collections	Yes
Maintained	Yes (last updated November 2015)

Table 2 Shielded Features

Shielded also provides a few more, seemingly interesting features, such as read-only transactions, commutes and automatic wrapping at runtime.

The only observed correctness issue is that nested transactions are not correctly isolated from their parents. Simply put, if a sub transactions updates a shared value and fails or aborts afterwards, its outer transaction still sees the values from its sub transaction, thus an inconsistent state.

Usability

Shielded provides a simple API for managing transactions. Transactions are defined using lambdas and shared objects are wrapped with a *Shielded* wrapper.

```
Shielded<int> count = new Shielded<int>(0);
Shield.InTransaction(() => ++count.Value);
```
Figure 30 Shielded usage example

A more sophisticated way to avoid wrapping of variables is provided by a factory that generates proxy classes for usage within transactions at runtime. A drawback of the factory is that it requires all properties to be virtual. Also the getter and setter must be declared using the same visibility.

```
class SomeClass {
  public virtual int Count { get; set; }
}

var instance = Factory.NewShielded<SomeClass>();
Shield.InTransaction(() => ++instance.Count);
```
Figure 31 Shielded factory example

Shielded is available as NuGet package and has no further dependencies. Hence integration into an application is simple and straightforward. Documentation is limited to a single readme file demonstrating the usage of the framework.

6.1.2 STMNet

STMNet is another software transactional memory implementation for the .NET framework. It lacks most of the required correctness properties making a lot of concurrency issues still possible to arise. It tends to deadlock with nested transactions, allows write-skews among running transactions and cannot handle exceptions within transactions in a decent way.

Feature	Supported
Nested Transactions	No
Retry / Rollback	No
Abort	Depends on usage
HTM Support	No
Transactional Collections	No
Maintained	No (last updated July 2013)

Table 3 STMNet Features

Due to the many shortcomings of this framework it is not taken into account for further evaluation purposes, since it cannot even execute all benchmarks correctly.

Usability

STMNet provides two ways to define transactions. Figure 32 demonstrates the simpler approach introducing several hazards that might lead to undesired commits. Committing is performed when the *dispose* method is invoked at the end of the using block. This means that if an exception occurred and the transaction is exited unexpectedly, committing is still performed.

```
StmObject<int> count = Stm.CreateObject(0);
Stm.ExecuteTransaction(transaction => {
  using(transaction) {
    _count.Write(_count.Read() + 1);
  }
}, transaction => true);
```
Figure 32 STMNet usage example

An alternative way to define transactions is by committing explicitly and avoiding using blocks which is outlined in Figure 33. However this moves the responsibility of committing to the developer.

```
StmObject<int> count = Stm.CreateObject(0);
Stm.ExecuteTransaction(transaction => {
  count.Write(count.Read() + 1);
  Transaction.Commit();
}, transaction => true);
```

Figure 33 STMNet explicit commit

Another possibility is to handle the state explicitly as well which is demonstrated in Figure 34. This way one would need to manually loop until the transaction succeeds (not shown in illustration).

```
StmObject<int> count = Stm.CreateObject(0);
Transaction transaction = Stm.ExecuteTransaction(transaction => {
  count.Write(count.Read() + 1);
  Transaction.Commit();
});

if(transaction.State == TransactionState.Committed) {
  // Transaction has committed successfully.
}

if(transaction.State == TransactionState.Aborted) {
  // Transaction has encountered a conflict and was aborted.
}
```

Figure 34 STMNet explicit state handling

The examples conclude that there is no uniform way to manage transactions within *STMNet*. Moreover each approach has several drawbacks and hazards one needs to be aware of. *STMNet* consists of a single library which must be built manually. No further documentation exists other than is provided in a small readme file.

6.1.3 Scala STM

Scala STM is currently one of the most popular and mature transactional memory implementations. It provides high performance, strict correctness properties and reasonable language integration, even though it also uses wrappers for variables. In contrast to the aforementioned frameworks, *Scala STM* runs on the Java Virtual Machine and not the .NET Common Language Runtime. However it is ideal as a comparison participant because it is most probably the best implementation of software transactional memory for a mainstream programming environment at the moment.

Feature	Supported
Nested Transactions	Yes
Retry / Rollback	Yes
Abort	Yes (via Exceptions)
HTM Support	No
Transactional Collections	Yes
Maintained	Yes (last updated April 2014)

Table 4 ScalaSTM Features

Regarding correctness, *Scala STM* achieves best results in every tested region. It neither suffers from starvation nor does it violate isolation in any observable way.

Usability

Scala STM requires wrapping variables which are created by factory methods. Transactions are defined by using lambdas as Figure 35 illustrates.

```
Ref.View<Integer> count = STM.newRef(0);
STM.atomic(() -> counter.set(counter.get() + 1));
```

Figure 35 ScalaSTM usage example

Keep in mind that Java does not provide properties as known in .NET. Hence wrapping variables is a bit more cumbersome.

6.2 Benchmarks

Each benchmark runs a total of 100 million transactions which are equally distributed amongst all available threads, meaning that two threads will run 50 million transactions concurrently. Since *[STM]4.NET* has the ability to run hardware transactions, there are two different axes. *[STM]4.NET STM* are results of transactions executed solely on the software layer. Benchmarks run with the hybrid approach are labeled with *[STM]4.NET HTM*. Furthermore each benchmark is run three times whereas the runs with the best results are shown on the graphs. *Shielded* is not shown in most graphs as it was out-performed to such an extent that it made it hard to read the other frameworks' performance on the graph.

6.2.1 No Contention

This benchmarks shows how well an implementation scales with concurrently running transactions which are completely autonomous and thus never cause conflicts. Each transaction executes the code illustrated in Figure 37 whereas the shared variables are unique per thread. Each transaction calculates the sum of adding the value of its shared variable to itself. Afterwards it stores the resulting sum in this shared variable. It is noteworthy that this is not a realistic example since in such cases no synchronization would be necessary at all.

```
Transaction {
  var sum = shared + shared;
  shared = sum;
}
```

Figure 36 Transaction with low contention

As Figure 37 denotes *[STM]4.NET* scales well with increasing threads. Furthermore, it is clearly visible that the HTM feature significantly reduces the execution time. On the other hand the HTM feature loses its strength with more threads. This happens because the more transactions are running concurrently, the higher the chance that a hardware transaction fails, requiring the complete system to downgrade to software transactions. Furthermore the chance that *[STM]4.NET* may re-upgrade to hardware transactions is minimized with more threads and thus more active transactions.

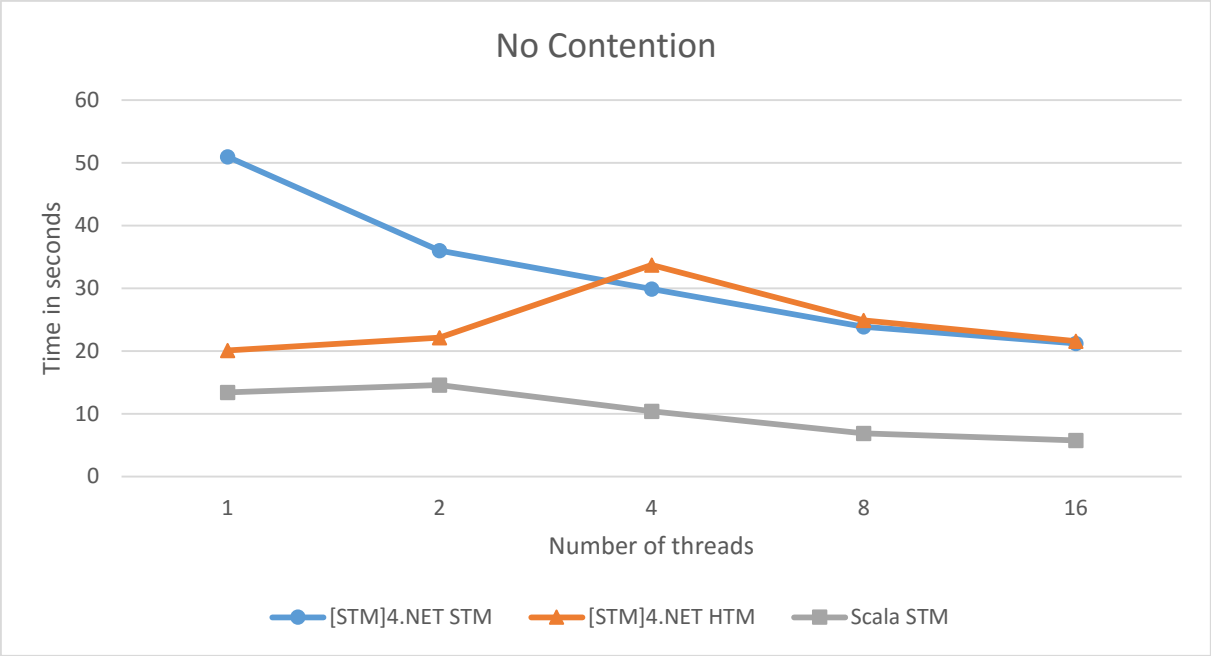


Figure 37 No contention results: Best out of three runs, 100M transactions, HTM Setup

Even though *[STM]4.NET* makes use of HTM, *Scala STM* is still up to 250% faster. Running the single threaded benchmark on a pure HTM based implementation resulted in an execution time of about eight seconds. This means that the hybrid approach is about 2.5 times slower than the purely HTM based and that the synchronization overhead of the hybrid approach is still quite significant. Further analyzations have shown that about 0.03 percent of all hardware transactions failed and had to degrade to a software transaction.

6.2.2 Low Contention

This benchmark causes conflicts from time to time, meaning there is a chance that a transaction succeeds without any conflicts whereas others do not. This was achieved by creating an array of values which are shared amongst all active transactions as sketched in Figure 36. Each transaction accesses a certain index for a read and write operation depending on the iteration counter of the executing thread. The offset is used to regulate the chance of conflicts. If the offset is for example 2, then conflicts are very likely to appear. With an increasing offset the likelihood of conflicts decreases. In this benchmark an offset of 15 was chosen resulting in conflicts for one in ten transactions.

```
Transaction {
  var index = (threadIteration * offset) % nofSharedObjects;
  shared[index] = shared[index];
}
```

Figure 38 Transaction with low contention

Figure 39 illustrates the results obtained from the low contention benchmark. *Scala STM* achieves the best performance which is due to its optimistic synchronization strategy which excels in such low contention scenarios. With increasing number of threads the advantage of hardware transactions vanishes and eventually reaches the same performance as software transactions.

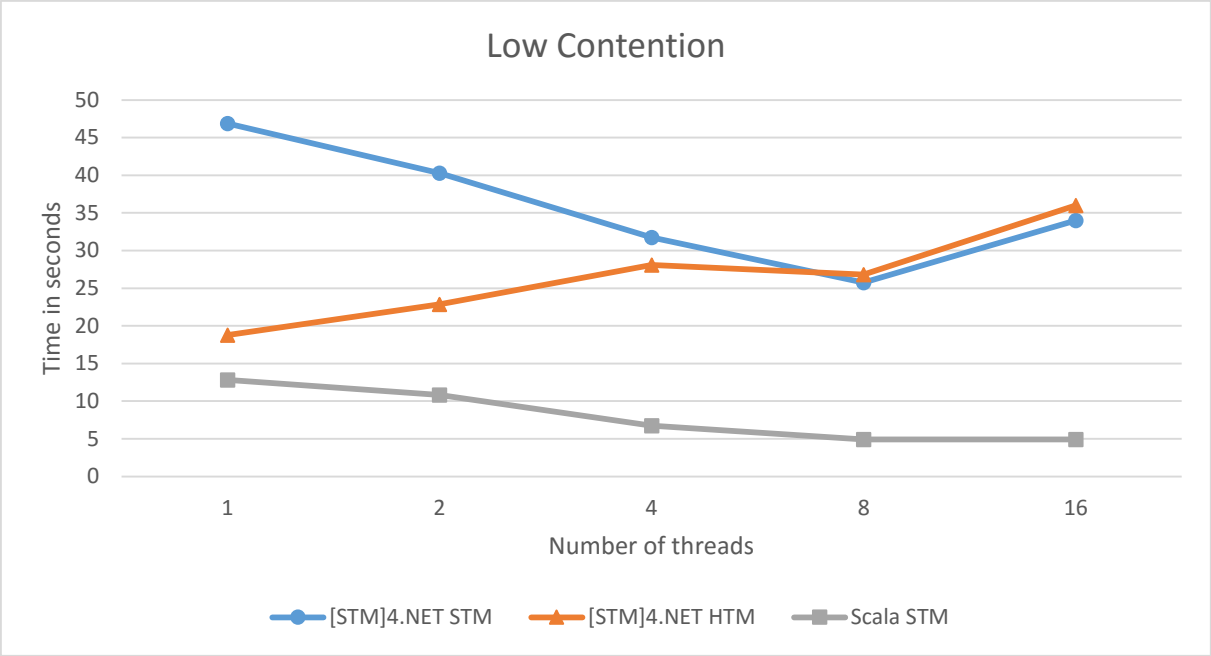


Figure 39 Low contention results: Best out of three runs, 100M transactions, HTM Setup

An interesting finding is, that *[STM]4.NET* suddenly drops in performance on 16 threads compared to eight. This happens due to unfortunate scheduling because the thread count exceeds the processor core count. Threads which have acquired a lock might be paused in favor of another threads execution, but this thread has to wait for the lock to be released.

6.2.3 High Contention

The high contention benchmark makes use of a global counter variable which is incremented by each thread concurrently. This means that each transaction reads the actual value, increments it by one and stores the new value. The transactional code is sketched in Figure 40.

```
Transaction {
  count++;
}
```

Figure 40 Transaction with high contention

Because this benchmark causes frequent conflicts, it was decided to use the *Many Cores Setup* to have more threads executing in parallel and therefore increase the chance of conflicts.

Figure 41 shows that *[STM]4.NET* scales better than *Scala STM* in high contention scenarios, or in other words, with a high chance of conflicts. With 4 threads, *[STM]4.NET* is only 14% slower than *Scala STM* and with eight threads it is already 15% faster.

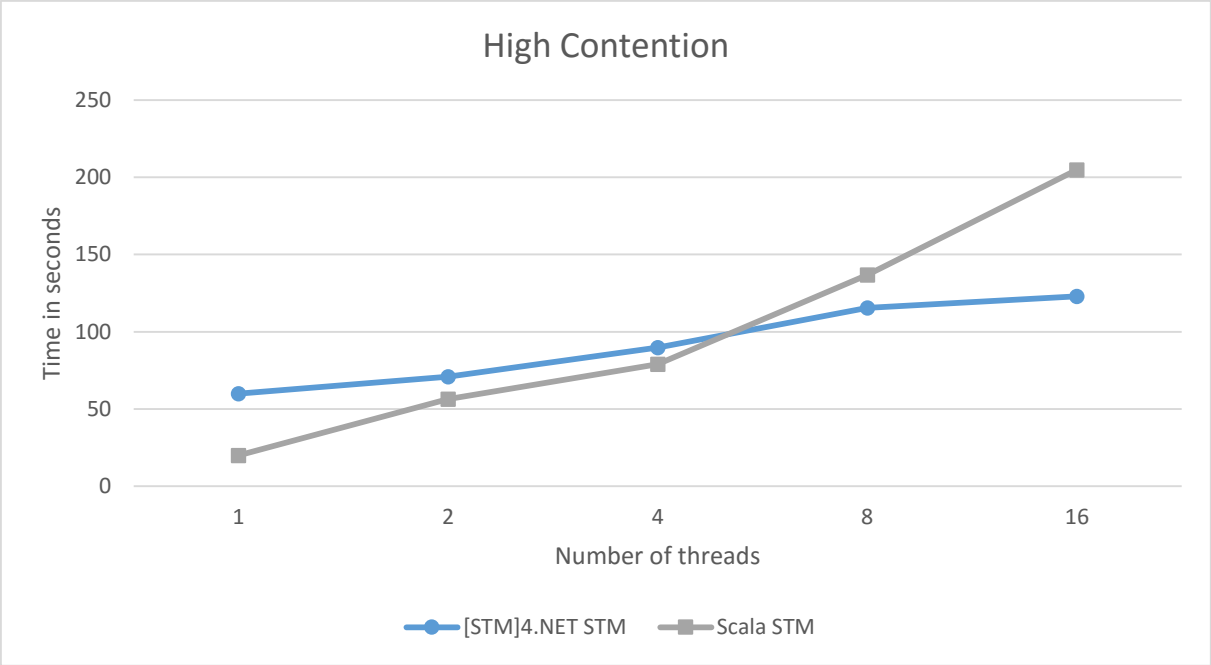


Figure 41 High contention results: Best out of three runs, 100M transactions, Many Cores Setup

6.2.4 Large Transactions

This benchmark evaluates the behavior of transactions performing time-intensive work. It simulates the bulk processing of booking records from two banks at the end of a working day (end of day processing). There is a set of 1M records which contain information about a transfer between two accounts. The threads constantly fetch a record and process it until there are no records left. Figure 42 sketches the transactional code. The primary synchronization points are the two banks to which the accounts belong. For each transfer, a bank's global balance (the sum of all balances of accounts belonging to the bank) is updated.

```
loop(untilNoRecordsLeft) {
  Transaction {
    record = GetNextRecord();
    record.Source.Transfer(record.Amount, record.Target);
  }
}
```

Figure 42 Long running transaction

Differing from the other benchmarks, this results only in 1M effective transactions. The reason for this is, that with a higher number the garbage collection mechanism of the virtual machines becomes a significant bottleneck. With 100M, the Java VM abruptly crashed, while the .NET CLR hardly made any progress.

This benchmark is comparable to the high contention benchmark but with significantly more work and nested transactions. The first observation in Figure 43 is, that *Scala STM* now achieves better results than *[STM]4.NET*. There is a fairly simple reason for this outcome: The isolation of nested transactions is an expensive task for *[STM]4.NET* since the registry is cloned with each nesting level. In contrast, *Scala STM* simply flattens transactions for as long as there is no strict abort requested or no exception occurs [25], which does not happen in this benchmark.

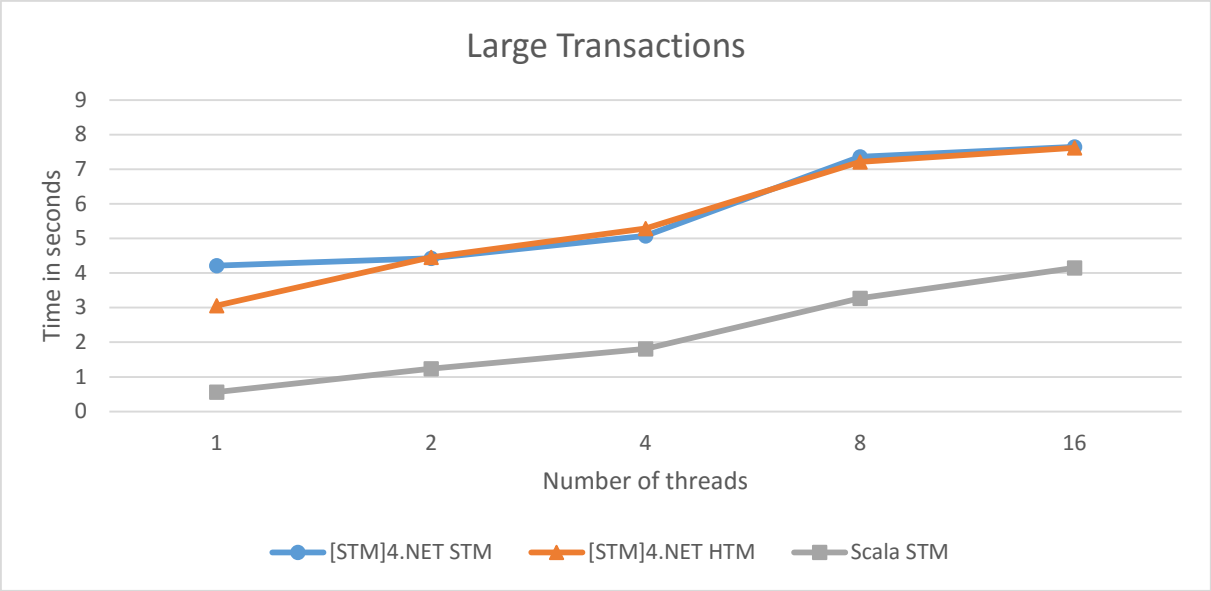


Figure 43 Large transactions results: Best out of three runs, 1M transactions, Many Cores Setup

Because of *Shielded*'s varying execution times (differences of up to factor 20), it was not included in this benchmark since such results are not considered representative.

6.2.5 Read Only

With this benchmark, all transactions share a single variable named *shared* which they read from as shown in Figure 44. Same as the low contention benchmark, this constellation would not require any synchronization at all.

```
Transaction {
  var result = shared;
}
```

Figure 44 Read Only Transaction

This benchmark shows that *Shielded* has a special handling for read-only threads because it is the only case it can compete with *Scala STM* and *[STM]4.NET*. As denoted in Figure 45, *Shielded* is nearly 40% faster than *[STM]4.NET*. Interesting is, is that both – *[STM]4.NET* and *Shielded* – have a slight slowdown with two and four threads compared to one thread. This happens because both implementations use a read-only lock in this case, thus requiring little synchronization.

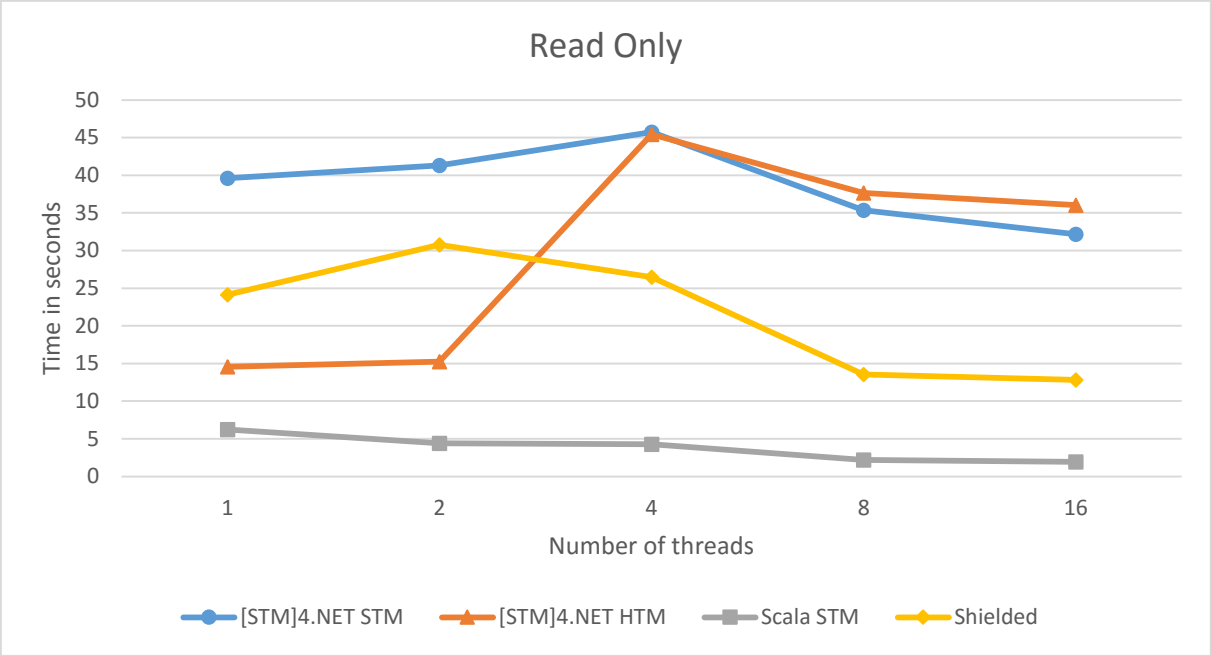


Figure 45 Read only results: Best out of three runs, 100M transactions, HTM Setup

The HTM feature gives *[STM]4.NET* a significant boost by reducing the required execution time by more than 60% with one and two threads. Starting with four threads, transactions are mostly run on the software layer since the chance that a software transaction is already running is significantly higher, thus preventing a switch to hardware transactions. This results in the significant drop of performance for the hybrid implementation. With eight and more threads, the hybrid approach seems to become a slight bottleneck. This happens because there is some chance that transactions may run on the hardware layer from time to time. But this blocks out all other threads waiting to execute software transactions, even if it would not be necessary for read-only transactions. Nevertheless, this synchronization is still necessary, because it is not known beforehand what the transactions will do.

Scala STM shows once more its strengths: an obstruction-free algorithm superior to all other implementations.

6.2.6 Read Only with varying object count

In order to test how the frameworks scale with the number objects, this benchmark is executed with a total of four threads with each execution. But instead of changing the number of threads, the number of objects read is increased. Apart from that, this benchmark does the same as the read only benchmark described in chapter 6.2.5.

As illustrated in Figure 46, *[STM]4.NET* does not scale well with increasing object counts. The reason for this behavior is the very expensive linear locking order, or to be more specific, the used data structure is considerably time-consuming.

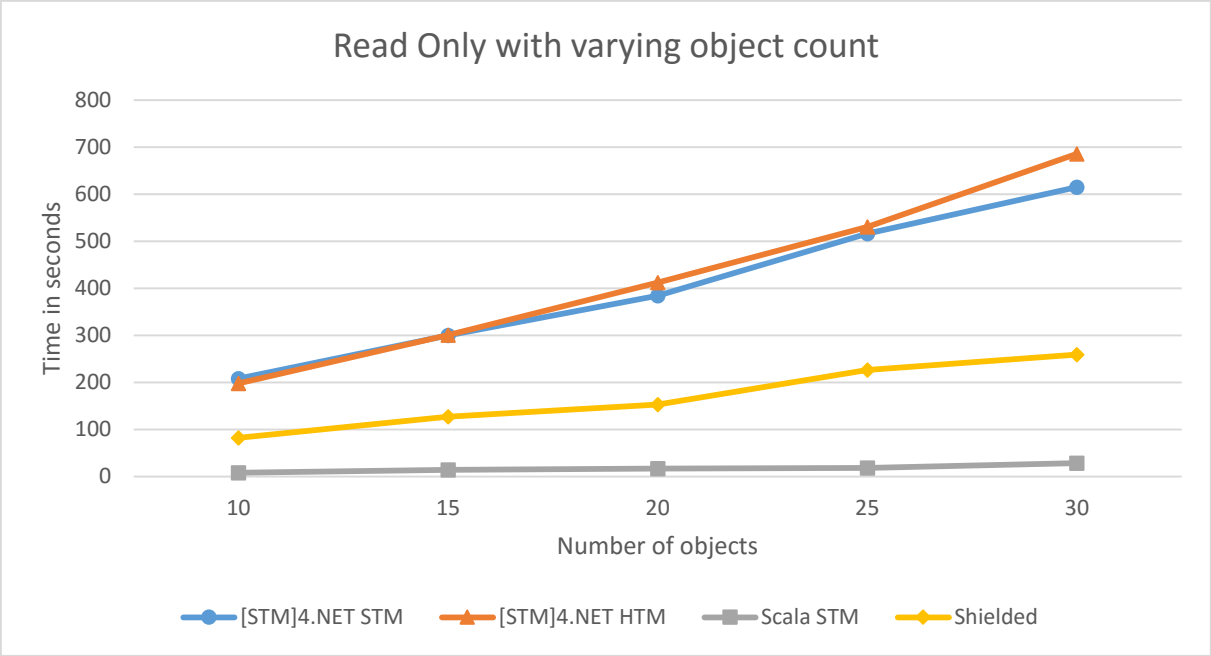


Figure 46 Read only with varying object count results: Best out of three runs, 100M transactions, HTM Setup

6.3 Summary

While not being able to reach *Scala STM*'s outstanding performance, *[STM]4.NET* is without doubt the most straightforward transactional memory framework to date in terms of usability. This is due to its innovative post-compilation code instrumentation mechanism to get rid of manual field wrapping. Furthermore it out-performs its competitors on the .NET platform in most cases and excels with its strong isolation. While the Intel TSX integration is without doubt a great enhancement, hardware transactions tend to fail frequently on the test setup. This leads to a rather quick degrade to software only transactions, especially on higher thread and concurrent transaction count.

7. Conclusion

[STM]4.NET achieves a high level of usability in the area of transactional memory. The runtime ensures correct isolation so the system may never enter an inconsistent state. Moreover, a significant performance boost is achieved by supporting hardware transactions using Intel TSX. It is expected that with future CPU generations the support for hardware transactions will improve, thus speeding up applications using *[STM]4.NET* without requiring a re-compilation.

Regarding efficiency, the solution is still improvable. The main reason for this, is the expensive ordering of locks required to avoid deadlocks. In terms of overall performance, *[STM]4.NET* may achieve best results on the .NET platform but is still no competitor for *Scala STM*. Also it lacks transaction safe collection types, which decreases the amount of possible applications.

In order to integrate the runtime into an application, a developer simply needs to reference the library and enclose transactional sections with the *Transaction* statement. Furthermore, with the assistance of the Visual Studio plugin, basic lock-statements can automatically be refactored to transactions. Because runtime does not stand out in its performance, it is obviously not the best choice if applications are performance critical.

There is still room for improvement. For example the linear locking order is the most significant bottleneck. Either by completely removing this mechanism or by improving the ordering. Also the synchronization of hardware and software transactions may be improved. Currently it is very strict by only allowing one of these two execution models to run at a time. This may be improved with a logic which is less strict by analyzing transactions beforehand. Additionally, the Read-Write Lock implementation may be replaced with a more specialized one, because there is no reason to wait for a lock if the value is already inconsistent. Moreover, the instrumentation can be improved so it does no longer require C# source code and rather directly analyzes the compiled CIL code. This way, full instrumentation support could be achieved for all .NET languages. Also replacing collections with their transactional equivalent would increase the benefit of the instrumentation process.

8. Glossary

ACID
An acronym for the properties: Atomic, Isolation, Consistency and Durability.

Atomic
Atomic is a property that something should happen only once or not at all. Furthermore atomic requires that whatever is done, is accomplished in a single step.

C#
C# is the most popular programming language for the .NET Framework.

Concurrency
Concurrency is the term used when parallel running threads access shared memory simultaneously thus requiring synchronization.

Consistency
Consistency is a property requiring that an instruction takes the system from one valid state to another.

Data Race
Data Race is a synonym for race condition describe above.

Deadlock
Deadlock is the state when two threads have mutually excluded each other so that none of them can proceed with computational work anymore.

Garbage Collection
Garbage Collection is a mechanism provided by virtual execution machines that removes objects in memory which are not used anymore.

HTM
Hardware Transactional Memory is the concept of executing transactions directly on the hardware layer.

Isolation
Isolation is a property requiring that threads are isolated from each other effects and their execution schedule is serializable.

JIT Compilation
Just in Time Compilation (JIT) is the process of translating an intermediate language to native code at runtime.

.NET Framework
The .NET Framework is a popular programming Framework developed and maintained by Microsoft.

Race Condition
Race condition is any effect, error or execution path caused by the concurrent execution of threads.

Rollback
When a transaction performs a rollback, all changes are discarded and the transaction restarts from the beginning.

RTM
Restricted Transactional Memory is the memory model introduced by Intel to describe their Hardware Transactional Memory implementation.

Starvation

Starvation is the state where a thread cannot make progress because other threads keep interfering and overtaking it.

STM

Software Transactional Memory is the concept of executing transactions on the software layer.

TSX

Transactional Synchronization Extensions is Intel's superset for hardware based synchronization.

9. Contents

9.1 Figures

9.2 Tables

10. References

[1] M. Herlihy and N. Shavit, "What is Wrong with Locking?," in *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2012, pp. 417-418.

[2] Microsoft, "Overview of the .NET Framework," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/zw4w595w(v=vs.110).aspx. [Accessed 08 12 2015].

[3] Microsoft, "Common Language Runtime (CLR)," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/8bs2ecf4(v=vs.110).aspx. [Accessed 08 12 2015].

[4] L. Bläser, "Race Condition," in *04_Gefahren_der_Nebenläufigkeit*, 2015, pp. 7-12.

[5] E. C. Coffmann, M. J. Elphick and A. Shoshani, "System Deadlocks," 1971.

[6] M. Herlihy and N. Shavit, "Mutual Exclusion," in *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2012, p. 24.

[7] Microsoft, "lock Statement (C# Reference)," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/c5kehkcz.aspx. [Accessed 8 12 2015].

[8] M. Herlihy and N. Shavit, "Properties of Mutual Exclusion," in *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2012, pp. 8-9.

[9] R. Holt, "Some deadlock properties of computer systems," Department of Computer Science - University of Toronto, Toronto, Ontario, Canada, 1971.

[10] M. Herlihy and E. Moss, "Transactional Memory: Architectural Support for Lock-Free Datastructures," 1993.

[11] T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," 1983.

[12] A. Fekete, N. Lynch and W. Weihl, "A Serilization Graph Construction for Nested Transactions," MIT Laboratory for Computer Science, Cambrige, MA, 1990.

[13] Microsoft, "Lambda Expressions (C# Programming Guide)," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/bb397687.aspx. [Accessed 12 09 2015].

[14] Microsoft, "Exceptions and Exception Handling (C# Programming Guide)," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/ms173160.aspx. [Accessed 9 12 2015].

[15] M. Herlhy and N. Shavit, "A Lock-Based Atomic Object," in *The Art of Multiprocessor Programming*, Waltham, Elsevier, Inc., 2012, pp. 438-445.

[16] M. Herlihy and N. Shavit, "Why It Works," in *The Art of Multiprocessor Programming*, Waltham, Elsevier, 2012, pp. 440-441.

[17] M. Herlihy and N. Shavit, "Contention Managers," in *The Art of Multiprocessor Programming*, Waltham, Elsevier, Inc., 2012, pp. 431-433.

[18] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual," 09 2015. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf. [Accessed 05 10 2015].

[19] Intel, "Intel® Transactional Synchronization Extensions (Intel® TSX) Programming Considerations," Intel, [Online]. Available: https://software.intel.com/en-us/node/582935. [Accessed 26 10 2015].

[20] Microsoft, "ReaderWriterLockSlim Class," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/system.threading.readerwriterlockslim(v=vs.110).aspx. [Accessed 01 12 2015].

[21] Microsoft, "Solutions and Projects in Visual Studio," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/b142f8e7.aspx. [Accessed 9 12 2015].

[22] ECMA, "Common Language Infrastructure (CLI) Partition I to VI," 2012. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf. [Accessed 30 11 2015].

[23] Microsoft, "dotnet / roslyn," [Online]. Available: https://github.com/dotnet/roslyn. [Accessed 30 11 2015].

[24] J. Evain, "Mono.Cecil," [Online]. Available: http://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/. [Accessed 30 11 2015].

[25] Scala STM, "Exceptions," Scala STM, [Online]. Available: https://nbronson.github.io/scala-stm/exceptions.html. [Accessed 10 12 2015].

[26] Intel, "Intrinsics for Restricted Transactional Memory Operations," Intel, [Online]. Available: https://software.intel.com/en-us/node/514074. [Accessed 23 11 2015].

[27] Microsoft, "__cpuid, __cpuidex," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/hskdteyh.aspx. [Accessed 18 09 2015].

[28] M. Herlihy and N. Shavit, "Transactions and Atomicity," in *The art of Multiprocessor Programming*, Waltham, Elsevier, Inc., 2012, p. 423.

[29] M. Herlihy and N. Shavit, "Transactional Memory," in *The Art of Multiprocessor Programming*, Elsevier, Inc., 2012, p. 447.

[30] R. Dementiev, "Making the Most of Intel Transactional Synchronisation Extensions," Software and Services Group, Intel, 2014.

[31] G. Weikum and G. Vossen, "Transactional Information Systems," in *Theory, Algorithms, and the Practice of Concurrency Control and Recovery*, 2002.

[32] M. L. Scott, "Best-effort Hybrid TM," in *Shared-Memory Synchronization*, Morgan & Claypool, p. 165.

[33] M. Herlihy and N. Shavit, "Monitor Locks and Conditions," in *The Art of Mutliprocessor Programming*, Morgan Kaufmann, 2012, pp. 178-193.

[34] P. Bernstein and E. Newcomer, "Locking," in *Principles of Transaction Processing*, Morgan Kaufmann, 2009, p. 184.

[35] M. Scott, "Safety and Liveness," in *Shared-Memory Synchronization*, Morgan & Claypool Publishers, 2013, pp. 7-8.

11. Appendix

11.1 HTM

With Hardware Transactional Memory, the transaction logic is provided by the CPU. That means, the changes to the software will be minimized as there is no need to manually track memory changes. On Intel CPUs, the HTM feature is called RTM (Restricted Transactional Memory) and is a part of Intel TSX. The usage is described in the Intel TSX documentation [26]. Figure 47 sketches a simple usage example of RTM.

```
#include <immintrin.h>
// ...
void doConcurrent() {
  if(_xbegin() == _XBEGIN_STARTED) {
    // Transaction was started
    // do something...
    _xend();
    // Transaction successfully completed and committed
  } else {
    // An error occurred during the transaction. Either retry
    // or fallback to an alternative, e.g. mutex
  }
}
```

Figure 47 Using RTM for transactions

11.1.1 Ensuring CPU capability

Because these instructions are new, an application has to check if the CPUs supports the required instruction-set. This can be accomplished using `void __cpuid(int cpuInfo[4], int function_id)` (or `__cpuidex`) [27]. RTM is supported if the 11th bit of `cpuInfo[1]` (EBX register) of function `0x00000007` is set.

Figure 48 shows an example how to check if an Intel CPU is capable of RTM in C. As this does not apply to all vendors, it should also be checked if the software is running on an Intel CPU.

```
bool isTsxCapable() {
  int cpuInfo[4];
  __cpuidex(cpuInfo, 0x00000007, 0);
  return cpuInfo[1] & (1 << 11);
}
```

Figure 48 Checking if the CPU is capable of RTM

11.1.2 Emulation of RTM

As RTM / Intel TSX is rather new, it is hard to find capable hardware. Therefore Intel provides an emulation toolkit called “Intel Software Development Emulator”. This emulator supports the emulation of new CPU features (such as RTM).

To run an application making use of RTM on a non-capable CPU using SDE, one must execute the application with the following command like:

```
sde.exe –rtm-mode full -- <application>
```

Figure 49 Emulating RTM with SDE

The `–rtm-mode` argument can be changed to different values to emulate for example different errors during transactions to ensure the alternative, non-transactional code paths are implemented properly.

11.2 Isolation Issues

Transactions run concurrently and under optimistic concurrency control. Therefore a transaction can encounter a concurrency conflict at any time. At the very moment where a conflict occurs the transaction will not be able to commit anymore. Depending on the type of conflict and implementation, the transaction will still keep on executing. This chapter describes the different problems, which arise due to such behavior, presented as concrete examples.

11.2.1 Write Skew

Even though snapshot isolation prevents many concurrency issues from arising it does not prevent the appearance of write skews. In a write skew two concurrently running transactions each read overlapping values e.g. `shared1` and `shared2`. If the first transaction solely writes `shared1` and the second one only writes to `shared2`, this conflict is not detected by using snapshot isolation because it only checks the integrity of written values and not the ones that have been read only. This might not be a problem but in many cases the first transaction will update `shared1` according to what it has read to be `shared2` and vice versa – hence resulting in a race condition. Figure 50 demonstrates such a malicious write skew.

```
// Thread 1
Transaction {
  value = shared2;
  if(value == 1) {
    shared1 = 1;
  } else {
    shared1 = 2;
  }
}
```

```
// Thread 2
Transaction {
  value = shared1;
  if(value == 1) {
    shared2 = 1;
  } else {
    shared2 = 2;
  }
}
```

Figure 50 Write skews problem

In this example, Thread 1 updates `shared1` according to what it has read to be `shared2` although the value of `shared2` might already be another one at the time Thread 1 commits. As mentioned above, simple snapshot isolation cannot prevent these issues from arising. Therefore a more robust strategy must be applied to fully guarantee that the system is serializable.

11.2.2 Phantom Read

This error occurs, when the developer first checks if the value of a shared variable is not equal to zero. If this is true, a division through this shared value is done. With optimistic concurrency, there is a high chance that another thread might change the value of the shared variable to zero right after the check is done but before the division is applied. This will lead into an exception.

```
// Thread 1
Transaction {
  int result = 10;
  if(shared != 0) {
    result /= shared;
  }
}
```

```
// Thread 2
Transaction {
  shared = 0;
}
```

Figure 51 Phantom read

$$s = r_1(shared)w_2(shared)c_2r_1(shared)c_1$$

Because a TM implementation cannot distinguish between semantic errors caused by the developer and errors that occurred due to concurrency, it is not possible to simply catch every exception and retry the transaction. So it is necessary to ensure on every read access that the value did not change

since the first read. Of course, this error can also happen to object references which may be nulled or anything else that can cause runtime errors.

11.2.3 Fuzzy Read

This case is related to the error described in chapter 11.2.2 and will result in the same errors. But rather than directly checking the targeted variable, it uses a separate variable to check the state of another.

```
// Thread 1
Transaction {
  int result = 10;
  if(!sharedFlag) {
    result /= sharedNumber;
  }
}
```

```
// Thread 2
Transaction {
  sharedFlag = true;
  sharedNumber = 0;
}
```

Figure 52 Fuzzy read

$$s = r_1(sharedFlag)w_2(sharedFlag)w_2(sharedNumber)c_2r_1(sharedNumber)c_1$$

With this case, it is required to ensure that not only the currently read variable did not change, but all previously read variables.

11.2.4 Starvation

If there are many threads that only read variables and few threads that write, there is a high chance that the writing threads run into starvation, especially if Read/Write lock semantics are used to ensure atomicity while committing changes.

```
// 1 writing thread
Loop {
  Transaction {
    shared += 1;
  }
}
```

```
// N reading threads
Loop {
  int current;
  Transaction {
    current = shared;
  }
  print(current);
}
```

Figure 53 Starvation

To avoid such starvation cases, it has to be ensured that every thread gets its chance to successfully complete once in a while.

11.2.5 ABA Problem

The ABA Problem describes a semantic error with e.g. three threads. Thread 1 starts a transaction and reads the value A from a shared variable. Thread 2 changes the value to B. Finally, Thread 3 changes the value back to A. Both, thread 2 and 3, commit successfully their changes while Thread 1 is still active. Now Thread 1 wants to complete its transaction. As the value of the shared variable is still (or again) A, it assumes there was no change. In many cases, this is probably not a big deal, but there are cases in which this is an error.

```
// Thread 1
Transaction {
  if(shared == A) {
    // ...
  }
}
```

```
// Thread 2
Transaction {
  if(shared == A) {
    shared = B;
  }
}
```

```
// Thread 3
Transaction {
  if(shared == B) {
    shared = A;
  }
}
```

Figure 54 ABA Problem

$$s = r_1(shared)r_2(shared)w_2(shared)c_2r_3(shared)w_3(shared)c_3c_1$$

Because of this phenomenon, it is not sufficient to simply check for value equality when committing.

11.2.6 Use of functions with side-effects

Depending on the restrictions of the STM implementation, it might be possible to call arbitrary functions, maybe even to an external component which is not under control by the STM implementation. A tangible example is causing output onto a console. The semantics require that if a transaction has not yet successfully committed, nothing outside of the transaction may observe any changes made. Causing an output to a console though, would be visible instantly and cannot be undone. The following example demonstrates such a case:

```
// Thread 1
Transaction {
  newValue = shared + 1
  print(newValue)
  shared = newValue
}
```

```
// Thread 2
Transaction {
  newValue = shared + 1
  print(newValue)
  shared = newValue
}
```

Figure 55 Side-Effects problem

Were the transactions really serialized then only two outputs on the console would occur, regardless of how many times the transactions conflict and retry. Since there is no way the STM system can intercept such changes, it is not possible to provide any implicit synchronization control, hence violating the rules of transactions. The STM implementation is essentially left with two approaches for the problem. Either actively restricting such instant side-effects (similar to the way HTM does) or by allowing them and handing over the responsibility to the developer. Of course the STM system could provide support for both solutions so the programmer can explicitly describe which behavior he expects by the transaction.

11.2.7 Retry

Modeling transactions has very descriptive characteristics. Critical sections of code are enclosed with a transaction block hence describing that this must be executed under transactional semantics. The concept of retry though is very imperative. Retry is a specific instruction stating that the transaction must be rolled back and started again. By the use of this instruction it is easily possible to create never ending transactions due to infinite retries (e.g. through a programming error). The transactions then essentially live-lock which violates the semantics introduced by the enclosing transaction block.

Unsatisfied retry condition

Depending on how a retry functionality is implemented, there is a possibility to create never ending transactions. This can happen, if Thread 1 awaits the change of a shared variable using a retry function. Thread 2, which is meant to change the value Thread 1 is awaiting, awaits the change of a shared value Thread 1 is meant to do.

```
// Thread 1
Transaction {
  if(shared2 == 0) {
    Retry;
  }
  shared1 = 1;
}
```

```
// Thread 2
Transaction {
  if(shared1 == 0) {
    Retry;
  }
  shared2 = 1;
}
```

Figure 56 Unsatisfied retry condition

In this example, both transaction will wait forever as their retry conditions will never be satisfied.

Retry with no shared value

There might be a chance that the developer uses the retry feature where the actual retry condition does not rely on any shared value but only on some local values. So it is not enough to only monitor the shared variables which have been read up to this point.

```
Transaction {
  if(systime() < Threshold) {
    Retry;
  }
}
```

Figure 57 Retry with no shared variable

The example shown in Figure 57 has a retry condition without even reading a shared variable before. If the retry mechanics were made to only fire upon a change on a variable which was read within the transaction, this thread would stop forever.

11.2.8 Referencing local variables

Accessing variables outside of the transaction block can result in multiple different behaviors. Some may be intended, others not.

Figure 58 shows a correct usage of an external, non-shared variable which may be of a method local scope to provide additional information to the transaction.

```
int value = 2;
Transaction {
  shared = value;
}
```

Figure 58 Pass data using a non-shared variable to a transaction

This works because there are no side effects. In contrast, Figure 59 shows an invalid usage as this will cause side effects which are most likely not what the developer expects.

```
int value = 2;
Transaction {
  value += shared;
}
```

Figure 59 Invalid usage of non-shared variable inside the transaction scope

Due to retry semantics of conflicting transactions, the local variable `value` will most likely contain garbage. There might be situations in which the developer wants this information, e.g. to simply track the number of retries a transaction needed to complete. Figure 60 shows such a use case.

```
int retries = -1;
Transaction {
  shared = 10;
  retries += 1;
}
```

Figure 60 Counting transaction retries with side-effects

This especially comes in handy if someone would like to limit the number of manual retries (Retry function) to a certain amount, as long as the implementation does not already provide such functionality.

Only writing to non-shared variables is also a correct implementation as is demonstrated in Figure 61.

```
int result;
Transaction {
  result = shared;
}
```

Figure 61 Using variables to get transaction results

But with assignment only actions, one has to be careful that all execution paths of the code are covered, or the variable has to be reset prior to each execution. Figure 60 Counting transaction retries with side-effects demonstrates how bugs may occur because of the retry semantics. As an example, the first try runs through the if-block setting `success` to `true`. But while committing, a concurrent change to `shared` may have happened, leading to a retry of the transaction. Now the alternative execution path is run and completing the transaction successfully. So, even though one would expect the `success` variable to be set to `false`, it will hold `true` because of the previously failed execution.

```
bool success = false;
Transaction {
  if(shared == 1) {
    success = true;
  }
}
```

Figure 62 Using variables without covering all execution paths

If side-effects are needed, such as previously demonstrated in Figure 59, one would need to manually declare the variable as shared, even though the variable is never really shared with other transactions. This is needed, so the implementation is aware of this variable and can ensure a consistent state. An alternative would be covering all execution paths as previously mentioned

11.2.9 Nested-Transactions with Exception-Handlers

During transactions, exceptions may arise. This can be due to programmatic errors by the developer or even intended situations. In the case that nested transactions are flattened – being executed on the same level as the outermost – this may lead to inconsistent data when used with exception handlers. Figure 63 illustrates such a case. On the level of the outer-most transaction, the value of `shared` is set to 1. But within the sub-transaction its value is changed to 2 and an exception is thrown immediately. The exception thrown by the sub-transaction is handled by the outer-most transaction which will commit successfully as no conflict with other transaction has occurred.

```
Transaction {
  shared = 1;
  try {
    Transaction {
      shared = 2;
      throw new Exception();
    }
  } catch(Exception) {}
}
```

Figure 63 Nested transactions with exception handler

With flattened transactions, the above code would result in the value of `shared` being set to 2 when committed, but a value of 1 would be expected.

This is not an issue as long as these exceptions are not handled within outer transactions. But in this case, it is handled by the outer-most transaction. So an implementation which simply flattens transactions will never know that an exception was raised and will not be able to restore the correct values. Furthermore it even publishes invalid values.

11.3 Retry Strategies

This section describes the various common retry strategies and their applicability.

Instant

This is probably the simplest implementation of a retry mechanic. With this implementation, no special handling is needed. It will simply retry the transaction just in time the retry was invoked. Therefore every execution path is surely covered but will lead to a spinning loop for as long as the conditions of the transaction are not met.

On Time

This strategy is mostly the same as an instant retry where simply a waiting time is given before the transaction is retried. This reduces hot spinning but leads to that changes are not immediately visible to the retrying transaction. This also holds a risk of starvation, especially when transactions use different wait times.

On Commit

With the on commit mechanics, a retrying transaction will be stopped until another transaction has successfully committed. This implementation will also cover every execution path beside the ones that are not dependent on shared variables. As an example, a transaction might wait until a specific time to come into action, but this is most likely not a case that should be covered by a transactional memory implementation.

On Read-Set Change

This implementation monitors the read-set of a transaction in order to restart the retrying transaction. As soon as a transaction commits it will check against the read-set to find out if any of the read variables have changed. If a change is detected, the transaction is restarted. If not, it simply waits until another transaction commits.

Observe

Observe enables the users to manually provide one or more shared variables that should be observed for changes. This is a very similar strategy as *On Read-Set Change* with the modification that the responsibility is moved from the implementation to the user.

11.4 Evaluation

Table 5 lists the raw results of all benchmarks in milliseconds. It shows once more why the Shielded benchmarks have not been added to the graphs. Because the execution times are usually very high compared to *[STM]4.NET*, especially on a higher number of threads, and would have stretched the scales too much.

Framework	**Number of threads**				
No Contention	1	2	4	8	16
[STM]4.NET STM	50941 ms	36017 ms	29886 ms	23867 ms	21210 ms
[STM]4.NET HTM	20055 ms	22119 ms	33718 ms	24863 ms	21546 ms
Scala STM	13401 ms	14586 ms	10388 ms	6879 ms	5724 ms
Shielded	130363 ms	121932 ms	183280 ms	237002 ms	271813 ms
HTM.NET	8040 ms	17803 ms	n/a	n/a	n/a
Low Contention	1	2	4	8	16
[STM]4.NET STM	46879 ms	40275 ms	31749 ms	25741 ms	34010 ms
[STM]4.NET HTM	18792 ms	22835 ms	28093 ms	26814 ms	35979 ms
Scala STM	12823 ms	10826 ms	6739 ms	4898 ms	4930 ms
Shielded	128513 ms	114983 ms	189247 ms	248361 ms	278933 ms

High Contention	1	2	4	8	16
[STM]4.NET STM	60030 ms	70865 ms	89814 ms	115444 ms	122969 ms
[STM]4.NET HTM	n/a	n/a	n/a	n/a	n/a
Scala STM	19888 ms	56425 ms	78964 ms	136791 ms	204716 ms
Shielded	451630 ms	703518 ms	947280 ms	1136055 ms	1376183 ms
Large Transactions	1	2	4	8	16
[STM]4.NET STM	4217 ms	4429 ms	5078 ms	7355 ms	7648 ms
[STM]4.NET HTM	3054 ms	4450 ms	5283 ms	7213 ms	7622 ms
Scala STM	562 ms	1232 ms	1809 ms	3275 ms	4149 ms
Shielded	4205 ms	4845 ms	5720 ms	7204 ms	38868 ms
Read Only	1	2	4	8	16
[STM]4.NET STM	39604 ms	41288 ms	45717 ms	35366 ms	32179 ms
[STM]4.NET HTM	14585 ms	15258 ms	45409 ms	37657 ms	36052 ms
Scala STM	6225 ms	4414 ms	4259 ms	2184 ms	1950 ms
Shielded	24135 ms	30745 ms	26474 ms	13565 ms	12834 ms
Read only with objects	10	15	20	25	30
[STM]4.NET STM	207567 ms	300020 ms	384040 ms	516280 ms	614913 ms
[STM]4.NET HTM	197564 ms	300076 ms	411881 ms	530576 ms	685353 ms
Scala STM	7754 ms	14227 ms	16568 ms	18374 ms	28297 ms
Shielded	81961 ms	127233 ms	152736 ms	225946 ms	258841 ms

Table 5 Raw Benchmark Results

11.5 Usage

Integrating a new transaction statement block would have been the desired way. However this would have most likely resulted in breaking existing tools like syntax checker and requiring an additional pre-processor. Therefore a syntax using lambda expressions was chosen, which is very similar to having a dedicated keyword. Figure 64 illustrates how a transaction is executed in *[STM]4.NET*.

```
Transaction.Execute(() => {
  // Do work
});
```

Figure 64 Executing transactions

Accessing fields from within transactions does not require additional statements compared to other transactional memory solutions. This is possible due to post-compilation code instrumentation. Figure 65 sketches how fields are accessed inside a transaction.

```
private int x = 3;
private int y = 2;

Transaction.Execute(() => {
  if(x > y) {
    x = y;
  }
});
```

Figure 65 Accessing fields within a transaction

To prevent the instrumentation from breaking class's interface, it is only allowed to use private fields within transactions. If a field needs a higher visibility than private, one should use properties where .NET's auto-properties are also supported as denoted by Figure 66.

```
public int X { get; set; } = 3;
public int Y { get; set; } = 2;

Transaction.Execute(() => {
  if(X > Y) {
    X = Y;
  }
});
```

Figure 66 Using auto-properties within a transaction

In order to verify that an assembly has been instrumented, a method is provided to do so as shown in Figure 67. If the verification fails, an exception of type `Stm4Net.Runtime.Exceptions.AssemblyVerificationException` is thrown.

```
try {
  Transaction.VerifyAssembly(Assembly.GetExecutingAssembly());
} catch(AssemblyVerificationException) {
  // Assembly was not instrumented
}
```

Figure 67 Ensuring that an assembly has been instrumented

In case a transaction requires a retrial – may it be that not all conditions are fulfilled or any other reason – this can be accomplished using a single statement. The retrying transaction will be restarted as soon as another transaction has committed.

```
Transaction.Execute(() => {
  Transaction.Retry();
});
```

Figure 68 Retrying transactions

Figure 69 shows that cancelling transactions requires the developer to manually catch an exception of type `Stm4Net.Runtime.Exceptions.AbortException`.

```
try {
  Transaction.Execute(() => {
    Transaction.Abort();
  });
} catch(AbortException) {
  // Transaction was aborted
}
```

Figure 69 Cancelling transactions

As nested transactions are fully supported, it can also handle the cancellation of a transaction inside another which is demonstrated in Figure 70.

```
Transaction.Execute(() => {
  try {
    Transaction.Execute(() => {
      Transaction.Abort();
    });
  } catch(AbortException) {
    // Nested transaction was aborted
  }
  // continue with work
});
```

Figure 70 Cancelling nested transactions

Since *[STM]4.NET* uses exceptions to handle conflicts during transactions, one must not use a general exception handler within transactions. Otherwise correct execution of transactions cannot be guaranteed anymore. Furthermore the orElse Feature [28] is also not supported.

11.6 Visual Studio Integration

[STM]4.NET provides tooling support for Visual Studio 2015. It assists the developer with refactoring lock-Statements to their transactional equivalent. As soon as the *[STM]4.NET* runtime has been referenced within the project, the plugin highlights all lock-statements and suggests the refactoring as illustrated in Figure 71.

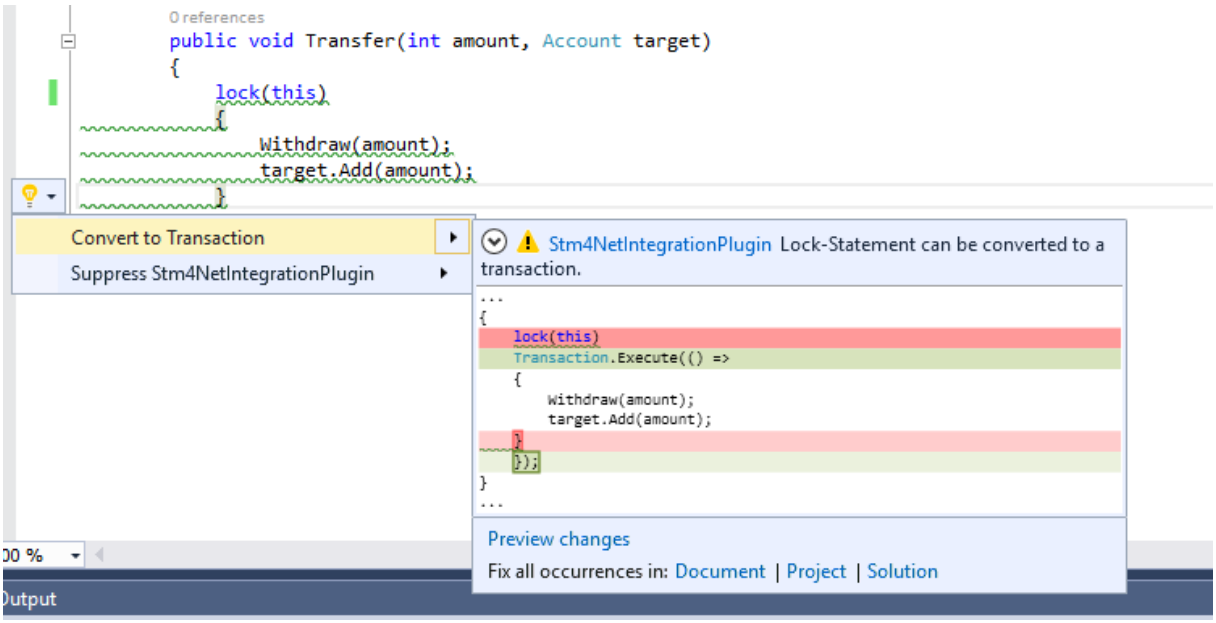


Figure 71 Visual Studio Refactoring Support

As mentioned in chapter 11.5, fields with a visibility other than private are not supported. To help the developer with this restriction, the instrumentation tool prints errors in a format which can be interpreted by visual studio making the usage fairly simple. Figure 72 demonstrates an instrumentation error where the field `Balance` on line 7 of the file *Account.cs* has no private visibility.

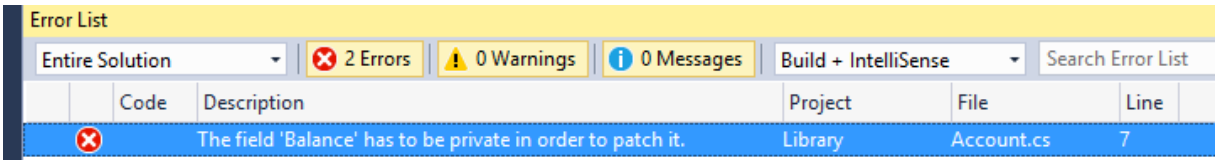


Figure 72 Instrumentation errors in Visual Studio

Also, the usage of functions with side-effects is detected and reported as illustrated in Figure 73. Every call to a method of a class not being part of the solution is considered a side-effect because it cannot be instrumented.

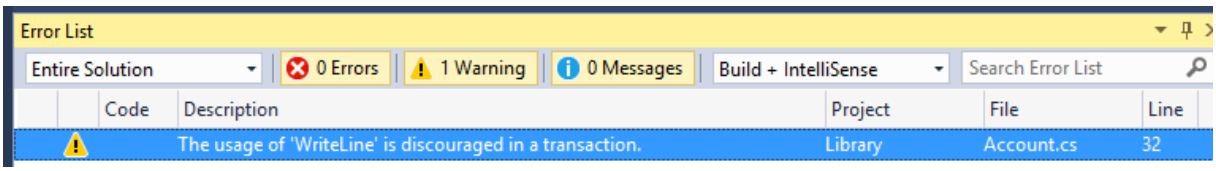


Figure 73 Side-effect warning in Visual Studio

Usually instrumented code leads to a degraded debugging experience. Therefore the *[STM]4.NET* instrumentation tool ensures that the debugging information is correctly updated. Figure 74 shows how that the local field *_balance* of type *int* has been converted to the type *Shared<int>*. While debugging, the developer sees the transaction's local value *Value* and the globally visible value *_value*.

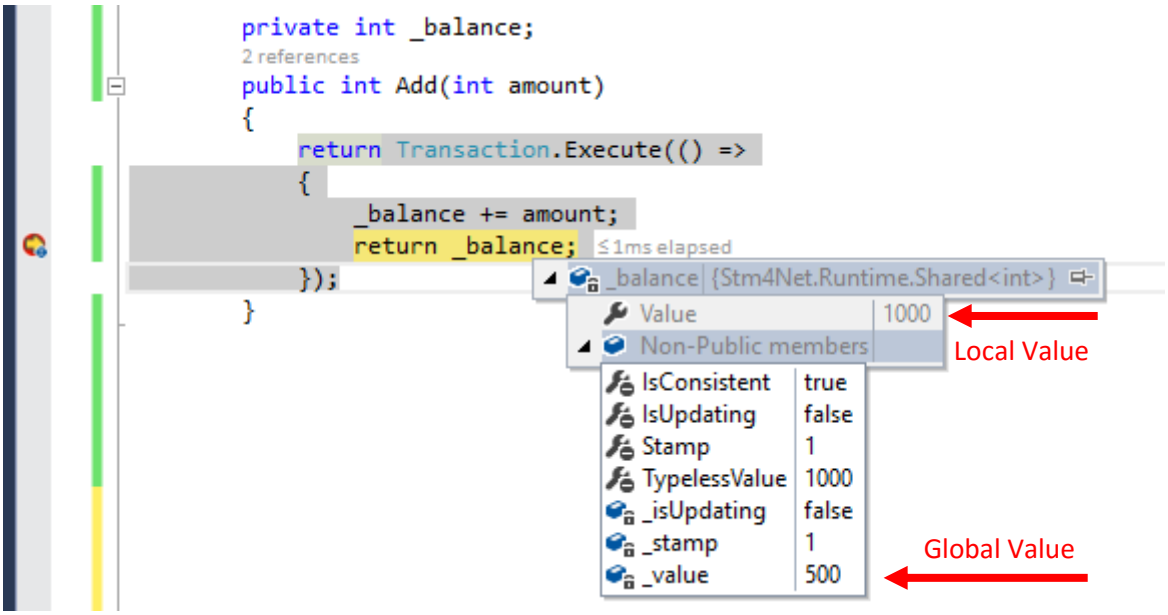


Figure 74 Debugging in Visual Studio