Université de Neuchâtel unine

IIUN

University of Neuchâtel
Computer Science Department (IIUN)

Master of Science in Computer Science

Non-Invasive Software Transactional Memory on top of the Common Language Runtime

Florian George

Supervised by Prof. Pascal Felber

Assisted by Patrick Marlier

August 16, 2010

This page is intentionally left blank

Table of contents

1 Abstract 3
2 Introduction 4
3 State of the art 6
4 The Common Language Infrastructure 7
4.1 Overview of the Common Language Infrastructure 8
4.2 Common Language Runtime 9
4.3 Virtual Execution System 9
4.4 Common Type System 10
4.5 Common Intermediate Language 12
4.6 Common Language Specification 13
4.7 Problematic CLI features for STM 14
5 Common STM Framework (CSTMF) 19
5.1 Overview of CSTMF 20
5.2 Offline assembly transformation 21
5.3 STM algorithms 28
5.4 Unsafe access API 30
6 STM .NET 31
7 Results 33
7.1 Test environment 34
7.2 Benchmarks 36
7.3 Exception throwing performance 37
7.4 Overhead 39
7.5 Runtimes comparison 41
7.6 CSTMF Performance 44
8 Conclusion 50
9 Table of Figures 51
10 References 53

Glossary

AOT	Ahead-Of-Time [compilation]
API	Application Programming Interface
Assembly	Loose equivalent of jar files of the Java environment. File containing managed modules (IL and metadata) and resources. Assemblies are valid PE32 files, same as executables or dynamic link libraries (DLL) on Windows.
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CLS	Common Language Specification
CPU	Central Processing Unit
CSTMF	Common STM Framework. Our STM implementation for the CLR.
CTS	Common Type System
GC	Garbage Collector
GL	Global Lock [STM context]
JIT	Just-In-Time [compilation]
JVM	Java Virtual Machine
LGL	Logged Global Lock [STM context]
STM	Software Transaction Memory
VES	Virtual Execution System

1 Abstract

As most of the effort toward STM is targeted either at C/C++ or Java environments, we found interesting to study if the Common Language Runtime (CLR) would reveal to be a good platform for Software Transaction Memory (STM). The CLR, better known as the *.NET Framework* or *Mono*, is the implementation of the standardized Common Language Infrastructure (CLI).

The CLI is interesting considering that it was designed to be independent of high-live programming languages. To that end, it contains a vast amount of features such as generics, type-safe data and function pointers, custom value types and more. As a consequence, the CLI is a lot more complex than the JVM. While this complexity is incredibly beneficial for programming language implementers, it could instead be an obstacle for a lower-level feature such as an STM. To get an answer, we decided to perform an in-depth analysis of the CLI and implement our own non-invasive language independent STM on top of the CLR.

Our analysis of the CLI showed that without explicitly customizing a CLR for STM, covering the entire feature set of the CLI is not feasible. Specifically, the managed pointers used for implementing, among other features, custom value types, cannot be supported inside transactions as they cannot be stored outside of the stack.

We also present a complete STM framework for the CLR, called CSTMF. It is intended as a platform for studying the suitability of the runtime for non-invasive STMs. CSTMF, which stands for Common STM Framework, uses a field-based STM approach allowing fine granularity. Our STM takes much inspiration from the Java STM framework called DEUCE which demonstrates a non-invasive Java STM with reasonable performance without compiler support [1]. This is due to the fact that both STM frameworks, albeit being on two different platforms, share a lot of objectives.

The results of this first release of CSTMF are very promising with a remarkable scalability, especially when running on the .NET Framework 4. We also compared it to the STM that Microsoft Research published during our study. It is a research implementation of the .NET Framework 4 with STM enabled. By modifying the runtime itself, it allows the STM to support more features of the CLI during transactions. However, while it performs comparably to CSTMF on the .NET Framework 4 at low conflict rate and low concurrency, its scalability drops quickly with higher update rates and more parallelism and is outperformed by CSTMF.

With this research, we showed that an efficient non-invasive STM can be implemented on top of the CLR, even if not of all of its features can be covered.

2 Introduction

With the processors hitting the limits of clock speed a few years ago, manufacturers turned to multicore CPUs to keep on releasing products with improved performance. Today, almost all portable, desktop or server computers sold have such CPUs. The trend has even reached the embedded market where dual-core ARM processors have emerged. However, this change in architecture means that software must be adapted to take advantage of the full power of these CPUs. Whereas advances are quickly made for the multicore hardware, the same cannot be said of the software.

The commonly used approach to multicore programming is based on thread synchronization using locks. However, usage of locks is difficult and error-prone; often leading to poor scalability with coarse-grain locking or to deadlocks, starvation, priority inversion, etc. with fined-grained locking. Software Transactional Memory (STM) aims to simplify concurrent programming by providing a trouble-free approach, yet with a good scalability.

In the STM model, all accesses to a shared memory are performed within transactions. The STM transactions are atomic and isolated. They provide an "all-or-nothing" methodology. They must either complete in their entirety or have no effect. In addition, unlike the locking approaches, STM is based on an optimistic model. Threads perform accesses, within transactions, to the shared memory concurrently but record them in the transaction log. It is only at the end of a transaction that they check that no conflict on the accessed memory occurred with another thread. Different threads can therefore safely and simultaneously modify disjoint parts of a data structure that would normally be protected under the same lock. If a conflict is detected, the transaction is aborted and its changes rolled back, leaving the shared memory untouched. Transactions are usually automatically retried until they succeed. They can also be aborted and rolled back at any time, for example in case of an exception in user code.

Efforts made in the past several years in STM research mainly targeted C/C++ and Java. This is not unexpected as according to studies on the popularity of programming languages [2], they are at the top. In addition, Java is omnipresent in the academic world where a lot of the STM research is done.

However, there is another runtime environment gaining popularity: the Common Language Runtime (CLR). Two main implementations exist. The best known is the .NET Framework of Microsoft which is slowly becoming the main software environment on the company's platforms. The second is the open source framework Mono [3] that, despite being very controversial in the open source world (such as in [4] and [5]), is also constantly becoming more prevalent. Both

runtimes are implementations of the Common Language Infrastructure (CLI) specification which defines a full-featured runtime environment.

In our opinion, the key feature of the CLI, which makes it stand apart from other runtimes or frameworks, is its independence from high-level programming languages. The list of CLI languages is quite considerable [6], and with the very complete type system of the CLI, most of these languages are first-class citizens. In fact, not even C#, which is standardized along the CLI, covers the whole set of features of the CLI.

The objective of this research is to analyze if the CLR is well adapted for the integration of a non-invasive STM framework implementation built on top of it. That is, an STM without compiler support and which only requires minimal modifications to existing code written in any compiled language supported by the CLI. The STM must run on official releases of the two CLRs; therefore, customizing a CLR to our needs is not an option. In addition, we want the framework to allow different STM algorithms to be plugged in so that they can be easily compared. As the requirements for our STM implementation are very similar to those of DEUCE [1] – a Java STM with convincing results to which the University of Neuchâtel contributed – we will take inspiration from it. The analysis of the CLI will therefore also use the JVM as a basis for comparison as it is the best known managed environment and DEUCE's platform.

The first part of this document will cover the study of the CLI; from a global overview to an examination of specific features which impact STM. The second part is a complete analysis of our Common STM Framework or CSTMF, the STM framework created during this research with the objectives previously mentioned.

3 State of the art

While the research community is very productive in term of STM research and implementations, the vast majority of the work is focused on the C, C++ and Java platforms. Many STMs have been implemented for these software environments; such as TinySTM [7] for C/C++ or DEUCE [1] and DSTM2 [8] for the JVM. However, dramatically less attention has been paid to the Common Language Infrastructure.

In 2005 Microsoft Research released its first publicly known effort as an implementation called SXM [9]. It uses an object-based STM approach with runtime code generation and atomic wrappers. It also integrates contention managers to enhance the performance of conflict resolution. While it is not as invasive as STM implementations where all data accesses have to be manually redirected, it requires the usage of factories to instantiate atomic objects and imposes considerable constraints on classes used atomically. Several different contention managers and factories are implemented and more can be plugged in.

In 2007, a new STM implementation, called NSTM [10], was released. Although it was developed outside of Microsoft Research, it was inspired by SXM. It is a more invasive STM than SXM as not only it uses factories, but all data accesses have to be done through the STM. However, any primitive, value or cloneable type can be wrapped atomically. In addition, it is fully integrated in the transaction infrastructure of the runtime used by transactional components such as the database providers. Using generics (introduced in the third edition of the CLI specification just after SXM's release), the API is very straightforward.

Finally, more recently at the end of July 2009, Microsoft Research released its second .NET STM effort in the form of a beta version of the .NET Framework 4.0 with STM support built-in: STM.NET [11]. This time the STM is directly integrated in the runtime itself. It is a non-invasive STM with very few constraints and limitations. However there is no API to change the STM algorithm. A more comprehensive analysis of STM.NET is available in chapter 6 of this document and its results are compared with CSTMF. Note that STM.NET was made public after the start of our research.

4 The Common Language Infrastructure

4.1 Overview of the Common Language Infrastructure 8

4.2 Common Language Runtime 9

4.3 Virtual Execution System 9

4.4 Common Type System 10

4.4.1 Primitive types 10

4.4.2 Reference types 11

4.4.3 Value types 11

4.5 Common Intermediate Language 12

4.6 Common Language Specification 13

4.7 Problematic CLI features for STM 14

4.7.1 Managed Pointers 14

4.7.2 Non-vector arrays 17

4.7.3 Exception throwing performance 18

4.1 Overview of the Common Language Infrastructure

The Common Language Infrastructure (CLI) is an open specification (published under ECMA-335 and ISO/IEC 23271) developed mainly by Microsoft that defines a complete runtime environment. The first version of the specification was ratified by the ECMA in December 2001, followed by ISO standardization in April 2003. It is currently at its fourth revision and ratification is under way for the fifth edition.

The CLI includes a type system, metadata, an intermediate language, rules for language interoperability and an execution environment. Its main components are described in the following subchapters. Figure 1 shows how the parts relate to each other.

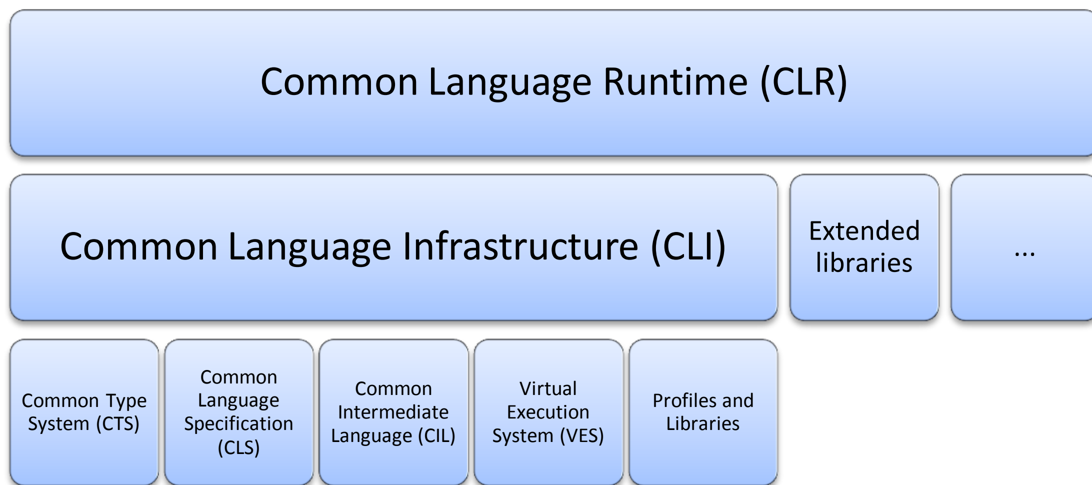


Figure 1 - The Common Language Infrastructure components

The CLI has been designed to support multiple programming languages on top of it and to stay completely independent from all of them. Therefore its type system and intermediate language offer features that could seem redundant or unnecessary for certain languages. For example, even though the C# language is commonly considered as the main programming language on the platform as it is standardized along the CLI under ECMA-334, it does not make use (as of now) of all the features exposed by the CLI such as tail calls, non-zero based arrays and exception filters to name a few. But any other languages (e.g. Visual Basic, Scheme, etc.) can take advantage of these features that greatly reduce the complexity of their implementation and increase their performance by not having to emulate them. This makes the CLI a powerful multi-language platform [12].

The CLR has a very extensive platform coverage. Currently Microsoft has a variety of supported implementations of the CLI with the .NET Framework on Windows, the .NET Compact Framework on Windows CE, the XNA Framework on the

Xbox360 and the Zune, Silverlight in the main web browsers on both Windows and Mac OS X and even the .NET Micro Framework directly on embedded 32-bit processors with as little as 64kB of RAM. Novell's Mono has nothing to envy with platform support for Linux on 10 different processor architectures and other operating systems including Windows, Mac OS X, the iPhone, Solaris, Nintendo's Wii and more[1].

4.2 Common Language Runtime

A Common Language Runtime is an implementation of the Common Language Infrastructure (CLI). It implements all the features defined in the CLI while completing it with its own Garbage Collector (GC) algorithm, a complete set of libraries that extend those defined in the CLI and additional features such as COM Interoperability[2] or SIMD instructions support[3]. Early literature only mentions Microsoft's .NET Framework when talking about the CLR as it was the only implementation at the beginning. For the sake of clarity, future references of the CLR in this document will always refer to any version of any implementation of the CLI, be it the .NET Framework or Novell's Mono.

4.3 Virtual Execution System

The Virtual Execution System (VES) provides an environment for executing managed code. It provides direct support for a set of built-in data types, defines a hypothetical machine with an associated machine model and state, a set of control flow constructs, and an exception handling model. To a large extent, the purpose of the VES is to provide the support required to execute the CIL instruction set. We do not have to pay more attention to it as we will not modify the runtime for our STM implementation.

---

[1] Mono Supported Platforms: http://www.mono-project.com/Supported_Platforms

[2] COM Interoperability: http://msdn.microsoft.com/en-us/library/bd9cdfyx.aspx

[3] SIMD support in Mono: http://tirania.org/blog/archive/2008/Nov-03.html

4.4 Common Type System

The Common Type System (CTS) is the central piece of the CLI's programming language independency. It provides a rich type system that supports the types and operations found in many programming languages. Indeed, as explained previously, it is designed to support the full range of features necessary for the implementations of an extensive variety of programming languages.

The CTS defines three basic kinds of types: primitive, reference and value types. The types presenting a challenge for an STM implementation are discussed further in the chapter 4.7 *Problematic CLI features for STM*.

4.4.1 Primitive types

The CTS defines a wide range of primitive types that are stack-based and for which there is dedicated CIL instructions. This includes the one found in the JVM such as the fundamental object class, a Unicode string (UTF-16), integral numbers from 8 to 64-bit and the two IEEE-754 floating-point numbers, a Unicode character value and a boolean. In addition, the CTS has a natural-size integer (32 or 64-bit depending on the processor architecture) and unsigned variants of integral numbers. They all have a corresponding type in the class library, meaning that methods can be invoked directly on them; in this case they behave like value types. The *String* class is also considered a primitive type, but it is heap-allocated. Programming languages usually alias the primitive types to their own built-in types; for example, the C# *int* type is simply an alias for *System.Int32*. All the primitive types, including *String*, are immutable; their methods and the CIL instructions always return a new instance instead of modifying them in-place. Therefore an STM implementation does not have to transform their manipulation instructions nor their methods. Figure 2 shows different usages of primitive types in C# and their corresponding CIL instructions

```
int i = 1;                  // loading constant with ldc.i4.1 instruction
i = i * 2;                  // arithmetic operation with mul instruction
string s1 = "foo";          // loading string with ldstr instruction
string s2 = i.ToString();   // call instruction to method
                            //   string System.Int32::ToString()
string s3 = s1 + s2;        // call instruction to method
                            //   string System.String::Concat(string, string)
```

Figure 2 – Usage of primitive types

4.4.2 Reference types

Java developers will also be familiar with reference types which are allocated from the managed heap. They include classes which store data in fields (which can be of any of the three basic kinds of types hereby described) and arrays. Reference type instances can be shared between threads; thus it is the accesses to the classes' fields and arrays' elements that an STM implementation must take care of.

Besides classes and single-dimensional arrays, the CTS also defines other reference types: true multi-dimensional arrays and safe function pointers (called delegates).

4.4.3 Value types

The third variety, the value types, does not exist in the JVM. They may seem similar to C structures (and the C# language call them *struct*) but have important differences. It can have methods and a value type is always allocated on the stack or inlined if it is a member of a heap object such as a reference type's field or an array member. Therefore, they are not individually tracked by the garbage collector and do not contribute to GC pressure. Like reference types, they can contain fields and methods (instance and static); however, they cannot implement interfaces or use inheritance. This is due to the fact that a value type instance has no *virtual method table* or any object header at all as found in instances of reference types; only the space occupied by the fields is present in memory. When calling a method of a value type instance, the *this* implicit first argument of the method is a managed pointer to its first field in memory. This makes value types problematic for an STM implementation (see next chapter 4.7 about problems with the use of managed pointers).

4.5 Common Intermediate Language

The Common Intermediate Language (CIL)[4] is the specification of the CLI's instruction set both in its binary (byte-code) and assembly forms. It is fully documented in the CLI specification and extremely well detailed in literature such as [13]. Both Mono and the .NET Framework include an assembler in the redistributable package along usual compilers such as the C# one. Understanding the CIL was a very important part of our research as the redirection of the data accesses to the STM is performed by modifying the CIL instructions.

To accommodate as much programming languages as possible, the CIL has a more diversified intermediate language than the JVM and a larger set of instructions. While the total number of the CIL instructions is not that bigger, the instruction set is however much less redundant. Indeed, unlike the JVM, most instructions do not have the types of their arguments hard-coded in them; they are polymorphic. There are also small details that make life easier for compiler writers; for example there is no distinction between calling a virtual method and an interface method.

The additional instructions come from multiple families of instructions not present in the JVM. The biggest supplement is to address CLI's support of indirect addressing; all types of load instructions (field, arguments, etc.) have a variant that returns a managed pointer to the value instead of the value itself and the CIL has a complete family of instructions dedicated to indirect loading and storing. The second one is made of all the arithmetic and conversion instructions with overflow detection (in the form of an overflow exception) and support for unsigned integers. Specific instructions also exist to address method's arguments. Unlike the JVM, they are separated from the local variables.

All those additional features greatly ease the development of compilers for a wide range of programming languages as most of the time direct intermediate language support is present for their features (including generics). For examples, languages that support passing arguments by reference (Visual Basic, C++/CLI, etc.) or perform intensive arithmetic with overflow detection (such as Ada95 or SML) can benefit from the straightforward, efficient and type-safe dedicated CIL instructions. However, all these benefits for compilers make the runtime itself much more complex and writing an STM implementation on top of it is more difficult with more features to cover and more instructions to transform.

[4] Formerly called Microsoft Intermediate Language (MSIL) such as in [13].

4.6 Common Language Specification

We saw that the purpose of the CTS is to enable the integration of all programming languages by allowing objects created in one language to be treated as equal citizens by code written in a completely different language. This integration is possible because of the standard set of types, metadata and common execution environment of the CTS.

However, while this language integration is a tremendous achievement, programming languages still vary greatly from one another. For example, some languages do not treat symbols with case-sensitivity and some do not offer unsigned integers, operator overloading or methods to support a variable number of arguments. To make possible the creation of types that are easily accessible from any other programming languages, only features that are guaranteed to be available in all other languages must be used. To help with this, the CLI specification defines a Common Language Specification (CLS) that details for compiler implementers the minimum set of feature their compilers must support if these compilers are to generate types compatible with other components written by other CLS-compliant languages on top of CLI. Figure 3 summarizes the idea hereby expressed.

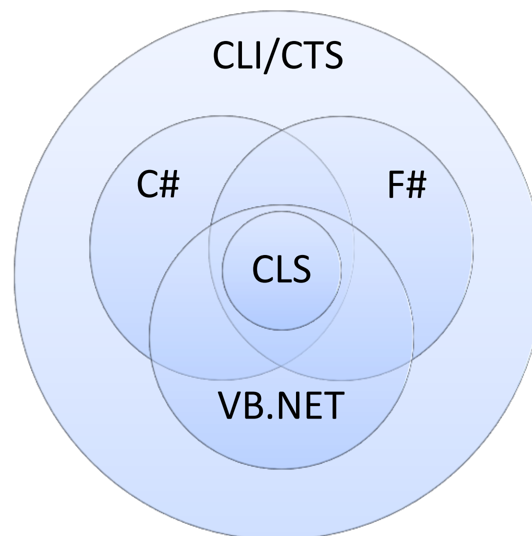


Figure 3 - CLS relation to languages and CLI/CTS

As shown, the CTS support a lot more features than the subset defined by the CLS. So if there is no need for this inter-language operability, richness of types is only limited by the selected language's feature set. Specifically, the CLS defines rules that externally visible types and methods must adhere to if they are to be accessible from any CLS-compliant programming language.

4.7 Problematic CLI features for STM

While broad and redundant features of the CLI greatly simplify the compilers built on top; this has quite the opposite effect when it comes to interpret and modify the output of those compilers to add new features such as STM support. In a simpler runtime as the JVM where there is only one way to access a field or an array's element, intercepting those accesses is quite easy. But when there are multiple ways like in the CLI with indirect addressing, managed pointers and multidimensional arrays, implementing an STM without the support of the runtime becomes much more complex.

The most problematic features in regard to STM are detailed hereafter.

4.7.1 Managed Pointers

The most problematic feature for an STM implementation on top of the CLI is certainly the managed pointers.

Despite their name, managed pointers have much more in common with object references of the managed world than to unmanaged C pointers (which also exist in the CLI in unsafe contexts where type safety and runtime integrity is not guaranteed). They can point to a field, an element of an array, a local variable or a method parameter. What makes them managed is that they are reported to the garbage collector (GC) and fixed-up when the data pointed is moved around. Their usage is far from being rare; calling methods on value types and passing method parameters by reference are both performed using managed pointers.

With this introduction to managed pointers, it would seem that an STM implementation can simply track all data accesses uniquely through managed pointers and support the whole range of features the CLI provides. Indeed, all the types of the CTS can be addressed by managed pointers and load/store instructions can be replaced by their managed pointer equivalent, while the reverse is not true (for example, there is no way to get the object reference that contains the field pointed by the managed pointer). However, the managed pointers, with all their qualities, come with a long list of restrictions that we will analyze. Without these limitations, a CLI implementation would be way too complex to be efficient. Here is the restriction list extracted from the CLI specification.

ECMA-335 at *Partition II - 14.4.2 Managed pointers* and *Partition III - 1.1.4.2 Managed pointers*):

1. Managed pointers can be passed as arguments, stored in local variables, and returned as values.
2. If a parameter is passed by reference, the corresponding argument is a managed pointer.
3. Managed pointers cannot be stored in static variables, array elements, or fields of objects or value types.
4. Managed pointers are not interchangeable with object references.
5. A managed pointer cannot point to another managed pointer, but it can point to an object reference or a value type.
6. A managed pointer can point to a local variable, or a method argument.
7. Managed pointers that do not point to managed memory (garbage collector heap) can be converted (using conv.u or conv.ovf.u) into unmanaged pointers, but this is not verifiable.
8. Unverified code that erroneously converts a managed pointer into an unmanaged pointer can seriously compromise the integrity of the CLI. This conversion is safe if any of the following is known to be true:
    a. the managed pointer does not point into the garbage collector's memory area
    b. the memory referred to has been pinned for the entire time that the unmanaged pointer is in use
    c. a garbage collection cannot occur while the unmanaged pointer is in use
    d. the garbage collector for the given implementation of the CLI is known to not move the referenced memory

The 3rd point straightaway prevents an STM implementation from storing data accesses as managed pointers in the transaction context; which would have been the most logical way. Therefore, they would need to be converted to unmanaged pointers as they are the only type of data managed pointers can be converted to and that can be stored anywhere. But to avoid corruption of memory located in the managed heap when a garbage collection occurs, one of the sub points of point 8 must be true. Here are the possibilities for each of them:

a. Disallow usage of garbage collector's memory area (the managed heap). If memory was not shared by threads, we would not need STM in the first place. Imposing that all shared data must reside in static locations (which are not managed by the garbage collector) is not realistic.
b. Pin memory references during transaction. Pinning is a CLI feature of method's local variables which instruct the GC to not move the referenced data during a collection. But pinning can only occurs at method scope as it

applies on local variables and memory would need to stay pinned until transaction commit or rollback. Therefore the implementation would need to be aware of all referenced managed memory locations at the start of the transaction, before they are actually accessed, so they are scoped at the atomic method's level.

c. A first solution would be to disable the GC during transactions. However neither Mono nor the .NET Framework allows that and, anyway, it could result in the GC never running with transactions overlapping in time (live lock of the GC). The second solution would be to rollback transactions during which a collection occurred to not commit data in memory locations that was moved. But even then, a collection could still occur during the commit itself and cause corruption. Disabling the GC during commits brings us to the first inapplicable solution.
d. The .NET Framework uses a compacting GC which moves the memory around during collection and Mono will switch shortly to a similar GC which is currently being tested. Therefore this cannot be true on both CLRs.

In conclusion it seems that there is no feasible CLI-compliant way to track data accesses through managed pointers and neither can they be safely converted to something else. Hence it is our reasoning that an STM implementation on top of a CLR cannot support managed pointers without coordination from the runtime itself; that is a custom CLR. That fact alone brings a long list of limitations to CSTMF:

- Atomic methods and their subroutines cannot pass parameters by reference. An exception could be made for stack-allocated data with sufficient static analysis.
- Value types' field cannot be accessed; all related instructions take managed pointer to the instance as argument. Instance methods cannot be called on value types as the first argument of the method is also a managed pointer ("this" implicit argument). Once again this would not apply to value types stored on the stack if correctly detected; but the still leaves value types instances in class fields and in arrays.
- As compilers can use managed pointers to perform any data accesses, some programming languages could be more impacted than others even in code that seems non-problematic at first sight.

4.7.2 Non-vector arrays

In addition to the simple zero-based single-dimensional arrays called vectors, the CTS also provides support for more complex forms of arrays: non-zero-based arrays, true multi-dimensional arrays and any mix of the two. However the CIL has only instructions for the manipulation of vectors; creation and access of other arrays must be performed through the methods of the *System.Array* class from which all arrays – including vectors – inherit.

While most programming languages use zero-based arrays, the CTS allows the bounds (a lower bound, an upper bound or both) of an array to be set when specifying an array type. However the CLS requires all arrays to be zero-based to allow a method written in one language to create an array and pass its reference safely to code written in another language without worrying about its indexing. Their use is therefore discouraged and they shall not appear in exported members. In addition to the lack of specific IL instructions, most languages including C# language have no direct support for these arrays anyway. For all these reasons, it is extremely rare to see any code using non-zero-based arrays and their performance has been a lot less optimized in the CLRs. Therefore supporting them in an STM implementation is not crucial and CSTMF does not. On top of that, the correct computation of the offsets for the redirection of its accesses by transactional code would be hazardous and with a very high overhead.

While arrays of arrays, or jagged-arrays, are supported and quite common, the CTS has also support for single-allocation multi-dimensional arrays. When zero-based, these arrays are CLS-compliant. The current implementation of CSTMF does not implement transactional support for them.

Transactional code using non-vector arrays will generate warnings at transformation due to the calls to the non-atomic methods of the *System.Array* class and will not have their accesses transformed.

4.7.3 Exception throwing performance

Effective exception throwing is crucial for good scalability of concurrent transactional code. Indeed, aborting a transaction due to a conflict can only be achieved by throwing an exception. This is the only efficient way of returning from a deep call-tree to the starting method of the transaction. To study the differences between the two CLRs on this aspect, a micro-benchmark was performed. The results are presented in chapter 7.3 on page 37.

5 Common STM Framework (CSTMF)

5.1 Overview of CSTMF ... 20
5.2 Offline assembly transformation ... 21
5.2.1 Why offline transformation ... 22
5.2.2 Atomic methods transformation ... 23
5.2.3 Data accesses redirection ... 25
5.2.4 Type and CIL transformation API ... 27
5.3 STM algorithms ... 28
5.3.1 Global Lock ... 29
5.3.2 Logged Global Lock ... 29
5.3.3 LSA ... 29
5.4 Unsafe access API ... 30

5.1 Overview of CSTMF

Common STM Framework (CSTMF) is our implementation of an STM framework on top of the Common Language Infrastructure. It is a non-invasive language independent STM with the possibility to plug in new STM algorithms.

The main goals in designing CSTMF are to provide an STM that required only minimal modification to make code atomic and allow implementers of STM algorithms to plug in their own. In addition, to take advantage of the multi-language aspect of the CLI, CSTMF cannot be tight to any specific programming language or require syntax additions. Thus it can neither rely on using atomic code blocks as a new language syntax or STM.NET's exception blocks and anonymous methods.

To that end, it uses custom attributes to mark methods as atomic. Custom attributes are a core functionality of the CLI allowing declarative addition of custom metadata to any metadata entry defined by the CLI such as types, fields, methods, method parameters, return values and more. Java annotations introduced in Java 5 are somewhat similar. Figure 4 shows how to mark a method as atomic using the well-known bank account transfer example in a variety of programming languages. The only change made to that method in all languages is the *Atomic* custom attribute above the method declaration.

```
// C#
[Atomic]
public void Transfer(Account source, Account dest, decimal amount)
{
    dest.Deposit(amount);
    source.Withdraw(amount);
}

' Visual Basic .NET
<Atomic> _
Public Sub Transfer(ByVal source As Account, ...

// C++/CLI
[Atomic]
public: void __gc* Transfer(Account __gc* source, ...

(* F# *)
[<Atomic>]
member x.Transfer(source:Account, ...

# Boo
[Atomic]
def Transfer(source as Account, ...
```

Figure 4 - Marking a method as atomic in different CLI languages

In contrast with DEUCE, CSTMF's transformation of methods to their atomic version is done by modifying the assemblies on disk at compile-time or prior to the execution. The modified application can be then launched exactly as it would have before the transformation (remember that .NET assemblies are executables and do not need to be run through a runtime launcher).

Similarly to DEUCE, CSTMF does not require any change to the runtimes. This allows it to run on the official releases of the .NET Framework or Mono. However due to the greater complexity of the CLI, all of its features are not supported inside transactions.

CSTMF is platform neutral; it has been designed to run on any architecture supported by the CLI and was tested on both x86 and x86-64 (AMD64) architectures.

5.2 Offline assembly transformation

The assembly transformation is done offline; the original assembly (executable or library) is loaded from disk, transformed and written back on disk. This is therefore done only once (typically just after compilation) and does not occurs at each run of the application. The transformation has two main sub processes. The first, the *atomic methods transformation*, adds the transaction handling code. The second, the *data accesses redirection*, redirects all accesses to fields or arrays to make them go through the STM context. The rest of this chapter clarifies the design choices and describes the mechanisms in details.

5.2.1 Why offline transformation

The CLI does not contain a managed API for instrumentation as found in the JVM. Although the .NET Framework has an API that could do the job, it is a complex COM-based native API designed for runtime hosting. Mono does not have any.

Even if one existed, it still would not work as expected. The reason is that both CLRs do not necessarily load and process CIL byte-code when executing an application. Both runtimes support and extensively use Ahead-Of-Time (AOT) compilation. All assemblies that are part of the frameworks have a native binary image generated by AOT at installation and application or third-party libraries can freely do the same. This allows reducing start-up time of applications by not having to perform Just-In-Time (JIT) compilation each time for frequently used libraries as well as reducing memory footprint of application by sharing those native images across processes. Byte-code transformation would therefore not work as there would not be any byte-code to transform.

Another possibility for transforming types at runtime is to generate a wrapper type at runtime instead of modifying them. SXM [9] uses this technique to generate an atomic type at runtime by producing a new type that inherits the one to transform and overriding its properties and methods with atomic wrapping code. However, that introduces a lot of limitations such as the requirements that all properties and methods must be virtual, type cannot be sealed (final), etc. Furthermore all transactional objects must be instantiated through factories which is incompatible with our design goal of providing a non-invasive STM framework.

As one of our objectives was to impose as few constraints as possible on types in order to achieve much higher compatibility with existing code, we decided to use offline transformation instead. Assemblies are passed post-compilation to a utility of CSTMF that rewrites the CIL instructions of methods marked as atomic and modify as well all types referenced by these atomic methods. Thus no additional transformation cost is incurred at run-time and there are much fewer constraints imposed on types and methods. Indeed, by modifying the code instead of wrapping it, any transformation can be done to a type's methods as long as no change is visible externally. In addition, by using basic static analysis, only types used inside transactions are transformed, thus reducing the memory overhead caused by unnecessarily duplicating methods. As a consequence of using offline transformation, only types defined outside of the framework libraries are supported inside transactions.

5.2.2 Atomic methods transformation

The transformation process of atomic methods works in the following way. It first looks for methods with an *Atomic* attribute declared. It then duplicates them, suffixing the copies so they have an identical signature. Finally, it replaces the code of the original method with transaction handling code (start, commit, rollback and retry) wrapping the call to the method copy. This way, all calls to the method which was marked with the custom attribute are now calls to the fully atomic transformed method.

Figure 5 and Figure 6 show a simple atomic method before and after its transformation as it would look in C#; however the process is really done at the CIL level and is totally independent of the high-level programming language used.

The added transaction handling code performs the following operations (line numbers between parentheses):

- Store the return value to return it only when the transaction is committed. (5, 18 and 39)
- Wrap the original method call between a transaction start and commit. Transaction is statically attributed a unique identifier. (14-21)
- Execute transaction in a try block and only catch transaction exceptions, letting other exceptions pass. (12-24)
- Call the suffixed method's copy, which is transformed by the *data accesses redirection process* explained next. (18)
- Rollback transaction if try block was exited with the transaction not committed (exception or commit failure). (26-33)
- Call Thread.Sleep(0) after rolling back a transaction to yield the CPU to other threads. This gives priority to threads currently executing a transaction and lets them complete it and release locks they may own.
- Retry transaction until successfully executed and committed. (10-36)

The original method is transformed like any other method reference by an atomic block. The next subchapter describes this process.

```
[Atomic]
private int AtomicMethod()
{
    return foo();
}
```

Figure 5 - Atomic method before transformation

```
1       [Atomic, Transformed]
2       public int AtomicMethod()
3       {
4           // For returned value
5           int ret;
6
7           // Get context
8           IContext context = Context.Current;
9           bool committed = false;
10          do
11          {
12              try
13              {
14                  // Start transaction
15                  context.StartTransaction(0x12345678);
16
17                  // Call original method
18                  ret = AtomicMethod$Atomic$(context);
19
20                  // Try to commit transaction
21                  committed = context.Commit();
22              }
23              // Catch transaction failure
24              catch(TransactionException) { }
25              // Always rollback transaction if not committed
26              finally
27              {
28                  if(!committed)
29                  {
30                      context.Rollback();
31                      Thread.Sleep(0);
32                  }
33              }
34          }
35          // Retry transaction until committed
36          while(!committed);
37
38          // Return value from original method
39          return ret;
40      }
41
42      [Transformed]
43      public int AtomicMethod$Atomic$(IContext $context$)
44      {
45          return foo$Atomic$($context$);
46      }
```

Figure 6 - Atomic method after transformation

5.2.3 Data accesses redirection

Data accesses redirection is a tricky process. The different ways to access data have to be adequately abstracted to the STM algorithm in order that the API of the contexts stays straightforward and non-redundant. After multiple tries using reflection, we found that the approach used by DEUCE is also the most efficient on the CLR. This approach consists of expressing all data accesses as an object and an offset. The offset represents the distance in memory of the field or array element to its containing object's reference. Figure 7 summarizes the possible types of data accesses and what values take the object and offset arguments when passed to the STM context, if supported.

Reference type instance field	
Instructions	ldfld and stfld + reference
Object value	Reference to object instance
Offset value	Field's offset at runtime

Value type instance field	
Instructions	ldfld and stfld + managed pointer
Unsupported (see chapter 4.7)	

Static field	
Instructions	ldflds and stflds
Object value	null
Offset value	Absolute address of field at runtime (does not move)

Vector element (zero-based single-dimension array)	
Instructions	ldelem and stelem + reference and index
Object value	Reference to vector instance
Offset value	Element's offset at runtime (is **not** the element's index)

Array element (all types of array)	
Instructions	call to GetValue/SetValue methods of array
Unsupported in current implementation	

Indirect addressing	
Instructions	ldind and stind + managed pointer
Unsupported (see chapter 4.7)	

Figure 7 - Types of data accesses and corresponding argument to STM context

The redirection of the data accesses is itself a two-steps process.

The first process is the computation of the fields' offsets. It starts with the identification of all types used inside transactions. This is done by recursively going through the complete call-tree below atomic methods and looking for types referenced. Then, for each non-read-only field of these types, an associated static offset field is added. The offset value cannot be statically computed during transformation as the offsets can vary from one CLR implementation to another (e.g. .NET Framework or Mono) and depend as well on the system architecture (x86, x64, etc.). In addition, as the computation is costly and offsets do not change during the lifetime of the application, they are computed once at type initialization and cached in these fields. The class constructor of the type is therefore extended, or created if not already existing, with the computation of these offsets which are stored in the added fields.

The second process is the actual redirection of the data accesses to the STM context. Once again, the call-tree below atomic methods is recursively browsed. All called methods are duplicated as in the *atomic methods transformation* process. But this time the original methods are left untouched. As they can still be called by code outside of a transaction, their behavior must not change at all. However, data accesses in the methods duplicates are changed to call to transactional context methods. Calls to methods are also redirected to their transformed copy.

Figure 8 shows the C# declaration of a field *x* with its associated property *X*; a very common pattern in .NET. Figure 9 shows the same property after being transformed. The additional static field `x$Offset$` contains the offset of the *x* field in memory at runtime in relation to the current instance's reference.

```
private int x;
public int X
{
   get { return x; }
   set { x = value; }
}
```

Figure 8 – C# Property declaration before transformation

```
private int x;
private static readonly IntPtr x$Offset$;
public int X$Atomic$[IContext $context$]
{
   get { return Context.ReadFieldInstance<int>(this, x$Offset$, $context$); }
   set { Context.WriteFieldInstance<int>(this, value, x$Offset$, $context$); }
}
```

Figure 9 – C# Property declaration after transformation (disassembled)

5.2.4 Type and CIL transformation API

Previous sections describe how the transformation process is done, but to get these results, a quite powerful assembly transformation API was mandatory. In summary, to perform the full transformation, an API able to perform the following tasks was required:

- Find methods marked with the *Atomic* custom attribute.
- Duplicate methods to have both an atomic and non-atomic version of them.
- Recursively analyze the type dependencies of a method to identify types used inside transactions.
- Add, to a type, the offset fields and the instructions necessary to compute their value.
- Modify the CIL instructions of a method to redirect accesses to fields and arrays to the current transaction's context.
- Optionally, keep debugging information (symbols) consistent so that transformed assemblies can still be properly debugged.

The CLI defines a reflection API, *System.Reflection*, which provides the ability to examine the structure of types, create instances of types, and invoke methods on types, all based on a description of the type. Both the .NET Framework and Mono extend it with an API, *System.Reflection.Emit*, which allows emitting new types and methods including CIL instructions even at runtime. However, the reflection API does not include the possibility to reflect on CIL instructions which is a required feature for the transformation performed by CSTMF.

As this feature is also needed for many other uses, e.g., in any static analysis tool, libraries extending or replacing the CLI API quickly appeared after the first releases of the CLRs. The one that became the de facto choice is the Cecil library [14] which is part of Mono. It is a complete replacement of the *System.Reflection* and *System.Reflection.Emit* APIs supporting all features of the CLI in both reflection and emission, including CIL; and while being part of Mono, the library is fully compatible with the .NET Framework. Therefore it is the natural choice for such task performed by CSTMF.

5.3 STM algorithms

Each thread that performs transactions has one context associated with it. The contexts are lazily created (per-thread singleton), thus a thread that never enters a transaction has no context.

In the same way as DEUCE, CSTMF provides a simple API for researchers to plug in their own STM implementation. The interface contract that STM implementations must implement is shown in Figure 10. We implemented three STM algorithms for CSTMF. They are individually detailed on the next page.

```
public interface IContext
{
        void StartTransaction(int atomicBlockId);
        bool Commit();
        void Rollback();

        T Read<T>(object obj, IntPtr offset);
        void Write<T>(object obj, IntPtr offset, T value);
}
```

Figure 10 - STM implementation contract interface

Transaction reentrancy is not an issue for the contexts. As explained, the additional code that manages transactions in an atomic method is in-place of the original method code. It only calls transformed version of methods which in turn do the same. Therefore, only code currently outside of a transaction can call an atomic method that will start a new transaction. As in the example shown in Figure 11, the resulting transactional behavior is flat nesting; the inner transaction *Bar* is merged entirely with the outer one *Foo* and inherits its scope. A rollback of *Foo* will also rollback any changes made by *Bar* (even after *Bar* has returned).

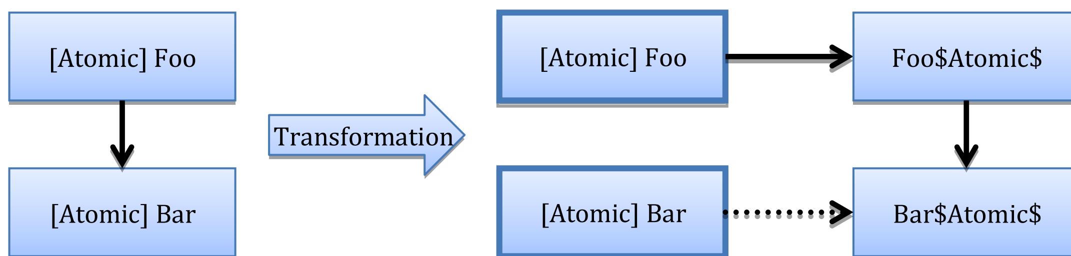


Figure 11 - Transaction reentrancy handling

5.3.1 Global Lock

The Global Lock (GL) context consists of the simpler transaction context possible. It uses a single global lock, a monitor, for all its instances and renders all transactions of an application completely mutually exclusive. The reads and writes are immediately perform on the field or array element and not logged. Therefore, this context does not support rollback.

5.3.2 Logged Global Lock

The Logged Global Lock (LGL) context is an enhanced version of the GL context. Identically to GL, it uses a single global lock, a monitor, for all its instances and renders all transactions of an application completely mutually exclusive. But unlike GL, LGL logs the accesses and only perform them at commit. As transactions are mutually exclusive, no conflict can happen during commit. But the result is that transaction can be rolled back in event of an exception occurring inside the transaction.

5.3.3 LSA

The LSA context is an implementation of the LSA algorithm [15]. LSA acquires locks as fields are written, not at commit time, and performs "incremental validation" to abort transactions that read data that has been modified after the start of the transaction by another concurrent transaction. The approach to conflict management is simply to abort the transaction which will be retried. The implementation in CSTMF is directly ported and adapted from DEUCE. We enhanced it with generics to avoid boxing and used value types where applicable to reduce allocation and GC pressure.

5.4 Unsafe access API

To compute the offsets of fields and vectors' elements as well as reading and writing through them, a small API was implemented. Indeed, this functionally is not part of the CLI or any CLR. Our API is similar in functionality to *sun.misc.Unsafe* used by DEUCE on the JVM. While taking advantage of unsafe and unverifiable CIL, it does so without resorting to undocumented features. Figure 12 shows the API as written in CIL. Figure 13 is the same API as seen from C#. Note that the API makes use of generics that fully exist at the intermediate language level and runtime in contrast to the JVM which uses type erasure. The API is succinct yet complete and does not require any boxing to compute and use the offsets for data of any type, including primitive and value types.

```
native int ComputeObjectFieldOffset<T>(object obj, !!T& f)
native int ComputeValueTypeFieldOffset<valuetype S, F>(!!S& s, !!F& f)
native int ComputeStaticFieldOffset<T>(!!T& f)

native int ComputeObjectElementOffset<class T>(int32 index)
native int ComputeValueTypeElementOffset<valuetype T>(int32 index)


!!T ReadAtObjectOffset<T>(object obj, native int offset)
!!T ReadAtValueTypeOffset<valuetype S, T>(!!S& s, native int offset)

void WriteAtObjectOffset<T>(object obj, native int offset, !!T 'value')
void WriteAtValueTypeOffset<valuetype S, T>(!!S& s, native int offset, !!T 'value')
```

Figure 12 - Unsafe access API in CIL

```
IntPtr ComputeObjectFieldOffset<T>(object obj, ref T f);
IntPtr ComputeValueTypeFieldOffset<S, F>(ref S s, ref F f) where S: struct;
IntPtr ComputeStaticFieldOffset<T>(ref T f);

IntPtr ComputeObjectElementOffset<T>(int index) where T: class;
IntPtr ComputeValueTypeElementOffset<T>(int index) where T: struct;


T ReadAtObjectOffset<T>(object obj, IntPtr offset);
T ReadAtValueTypeOffset<S, T>(ref S s, IntPtr offset) where S: struct;

void WriteAtObjectOffset<T>(object obj, IntPtr offset, T value);
void WriteAtValueTypeOffset<S, T>(ref S s, IntPtr offset, T value) where S: struct;
```

Figure 13 - Unsafe access API in C#

6 STM .NET

In July 2009 Microsoft publicly made a beta version of the .NET Framework 4.0 *STM-enabled* available. It is a non-invasive STM, but the STM algorithm cannot be replaced by another as we would like. It is an experimental release with no promise of it ever becoming part of a future release of the .NET Framework. On May 11th, the project was officially closed[5]. Its support is limited to 32-bit Windows (XP and up) and officially to the C# language. The latter is due to the missing support of CLR functionalities such as non-zero-based array and exception filters which are not exposed to C# programmers. But code written in these languages that does not use these features should work, albeit officially untested.

Even if the support is limited to C#, there is no construct added to the language to delimit transactions. Instead this is done via a *try/catch* block with a marker exception (Figure 14) or by passing an anonymous method to *Atomic.Do* (Figure 15).

```
try
{
	<transactional statements>
}
catch(AtomicMarker) {}
```

Figure 14 - Delimiting a transaction via a try/catch block in STM.NET

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

Figure 15 - Delimiting a transaction with an anonymous method in STM.NET

There are few published details about the STM implementation itself. It is known that the runtime had to be modified to support *value types* and *by reference* method parameters which manifest through managed pointer. The garbage collector was also adapted to play nice with the STM. In addition, many custom attributes are provided to allow developers to:

- Signal methods that must, must not or may run in a transaction
- Signal fields that must be accessed within a transaction
- Suppress the transaction for a method
- Redirect transacted calls to another method

[5] STM.NET DevLab Incubation Complete: http://tinyurl.com/2vkdmdf

This page is intentionally left blank

7 Results

7.1 Test environment 34
7.1.1 Hardware 34
7.1.2 Operating systems 34
7.1.3 Runtimes 34
7.2 Benchmarks 36
7.2.1 Array 36
7.2.2 Linked list 36
7.2.3 Skip list 36
7.3 Exception throwing performance 37
7.4 Overhead 39
7.5 Runtimes comparison 41
7.6 CSTMF Performance 44
7.6.1 .NET Framework 4 44
7.6.2 Mono on Linux 47

7.1 Test environment

7.1.1 Hardware

Processors: 2 x 4-core Xeon X5365 @ 3 GHz processors (8 cores/threads total)

Caches: L1: 32 KB data and 32 KB instruction caches per core

L2: 4 MB per 2 cores (8 MB per processor, 16 MB system total)

Memory: 5 GB

7.1.2 Operating systems

Windows: Windows Vista 32-bit with Service Pack 2

Linux: Ubuntu 9.10 x86-64

7.1.3 Runtimes

The tests of CSTMF will be performed on four runtimes. The two last versions of the .NET Framework (version 4 was released near the end of the development of CSTMF) and the last revision, at the time, of Mono on both Windows and Linux.

STM.NET was also tested using the only publicly available release. The version number of the runtime confirms that it is based on an early release of the .NET Framework 4.

Runtime	Version	Architecture
.NET Framework 2.0	SP2 (2.0.50727.4200)	x86
.NET Framework 4	RTM (4.0.30319.1)	x86
Mono on Windows	2.6.3	x86
Mono on Linux	2.6.3	x86-64
STM.NET 1.0	Beta 2 (4.0.20506.24)	x86

Figure 16 - Tested runtimes versions

The .NET Framework 3.0 and 3.5 are not present for the reason that these versions use the exact same runtime as the 2.0 version. They only add numerous new libraries on top of it; fixes to the runtime itself were added as service packs for it.

Tests on the .NET Framework were done using its "Server" garbage collector. This variant, which does not exists in Mono, is specifically designed for high-throughput loads. This contrasts with the default GC which is meant for applications with user interfaces. Indeed, it performs smaller collections more frequently in order to not interfere with the responsiveness of the user interface.

7.2 Benchmarks

7.2.1 Array

The array benchmark is the less complex of all three. It is a simple array only accessible through two atomic methods: one to read and the other to write an element. It is therefore a benchmark with very short transactions as only one data access is performed in each of them. With a size of 65536 elements, that allows us to characterize the STM and runtimes in a low conflict rate scenario.

7.2.2 Linked list

The linked list benchmark uses an atomic singly-linked list with a O(n) complexity. Its initial size is of 256 elements. It is the most demanding benchmark of the three for the STM algorithms. Statistically, on average, 128 accesses (half the size) are necessary to reach an element's location. In addition, conflicts are extremely frequents as a write instantly invalidate all other transactions that access elements further down the list. Reads are calls to the *Contains* method, writes to the *Add* and *Remove* methods. The benchmark calls *Remove* only after a successful *Add* in order to keep the size of the linked list constant.

7.2.3 Skip list

This benchmark uses an atomic skip list[6] which is an optimized singly-linked list with a complexity of O(log(n)) instead of O(n). This benchmark should offer results in between the two others. Two sizes are used: 256 en 65536 elements. We should expect a better scalability with more elements as this reduces the probability that two operations on the skip list perform modifications at the same location and conflict. As with the linked list, reads are calls to the *Contains* method, writes to the *Add* and *Remove* methods. The benchmark calls *Remove* only after a successful *Add* in order to keep the size of the skip list constant.

[6] Skip list data structure: http://en.wikipedia.org/wiki/Skip_list

7.3 Exception throwing performance

The purpose of this benchmark is to exhibit the differences between the CLR implementations concerning exception throwing performance. It shows the impact of caching exceptions – that is instancing a single exception object and throwing it for each exception of this type – and multi-threaded exception throwing.

The results, presented in Figure 17, unveil interesting findings which indicate that both CLRs are implemented quite differently for this part. But first, a word is required about the result of Mono with non-cached exceptions and 8 threads. With these settings and due to a bug in the version of Mono used at the time, the benchmark crashes almost at every runs on Linux. The result shown is for a (rare) successful run. Under Windows the crash is not happening, however the result is much lower than expected. So these results are to be taken lightly. Speed-up computations affected are also colored.

Throwing exceptions is much faster on Mono than on the .NET Framework but scales better on the later; however, this is not sufficient to catch up with Mono at 8 threads. The Framework .NET 4 shows a slight improvement over the previous version. Caching an exception object under both versions of the .NET Framework has little impact; it only improves the throughput by 8-10%. However, under Mono, this improvement is of at least of 62%. It can go as high as an impressive 840%, but this is also due to the odd non-cached 8 threads result on Windows. Nonetheless, Mono seems to be optimized to allocate more information at exception object creation rather than when throwing the exception and benefits more than .NET from reusing exception objects. This also may be magnified by its currently less efficient garbage collector.

Runtime	Non-cached		Cached	
	1 thread	8 threads	1 thread	8 threads
.NET 2	**1.0**	6.0	1.1	6.5
.NET 4	1.2	6.5	1.3	7.1
Mono Windows	4.7	*1.9*	7.6	17.7
Mono Linux	5.9	*6.8*	13.5	26.9

Values relative to the .NET 2/non-cached baseline with a result of 58k exceptions/s

Runtime	Caching speed-up		8 threads speed-up	
	1 thread	8 threads	Non-Cached	Cached
.NET 2	8%	9%	497%	506%
.NET 4	8%	10%	456%	466%
Mono Windows	62%	*840%*	*-60%*	134%
Mono Linux	131%	*293%*	*17%*	98%

Figure 17 - Exception throwing performance on various CLR implementations

7.4 Overhead

The objective of this test is to quantify the minimal overhead for a transaction context. The overhead comes from multiple sources:

- Transaction handling code added to each atomic method.
- Redirection of all reads and writes to calls of transaction context.
- Computation of final offset for array elements.
- Performing all reads and writes through unsafe access helper using object and offset.
- Work performed by the context itself. Inexistent in the case of the *Global Lock* context which do not track any information about accesses inside a transaction (see *Global Lock* chapter for details).

The computation of the fields' offset is a one-time per-type cost rather than a per-instance cost. Therefore, it does not cause any overhead on the usage of a transaction context.

To quantify this overhead, the offline transformer was adapted to output a simpler variant of the atomic methods transformation. *Figure 18* and *Figure 19* show the transformation of an atomic method using this variant.

```
[Atomic]
private int AtomicMethod()
{
    return foo();
}
```

Figure 18 - Atomic method before simple lock transformation

```
[Atomic,Transformed]
private int AtomicMethod()
{
    lock(Utility.GlobalSyncRoot)
    {
        return foo();
    }
}
```

Figure 19 - Atomic method after simple lock transformation

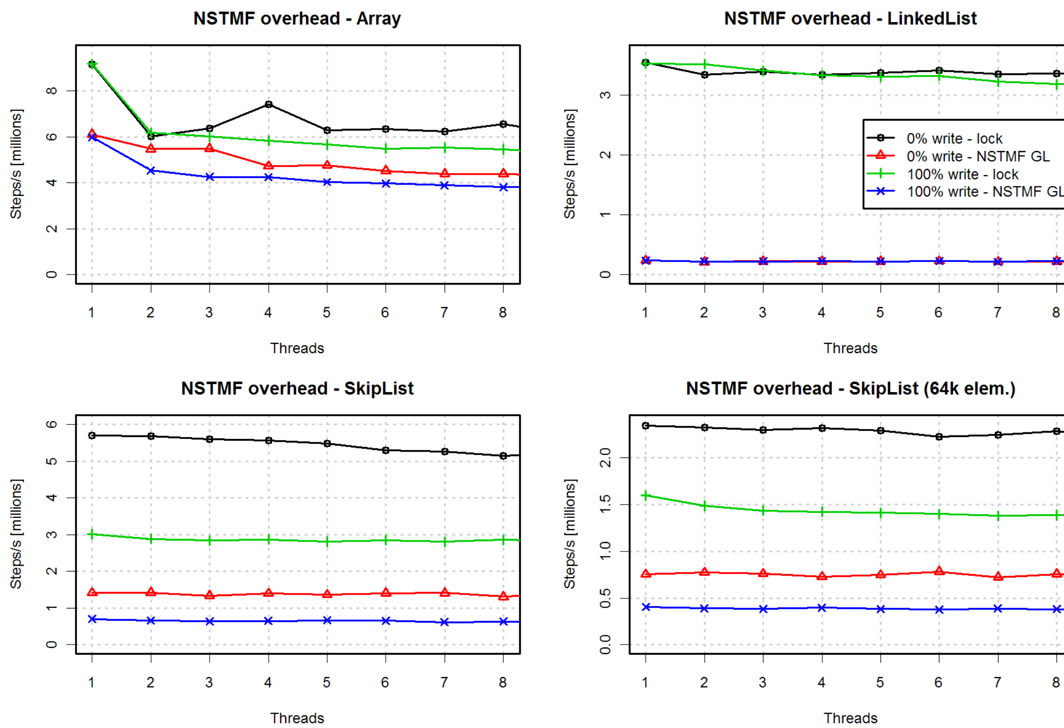


Figure 20 - Overhead of GL context on the four benchmarks

In the array benchmark, CSTMF's Global Lock context has a 25% reduction of performance which is not much considering the additional work compared to a simple array access. This is a good appreciation of the minimum overhead.

In the linked list benchmark which needs to perform much more reads to find the correct emplacements, the lock transformation completely distances CSTMF with a factor of about 14. Combined with the array's results, this shows that the real overhead comes more from redirection of reads and writes rather than transaction management. Results with 0% and 100% writes are analogous; this is due to the fact that the cost of going through the elements of the linked list far outweighs the insertion or removal cost.

The skip list benchmarks shows results in between the two others. It is consistent with the fact that a skip list is optimized for performing fewer accesses than a linked list to get to a node emplacement while being more complex than a simple atomic array. The benchmark also shows that adding or removing a node in a skip list is almost twice slower than just looking for an existing one. This contrasts with linked list that has similar performance for both types of operations. Usage of CSTMF causes a drop of performance by a factor of about 4 with 256 elements and 3 with 65536.

7.5 Runtimes comparison

The runtime comparison test shows CSTMF's transaction throughput on the four previously discussed CLRs using our three benchmarks. In addition, we present the results of STM.NET on the array and linked list benchmarks. No results for STM.NET are available for the two skip list benchmarks because a bug in the STM's implementation makes it always crash even with no concurrency.

The test shows clear results as presented in the figures on the next pages. The last version of the .NET Framework undoubtedly beats all the other tested CLRs when running CSTMF. It even shows consistently better results than STM.NET with the exception of the linked list benchmark with no writes. Great optimization work has been performed by Microsoft on the .NET Framework 4 seeing that it manages to outperform the previous version by more than 50% on almost all benchmarks.

As we expected, CSTMF scales better when increasing the size of the skip list from 256 to 65536 elements; having a higher number of elements reduces the conflict probability. With this bigger size, CSTMF keeps a good scalability even at a write rate of 50%.

However, the results of Mono on both operating systems are less promising. But at least the throughput never scales back with increase of concurrency.

STM.NET seems to have a lot of troubles coping with conflicts. It even has the exact inverted scalability of CSTMF running on .NET 4 on the linked list benchmark with 50% of write.

The dashed line on the graphs represents the best result at one thread when scaled perfectly. It gives a good overview of the general scalability of the STMs for each benchmark.

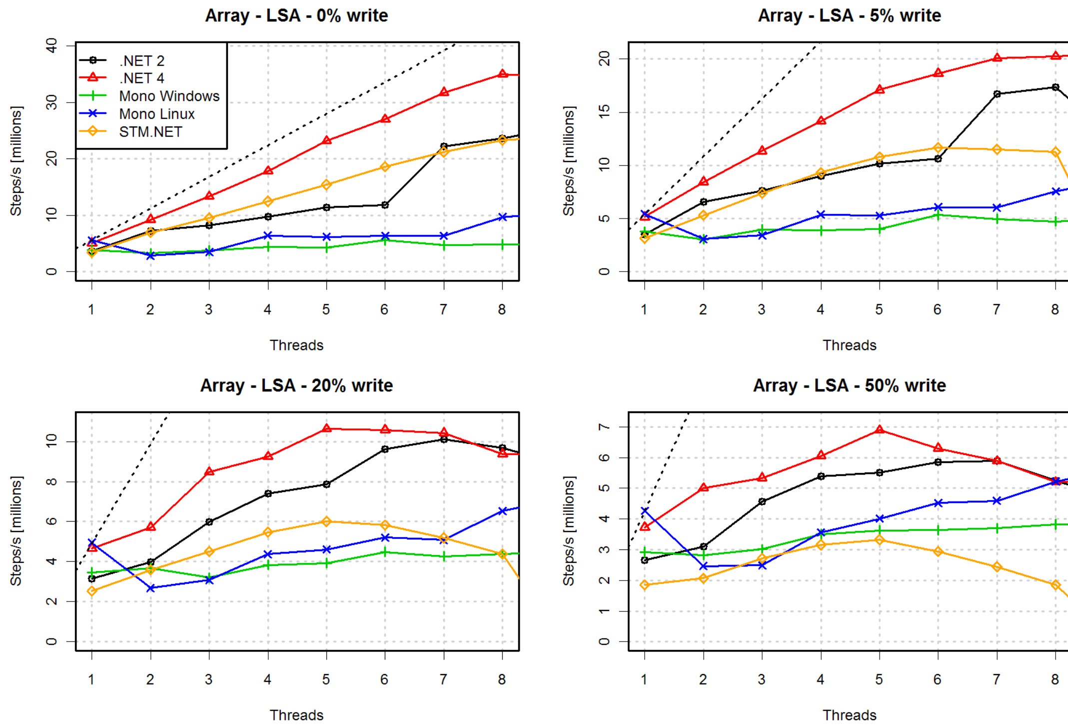


Figure 21 - The array benchmark

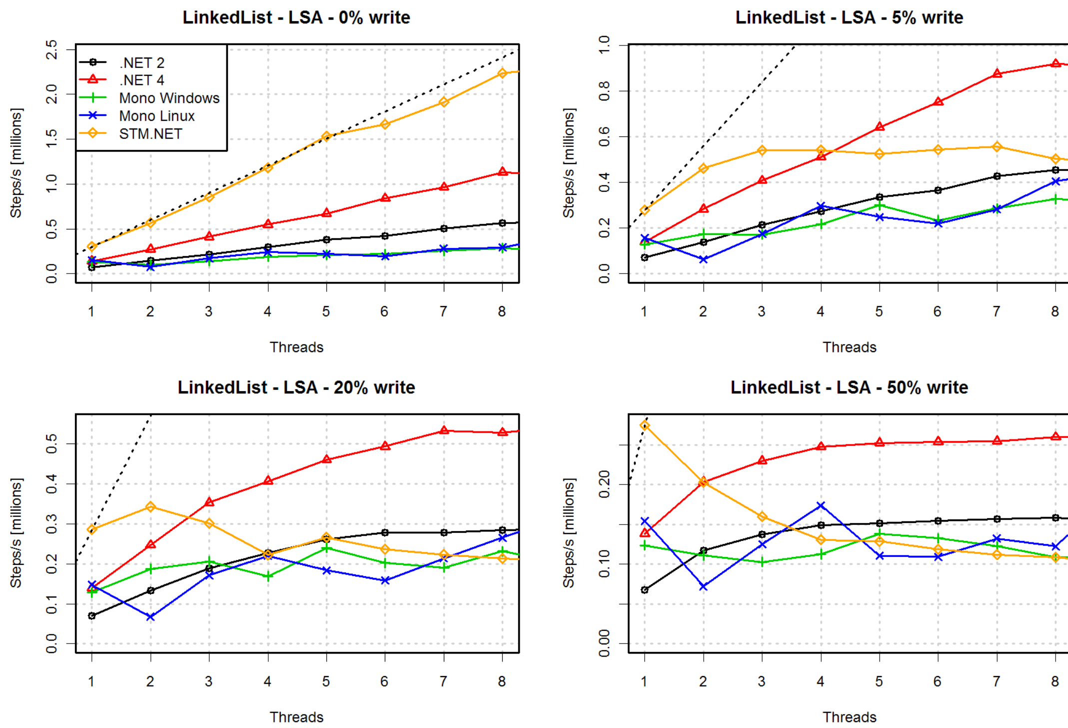
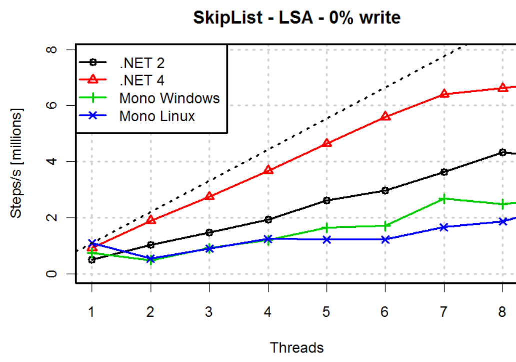
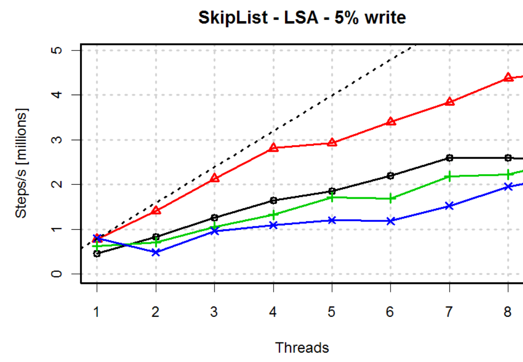
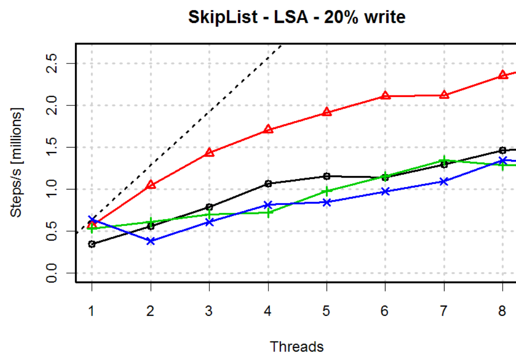


Figure 22 - The linked list benchmark (256 elements)







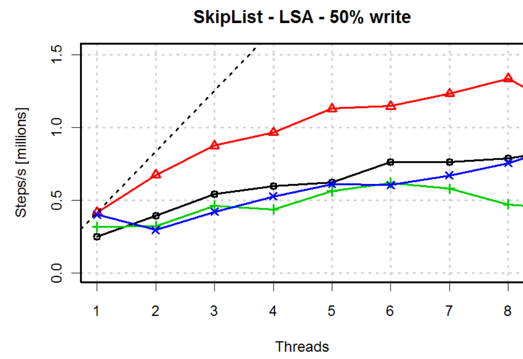
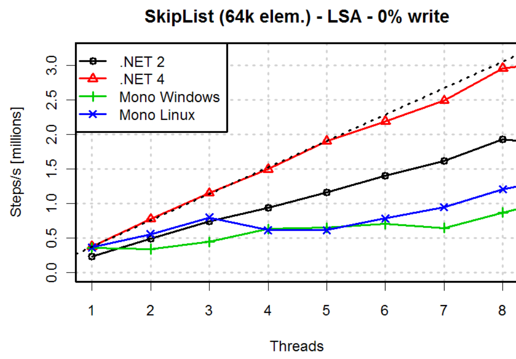
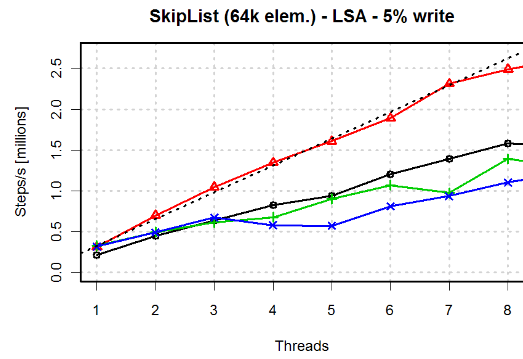


Figure 23 - The skip list benchmark (256 elements)





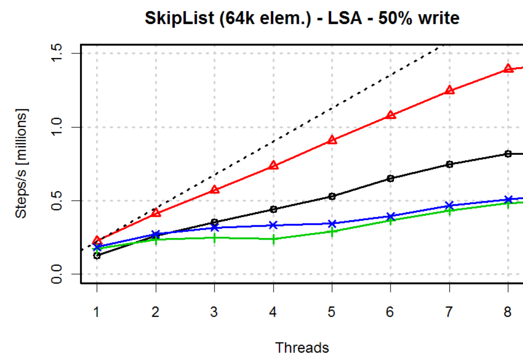


Figure 24 - The skip list benchmark (64k elements)

7.6 CSTMF Performance

The following benchmarks compare CSTMF's LSA context against the lock transformation, which uses a monitor, and custom variants of the data structures using the single-writer multiple-readers lock of the CLR (RWL)[7]. The comparisons were made on the .NET Framework 4 on Windows and Mono on Linux (the runtime comparison test shows that Mono's results are similar on Windows and Linux).

7.6.1 .NET Framework 4

In the array benchmark, the very good scalability of LSA with few accesses per transaction gives CSTMF a huge advantage over both locks at 0% and 5% of write. The advantage is reduced at 20% as scalability fades and further at 50%. The overhead of write operations is too high at 80% and 100% and the LSA context ends up below both locks.

The linked list benchmark with its very high count of accesses per transaction gives an unquestionable lead to the lock. The overhead seen previously and poor scalability of the linked list kill LSA's performances.

In the skip list benchmarks, while the LSA context starts way behind with a low concurrency due to its overhead, the good scalability with the skip list allows it to catch up. The bigger skip list with 65536 elements additionally improves the scalability and the context catches up even at 20% of writes. It is safe to say that with this scalability, CSTMF using LSA could surpass the lock at any write rate on a machine with more cores.

Overall, the RWL is very disappointing at it never surpass the monitor, even at a write rate of 0% where accesses should be concurrent. This is certainly due to the fact that the RWL is heavily optimized for more consequential work done inside the lock. In our algorithm, the time passed with the RWL acquired would be too short for the RWL to compensate its overhead.

---

[7] The *System.Threading.ReaderWriterLockSlim* class.

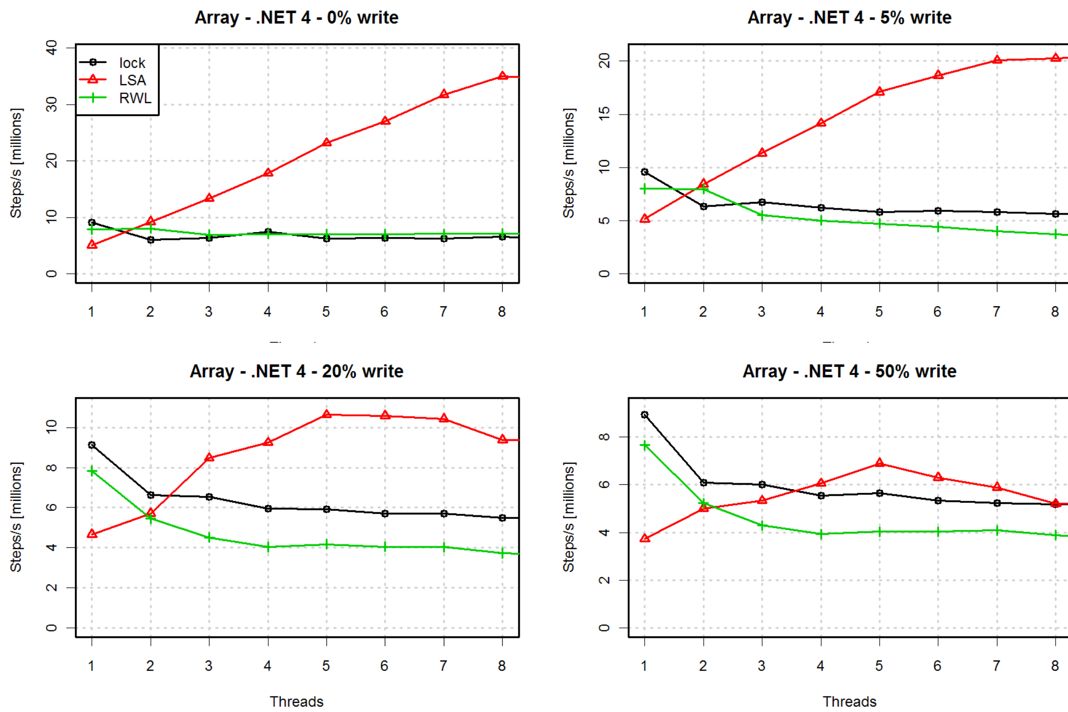


Figure 25 - The array benchmark on .NET 4

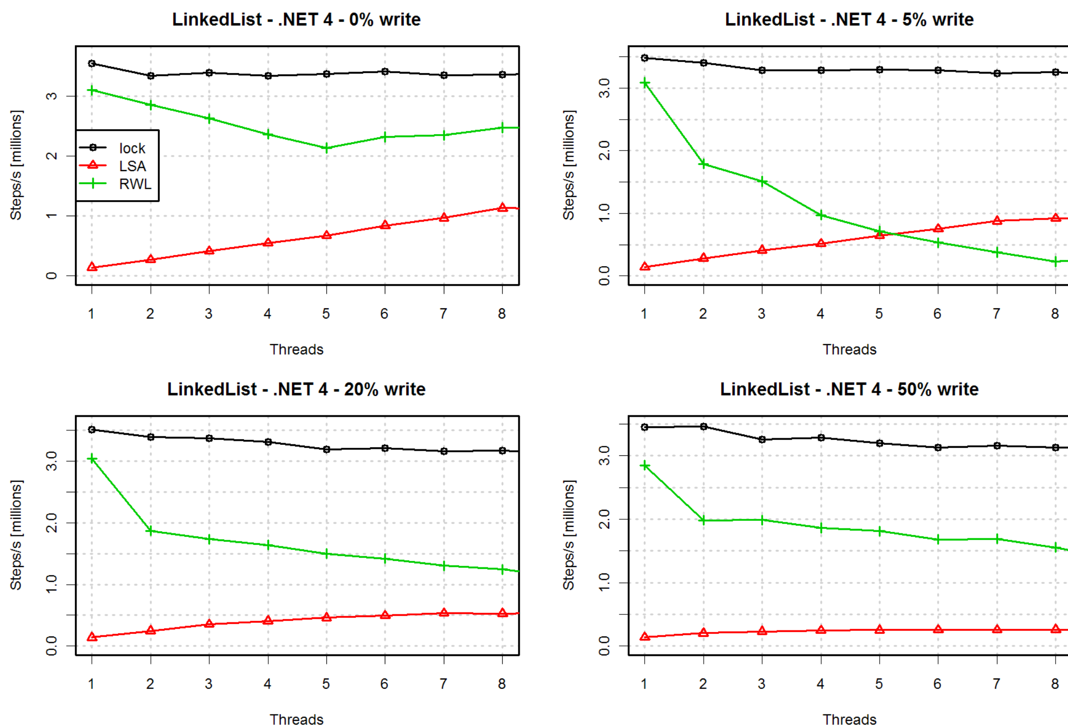


Figure 26 - The linked list benchmark on .NET 4 (256 elements)

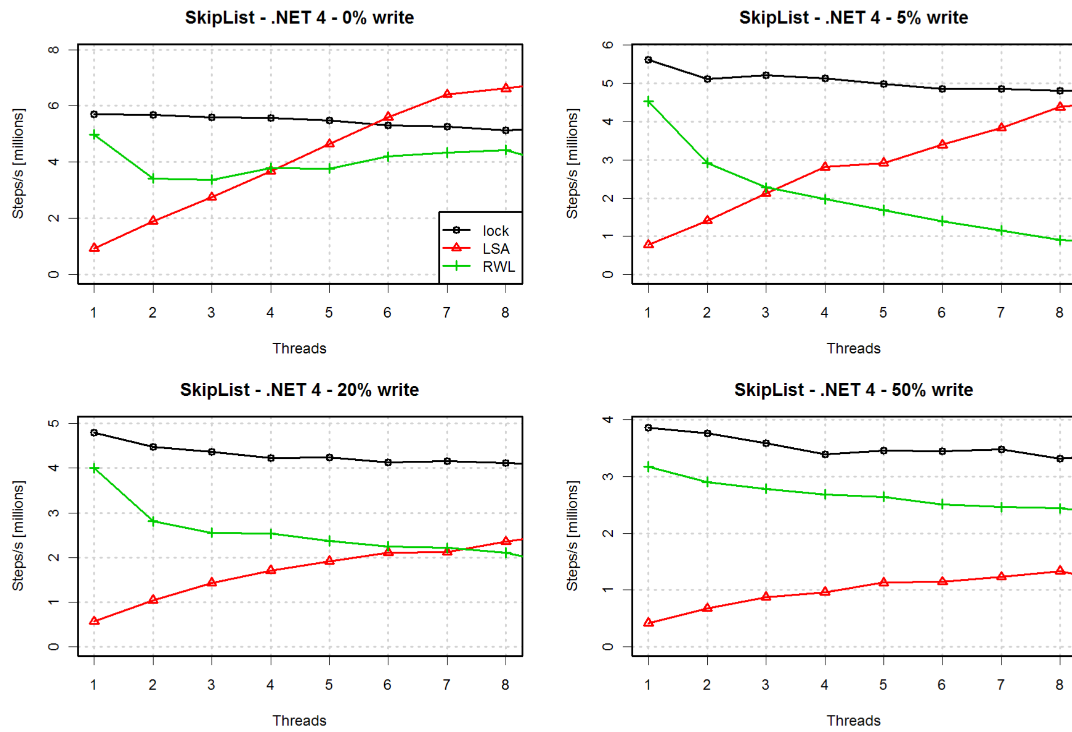


Figure 27 - The skip list benchmark .NET 4 (256 elements)

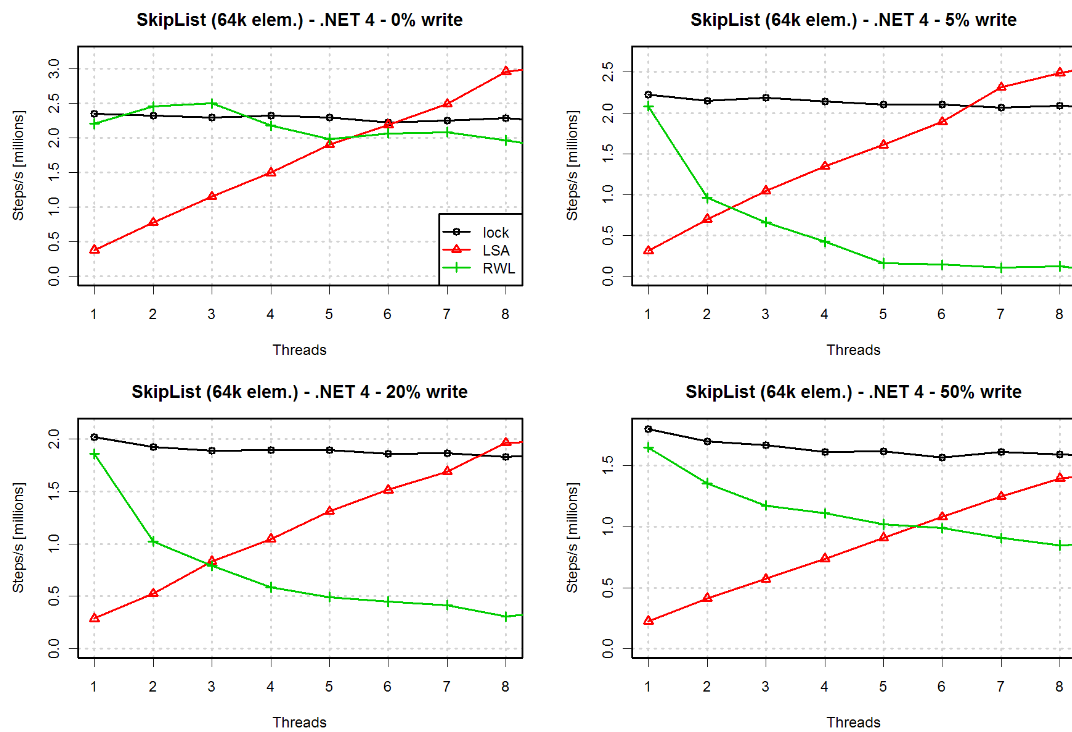


Figure 28 - The skip list benchmark .NET 4 (64k elements)

7.6.2 Mono on Linux

When looking at the following results of CSTMF on Mono and comparing them with those of the .NET Framework, it could seem that CSTMF runs a lot better on Mono that on Microsoft's CLR. Yet it was clear from the runtime comparisons that this is not true. The illusion comes from the fact that the monitor's implementation of Mono has much poorer scalability. While the lock transformation stays almost flat on the .NET Framework with the increase of the thread count, there is a huge drop between one and two threads on Mono. Therefore, CSTMF, which does not make use of monitors, gains a substantial advantage as it does not have to compensate the initial gap due to its overhead.

CSTMF is almost at the same level as the lock in the linked list benchmark and has a lead in all others benchmarks at all rates. The linked list and skip list (especially with 65536 elements) seem to keep locks a sufficient amount of time so that the RWL variant of these data structures can distance themselves from the lock variants at a write rate of 0%. This shows that the implementations of the RWL are quite different between Mono and the .NET Framework. However, as soon as there are writes involved, the RWL behaves similarly to an exclusive lock such as a monitor and its performance becomes analogous with a slight overhead.

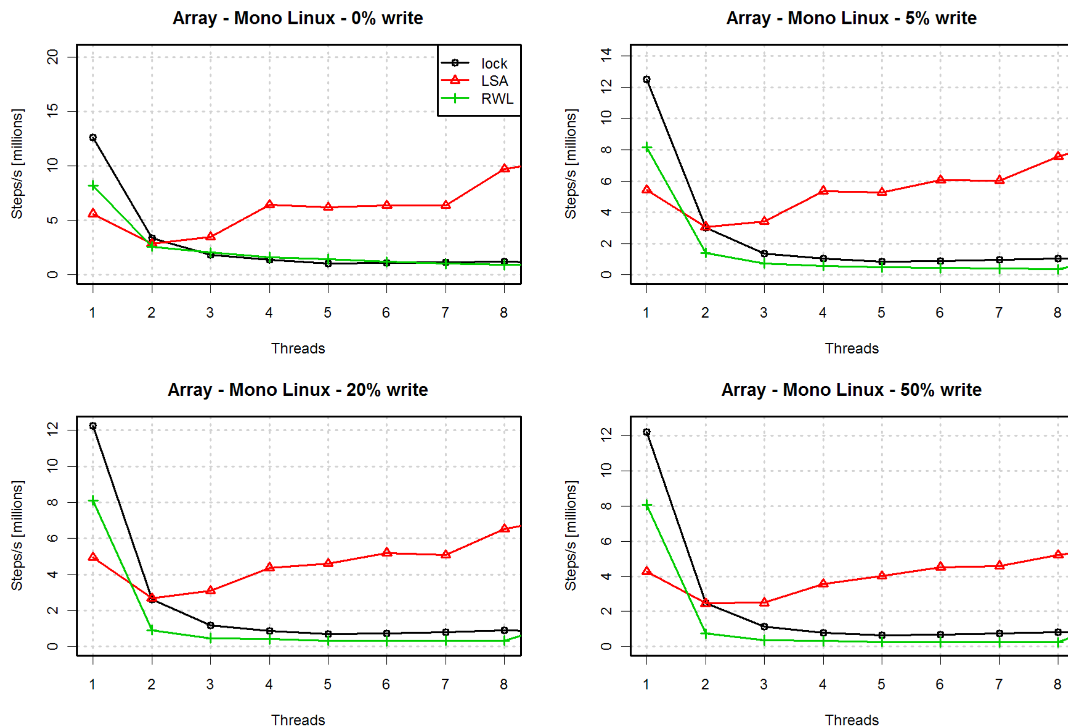


Figure 29 - The array benchmark on Mono Linux

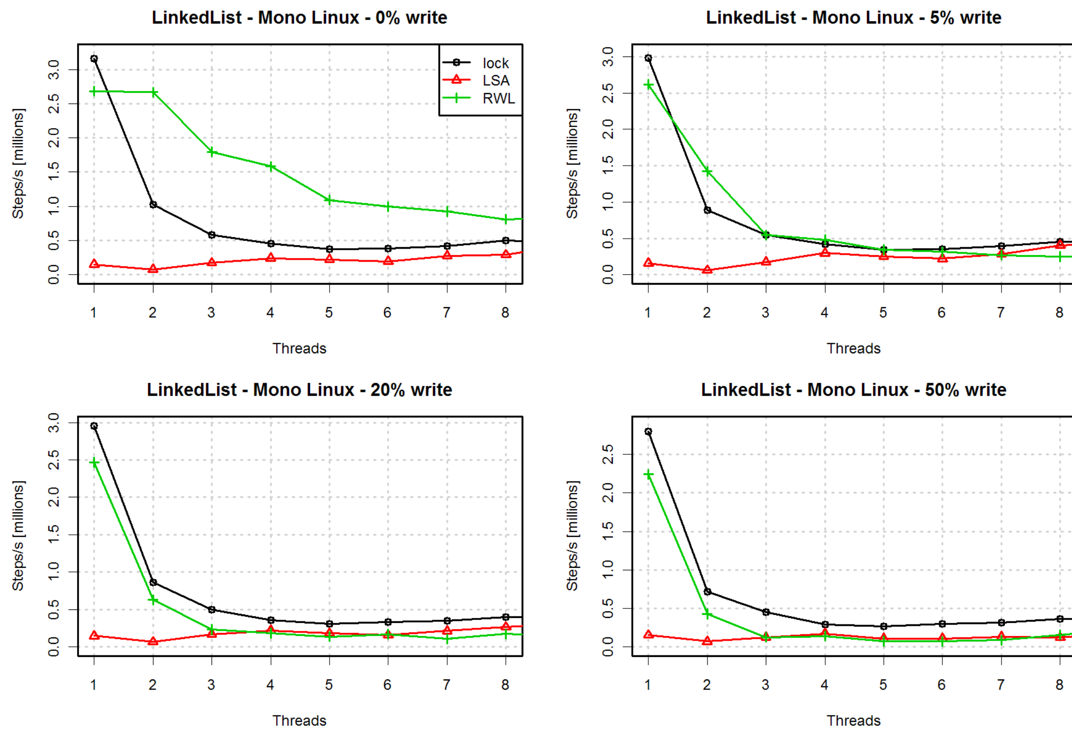


Figure 30 - The linked list benchmark on Mono Linux (256 elements)

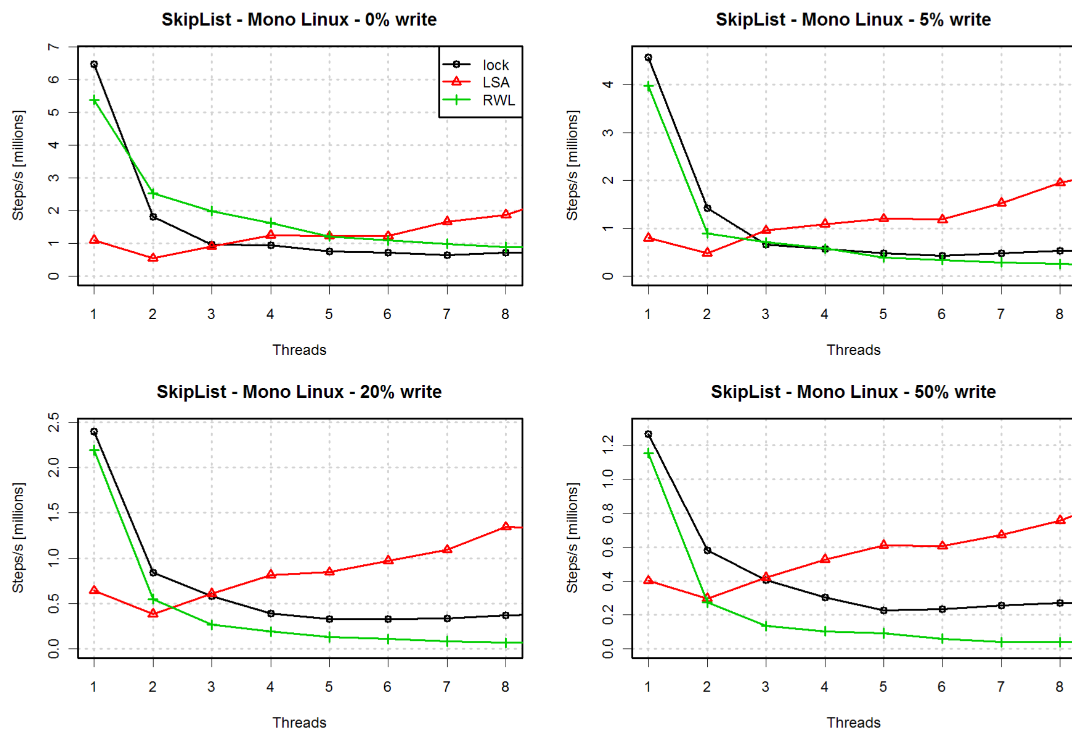


Figure 31 - The skip list benchmark on Mono Linux (256 elements)

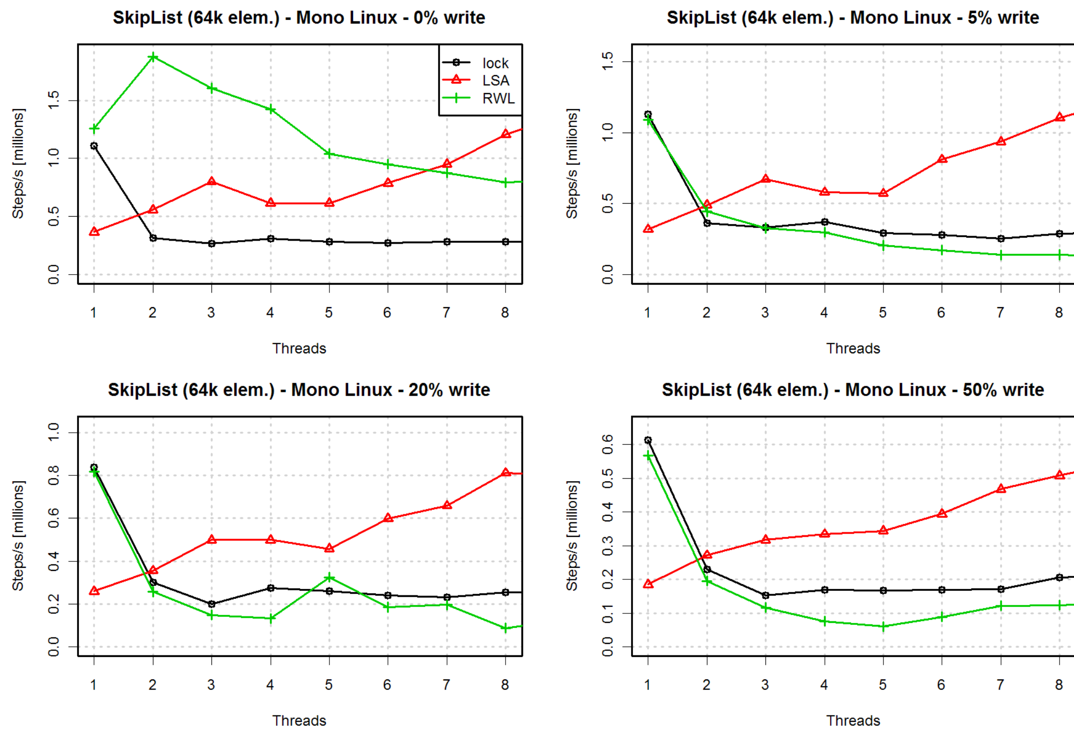


Figure 32 - The skip list benchmark on Mono Linux (64k elements)

8 Conclusion

We presented the Common Language Infrastructure from an STM perspective with its good and bad aspects. While it defines a powerful runtime environment with great support for a wide variety of languages, it is extremely difficult to implement an STM framework that covers all its features. Specifically, the managed pointers represent a considerable obstacle in the full coverage of the CLI without a custom CLR. Without them, support for features such as value types and arguments passed by reference is unachievable. However, while having only partial support of all the features of the CLI, CSTMF showed that is sufficient to implement relatively complex algorithms such as a skip list. In fact, all the features present in both the JVM and the CLI, and covered by DEUCE, are supported by CSTMF.

CSTMF uses a simple mechanism based on custom attributes to specify atomic methods. No other modification to the methods is required by developers to make them atomic, besides where previously mentioned limitations apply. Therefore, almost all programming languages with a compiler for the CLI can mark methods as atomic and use CSTMF; the rare exceptions being languages without custom attributes support or that rely too much on managed pointers. Thus, CSTMF clearly achieves the objective to provide a non-invasive multi-language STM framework.

CSTMF performance shows that an efficient STM can be implemented on top of the CLR. However, not all CLRs are on the same level. Indeed we saw that remarkable performance work has been done in the version 4 of .NET Framework which distances both the previous version of Microsoft's CLR and Mono. The latter shows results similar between its Windows and Linux versions, well in retreat compared to the .NET Framework 4. However, CSTMF performs a lot better than naïve lock-based implementations of the data structures on Mono due to the rather poor performance of the monitors in this CLR.

As the scalability of CSTMF is very promising when performing the tests on a machine with a relatively low number of cores, especially with the .NET Framework 4; tests on machines with a much higher number of cores could be interesting.

In conclusion, while clearly still much work would be needed for CSTMF to be ready for usage outside research, we showed that it is possible to implement an efficient non-invasive STM on top of the CLR. However, it cannot cover all the features of the CLR without customizing it.

9 Table of Figures

Figure 1 - The Common Language Infrastructure components 8
Figure 2 – Usage of primitive types 10
Figure 3 - CLS relation to languages and CLI/CTS 13
Figure 4 - Marking a method as atomic in different CLI languages 20
Figure 5 - Atomic method before transformation 23
Figure 6 - Atomic method after transformation 24
Figure 7 - Types of data accesses and corresponding argument to STM context 25
Figure 8 – C# Property declaration before transformation 26
Figure 9 – C# Property declaration after transformation (disassembled) 26
Figure 10 - STM implementation contract interface 28
Figure 11 - Transaction reentrency handling 28
Figure 12 - Unsafe access API in CIL 30
Figure 13 - Unsafe access API in C# 30
Figure 14 - Delimiting a transaction via a try/catch block in STM.NET 31
Figure 15 - Delimiting a transaction with an anonymous method in STM.NET 31
Figure 16 - Tested runtimes versions 34
Figure 17 - Exception throwing performance on various CLR implementations 38
Figure 18 - Atomic method before simple lock transformation 39
Figure 19 - Atomic method after simple lock transformation 39
Figure 20 - Overhead of GL context on the four benchmarks 40
Figure 21 - The array benchmark 42
Figure 22 - The linked list benchmark (256 elements) 42
Figure 23 - The skip list benchmark (256 elements) 43
Figure 24 - The skip list benchmark (64k elements) 43
Figure 25 - The array benchmark on .NET 4 45
Figure 26 - The linked list benchmark on .NET 4 (256 elements) 45
Figure 27 - The skip list benchmark .NET 4 (256 elements) 46
Figure 28 - The skip list benchmark .NET 4 (64k elements) 46

Figure 29 - The array benchmark on Mono Linux 47
Figure 30 - The linked list benchmark on Mono Linux (256 elements) 48
Figure 31 - The skip list benchmark on Mono Linux (256 elements) 48
Figure 32 - The skip list benchmark on Mono Linux (64k elements) 49

10 References

[1]. **Korland, Guy, Shavit, Nir and Felber, Pascal.** *Noninvasive concurrency with Java STM.* 2009.

[2]. *Programming Language Popularity.* [Online] [Cited: August 16, 2010.] http://www.langpop.com/.

[3]. *Mono.* [Online] Novell. http://www.mono-project.com/.

[4]. *Microsoft's Empty Promise.* [Online] Free Software Foundation, July 16, 2009. [Cited: August 16, 2010.] http://www.fsf.org/news/2009-07-mscp-mono.

[5]. *Mono Position Statement.* [Online] Ubuntu Devel Mailing List, June 30, 2009. [Cited: August 16, 2010.] https://lists.ubuntu.com/archives/ubuntu-devel-announce/2009-June/000584.html.

[6]. *List of CLI languages.* [Online] Wikipedia. [Cited: August 16, 2010.] http://en.wikipedia.org/wiki/List_of_CLI_languages.

[7]. **Pascal Felber, Christof Fetzer and Torvald Riegel.** *Dynamic Performance Tuning of Word-Based Software Transactional Memory.* Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). 2008.

[8]. **Maurice Herlihy, Victor Luchangco and Mark Moir.** *A flexible framework for implementing Software Transactional Memory.* Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). 2006.

[9]. **Maurice Herlihy.** *SXM: C# Software Transactional Memory.* [Online] February 17, 2005. [Cited: August 16, 2010.] http://research.microsoft.com/en-us/downloads/6cfc842d-1c16-4739-afaf-edb35f544384/default.aspx.

[10]. **Ralf Westphal.** *NSTM - .NET Software Transactional Memory.* [Online] August 5, 2007. [Cited: August 16, 2010.] http://netstm.codeplex.com/.

[11]. *STM.NET.* [Online] Microsoft, July 28, 2009. [Cited: August 16, 2010.] http://blogs.msdn.com/b/stmteam/archive/2009/07/28/stm-net-released.aspx.

[12]. **Erik Meijer and Jim Miller.** *Technical Overview of the Common Language (or why the JVM is not my favorite execution environment).* 2001.

[13]. **Serge Lidin.** *Inside Microsoft .NET IL Assembler.* s.l. : Microsoft Press, 2001. 0-7356-1547-0.

[14]. **Jb Evain.** *Cecil library.* [Online] Mono, Novell. [Cited: August 16, 2010.] http://www.mono-project.com/Cecil.

[15]. **T. Riegel, P. Felber and C. Fetzer.** *A lazy snapshot algorithm with eager validation.* Proceedings of the International Symposium on Distributed Computing (DISC). 2006.

[16]. **Joe Duffy.** *Concurrent Programming on Windows.* s.l. : Addison Wesley, 2008. 978-0-321-43482-1.

[17]. **Jeffrey Richter.** *CLR via C#, Second Edition.* s.l. : Microsoft Press, 2005. 978-0-7356-2163-3.