STM for games

Research Report

Author: Arkadiusz Bielecki
ID: C00139358

Date: 12.11.2013

Project Supervisor: Joseph Kehoe

Table of Contents

1. Introduction

2. Concurrency

  The Benefits of Parallel Programming

  The Benefits of Distributed Programming

  Amdahl's law

  Concurrent interaction and communication

3. Software Transactional Memory

  Overheads

  Semantics

4. Development Tools

  RSTM Library

    RSTM Algorithms

    Building RSTM

    Writing apps in RSTM

  SFML

  Microsoft Visual Studio

1. Introduction

In the last years we have seen many big improvements in the digital scene. Lots of new discoveries were accomplished, mainly because of the computer hardware which is constantly improving in an astonishing rate. According to Moore's law the number of transistors on integrated circuits doubles approximately every two years*[1]*. That happens at exponential rates. His predictions were really accurate and many of the digital electronic devices are strongly linked to Moore's law, like:

- Processor speed
- Memory capacity
- Sensors
- Number and size of pictures in digital cameras

It looks like the doubling can go on forever, but even Moore himself said that it's not possible.

The main area I'm interested in is the Processor speed. First CPU used in commercial PC was an Intel 8080 chip running at a clock rate of 2 MHz in year 1975*[2]*. Since then the speeds of CPU's were constantly improving, as there was more competition between companies and new technologies were discovered. Current highest clock rate is about 5 GHz, and was introduced in June 2013. Thats an increase of 2,500 times !

On the other hand, 1 GHz CPU was introduced in year 2000, so we can see that since then the speed didn't improve as much as before. One of the reasons is that manufacturers are trying to improve existing components, by lowering energy consumption, adding extra capabilities etc.

There's also other reasons why CPU speed constantly go up. Two main reasons are as follows:

- The amount of transistors on the chip can't increase forever, because the closer they get to each other - the bigger the heat that can cause a damage.
- As transistors shrink, the wires that connect them come close to a quantum scale. The problem with that is that the voltage can run very low and escape.

So CPU manufacturers started to look for different ways to improve their products than just speeding the frequency. One of the main improvements was introducing more CPU cores on a single chip. The main idea behind having a multi-core processor is that separate cores can work concurrently (at the same time) on processing different chunks of data.

Software industry had to improve, as in order for software to run concurrently, it must be designed to work that way. There are a few methods to do it but one that I'm interested in is Software Transactional Memory. STM is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing*[3]*. A transaction occurs when a piece of code runs a series of reads and write to the shared memory. STM libraries can be implemented in many programming languages.

Currently we can see 4 or even 8-core processors, but does more cores and threads necessary mean a better thing ? In my project I will compare and benchmark single algorithms to parallel ones and try to come to a conclusion which one is better.

2. Concurrency

When people hear the word ***concurrency*** they often think of ***parallelism***, a related but quite distinct concept. **Concurrency is not parallelism !**

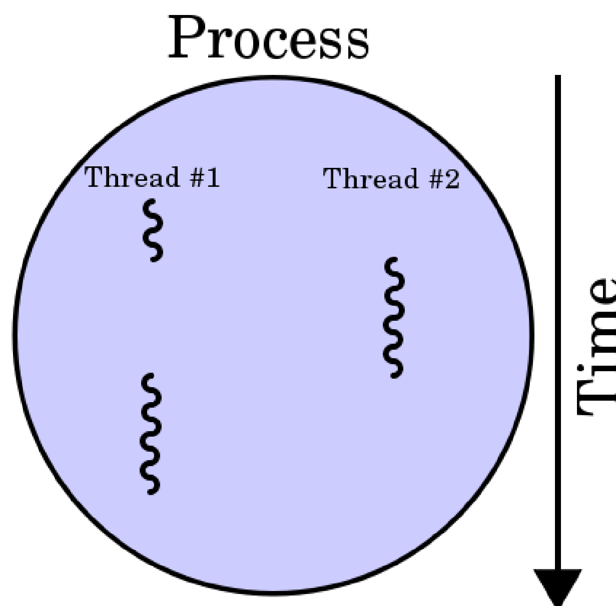
In programming, **concurrency** is the **composition** of independently executing processes, while **parallelism** is the simultaneous **execution** of (possibly related) computations. Concurrency is about **dealing with** lots of things at once. Parallelism is about **doing** lots of things at once.

Two events are said to be concurrent if they occur within the same time interval. Two or more tasks executing over the same time interval are said to execute concurrently. Two programs performing some task within the same hour continuously make progress on the task during that hour, although they may or may not be executing at the same exact instant. We say that the two programs are executing concurrently for that hour. Tasks that exist at the same time and perform in the same time period are concurrent*[4]*. Concurrent tasks can execute in a single or multi-processing environment. In a single processing environment, concurrent tasks exist at the same time and execute within the same time period by context switching. In a multiprocessor environment, if enough processors are free, concurrent tasks may execute at the same instant over the same time period. The determining factor for what makes an acceptable time period for concurrency is relative to the application*[5]*.

Concurrency techniques are used to allow a computer program to do more work over the same time period or time interval. Rather than designing the program to do one task at a time, the program is broken down in such a way that some of the tasks can be executed concurrently. In some situations, doing more work over the same time period is not the goal. Rather, simplifying the programming solution is the goal. Sometimes it makes more sense to think of the solution to the problem as a set of concurrently executed tasks. Sometimes concurrency is used to make software faster or get done with its work sooner. Sometimes concurrency is used to make software do more work over the same interval where speed is secondary to capacity. For instance, some web sites want customers to stay logged on as long as possible. So it’s not how fast they can get the customers on and off of the site that is the concern - it’s how many customers the site can support concurrently. So the goal of the software design is to handle as many connections as possible for as long a time period as possible. Finally, concurrency can be used to make the software simpler. Often, one long, complicated sequence of operations can be implemented easier as a series of small, concurrently executing operations. Whether concurrency is used to make the software faster, handle larger loads, or simplify the programming solution, the main object is software improvement using concurrency to make the software better.

Parallel programming and distributed programming are two basic approaches for achieving concurrency with a piece of software. They are two different programming paradigms that sometimes intersect. **Parallel programming techniques** assign the work a program has to do to two or more processors within a single physical or a single virtual computer. **Distributed programming techniques** assign the work a program has to do to two or more processes - where the processes may or may not exist on the same computer. That is, the parts of a distributed program often run on different computers connected by a network or at least in different processes. A program that contains parallelism executes on the same physical or virtual computer. The parallelism within a program may be divided into processes or threads*[6]*.

Fig. 1. A process with two threads of execution on a single processor [7]



The Benefits of Parallel Programming

Programs that are properly designed to take advantage of parallelism can execute faster than their sequential counterparts, which is a market advantage. Faster is not necessarily better, but it is in this case. The solutions to certain problems are represented more naturally as a collection of simultaneously executing tasks. This is especially the case in many areas of scientific, mathematical, and artificial intelligence programming. This means that parallel programming techniques can save the software developer work in some situations by allowing the developer to directly implement data structures, algorithms, and heuristics developed by researchers. Specialized hardware can be exploited. For instance, in high-end multimedia programs the logic can be distributed to specialized processors for

increased performance, such as specialized graphics chips, digital sound processors, and specialized math processors. These processors can usually be accessed simultaneously. Parallel programming techniques open the door to certain software architectures that are specifically designed for parallel environments, there are certain architectures designed specifically for a parallel processor environment*[8]*.

The Benefits of Distributed Programming

Distributed programming techniques allow software to take advantage of resources located on the Internet, on corporate and organization intranets, and on networks. Distributed programming usually involves network programming in one form or another. That is, a program on one computer on a network needs some hardware or software resource that belongs to another computer either on the same network or on some remote network. Distributed programming is all about one program talking to another program over some kind of network connection, which may involve everything from modems to satellites. The distinguishing feature of distributed programs is they are broken into parts. Those parts are usually implemented as separate programs. Those programs typically execute on separate computers and the program's parts communicate with each other over a network. Distributed programming techniques provide access to resources that may be geographically distant. For example, a distributed program divided into a Web server component and a Web client component can execute on two separate computers. The Web server component can be located in Africa and the Web client component can be located in Japan. Distributed computing can be used for redundancy. If we divide the program up into a number of parts with each running on different computers, then we may assign some of the parts the same task. If one of the computers fails for some reason then another part of the same program executing on a different computer picks up the work. Databases can be used to hold billions, trillions, even quadrillions of pieces of information. It is simply not practical for every user to have a copy of the database. The problem is some users are located in different buildings than where the computer with the database is located. Some users are located in different cities, states, and in some instances, countries. Distributed programming techniques are used to allow users to share the massive database regardless of where they are located*[9]*.

Parallel and distributed programming come with a cost. Although there are many benefits to writing parallel and distributed programming, there are also some challenges and prerequisites. The most popular problems are:

- Decomposition
- Communication
- Synchronization

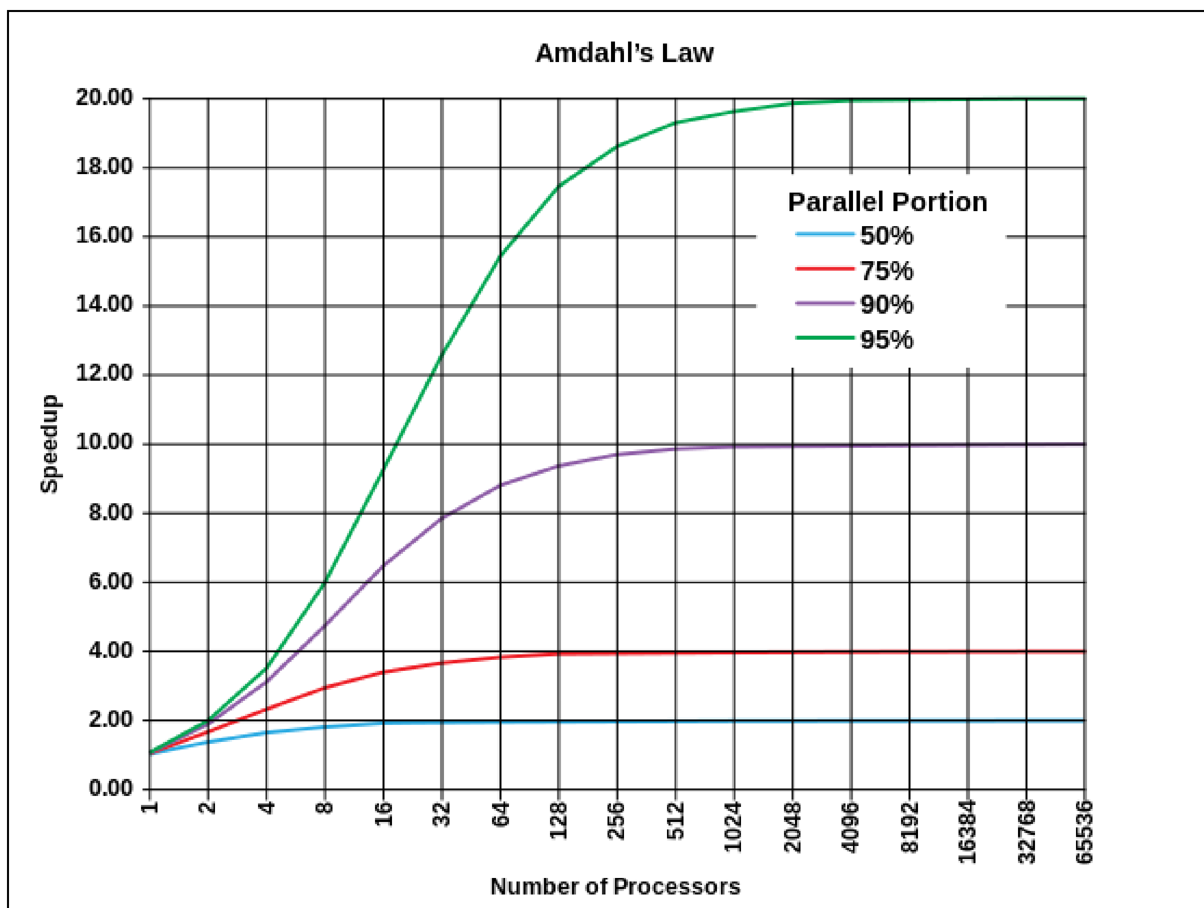
Amdahl's law

Amdahl's law is used to find the maximum expected improvement to an overall system when only part of the system is improved. It is often used in parallel computing to

predict the theoretical maximum speedup using multiple processors. The law is named after computer architect Gene Amdahl, and was presented in 1967.

The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. For example, if a program needs 20 hours using a single processor core, and a particular portion of the program which takes one hour to execute cannot be parallelized, while the remaining 19 hours (95%) of execution time can be parallelized, then regardless of how many processors are devoted to a parallelized execution of this program, the minimum execution time cannot be less than that critical one hour. Hence the speedup is limited up to 20×, as we can see below. *[10]*

Amdahl's law predictions diagram [10]



Concurrent interaction and communication

In some concurrent computing systems, communication between the concurrent components is hidden from the programmer, while in others it must be handled explicitly. Explicit communication can be divided into two classes:

Shared memory communication

Concurrent components communicate by altering the contents of shared memory locations (exemplified by Java and C#). This style of concurrent programming usually requires the application of some form of locking (e.g., mutexes, semaphores, or monitors) to coordinate between threads.

Message passing communication

Concurrent components communicate by exchanging messages (exemplified by Scala, Erlang and occam). The exchange of messages may be carried out asynchronously, or may use a rendezvous style in which the sender blocks until the message is received. Asynchronous message passing may be reliable or unreliable. Message-passing concurrency tends to be far easier to reason about than shared-memory concurrency, and is typically considered a more robust form of concurrent programming. A wide variety of mathematical theories for understanding and analyzing message-passing systems are available, including the Actor model, and various process calculi. Message passing can be efficiently implemented on symmetric multiprocessors, with or without shared coherent memory.

Shared memory and message passing concurrency have different performance characteristics. Typically (although not always), the per-process memory overhead and task switching overhead is lower in a message passing system, but the overhead of message passing itself is greater than for a procedure call. These differences are often overwhelmed by other performance factors*[11]*.

3. Software Transactional Memory

Transactional Memory (TM) is a concurrency control paradigm that provides atomic and isolated execution for regions of code. TM is considered by many researchers to be one of the most promising solutions to address the problem of programming multicore processors. Its most appealing feature is that most programmers only need to reason locally about shared data accesses, mark the code region to be executed transactionally, and let the underlying system ensure the correct concurrent execution. This model promises to provide the scalability of fine-grain locking, while avoiding common pitfalls of lock composition such as deadlock.

Different implementations of transactional memory systems make tradeoffs that impact both performance and programmability. Here are some of the design choices:

- STM (software-only TM) Software Transactional Memory, or STM, is an abstraction for concurrent communication. The main benefits of STM are composability and modularity. That is, using STM you can write concurrent abstractions that can be

easily composed with any other abstraction built using STM, without exposing the details of how your abstraction ensures safety. This is typically not the case with other forms of concurrent communication, such as locks. STM implementations exist for many different compilers. While offering flexibility and no hardware cost, it leads to overhead in excess of most users tolerance.

- HTM (hardware-only TM) suffers from two major impediments: high implementation and verification costs lead to design risks too large to justify on a niche programming model. Hardware capacity constraints lead to significant performance degradation when overflow occurs, and proposals for managing overflows incur false positives that add complexity to the programming model. Therefore, from an industrial perspective, HTM designs have to provide more benefits for the cost on a more diverse set of workloads (with varying transactional characteristics) for hardware designers to consider implementation. (Reuse of hardware for other purposes can also justify its inclusion)

- Hybrid systems are the most likely platform for the eventual adoption of TM by a wide audience, although the exact mix of hardware and software support remains unclear. A special case of the hybrid system is the hardware-accelerated STM. In this scenario, the transactional semantics are provided by STM, and hardware primitives are used only to speed up critical performance bottlenecks in the STM system. Such systems could offer an attractive solution if the cost of hardware primitives is modest and may be further amortized by other uses.
[12]

Independent of these implementation decisions, there are transactional semantics issues that break the ideal transactional programming model for which the community had hoped. TM introduces a variety of programming issues that are not present in lock-based mutual exclusion. For example, semantics are muddled by:

- Interaction with non-transactional codes, including access to shared data from outside of a transaction (tolerating weak atomicity) and the use of locks inside a transaction (breaking isolation to make locking operations visible outside transactions).
- Exceptions and serializability—how to handle exceptions and propagate consistent exception information from within a transactional context, and how to guarantee that transactional execution respects a correct ordering of operations.
- Interaction with code that cannot be transnationalized, as a result of either communication with other threads or a requirement barring speculation.
- Livelock, or the system guarantee that all transactions make progress even in the presence of conflicts.

STM implements all the transactional semantics in software. That includes conflict detection, guaranteeing the consistency of transactional reads, preservation of atomicity and isolation

(preventing other threads from observing speculative writes before the transaction succeeds), and conflict resolution (transaction arbitration).

STM consist of four main implementations to the code:

- STM_BEGIN() - starts the STM transaction
- STM_READ() - reads the data
- STM_VALIDATE() - validates the data
- STM_END() - ends the STM transaction

The advantage of STM for programmers is that it offers flexibility in implementing different mechanisms and policies for these operations. For end users, the advantage of STM is that it offers an environment to transactionalize (i.e., port to TM) their applications without incurring extra hardware cost or waiting for such hardware to be developed.

On the other hand, STM entails nontrivial drawbacks with respect to performance and programming semantics:

- **Overheads**

  In general, STM results in higher sequential overheads than traditional shared-memory programming or HTM. This is the result of the software expansion of loads and stores to shared mutable locations inside transactions to tens of additional instructions that constitute the STM implementation (for example, the STM_READ code). Depending on the transactional characteristics of a workload, these overheads can become a high hurdle for STM to achieve performance. The sequential overheads (that is, conflict-free overheads that are incurred regardless of the actions of other concurrent threads) must be overcome by the concurrency-enabling characteristics of transactional memory.

- **Semantics**

  To avoid incurring high STM overheads, nontransactional accesses (i.e., loads and stores occurring outside transactions) are typically not expanded. This has the effect of weakening—and hence complicating—the semantics of transactions, which may require the programmer to be more careful than when strong transactional semantics are supported. The following are some of the weakened guarantees that are usually associated with such STMs:

  - *Weak atomicity*. Typically, the STM runtime libraries cannot detect conflicts between transactions and nontransactional accesses. Thus, the semantics of atomicity are weakened to allow undetected conflicts with nontransactional accesses (referred to as weak atomicity), or equivalently put the burden on the programmer to guarantee that no such conflicts can possibly take place.

- *Privatization*. Some STM designs prohibit the seamless privatization of memory locations (that is, the transition from being accessed transactionally to being accessed privately—or nontransactionally in general, by using locks). For some STM designs, once a location is accessed transactionally, it must continue to be accessed that way. Sometimes, the programmer can ease the transition by guaranteeing that the first access to the privatized location—such as after the location is no longer accessible by other threads—is transactional.
- *Memory reclamation*. Some STM designs prohibit the seamless reclamation of the memory locations accessed transactionally for arbitrary reuse, such as using malloc and free. With such STM designs, memory allocation and deallocation for locations accessed transactionally are handled differently than for other locations.
- *Legacy binaries*. STM needs to observe all memory activities of the transactional regions to ensure atomicity and isolation. STM designs that achieve this observation by code instrumentation generally cannot support transactions calling legacy codes that are not instrumented (for example, third-party libraries) without seriously limiting concurrency, such as by serializing transactions.

  [13]

4. Development Tools

RSTM Library

Introduction

RSTM is a C++ package that now contains thirteen different STM library implementations, and a smart-pointer based API that allows consistent, high-performance, safe, and relatively transparent access to STM for user applications. RSTM is a research prototype, but has been successfully tested on a variety of benchmarks and applications.

RSTM Algorithms

The RSTM library supports several different underlying STM algorithms. All algorithms are blocking, and all operate at the granularity of individual words, as opposed to language-level objects*[14]*.

1. Single-Lock Algorithms

These algorithms do not allow for concurrent execution of transactions. However, they have very low single-thread latency. In all cases, a single lock is acquired at transaction begin, and released at transaction commit. Unless otherwise stated, these algorithms do not allow explicit self-abort. These algorithms all offer privatization safety.

2. Sequence-Lock Algorithms

These algorithms use a sequence lock as the only global metadata. A sequence lock is odd when held, and even when available. These algorithms all offer privatization safety.

- **TML**: Optimized for workloads with lots of read-only transactions. Transactions do not log their reads, and writes are performed in-place. Whenever a transaction performs its first write, all concurrent transactions abort, and no new transactions start until the writer commits. Self-abort of writing transactions is not allowed. Details are available in (Dalessandro EuroPar 2010).
- **TMLLazy**: Like TML, but writes are buffered until commit time. This allows more concurrency, and supports self-abort. However, there is still no concurrency among writers.
- **NOrec**: Extends TMLLazy with value-based validation. This means that when a writer commits, all concurrent transactions can check the actual values they read, in order to determine if they can continue operating or not. Details are available in (Dalessandro PPoPP 2010).
  - NOrec Variants: There are several available variants of NOrec, based on the behavior taken upon abort. The variants are NOrec (immediate restart on abort), NOrecHour (uses the Hourglass protocol (Liu Transact 2011)), NOrecBackoff (exponential backoff on abort), NOrecHB (Hourglass + backoff)
- **NOrecPrio**: Extends NOrec with simple priority. Transactions block at their commit point if there exist higher-priority in-flight transactions.

3. Orec-Based Algorithms (Not Privatization Safe)

Ownership records, or "orecs", are used to detect conflicting accesses to memory. These algorithms are not privatization safe.

- **LLT**: Writes are buffered, and locks are acquired at commit time. A timestamp is used to limit validation, using the GV1 algorithm. This algorithm resembles TL2 (Dice DISC 2006).
- **OrecEager**: Writes are performed in-place, and on abort, an undo log is used. This algorithm resembles TinySTM (Felber PPoPP 2008) with write-through, except that we do not use incarnation numbers.

    - OrecEager Variants: OrecEager (immediate restart on abort), OrecEagerHour (Hourglass on abort), OrecEagerBackoff (exponential backoff on abort), OrecEagerHB (Hourglass + backoff)
- **OrecEagerRedo**: Like OrecEager, but with redo logs (still using encounter-time locking). Resembles TinySTM with write-back.
- **OrecLazy**: Like OrecEager, except that commit-time locking is used. This algorithm resembles the unpublished "CTL" version of TinySTM, as well as the "Patient" algorithm in (Spear PPoPP 2009). However, this algorithm uses timestamps in the same manner as (Wang CGO 2007), unlike previously published lazy STM algorithms.
    - OrecLazy Variants: OrecLazy, OrecLazyHour, OrecLazyBackoff, OrecLazyHB
- **OrEAU**: These are OrecEager variants extended to support remote abort of conflicting transactions.
    - OrEAU Variants: OrEAUBackoff (backoff on abort), OrEAUFCM (a timestamp mechanism to manage aborts), OrEAUNoBackoff (always abort other on conflict), OrEAUHour (use Hourglass protocol on abort)
- **OrecFair**: This is like OrecLazy, extended to support priority. Transactions can gain priority, and then be guaranteed to win conflicts, while remaining lazy. The algorithm most closely resembles one from (Spear PPoPP 2009).
- **Swiss**: Like SwissTM (Dragojevic PLDI 2009). The algorithm uses eager locking with redo logs, and a two-pass commit to implement mixed invalidation as in (Spear DISC 2006).
- **Nano**: A locking version of WSTM (Fraser OOPSLA 2003). There is **quadratic** validation overhead, and no global timestamp. For frequent, tiny transactions, this algorithm is very good. For larger workloads, it is much worse (asymptotically and in practice).

4. Orec-Based Algorithms (Privatization Safe)

These algorithms use a "two-counter" technique and polling to add privatization safety. The technique has been described in (Marathe ICPP 2008), (Spear OPODIS 2008), (Detlefs (unpublished)), and (Dice Transact 2009).

- **OrecELA**: Extends LLT (TL2) to achieve ALA publication safety and privatization safety.
- **OrecALA**: Extends OrecLazy to achieve only privatization safety.

5. Signature-Based Algorithms

These algorithms use signatures (single-hash-function Bloom Filters) for concurrency control.

- **RingSW**: The single-writer variant of RingSTM (Spear SPAA 2008), extended to support SSE instructions on the x86.
- **RingALA**: Extends RingSW with an extra per-thread 'conflicts filter' to achieve ALA publication safety.

6. Byte Lock Algorithms

These algorithms use bytelocks, as proposed in (Dice Transact 2009) and (Dice SPAA 2010). Readers are visible, which avoids the need for validation.

- **ByteEager**: Uses in-place update with undo logs and timeout-based conflict resolution. Very similar to TLRW (Dice SPAA 2010).
- **ByteEagerRedo**: Like ByteEager, but uses in-place update with redo logs.
- **ByEAU**: Adds remote-abort support to ByteEager, so that conflicts can be resolved without timeout. Note that this does not usually improve performance.
    - ByEAU Variants: ByEAUBackoff (backoff on abort), ByEAUFCM (timestamps for managing aborts), ByEAUNoBackoff (immediate restart on abort), ByEAUHour (Hourglass protocol on abort)
- **ByEAR**: Adds remote-abort support to ByteEagerRedo.
- **ByteLazy**: Like ByteEager, but with commit-time write locking and redo logs. Note that read locks are still acquired eagerly.

7. Bit Lock Algorithms

Bitlocks, as proposed in (Marathe Transact 2006), provide an alternative to bytelocks for implementing visible reads.

- **BitEager**: Like ByteEager, but with Bitlocks.
- **BitEagerRedo**: Like ByteEagerRedo, but with Bitlocks.
- **BitLazy**: Like ByteLazy, but with Bitlocks.

8. Invalidation Algorithms

With invalidation, committing transactions forcibly abort conflicting transactions.

- **TLI**: A version of InvalSTM that does not require the use of per-thread mutexes, instead favoring ordered writes of per-thread metadata and a sequence lock for commits.

9. Globally Ordered Algorithms

These algorithms are based on work on thread-level speculation with transactions (Spear LCPC 2009). Transactions are assigned a commit order very early (details vary by algorithm), and exploit order to reduce overheads.

- **Pipeline**: Every transaction gets an order at begin time. The 'oldest' transaction performs writes in-place, while other transactions buffer writes. Note that self-abort is not allowed.
- **CTokenTurbo**: Only writer transactions get an order, and they do so at the time of their first write. Self-abort is not allowed by writer transactions. The oldest transaction performs writes in-place.
- **CToken**: Like CTokenTurbo, except that the oldest transaction still buffers its writes until commit time, so that self-abort can be supported.

RSTM is one of the oldest open-source Software Transactional Memory libraries. Since its first release in 2006, it has grown to include several distinct STM algorithms. It also supports several architectures and operating systems (x86 / SPARC; Linux, Solaris, MacOS).

As a research system, not all configurations are currently supported. However, among the various options one can find support for strong semantics (privatization, publication), irrevocability, condition synchronization (via 'retry'), and strong progress guarantees.

Building RSTM

There are two main targets involved in building RSTM:

- **Library API**: This target is suitable for SPARC, and for x86 platforms that use gcc as a compiler.
- **CXXTM API**: This target is suitable for x86 platforms that have the Intel transactional C++ compiler and is in addition to the Library API on such platforms.

If you are building to use the Library API, then you will need to build libstm.a. If you are building to use the CXXTM API, then you will need to build both libstm.a and libitm2stm.a. Cmake will help you do this easily.

Note that libitm2stm requires certain settings for libstm, which can introduce overheads (byte-granularity support, cancel-and-throw, stack protection) that can be avoided with the Library API. However, with the Library API, it is the programmer's responsibility to correctly annotate every shared memory access within each transaction (particularly in order to avoid the need for these features!). In cmake, if you enable the option for building libitm2stm, support for these features will be turned on automatically.

CMake

As of the 7th release, RSTM uses the CMake configuration infrastructure which is available in binary form for many different platforms. A command cmake <path-to-RSTM> will

configure RSTM with default parameters for your platform, using the Makefile generator. After configuration user can simply make and all of the default libraries and executables will be generated.

```
unpack rstm to /home/me/rstm_src
```

```
mkdir /home/me/rstm_build
```

```
cd /home/me/rstm_build
```

```
cmake ../rstm_src
```

```
make
```

There is three options available for customizing your configuration:

- User may do an *interactive configuration* using cmake's -i flag. This will prompt him for answers to all of the possible configuration options.
- User may explicitly set options on the command line using cmake's -D command line functionality.
- User may configure using a cmake gui, such as the ccmake ncurses gui included in the cmake distribution.

Note that our configurations are adaptive in the sense that enabling certain options may add additional options to your gui. User configuration options are declared in UserConfig.cmake files in the appropriate directories.

RSTM Options

RSTM options are defined in <RSTM>/UserConfig.cmake. These options control the RSTM configuration itself.

CMake Options

This describes the cmake configuration options as of RSTM's 7th release. Note that there are some dependencies between these options, so some may not be visible on your platform given the rest of configuration settings.

libstm Options

libstm options are defined in <RSTM>/libstm/UserConfig.cmake. These options control the libstm build and are embedded in the configuredinclude/stm/config.h header so that libstm-dependent libraries and applications know how the library was configured.

libitm2stm Options

These options are defined in <RSTM>/libitm2stm/UserConfig.cmake and control how the libitm2stm build is performed.

Bench Options

Bench options are defined in <RSTM>/bench/UserConfig.cmake and control how the bench build occurs. Each benchmark can be built as a single or a multi-source source build, which provides the compiler the best opportunity to optimize the results.

STAMP Options

The STAMP options are defined in <RSTM>/stamp-0.9.10/UserConfig.cmake and control how the C++ STM build occurs.

Mesh Options

Mesh options are defined in <RSTM>/mesh/UserConfig.cmake and control which versions of the mesh application to build. Mesh currently only supports 32-bit execution, so this option is only available if you have selected rstm_build_32-bit. Mesh will always be built with an libitm and libitm2stm version depending on your selections for rstm_enable_itm and rstm_enable_itm2stm.

[15]

There are multiple applications that are included with RSTM, they include:

- Microbenchmarks:
    - Data Structure Microbenchmarks
    - Behavior Microbenchmarks
    - Pathologies and Unit Tests
    - Configuration Parameters
- STAMP (Minh IISWC 2008):
    - C++ STM Stamp
- Delaunay triangulation (Scott IISWC 2007)

Writing apps in RSTM

RSTM supports two APIs. The Library API is the legacy interface for RSTM. On SPARC, it is currently the only supported interface. On x86 with the Intel transactional C++ compiler, it is also possible to use the C++ STM Draft API. It is also possible to write code that works with both APIs.

C++ STM API

If user is planning on using the Intel TM C++ compiler, you could just follow the rules in the draft C++ TM specification. That is, you could:

- wrap transactional regions with __transaction[[atomic]] { ... }
- mark transactional functions as [transaction_safe]
- mark regions that should not be instrumented with transaction [waiver] { ... }
- place the proper calls to initalize and shut down the STM library, and each thread's STM context.

By adding an include of <api/api.hpp>, you will also gain the following:

- TM_SET_POLICY(P) to set the algorithm or adaptivity policy during execution.
- TM_GET_ALGNAME() to get the initial algorithm with which libstm was configured.

Library API

The Library API allows maximum flexibility. You can create new interfaces to libstm, and exploit them immediately. However, it is also difficult. With the library API, the programmer is responsible for making any shared memory access safe. The programmer is also responsible for ensuring that allocation/reclamation are safe, and for adding API calls necessary for good performance. Some of the Library API examples:

- libSTM
- fastSTM
- fastBT

Supporting Both APIs

If user programs to the Library API, and then compiles with the Intel C++ compiler, everything will “just work” regardless of which API you use to configure and build.

[16]

SFML

What is SFML?

SFML (Simple and Fast Multimedia Library) is a portable and easy-to-use API for multimedia programming. It is written in C++ but bindings are available for C, D, Python, Ruby, OCaml, .Net and Go. It can be thought of as an object oriented alternative to SDL. SFML also provides different modules made to ease programming games and multimedia applications

SFML provides 2D graphics that are hardware accelerated with OpenGL and can be used as a minimalist window system, or as a complete multimedia library full of features to build video games or multimedia softwares. It can also be used for OpenGL windowing. SFML site offers complete SDK bundle in single pack, and tutorials to ease the developers. SFML Source code is provided under the terms of the zlib/png license.

On which platforms is SFML currently available?

SFML is currently available and fully functional in Windows (8, 7, Vista, XP, 2000, 98), Linux and Mac OS X. SFML works on both 32 and 64 bit systems.

Which programming languages are supported by SFML?

SFML is implemented in C++. That said, several bindings have been created for other languages that allow SFML to be used from C, C#, C++/CLI, D, Ruby, OCaml, Java, Python and VB.NET.

What dependencies does SFML have?

SFML depends on a few other libraries, so before starting to compile you must have their development files installed.

On Windows and Mac OS X, all the needed dependencies are provided directly with SFML, so you don't have to download/install anything. Compilation will work out of the box.

On Linux however, nothing is provided and SFML relies on your own installation of the libraries it depends on. Here is a list of what you need to install before compiling SFML:

- pthread
- opengl
- xlib
- xrandr
- freetype
- glew
- jpeg
- sndfile

- openal

The exact name of the packages depend on each distribution. And don't forget to install the development version of these packages.

SFML has also internal dependencies: Audio and Window depend on System, while Graphics depends on System and Window. In order to use the Graphics module, you must link with Graphics, Window, and System (the order of linkage matters with GCC).

What and how do I link to use SFML?

When you want to use SFML, you need to link to the library files that provide the functionality you make use of in your application.

SFML is divided into 5 modules:

- **System** *provided by sfml-system*
- **Window** *provided by sfml-window*
- **Graphics** *provided by sfml-graphics*
- **Audio** *provided by sfml-audio*
- **Network** *provided by sfml-network*

Be aware that the modules have interdependencies on each other. For instance, if you plan on using the Graphics module, you will also have to link against the Window and System modules as well.

Dependencies:

- **System** does not depend on anything and can be used by itself.
- **Window** depends on **System**.
- **Graphics** depends on **Window** and **System**.
- **Audio** depends on **System**.
- **Network** depends on **System**.

[17]

Sample SFML program

The program below provides a short overview of the SFML. This code just opens a window and fills it with blue:

```
#include <SFML/Graphics.hpp>

int main()
{
    // Create the main window
    sf::RenderWindow App(sf::VideoMode(800, 600, 32), "Hello World - SFML");
```

```
// Start the main loop
while (App.isOpen())
{
    // Process events
    sf::Event Event;
    while (App.pollEvent(Event))
    {
        // Close window : exit
        if (Event.type == sf::Event::Closed)
            App.close();
    }

    // Clear screen, and fill it with blue
    App.clear(sf::Color(0x00, 0x00, 0xff));

    // Display the content of the window on screen
    App.display();
}

return 0;
}
```

[18]

Microsoft Visual Studio

Introduction

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft. It is used to develop console and graphical user interface applications along with Windows Forms or WPF applications, web sites, web applications, and web services in both native code together with managed code for all platforms supported by Microsoft Windows, Windows Mobile, Windows CE, .NET Framework, .NET Compact Framework and Microsoft Silverlight.

Visual Studio includes a code editor supporting IntelliSense as well as code refactoring. The integrated debugger works both as a source-level debugger and a machine-level debugger. Other built-in tools include a forms designer for building GUI applications, web designer,class designer, and database schema designer. It accepts plug-ins that enhance the functionality at almost every level—including adding support for source-control systems (like

Subversion and Visual SourceSafe) and adding new toolsets like editors and visual designers for domain-specific languages or toolsets for other aspects of the software development lifecycle (like the Team Foundation Server client: Team Explorer).

Visual Studio supports different programming languages by means of language services, which allow the code editor and debugger to support (to varying degrees) nearly any programming language, provided a language-specific service exists. Built-in languages includeC/C++ (via Visual C++), VB.NET (via Visual Basic .NET), C# (via Visual C#), and F# (as of Visual Studio 2010). Support for other languages such as M, Python, and Ruby among others is available via language services installed separately. It also supports XML/XSLT,HTML/XHTML, JavaScript and CSS. Individual language-specific versions of Visual Studio also exist which provide more limited language services to the user: Microsoft Visual Basic, Visual J#, Visual C#, and Visual C++.

Architecture

Visual Studio does not support any programming language, solution or tool intrinsically, instead it allows the plugging of functionality coded as a VSPackage. When installed, the functionality is available as a *Service*. The IDE provides three services: SVsSolution, which provides the ability to enumerate projects and solutions; SVsUIShell, which provides windowing and UI functionality (including tabs, toolbars and tool windows); and SVsShell, which deals with registration of VSPackages. In addition, the IDE is also responsible for coordinating and enabling communication between services. All editors, designers, project types and other tools are implemented as VSPackages. Visual Studio uses COM to access the VSPackages. The Visual Studio SDK also includes the *Managed Package Framework* (*MPF*), which is a set of managed wrappers around the COM-interfaces that allow the Packages to be written in any CLI compliant language. However, MPF does not provide all the functionality exposed by the Visual Studio COM interfaces. The services can then be consumed for creation of other packages, which add functionality to the Visual Studio IDE.

Support for programming languages is added by using a specific VSPackage called a *Language Service*. A language service defines various interfaces which the VSPackage implementation can implement to add support for various functionalities. Functionalities that can be added this way include syntax coloring, statement completion, brace matching, parameter information tooltips, member lists and error markers for background compilation. If the interface is implemented, the functionality will be available for the language. Language services are to be implemented on a per-language basis. The implementations can reuse code from the parser or the compiler for the language. Language services can be implemented either in native code or managed code.

Features

- **Code editor**

Like any other IDE, it includes a code editor that supports syntax highlighting and code completion using IntelliSense for not only variables, functions and methods but also language constructs like loops and queries

- **Debugger**
  Visual Studio includes a debugger that works both as a source-level debugger and as a machine-level debugger. It works with both managed code as well as native code and can be used for debugging applications written in any language supported by Visual Studio. In addition, it can also attach to running processes and monitor and debug those processes. If source code for the running process is available, it displays the code as it is being run. If source code is not available, it can show the disassembly. The Visual Studio debugger can also create memory dumps as well as load them later for debugging. Multi-threaded programs are also supported. The debugger can be configured to be launched when an application running outside the Visual Studio environment crashes.
- **Designer**
  Visual Studio includes a host of visual designers to aid in the development of applications. These tools include:
    - Windows Forms Designer
    - WPF Designer
    - Web designer/development
    - Class designer
    - Data designer
    - Mapping designer

Included Products

- **Microsoft Visual C++**

Microsoft Visual C++ is Microsoft's implementation of the C and C++ compiler and associated languages-services and specific tools for integration with the Visual Studio IDE. It can compile either in C mode or C++ mode. Visual C++ can also be used with the Windows API. It also supports the use of *intrinsic functions*, which are functions recognized by the compiler itself and not implemented as a library. Intrinsic functions are used to expose the SSE instruction set of modern CPUs. Visual C++ also includes the OpenMP (version 2.0) specification.

- **Microsoft Visual C#**

Microsoft Visual C#, Microsoft's implementation of the C# language, targets the .NET Framework, along with the language services that lets the Visual Studio IDE support C# projects. While the language services are a part of Visual Studio, the compiler is available separately as a part of the .NET Framework. The Visual C# 2008, 2010 and 2012 compilers support versions 3.0, 4.0 and 5.0 of the C# language specifications,

respectively. Visual C# supports the Visual Studio Class designer, Forms designer, and Data designer among others.

- **Microsoft Visual Basic**

Microsoft Visual Basic is Microsoft's implementation of the VB.NET language and associated tools and language services. It was introduced with Visual Studio .NET (2002). Visual Basic can be used to author both console applications as well as GUI applications. Like Visual C#, Visual Basic also supports the Visual Studio Class designer, Forms designer, and Data designer among others. Like C#, the VB.NET compiler is also available as a part of .NET Framework, but the language services that let VB.NET projects be developed with Visual Studio, are available as a part of the latter.

- **Microsoft Visual Web Developer**

Microsoft Visual Web Developer is used to create web sites, web applications and web services using ASP.NET. Either C# or VB.NET languages can be used. Visual Web Developer can use the Visual Studio Web Designer to graphically design web page layouts.

- **Team Foundation Server**

Included only with Visual Studio Team System, Team Foundation Server is intended for collaborative software development projects and acts as the server-side backend providing source control, data collection, reporting, and project-tracking functionality. It also includes the *Team Explorer*, the client tool for TFS services, which is integrated inside Visual Studio Team System.

Extensibility

Visual Studio allows developers to write extensions for Visual Studio to extend its capabilities. These extensions "plug into" Visual Studio and extend its functionality. Extensions come in the form of *macros*, *add-ins*, and *packages*. Macros represent repeatable tasks and actions that developers can record programmatically for saving, replaying, and distributing. Macros, however, cannot implement new commands or create tool windows. They are written using Visual Basic and are not compiled. Add-Ins provide access to the Visual Studio object model and can interact with the IDE tools. Add-Ins can be used to implement new functionality and can add new tool windows. Add-Ins are plugged into the IDE via COM and can be created in any COM-compliant languages. Packages are created using the Visual Studio SDK and provide the highest level of extensibility. They can create designers and other tools, as well as integrate other programming languages. The Visual Studio SDK provides unmanaged APIs as well as a managed API to accomplish these tasks. [1]Extensions are supported in the Standard (and higher) versions of Visual Studio 2005. Express Editions do not support hosting extensions.

Visual Studio 2012

Final build of Visual Studio 2012 was announced on August 1, 2012 and the official launch event was held on September 12, 2012. Unlike prior versions, Visual Studio 2012 can't record and playback macros and the macro editor is gone.

A major new feature is support for WinRT and C++/CX (Component Extensions). Support for C++ AMP (GPGPU programming) is also included.

On 16 September 2011 a complete 'Developer Preview' of Visual Studio 11 was published on Microsoft's website. Visual Studio 11 Developer Preview requires Windows 7, Windows Server 2008 R2, Windows 8, or later operating systems. Versions of Microsoft Foundation Class Library (MFC) and C runtime (CRT) included with this release cannot produce software that is compatible with Windows XP or Windows Server 2003 except by using native multi-targeting and foregoing the newest libraries, compilers, and headers. However, on June 15, 2012, a blog post on the VC++ Team blog announced that based on customer feedback, Microsoft would re-introduce native support for Windows XP targets (though not for XP as a development platform) in a version of Visual C++ to be released later in the fall of 2012. "Visual Studio 2012 Update 1" (Visual Studio 2012.1) was released in November 2012. This update added support for Windows XP targets and also added other new tools and features (e.g. improved diagnostics and testing support for Windows Store apps).

On the 24 August 2011, a blog post by Sumit Kumar, a Program Manager on the Visual C++ team listed some of the features of the upcoming version of the Visual Studio C++ IDE:

- Semantic Colorization: Improved syntax coloring, various user-defined or default colors for C++ syntax such as macros, enumerations, typenames, functions etc.
- Reference Highlighting: Selection of a symbol highlights all of the references to that symbol within scope.
- New Solution Explorer: New solution explorer allows for visualization of class and file hierarchies within a solution/project. Searching for calls to functions and uses of classes will be supported.
- Automatic Display of IntelliSense list: IntelliSense will automatically be displayed whilst typing code, as opposed to previous versions where it had to be explicitly invoked through use of certain operators (i.e. the scope operator (::)) or shortcut keys (*Ctrl-Space* or *Ctrl-J*).
- Member List Filtering: IntelliSense uses fuzzy logic to determine which functions/variables/types to display in the list.
- Code Snippets: Code snippets are included in IntelliSense to automatically generate relevant code based on the user's parameters, custom code snippets can be created.

The source code of Visual Studio 2012 consists of approximately 50 million lines of code.

[19]

[1] http://en.wikipedia.org/wiki/Clock_rate

[2] http://en.wikipedia.org/wiki/Moore's_law

[3] "AI for Games" book, Author: Ian Millington

[4] "The Art of Concurrency" book, Author: Clay Breshears

[5] http://arxiv.org/ftp/arxiv/papers/1003/1003.1476.pdf

[6] "The Art of Multiprocessor Programming" book, Authors: Maurice Herlihy & Nir Shavit

[7] http://en.wikipedia.org/wiki/Thread_(computing)

[8] https://computing.llnl.gov/tutorials/parallel_comp/

[9] http://milindshakya.tumblr.com/post/5399363431/the-need-for-parallel-programming

[10] http://en.wikipedia.org/wiki/Amdahl's_law

[11] http://en.wikipedia.org/wiki/Concurrent_computing

[12] http://pages.cs.wisc.edu/~cain/pubs/cascaval_cacm08.pdf

[13] http://queue.acm.org/detail.cfm?id=1400228

[14] https://code.google.com/p/rstm/wiki/AvailableAlgorithms

[15] https://code.google.com/p/rstm/wiki/BuildingRSTM

[16] https://code.google.com/p/rstm/wiki/WritingApps

[17] https://github.com/LaurentGomila/SFML/wiki/FAQ#wiki-grl-whatis

[18] http://en.wikipedia.org/wiki/SFML

[19] http://en.wikipedia.org/wiki/Microsoft_Visual_Studio