

Software Transactional Memory

Michel Weimerskirch
22nd January 2008

Technical University of Kaiserslautern, 67653 Kaiserslautern, Germany
michel@weimerskirch.net
WWW home page: <http://michel.weimerskirch.net/>

Abstract. The paper starts with the observation that it is less than ideal to use locks in multiprocessor environments. The solution proposed is to use Software Transactional Memory (STM), a newly approach that accesses memory objects in a way similar to database transactions. After the operation principle of the original lock-free algorithm is presented, a small overview of the current state of language support for STM is given. The subsequent section gives an introduction to several optimisation approaches with a focus on lock-based STMs. This leads to the TL2 algorithm, which can be considered state of the art. After this section, a few of the optimised algorithms will be compared to each other. Finally, the concluding section enumerates a few problems that need to be solved before transactional memory algorithms are ready for the end-user market.

Key words: Software Transactional Memory (STM), Transactional Locking (TL), multiprocessors, multithreading, concurrency

1 Introduction

A common problem of multithreaded applications or separate applications accessing shared memory is concurrency control. Current applications usually use locking to access shared memory, but this has several well-known disadvantages like bad scalability or bad maintainability. As many developers will have experienced, it is an extremely tedious task to choose the right locking granularity (not too many, not too few) and to apply and release them in the right order to avoid deadlocks.

This gets even more critical in multiprocessor environments where locking-based concurrency control can reduce the theoretical performance gain of having multiple processors. On one hand, two threads might for example unnecessarily lock each other out if locks are too coarse-grained even if both threads access different data objects. On the other hand, if locks are too fine grained, the perpetual locking and unlocking operations can cause a lot of unnecessary overhead. [8]

Considering that recent developments in the personal computer area have introduced multicore processors intended for home use, it is extremely important to have efficient and effective concurrency control mechanisms. Developers

should thus be able to easily optimize applications in order to explicitly benefit from running on multiple cores.

The approach described in this paper, Software Transactional memory (STM), enables developers to operate on the memory in a similar way to using database transactions. The idea to implement memory operations in a transactional way originates from a 1986 patent by Tom Knight [1]. Therein he describes a hardware based transactional memory system. While a hardware based transactional memory system would have a clear performance advantage, the software based approach described in this paper offers more flexibility and minimises portability issues on today's machines.

2 Basics

Transactional operations have been around for many years in the area of database applications. It is an efficient and safe way to manage concurrent access to data structures that is easy to understand and to apply by the users of such a system.

In the general software development context, a transaction is a series of read and write memory operations that appear as one single atomic operation to other threads. According to [2], any transaction may either fail, or complete successfully, in which case its changes are visible *atomically* to other processes. *Atomic* means that the operations inside the transaction are either fully executed or reverted, in which case other successful transactions will not be affected by the intermediate states the transaction went through.

The approach has several distinct advantages. Above all, developers do not need to worry about writing parallel code. Instead, they can write code just as if it was executed serially. A layer in between application and memory handles all the system specific details for making the application run in parallel. Current implementations typically have a special language construct that is used to mark the code that is to be executed in a transaction. This will be explained in more detail in section 3.

In some cases, STM might not be the ideal approach, as it may introduce performance impacts thus making lock based memory faster than STM. This is for example the case when many successive collisions and ensuing rollbacks happen. Another problem is related to the fact that not every operation can be rolled back or reverted, which is why I/O operations like for example writing to a log file can not be included in a transaction but have to happen outside of any transactional context.

2.1 Operating principle

This section describes the operating principles of the original STM algorithm as characterised in [2]. Different optimisation approaches of current research will be addressed in section 4. The algorithm described in this section does not use locks, which is why liveness is guaranteed.

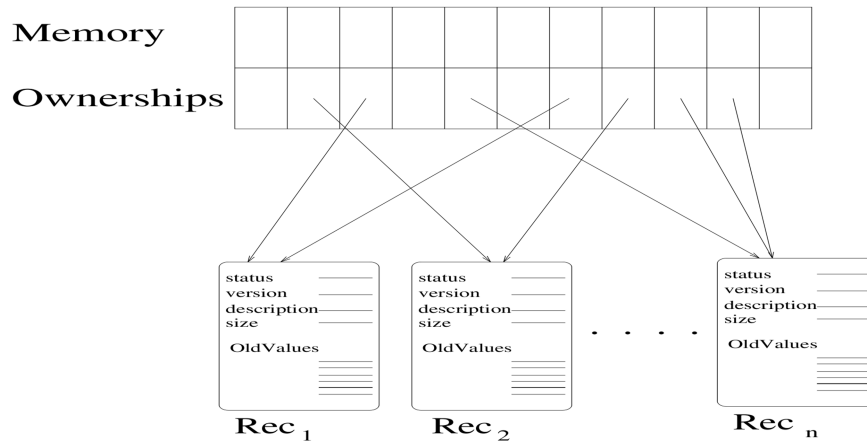


Fig. 1. STM datastructures [2, p. 8]: STM “objects” are grouped in a vector called **Ownerships**. Metadata for a transaction i is managed in a record **Rec _{i}** .

STMs are memory “objects” that are being operated on. As depicted in figure 1, those objects are grouped in a vector we call **Memory**. Each entry in **Memory** has a corresponding record in a vector **Ownerships** that points to records **Rec _{i}** for each transaction i . Those records contain the addresses of the currently owned memory objects as well as the old size and old value of each owned memory object. Additionally they have a version counter that reflects the starting order of the transaction.

Each time a transaction i is initiated, it verifies that none of the objects from the write set are owned by another transaction. If this is the case the transaction aborts. Otherwise the ownership of the memory object is attributed to that process by inserting pointers from the respective elements in the **Ownerships** vector to the record **Rec _{i}** of the current transaction. Calculations are now done without referring to other transactions.

The tricky part of the STM algorithm lies within the commit operation. It is important to notice that the responsibility for avoiding collisions is not borne by writing transactions but by reading transactions. This means that a reading transaction will have to verify at commit time that its read set has not been modified by a writing transaction. Otherwise the reading transaction is aborted and changes to memory objects are written back to their initial state (which is saved in the **OldValues** vector inside the record **Rec _{i}**).

If a transaction conflicts with an already committed change by another transaction it is either aborted or automatically repeated, depending on the implementation, without having the developer worry about the details. This makes it of course important that inside the transactions no external operations are executed that can not be easily reversed, like for example access to the file system.

In addition to this, Nir Shavit and Dan Touitou describe in their “Software Transactional Memory” paper a non-redundant-helping approach. This basically means that a transaction encountering a lock will help the locking transaction complete its current operation. Due to the ownership concept, a transaction only needs to help the single owner of a location in order to free it. [2, pp. 2-3]

The STM algorithm is obviously a very optimistic approach that assumes that collisions do not happen very often. As explained in section 6 (Empirical evaluation), the concept does work nevertheless.

3 Language support

This section will give an introduction into what language constructs of STM based software could look like. There are currently several different approaches to integrate transactional memory but this section will focus on a straightforward declarative approach.

As an introductory non-STM example we take the following code segment that shows how a new node is inserted into a double-linked list. An obvious problem of this type of coarse-grained lock-based code is that other `synchronized` methods will also be blocked, which means that other threads accessing other synchronized methods might be unnecessarily retained from processing:

```
public synchronized void insertNode(node, precedingNode) {
    node.prec = precedingNode;
    node.succ = precedingNode.succ;
    precedingNode.succ.prec = node;
    precedingNode.succ = node;
}
```

Alternatively, the method could theoretically be implemented using more fine-grained locks. Because the double-linked list is not be locked in its entirety anymore, this reduces the problem of one thread unnecessarily blocking another one while the `insertNode` method is being processed:

```
public void insertNode(node, precedingNode) {
    synchronized(precedingNode) {
        synchronized(precedingNode.succ) {
            node.prec = precedingNode;
            node.succ = precedingNode.succ;
            precedingNode.succ.prec = node;
            precedingNode.succ = node;
        }
    }
}
```

However, this variation is far less readable and potentially introduces deadlocks if the code is not carefully written, for example if another method contains the following locking schema:

```

synchronized(precedingNode.succ) {
    synchronized(precedingNode) {
        ...
    }
}

```

The example would obviously profit from the STM approach because the two major problems described above (unnecessary blocking and risk of deadlocks) will be eliminated. A straightforward STM language construct is proposed in [3]. The quite simplistic approach is to declaratively mark code that is to be executed atomically. The following example demonstrates the use of such a keyword.

```

public void insertNode(node, precedingNode) {
    atomic {
        node.prec = precedingNode;
        node.succ = precedingNode.succ;
        precedingNode.succ.prec = node;
        precedingNode.succ = node;
    }
}

```

Because transactions are used, operations on other nodes can run concurrently. Collisions are thereby handled transparently. At the end of the atomic block the transaction is committed automatically without further intervention by the developer.

The paper additionally suggests the introduction of so-called guard conditions that block calling threads until they fulfilled. This lightweight construct replaces the overuse of while-loops as a waiting condition.

```

public int get() {
    atomic (items != 0) {
        items--;
        return buffer[items];
    }
}

```

Such a group of operations that has to be executed in isolation is called Conditional Critical Region (CCR), a concept introduced by Hoare. Ideally, as few restrictions as possible should affect the applicability of such a CCR to a piece of code. By not restricting the methods or operations that can be included in a CCR, existing standalone single-threaded libraries can easily be made thread safe by marking any call on them as *atomic*.

Obviously, regions outside of CCRs should not be affected too much by the implementation of the *atomic* operator. It would be counterproductive if access to variables outside of CCRs would be overly complicated or even slower because of the overhead of managing separate copies.

Those two properties were the leading design principles for the *atomic* operator [3, p. 3]. The implementation is based on a research version of the JVM from Sun and is available under an open source license¹.

A language extension like a special construct to mark atomic blocks is not the only way to implement STM algorithms. Other implementations depend on other mechanisms like for example code generation.

Today, implementations of STM algorithms are available for many languages including C², C++³, C#⁴ and Java⁵

4 Optimisation approaches

The original STM algorithm implemented a lock-free procedure in order to ensure liveness. However, as explained in the following paragraphs, this is not always unproblematic. Instead, a lock-based algorithm will be presented.

4.1 About obstruction freedom in STM algorithms

Two basic types of multithreaded applications can be distinguished. The first type of applications uses threading as a way of "convenience", i.e. allowing user interaction while a computation is running in the background. The second type uses threading to improve performance, i.e. to make it profit from a multiprocessor environment.

For both of these types, it is important to notice that it is acceptable for one transaction to block other transactions with equal or lower priority because this would just be the behaviour if the program was executed sequentially [4, pp. 2-3]. Because of the simpler algorithms, the lock-based approach significantly reduces overhead and makes STM transactions faster than with non-locking approaches.

4.2 Why obstruction-freedom is not needed

In order to prove that obstruction-freedom is counterproductive we first need to show that it is unnecessary. Three arguments in favor of obstruction-freedom need to be disproven [4, pp. 3-4]

1. *Obstruction-freedom prevents a long-running transaction blocking others.*

¹ Available here: <http://www.cl.cam.ac.uk/research/srg/netos/lock-free/>

² E.g. Lightweight Transaction Library (LibLTX) for C:
<https://sourceforge.net/projects/libltx>

³ E.g. Intel C++ STM Compiler (Prototype Edition):
<http://softwarecommunity.intel.com/articles/eng/1460.htm>

⁴ E.g. C# Software Transactional Memory:
<http://research.microsoft.com/research/downloads/Details/6cfc842d-1c16-4739-afaf-edb35f544384/Details.aspx>

⁵ E.g. JSTM:
<http://jstm.sourceforge.net/>

2. *Obstruction-freedom prevents the system locking up if a thread is switched part-way through a transaction.*
3. *Obstruction-freedom prevents the system locking up if a thread fails.*

It may be partially true that long-running transactions block others if obstruction-freedom is abolished. However, the crucial point is that this is only true if no conflicts appear between the long-running transaction and the intermediate ones. Otherwise, the intermediate transactions will repeatedly hold the long-running one from ever completing. Therefore, this argument in favour of obstruction-freedom does not hold.

The second argument states that obstruction-freedom prevents the system from locking up if a thread is switched part-way through a transaction. In his paper, Ennals claims that this not critical. In fact, the runtime system will adapt the task switching appropriately as to make the time that is lost by a switched-out process practically irrelevant for the computation time.

The third and last argument in favour of obstruction-freedom states that obstruction-freedom prevents the system locking up if a thread fails. This argument is quickly nullified because a software failure would also break a non-STM version of the same application. It would thus make no difference for an STM application.

At this point it has been established that obstruction-freedom can be abolished because it is unnecessary. We now need to demonstrate that it is also counterproductive.

4.3 Why obstruction-freedom is counterproductive

There are two main reasons why obstruction-freedom is counterproductive [4, p. 5]:

1. Firstly, previous obstruction-free algorithms made it impossible for metadata to be stored locally along with the memory objects. This is an undesirable performance-intensive detour that can be solved by using locks.
2. Secondly, if the number of transactions exceeds the number of available processor cores, context switching becomes more frequent because due to the obstruction freedom, no thread will ever wait for another thread to complete. If context switching becomes too excessive, overall performance will decrease unnecessarily.

4.4 Ennals' lock-based STM algorithm

The STM algorithm developed by Ennals uses this idea that obstruction-freedom is unnecessary [4, pp. 6-7]. Instead, it introduces a locking mechanism that locks memory locations in the order they are encountered in the code. In case of a collision, changes are reverted and the locks are released.

Each object has a version counter that is incremented on every update to the object. Reading transactions log the current version number of each object and

verify at commit time if the counter has been incremented since the last read. If this is the case, the transaction is aborted.

The Transactional Locking (TL) algorithm described in [5], takes this approach a little further and includes several optimisations, some of which will be described in the following paragraphs.

4.5 Further optimization approaches: The TL algorithm

In their paper called “What really makes transactions faster” [5], Dave Dice and Nir Shavit, introduce an algorithm called TL (*Transactional Locking*). They describe a commit-time locking scheme in addition to the encounter-order locking. The approach suggests to lock the memory locations to be written only at commit time and verify just instants before the commit that read locations have not been changed. Right after writing the changes, the locks are released and the global version clock is incremented. This way locks are only held for a very short amount of time and global consistency is upheld. This differs strongly from the initial obstruction-free approach by Ennals [4] where locks are acquired up front in the order they are encountered.

For obvious reasons, locks are held longer in encounter-mode than in commit-mode. In contention phases this can lead to multiple transactions aborting each other. In order to improve performance nevertheless, this variation of the TL algorithm introduces a *bounded spin and back-off delay* similar to CSMA-CD⁶ multiple access algorithms [4]. The time between retries is calculated using a random component in order to allocate processing power using a fair method.

5 The TL2 algorithm

As previously described, lock-based STM algorithms work faster than lock-free ones. However, Dave Dice, Ori Shalev and Nir Shavit highlight two remaining limitations that they try to overcome with the new version of their TL algorithm called TL2. [6]

First of all, STM algorithms described until this point only work with closed memory systems which means that memory used transactionally can not be recycled non-transactionally.

Secondly, the described algorithms need *specialized managed runtime environments* in order to avoid inconsistent behaviour due to the fact that intermediate memory states may be inconsistent.

These downsides are solved by the TL2 algorithm which includes many optimisations compared to the original STM approach. It introduces a *global version clock* that is incremented on every write operation. This *global* version counter prevents a transaction from reading an inconsistent memory state.

⁶ CSMA-CD (*Carrier Sense Multiple Access with Collision Detection*) is a multiple access protocol that relies on aborting operation as soon as a collision is detected.

5.1 Locking schemes

The TL2 algorithm can operate with two different locking schemes. Locks can be acquired either on a per-object (PO) basis or on a per-stripe (PS) basis [6, p. 6]. The latter means that memory is partitioned (striped) and that locks are grouped in arrays.

The PO scheme has performance advantages because metadata is stored locally along with the objects. It does however need additional compiler support, whereas PS can work on unmodified data structures.

5.2 Operation

Writing transactions run through a complex process. The algorithm uses a commit-time locking mechanism as described in section 4.5:

1. At the start of a transaction, the global version counter is read so that it can later on be used to check if read memory locations have been modified.
2. Then, the transaction is executed internally without changing the memory. At the end of this *speculative* execution, the transaction has a list of read memory locations (*read-set*) and written locations (*write-set*). If any element from the read set has been modified after this step (i.e. if its version number is larger than the previously read version clock), the transaction is aborted.
3. The write-set is now being locked. If any of the locks can not be acquired, the transaction aborts.
4. After this, the global version clock is incremented.
5. The value of the version clock from the first step is used to validate if no of the elements from the read-set have been modified in the meantime. If either the validation fails or any of the locations in the read-set are locked by another transaction, the transaction fails as well.
6. Finally, the new values are written, the write-set will be marked with the newly incremented counter and the locks are released. With this step the transaction is committed.

Reading transactions are executed much faster. If there are no write operations, no locks need to be acquired and the version counter does not need to be incremented. Thus, only two steps are processed:

1. As with writing transactions, the global version counter is read.
2. Then, the transaction is executed speculatively. As before, it is verified at this point that the read-set is consistent.

6 Empirical Evaluation

Before coming to any empirical test case, one thing has to be said first. The test cases described in the following all point to a very clear end: STM are either

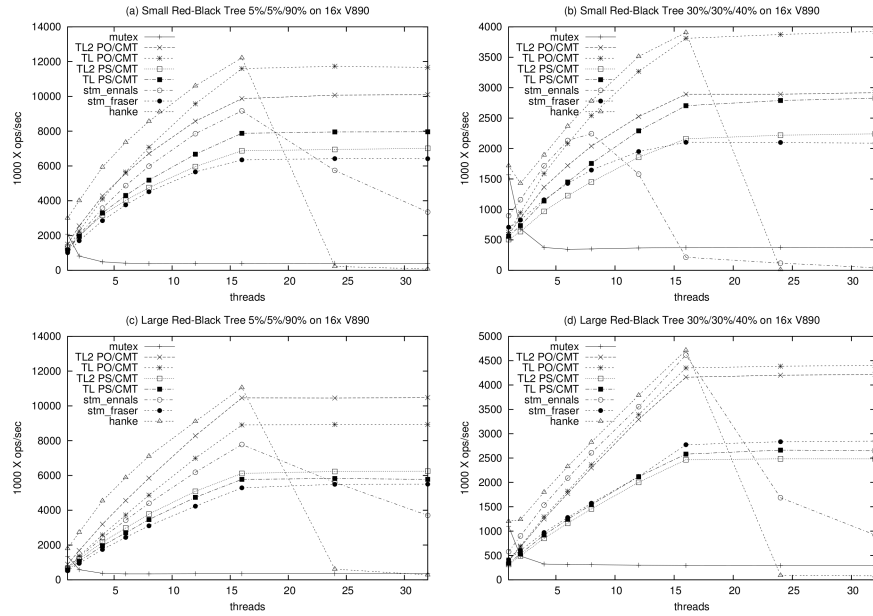


Fig. 2. Comparison of different STM algorithms. The benchmark includes 4 different test run and is executed on a 16-processor machine. [6]

faster or only slightly slower than lock based operations. However, as stated in section 2, it is still possible to define test scenarios where STMs are slower.

The benchmark depicted on figure 2 compares four different test runs. It is executed on a 16-processor machine running the Solaris operating system. It includes several different algorithms among which the following are interesting for this paper:

- **Mutex:** A coarse-grained locking algorithm for reference purpose.
- **stm_ennals:** The lock-based encounter-time algorithm from Ennals described in section 4.1.
- **TL:** A variation of the TL algorithm described in the section 4.5.
- **TL2:** The TL2 algorithm described in section 5. Two variations are tested: One on a per-object basis (PO) and one on a per-stripe (PS) basis.

As expected, all of the STM algorithms show much higher performance than the coarse grained locking algorithm in all of the four test runs. Most of the algorithms show excellent scaling capabilities below the 16 processor-threshold. After this threshold, the Ennals algorithm shows a fall-down⁷, which is an implementation specific disadvantage. The problem lies within the use of an encounter-time

⁷ This hold also for the Fraser algorithm, which has not been analysed in this paper. It will therefore not be considered in this evaluation.

locking process. Because of this, locks are being held longer and collisions become more frequent. This leads to perpetual retries and decreasing performance.⁸

Among the STM algorithms analysed in this paper, the TL and TL2 algorithms seem to be generally faster. However, small variations can be noticed depending on the test case. Those variations scale by a static factor, which is probably due to the different overheads of each individual algorithm [6]. For both algorithms, the PO variations are faster than the PS ones.

7 Conclusion and Outlook

By now, it should be clear that STMs are more straightforward for developers than using locks. The downside is of course that the mechanisms behind the scene are much more complex, which is why transactional memory is a field that has undergone many research efforts and will probably undergo more research efforts in the near future.

The empirical evaluation from the previous section as well as the rising number of implementations⁹ lead to the idea that STMs are coming closer to being production ready. A few issues will still have to be resolved though [7]:

First of all, legacy systems will still need to work alongside STM-based systems. Developers and investors would probably like to profit from the advantages of STM, but legacy systems can not be deprecated because of this. This is why backwards compatibility has to be ensured.

Beside this, tool support (e.g. for debugging) needs to be pushed. This is an important criterion for transactional memory implementations to be accepted by the market.

In order to decrease overhead and improve performance, research groups are currently focussing on hybrid solutions combining Software Transactional Memory (STM) and Hardware Transactional Memory (HTM). This way, applications can run on existing non-HTM processor without changes and can run faster on new processors supporting HTM operations.

It will probably take a few more years before transactional memory, be it software-only or hardware-supported, reaches the end-user market. But the recent rise of research efforts in this domain is a clear sign towards the industry that the general interest in the subject is rising.

References

1. Tom Knight: System and method for parallel processing with mostly functional languages. Patent number: 4825360. <http://www.google.com/patents?id=L3QWAAAAEBAJ>
2. Nir Shavit, Dan Touitou: Software Transactional Memory. *14th ACM Symposium on Principles of Distributed Computing (1995)*

⁸ As described in section 4.5, this is resolved by the TL algorithm.

⁹ Please refer to section 3

3. Tim Harris, Keir Fraser: Language Support for Lightweight Transactions. *Object-Oriented Programming, Systems, Languages, and Applications (October 2003)*
4. Robert Ennals: Software Transactional Memory Should Not Be Obstruction-Free
5. Dave Dice, Nir Shavit: What Really Makes Transactions Faster? *Proceedings of the 1st TRANSACT 2006 workshop (2006)*
6. Dave Dice, Ori Shalev, Nir Shavit: Transactional Locking II. *Proceedings of the 20th International Symposium on Distributed Computing (2006)*
7. Ali-Reza Adl-Tabatabai, Christos Kozyrakis, Bratin Saha: Unlocking concurrency. *ACM Queue* 4, 10 (December 2006)
8. Derrick Coetzee: Software transactional memory. <http://blogs.msdn.com/devdev/archive/2005/10/20/483247.aspx> (October 20, 2005)