Software Transaction Memory

An Overview

Jean Lucas Ferreira

November 2017

Contents

1 Introduction

This document goes over some of the shortcomings of lock-based concurrency control, and how software transactional memory addresses some of these issues. Additionally, it will provide a quick overview of transactional memory, and some design decisions for software transactional memory. Furthermore, it will give an introduction to Clojure, a powerful programming language that aims to facilitate programming concurrent systems through the use of software transactional memory. Finally, we will conclude with some discussions.

2 Lock-Based Programs

When first introduced to programming multi-threaded programs, most students are taught to handle critical sections and shared-data access by following a lock-based approach. Usually this is done through using locks explicitly, conditional variables, semaphores, or even non-trivial algorithms, such as Peterson's algorithm, depending on the desired behaviour of the given system.

Additionally, when developing data structures for mulit-threaded programs, we must decide if we wish to follow and coarse-grain or fined-grain locking locking. Coarse-grain locking is easy to implement, but provide little concurrency, whereas fined-grain locking is meant to provide greater concurrency performance but are difficult to implement [1].

These approaches provide, more than often, the desired behaviour needed to protect critical sections, and manage shared-data access. The downside of lock-based algorithms, is that they require special care in order to avoid common pitfalls such as deadlocks, or inconsistent data, and in more rare cases, priority inversions or convoying. As a program's complexity grows, management of the locks became more difficult to maintain. Furthermore, these programs are not very modular, and are designed around the specific problem at hand [2, 1].

3 Transactions

Before discussing the behaviour of Transactional Memory, it is important to understand the concept, and behaviour of transactions.

Transactions are defined to be a specific block of code that is to be executed by a single thread. The finite set of instructions within this block of code can access and update its own locally defined variables, as well as shared-data outside of its block. Where the shared-data outside of the block can be accessed by other concurrently executing transactions [4].

In order to reason upon the correctness of transactions, it is critical that transactions always maintain the following properties [5, 2]:

- *Serializable*: The instructions within a transaction does not appear to overlap with the instructions of another concurrently executing transaction.
- *Atomic*: Each transaction is either seen as completed, or to not have been executed at all.
- *Consistent*: Transactions always leave the program in a valid state.
- *Isolated*: Transactions do not cause side-effects. The effects of one active transaction, does not affect another actively executing transaction.

4 Transactional Memory - Background

The concept of Transactional Memory is noted to be first defined in 1986, by Tom Knight, as an approach to verify the correctness of concurrent programs in Lisp [3]. Although this was initially intended to be implemented in the hardware level (HTM). By the mid 1990s, the concept was applied to a software level as a mechanism for lock-free data structures [2], and later the term Software Transactional Memory was introduced by [4], as a new way to provide synchronization in concurrent programs.

5 Overview of Transactional Memory

Transactional memory provide transactions with mechanisms to keep track of its changes, and be able to tell if the changes of a transaction will cause conflict with the changes of another transaction. This is done by having a read-set and write-set for each transaction, and allowing transactions to commit or abort (rollback) all its changes [6, 2].

When a transaction is executed it creates a read-set and a write-set. A read-set is a list of read instructions made to shared-memory by that transaction, similarly, a write-set is a list of tentative changes that transaction is wishes to make on some shared memory.

A transaction will be allowed to commit its changes only if its read-set and write-set do not cause any conflicts with another transaction's read and write sets. When a commit is made by a transaction, its changes will be visible to the rest of the program.

If it is the case that a transactions read and write set conflicts with another transaction's read and write set, then the transaction is aborted. Meaning that all its local, and tentative changes are undone, and the transaction is re-executed from the beginning.

6 Software Transactional Memory (STM)

Software transactional memory provide a programmer with the ability to use the transactional model for dealing with shared-data access in a multi-threaded program. When using an STM, the programmer is not expected to deal with transaction commits and rollbacks, or verifying that the transactions are following all the necessary properties. This will be handled by the underlining transactional memory.
STM provides [7, 5]:

- Abstraction of lock complexity.
- Removes the burden on having to verify if the locking protocol is correct.
- Allows for modular design of components.

It is important to note that STM does not guarantee a deadlock-free implementation, nor does it guarantee better performance.

7 Behaviour of STM

When creating an STM system there are important design decisions that must be made, concerning the expected behaviour. Some important topics to discussed below.

7.1 Optimistic vs Pessimistic Concurrency Control

Optimistic concurrency control would assume good things happen often, in this case, it assumes shared-data access conflicts are unlikely. Thus, deal with conflicts only once they arise. Where as a pessimistic concurrency control would assume it is likely conflicts will happen often thus it always declares exclusive access to the shared-data before entering. STMs are often implemented using an optimistic concurrency control scheme [5].

7.2 Weak vs Strong Isolation

This deals with how threads executing in a transaction interact with other threads that are not in another transaction. When an STM implements strong isolation, instructions in a transaction can not interleave with instruction from a non-transactional block, this is referred as strong atomicity. As transactions are seen to execute atomically from a non-transactional thread's point of view. Where weak isolation does not guarantee an atomic behaviour between interleaved transactional and non-transactional executions. It is difficult to design an STM that provides strong isolation, since it often requires special hardware, or it is difficult to provide it in main stream languages [8, 7, 5].

7.3 Lazy vs Eager Updates

Eager update (or in-place update) is a system where transactions are allowed to modify the shared-data directly, even before it commits. But in order to avoid data inconsistency from two transactions updating the shared-data, contention managers are used to govern the updates made to each shared-data by different transactions. If a contention manager detects a conflict from a transaction updating the shared-data, then the transaction consults its undo-log to revert the changes. In a system that implements lazy updates, then transactions perform operations on a local copy of the shared-data. The update is only applied to the original shared-data if the transaction is allowed to commit. When designing an STM if we can predict that commits will happen more often than aborts, then it is more efficient to implement eager updates. Similarly, if it is expected that transactions will abort often, then a lazy update system will be more efficient [7, 5].

7.4 Contention Management

A contention manager allows the STM to make conflict resolutions when an conflict occurs between two or more transactions. A conflict resolution will state whether a given transaction should continue, and commit, if a transaction should pause its executions and wait for another transaction to complete, or if a transaction should abort.
In order to come up with such decisions, the contention manager consults the policies that are defined by the implemented STM system. Such policies can be based on priority, work completion, time, or any scheme that can guarantee that a transaction eventually completes (commits or aborts) [7, 5, 1].

7.5 Opacity

Is a correctness criteria that tries to ensure every active transaction only sees a valid system state. We must not allow active, or zombie transactions to ever execute in an inconsistent state. This can lead the system into a faulty state , such as entering an infinite loop or accessing null values. Zombie transactions are those transactions that have caused some type of conflict but have not yet halted. Zombie transactions will always abort [7, 5, 1, 9].

8 Clojure - Programming Language

8.1 Overview

Clojure is a dynamic general-purpose programming language that belongs to that Lisp language family. Additionally the Clojure compiler produces bytecode that can be interpreted by the JVM, thus allowing Clojure to share many of Java's interface and classes. Other notable features of Clojure is that it provides a functional programming paradigm, and it provides concurrent programming without the need of manual locks, as it has a built in STM system.[10]

8.2 Concurrency in Clojure

In Clojure, the core data structures are immutable, this allows for threads to easily share common data structures in a safe manner. When it is necessary for threads to update the state of the system in a synchronized manner, it can be done so without the need for manual locking. This is a result of Clojure's built-in STM system, that provides mechanisms for managing threads.

8.3 STM in Clojure

In Clojure, the STM is implemented with multi-version concurrency control and snapshot isolation. MVCC removes lock contention in a multiple transactions (doing reads and writes) by providing a snapshot of the current state of the system to each transaction. Because transactions are isolated, the changes done by a transaction will not be visible until it is committed (recall that it will only be allowed to commit if it will not induce any conflicts).

Refs (transactional references) in Clojure allow threads to safely share a mutable memory location. The underlining STM system restricts that the storage location bounded by the Ref is only allowed to update, if the instruction happens within a transaction. As discussed earlier, that transactions must be Atomic, Consistent and Isolated, all Refs in Clojure must also follow these three properties.

A transaction may make changes to Ref through the commands ref-set, alter, and commute. Each of these must be called within the *dosync* body, and they describe what constitute as a safe/valid change of a *Ref* [11].

```
(ref-set <ref-name> <value-to-set>)
(commute <ref-name> <update-function>)
(alter <ref-name> <update-function>)
```

- `ref-set`: Set a new value for the Ref. This is not used for updating the Ref.
- `commute`: Used for updating a Ref, where order does not matter, for example, increasing a counter
- `alter`: A more strict update than *commute*, it is used when the *ordering* of the updates occur. For example, performing different operations on a single value, where the operations are not *commutative.*

The important distinction between alter and commute is that commute will *not* retry if its Ref value is changed. Where as alter will always retry, if its Ref has been updated by another transaction. In order for this to be possible, there exists some additional background work by the STM when commute is used.

It is important to not that Clojure provides other constructs to deal with synchronous and asynchronous updates. Such as `Atom`, `Vars`, and `Agent`, these do not provide the attribute of *shared* updates as Ref does, but each have their own purpose, and will not be discussed in this document.

8.4 Example in Clojure

The following is a simple example of dealing with updates of a bank account, idea for this example has been derived from [12]. This program will start 5 threads, that will execute the `my_update` function, where it uses a transaction to update the ref variable `counter` through a `commute` operation.

```
(def counter (ref 0))

(defn my_update []
        (dosync
                 ;(ref-set counter 2 )
                 ;(alter   counter inc )
                 (commute counter inc)))

(defn run_test []
  (let [threads (for [x (range 0 5)]
    (Thread. #(my_update)))]
    (do
      (doall (map #(.start %) threads))
      (doall (map #(.join %) threads)))))
```

9 Discussion of STM

Although STM provides a simplification in programming concurrent systems, it has some shortcomings. These are some of the results:

- Trying to translate directly from a lock-based approach to a transactional approach can introduce deadlocks[8]
- If a programmer is not familiar with the STM system in use, undesired results may occur (ie: strong atomicity vs weak atomicity) [8]
- TxLinux Project: A project that is attempting to replace most of the locks in the Linux kernel with transactions [7]
- Swapping critical sections for transactions resulted in decrease concurrent performance (Berkeley DB lock manager) [7]

- [13, 14] conducted studies, and concluded that most students found STM easier to implement and understand (compared to fine-grained locking), produced fewer implementation errors, but was harder to fine tune performance

References

[1] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[2] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[3] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, Feb 1997.

[4] Sabri Pllana and Fatos Xhafa. *Programming Multi-core and Many-core Computing Systems*. Wiley Publishing, 1st edition, 2014.

[5] Tom Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 105–112, New York, NY, USA, 1986. ACM.

[6] Bartosz Milewski. Beyond locks: Software transactional memory, September 2010.

[7] *Transactional Memory Today*. Springer, Berlin, Heidelberg, 2010.

[8] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17–17, July 2006.

[9] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[10] Rich Hickey. https://clojure.org/about/rationale.

[11] R. Mark Volkmann. Clojure - functional programming for the jvm. https://objectcomputing.com/resources/publications/sett/march-2009-clojure-functional-programming-for-the-jvm/Concurrency.

[12] Konrad Garus. Software transactional memory in clojure (alter vs. commute). http://squirrel.pl/blog/2010/07/13/clojure-alter-vs-commute/.

[13] Victor Pankratius and Ali-Reza Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 43–52, New York, NY, USA, 2011. ACM.

[14] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? *SIGPLAN Not.*, 45(5):47–56, January 2010.