Concurrent Datastructure Design for Software Transactional Memory

Bachelor Project

Daniel Köves

Supervisor: Prof. Wan Fokkink
Second Reader: Dr. Thilo Kielmann

Vrije Universiteit Amsterdam

June 2021

Outline

Outline

Introduction

In the multi- and many-core area, we need *parallelisation* to fully utilise available machines.

Definition

Parallel computing is a programming paradigm in which multiple parallel processes or *threads* execute at the same period of time.

Parallel computing is used to speed up computation on a multi-core system.

Synchronisation

Synchronisation refers to how concurrent threads manage and operate on shared data.

Concurrent execution should not:

- Overwrite others
- See inconsistent states

Possible ways to achieve synchronisation: locks, lock-free primitives, transactions

Synchronisation

Synchronisation refers to how concurrent threads manage and operate on shared data.

Concurrent execution should not:

- Overwrite others
- See inconsistent states

Possible ways to achieve synchronisation: locks, lock-free primitives, transactions

Synchronisation

Synchronisation refers to how concurrent threads manage and operate on shared data.

Concurrent execution should not:

- Overwrite others
- See inconsistent states

Possible ways to achieve synchronisation: locks, lock-free primitives, transactions

Locking

Locks provide a way to achieve *mutual exclusion* by guarding *critical sections* – a block of code only a single thread is allowed to execute at a time.

Locks provide (at least) two operations: `lock()` and `unlock()`

Programming with locks is generally hard and difficult to debug

Other problems with locking: priority inversion, convoying, deadlock, etc.

Locking

Locks provide a way to achieve *mutual exclusion* by guarding *critical sections* – a block of code only a single thread is allowed to execute at a time.

Locks provide (at least) two operations: `lock()` and `unlock()`

Programming with locks is generally hard and difficult to debug

Other problems with locking: priority inversion, convoying, deadlock, etc.

Locking

Locks provide a way to achieve *mutual exclusion* by guarding *critical sections* – a block of code only a single thread is allowed to execute at a time.

Locks provide (at least) two operations: `lock()` and `unlock()`

Programming with locks is generally hard and difficult to debug

Other problems with locking: priority inversion, convoying, deadlock, etc.

Locking

Locks provide a way to achieve *mutual exclusion* by guarding *critical sections* – a block of code only a single thread is allowed to execute at a time.

Locks provide (at least) two operations: `lock()` and `unlock()`

Programming with locks is generally hard and difficult to debug

Other problems with locking: priority inversion, convoying, deadlock, etc.

Lock-Free Primitives

Lock-free primitives provide safe access the shared data without the use of locks.

Practical if hardware-assisted, like Compare-and-Swap (Intel) or Load-link/Store-conditional (ARM)

Compare-and-Swap(A, E, V)

Execute atomically the following: if the value at A is E then set it to V and return true, else do nothing and return false.

Load-link/Store-conditional

LL instruction loads the value at address A. Subsequent SC call with some value V succeeds only if the value of A has not changed since.

Lock-Free Primitives

Lock-free primitives provide safe access the shared data without the use of locks.
Practical if hardware-assisted, like Compare-and-Swap (Intel) or Load-link/Store-conditional (ARM)

Compare-and-Swap(A, E, V)

Execute atomically the following: if the value at A is E then set it to V and return true, else do nothing and return false.

Load-link/Store-conditional

LL instruction loads the value at address A. Subsequent SC call with some value V succeeds only if the value of A has not changed since.

Lock-Free Primitives

Lock-free primitives provide safe access the shared data without the use of locks.
Practical if hardware-assisted, like Compare-and-Swap (Intel) or Load-link/Store-conditional (ARM)

Compare-and-Swap(A, E, V)

Execute atomically the following: if the value at A is E then set it to V and return true, else do nothing and return false.

Load-link/Store-conditional

LL instruction loads the value at address A. Subsequent SC call with some value V succeeds only if the value of A has not changed since.

Lock-Free Primitives

Lock-free primitives provide safe access the shared data without the use of locks.
Practical if hardware-assisted, like Compare-and-Swap (Intel) or Load-link/Store-conditional (ARM)

Compare-and-Swap(A, E, V)

Execute atomically the following: if the value at A is E then set it to V and return true, else do nothing and return false.

Load-link/Store-conditional

`LL` instruction loads the value at address A. Subsequent `SC` call with some value V succeeds only if the value of A has not changed since.

Lock-Free Primitives

Limitations of lock-free primitives:

- Usually operate on a single word in memory
- Complex lock-free algorithms tend to have an unnatural structure

Lock-Free Primitives

Limitations of lock-free primitives:

- Usually operate on a single word in memory
- Complex lock-free algorithms tend to have an unnatural structure

Lock-Free Primitives

Limitations of lock-free primitives:

- Usually operate on a single word in memory
- Complex lock-free algorithms tend to have an unnatural structure

Transactional Memory

Introduced by Herlihy and Moss in 1993 [1]

Definition

Transactions consist of a set of instructions and are *atomic* and *serializable*.

Transactions make speculative changes to memory which they make atomically visible upon *committing*.

If an inconsistent state is encountered, the transaction *aborts* and can be retried.

Transactional Memory

Introduced by Herlihy and Moss in 1993 [1]

Definition

Transactions consist of a set of instructions and are *atomic* and *serializable*.

Transactions make speculative changes to memory which they make atomically visible upon *committing*.

If an inconsistent state is encountered, the transaction *aborts* and can be retried.

Transactional Memory

Introduced by Herlihy and Moss in 1993 [1]

Definition

Transactions consist of a set of instructions and are *atomic* and *serializable*.

Transactions make speculative changes to memory which they make atomically visible upon *committing*.

If an inconsistent state is encountered, the transaction *aborts* and can be retried.

Transactional Memory

Introduced by Herlihy and Moss in 1993 [1]

Definition

Transactions consist of a set of instructions and are *atomic* and *serializable*.

Transactions make speculative changes to memory which they make atomically visible upon *committing*.

If an inconsistent state is encountered, the transaction *aborts* and can be retried.

Transactional Programming

Provides a straightforward abstraction to deal with concurrency:

Transactional Programming

- While transaction is not done:
- Perform actions on shared data
- If inconsistent state is found:
- Abort transaction and retry

Transactional Programming

Provides a straightforward abstraction to deal with concurrency:

Transactional Programming

- While transaction is not done:
- Perform actions on shared data
- If inconsistent state is found:
- Abort transaction and retry

Thesis Goals

- Compare the different variants of Software Transactional Memory
- Investigate how transactional programming can be applied to concurrent datastructure design

Research Questions

- How does the locking scheme of lock-based STM implementations affect the insertion performance of concurrent Red-Black Trees and Skiplists?
- How well suited is transactional programming for concurrent datastructures in terms of ease-of-design?

Thesis Goals

- Compare the different variants of Software Transactional Memory
- Investigate how transactional programming can be applied to concurrent datastructure design

Research Questions

- How does the locking scheme of lock-based STM implementations affect the insertion performance of concurrent Red-Black Trees and Skiplists?
- How well suited is transactional programming for concurrent datastructures in terms of ease-of-design?

Outline

Transactional Memory Variants

Most Transactional Memory implementations make use of hardware-assist and are mostly implemented in software.

In this thesis, we focus on Software Transactional Memory introduced by Shavit and Touitou in 1996 [2]

Two main approaches: *non-blocking* and *blocking* STM

Transactional Memory Variants

Most Transactional Memory implementations make use of hardware-assist and are mostly implemented in software.

In this thesis, we focus on Software Transactional Memory introduced by Shavit and Touitou in 1996 [2]

Two main approaches: *non-blocking* and *blocking* STM

Transactional Memory Variants

Most Transactional Memory implementations make use of hardware-assist and are mostly implemented in software.

In this thesis, we focus on Software Transactional Memory introduced by Shavit and Touitou in 1996 [2]

Two main approaches: *non-blocking* and *blocking* STM

Non-Blocking STM

Early STM implementations were *non-blocking*.

Definition

An algorithm is **non-blocking** if the delay of one thread does not stop others from making progress.

Non-blocking algorithms utilise lock-free primitives to achieve synchronisation.

Non-Blocking STM

Early STM implementations were *non-blocking*.

Definition

An algorithm is **non-blocking** if the delay of one thread does not stop others from making progress.

Non-blocking algorithms utilise lock-free primitives to achieve synchronisation.

Non-Blocking STM

Early STM implementations were *non-blocking*.

Definition

An algorithm is **non-blocking** if the delay of one thread does not stop others from making progress.

Non-blocking algorithms utilise lock-free primitives to achieve synchronisation.

Blocking STM

- In 2006, Robert Ennals suggested [3] that the non-blocking property is detrimental to performance.
- Following that, attention shifted to *blocking* STM implementations.

Definition

An algorithm is **blocking** if the delay of one thread prevents others from progressing.

Blocking algorithms utilise locks to achieve synchronisation.

Blocking STM

- In 2006, Robert Ennals suggested [3] that the non-blocking property is detrimental to performance.
- Following that, attention shifted to *blocking* STM implementations.

Definition

An algorithm is **blocking** if the delay of one thread prevents others from progressing.

Blocking algorithms utilise locks to achieve synchronisation.

Blocking STM

- In 2006, Robert Ennals suggested [3] that the non-blocking property is detrimental to performance.
- Following that, attention shifted to *blocking* STM implementations.

Definition

An algorithm is **blocking** if the delay of one thread prevents others from progressing.

Blocking algorithms utilise locks to achieve synchronisation.

Blocking STM

- Blocking (lock-based) STM implementations lock transactional objects when they wish to modify them
- There are two main approaches on how to do that:

Definition

Encounter-order transactions lock object as they are encountered.

Definition

Commit-time transactions lock object only at commit time, and make tentative changes to memory before that.

Blocking STM

- Blocking (lock-based) STM implementations lock transactional objects when they wish to modify them
- There are two main approaches on how to do that:

Definition

Encounter-order transactions lock object as they are encountered.

Definition

Commit-time transactions lock object only at commit time, and make tentative changes to memory before that.

Blocking STM

- Blocking (lock-based) STM implementations lock transactional objects when they wish to modify them
- There are two main approaches on how to do that:

Definition

Encounter-order transactions lock object as they are encountered.

Definition

Commit-time transactions lock object only at commit time, and make tentative changes to memory before that.

Transactional Programming: Interface

```
template<class T>
struct Transaction {
    virtual void begin()          = 0;
    virtual void write(T *, T)    = 0;
    virtual T    read(T *)        = 0;
    virtual bool commit()         = 0;
    virtual void abort()          = 0;
};
```

Transactional Programming: Example

```
Transaction Tx;
bool done = false;
while (!done) {
    try {
        Tx.begin();

        /* atomic block */

        done = Tx.commit();
    }
    catch (AbortException&) {
        Tx.abort();
        done = false;
    }
}
```

Outline

Transactional Datastructures

- Two transactional datastructures, a Red-Black Tree and a Skiplist have been implemented
- With underlying encounter-order and commit-time transactions taking care of concurrent insertions
- Commit-time transactions cannot successfully be applied to Red-Black Trees, as there are direct dependencies between the transactional writes
- Such restrictions do not exist for Skiplists
- However, the transactional insertion algorithm for both is much simpler than the lock-based or lock-free approaches

Transactional Datastructures

- Two transactional datastructures, a Red-Black Tree and a Skiplist have been implemented
- With underlying encounter-order and commit-time transactions taking care of concurrent insertions
- Commit-time transactions cannot successfully be applied to Red-Black Trees, as there are direct dependencies between the transactional writes
- Such restrictions do not exist for Skiplists
- However, the transactional insertion algorithm for both is much simpler than the lock-based or lock-free approaches

Transactional Datastructures

- Two transactional datastructures, a Red-Black Tree and a Skiplist have been implemented
- With underlying encounter-order and commit-time transactions taking care of concurrent insertions
- Commit-time transactions cannot successfully be applied to Red-Black Trees, as there are direct dependencies between the transactional writes
- Such restrictions do not exist for Skiplists
- However, the transactional insertion algorithm for both is much simpler than the lock-based or lock-free approaches

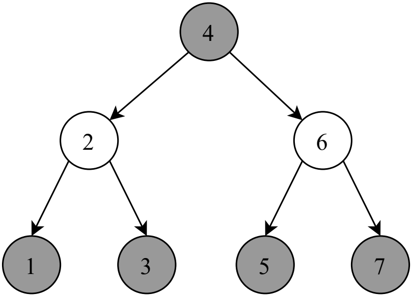
Transactional Datastructures

- Two transactional datastructures, a Red-Black Tree and a Skiplist have been implemented
- With underlying encounter-order and commit-time transactions taking care of concurrent insertions
- Commit-time transactions cannot successfully be applied to Red-Black Trees, as there are direct dependencies between the transactional writes
- Such restrictions do not exist for Skiplists
- However, the transactional insertion algorithm for both is much simpler than the lock-based or lock-free approaches

Transactional Datastructures

- Two transactional datastructures, a Red-Black Tree and a Skiplist have been implemented
- With underlying encounter-order and commit-time transactions taking care of concurrent insertions
- Commit-time transactions cannot successfully be applied to Red-Black Trees, as there are direct dependencies between the transactional writes
- Such restrictions do not exist for Skiplists
- However, the transactional insertion algorithm for both is much simpler than the lock-based or lock-free approaches

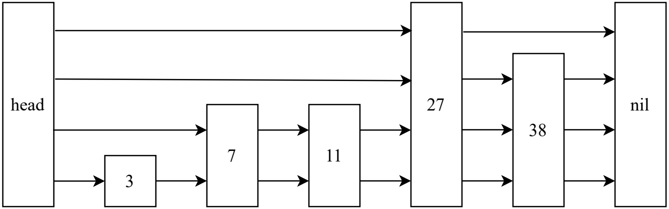
Red-Black Trees



Properties of Red-Black Trees:

- Every node is either red or black
- The root and leaf nodes are black
- Every red node's children must be black
- From each node to its descendant leaves, all paths contain the same number of black nodes

Skiplists



Properties of Skiplists:

- Each node has a certain height h chosen with probability p
- Nodes are organised into levels of linked lists
- Maximum level L is bounded by the number of elements N

Outline

Evaluation

In order to evaluate the performance of the lock-based STM variants, the following metrics are measured:

- Time of 10k insertions into the datastructures
- Abort rate of the transactional insertion
- Speedup per thread compared to the sequential execution
- Relative execution time of the transactional API operations

The tests were run on 1, 2, 4, 8 and 16 threads respectively on the DAS5 [4]

Evaluation

In order to evaluate the performance of the lock-based STM variants, the following metrics are measured:

- Time of 10k insertions into the datastructures
- Abort rate of the transactional insertion
- Speedup per thread compared to the sequential execution
- Relative execution time of the transactional API operations

The tests were run on 1, 2, 4, 8 and 16 threads respectively on the DAS5 [4]

Evaluation

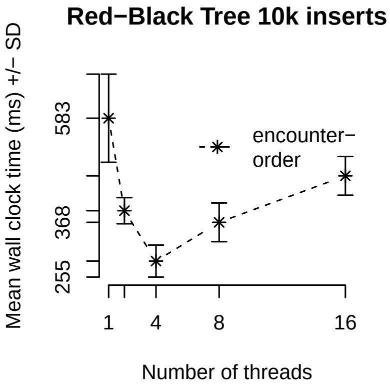
In order to evaluate the performance of the lock-based STM variants, the following metrics are measured:

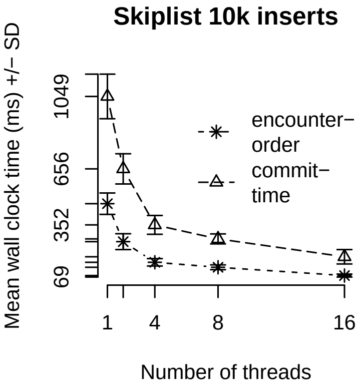
- Time of 10k insertions into the datastructures
- Abort rate of the transactional insertion
- Speedup per thread compared to the sequential execution
- Relative execution time of the transactional API operations

The tests were run on 1, 2, 4, 8 and 16 threads respectively on the DAS5 [4]

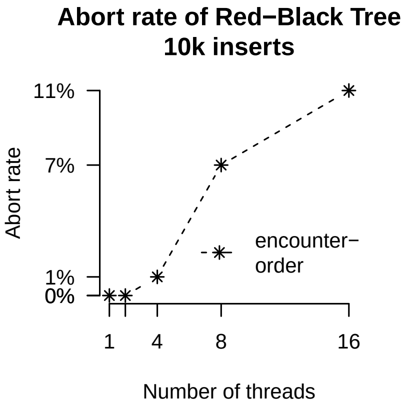
Outline

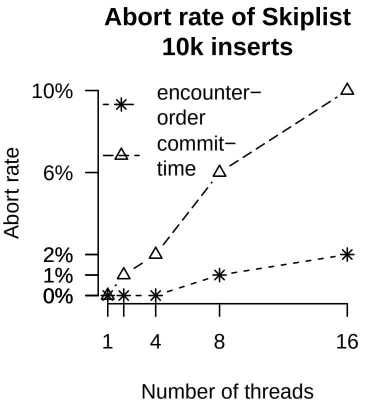
10k Insertions



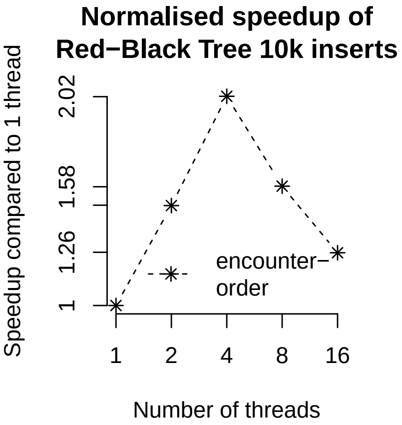


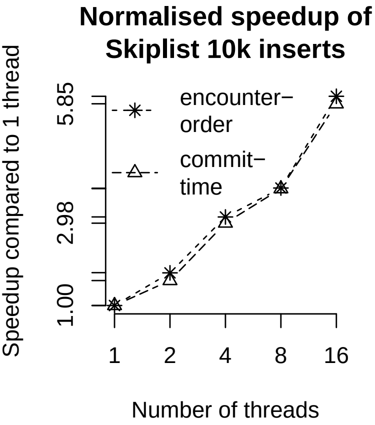
Abort Rate of 10k Insertions



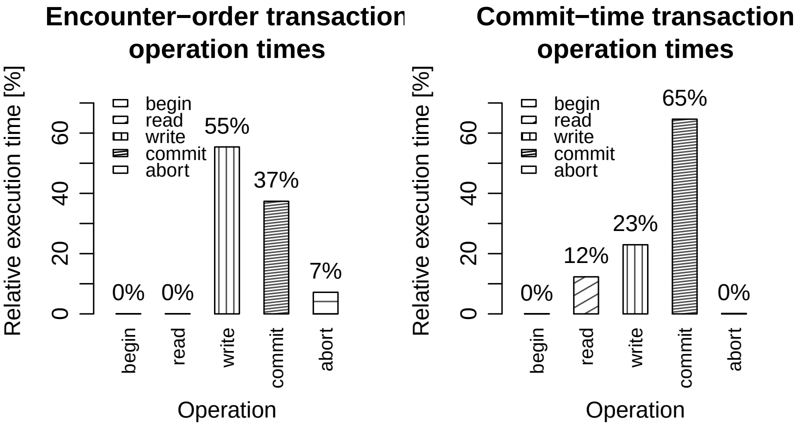


Speedup


Normalised speedup of
Red−Black Tree 10k inserts
Speedup compared to 1 thread
1
1.26
1.58
2.02
1
2
4
8
16
encounter−
order
Number of threads


Normalised speedup of
Skiplist 10k inserts
Speedup compared to 1 thread
1.00
2.98
5.85
1
2
4
8
16
encounter−
order
commit−
time
Number of threads

Operation Times



Outline

Conclusion

- In this thesis the transactional approach for concurrent datastructure design was investigated
- For Red-Black Trees, the application of commit-time transactions proved unsuccessful
- For Skiplists, both encounter-order and commit-time transactions had optimal scaling properties
- As opposed to other papers like [5-6], encounter-order transactions outperformed commit-time transactions by a factor of two on all metrics
- Limitations of transactional design is its performance
- Further research can investigate other common datastructures

References

[1] Maurice Herlihy and J. Eliot B. Moss. “Transactional Memory: Architectural Support for Lock-Free Data Structures”. In:Proceedings ofthe 20th Annual International Symposium on Computer Architecture.ISCA ’93. San Diego, California, USA: Association for Computing Ma-chinery, 1993, pp. 289–300

[2] Nir Shavit and Dan Touitou. “Software Transactional Memory”. In:Proceedings of the Fourteenth Annual ACM Symposium on Principlesof Distributed Computing. PODC ’95. Ottowa, Ontario, Canada: Asso-ciation for Computing Machinery, 1995, pp. 204–213.isbn: 0897917103.doi:10.1145/224964.224987

[3] Robert Ennals. “Software transactional memory should not be obstruction free”. In: In Intel Research Cambridge Tech Report. 2006. doi: 10.1.1.702.1468.

[4] Henri Bal et al. “A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term”. Computer 49.5 (2016), pp. 54–63. doi: 10.1109/MC.2016.127.

[5] David Dice and Nir Shavit. “What Really Makes Transactions Faster?” TRANSACT06 ACM Workshop (Jan. 2006).

[6] Dave Dice, Ori Shalev, and Nir Shavit. “Transactional Locking II”. In: DISC’06. Stockholm, Sweden: Springer-Verlag, 2006, pp. 194–208

Thank you for your attention!

Questions?