NESTING TRANSACTIONS OF MIXED FORMS

Sasha Bermeister
Dr Vincent Gramoli
School of Information Technologies
FACULTY OF ENGINEERING & INFORMATION TECHNOLOGIES
THE UNIVERSITY OF SYDNEY

INTRODUCTION

Software Transactional Memory (STM)

- Works in a similar way to *database transactions*
- A *transaction manager* intercepts reads and writes to memory, storing them in sets
- The transaction manager can *abort* two concurrent transactions if they are determined to *conflict*

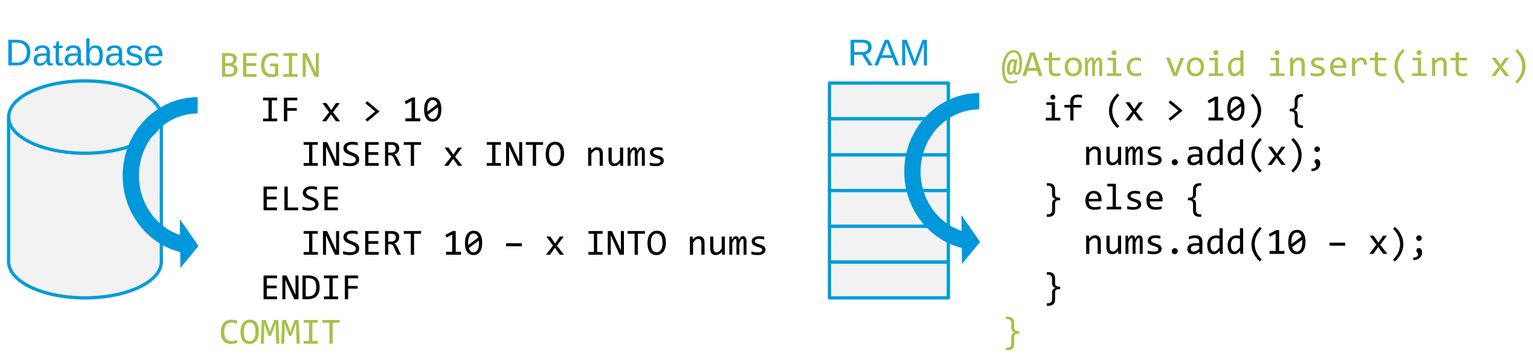


Fig 1. STM transactions are similar to database transactions, except they affect RAM and can be written in almost any programming language.

Nested Transactions

- If a transaction is created inside another transaction, that transaction is said to be *nested*
- Nested transactions are important for programmers as they allow *composition* (using functions from libraries)
- Without nested transactions, transactional memory would not have the benefit of allowing *reusable concurrent data structures*

Transactional Forms

- The *form* of a transaction determines which *abort algorithm* is used
- The two main forms of transactions are *normal* and *relaxed* transactions
- Depending on the data structure, a different form of transaction may be appropriate

```
class MatrixLibrary {
  @Atomic
  Matrix add(Matrix b);
}

@Atomic
Matrix solveFormula() {
  Matrix a = ...;
  Matrix b = ...;
  return a.add(b);
}
```

Fig 2. Nested transactions allow programmers to write their own transactions that use existing functions from libraries, without having to worry about whether they use transactions or not.

Normal Transactions	Relaxed Transactions
Guarantees *serializability* of the reads and writes	Guarantees *linearizability* of the object's operations
Strict conflict detection (low concurrency)	*Relaxed* conflict detection (high concurrency)
Guarantees consistency in *all* data structures	Only guarantees consistency in *some* data structures

Table 1. Comparing the properties of normal and relaxed-form transactions.

PROBLEM & MOTIVATION

Nesting mixed-form transactions

- Since programmers choose different forms of transactions, STM needs to ensure *correct semantics* when any combination of these are nested
- Without allowing nesting for mixed-form transactions, the *correctness* of the whole transaction could be compromised, as well as resulting in a *performance loss*

Current nesting implementations

- One of the most recent and popular implementations is *DeuceSTM* [2], an open-source Java library
- The only type of transactional nesting supported by DeuceSTM is *flat nesting*, which *ignores* the nested child transaction and treats it as a continuation of the parent
- This is comparable to not nesting the child transaction at all

Library method

```
class MatrixLibrary {
  @Atomic(metainf="normal")
  Matrix add(Matrix b);
}
```

Flat nesting

```
@Atomic(metainf="elastic")
Matrix solveFormula() {
  Matrix a = ...;
  return a.add(a);
}
```

Full-heritance nesting

```
@Atomic(metainf="elastic")
Matrix solveFormula() {
  Matrix a = ...;
  return c.add(c);
}
```

Fig 3. With flat nesting, the form of any nested transactions are ignored. In this case, the add() method will be run as an elastic transaction, even though it was defined as normal in its library. This could result in unexpected behavior, as although the data structure may maintain consistency with normal transactions, it may not with elastic transactions.

IMPLEMENTATION

A model for mixed-form nesting

- To allow for nesting of mixed-form transactions, some *communication* is needed between the nested transaction and its parent
- *Inheritance* is when the parent transaction sends the child transaction its read and write sets
- *Outheritance*[1] is when the child transaction sends its read and write sets back to the parent transaction
- Both are needed to achieve proper mixed-form transactional nesting – this is called *full-heritance nesting*

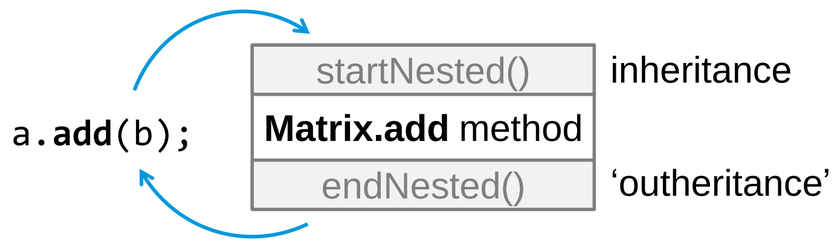
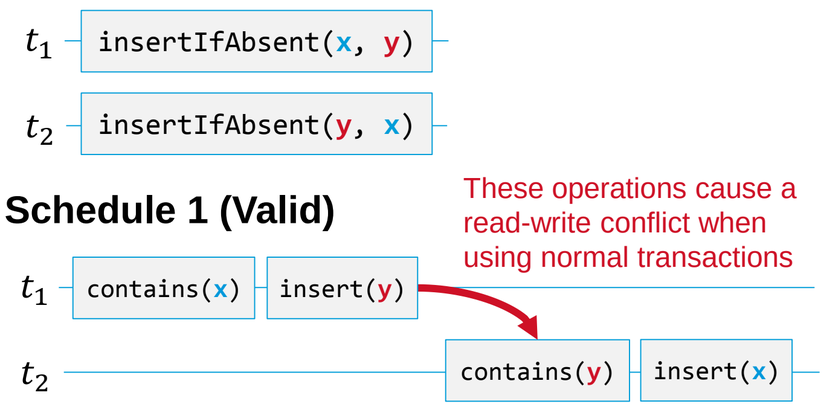


Fig 4. Inheritance and outheritance can be achieved by calling extra functions before and after the nested transaction is run.

RESULTS

Correctness & Performance Results

- To test full-heritance nesting, we implemented the *insertIfAbsent* method on a red-black tree
- InsertIfAbsent(x, y) will add y to the tree if x is not in the tree
- If implemented with flat-nested transactions, insertIfAbsent will have very low performance (with normal form) and will not perform correctly (with elastic form)
- When implemented with full-heritance nesting, insertIfAbsent maintains the consistency of the data structure without sacrificing efficiency



Schedule	Expected Result	Normal-form Flat-nesting	Elastic-form Flat-nesting	Full-heritance nesting
1	Commit	✘ Aborts	✓	✓
2	Abort	✓	✘ Commits	✓

Fig 5. To test full-heritance nesting, we ran two insertIfAbsent transactions in parallel with inverse arguments. During the test, we were able to observe the two schedules shown – both of which performed correctly with full-heritance nesting.

REFERENCES

1. V. Gramoli, R. Guerraoui, and M. Letia, "Composing Relaxed Transactions," in 27th IEEE International Parallel & Distributed Processing Symposium, 2013.
2. *Y. Afek et al., "The Velox Transactional Memory Stack," IEEE Micro, vol. 30, no. 5, pp. 76–87, Sep. 2010.*