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

Database

BEGIN
IF x > 10
INSERT x INTO nums
ELSE
INSERT 10 - x INTO nums
ENDIF
COMMIT

Database

Ram

BEGIN
IF x > 10
INSERT x INTO nums
ELSE
INSERT 10 - x INTO nums
ENDIF
COMMIT

Class MatrixLibrary

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

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

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

Nest 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

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.

Class MatrixLibrary

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

Current nesting implementations
- One of the most recent and popular implementations is DeuceSTM², 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
class MatrixLibrary {
@Atomic(metaInf="elastic")
Matrix solveFormula() {
Matrix a = ...;
return a.add(a);
}

Full-heritage nesting
class MatrixLibrary {
@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.

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

Library method
class MatrixLibrary {
@Atomic(metaInf="normal")
Matrix add(Matrix b);
}

Flat nesting
class MatrixLibrary {
@Atomic(metaInf="elastic")
Matrix solveFormula() {
Matrix a = ...;
return a.add(a);
}

Full-heritage nesting
class MatrixLibrary {
@Atomic(metaInf="elastic")
Matrix solveFormula() {
Matrix a = ...;
return c.add(c);
}

REFERENCES