Transactional Memory for Smalltalk

Lukas Renggli
Oscar Nierstrasz

Software Composition Group
University of Bern

Concurrent Programming

```
Semaphore forMutualExclusion

RecursionLock new

Mutex new
```

Problems

Deadlocks
Starvation
Priority Inversion

Complexity

Software Transactional Memory

Programming with Transactions

Lock Based

```
tree := BTree new.
lock := Semaphore forMutualExclusion.

" writing "
lock critical: [ tree at: #a put: 1 ].

" reading "
lock critical: [ tree at: #a ].
```

Transactional

```
tree := BTree new.

" writing "
[ tree at: #a put: 1 ] atomic.

" reading "
tree at: #a.
```

applied atomically

Inside Transactions

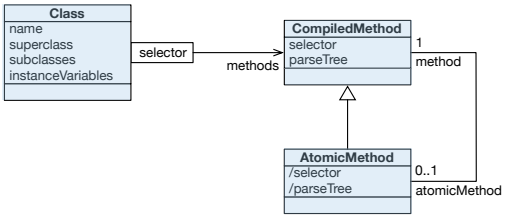
Implementation in Smalltalk

Lazy code transformation

Method annotations

Context dependent code execution

Static Model



Code Transformation

Message Sends

Instance Variables

Variable Bindings

Original Source Code

```
BTree>>at: aKey put: anObject
    | leaf |
    leaf := root
        leafForKey: aKey.
    leaf insertKey: aKey value: anObject.
    root := leaf root.
    ^ anObject
```

1. Message Sends

```
BTree>>__atomic__at: aKey put: anObject
    | leaf |
    leaf := root
        __atomic__leafForKey: aKey.
    leaf __atomic__insertKey: aKey value: anObject.
    root := leaf __atomic__root.
    ^ anObject
```

2. State Access

```
BTree>>__atomic__at: aKey put: anObject
    | leaf |
    leaf := (self atomicInstVarAt: 1)
        __atomic__leafForKey: aKey.
    leaf __atomic__insertKey: aKey value: anObject.
    self atomicInstVarAt: 1 put: leaf __atomic__root.
    ^ anObject
```

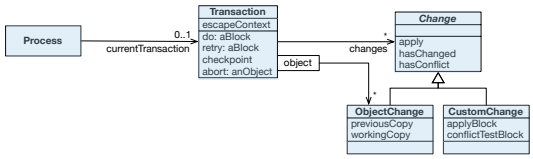
Code Transformation

Infrastructural code
Exception handling
Execution contexts

Many primitives
Variable sized objects

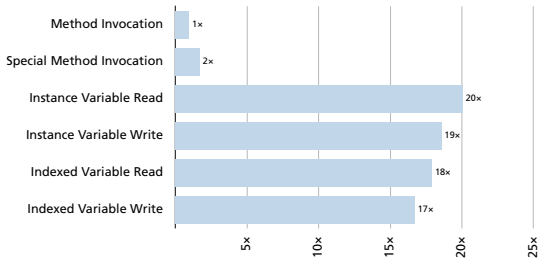
} Method Annotation

Dynamic Model

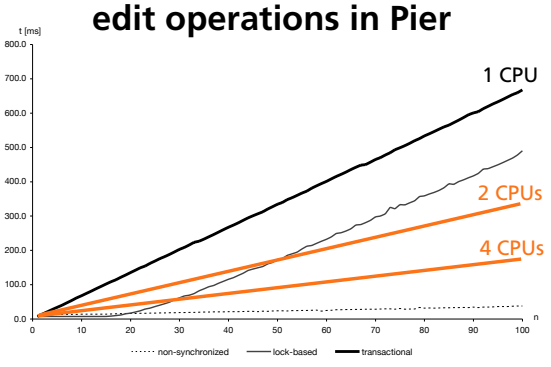


Benchmarks

Speed Comparison



Performance of *n* concurrent edit operations in Pier



Future Work

Implement within a multi-core environment

Improve speed

Applications

Concurrency Control

Source Code Loading

Context Oriented Programming