# Avoiding Publication and Privatization Problems on Software Transactional Memory

Holger Machens and Volker Turau

Frühjahrstreffen 2011 – GI/ITG Fachgruppe Betriebssysteme
8th April, 2011

Institute of Telematics
Hamburg University of Technology
TUHH

# Overview

1

# Aims and Objectives

# Move to Multi-Core/Multi-Processor

- Current challenge:
  - Performance boost achieved with concurrency only
  - But: Concurrent programming is still very complex
- Ongoing development:
  - Establishment of concurrent programming languages
  - Extension of the set of utilities for concurrency
- A proposed utility:
  - Transactional Memory (TM) to improve concurrency

# Transactional Memory

- Transactional Memory
  - Transactions in memory (CC with rollbacks)
  - Originally proposed as hardware extension
  - Common as Software Transactional Memory (STM)
- Promoted as a solution to known problems:
  - Deadlocks
  - Bad scalability
- Unfortunately has its very own problems
  - Blocking interferes with transaction mechanisms
  - Irrevocable actions are incompatible with rollbacks
  - Privatization causes data inconsistency
  - …

# Aims and Objectives

- Important aspects for the acceptance of new technologies
  - Learning expense
  - Programming effort
  - Legacy support
  - …
- Address problems using a programming model providing:
  - Seamless integration with a common programming language
  - Simple API
    - Transparent instrumentation
    - Default: Error prevention at acceptable performance
    - Low level optimization API for expert programmers
  - Support of modular software designs
  - No specific restrictions on TM mechanism
- **Focus here: Privatization and publication problem**

2

# State of the Art

# Privatization/Publication Problem

## Privatization of a List Element

| Initially: top.data = 6; top.next.data = 7 | | | |
|---|---|---|---|
| Thread T1 | | Thread T2 | |
| `atomic {` | 1 | | 1 |
| | 2 | `atomic {` | 2 |
| `  elem = list.top;` | 3 | `  elem = list.top;` | 3 |
| `  list.top = elem.next;` | 4 | | 4 |
| `}` | 5 | | 5 |
| `elem.data = 42;` | 6 | | 6 |
| | 7 | `  val = elem.data;` | 7 |
| | 8 | `}` | 8 |

- Can variable `val` get the value `42`?
- Fundamental problem: Unprotected shared variables
- **Known solution: Strong isolation**

# Strong Isolation

- **Weak Isolation:**
  - Isolation of transactions from each other
- **Strong Isolation:**
  - Isolation of transactions from each other
  - **+** Isolation of transactions from non-transactional operation

- Runtime-level realisation requires
  - Observation of r/w access
    - E.g. instrumentation of read/write instructions
    - Or modified runtime environment (e.g. JVM-TI agent)
  - Or single global lock semantics
    - Restrictions on TM implementation
    - Reduced concurrency

# Existing Programming Models

## Block-oriented

```
atomic {
   // ...
}
```

## Object-oriented

```
@TransactionalObject
class MyObject{
   @TransactionalMethod(Readonly)
   public void method(){
      // ...
   }
}
```

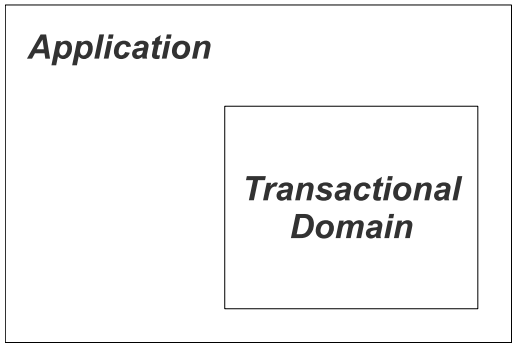
## Type-level enforcement of strong isolation

```
atomic {
   stm_operation()
   stm_operation()
   ...
   stm_operation()
}
```

3

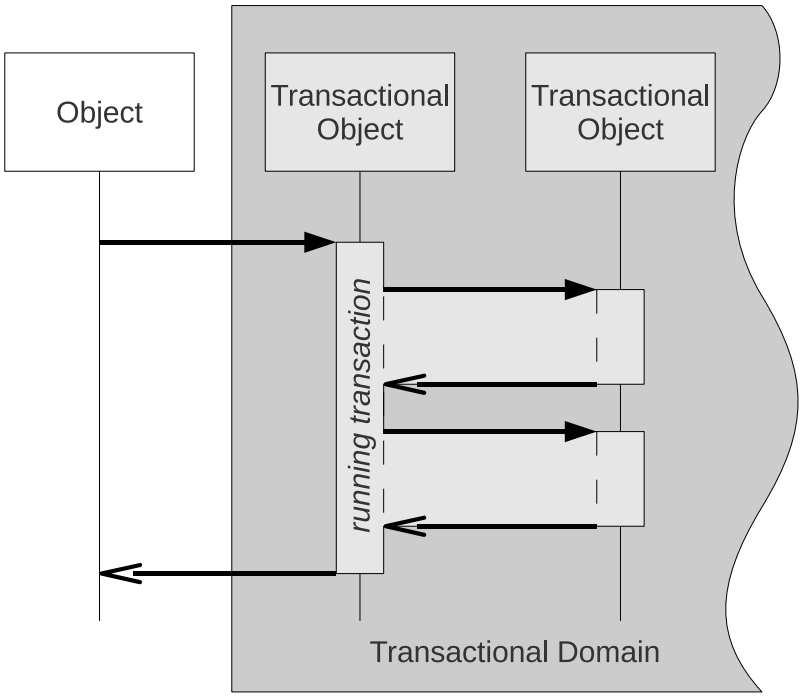
# Proposed Programming Model

# Isolation by Separation




- Separating a program into:
  - *Transactional domain* (TD): Tx code and data
  - *Non-transactional domain*: Non-tx code and data
- Scope of a transaction
  - Entering the TD starts transaction
  - Leaving the TD finishes transaction
- Provides strong isolation
- Avoids privatization issues

# Java-Specific Programming Model




## Declaration

```
class TxObj implements Transactional {
  // tx code and data
}
```

- Transactional object constitute transactional domain
- Additional programming rules preserve strong isolation

# Rules for Transactional Objects

- Non-final variables must be private
- No access to non-transactional classes/objects
- Parameters of methods or constructors:
  - Primitive types or `java.lang.String`
  - References on transactional objects
  - Arrays need a transactional wrapper object
- No inheritance between domains
  - Only exception: `java.lang.Object`
- ...
- **Those rules are validated by a tool!**

# Privatization Support

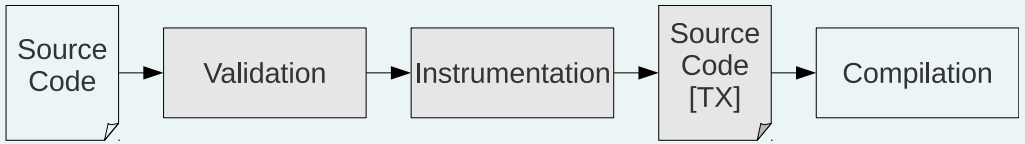
- A proposed API for expert programmers
- Supports two modes of transactional objects
  - Shared: Object protected against concurrent access
  - Privatized: Object to be used by the owning thread only
- Proposed (revised) functionallity for transactional object
  - privatize object
  - publish object
  - check if the object is *privatized* by given thread
  - check if the object is *shared*

4

# Prototype

# Toolchain

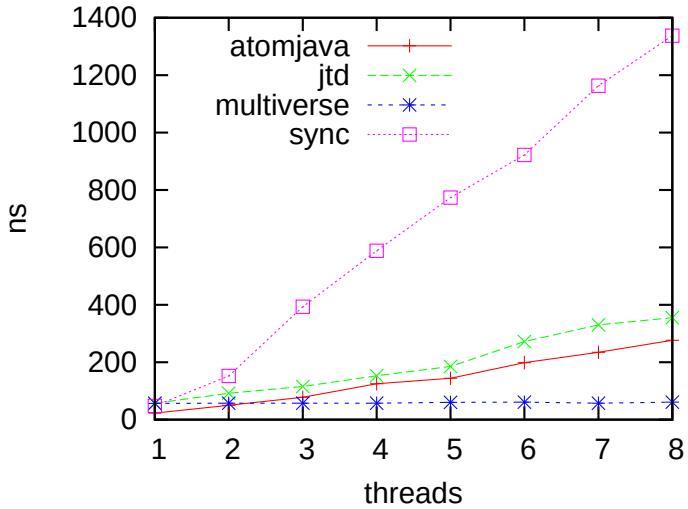
## Build process




- Java Transactional Domain (JTD) Toolchain
- Based on modified AtomJava (Univ. of Washington)
- Integrated with Eclipse
- Provides
  - Validation against programming rules
  - Automated source code instrumentation

# Evaluation

- Evaluated ours (jtd) vs.
  - Original (atomjava)
  - BC instr. (multiverse)
  - Java monitors (sync)
- Test platform
  - 2 x Intel quadcore @ 3GHz
  - 64bit Linux, kernel 2.6.26
  - 64bit Sun JVM 1.6.0_21




- Average over $10^7$ iterations à 100 transactions
- Transaction of around 10 bytecode instructions
- Overhead: $35 - 75 ns/call$

5

# Conclusion

# Summary

- A safe programming model for transactional memory
  - Ensures strong isolation
  - Prevents privatization problem
  - Supports software modularization
  - No restrictions on underlying TM algorithm
- A prototypical toolchain for Eclipse
  - Validation integrated with Java editor
  - Automated source code instrumentation

# Future Work

- Proceede with reviewing of existing approaches
  - Condition-driven synchronization
  - Irrevocable actions
  - Long running transactions
  - Application-level debugging

# Avoiding Publication and Privatization Problems on Software Transactional Memory

Holger M

Frühjahrstreffen 2011

**Holger Machens**

Research Assistant

Phone +49 / (0)40 428 78 3448

e-Mail machens@tu-harburg.de

http://www.ti5.tu-harburg.de/staff/machens

Institute of Telematics
Hamburg University of Technology

TUHH