

# Simplifying Multithreading & Boosting Performance Programming with Transactional Memory

Transactional memory provides an easy-to-use mechanism for controlling concurrent access to shared data

- Simple programming model – transactions guarantee atomicity and isolation
- Underlying implementation provides scalability and performance
- Avoids common problems of lock-based synchronization

## Multi-threaded programs require concurrency control

John and Mary Smith have \$1000 in a joint account.

Thread 1: John deposits \$100

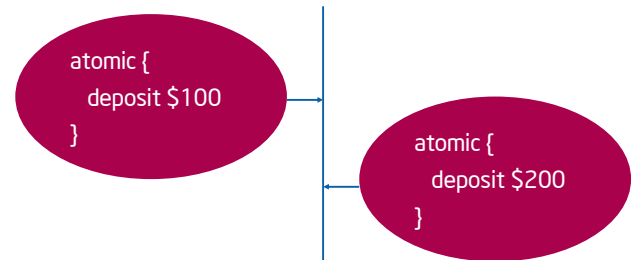
1. bal = acc.get ("Smith")
2. bal = bal + 100 (= \$1100)
6. acc.put ("Smith", bal)

Thread 2: Mary deposits \$200

3. bal = acc.get ("Smith")
4. bal = bal + 200 (= \$1200)
5. acc.put ("Smith", bal)

Account balance is \$1100. Mary and John lost \$200.

## Transactions



*Strong atomicity provides an intuitive programming model*

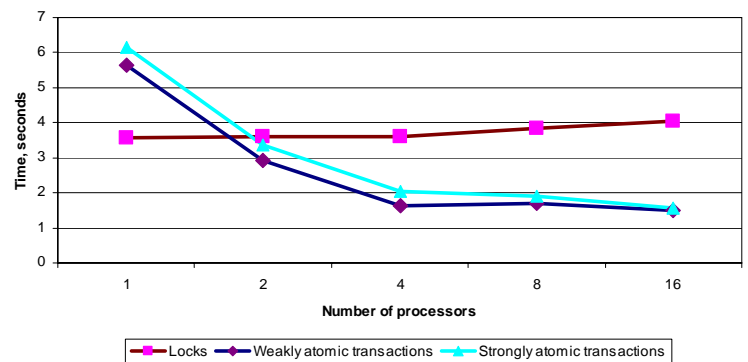
## Weak atomicity

- Commonly accepted model for software transactional memory
- Falsely assumes only transactions access shared data
- May lead to unintuitive implementation-dependent behavior
- Fails where locks work

## Strong atomicity

- Always guarantees transactional properties
- Treats all memory operations as mini-transactions
- May impose overhead on non-transactional code
- Allows for more reliable multi-threaded programming

Scalability on OO7 benchmark



## Programming Systems Lab

Ali-Reza Adl-Tabatabai, Rick Hudson, Vijay Menon, Yang Ni, Bratin Saha, Tatiana Shpeisman, Adam Welc

Collaborator: Dan Grossman, University of Washington

Research at Intel  
[www.intel.com/research](http://www.intel.com/research)

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. \*Other names and brands may be claimed as the property of others. Copyright © 2007, Intel Corporation. All rights reserved.

