

# Software Transactional Memory

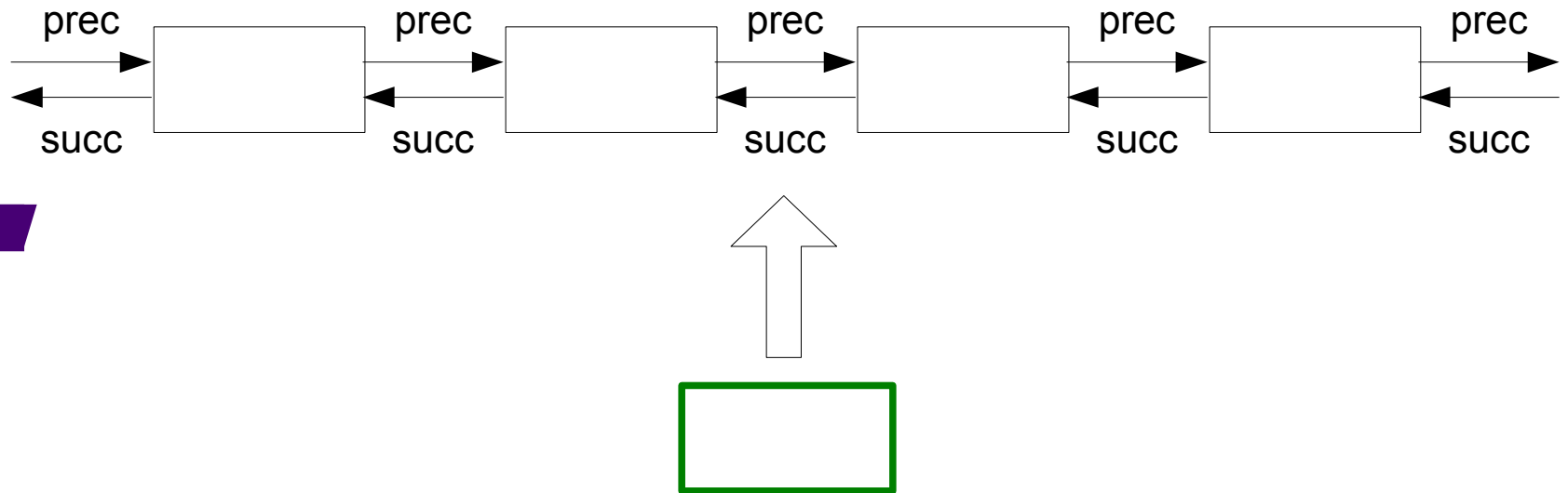
Michel Weimerskirch, 1st February 2008

- Motivation
- Basics
- Optimisation Approaches
- Empirical Evaluation
- Conclusion and Outlook

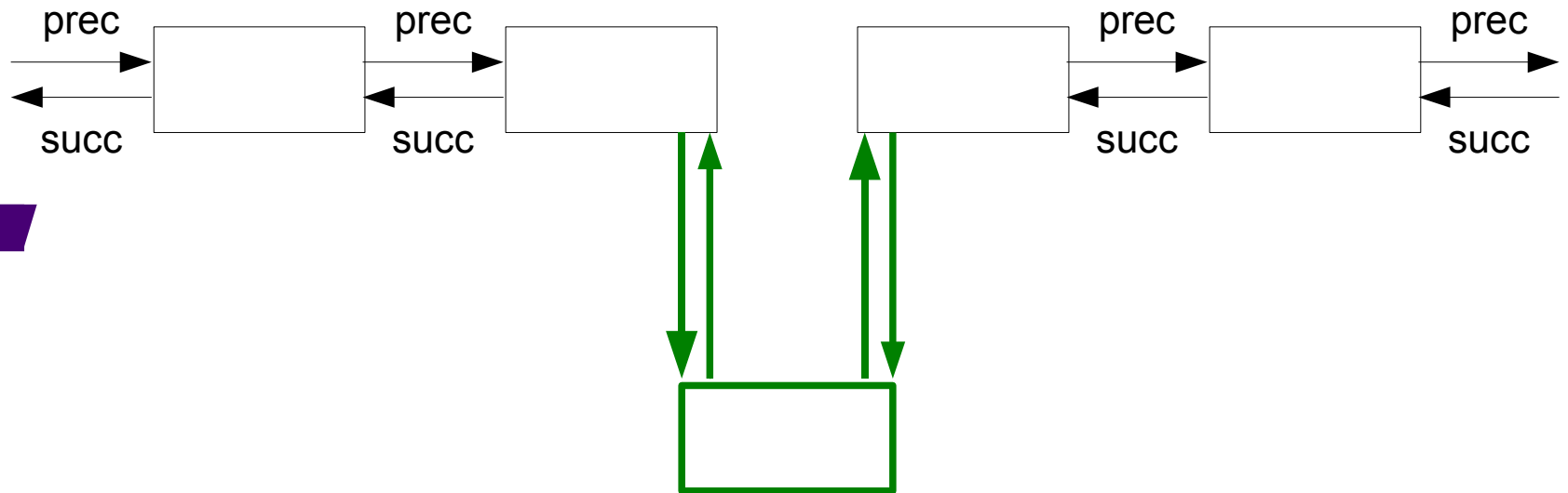
# Motivation

- Problem: Concurrency control
- Today this is often solved using locks

# Example (1): Double-linked list



# Example (2): Double-linked list



# Example (3): Coarse-grained locks

```
public synchronized void  
    insertNode(node, precedingNode) {  
        node.prec = precedingNode;  
        node.succ = precedingNode.succ;  
        precedingNode.succ.prec = node;  
        precedingNode.succ = node;  
    }
```

# Example (4): Fine-grained locks

```
public void insertNode(node,  
    precedingNode) {  
    synchronized(precedingNode) {  
        synchronized(precedingNode.succ) {  
            node.prec = precedingNode;  
            node.succ = precedingNode.succ;  
            precedingNode.succ.prec = node;  
            precedingNode.succ = node;  
        }  
    }  
}
```

# Example (5): Fine-grained locks

```
synchronized(precedingNode) {  
    synchronized(precedingNode.succ) {  
        ..  
    }  
}
```

```
synchronized (precedingNode) {  
    synchronized (precedingNode.succ) {  
        ..  
    }  
}
```

```
synchronized (precedingNode.succ) {  
    synchronized (precedingNode) {  
        ..  
    }  
}
```

- Problem: Locking granularity:
  - Coarse grained locks
    - Threads lock each other out
  - Fine-grained locks
    - Higher overhead
    - Risk of deadlocks
    - Lower maintainability
- Recent development: Multiprocessors even at home
  - Effective concurrency control needed

- First solution approach:  
hardware-based transactional memory (1986)
- Solution presented during this talk:  
**STM** (Software Transactional Memory)

# Example (7): Transactional Memory

```
public void insertNode(node,  
    precedingNode) {  
    atomic {  
        node.prec = precedingNode;  
        node.succ = precedingNode.succ;  
        precedingNode.succ.prec = node;  
        precedingNode.succ = node;  
    }  
}
```

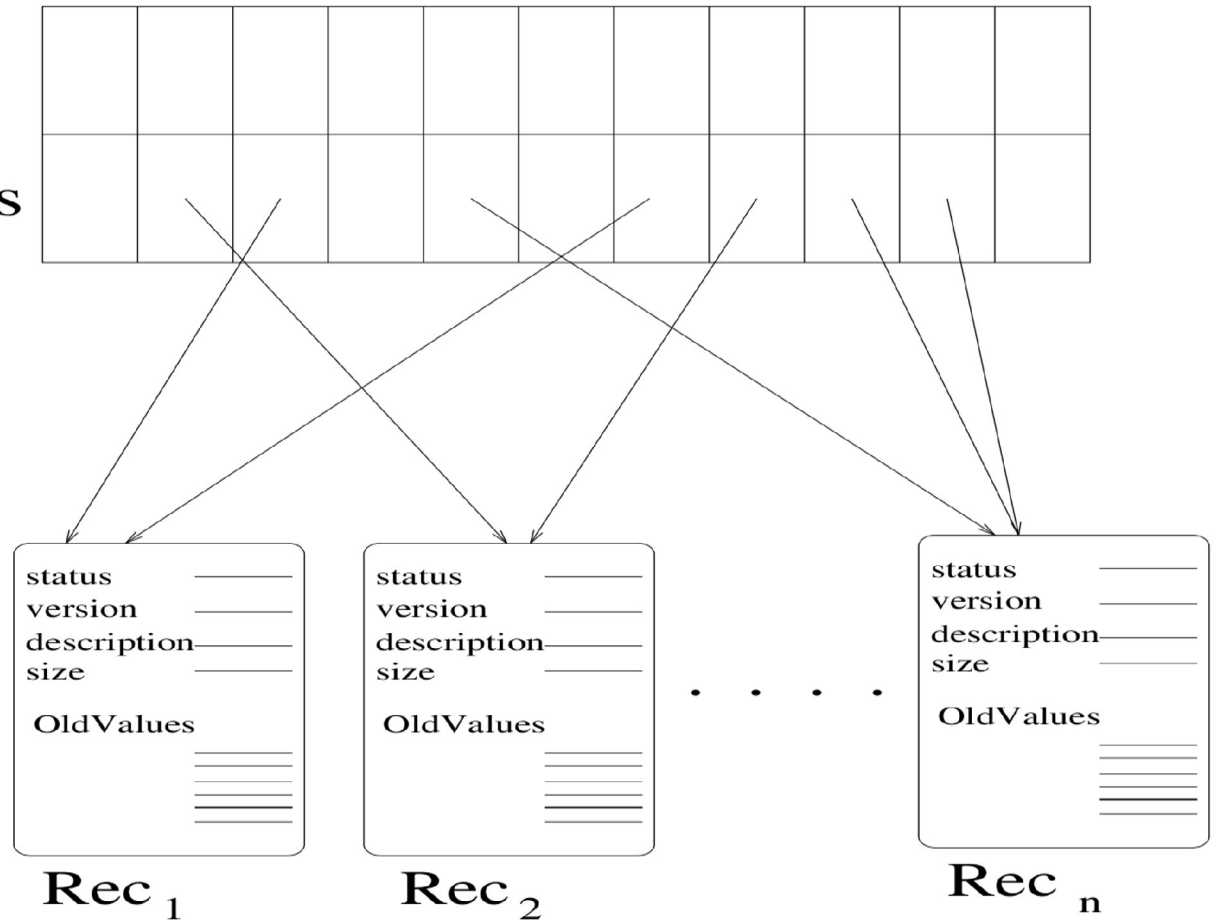
# Basics

- Transaction
  - Series of read and write operations
  - Executed atomically
  - On conflict: Transaction repeated
- Transactional operations
  - Established in the domain of database applications
  - Easy to understand

- Advantages
  - Developer does not need to worry about parallelism
    - Code is marked as atomic
    - Code is generated
- Disadvantages
  - Not always ideal
    - Successive collisions → repetitive rollbacks
    - Transactions can not contain certain operations

Memory

Ownerships



- Responsibility: reading transaction
  - Verification if read-set has been modified
- Only repeatable operations (e.g. **no** I/O operations)
- Non-redundant-helping
- Lock free → Optimistic

# Optimisation approaches

# Why obstruction-freedom is not needed

- ✗ Obstruction-freedom prevents a long-running transaction blocking others.
  - ✓ But: Only true if no conflicts appear between the long-running transaction and the intermediate ones
- ✗ Obstruction-freedom prevents the system locking up if a thread is switched part-way through a transaction.
  - ✓ Not critical: Runtime system will adapt the task switching appropriately.
- ✗ Obstruction-freedom prevents the system locking up if a thread fails.
  - ✓ Not true: A software failure would also break a non-STM version of the same application.

- Cache-Locality
  - Obstruction-free algorithms made it impossible for metadata to be stored locally along with the memory objects.
- Excessive Active Transactions
  - Due to obstruction freedom, no thread will ever wait for another thread to complete.

# Ennals' lock-based STM algorithm (1)

- Encounter-order locking
- Version counter **for each object**
  - When reading an object its current version is logged
  - Commit-time verification

# Further optimisations: The TL algorithm

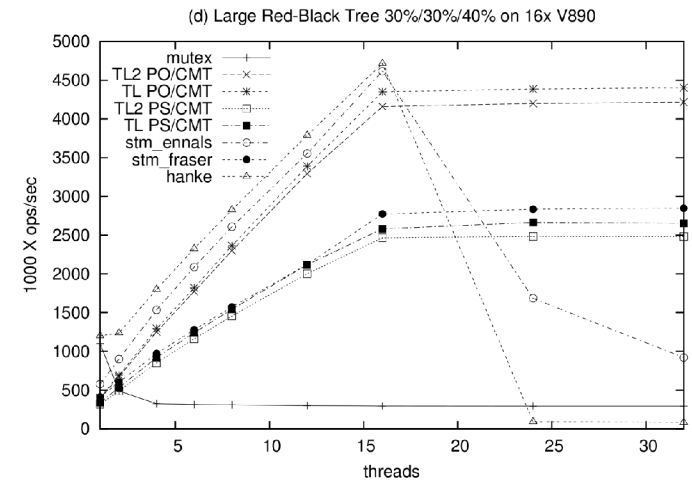
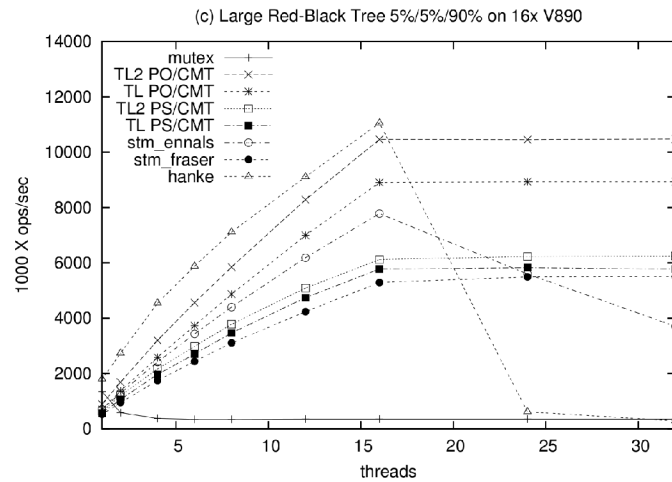
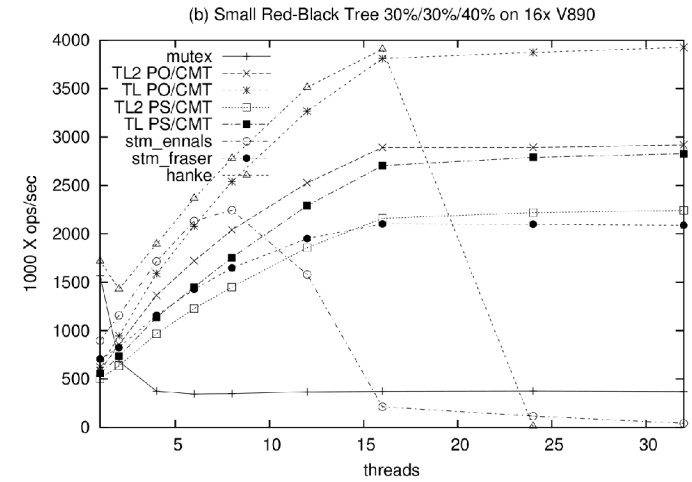
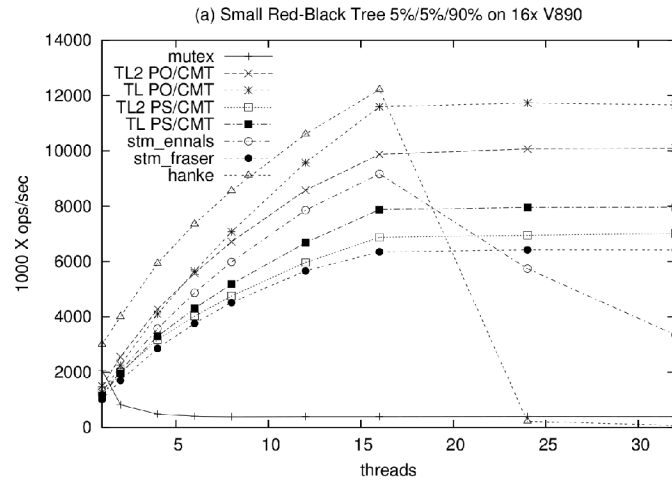
- **Global** version clock
- Commit-time locking
  - Locks are held for a shorter time
- Encounter-time locking still exists, but: bounded spin and back-off delay similar to CSMA-CD

# The TL2 algorithm

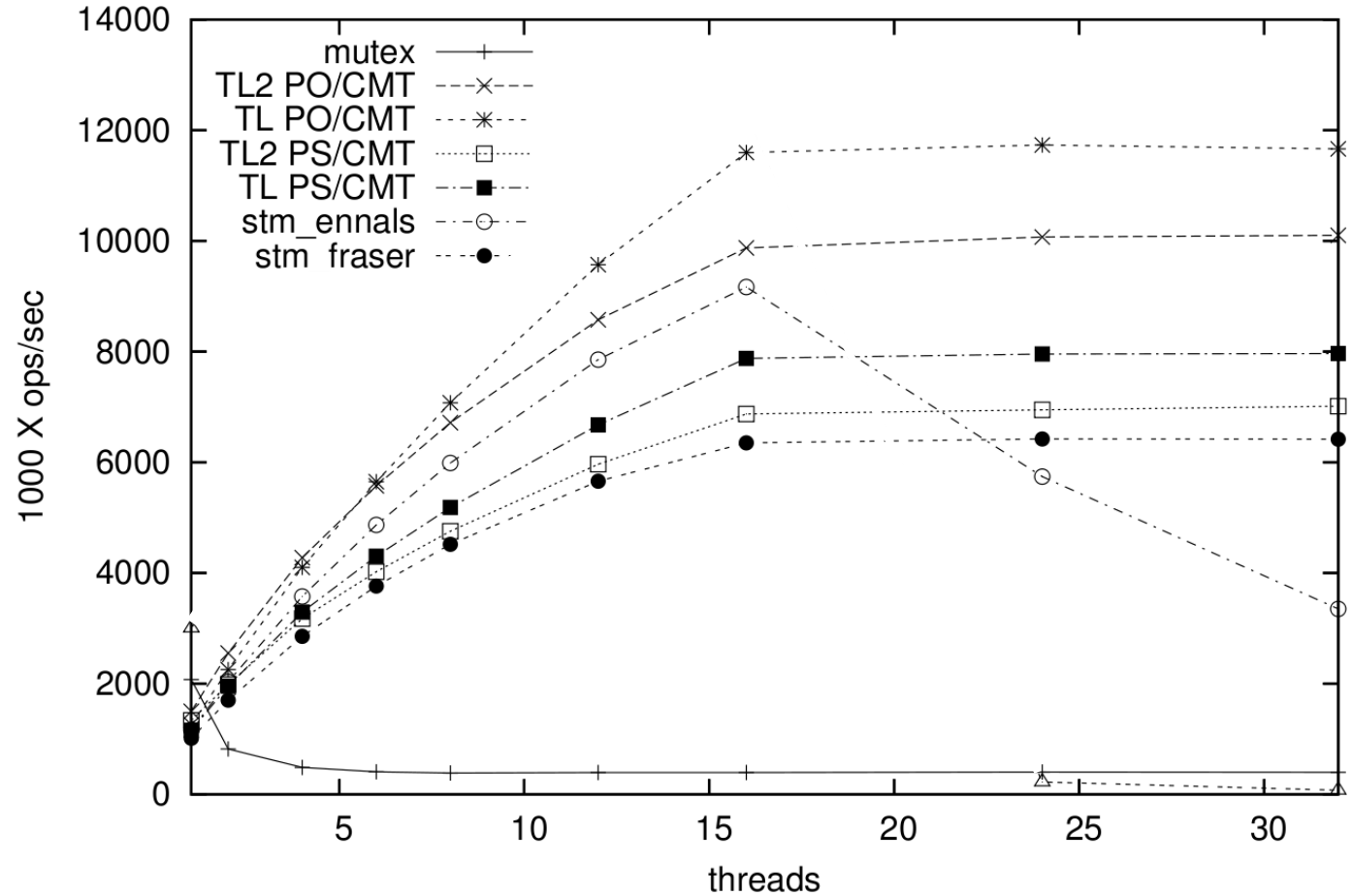
- TL2 solves the two following problems:
  - Only closed memory systems
  - Specialized managed runtime environments needed

- Read global version counter
- Speculative execution
- Lock write set
- Increment global version counter
- Verify that no read objects have been modified
- Write new values (with new counter value) and release locks

# Empirical Evaluation



(a) Small Red-Black Tree 5%/5%/90% on 16x V890



## Conclusion and Outlook

- STMs are more straightforward to program than locks
- Downside: mechanisms behind the scene are much more complex, which is why transactional memory is a field that has undergone many research efforts and will probably undergo more research efforts in the near future.
- STMs are coming closer to being production ready. A few issues will still have to be resolved:

- Legacy systems will still need to work alongside STM-based systems.
- Beside this, tool support (e.g. for debugging) needs to be pushed.
- New focus: hybrid solutions combining STM and HTM.

- Tom Knight:  
**System and method for parallel processing with mostly functional languages -**  
*Patent number: 4825360*
- Nir Shavit, Dan Touitou:  
**Software Transactional Memory**  
*14th ACM Symposium on Principles of Distributed Computing*
- Tim Harris, Keir Fraser:  
**Language Support for Lightweight Transactions**  
*Object-Oriented Programming, Systems, Languages, and Applications*
- Robert Ennals:  
**Software Transactional Memory Should Not Be Obstruction-Free**
- Dave Dice, Nir Shavit:  
**What Really Makes Transactions Faster?**  
*Proceedings of the 1st TRANSACT 2006 workshop*
- Dave Dice, Ori Shalev, Nir Shavit:  
**Transactional Locking II**  
*Proceedings of the 20th International Symposium on Distributed Computing*
- Ali-Reza Adl-Tabatabai, Christos Kozyrakis, Bratin Saha:  
**Unlocking concurrency**  
*ACM Queue 4, 10 (December 2006)*