Coping With Context
Switches in Lock-Based STMs

Yoav Cohen
Joint with
Yehuda Afek and Adam Morrison
Tel Aviv University
Agenda

- Background and motivation
  - The Lock Stealing Algorithm
  - TL2 Implementation
  - Empirical Evaluation
Software Transactional Memory

- Programmers define blocks of code as *transactions*:

  ```
  atomic {
  <code block> 
  }
  ```

- Transactions take effect atomically

Simplicity of Global Clock with performance of Fine-Grained Locking
Lock-Based STMs

Basic design:

Application Memory

Array of Versioned Locks

Map

$\text{Version} = \text{Lock Bit}$
Incrementing a Shared Counter

```c
atomic {
    int c = counter;
    c = c+1;
    counter = c;
}
```

counter = 13

Start TX
Read counter
Take
Rollback

Start TX
Read counter
Take Lock
Commit
Write counter

Start TX
Read counter
Take Lock
Commit
Write counter

Start TX
Read counter
Take Lock
Commit
Write counter

V = 0
V = 1
V = 0
Context Switches

Threads may be switched-out when:
- \# S/W threads > \#H/W threads
- Interrupts
- Page faults

Q: A thread with a lock is switched out. What happens?

A: Transactions that need this lock abort or wait
The Result: Throughput Degradation

Deuce TL2 running on Intel i7 with 8 hyper threads

More context switches →
Agenda

- Background and motivation
- The Lock Stealing Algorithm
- TL2 Implementation
- Empirical Evaluation
The Solution: Lock Stealing

Instead of waiting for a switched-out lock, **steal it:**

- Abort the switched-out transaction
- Take the lock
Lock Stealing

- Status field per thread:
  - RUNNING, COMMITTED or ABORTED

- Enhanced locks:
  - The pair \(<\text{Owner}, \text{Local Clock}\)> is a unique transaction identifier.
Lock Stealing

- <T1,24> aborts <T2,10>:
  - CAS(T2, <RUNNING,10>, <ABORTED,10>)

- <T1,24> steals L from <T2,10>:
  - CAS(Lock,
    <\!\!l=1, v=2, owner=T2, local\_clock=10>,
    <\!\!l=1, v=2, owner=T1, local\_clock=24>)
Q: Can we always do this trick?

A: Nope. When a transaction is COMMITTED, it can’t be aborted.
Brief Summary

- Context switches cause throughput degradation

- Because switched out locks result in lots of aborts

- New approach: instead of waiting for locks, abort other and steal the lock
Agenda

- Background and motivation
- The Lock Stealing Algorithm
- TL2 Implementation
- Empirical Evaluation
Lock Stealing for TL2

- Based on Deuce
  - An open-source Java STM framework

- Added Contention Management support:
  - Upon conflict contention manager invoked
  - Decides what to do:
    - Restart current transaction
    - Wait for lock
    - Abort other transaction and steal lock
Lock Stealing for TL2

- Lock-Waiting Contention Managers:
  - Suicide, Aggressive, Karma and Polka

- Lock-Stealing Contention Managers:
  - AggressiveLS, KarmaLS and KillPrioLS
Agenda

- Background and motivation
- The Lock Stealing Algorithm
- TL2 Implementation
- Empirical Evaluation
Empirical Evaluation

- **Benchmarks:**
  - Integer-Set microbenchmarks
  - STAMP – simulates real applications

- **Hardware:**
  - Intel i7 920 Extreme Edition (Nehalem)
    - 2.67 GHz
  - 4 cores, each running 2 hardware threads
Red-Black Tree Integer Set
Red-Black Tree Integer Set

64K, 20% updates

Throughput (transactions/s)

Number of threads

TL2
TL2-AggressiveLS

-10%
+19%
STAMP Intruder

The graph shows the speedup of different thread configurations as the number of threads increases. The x-axis represents the number of threads, ranging from 0 to 64, while the y-axis represents speedup, ranging from 0 to 3.0.

- **TL2**
- **TL2-Aggressive**
- **TL2-Karma**
- **TL2-AggressiveLS**
- **TL2-KarmaLS**
- **TL2-KillPrioLS**

The different lines and markers represent the performance of each configuration. As the number of threads increases, the speedup decreases for all configurations, indicating diminishing returns with additional threads.
Thank You
Links

- Deuce STM project
  - http://sites.google.com/site/deucestm/
  - org.deuce.transaction.tl2cm package