

# Reconsidering Transactional Memory

James Larus

Microsoft Research

Dagstuhl Seminar, September 2007

# Transactional Memory

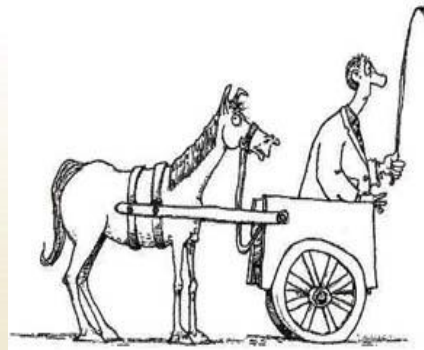
- Lightweight transactions for mutual exclusion/atomicity in parallel programming
- Replace explicit synchronization with declarative specification
  - “this code should appear atomic” not “acquire this lock”
  - Synchronization is major source of errors in concurrent programs
- All-or-nothing semantics helps ensure consistent program state
- Construct composable parallel abstractions
  - Need not be aware of abstraction’s locking discipline
- Active research community



MORGAN & CLAYPOOL PUBLISHERS

# Transactional Memory

Jim Larus  
Ravi Rajwar



*SYNTHESIS LECTURES ON  
COMPUTER ARCHITECTURE*

Mark D. Hill, *Series Editor*

Buy this book!

Great book!

# Is TM a Panacea?

- With benefit of experience, should we still be excited about potential of TM?
- Challenges are software
  - No strong opinions about practicality of hardware
- It isn't as simple as we thought
- Lots of good research problems
  - (not necessary the ones being researched)
  
- My opinions, not Microsoft's (but they should be)

# Strong/Weak Isolation

- Does transaction appear atomic wrt non-transactional code?
  - TM will co-exist with legacy code for a long time
- HTM: strong isolation is (relatively) easy
- STM: strong isolation is very costly
  - Cost imposed on all code
  - Amdahl's law amplifies effect on sequential code
- $\Rightarrow$  STM  $\neq$  HTM
  - HW has the better model (good)
  - Initial TM provides the most gotchas (bad)
- Problem disappears if transactional data is segregated
  - Is bifurcated model practical in non-functional language?

# Privatization Problem

Transaction  $T_1$ :

```
ListNode res;  
atomic {  
    res = lHead;  
    if (lHead != null)  
        lhead =  
        lhead.next;  
}  
res  
res
```



Transaction  $T_2$ :

```
atomic {  
    ListNode n = lHead;  
    while (n != null) {  
        n.val ++;  
        n = n.next  
    }  
}
```

# Partial Replacement for Synchronization

- Synchronization ensures isolation, provides coordination
- TM ensures isolation
- How do two parallel threads coordinate?
  - Harris & Peyton-Jones: `retry/orElse`
  - Marathe & Larus: data parallel operators
- More is needed

# Native Code

- Native (C/C++) code is challenge for STM
  - Cannot rely on language safety to prevent side-effects from corrupting state in doomed transaction
  - Internal pointers complicate algorithms
- Proposed solutions change memory allocation
  - Backward/performance compatibility issue

# Simple Semantics are Not Formal

# Formal Semantics are Not Simple

- Single lock atomicity: transactions equivalent to execution in critical sections protected by a single lock
  - Easily explicable model (serializability made concrete)
- What if atomic block contains an infinite loop?
  - Hangs program with single lock
  - Hang single transaction in most TM systems
- Non-termination is special case (bottom?), ...

# Single Lock Problems, cont'd

```
int num_aborts = 0;
int aborted = true;
do {
    atomic {
        ...
        aborted = false;
    }
    if (aborted) num_aborts ++;
} while (aborted);
```

- `num_aborts == 0` with single lock
- `num_aborts >= 0` with most (any) TM system

# I/O

- I/O is shorthand for an action that affects state outside of process
  - Difficult to hide side effects and roll back on abort
  - Distributed Transaction Monitors
  - Many flavors of transactions (WinFS)
- Possible solutions
  - Buffer input/output
  - Serialize I/O (token)
  - Leave non-transactional

# Existing Libraries and Abstractions

- Can existing run-time systems, libraries, components, applications, ... be used in TM world
  - Will existing synchronization mechanisms be respected?
  - Does existing code need to be updated to be aware of TM?
- Is starting afresh (“boiling the ocean”) a practical starting point?

# Semantic Level Causes Performance Problems

- TM defines transactions in terms of memory read-write
  - DB world saw shortcoming of this approach
    - Memory-level conflicts may not affect program semantics
    - Low level can cause performance problems by introducing unnecessary conflicts
- ```
x = unique_marker ++;
```
- Open nesting suspends transaction for an interval of statements
    - Semantics are complex and difficult to apply

# Can Long Transactions Complete?

- Long running transactions become increasingly likely to encounter a conflict and abort
- With modern libraries and many levels of abstractions, cost of many operations is hidden
  - As is state an operation touches
- Conflict resolution policies can ameliorate problem
  - What if both conflicting transactions are long-running?

# Modest Proposal

- Original TM idea
  - Transaction for manipulating in-core data structures
  - Limited (not necessarily bounded) amount of computation in atomic block
  - No I/O
  - Simple nesting
- Haskell TM?
- More research on hard SW problems