PODC '05

Randomization in STM Contention Management

Bill Scherer and Michael L. Scott
University of Rochester

Dynamic Software Transactional Memory

(DSTM) [Herlihy et al., PODC 2003]

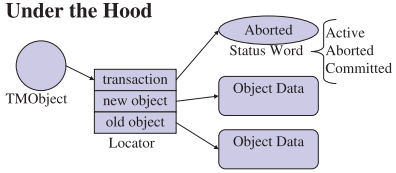
- Universal construction for non-blocking synchronization
  - Treat memory as a database
  - Use *transactions* to update data structures
  - Obstruction-Free
- Highly concurrent operations – full disjoint access parallelism
- Software version of a hardware scheme
  - [Herlihy and Moss, 1993]
  - Term STM due from [Shavit and Touitou, 1995]

From the User's Perspective

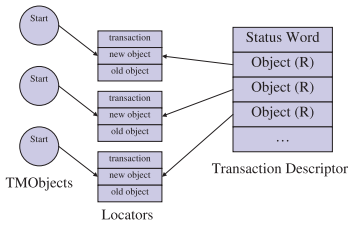
- Divide memory into a series of objects
- Use special interface to access objects
  - STM copies object data behind the scenes
  - Read-only, read/write access supported
  - Manipulate data with normal reads/writes
  - Can early-release an object opened read-only
    - Requires programmer to ensure consistency
  - In cases of conflict, one transaction is aborted
- At end of transaction attempt, **`commit()`** effects all changes atomically
  - Retry transaction if **`commit()`** fails

```
void sortedListInsert(List L, int v) {
    ListNode newList = new ListNode(v);
    TMObject newNode = new TMObject(newList);
    do {
        beginTransaction();
        try {
            ListNode curr = (ListNode)L.head.openWrite();
            ListNode next = (ListNode)curr.next.openWrite();
            while (next.key < v) {
                curr = next;
                next = (ListNode)curr.next.openWrite();
            }
            curr.next = newNode;
            newList.next = next;
        } catch (DeniedException d) {}
    while (!commitTransaction());
}
```

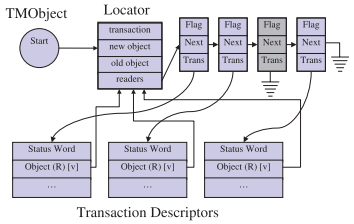
Under the Hood



Invisible Reads



Visible Reads



Contention Management

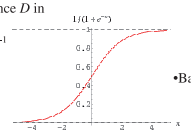
- One manager for each thread
- Inputs: notification messages
  - Begin/End Transaction; Opening Object
- Outputs: decisions
  - Abort competitor holding the object I want?
  - Choices: abort self, abort other, stall
  - Abort often enough to avoid deadlock, but rarely enough to avoid livelock
- How do we decide what policy to use?

The Karma Manager

- [Scherer & Scott, CSJP 2004]
- Priority = acquired objects = work invested so far
  - Increment every time we open an object
- Priority reset to 0 on commit
  - *Not* reset if aborted: better chance to finish "next time"
- Abort enemy transaction if priority lower
  - Otherwise, wait for a fixed period
  - If priority + #times waited > enemy priority, abort
  - Better to abort a lower-priority transaction
  - Less rework = higher overall throughput

Randomization

- Abortion
  - Basic: abort if #backoff periods exceed difference D in priorities
  - Randomized: abort with probability $(1 + e^{-1/2D})^{-1}$
  - Sigmoid function: see right
- Backoff
  - Basic: wait T μsecs between access attempts
  - Randomized: uniform from 0..$2T$
- Gain
  - Basic: 1 point of priority with each object opened
  - Randomized: uniform from 0..200



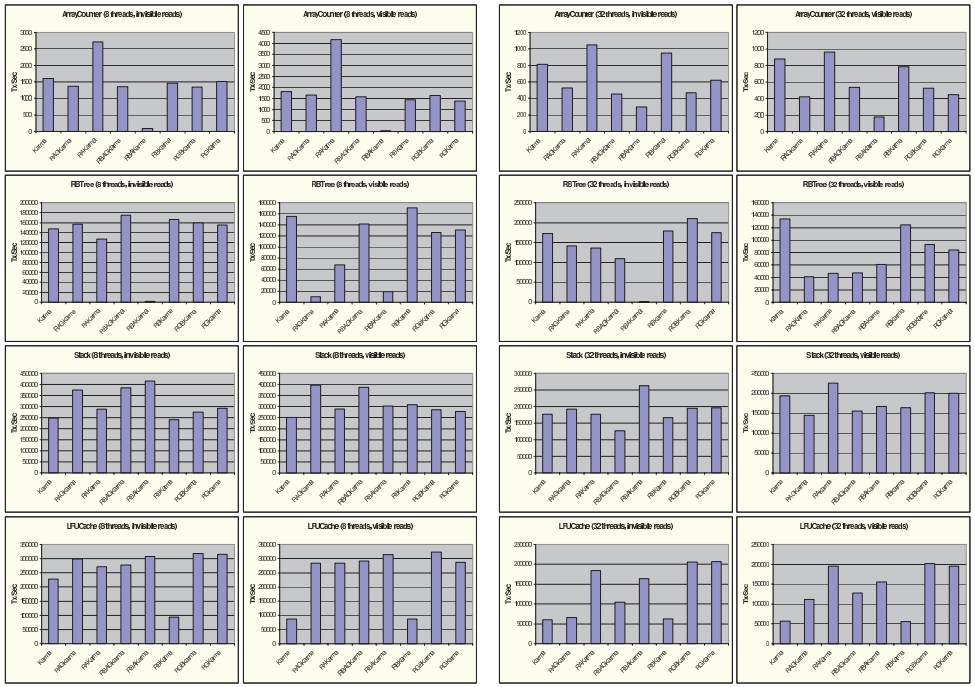
The Sigmoid Function

Analysis

- Some combination of randomization improves throughput for all benchmarks
- ArrayCounter. IntSet, IntSetUpgrade: randomizing just abortion best
- Randomizing both abortion and backoff:
  - Very poor results for ArrayCounter and RBTree
  - Improves throughput for LFUCache and Stack
- Randomizing gain (alone or with others) improves LFUCache and RBTree
- Little difference between visible and invisible read patterns

Interpretation

- Abortion
  - Powerful for breaking up semi-deterministic livelock patterns
  - Particular visible in ArrayCounter, where increment and decrement transactions are highly prone to repeated mutual abortion
- Gain
  - Similat in effect to randomizing abortion
  - Uniformly random vs biased abortion randomization from sigmoid
- Abortion + Backoff
  - Produces great variance in how long a thread waits to abort an enemy
  - Reducing wait period hurts longer transactions
    - Typified by ArrayCounter, RBTree
  - Increasing wait period decreases contention for short transactions
    - Typified by LFUCache, Stack
- Backoff
  - Good for locking algorithms (avoids simultaneous retry pathology)
  - Less important for two-transaction case
    - One continues oblivious to conflict; one backs off



8 Threads: throughput **32 Threads: throughput**

Test Environment

- 16-processor SunFire 6800 machine
  - Cache-coherent Multiprocessor
  - 1.2 GHz UltraSparc III processors
  - Donation from Sun's Scalable Synchronization Research group
- Sun's HotSpot Java 1.5 VM
- 10-second test runs
- All 8 combination of randomizing three facets of Karma

IntSet

- Sorted Linked-list set implementation
- Insert, remove transactions
- Each list node opened for read/write access

IntSetUpgrade

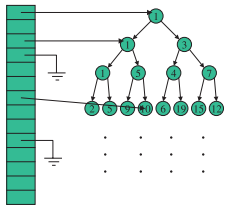
- Another sorted Linked-list set implementation
- Objects opened for read-only access until insertion/deletion point found
- Access upgraded to read/write for nodes to be modified

RBTree

- Add/remove numbers from a balanced binary tree
  - Tight range (0…255) increases contention
- Two-step operations
  - Work down the tree to find insertion/deletion point
  - Work back up the tree restoring balance
- Transactions can mutually block each other
  - Interesting opportunities for contention mgmt.

LFUCache

- Simulates behavior of a web cache
  - Least-Frequently Used replacement policy
  - Operations are cache updates from page hits
- Two-part data structure
  - Big array represents all pages
  - Priority queue heap represents the cache itself
    - Tree structure; least-used nodes bubble to the root
    - Fixed-size cache
- Unidirectional object access pattern
  - no mutual blocking
- Two trans. can need same sequence of objects
  - Risk of livelock
  - Waiting works



Stack

- Concurrent stack
- Push, Pop operations

ArrayCounter

- Ordered list of 255 simple counters
- Increment transactions raise 0,1,2, … 255
- Decrement transactions lower 255, 254, …, 0
- Exacerbates proneness to livelock