Randomization in STM Contention Management
Bill Scherer and Michael L. Scott
University of Rochester

Contention Management
- One manager for each thread
- Impute notification messages
- Reconfigurable Transaction: Opening Object
- Output: abort commit holding the object I want?
  - Choose abort, still abort other, stall
  - Abort ultra-sticky to avoid deadlock, but only enough to avoid livelock
- How do we decide what policy to use?

The Karma Manager
- [Scherer & Scott, CSP 2003]
- Privity = acquired objects object invented so far
- Privity controls state transition
- Privity aborts on commit
- Nickname = times wanted - enemy priority
- Better to abort a lower-priority transaction
- Less work = higher overall throughput

Randomization
- Abortion
  - Basic: abort if backlog period exceeds difference D in priority
  - Randomized abort with probability \( \frac{1}{1 + \alpha D} \)
  - Sigmoid function: see right

[Diagram of sigmoid function]

Analysis
- Some combination of randomization improves throughput for all benchmarks
- ArrayCounter, Inset, InsetUpgrade: randomization just abort-boost
- Randomizing both abort and backoff
- Very poor results for ArrayCounter and RBTrees
- Improves throughput for LFUCache and Stack
- Randomizing gain (abort or both) improves LFUCache and RBTrees
- Little difference between visible and invisible read patterns

Interpretation
- Abortion
  - Powerful for breaking up semi-deterministic livelock patterns
  - Privity visible in ArrayCounter, where incremental and decremental transactions are highly prone to repeated mutual abortion
- Gain
  - Similar in effect to randomizing abortion
  - Uniformly random vs. based 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
  - Empirical by ArrayCounter, RBTrees
  - Increasing wait period decreases contention for short transactions
  - Empirical by LFUCache, Stack
- Backoff
  - Good for locking algorithms (avoids simultaneous retry pathology)
  - Less important for two-transaction race
  - One continues oblivious to conflict, one backs off

Test Environment
- 16-processor SunFire 8600 machine
- Cache-coherent Multiprocessor
- 1.2 GHz UltraSPARC III processors
- Donation from Sun’s Scalable Transaction Research group
- Sun’s Hotspot Java 1.5 VM
- 10-second test runs
- All 8 combinations of randomizing three facets of Karma

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

IntSetUpgrade
- Ancestor 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

RBTrees
- Addressing numbers from a balanced binary tree
- Height range \( (0, 0.25) \) increases contention
- Two-step operations
- 3-way down the tree to find insertion/deletion point
- 3-way up the tree reversing balance
- Transaction can mutually block such other
- Interesting opportunities for contention mitigate

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 keep represents the cache itself
- Tree structure: leaf-ward nodes bubble to the root
- Fixed-size cache

Unidirectional object access pattern
- no mutual blocking
- Two trees can serve same sequence of objects
- Risk of livelock
- Waiting works

Stack
- Consists of stack
- Push, Pop operations

ArrayCounter
- Ordered list of 255 simple counters
- Increment transactions \( 0, 1, 2, \ldots, 255 \)
- Decrement transactions \( 255, 254, \ldots, 0 \)
- Enqueue processes to livelock