Beyond Simple Transactions and Atomic Blocks

Victor Luchangco
Sun Microsystems Laboratories

DISC Workshop on
What Theory for Transactional Memory

22 September 2009
Concurrent programming

- multicore will make this mainstream
  > need to make it easier
- transactional memory is just one tool
  > there are other tools
    > e.g., nonblocking data structures, conditional waiting, etc.
    > transactional memory must work with them
- programmers must think about performance
  > good performance will always be a challenge
  > size matters
  > need good algorithms and data structures
Transactional programming

Enrich the transactional interface.

Alternative viewpoints

- status quo is fine
- single global lock atomicity
- simple atomic blocks
- different programming models
  > functional programming, message passing, etc.
Alternative viewpoints

lock L
<critical section>
unlock L

atomic {
    <atomic block>
}

• single global lock atomicity
  > well-defined semantics
  > familiar, well-understood
  > backward compatible
  > gives programmer control
  > TM as implementation technique

• simple atomic blocks
  > simple, “natural”
  > doesn't overconstrain implementation
  > system handles retries
  > TM as implementation technique
Richer transactional interface

• transaction is atomic block that may abort
  > programmer must deal with aborted transactions
  > sensible defaults (e.g., simple atomic blocks)
  > specifies semantics, not implementation
    > TM as implementation technique

• transactions as basic structuring construct for concurrent programs
  > as functions are for sequential programs
  > key is modularity (primarily due to “isolation”)
Richer transactional interface

- nontransactional accesses
- other synchronization mechanisms
- escape mechanisms
- more intra-transactional operations
- transactions as abstract operations
- relaxed consistency/ordering guarantees
- modeling performance
- "meta" programming

weak isolation
allow txns to be reordered
different txns see different views

give programmers means of comparing algorithms
different metrics: latency, throughput, scalability

allows distributed implementation
need not violate atomicity/isolation

consider txn as single operation
undo abstract operation rather writes

abort handlers for boosting/open nesting

commuting transactions

ignoring "irrelevant" differences
Transactional synchronizers

- joint work with Virendra Marathe (SCOOL 2005)
- mechanism for transactions to interact
  > only interact on data protected by synchronizer
- interacting transactions must commit together
  > if any abort, then all must abort
  > isolation maintained for nonsynchronizing transactions
- implementation required notion of “sealing” to prevent new transactions from accessing synchronizer data while others are trying to commit
Transactional communicators

- generalization of transactional synchronizers
- based on dependency tracking
- eliminates need for “sealing”
- additional mechanisms provided
  - communicator-isolating transaction
  - wait-notify synchronization
- example applications
  - exchanger
  - producer-consumer queue
Performance models

- programmers need some way to decide which implementation strategy to pursue
- different metrics:
  > latency
  > throughput
  > scalability
- important to compare *relative* performance
- “quality of service” guarantees, perhaps programmer-specified
What theory for TM?

- formal specification of transactional memory
  > with enriched interface
- proofs of TM implementations
- formal specification of programming disciplines
  > guarantees under programming disciplines
- performance models
- what enrichments help?