Enhanced Concurrency Control with Transactional NACKs

Woongki Baek, **Richard Yoo**, and Christos Kozyrakis
Pervasive Parallelism Laboratory
Stanford University

Controlling the Concurrency

- Transactional memory
  - What: declare code sections as transactions
  - How: underlying system tries to concurrently execute transactions (atomic and isolated)

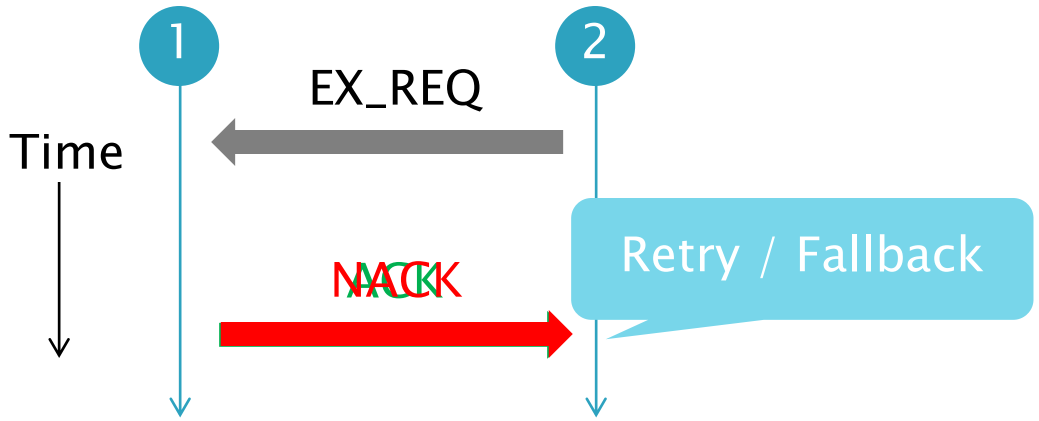
- Transactions may abort due to contention
  - For efficient transaction execution, the system must *control the concurrency*

- Concurrency controller
  - Adaptively controls the system-wide concurrency
  1. Contention managers: determine priority *after* conflict
  2. Adaptive scheduling: try to *predict* contention

Efficient Concurrency Controller

- An efficient concurrency controller requires low-level run-time information
  - E.g., dependencies among transactions, and system utilization level
- Challenges
  - Need to obtain such information in a *timely* fashion
  - Leverage existing hardware features to *lower cost*
- Utilize transactional NACKs

Transactional NACKs



- Cache coherence: used to deny unsatisfiable coherence requests
- TM systems with *eager conflict detection*: used to signal transactional conflicts
  - E.g., NACK request to a transactionally accessed line

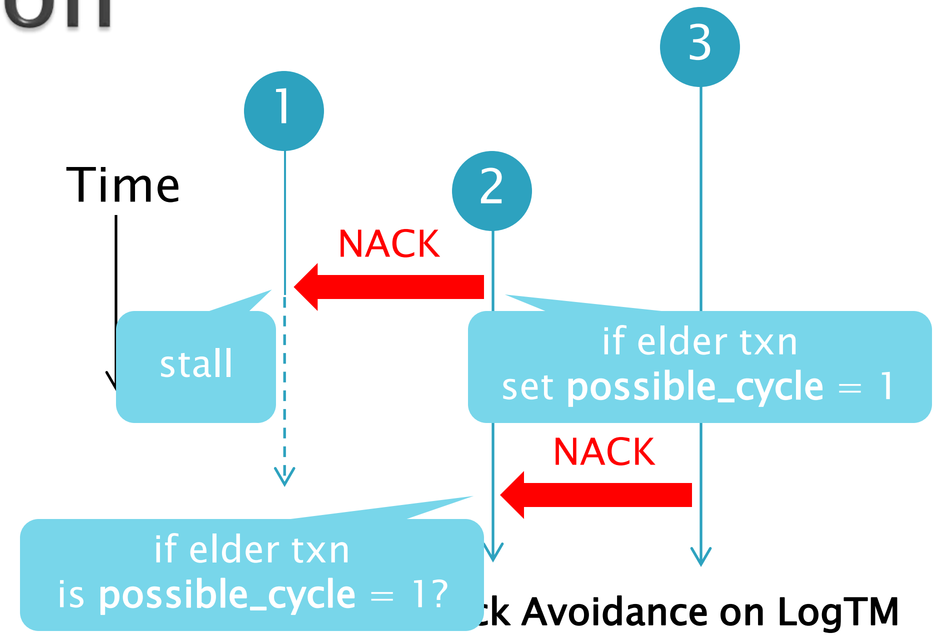
- NACK messages
  - Detailed dependency information / system utilization level
  - Already implemented in many TM systems (cheap)

Enhanced Concurrency Control w/ Transactional NACKs

- Previous work: use NACK for non-busy waiting [zilles'06] and conservative deadlock avoidance [moore'06]
- We propose *3 novel NACK use cases* that enable enhanced concurrency control

Use Case	Concurrency Control
Accurate Deadlock Detection	Aggressive Stalling
Dependency Tree Construction	Dependency Chain Cutting
Carrier Sensing	Exponential Backoff w/ Overshoot Avoidance

Case 1: Accurate Deadlock Detection



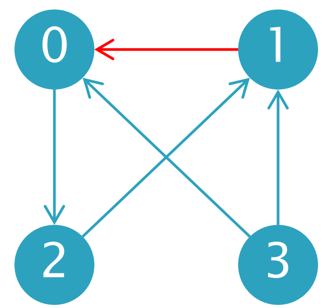
- On conflict, eager conflict detection TMs stall the attacker
  - Risks deadlock: implement *conservative deadlock avoidance*
- Abort the transaction when there is a *possible* deadlock
  - False positives may degrade performance

Using NACKs for Accurate Deadlock Detection

"P_0 NACK P_1"

Attacker \ Defender	0	1	2	3
0				
1	1			
2				
3				

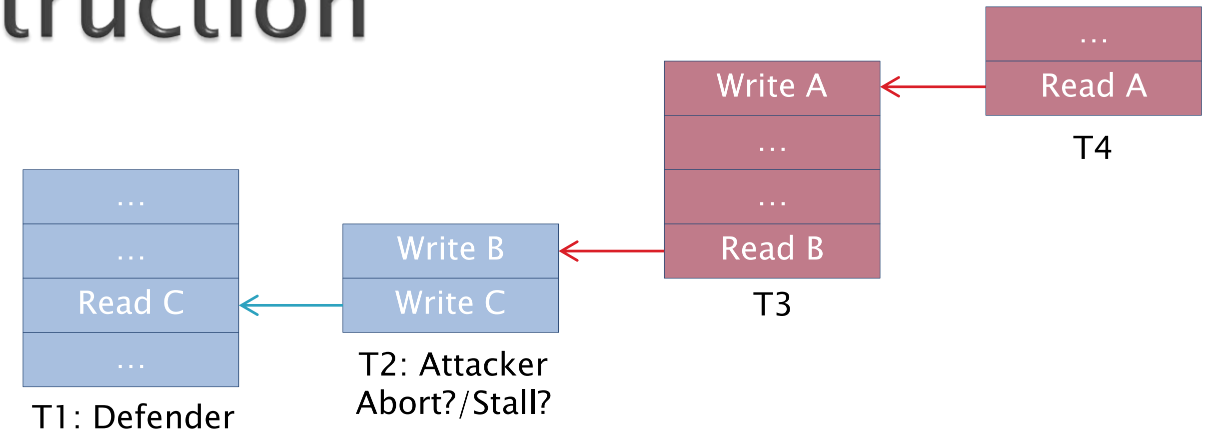
Wait-For Table



Wait-For Graph

- Arbiter snoops NACK messages and updates *wait-for table*
  - (i, j) => is P_i (attacker) stalling for P_j (defender)?
  - When P_i commits or aborts, clear row / column i
- Wait-for table encodes wait-for graph
  - Hardware can walk the table to detect deadlock
  - Distributed / low cost implementations are possible [shiu'01]

Case 2: Dependency Tree Construction

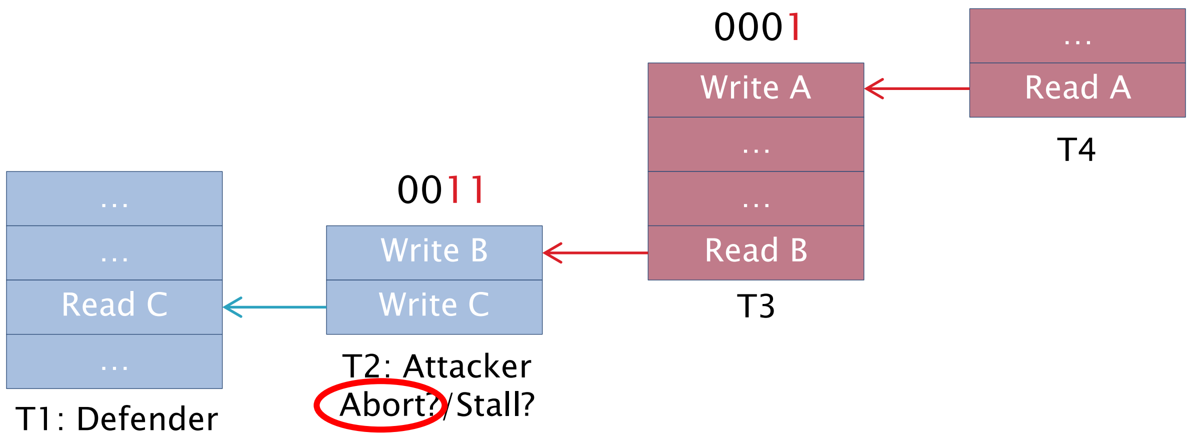


Cascaded Stalls of Transactions

- For TM systems w/ eager conflict detection
  - NACKing requests = fine grain locking
  - Stalling a single attacker may stall large number of txns
  - Avalanche effect: other transactions will soon get stuck

- Better off abort highly depended transactions
  - Need to know the # of both direct / *indirect* dependents

Case 2: Dependency Tree Construction (contd.)



Cascaded Stalls of Transactions

- Use NACK to track dependency relationship
  - Each transaction records the dependency as bit vector
  - Propagate the bit vector through coherence messages
  - Based on the info determine whether to abort / stall

Case 3: Carrier Sensing

- Exponential backoff
  - On abort, exponentially increase retry interval
  - Good: quickly escape contention
  - Bad: system is underutilized (overshoot)

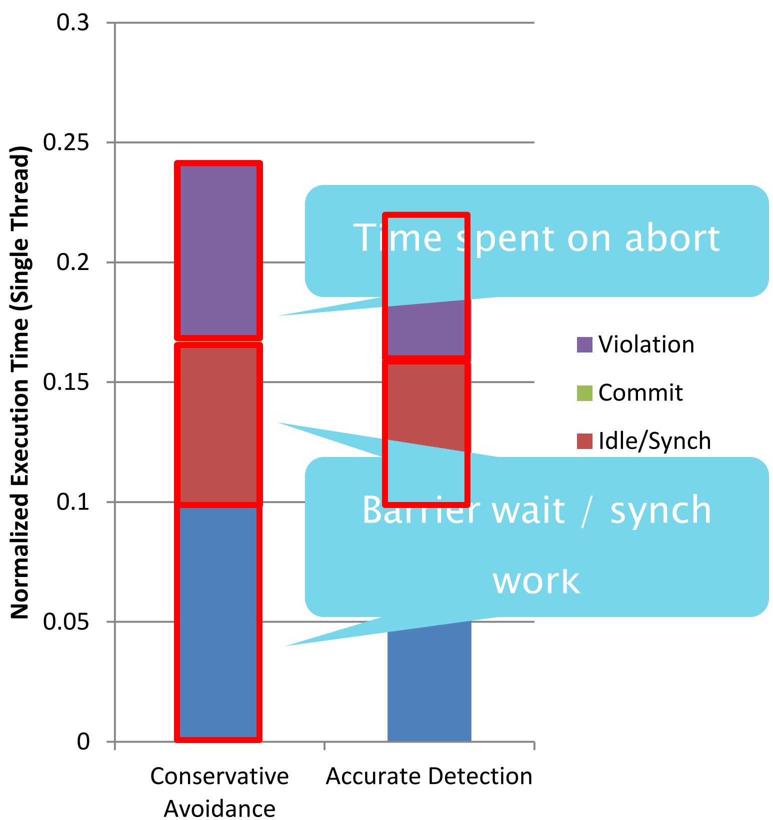
- Avoiding the overshoot problem
  - Monitor system utilization and early terminate backoff

- Borrow carrier-sensing technique from communications
  - Measure number of snooped NACK messages per period
  - Use that as an indicator for system utilization
  - Can be implemented w/ performance counter interface

Experiment Settings

- Execution-driven simulator
  - 16 x86 cores, core private L1, shared L2
  - Assume a shared bus interconnect
    - Can be generalized to directory-based environment
- Eager conflict detection HTM [moore'06] (LogTM) and hybrid TM [minh'07] (eager SigTM)
  - Both use NACK to handle conflict detection / stalling
- Work in progress
  - Results from `Genome`, `Kmeans`, and hash table
  - Plan to experiment on more workloads / larger system

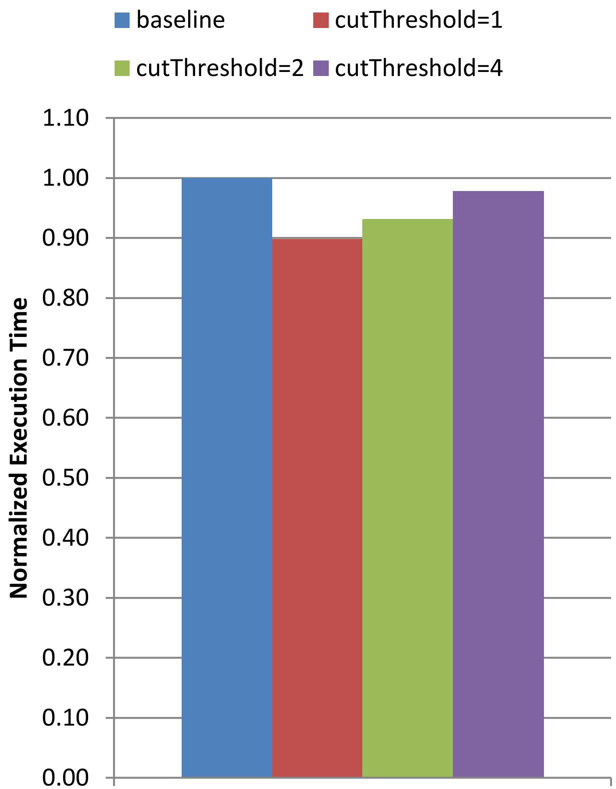
Results: Accurate Deadlock Detection



Effect of Aggressive Stalling on `Genome`

- On HTM, use ADD to perform ***aggressive stalling***
  - Transactions aggressively stall, unless arbiter overrides to abort
  - Baseline: conservative deadlock avoidance
- Many transactions eventually commit
  - Aggressive stalling reduces aborts by 20.5%
  - Improved load balance
  - 9.9% performance improvement

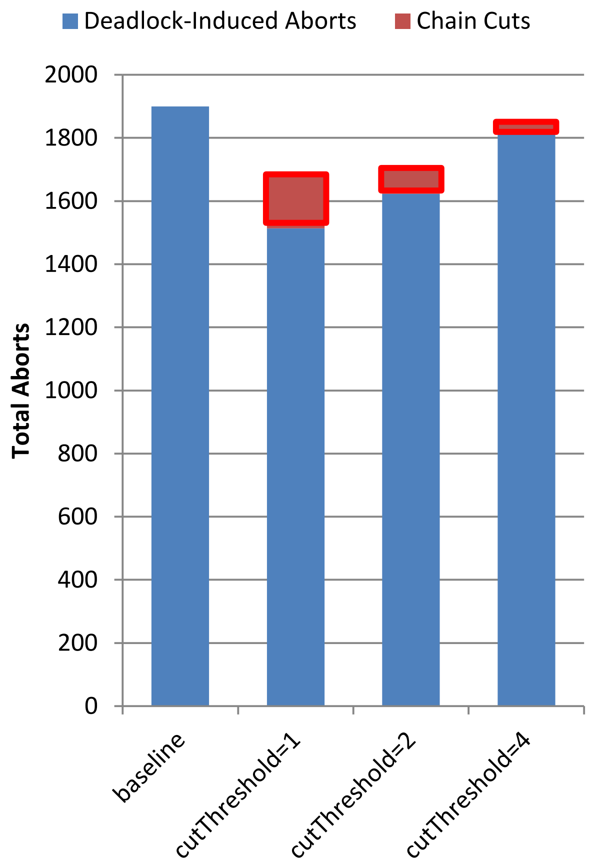
Results: Dependency Tree Construction



Effect of Dependency Chain Cutting on `Hashtable`

- Enhance HTM to maintain / propagate dependency bit vectors
- Implement ***dependency chain cutting*** mechanism
  - Abort attacker if # dependents >= `cutThreshold`
- 10% performance improvement at `cutThreshold = 1`
  - Baseline: conservative deadlock avoidance

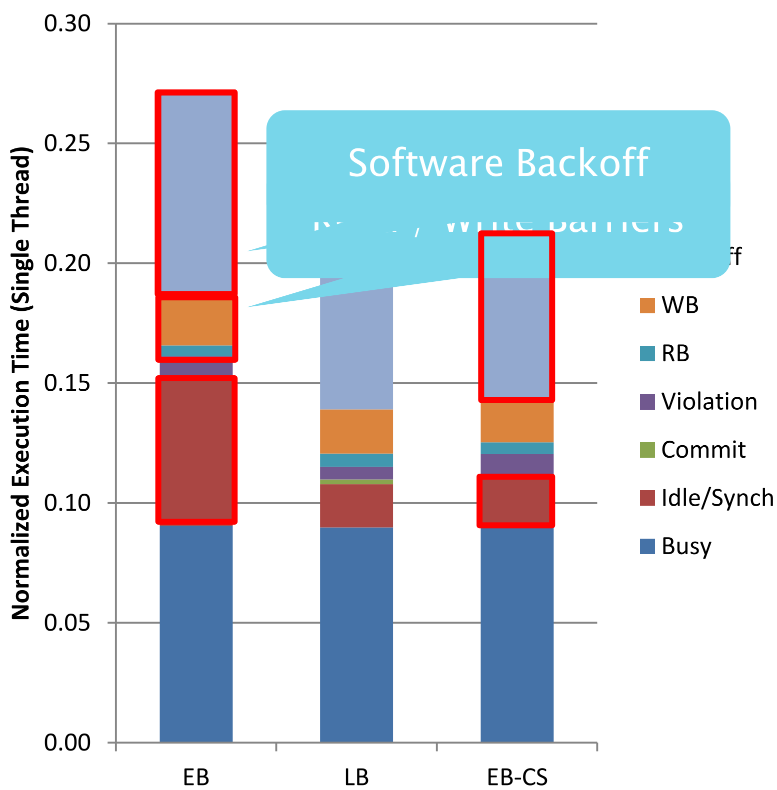
Results: Dependency Tree Construction (contd.)



Abort Breakdown on `Hashtable`

- Breakdown of aborts
  1. Those induced by conservative deadlock avoidance
  2. Proactive dependency chain cutting
- Overall performance shows high correlation to # aborts
  - Injection of chain cuts reduces total # aborts

Results: Carrier Sensing



Impact of Backoff Schemes on `Kmeans`

- 3 backoff schemes on hybrid TM
  1. Exponential Backoff (EB)
  2. EB w/ Carrier Sensing
  3. Linear Backoff (LB)
- Carrier sensing reduces wasteful backoff
  - Side effect: less load imbalance
  - 21.5% improvement compared to EB
- EB-CS matches LB
  - `Kmeans` transactions exhibit short, bursty contention
  - Best of both worlds

Conclusion

- TM concurrency controllers require low-level information
  - Dependencies among transactions
  - Utilization level of the system

- NACKs can be used to efficiently collect such info
  1. Accurate deadlock detection
  2. Dependency tree construction
  3. Carrier sensing
  - Enables advanced concurrency control

- Future work
  - Evaluate performance with more workloads / larger system
  - Investigate hardware complexity and overheads in detail

Questions?

- Pervasive Parallelism Laboratory
  - http://ppl.stanford.edu/

References

[zilles'06] C. Zilles and L. Baugh. "Extending hardware transactional memory to support nonbusy waiting and nontransactional actions." In *TRANSACT 2006.*

[moore'06] K. E. Moore et al. "LogTM: Log-based transactional memory." In *Proceedings of HPCA 2006.*

[shiu'01] P. H. Shiu, Y. Tan, and V. J. Mooney III. "A novel parallel deadlock detection algorithm and architecture." In *Proceedings of CODES 2001.*

[minh'07] C. Cao Minh et al. "An effective hybrid transactional memory system with strong isolation guarantees." In *Proceedings of ISCA 2007.*