# Transactional Memory

Memory semantics in support of parallelism

# Herlihy and Moss, ISCA 93

Transactional Memory: Architectural Support for Lock-Free Data Structures

# Problems

- Shared memory needs concurrency control
- Locking is simple, but has serious issues
  - Priority inversion (preempted low priority process holds lock needed by high priority process)
  - Convoying (process holding lock is swapped out)
  - Deadlock (mutual exclusion through multiple locks)
- Locks are atomic, but activites within them are not

# Transactions

- Goal is to let user create custom atomic operations
- Finite sequence of instructions (in a single process)
- Serializable: logically ordered and don’t interleave
- Atomicity: All changes appear at once
  - Upon completion it either commits or aborts
- Despite what the paper says, nested transactions are useful and are problematic

# Memory Access Primitives

- Load transactional (read shared mem to private reg)
- Load trans exclusive (read with hint of future update)
- Store transactional (local write, may be rolled back)
  - Read set is everything touched by LT
  - Write set is everything touched by LTX and ST
  - Data set is read and write sets

# Transaction Operations

- Commit: Try to make write set visible. Fails if anyone else has read the write set or written the data set
- Abort: Clears the write set
- Validate: Tests whether the current transaction is active or aborted

# Implementation

- Short code sequences only please!
- Works by modifying cache coherence protocol
- Separate transactional cache, small, fully associative
- Tags: Empty, Normal (committed data), XCommit (discard on Commit), XAbort (discard on abort)
- Also supports usual MSI protocol states

# Evaluation

- Simulation with a bus based coherence protocol and a directory-based network protocol
- Not validated for timing
- Microbenchmarks (Counting, Producer/Consumer with shared FIFO buffer, Doubly Linked List)
- Single active transaction per processor (32 nodes)

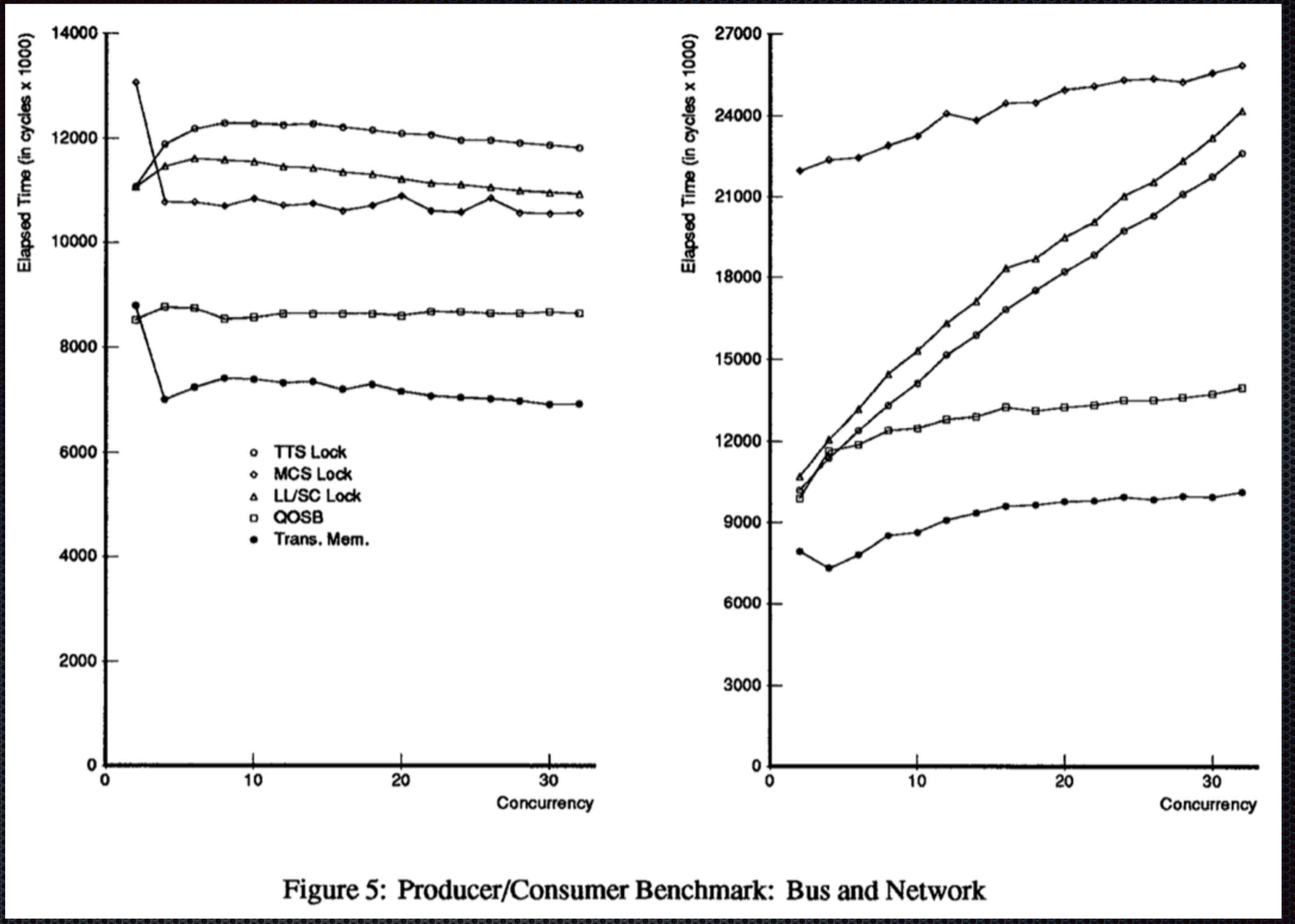


Figure 5: Producer/Consumer Benchmark: Bus and Network

TTS: Spin lock, MCS: software queueing, LL/SC: Load-Linked/ Store Conditional, QOSB: Hardware queueing, Trans. Mem.

# Discussion

# Nakalke ISCA 2015

## Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and Power 8

# 22 Years Later...

- Software TM has been tried
- Hardware TM support on four systems
- STAMP benchmarks test transaction performance

# Conflict Detection Granularity

- When cache lines are tagged (as in the original proposal) they can contain values from different transactions, causing false conflicts
- Cache lines have grown (64 to 256 bytes)
- Note that these are not using a separate cache

# Transaction Capacity

- Maximum data a transaction can access
- Space for conflict detection, uncommitted writes
- Blue Gene: 20MB load, 20MB store (1.25 MB/core)
- zEC12: 1MB load, 8KB store
- Core i7: 4MB load, 22KB store
- Power 8: 8K load, 8 KB store

What happened to the idea that transactions are small?

# Transaction Retry

- On transaction abort, can retry
  - Nothing to prevent infinite retries
- Fall back to a global lock to force an irrevocable transaction to take place
- Ensures a transaction completes
- Forces others to wait

# STAMP Benchmarks

- Badly coded
- Many false conflicts
- Many unnecessary aborts
- Hardware TM support doesn’t help
- Fixed problems and created own version
- Only report speedup, not absolute speed

# System Configurations

16-core 1.6-GHz A2 with 4 SMT threads (Blue Gene/Q), V1R2M2, 16 GB RAM

16-core 5.5-GHz zEC12, z/OS V2.01, 64 GB RAM

4-core 3.4-GHz Core i7-4770 with 2 SMT threads, Linux 3.14.5, 4 GB RAM

6-core 4.1-GHz POWER8 with 8 SMT threads, AIX 7.1.3.16, 28.5 GB RAM

# Speedup Ratio (1 thread)

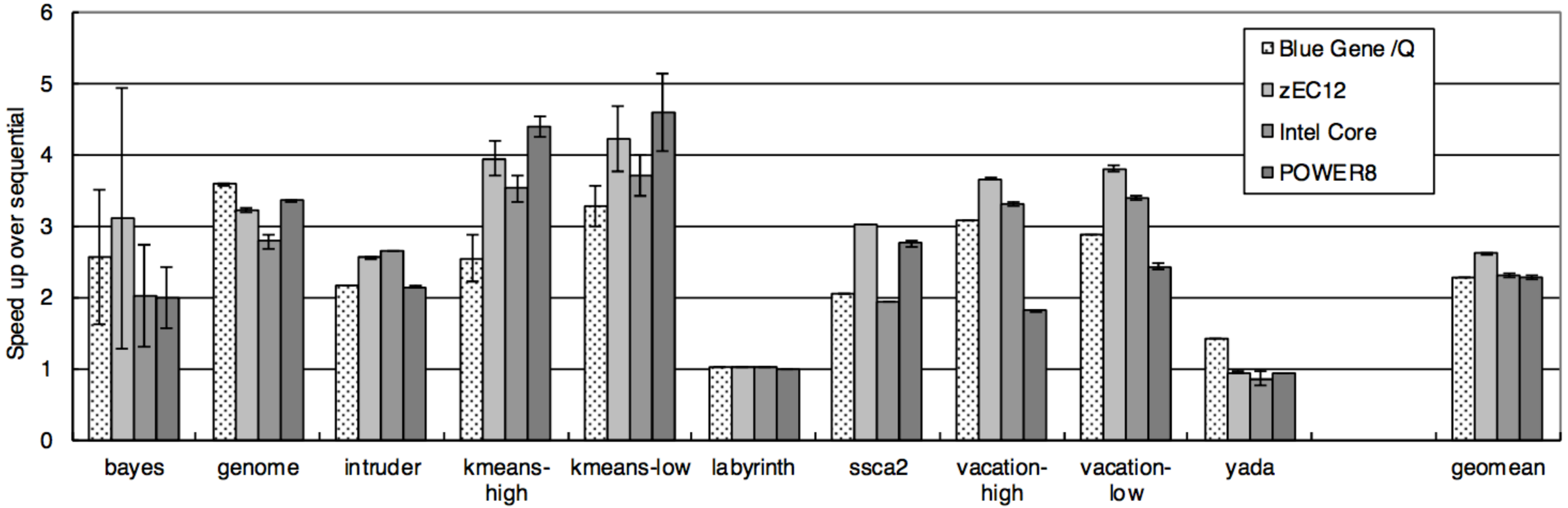


**Figure 2. Speed-up ratios of transactional execution over serial execution in Blue Gene/Q, zEC12, Intel Core, and POWER8 with 4 threads. Our modified STAMP benchmarks were used.**

# Core i7

- Prefetching looks for access patterns and loads cache lines
- Doesn’t distinguish prefetch from transaction accesses
- Results in extra aborts
- Disabling reduces aborts from 16% and 24% on kmeans benchmarks to 10%

# Speedup (4 threads)

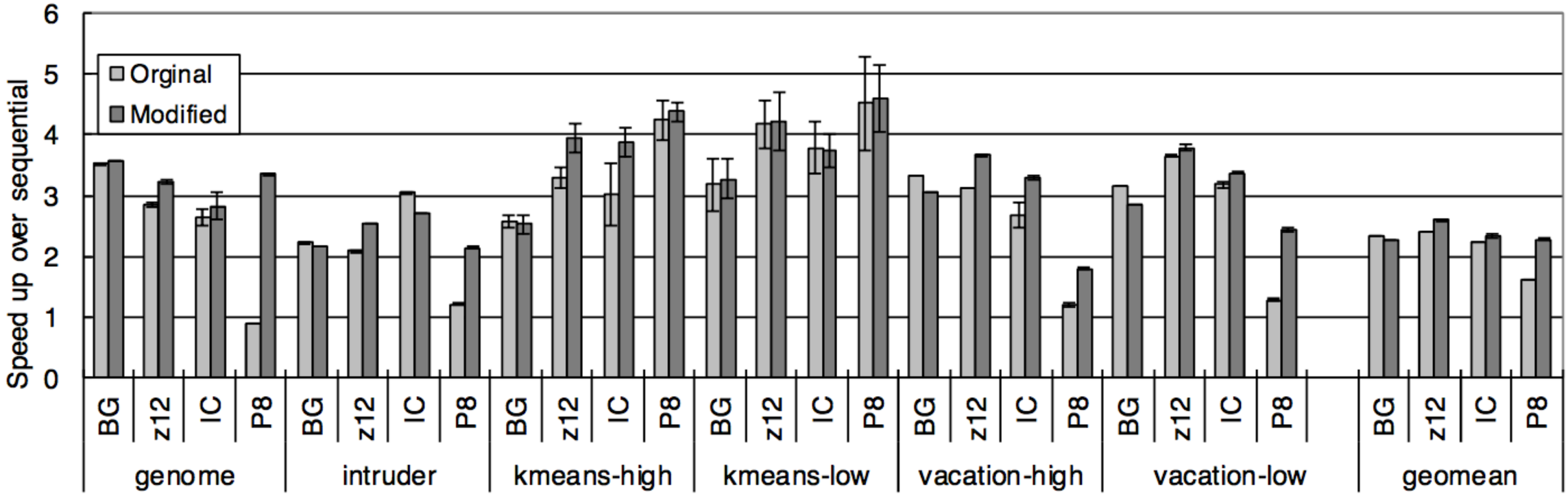


**Figure 4. Speed-up ratios of transactional execution over sequential execution in Blue Gene/Q (BG), zEC12 (z12), Intel Core (IC), and POWER8 (P8) with 4 threads. The two bars are the speed-up ratios of the original and modified versions of the STAMP benchmarks, respectively.**

Selected benchmarks, limited speedup

# Speedup (More threads)

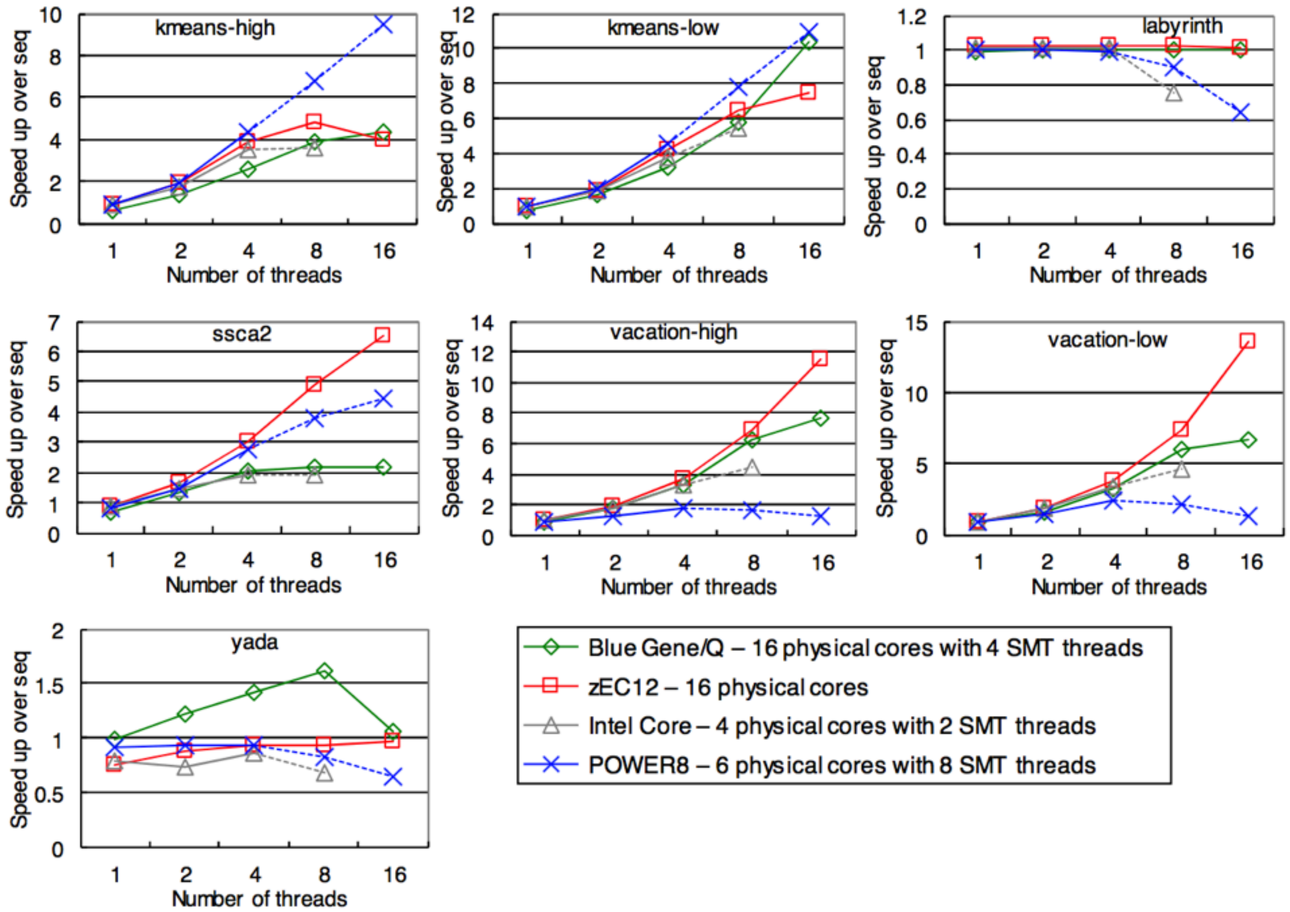


**Figure 5. Speed-up ratios of transactional execution over serial execution in Blue Gene/Q, zEC12, Intel Core, and POWER8. Our modified version of the STAMP benchmarks was used with 1, 2, 4, 8, and 16 threads. Dotted lines are the speed-up ratios when the number of physical cores is smaller than the number of threads.**

# Conclusions

- No clear winner
- Scaling isn’t consistent
  - Different processors scale on different benchmarks
- In some cases, scaling limited by transaction capacity
- In others by abort rates

# Recommendations

- Make conflict detection more precise (reduce false positives)
- Make it easier for HTM to gracefully scale to using software TM
- Enable tagging non-transaction accesses to avoid false conflicts
- Increase transactional store capacity

# Discussion