The Common Case Transactional Behavior of Multithreaded Programs

JaeWoong Chung
Hassan Chafi, Chi Cao Minh,
Austen McDonald, Brian D. Carlstrom,
Christos Kozyrakis, Kunle Olukotun

Computer Systems Lab
Stanford University
http://tcc.stanford.edu

The Parallel Programming Problem

- CMPs are here but no parallel software to run on them
- Lock-based parallel programming is simply broken
  - Coarse-grained locks: serialization
  - Fine-grained locks: deadlocks, races, priority inversion, …
  - Poor composability, not fault-tolerant, …
- Transactional Memory (TM): an promising alternative
  - Transactions: atomic & isolated access to shared-memory
  - Performance through optimistic concurrency
- Parallel programming with TM
  - Coarse grain **Non-blocking synchronization** for parallel algorithms
  - **Speculative parallelization** for sequential algorithms

The Design Space for TM

- A transactional memory system provides
  - Basics: versioning, conflict resolution, commit, abort
  - Desired: nesting for libraries, virtualization
- Several proposed designs
  - Software-only: [DSTM], [OSTM], [ASTM], [SXM], [McRT-STM]
  - Hardware-assisted: [TLR], [TCC], [U/LTM], [VTM], [LogTM]
  - Hybrids: [HyTM], [Hybrid-TM]
  - Different tradeoffs in implementing basic/desired features
- **Key questions**
  - **Which is the common case to optimize for?**

In Search of the Common Case

- Key metrics of transactional program
  - Transaction length
    - Cost of fixed overheads, time virtualization issues
  - Read-/write-set size
    - Buffer space requirements, buffer virtualization issues
  - Write-set to length ratio
    - Amortize commit/abort overheads
  - Frequency of nesting & I/O in transactions
    - Support for nesting, syscalls, …
  - Frequency of conflicts
    - Scheduling and contention management policies

- The "chicken & egg problem"
  - Programmers need efficient TM systems to support development
  - Designers need TM applications to derive common case

- Can we break the deadlock?

Paper Summary

- Study the TM behavior of existing parallel programs
  - Map existing parallel constructs to transactions
  - 36 applications, multiple domains, 4 programming models

- **Analyzed common case** for
  - Transaction length, read-/write-set size, write-set to length ratio, nesting & I/O
  - For both non-blocking synchronization & spec. parallelization
  - Implementation agnostic measurements

- **Derived guidelines** for TM system design
  - Buffering requirements and virtualization approach
  - Overhead amortization, nesting & I/O support, …

Methodology Overview

- Key assumption
  - The inherent parallelism & synchronization patterns are likely the same regardless of language primitives used
- Methodology
  1. Trace parallel application on existing hardware
  2. Map parallel constructs to transaction boundaries
     - E.g. lock/unlock -> transaction begin/end
  3. Process trace to analyze metrics
- Measurements are agnostic to TM design
  - Limitation: cannot measure violation behavior

Parallel Applications

Transaction Usage	Languages	Applications
Non-blocking Synchronization	Java	MolDyn, MonteCarlo, RayTracer, Crypt, LUFact, Series, SOR, SparseMatmult, SPECjbb2000, PMD, HSQLDB
	Pthread	Apache, Kingate, Bp-vision, Localize, Ultra Tic Tac Toe, MPEG2, AOL Server
	ANL	Barnes, Mp3d, Ocean, Radix, FMM, Cholesky, Radiosity, FFT, Volrend, Water-N2, Water-Spatial
Speculative Parallelism	OpenMP	APPLU, Equake, Art, Swim, CG, BT, IS

- Different domains : scientific, enterprise, AI/robotics, multimedia
- Different qualities : highly optimized Vs. less optimized
- Java, Pthreads, ANL → studied for non-blocking synchronizations
- OpenMP → studied for speculative parallelization

Non-Blocking Synchronization

- Transactions are used **for critical sections in parallel algorithms**
- Original primitives mapped to transactional boundaries
  - E.g. Java synchronized block, pthread_mutex, ANL LOCK macro mapped transaction boundaries
- Semantics issue
  - To conserve the original program semantics, ***wait*** splits transaction
  - This mapping is not always safe, but was fine in our study

Original Threading Primitive	Transaction Mapping
Lock	**BEGIN**
Unlock	**END**
Wait	**END-BEGIN**

Transaction Length

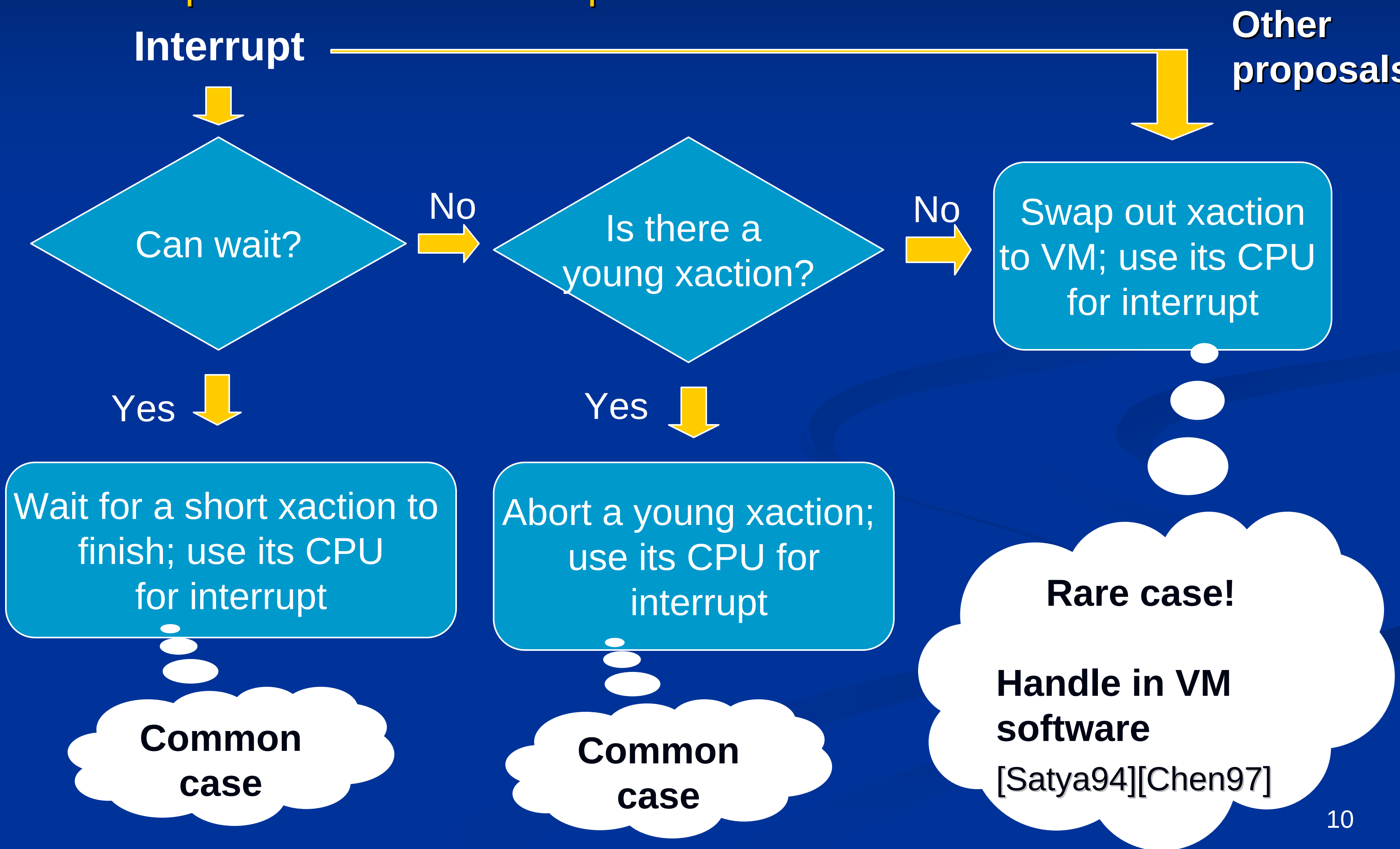
- Number of instructions executed in transaction

Application	Length in Instructions			
	Avg	50th %	95th %	Max
Java average	5949	149	4256	13519488
Pthreads average	879	805	1056	22591
ANL average	256	114	772	16782

- Up to **95%** of transactions have less than **5000 instructions**
  => **Light-weight transactional primitives** are required
- Some programs have **rare but long transactions**
  => **Time virtualization** is needed (transaction context-switching)

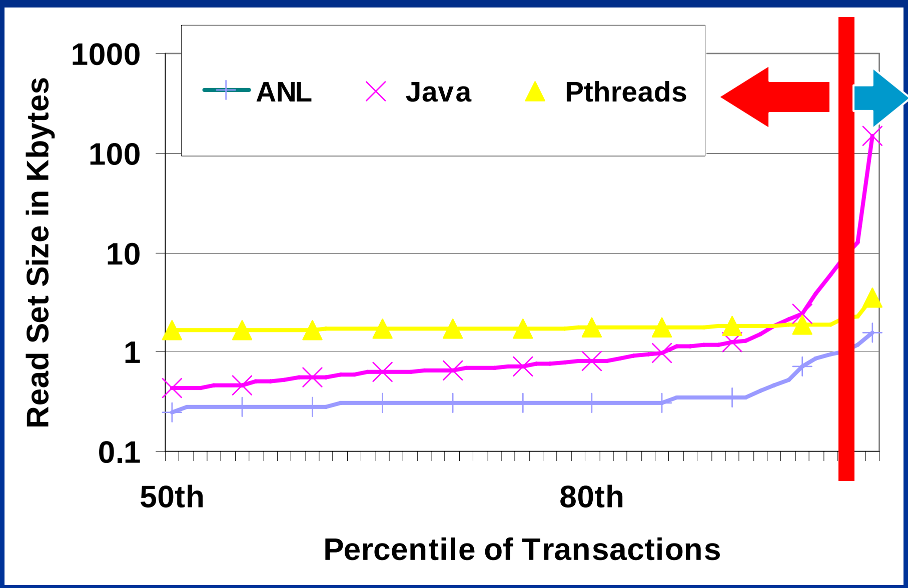
Time Virtualization for TM

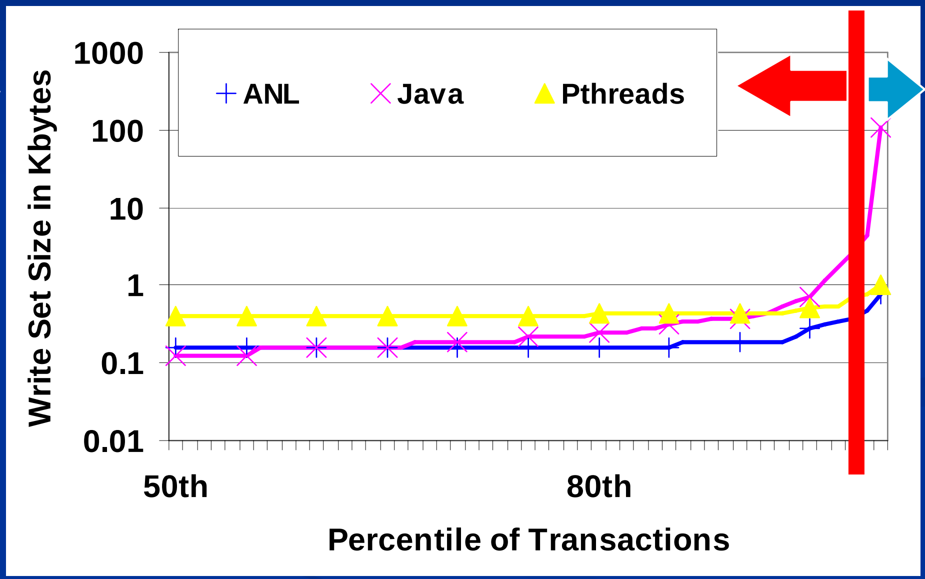
- Interrupt and context-switch procedure



Read-/Write-Set Size

- Bytes of data read/written by transaction

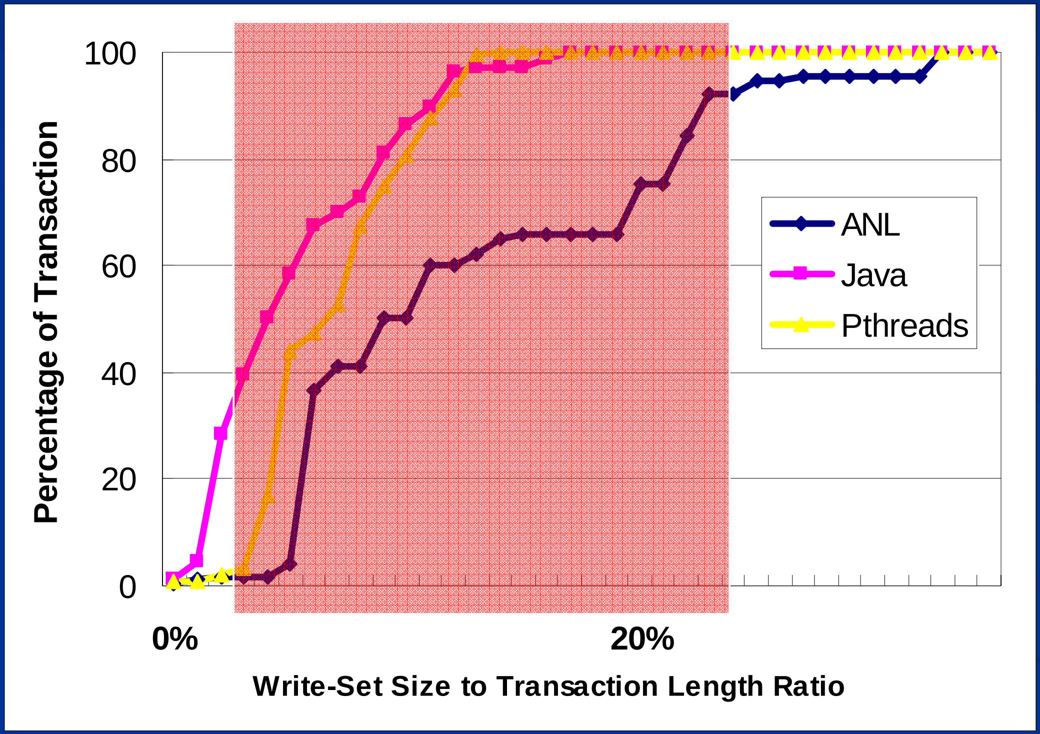




- **98% of transactions: <16KB read-set, <6KB write set**
  => **<u>32K L1 Cache</u>** will be enough for most transactions
- There are **few very large transaction** > 32K
  => **<u>space virtualization</u>** is needed but it's better be cheap

Write-Set to Length Ratio

- Ratio of # unique addresses written to # instructions in transaction



- **< 25% in most transactions**

  => Big challenge for SW TM because of high per-write overhead

  => Even HW TM needs sufficient bandwidth for versioning and commit

Transaction Nesting and I/O

- Nesting occurs only in java VM code
  - 2.2 average depth

  => **Limited support** for nesting is sufficient for now

- I/O within transactions is rare
  - 27 applications have less than 0.1% of transactions with I/O
  - 8 applications have up to 1% of transactions with I/O
  - No transactions include both input and output

  => **Buffered I/O** would not deadlock

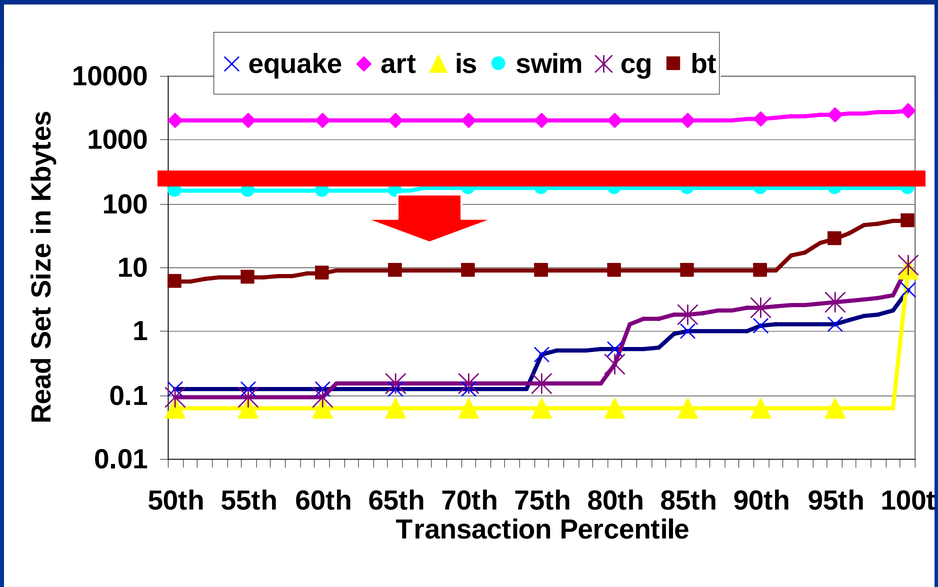
Speculative parallelization

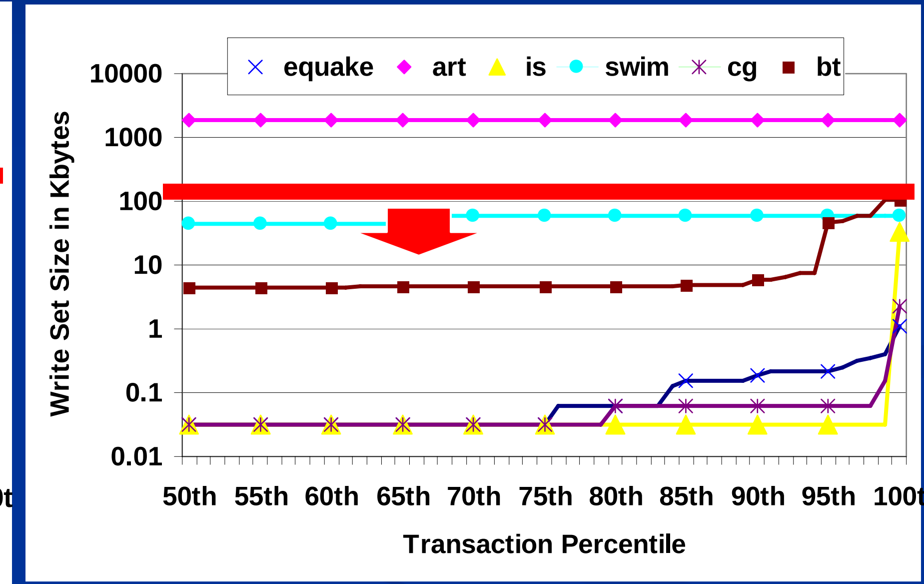
- Speculatively parallelize loops in sequential algorithms
  - E.g. each loop iteration becomes a transactions
- This study
  - 6 loop based applications
  - Mapped outermost loop iteration to single transaction

Original Threading Primitive	Transaction Mapping
Outermost Iteration Start	BEGIN
Outermost Iteration End	END

Read-/Write-Set Size

- Bytes of data read/written by transaction





- The read-/write-sets get larger up to L2-sized buffers (~128K)
  => They doesn't fit in L1 cache but still fits into **L2-sized buffer**
  => **Inner loop parallelization** might be better to reduce buffer requirement

Take-away Points

Transaction Usage	Observation	TM Design Guidelines
Non-blocking Synchronization	Short-lived transactions	Light-weight TM primitives
	Read-/write-sets < 16K	L1 cache for versioning
	High write-set to length ratio	Per write overhead is critical Challenge for STM
	Few nested transactions	Limited nesting support
	Few transactions with I/O	Buffered I/O
Speculative Parallelism	Large read-/write-sets	L2 cache for versioning

Conclusion

- Extensive study of transactional behavior of programs
  - 36 parallel applications from multiple domains
  - Map existing parallel constructs to transactions
  - Covered both non-blocking synchronization & speculative parallelization

- Contributions
  - **Quantitative Observations** on transactional characteristics
    - Most transactions are short-lived, small, and not nested
  - **Design Guidelines** for Transactional Memory systems

Effective Guideline for TM Architects

Questions?

Jae Woong Chung
jwchung@stanford.edu

Computer Systems Lab.
Stanford University
http://tcc.stanford.edu