Lecture 17: Transactional Memories I

Papers:

- A Scalable Non-Blocking Approach to Transactional Memory, HPCA'07, Stanford
- The Common Case Transactional Behavior of Multi-threaded Programs, HPCA'06, Stanford
- Characterization of TCC on Chip Multiprocessors, PACT'05, Stanford

TM Overview

- Recall the basic TM implementation:
  - Every transaction maintains read-set and write-set
  - Writes are not propagated until commit time
  - At commit, acquire permission to commit from a central arbiter (no parallel commit for now)
  - Send write-set to other nodes – if the write-set intersects with the node's read-set, the node's transaction is aborted and re-started

Implementation Issues

- Design details:
  - On attempting a write (the Tx is not yet ready to commit), must obtain the most recently committed version of the block in read-only state (the block is not yet part of the read-set, though)
  - At the time of commit, either write-thru to memory (there is no M state in the coherence protocol), or move from S to M state (write-back policy) (a dirty line can't handle speculative writes, though)
- For parallel commits:
  - is an ordering implied between these transactions?
  - is a write-set/read-set conflict allowed?
  - is a read-set/write-set conflict allowed?
  - is a write-set/write-set conflict allowed?

Parallel Commits I

- Ordering is implied: a programmer believes that a transaction executes "in isolation"

- Write-set/Read-set conflict should cause an abort: hence, the second transaction must:
  - confirm there is no conflict before propagating writes
  - or propagate writes in a manner that does not affect correctness (can't employ write-thru or write-update, can't respond to others' read requests / or must keep track of dependences)

- Read-set/Write-set conflict need not cause an abort: the ordering should indicate that the first transaction need not abort

- Write-set/Write-set conflict need not cause an abort: a mechanism is required to ensure that everyone sees writes in the same correct order

* Reading your own write is truly not a problem, but since info is maintained at block granularity, the block is included in the read-set

Parallel Commits II

- Conflicting writes must be merged correctly (note that the writes may be to different words in the same line)

- If we're not checking early for Write-set/Read-set conflicts, the first transaction must inform the next transaction after it is done (received all acks)

- Consistency model: two parallel transactions are sending their write-sets to other nodes over unordered networks:
  - Just as we saw for SC, a reader must not proceed unless everyone has seen the write (the entire transaction need not have committed)
  - Writes to a location must be serialized

The Stanford Approach

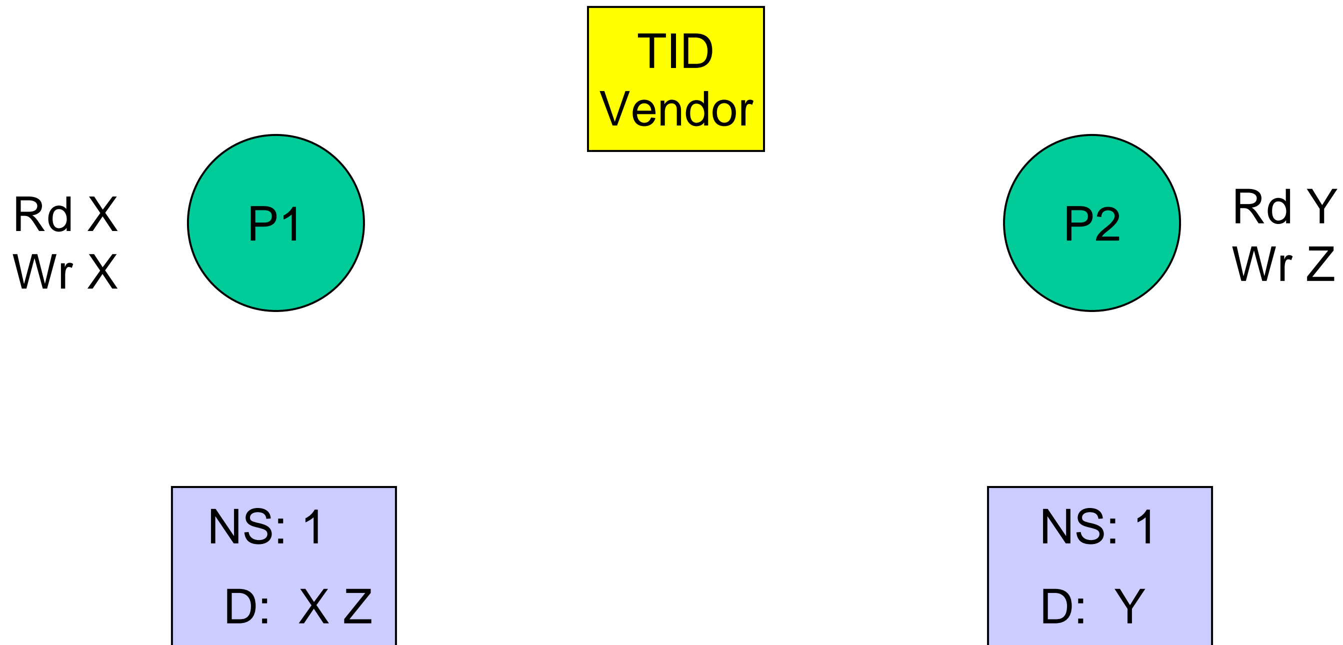
- Transactions are ordered (by contacting a central agent)

- A transaction first engages in validation and proceeds with commit only if it is guaranteed to not have any conflicts
  - Write-set/Read-set conflicts are not allowed (T_i checks its read-set directories and proceeds only after those directories have seen previous writes)
  - Read-set/Write-set conflicts are allowed (Ti will ignore write notices from transactions >i)
  - Write-set/Write-set conflicts are allowed To maintain write serialization, T_i confirms that a directory is done with all transactions <i

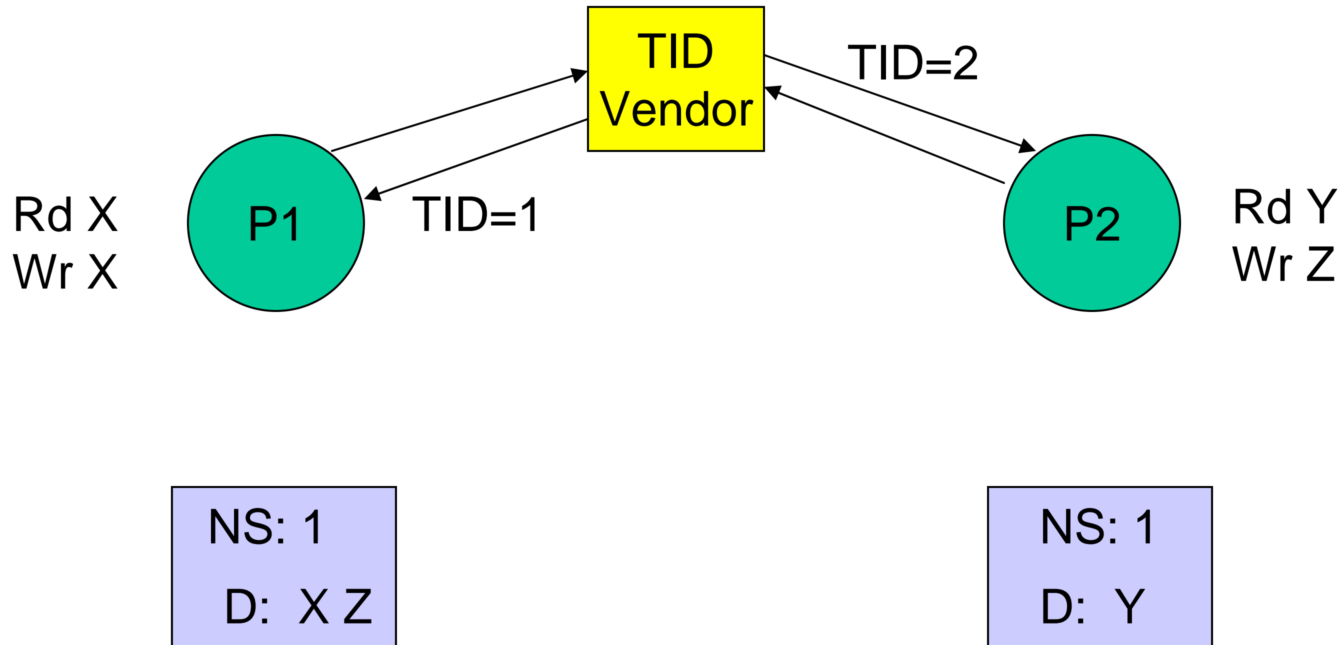
Algorithm

- Probe your write-set to see if it is your turn to write (helps serialize writes)
- Let others know that you don't plan to write (thereby allowing parallel commits to unrelated directories)
- Mark your write-set (helps hide latency)
- Probe your read-set to see if previous writes have completed
- Validation is now complete – send the actual commit message to the write set

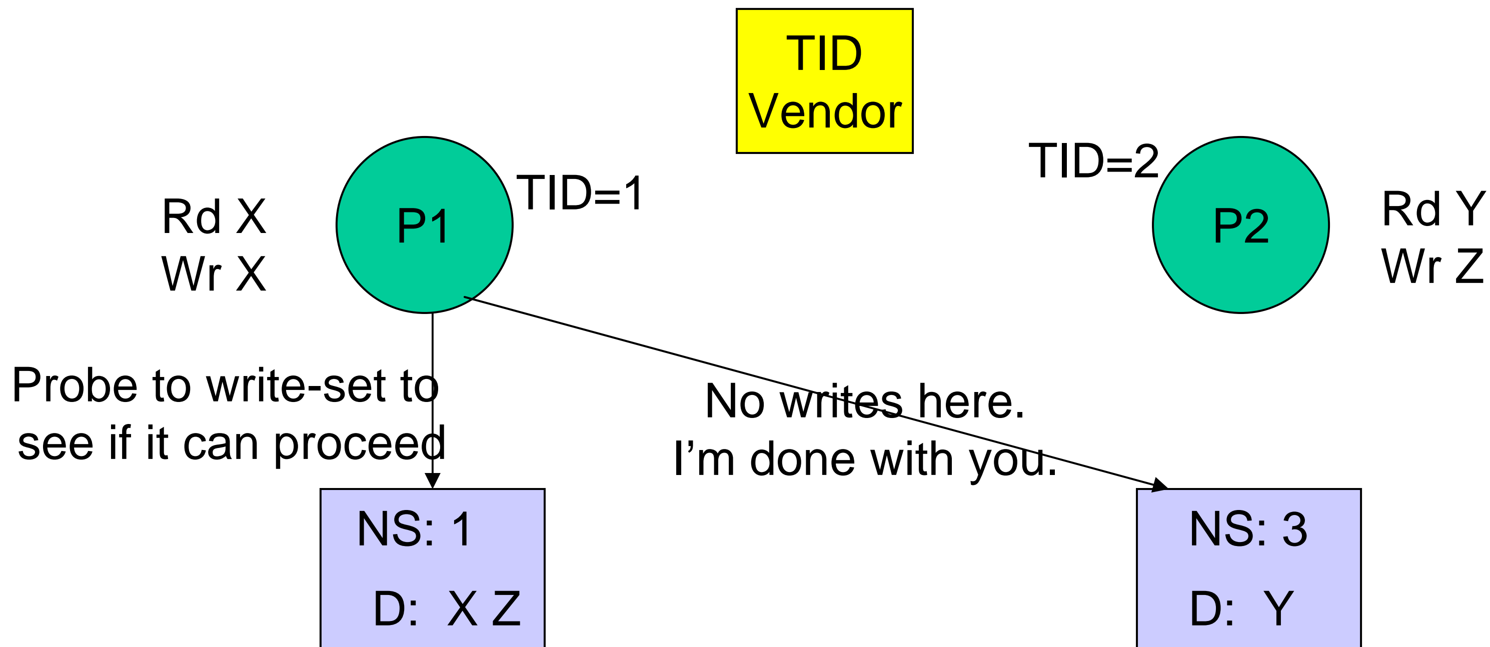
Example



Example

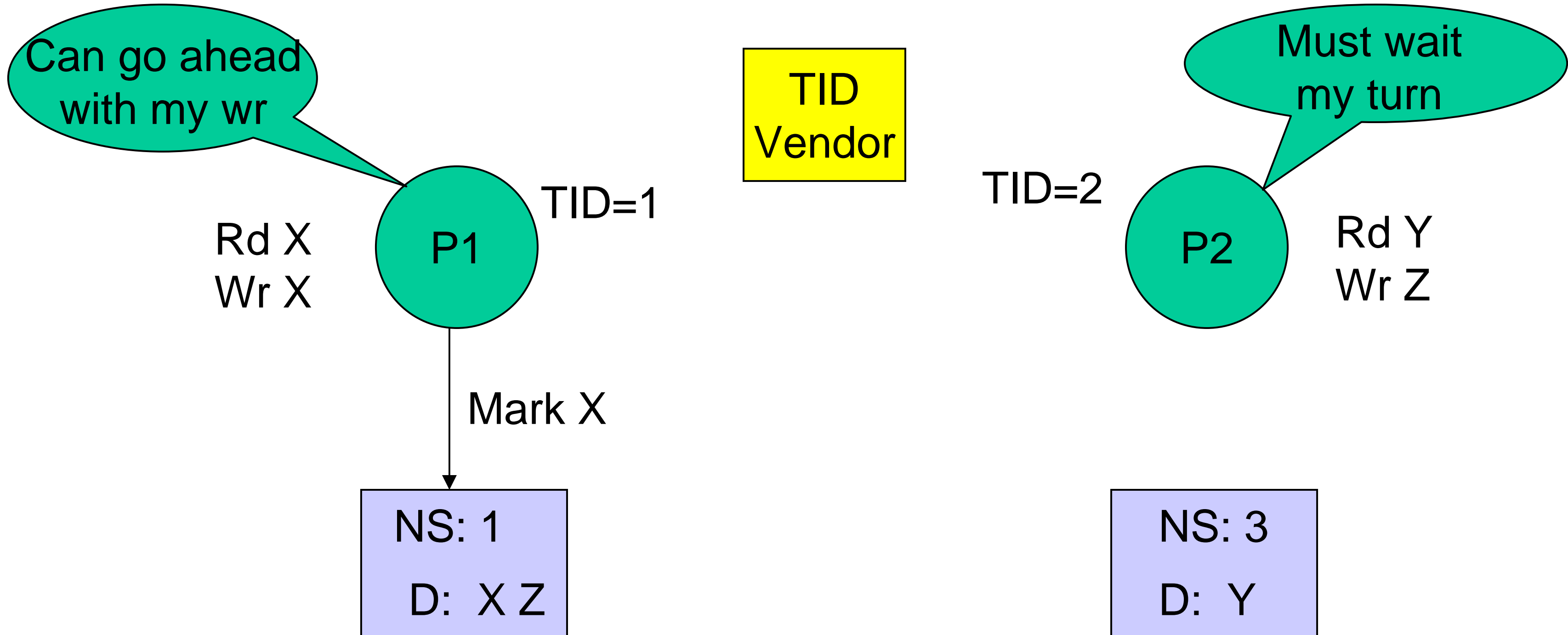


Example



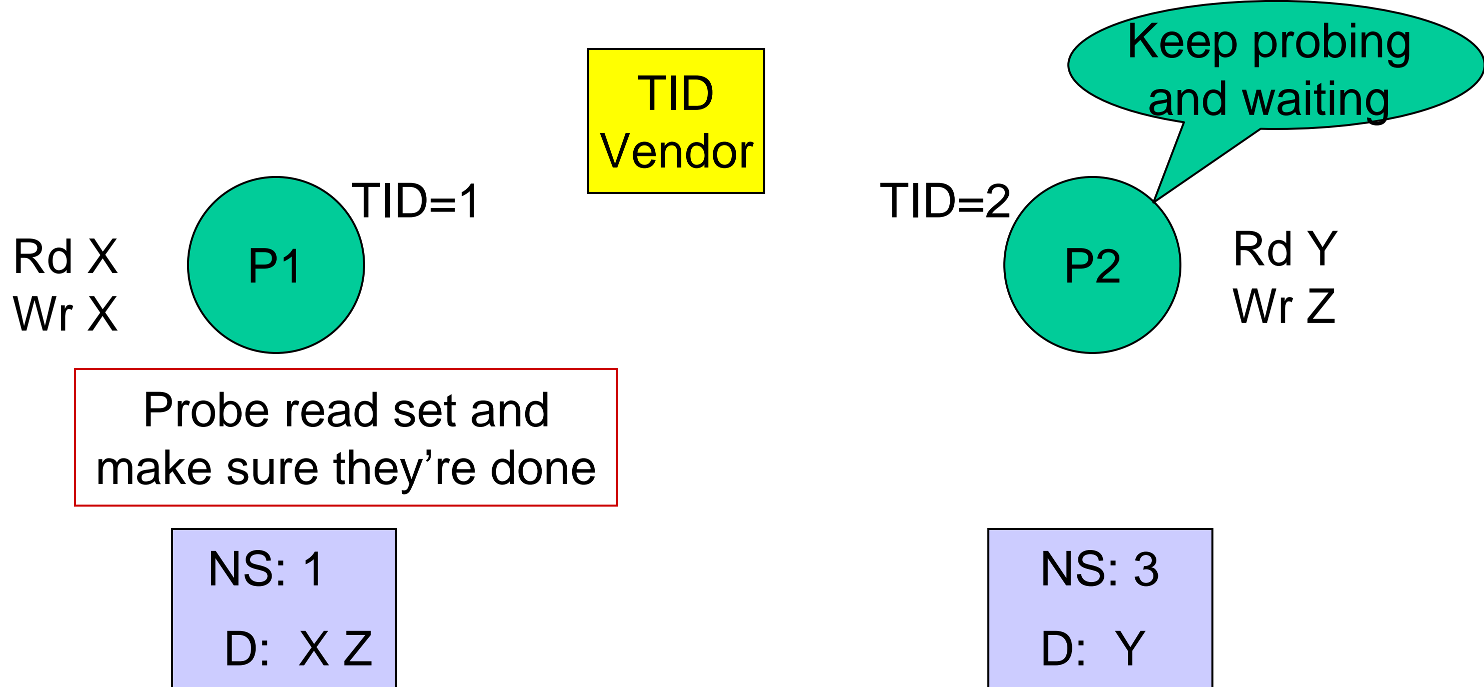
P2 sends the same set of probes/notifications

Example

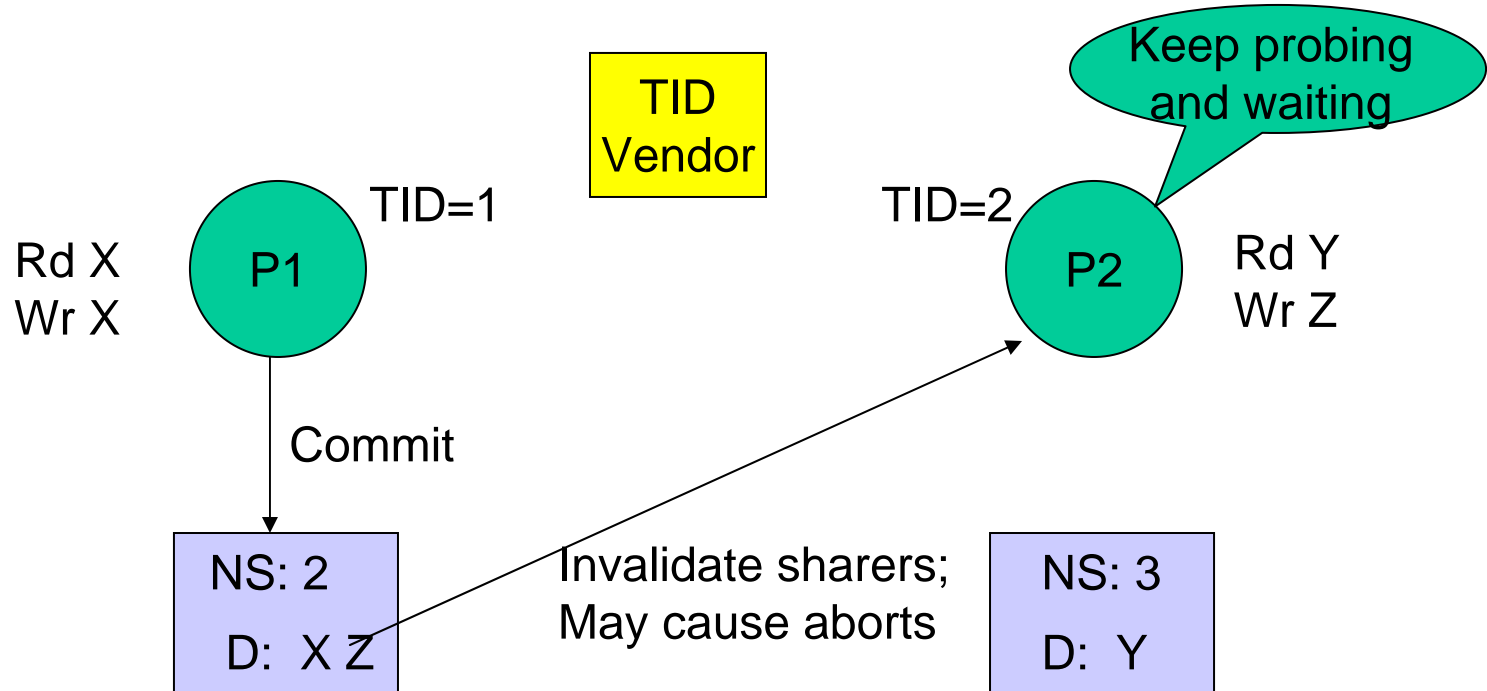


Mark messages are hiding the latency for the subsequent commit

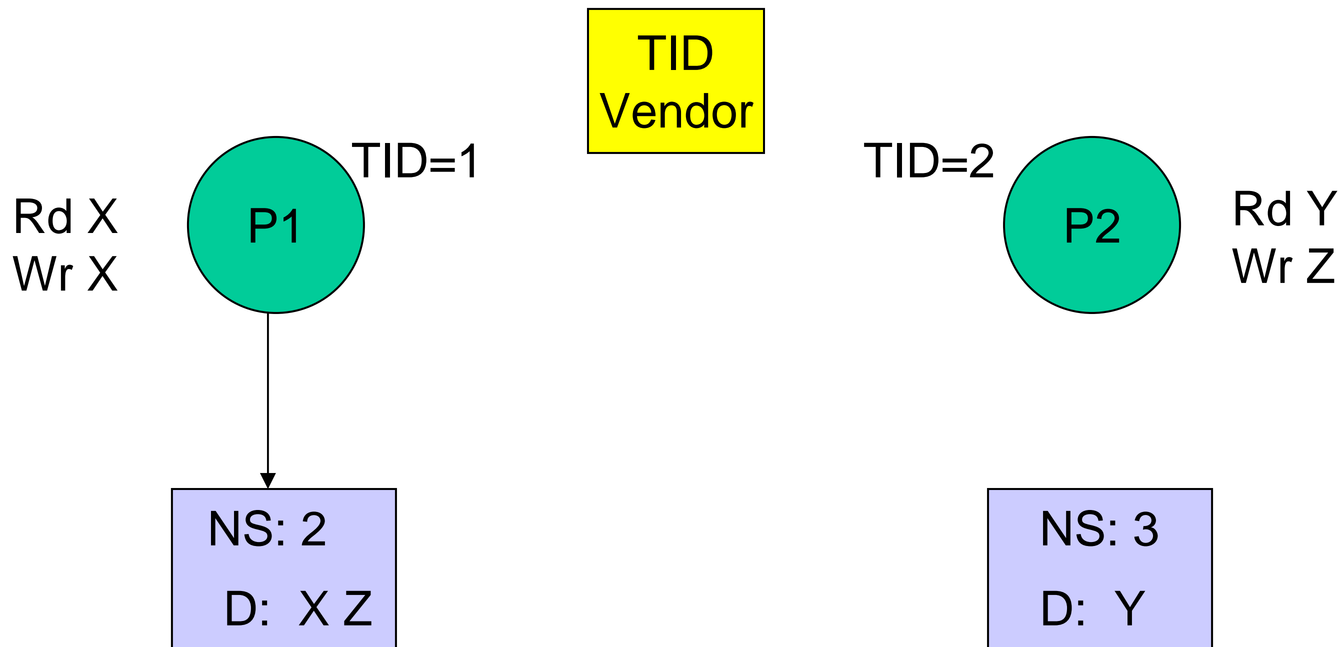
Example



Example



Example



P2: Probe finally successful.
Can mark Z.
Will then check read-set.
Then proceed with commit

Algorithm

- Probe your write-set to see if it is your turn to write (helps serialize writes)
- Let others know that you don’t plan to write (thereby allowing parallel commits to unrelated directories)
- Mark your write-set (helps hide latency)
- Probe your read-set to see if previous writes have completed
- Validation is now complete – send the actual commit message to the write set

Issues

- The protocol requires many messages: enables latency hiding for the commit process, though
- There may be high directory locality for a NUMA machine, but possibly not for a single large-scale CMP with a directory-based cache coherence protocol
- To support a write-back cache policy, a new non-speculative cache is introduced (so that a transaction does not speculatively over-write a dirty line)

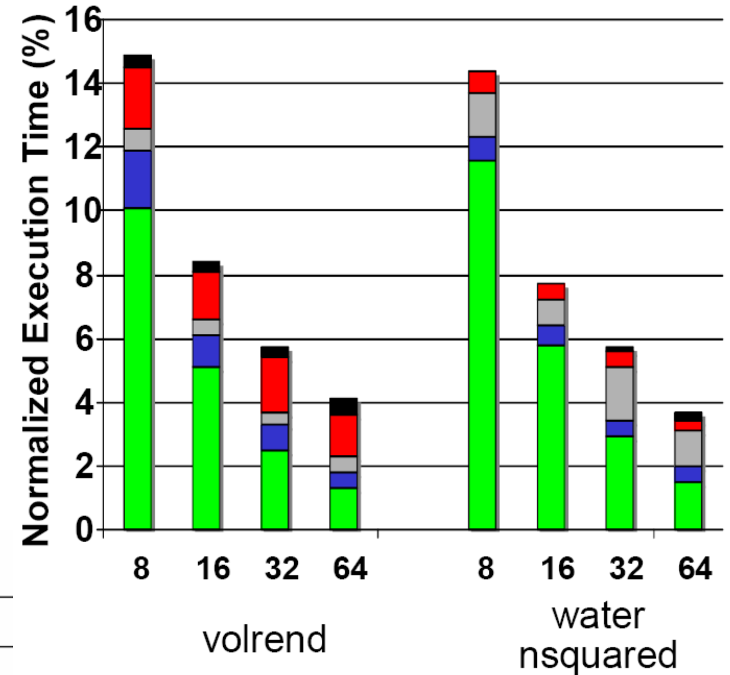
Evaluation

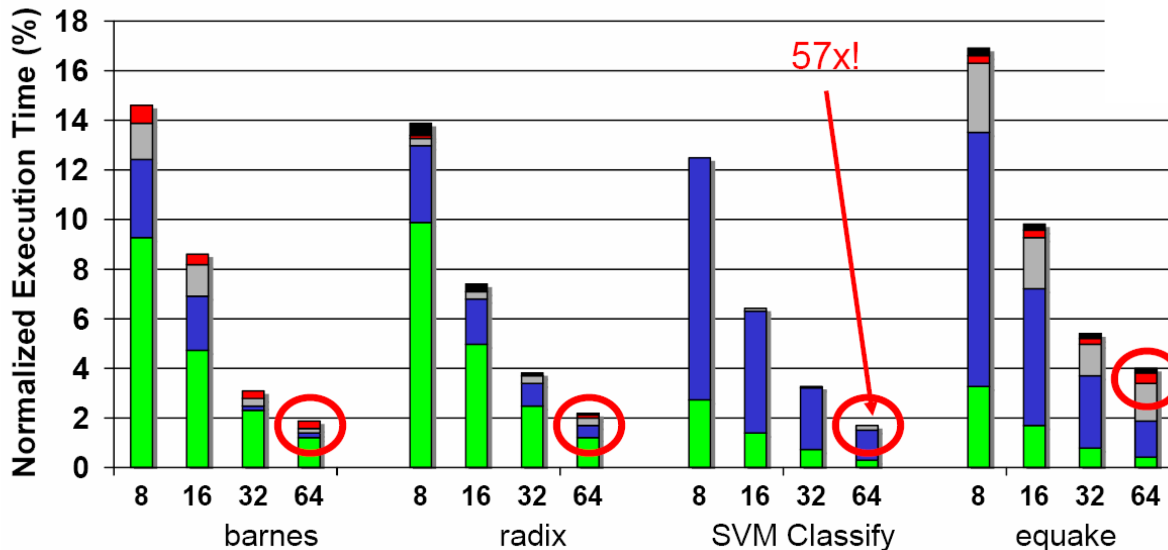
Application	Input	Trans. Size 90th % (Inst)	Trans. Wr. Set 90th % (KB)	Trans. Rd. Set 90th % (KB)	Ops. per Word Written 90th %	Directories per commit 90th %	Working set (Dir.) 90th % (Entries)	Directory Occupancy 90th % (Cycles)
barnes [39]	16,384 mol.	7,462	0.66	1.69	11.04	1	384	813
Cluster GA [5]	ref	238	0.01	0.14	7.25	1	266	354
equake [34]	ref	866	0.35	1.73	11.00	3	8926	485
radix [39]	1M keys	32,681	7.41	8.16	17.20	32	13725	643
SPECjbb2000 [35]	1,472 trans.	5,556	0.12	0.16	180.60	2	14422	229
SVM Classify [5]	ref	13,054	0.62	9.72	84.27	2	61	248
swim [34]	ref	45,876	18.00	52.00	9.60	1	941	765
tomcatv [34]	ref	21,060	10.50	15.90	7.83	2	1572	426
volrend [34]	ref	1,098	0.31	0.39	2.09	1	977	560
water-nsquared [39]	512 mol.	948	0.45	0.45	8.12	1	139	323
water-spatial [39]	512 mol.	7,466	1.26	1.27	23.14	2	752	312

Table 3. Applications and their scalable TM characteristic for performance. The 90th percentile transaction size in instructions, transaction write- and read-set sizes in KBytes, and operations per word written. We also show the number of directories touched per commit and the 90th percentile of both the working set cached at the directory in number of entries and the directory's occupancy in cycles per commit

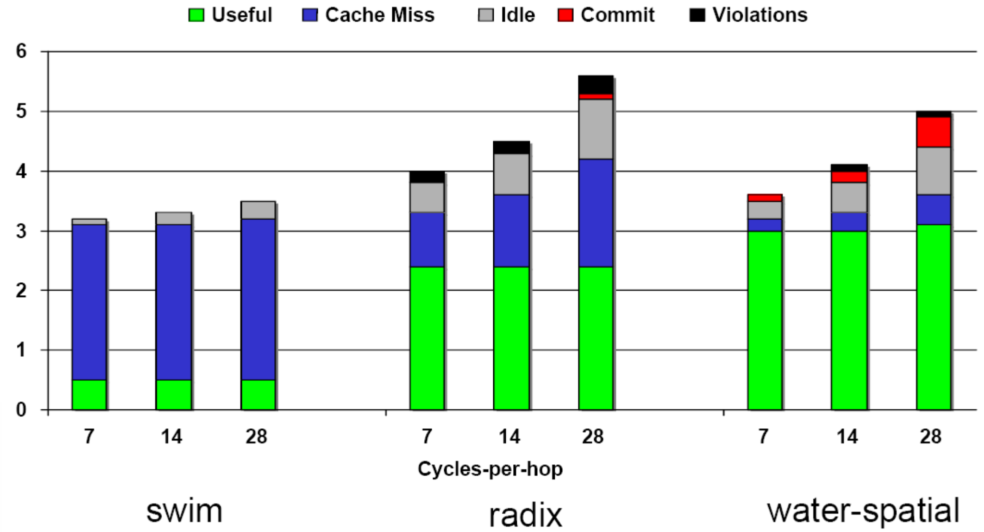
Results

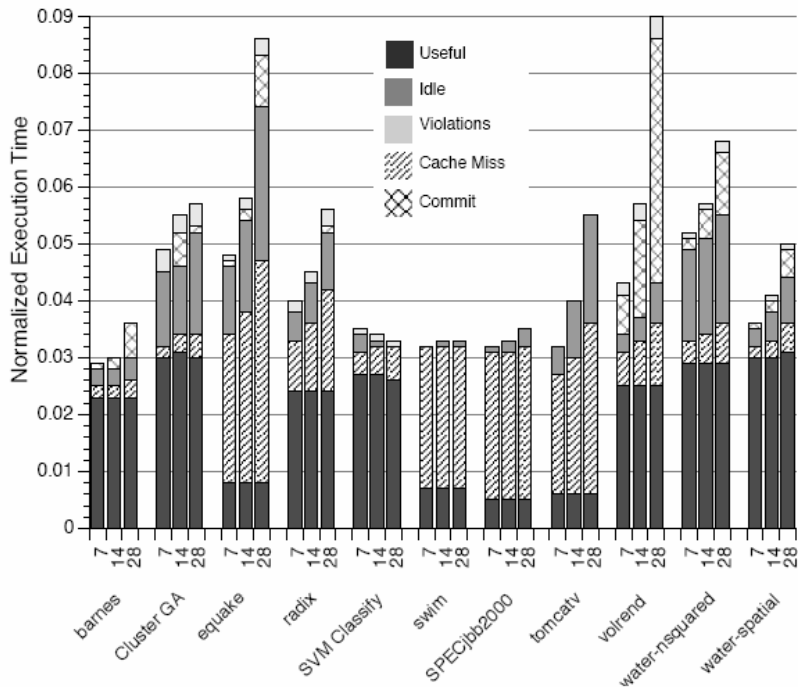
Applications with small transactions suffer more from commit latency





Results





Transaction Characteristics

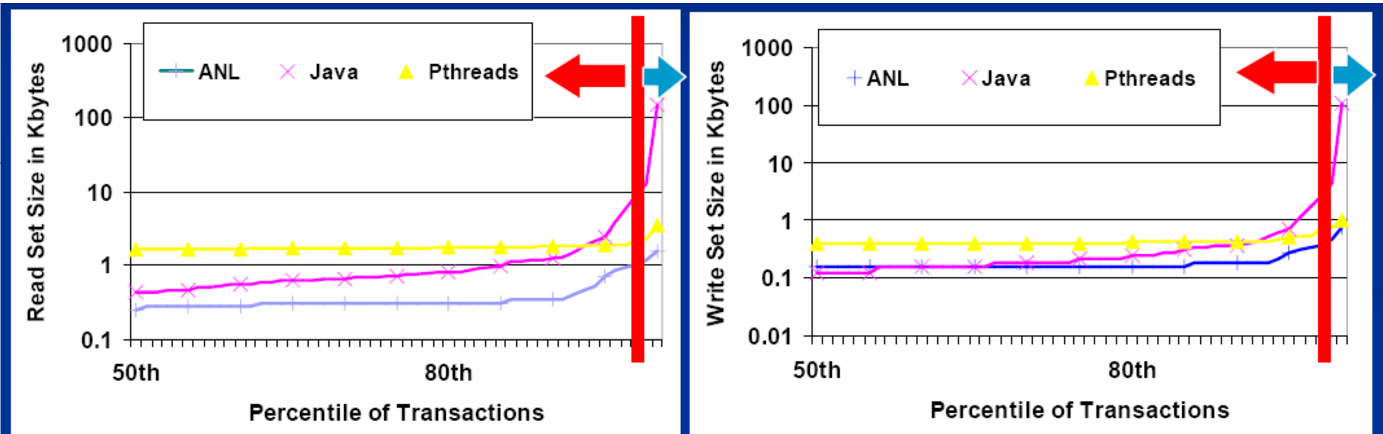
- An evaluation of 35 multi-threaded programs

Transaction length →

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**

Sizes of read and write sets →



- 98% of transactions: <16KB read-set, <6KB write set

Transaction Characteristics

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

  => <u>Limited support</u> 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

Title

- Bullet