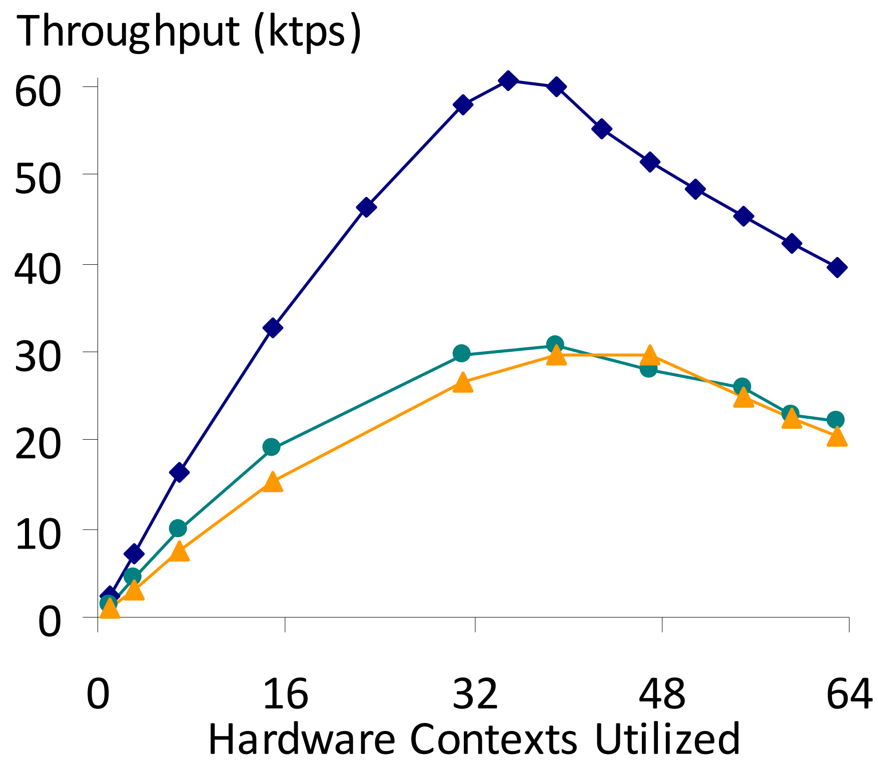
Improving OLTP scalability using speculative lock inheritance

Ryan Johnson, Ippokratis Pandis, Anastasia Ailamaki

OLTP scalability limitations

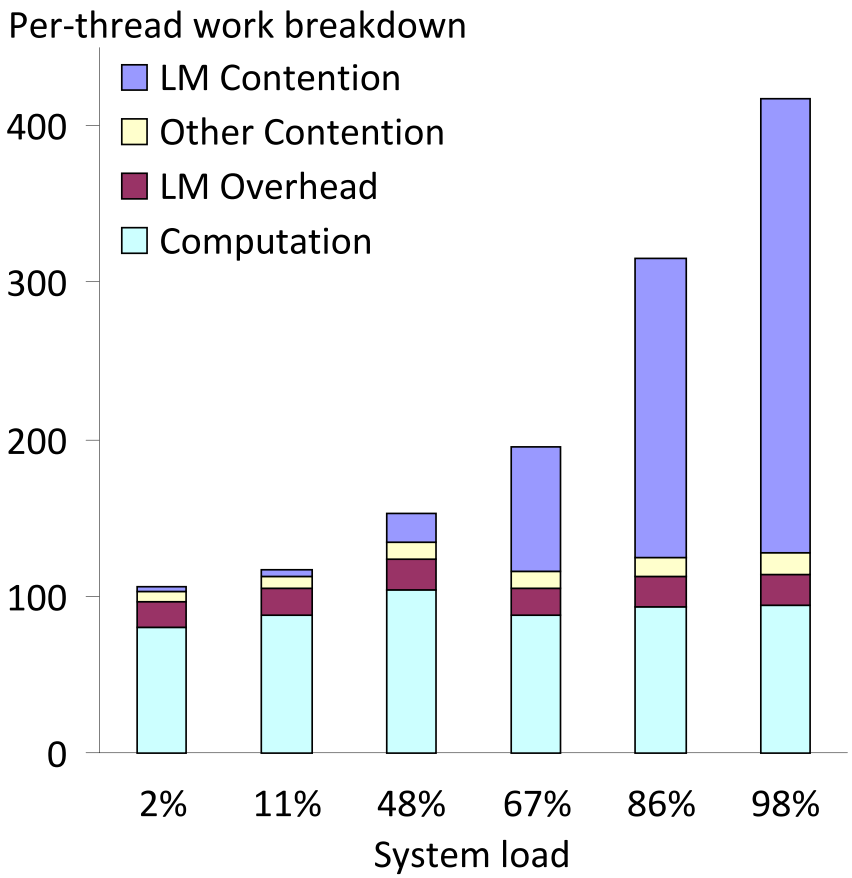
- Application: scalable
- Hardware: scalable
- Storage manager?
  - Good at interleaving threads
  - True concurrency is harder

  => Increased latch contention



Contention within storage manager limits scalability

Contention in the lock manager



- Centralized service
  - Locks managed globally
- Fine-grained parallelism
  - Each lock has its own latch
- Skewed access
  - Some hotter than others

Culprit: shared high-level locks

Contributions

- Identify lock manager bottleneck
  - Latch contention during lock state updates
- Eliminate most requests for hot locks
  - Pass hot locks directly between transactions
  - No changes at application level
- Improve throughput by 20-50%

In this talk…

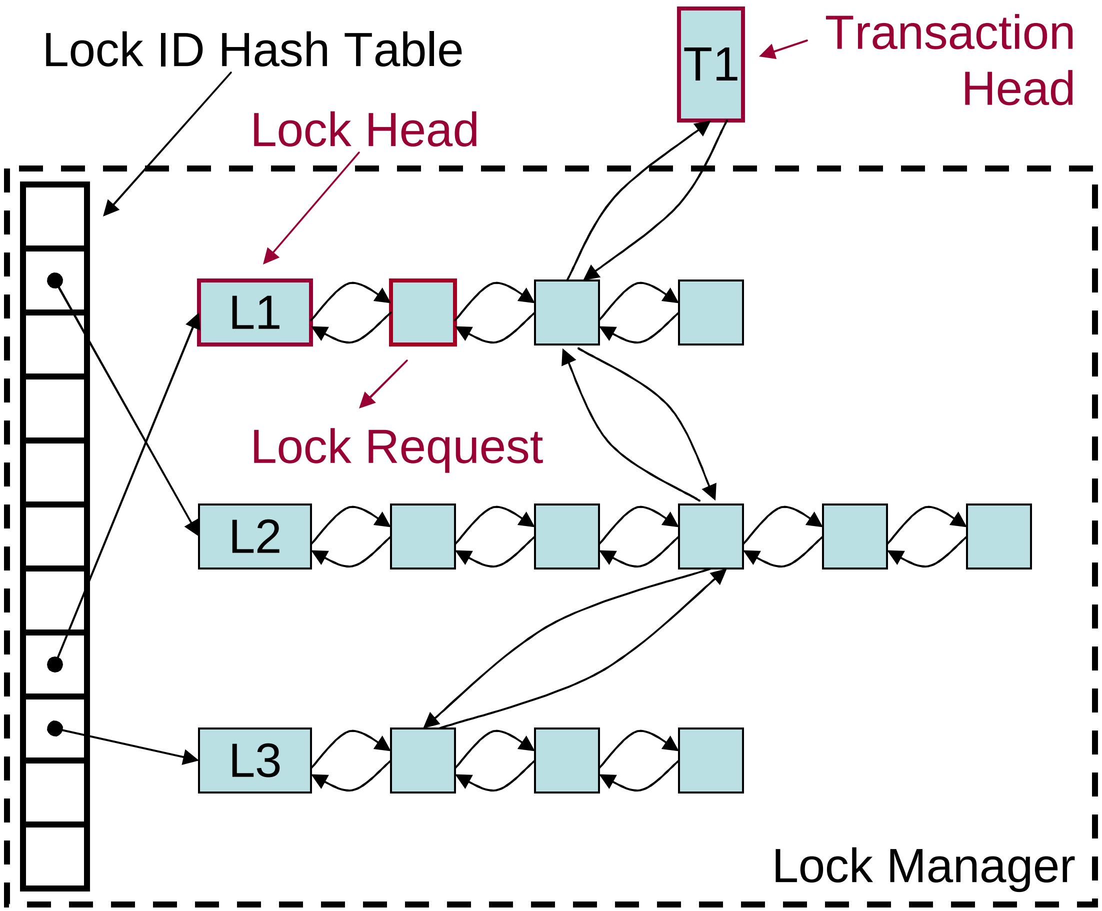
- Introduction
- Inside the lock manager
  - Locking vs. latching
  - Lock acquire and release
  - Approaches for reducing overheads
- Speculative lock inheritance
- Conclusions

Overview of latching vs. locking

	Latch	Lock
Serializes	Code segments	Transactions
Protects	Data structures	Database records
Duration	Short	Long
# Held at once	0-2	Unbounded
Implementation	Straightforward	Complex

Latches outnumber lock operations 5:1 or more

Inside the lock manager - acquire



Requirements

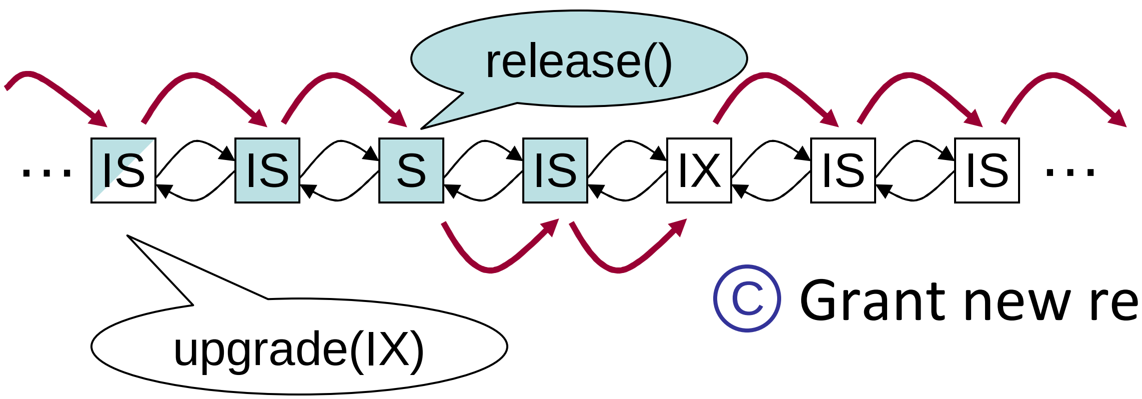
⇒ Find/create many locks in parallel

⇒ Each lock tracks many requests

⇒ Each transaction tracks many locks

Inside the lock manager - release

Ⓐ Compute new lock mode (supremum)



Ⓒ Grant new requests

Ⓑ Process upgrades

Lock strengths

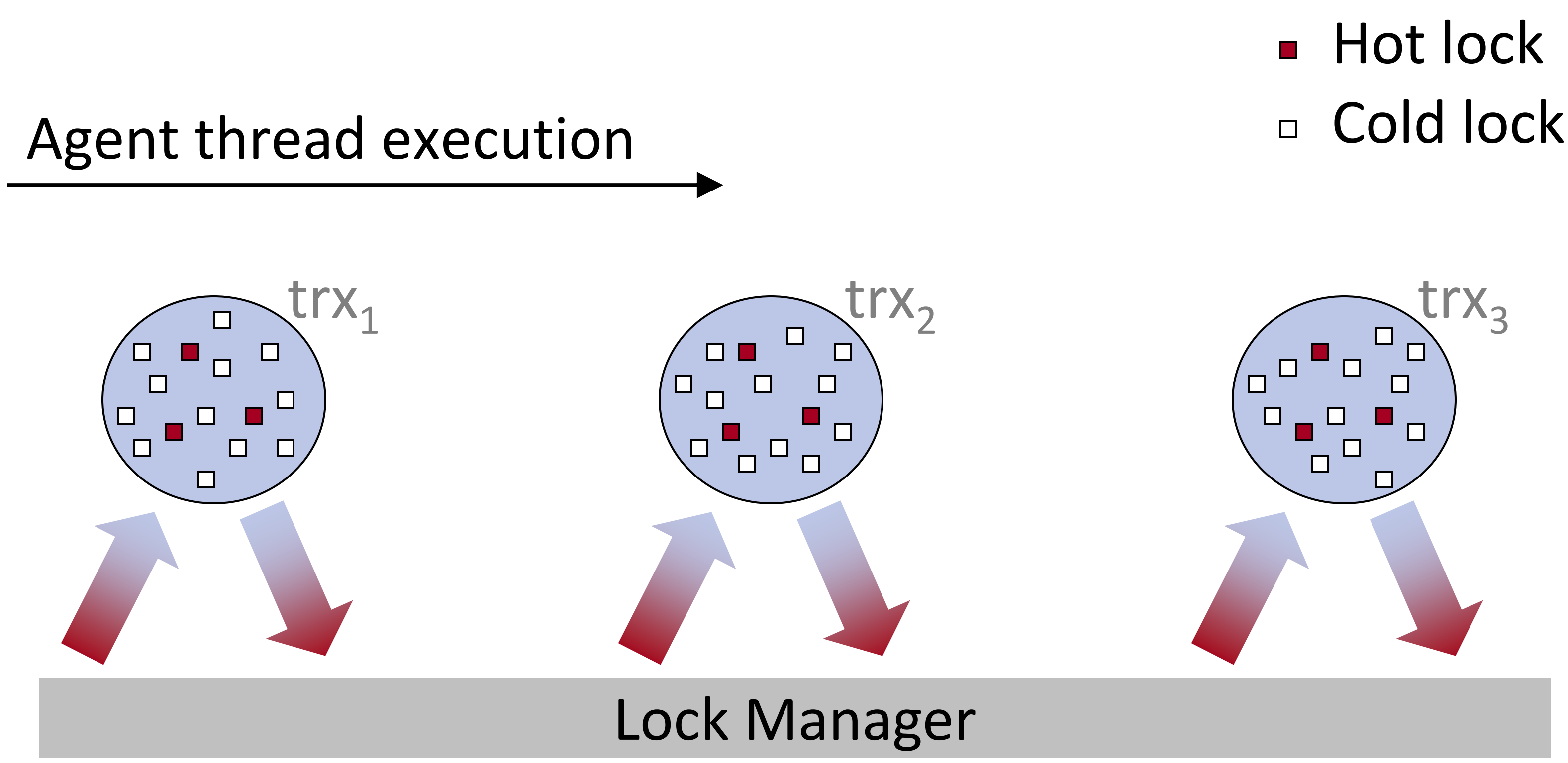
IS < IX < S

Intent locks => long request chains

In this talk…

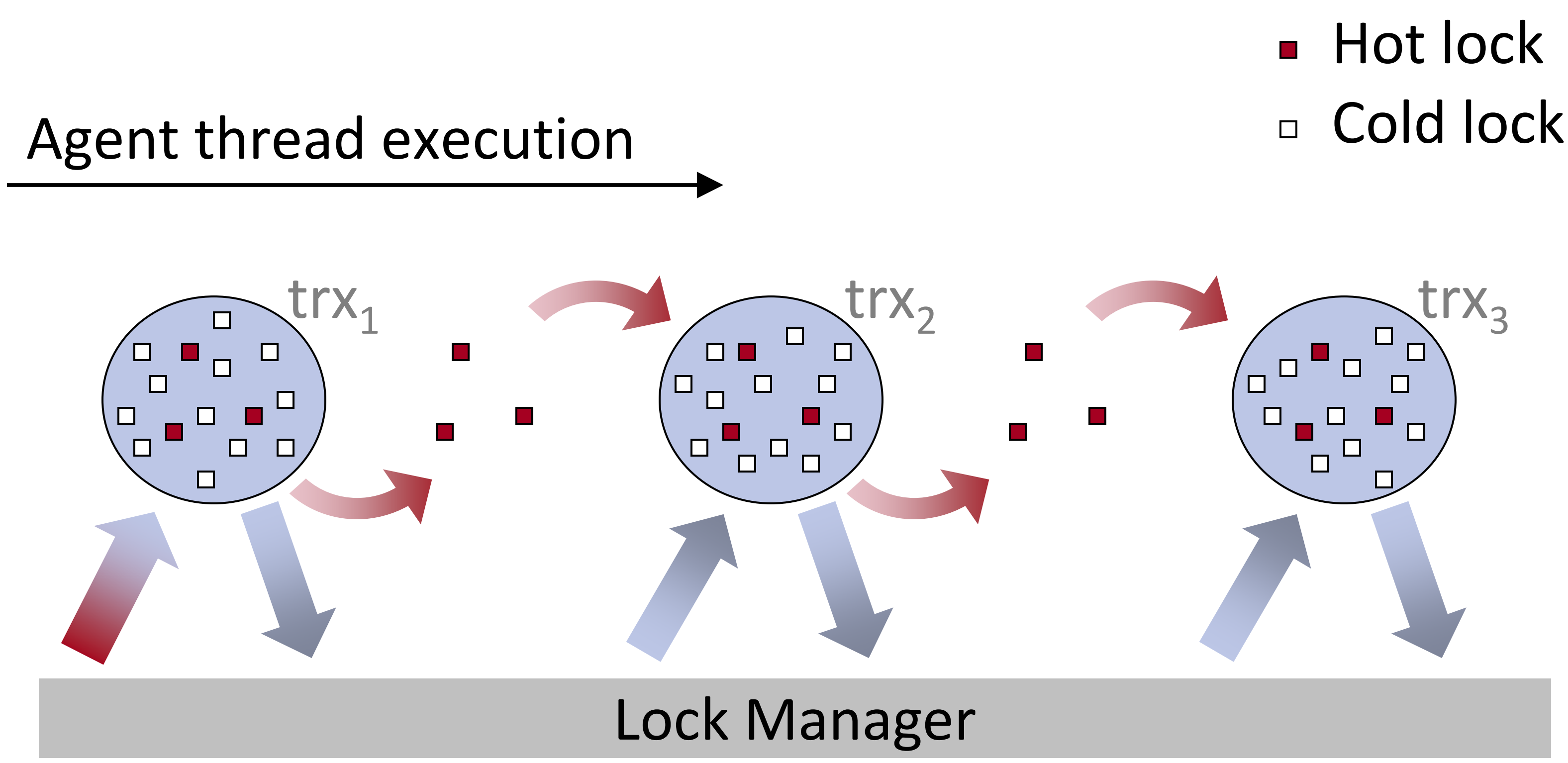
- Introduction
- Inside the lock manager
- Speculative lock inheritance
  - Approach and implementation
  - Experimental results
- Conclusions

Speculative lock inheritance



Hot locks bounce between agents and LM

Speculative lock inheritance

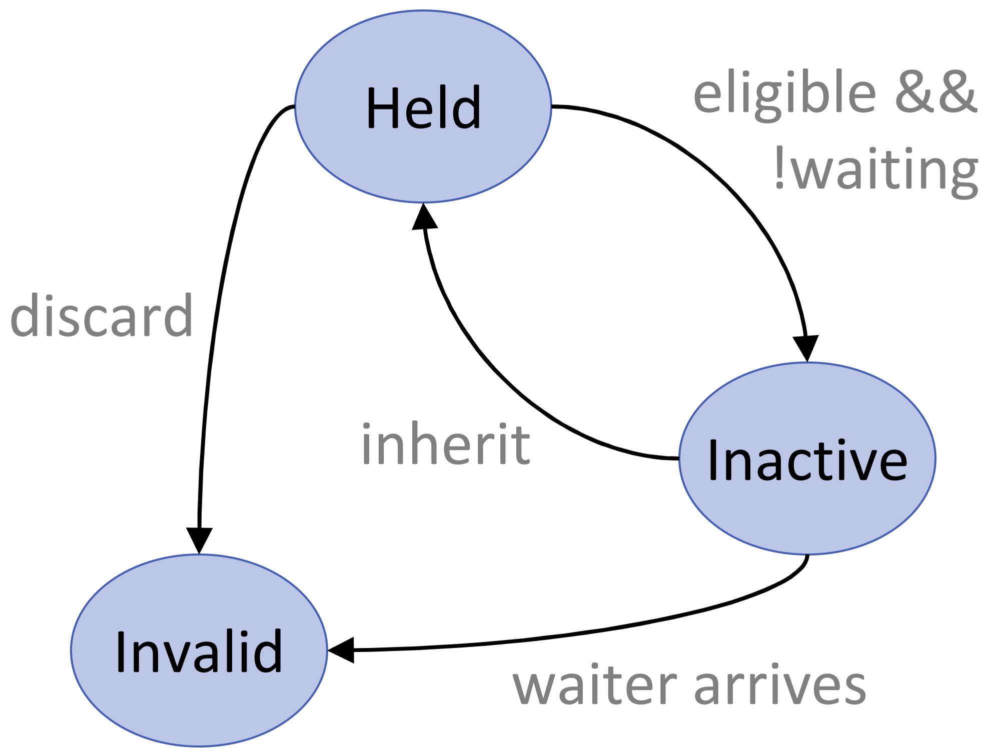


Agent thread keeps hot locks, bypasses lock manager

Criteria for lock inheritance

- Hot (latch contention)
- Held in shared mode
- No other trx waiting (fairness)
- Not row-level (too many of those)
- Parent (if any) also eligible

Avoiding starvation



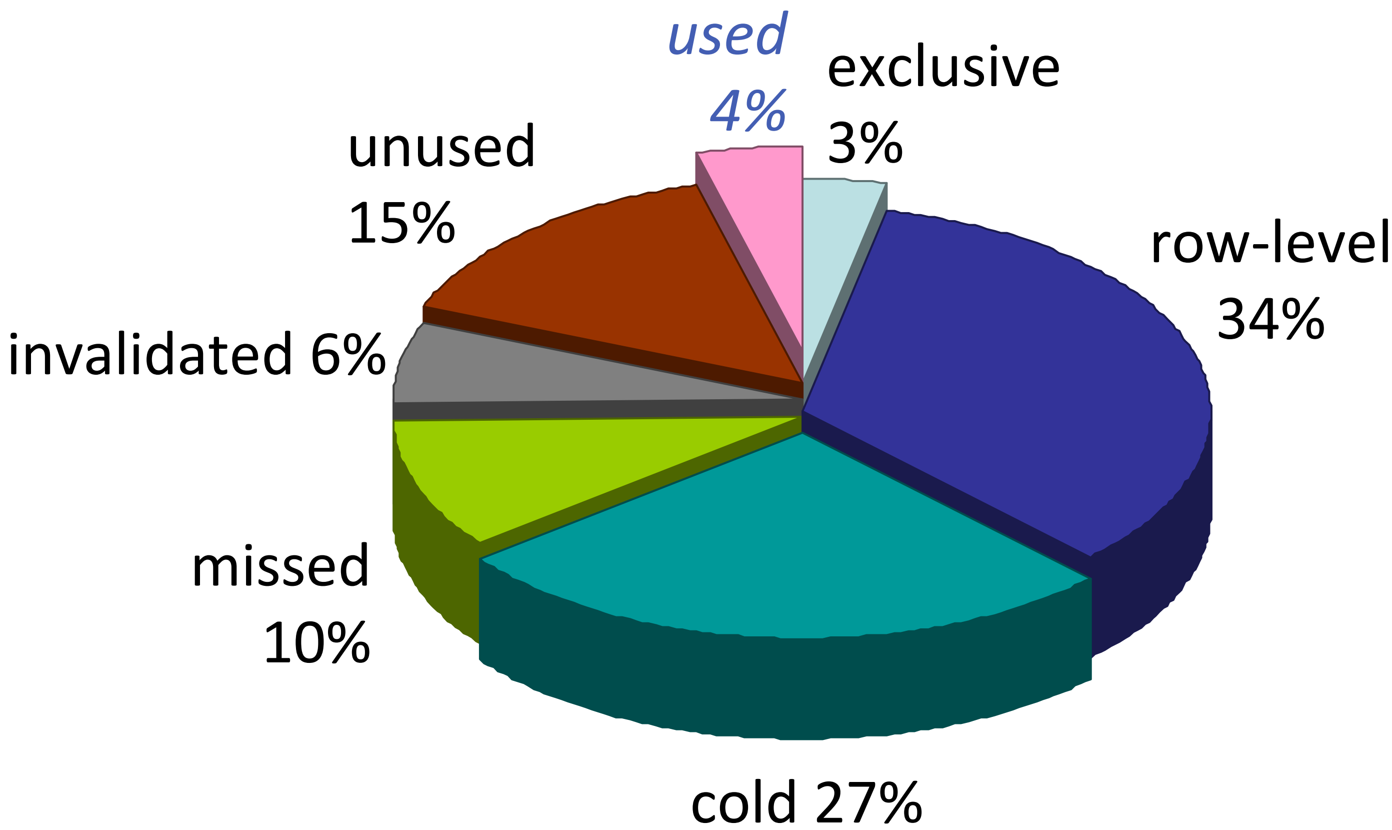
Update lock requests using atomic ops

⇒ sidestep the lock manager

⇒ clean up invalid requests lazily

Speculation is lightweight, non-intrusive

Breakdown of lock behavior

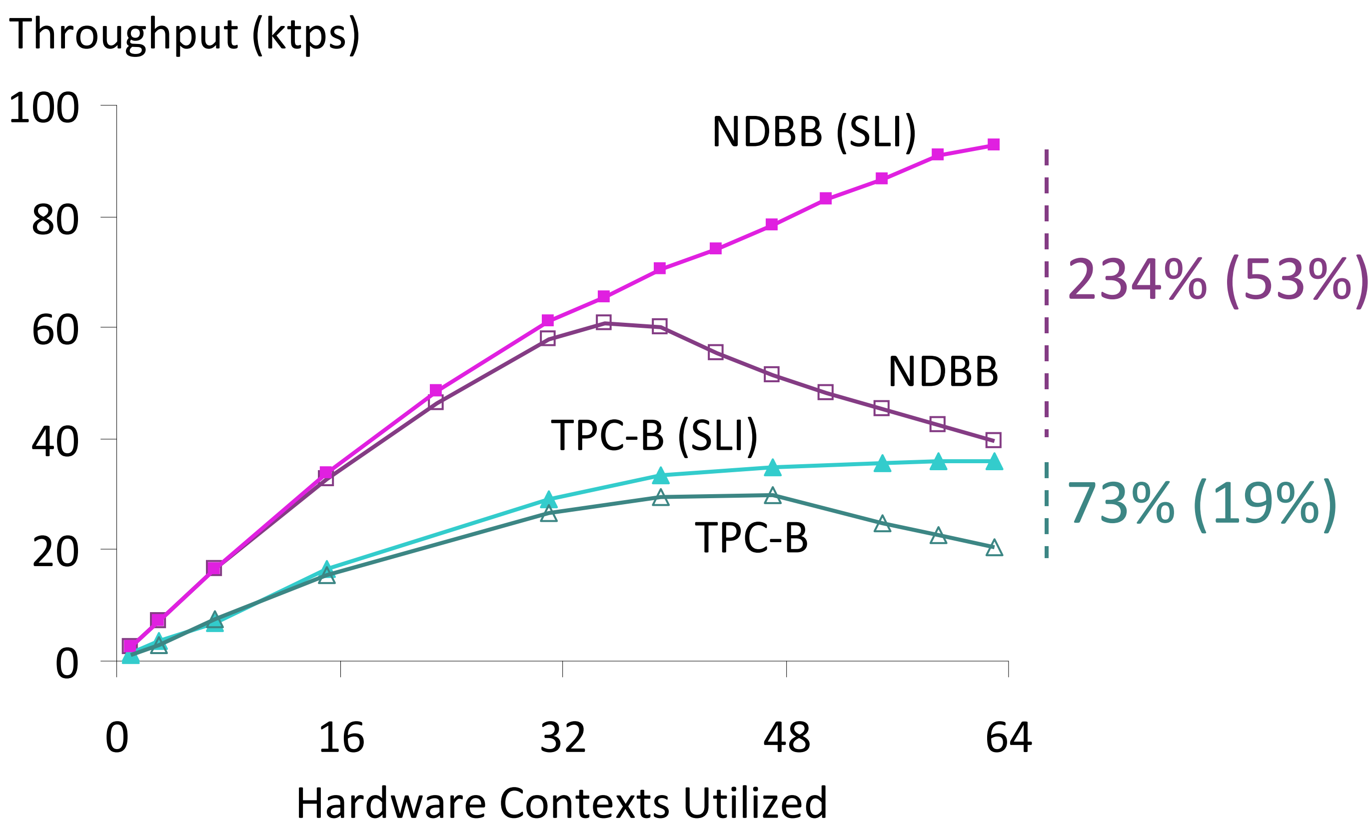


A few locks cause 97% of contention

Experimental setup

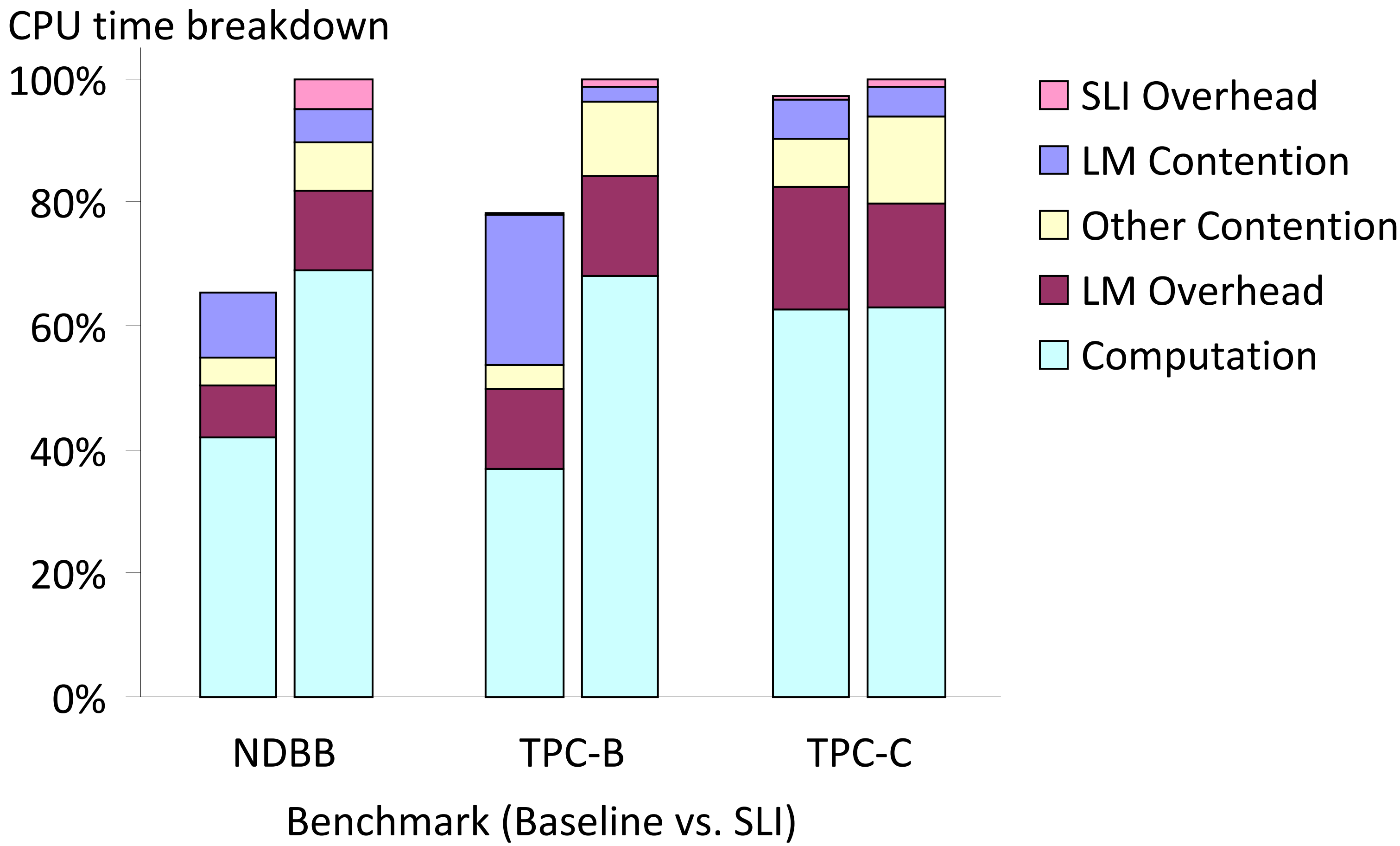
- Hardware
  - Sun T5220 “Niagara II” (sparcv9-solaris10)
  - 16 pipelines, 64 contexts, 64GB RAM
- Software
  - Shore-MT
  - Extended to include lock inheritance
- Benchmarks (mem-resident)
  - Telecom: Nokia Benchmark (NDBB)
  - Banking: TPC-B
  - Online Sales: TPC-C

Scalability comparison



Eliminate contention with low overhead

Impact of lock inheritance



Lock bottleneck gone (but others may arise)

Reducing overheads of locking

- Rdb/VMS
  - Distributed DBMS
  - Lock "carry-over" reduces network traffic
- DB2 "keep table lock" setting
  - Connection holds all table locks until close
  - Leads to "poor concurrency"
- H-Store
  - Single-threaded, non-interleaved execution model
  - No locking or latching at all

Conclusions

- All latches are potential bottlenecks
  - Even “fine-grained” ones if there’s skew
- Best solutions may be indirect
  - Sidestep hard problems
  - Look to distributed databases
- Lock inheritance eliminates one bottleneck
  - Distribute accesses to hot locks
  - Improve throughput 20-50% today
  - Benefit increases with more cores