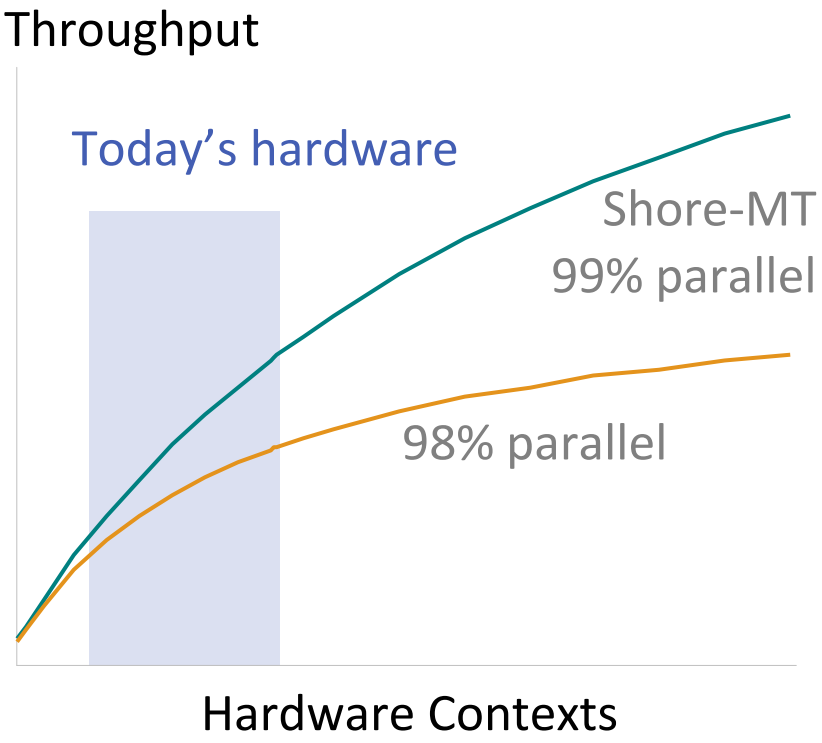
A new look at the roles of spinning and blocking

Ryan Johnson, Manos Athanassoulis, Radu Stoica, Anastasia Ailamaki

OLTP – a challenging workload

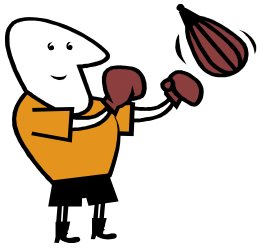
- Memory-resident
- High concurrency
  - 16-64 ctx today, more coming
  - Application is scalable
  - DBMS is “fairly scalable”
- Exposes OS overheads
  - Synchronization, scheduling
  - Any extra serialization hurts!



Not the first time OS gets in the way…

Latching: meet the "contenders"

- Spinning
  - Waste CPU for fast response
  - Vulnerable to OS scheduler
  - Favored for scientific workloads (high perf.)
  - Ex: time-published MCS[HiPC'05]
- Blocking**
  - Give CPU to other threads
  - Integrated with scheduler
  - Favored for commercial workloads (robust)
  - Ex: Solaris adaptive mutex

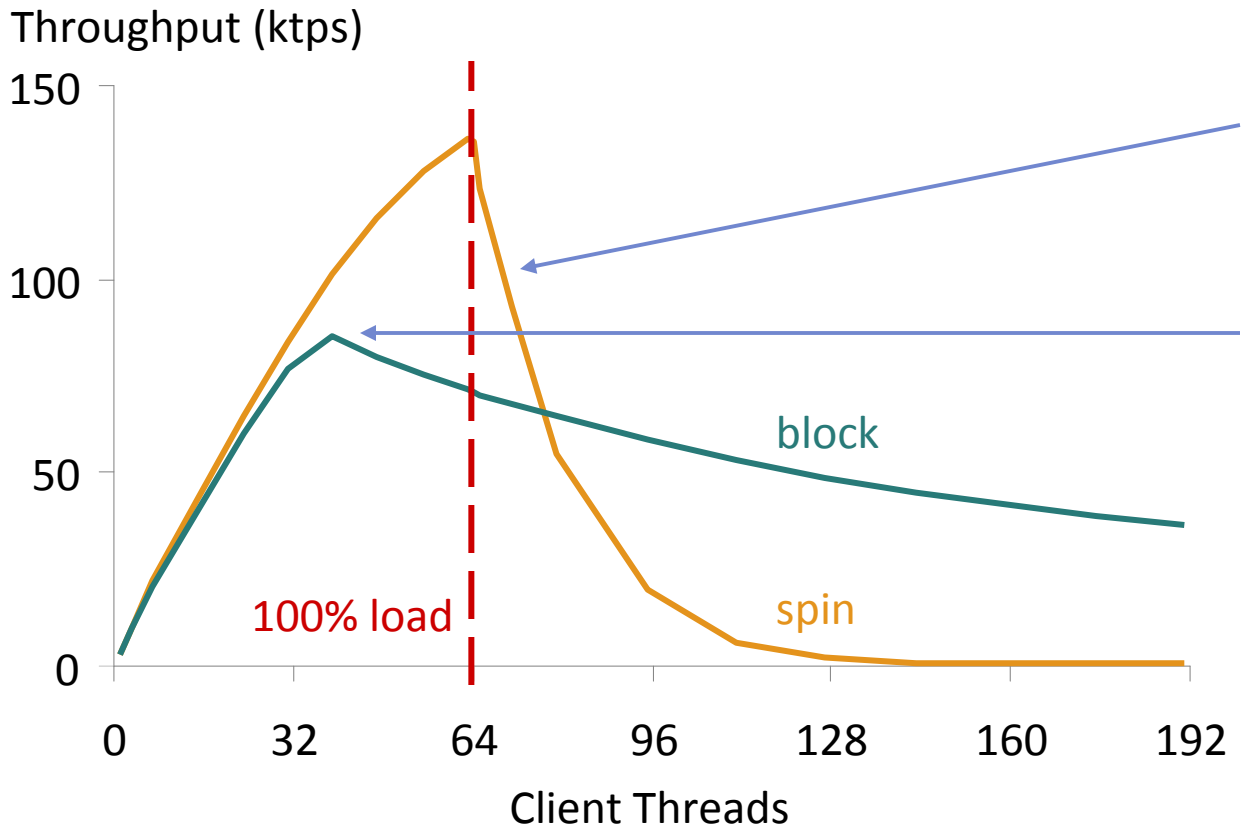


Philosophies are fundamentally opposed

=> Neither is best for all situations

** In practice needs some spinning

OLTP benchmark performance



Spinning falls off under load

Blocking does not scale well

Load + parallelism both high = 50% drop in throughput

Contributions

- Problem: OS-related scalability limitations
  - Undesirable scheduling decisions
  - Expensive synchronization primitives
- Cause: Trade-offs and conflicting goals
  - Spinning vs. blocking
  - Load vs. contention mgt.
- Solution: Decouple load and contention mgt.
  - Address orthogonal issues separately
  - Make spinning and blocking complement each other
  - Outperform existing solutions by 50%

In this talk…

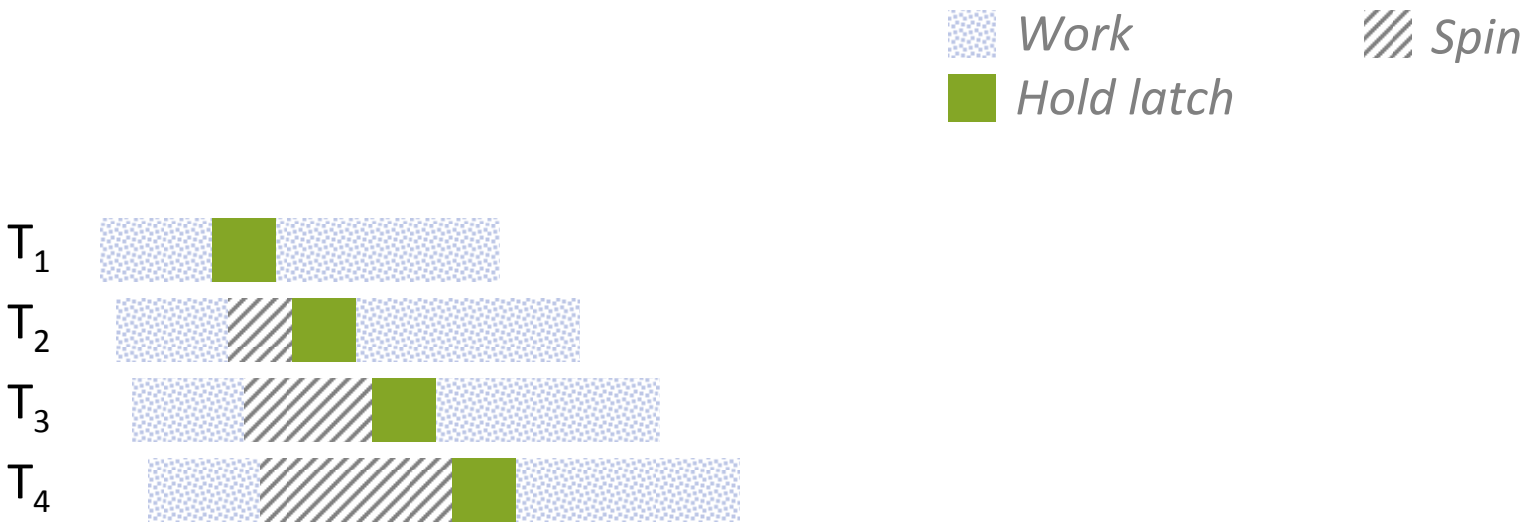
- OS-related scalability limitations
- Trading off spinning vs. blocking
- Decoupling load from contention
- Conclusions

Experimental Setup

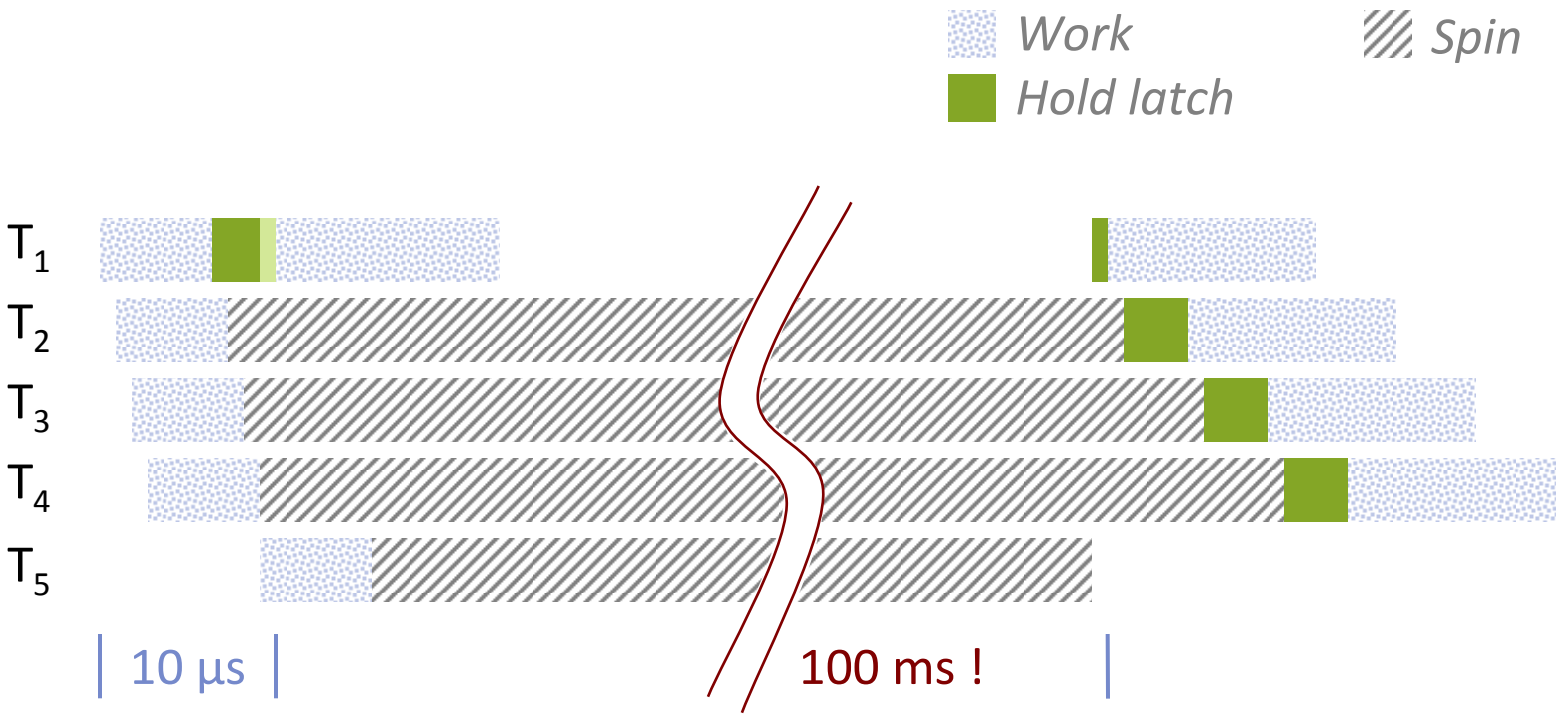
- Sun T5220 “Niagara II” Server
  - 16 cores** with 64 hardware contexts total
  - Solaris 10
- Shore-MT storage manager
  - Modified to use different latch types
  - Nokia Network Database Benchmark (aka “TM-1”)
- Measurements
  - Hand-instrumented code (e.g. gethrtime)
  - Sun profiling tools
  - DTrace

** technically 8 dual-pipelined cores

Spinning and thread preemption



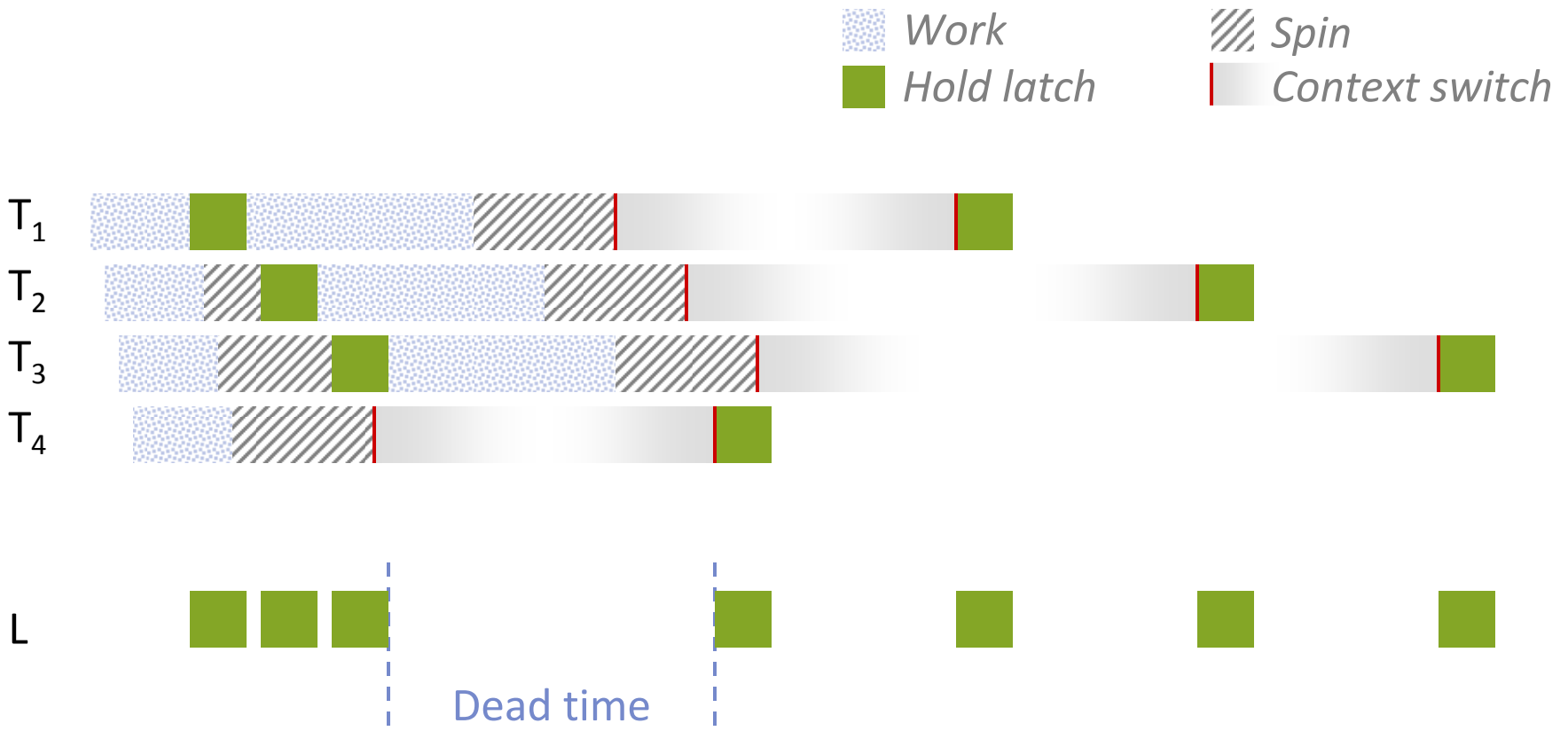
Spinning and thread preemption



Preempted latch holder = 10000x longer wait times

Next latch holder near end of time slice…

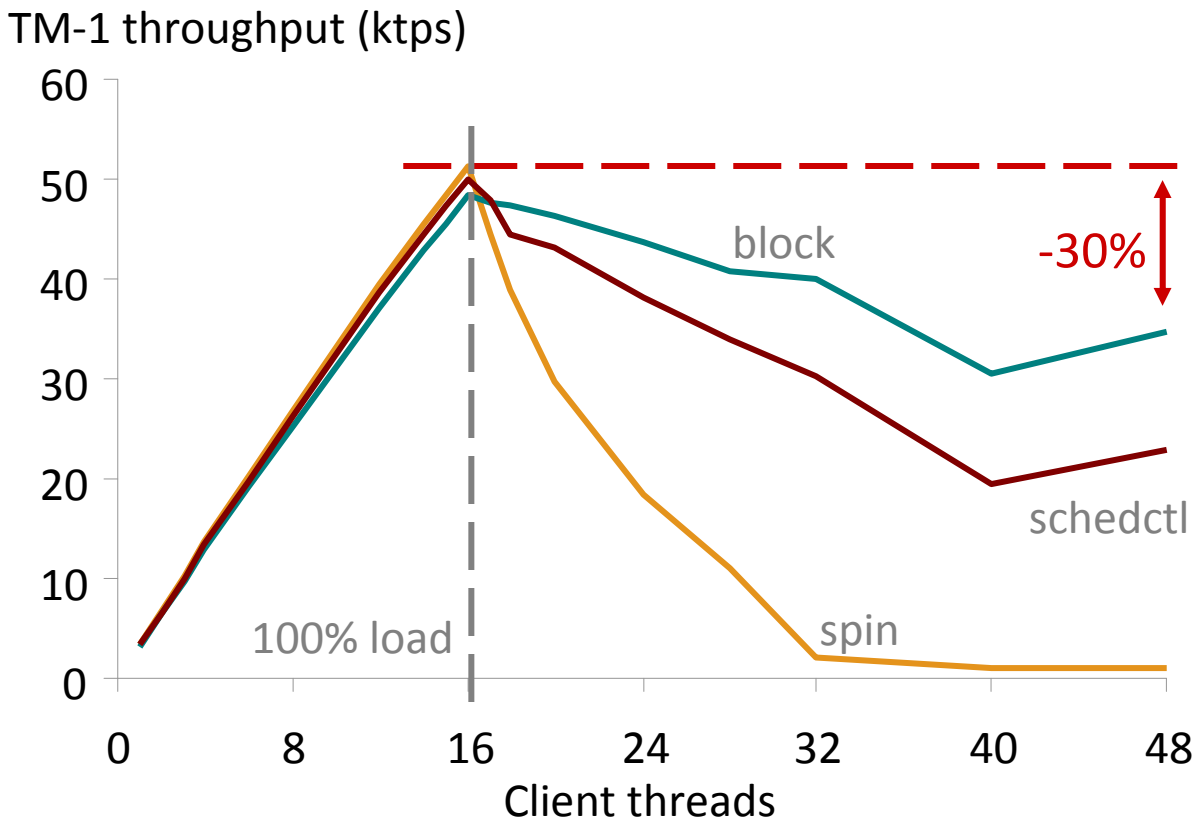
Blocking and latch dead time



Hand-off to sleeping thread = 10-20µs on critical path

Other waiting threads likely to give up spinning…

A small step back in time



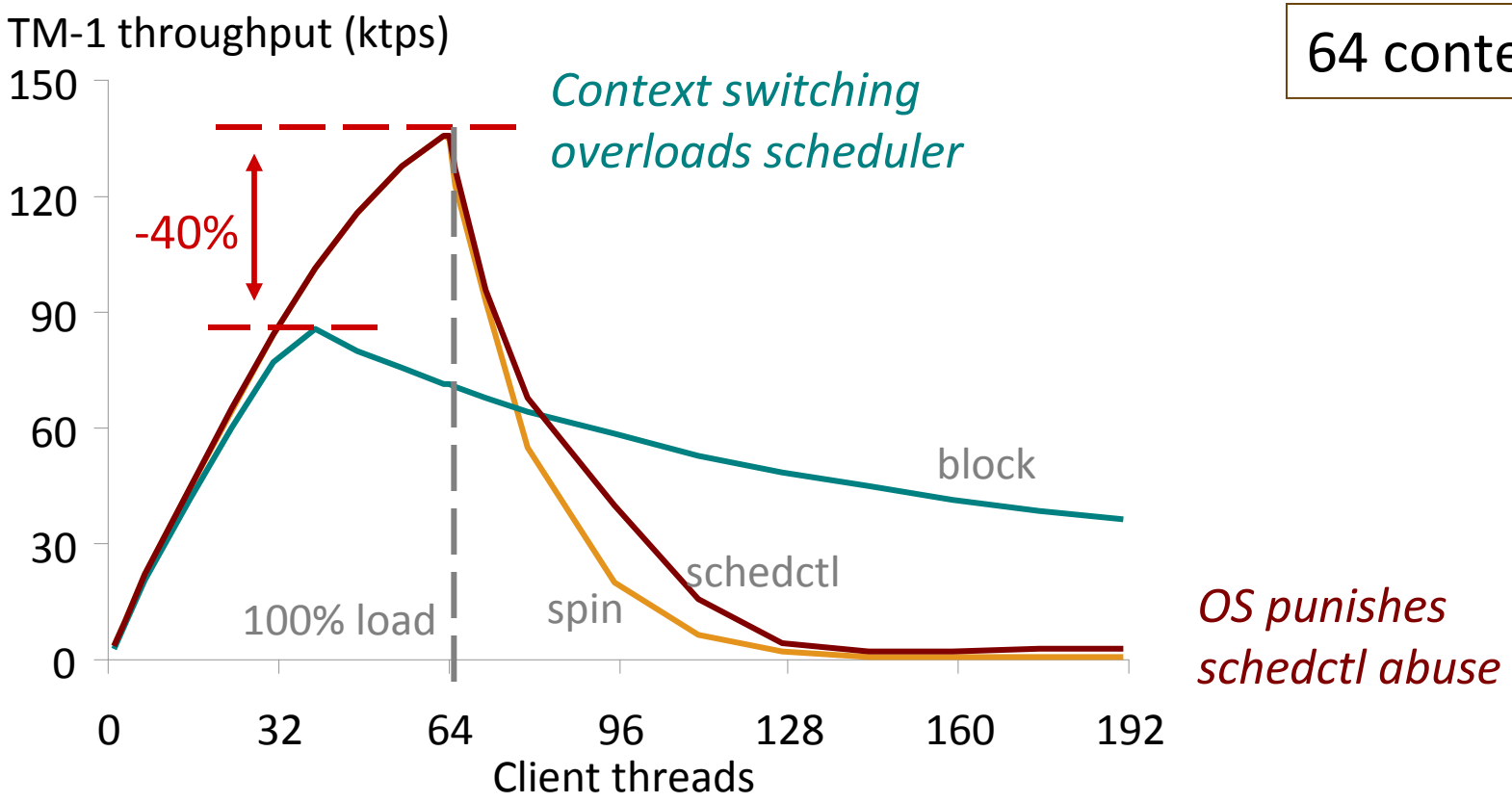
16 contexts

schedctl
Tell OS I hold latch
=> fewer unwanted
=> preemptions

Preempted latch holders

Database engines justified in using pthread_mutex so far

Scalability limits of blocking



64 contexts

OS punishes schedctl abuse

Techniques which used to work no longer useful

=> Cannot hide tension between spinning and blocking

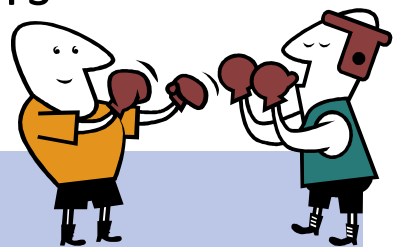
In this talk…

- OS-related scalability limitations
- Trading off spinning vs. blocking
- Decoupling load from contention
- Conclusions

Related Approaches

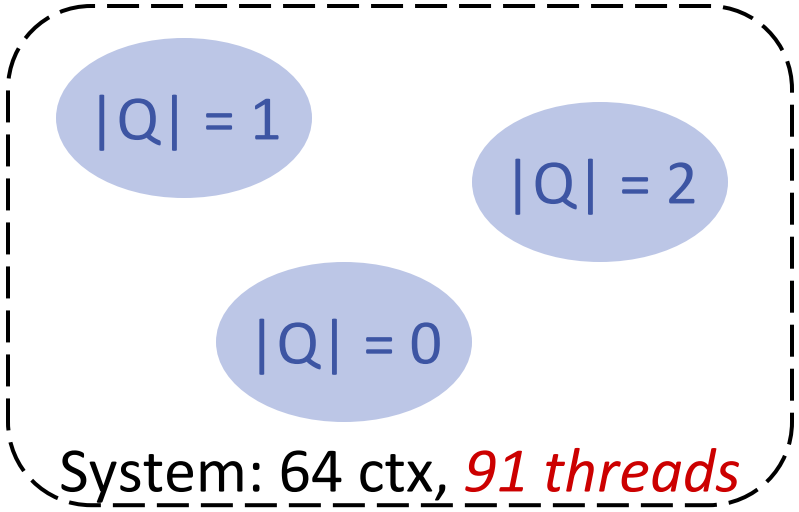
- Admission control
  - Request level is too coarse grained
  - *Knobs*: #contexts, request sizes, prob to block, …
  - Too many threads = load spikes
  - Too few threads = underutilization
- Adaptive/hybrid primitives
  - Implicit load control
  - *Knobs*: #threads, #contexts, latch hold time, cache, …
  - Too much spinning = preempted latch holders
  - Too much blocking = scheduling bottlenecks

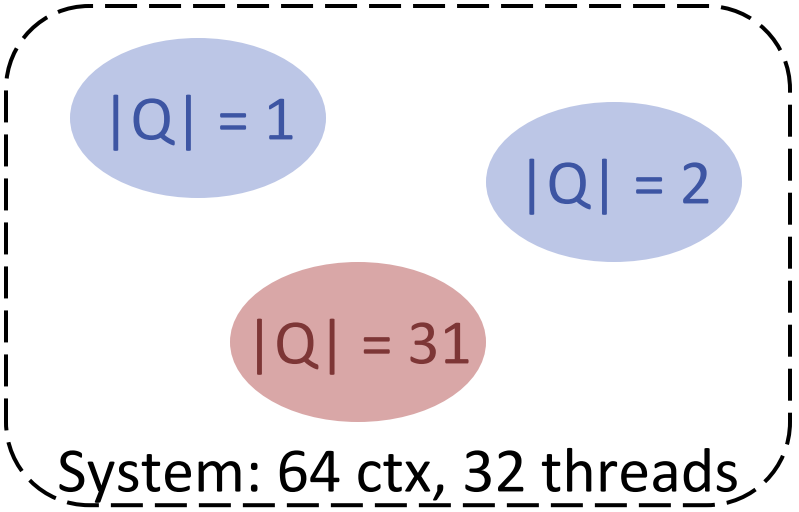
Fundamental tensions remain unresolved



Load and contention up close

- Load control
  - # active threads?
  - # HW contexts?
  - *Global* property
  - *Long* time scales (ms)
- Contention mgt.
  - Latch queue length?
  - Latch hold time?
  - *Local* property
  - *Short* time scales (µs)





Load and contention up close

- Load control
  - # active threads?
  - # HW contexts?
  - *Global* property
  - *Long* time scales (ms)
- Blocking
  - Central OS scheduler
  - Decisions every 10-100 ms

=> Ideal for load control!

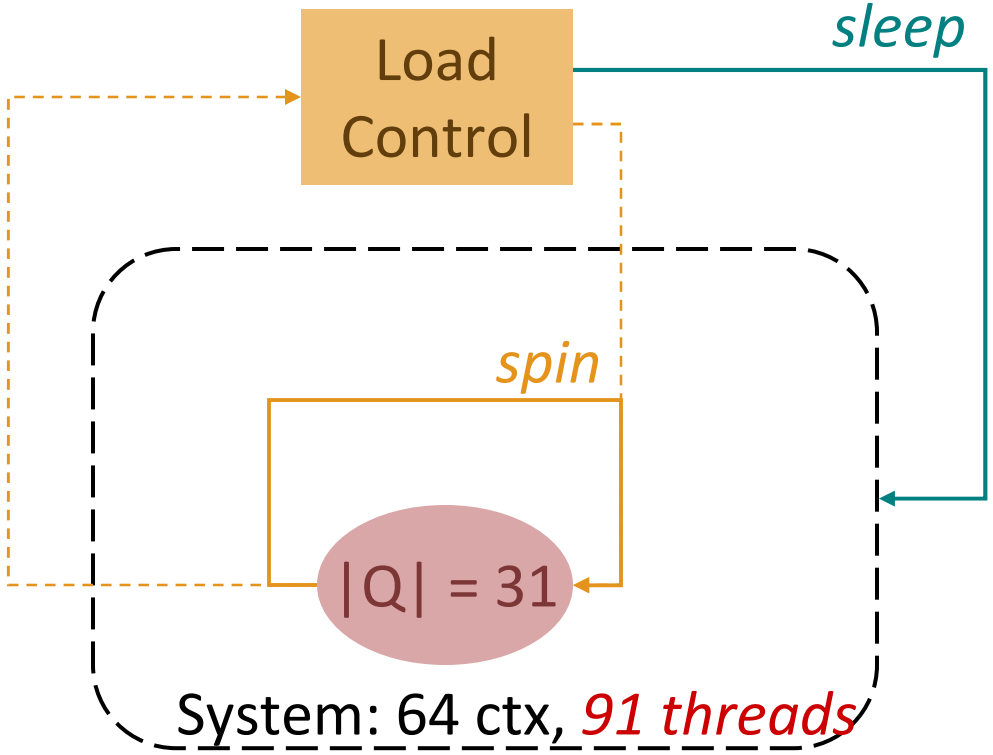
- Contention mgt.
  - Latch queue length?
  - Latch hold time?
  - *Local* property
  - *Short* time scales (µs)
- Spinning
  - Arbitrary memory location
  - Cache miss costs ns

=> Ideal for contention mgt!

Keep separation even when load, contention combine

Decoupling load from contention

Threads check load while spinning

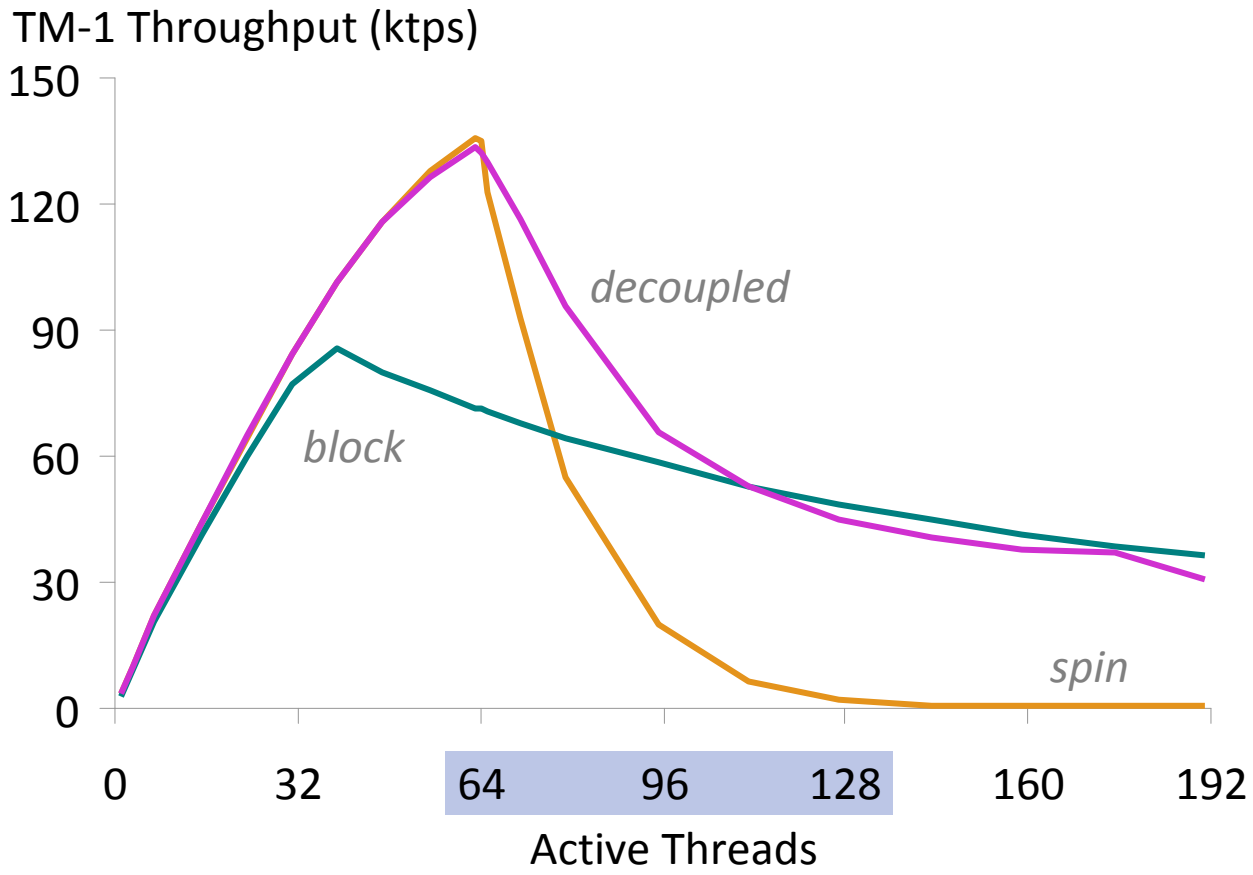


blocking
extra threads leave
=> no preemptions

spinning
fast latch hand-off
=> short critical path

Spinning and blocking cooperate instead of competing

Load control benefit for OLTP



Decoupled scheme tracks best across whole spectrum

Conclusions

- OS getting in the way of DBMS
  - … yet again …
  - Synchronization and scheduling this time
- Overheads come from tensions between
  - Spinning vs blocking
  - Load vs contention management
- Decoupling load from contention
  - Allows spinning and blocking to cooperate
  - Matches best behavior of other schemes
  - Gives up to 50% higher throughput under load

Thank you!

http://dias.epfl.ch