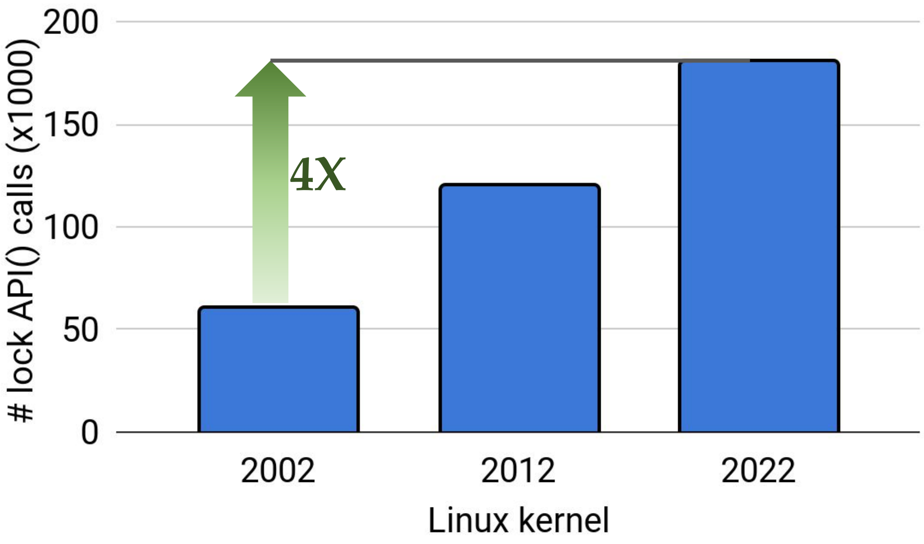
Ship your Critical Section Not Your Data: Enabling Transparent Delegation with TCLocks

Vishal Gupta Kumar Kartikeya Dwivedi Yugesh Kothari
Yueyang Pan Diyu Zhou Sanidhya Kashyap

EPFL

SRM Institute of Science and Technology
Learn · Leap · Lead

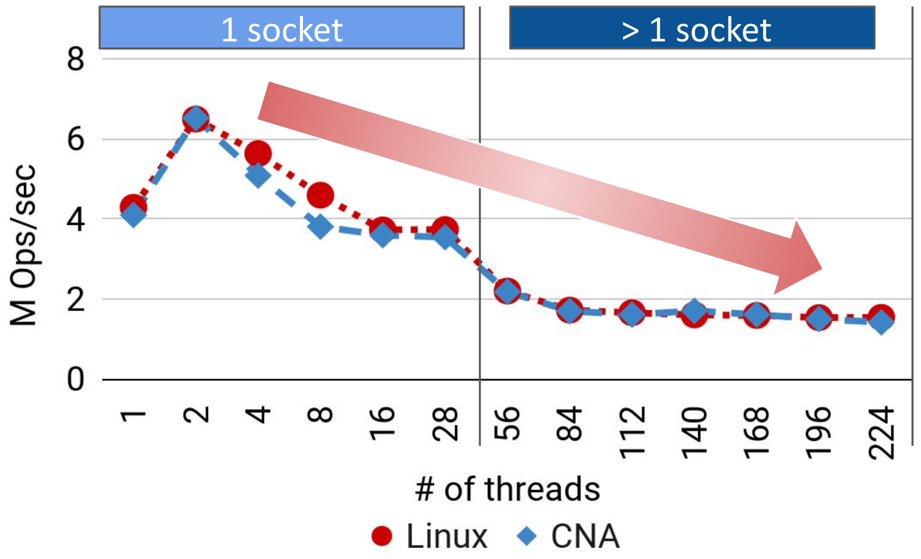
Locks: MOST WIDELY used mechanism



More locks are in use to improve OS scalability

Performance: Micro-benchmark

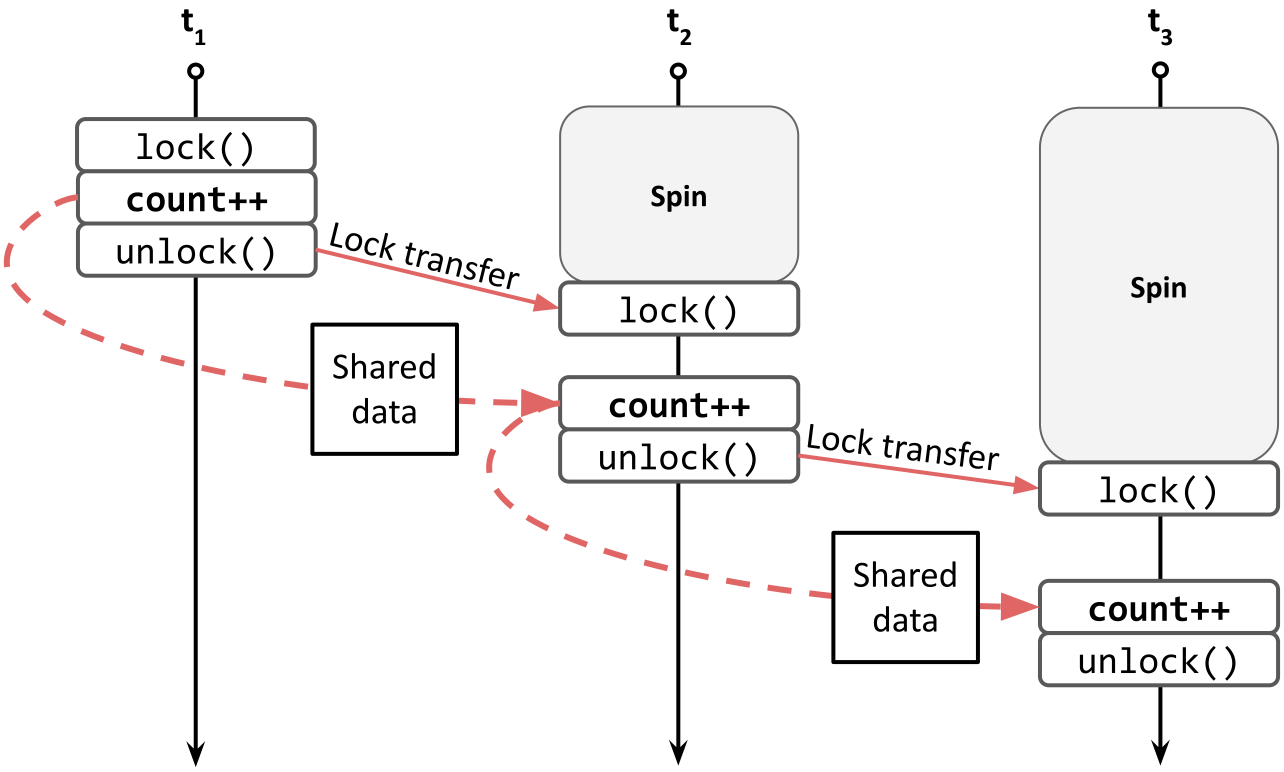
Benchmark: Each thread enumerates files in a directory, serialized by a directory lock



- Performance decreases with increasing core count
- NUMA-aware locks (CNA) follow a similar trend

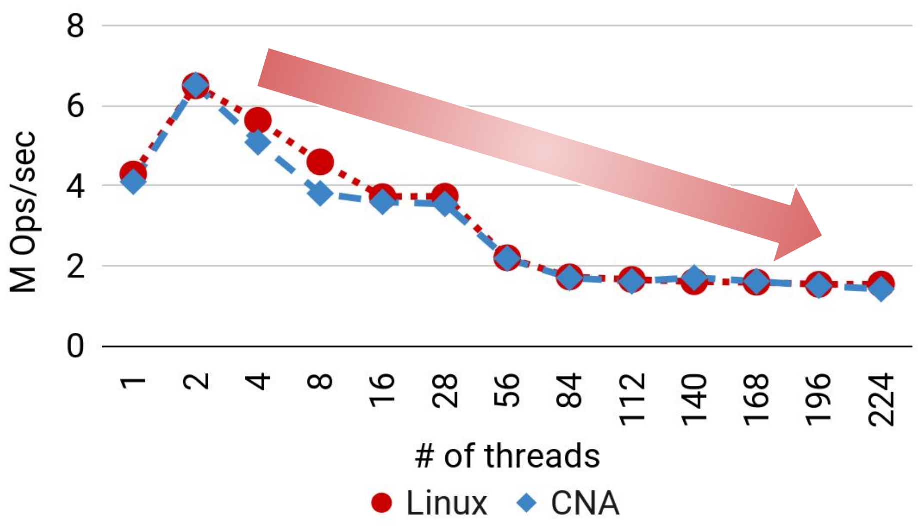
Setup: 8-socket/224-core machine

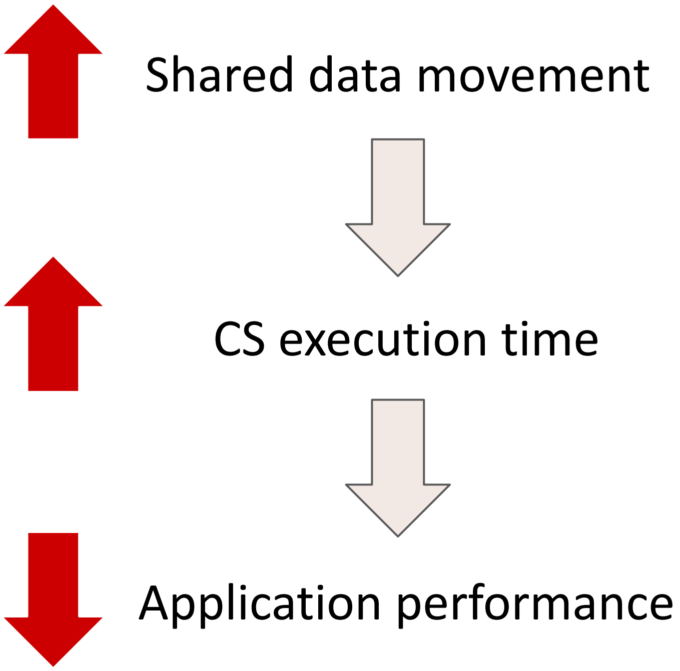
Traditional lock design: Large data movement



t_i: thread i

Traditional lock design: Not ideal





Delegation-style locks

- Similar to a server-client model
  - Server: Lock holder
  - Client: Waits to acquire the lock

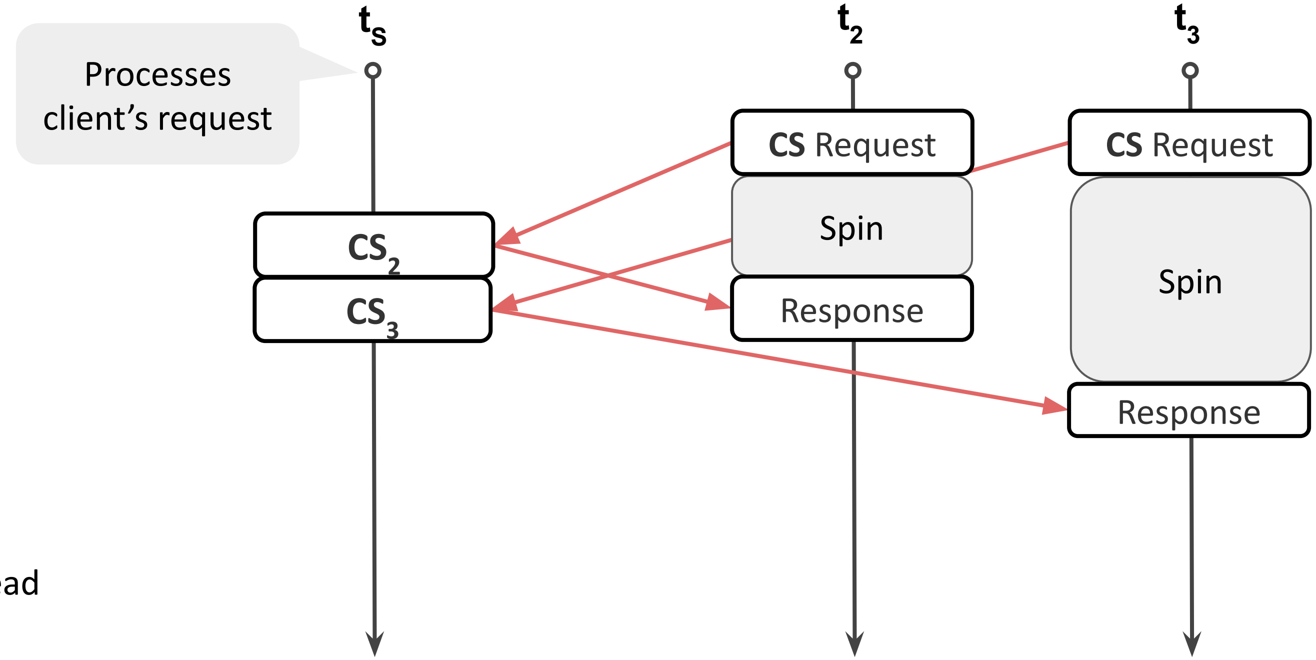
```
lock()
count++
unlock()
```

- Client ships its critical section request in the form of a function to the server thread

```
void incr_func() =
     count++

send_req_to_server(&incr_func)
```

Delegation-style locks



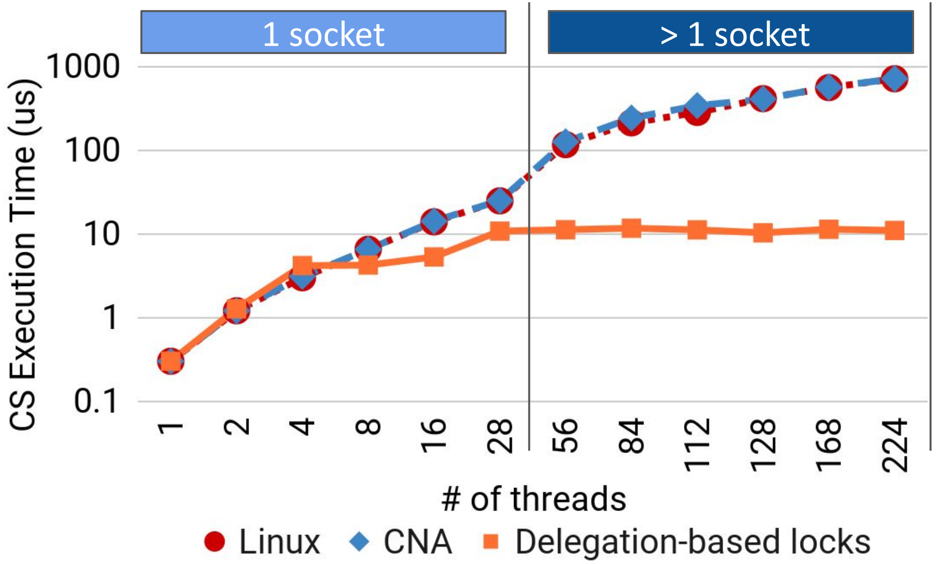
t_s: server thread

t_i: thread i

CS: critical section

Delegation-style locks

Benchmark: Each thread enumerates files in a directory, serialized by a directory lock



CS execution time similar with increasing core count

- Minimal shared data movement

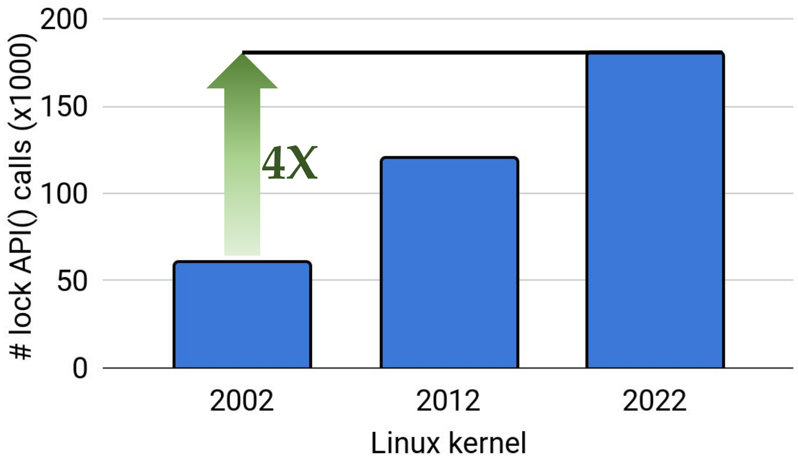
Setup: 8-socket/224-core machine

Delegation locks require app. modification

```
lock()
count++
unlock()
```

```
void incr_func() =
     count++

send_req_to_server(&incr_func)
```



Delegation is impractical for complex applications

TCLocks: Goals

- Transparency
  - Use standard lock/unlock APIs without rewriting applications

- Delegation
  - Minimal shared data movement

Transparent delegation

How to achieve transparent delegation?

- **How to capture the thread's context?**
  - Without application rewrite

- **Where to capture the thread's context?**
  - Such that only critical section is captured

- **Does the waiter's thread modify its context?**
  - While the server is executing waiter's critical section

Key idea: Transparent delegation

- **How to capture the thread's context?**
  - Instruction pointer + stack pointer + general-purpose registers

- **Where to capture the thread's context?**
  - Start and end of lock/unlock API

- **Does the waiter's thread modify its context?**
  - No, lock waiter busy waits to acquire the lock

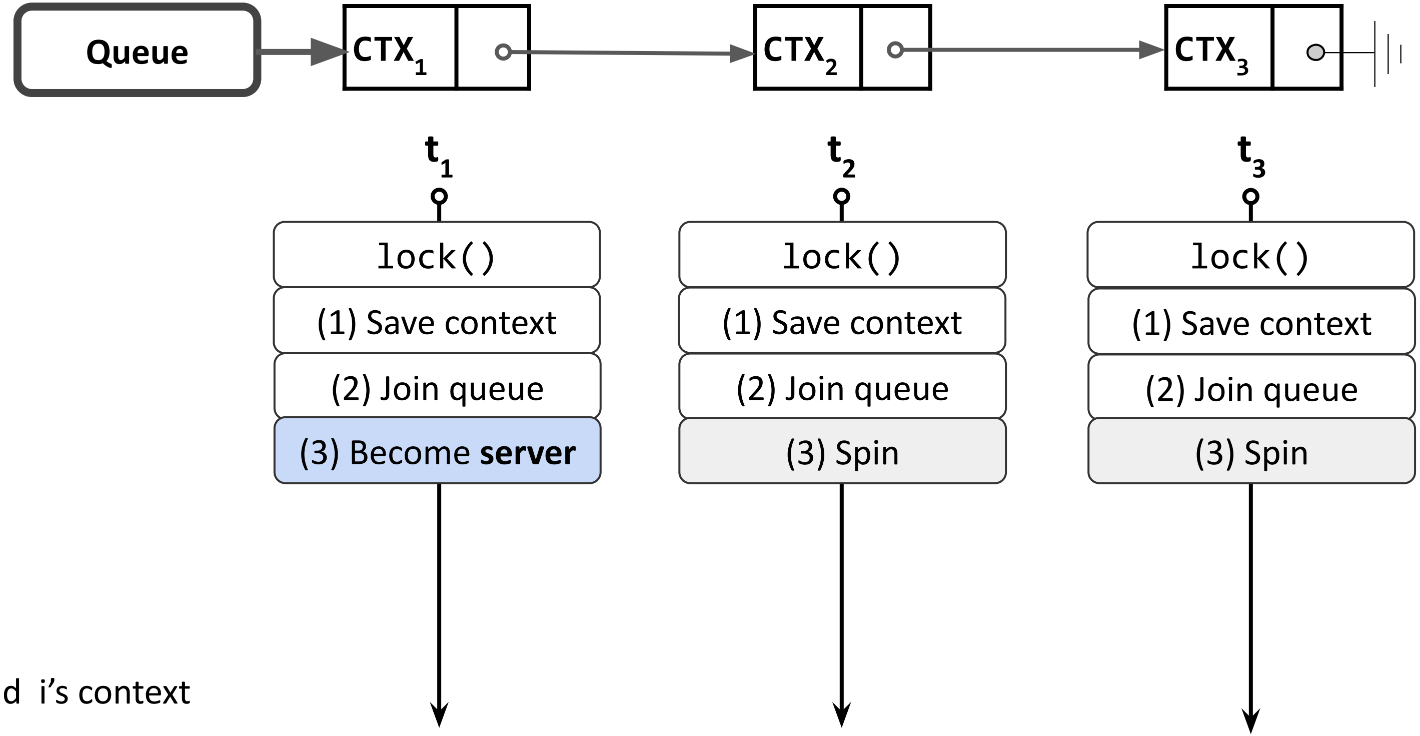
TCLocks: Putting it all together

- Queue-based lock
  - List of waiters maintained as a queue
  - Supports different queue reordering policy[1]
- Same `lock`/`unlock` API

- Server thread batches each waiters' request
- No dedicated server
  - Head of the queue becomes the server
  - The role is transferred to the next waiter after some threshold

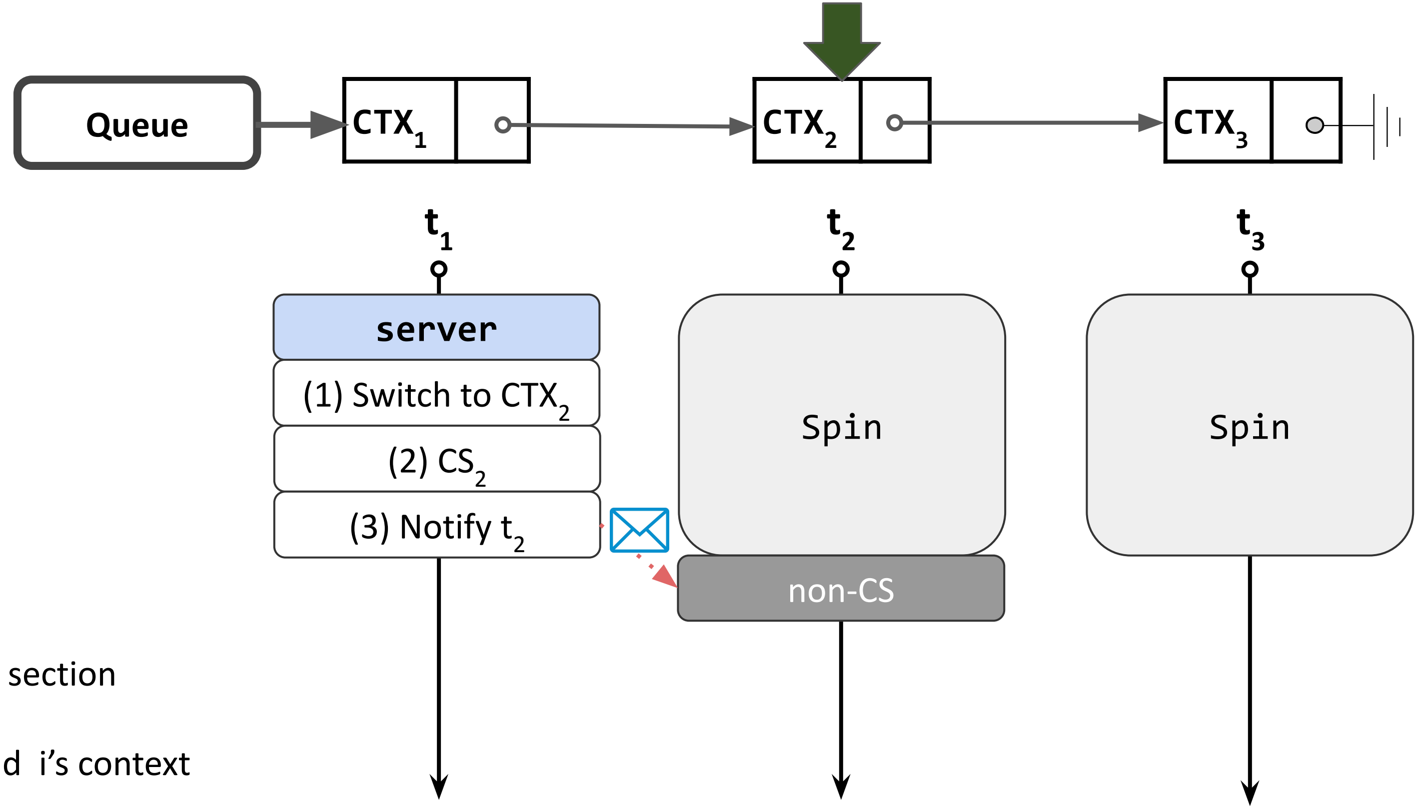
1. Scalable and Practical Locking With Shuffling, SOSP'19

TCLocks in action: Phase 1



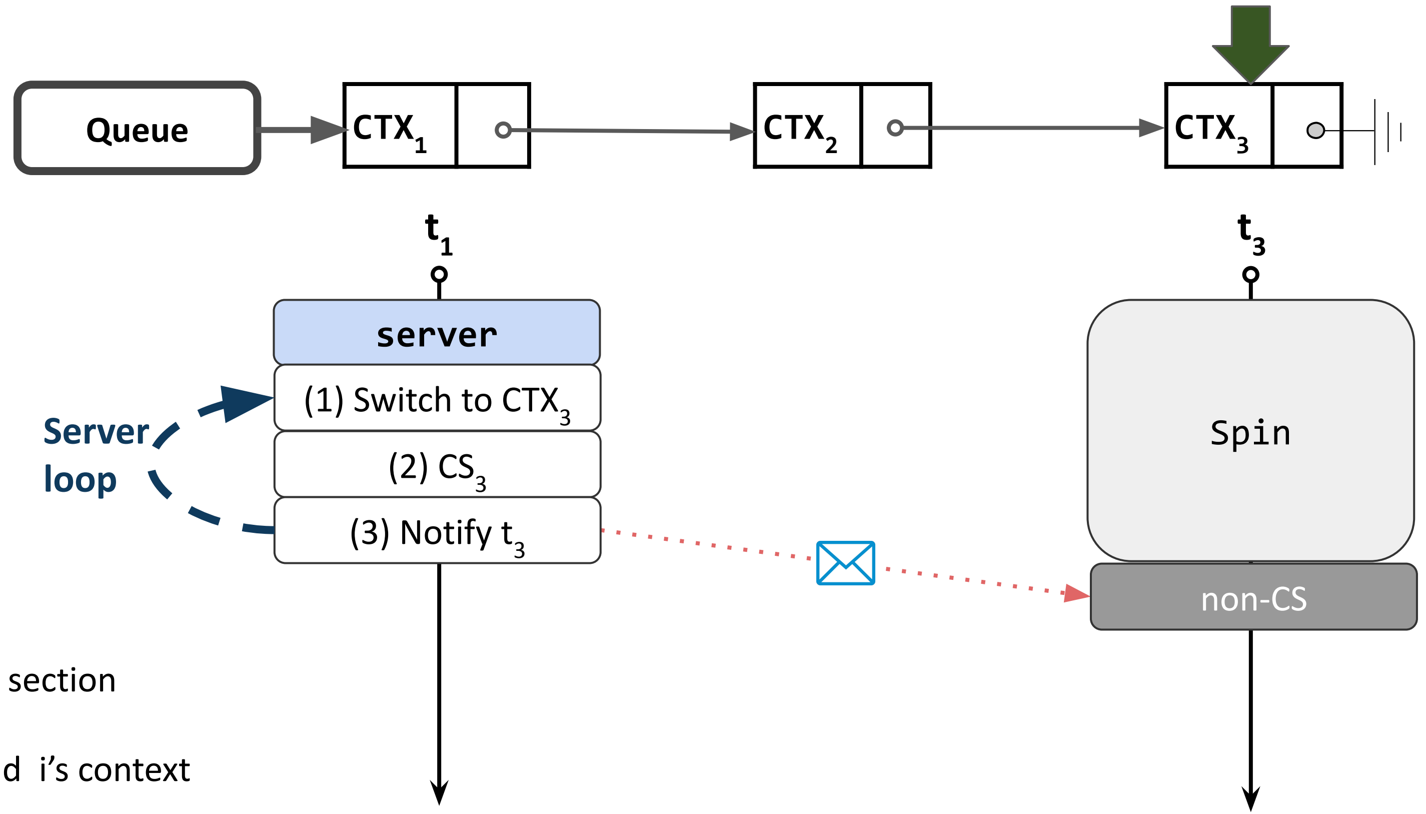
t_i: thread i
CTX_i: thread i's context

TCLocks in action: Phase 2



CS: Critical section
t_i: thread i
CTX_i: thread i's context

TCLocks in action: Phase 2



CS: Critical section

t_i: thread i

CTX_i: thread i's context

TCLocks: Practical considerations

- Ideal case
  - Waiter’s thread does not modify its context
- Reality
  - External events can modify waiter’s context
    - Interrupts: Require stack access
    - Waiter’s parking/wakeup mechanism: Require stack access

- **Ephemeral stack**
  - An empty piece of memory used only during critical section execution
  - Handles:
    - Interrupts on waiter’s CPU
    - Waiter’s thread parking/wakeup mechanism

TCLocks: Making it practical

- Algorithmic support:
  - Blocking and reader-writer locks
  - NUMA-aware policy
- Lock usage:
  - Nested locking and OOO unlocking
  - Special execution contexts and per-CPU variables
- Performance optimization:
  - Reduced context-switch overhead
  - Stack prefetching

Checkout the paper for more details

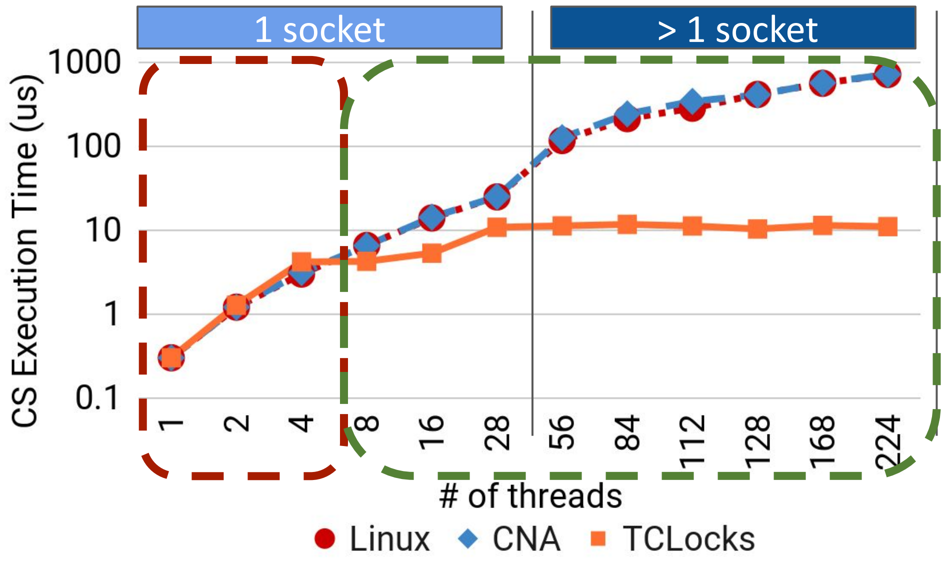
TCLocks: Evaluation

- Does TCLocks reduce the time spent in critical section?
- Does TCLocks improve application performance?

Hardware: 8-socket/224-core Intel machine

Evaluation: CS execution time

Benchmark: Each thread enumerates files in a directory, serialized by a directory lock

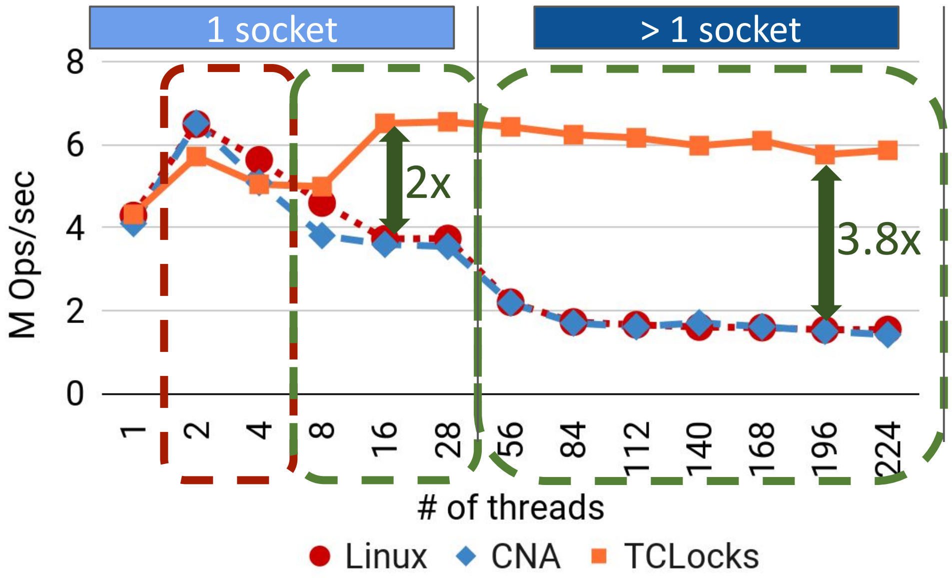


- **> 4 threads**
  - Minimal shared data movement
- **≤ 4 threads**
  - Context-switch overhead
  - Not enough batching

Setup: 8-socket/224-core machine

Evaluation: Micro-benchmark

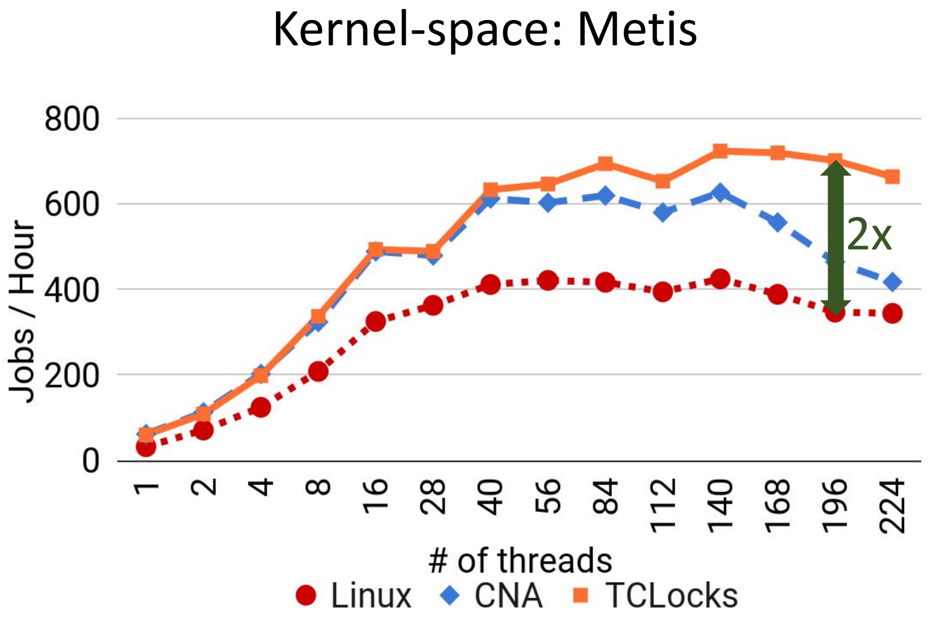
Benchmark: Each thread enumerates files in a directory, serialized by a directory lock

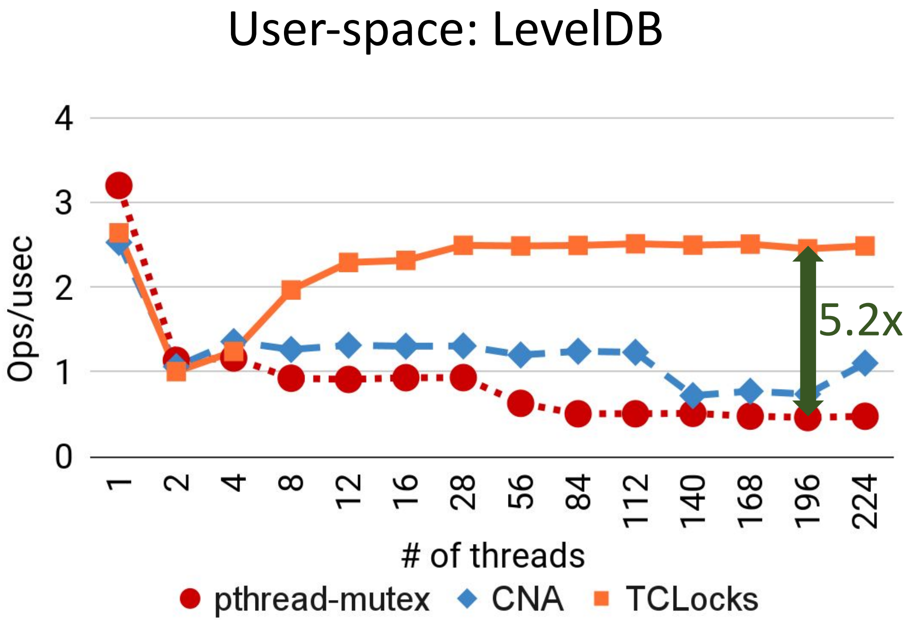


- **Within a socket:**
  - Minimal shared data movement
- **Across socket:**
  - NUMA-aware policy
- **2 - 4 cores:**
  - Context-switch overhead
  - Not enough batching

Setup: 8-socket/224-core machine

Evaluation: Real-world applications





TCLocks provides similar or better performance irrespective of thread count

Conclusion

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

- Existing lock design:
  - Traditional lock design has more shared data movement
  - Delegation-based lock design requires application modification
- **TCLocks**: Provides transparent delegation
  - Capture thread's context at right time
- Key takeaway:
  - Applications can now use delegation-style locks without modification

https://rs3lab.github.io/TCLocks/

Thank you!