Scalable and Practical Locking with Shuffling

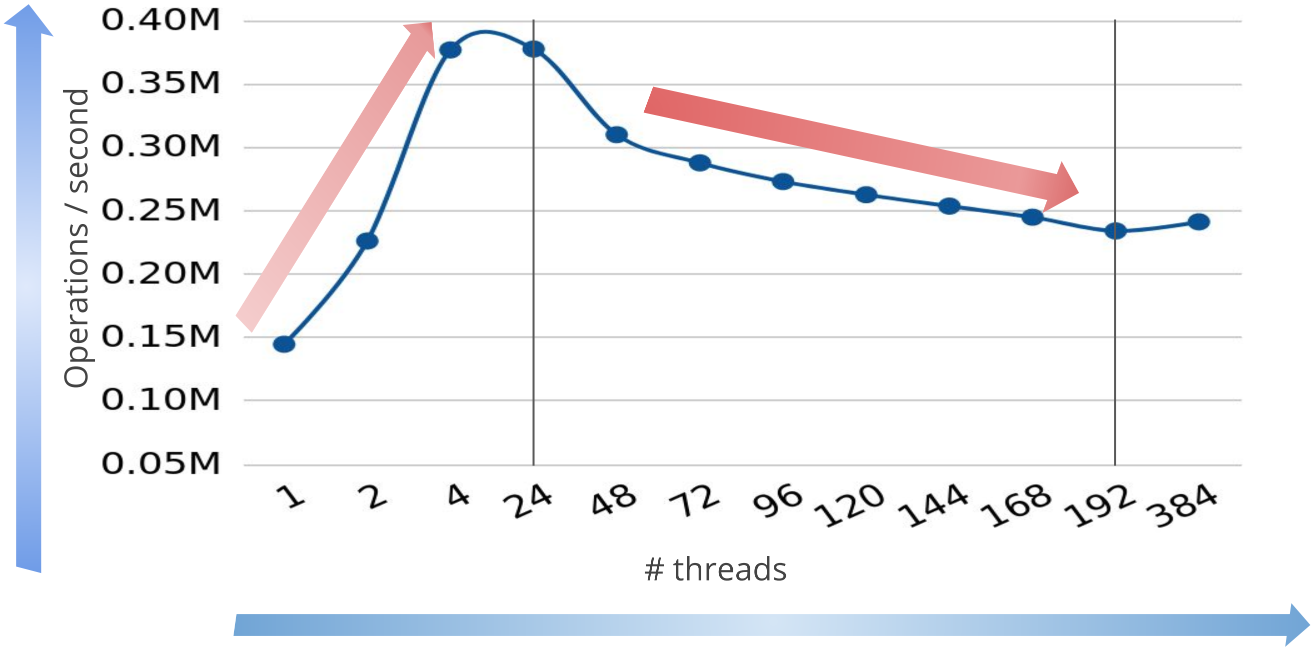
Sanidhya Kashyap*   Irina Calciu   Xiaohe Cheng

Changwoo Min   Taesoo Kim

*On the job market

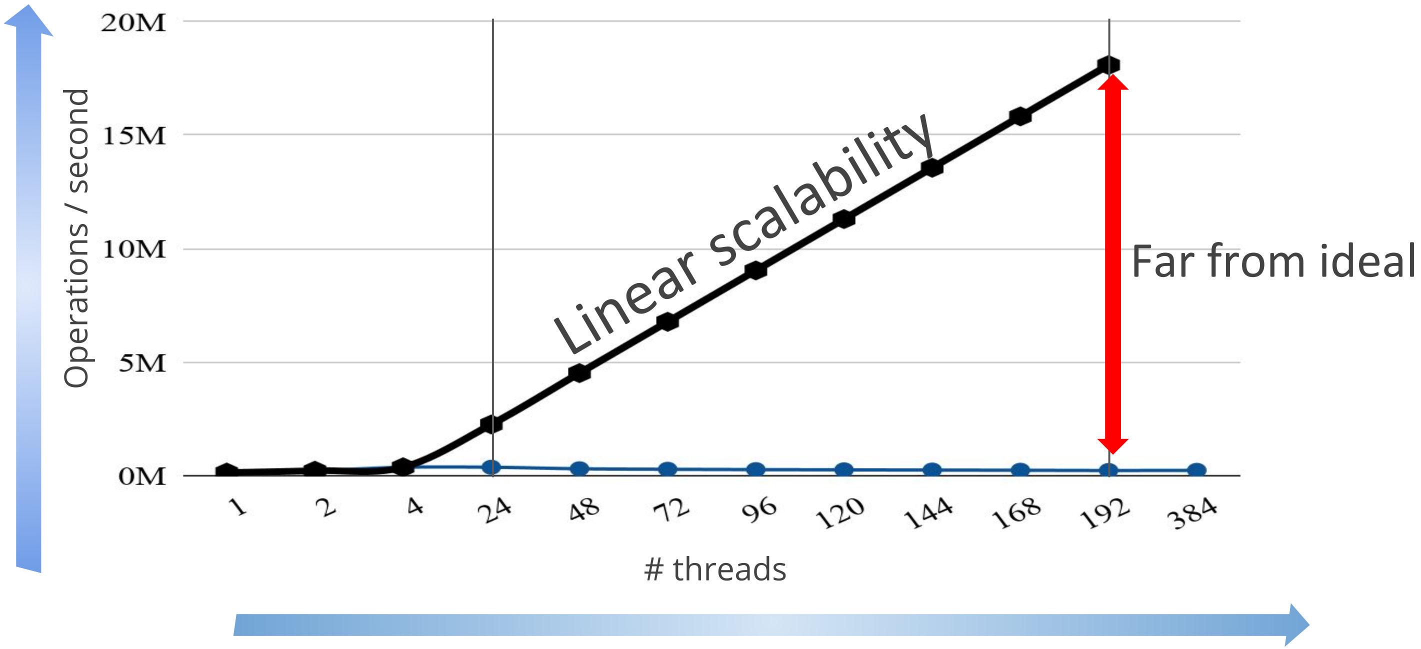
Locks are critical for application performance

A ***typical*** performance graph on manycore machines (e.g., 192-core/8-socket)



Locks are critical for application performance

A ***typical*** performance graph on manycore machines (e.g., 192-core/8-socket)



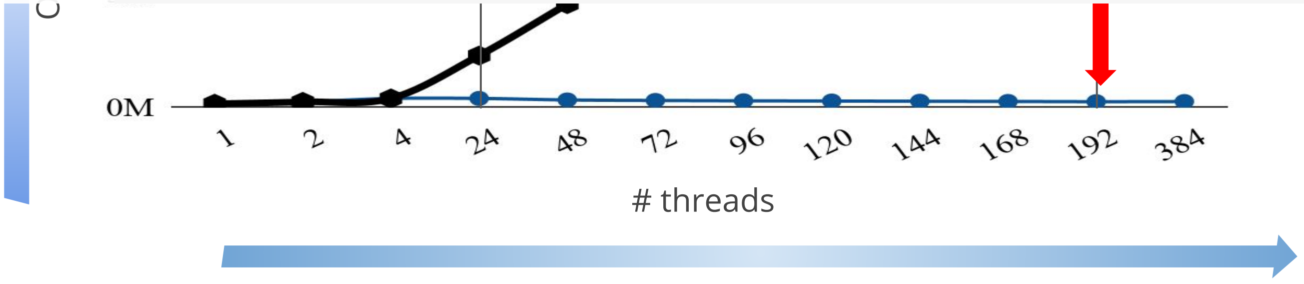
Future hardware further exacerbates the problem

Intel to Offer Socketed 56-core Cooper Lake Xeon Scalable in new Socket Compatible with Ice Lake

by Dr. Ian Cutress on August 6, 2019 8:01 AM EST

AMD's New 280W 64-Core Rome CPU: The EPYC 7H12

by Dr. Ian Cutress on September 18, 2019 9:15 AM EST



Two dimensions of lock design/goals

1) High throughput

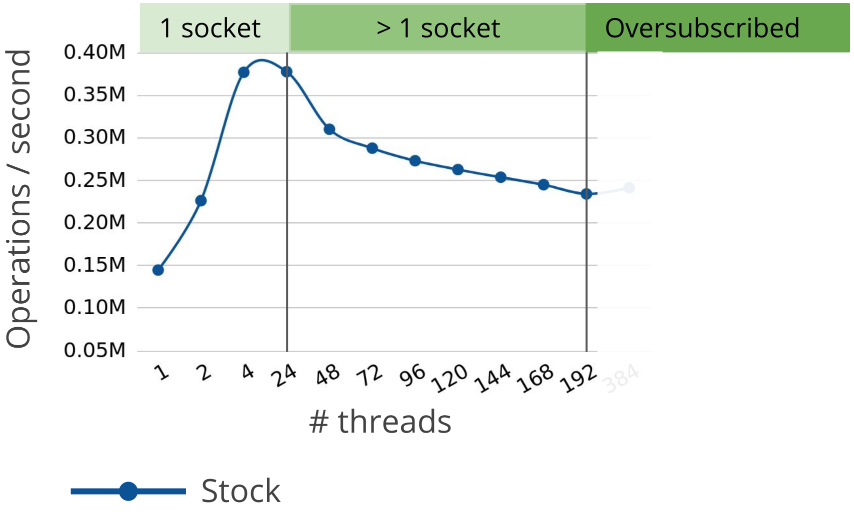
- In high thread count ⇒ Minimize lock contentions
- In single thread ⇒ No penalty when not contended
- In oversubscription ⇒ Avoid bookkeeping overhead

2) Minimal lock size

- Memory footprint ⇒ Scales to millions of locks (e.g., file inode)

Locks performance: Throughput

(e.g., each thread creates a file, a serial operation, in a shared directory)



- **Performance crashes after 1 socket**

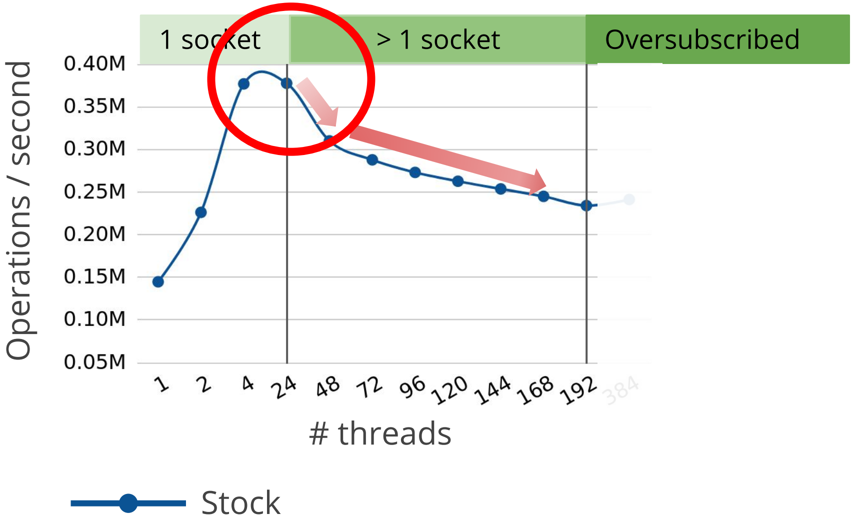
Due to non-uniform memory access (NUMA)

Accessing local socket memory is faster than the remote socket memory.

Setup: 192-core/8-socket machine

Locks performance: Throughput

(e.g., each thread creates a file, a serial operation, in a shared directory)



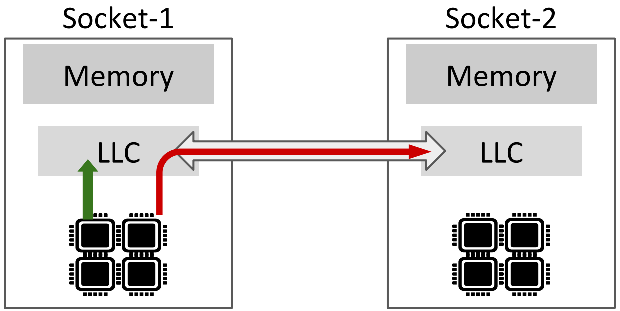
Setup: 192-core/8-socket machine

- **Performance crashes after 1 socket**

Due to non-uniform memory access (NUMA)

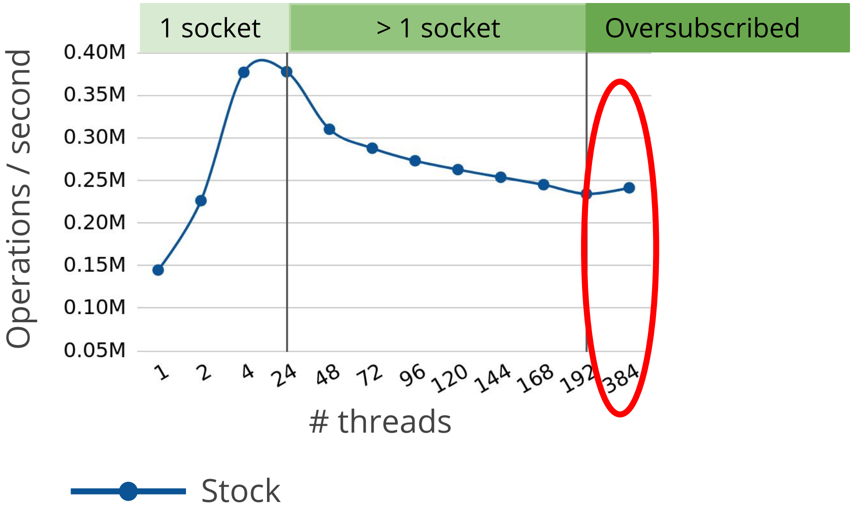
↓

Accessing local socket memory is faster than the remote socket memory.



Locks performance: Throughput

(e.g., each thread creates a file, a serial operation, in a shared directory)



Setup: 192-core/8-socket machine

- **Performance crashes after 1 socket**

Due to non-uniform memory access (NUMA)

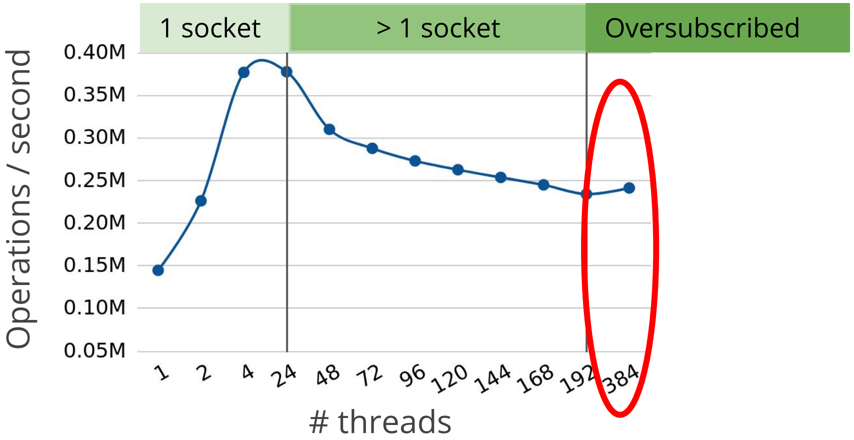
↓

Accessing local socket memory is faster than the remote socket memory.

- **NUMA also affects oversubscription**

Locks performance: Throughput

(e.g., each thread creates a file, a serial operation, in a shared directory)



- **Performance crashes after 1 socket**

  Due to non-uniform memory access (NUMA)

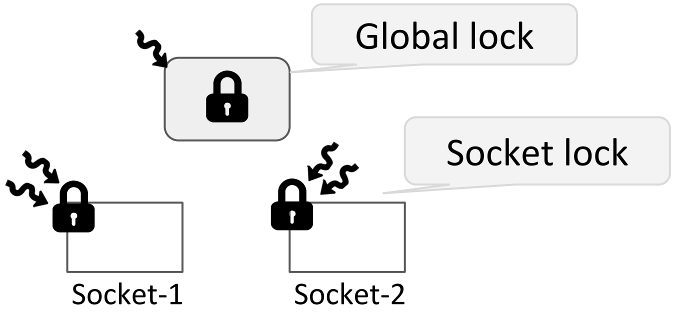
  Accessing local socket memory is faster than the remote socket memory.

- **NUMA also affects oversubscription**

Prevent throughput crash after **one socket**

Existing research efforts

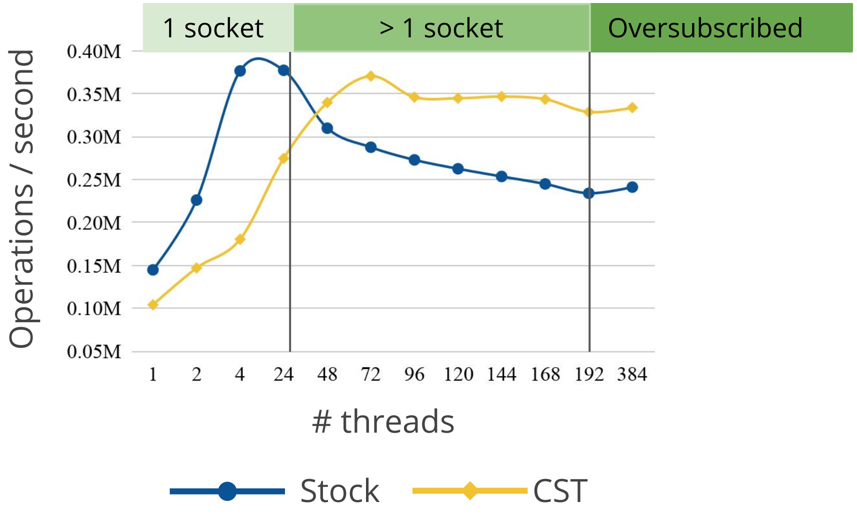
- Making locks NUMA-aware:
  - Two level locks: per-socket and global
  - Generally **hierarchical**
- Problems:
  - **Require extra memory allocation**
  - **Do not care about single thread throughput**
- Example: CST[1]



1. Scalable NUMA-aware Blocking Synchronization Primitives. ATC 2017.

Locks performance: Throughput

(e.g., each thread creates a file, a serial operation, in a shared directory)

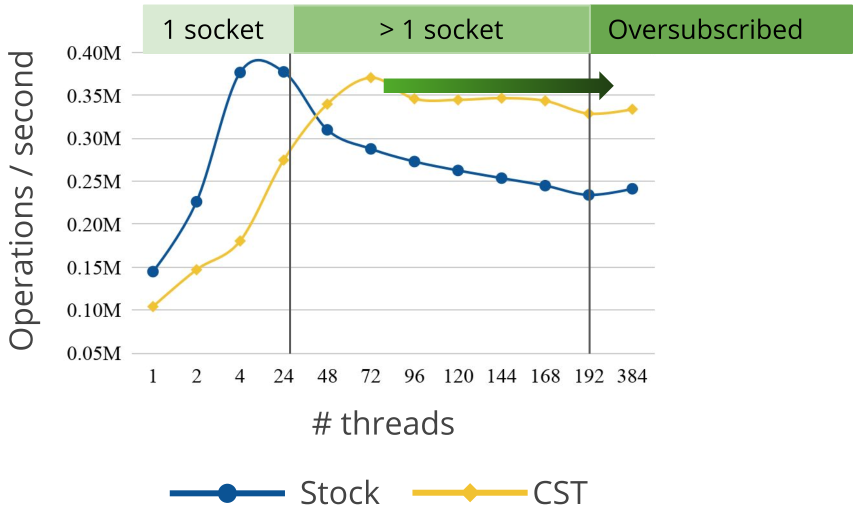


- **Maintains throughput:**
  Beyond one socket (high thread count)
  In oversubscribed case (384 threads)
- **Poor single thread throughput**
  Multiple atomic instructions

Setup: 192-core/8-socket machine

Locks performance: Throughput

(e.g., each thread creates a file, a serial operation, in a shared directory)

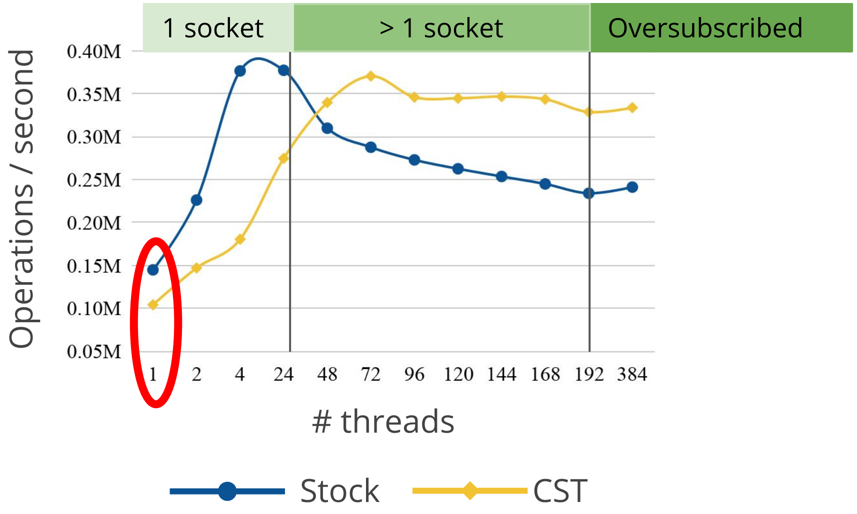


- **Maintains throughput:**
  Beyond one socket (high thread count)
  In oversubscribed case (384 threads)
- **Poor single thread throughput**
  Multiple atomic instructions

Setup: 192-core/8-socket machine

Locks performance: Throughput

(e.g., each thread creates a file, a serial operation, in a shared directory)

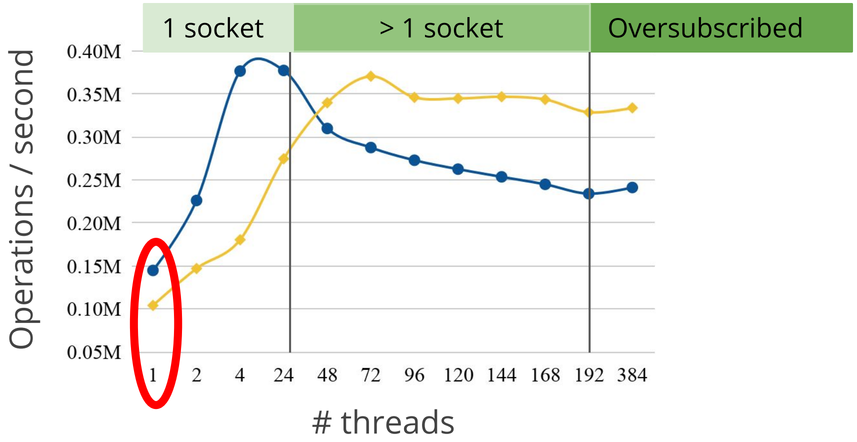


- **Maintains throughput:**
  Beyond one socket (high thread count)
  In oversubscribed case (384 threads)
- **Poor single thread throughput**
  Multiple atomic instructions

Setup: 192-core/8-socket machine

Locks performance: Throughput

(e.g., each thread creates a file, a serial operation, in a shared directory)

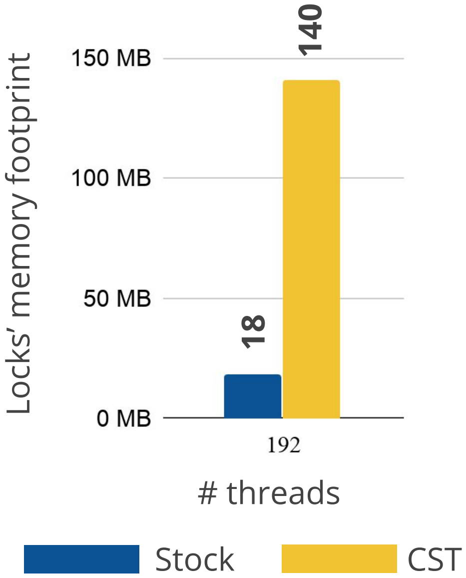


- **Maintains throughput:**
  Beyond one socket (high thread count)
  In oversubscribed case (384 threads)
- **Poor single thread throughput**
  Multiple atomic instructions

Single thread matters in non-contended cases

Locks performance: Memory footprint

(e.g., each thread creates a file, a serial operation, in a shared directory)



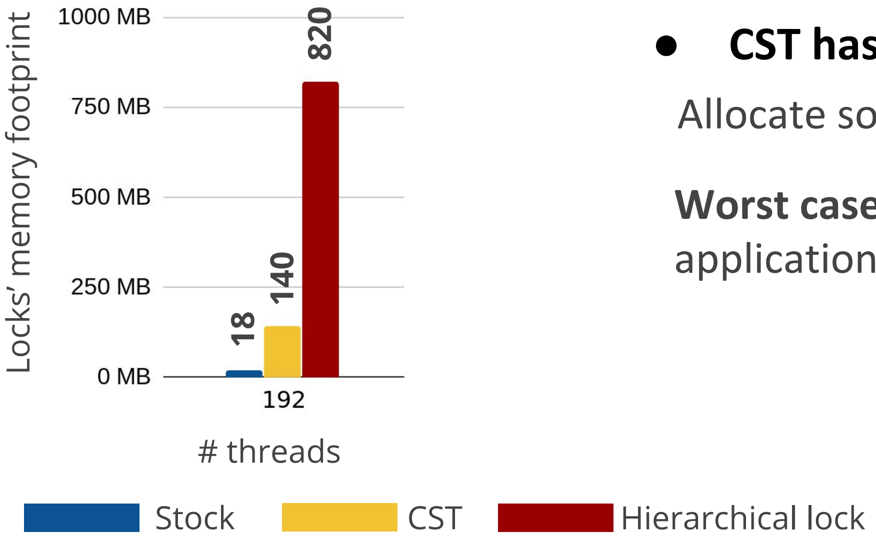
- **CST has large memory footprint**

Allocate socket structure and global lock

Worst case: ~1 GB footprint out of 32 GB application's memory

Locks performance: Memory footprint

(e.g., each thread creates a file, a serial operation, in a shared directory)



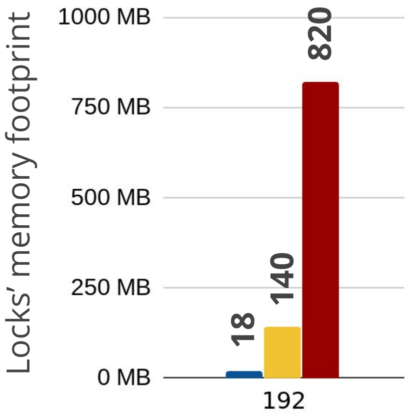
- **CST has large memory footprint**

  Allocate socket structure and global lock

  Worst case: ~1 GB footprint out of 32 GB application's memory

Locks performance: Memory footprint

(e.g., each thread creates a file, a serial operation, in a shared directory)



- **CST has large memory footprint**

  Allocate socket structure and global lock

  Worst case: ~1 GB footprint out of 32 GB application's memory

Lock's memory footprint affect its adoption

Two goals in our new lock

1) **NUMA-aware** lock with **no memory** overhead

2) **High throughput** in **both** low/high thread count

Key idea: Sort waiters on the fly

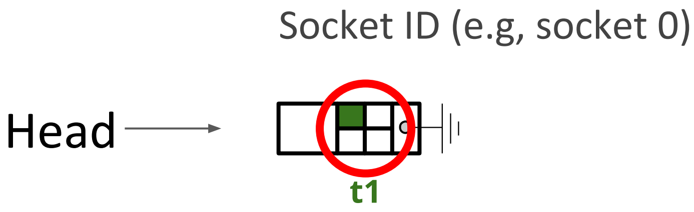
Observations:

Hierarchical locks avoid NUMA by passing the lock within a socket

Queue-based locks already maintain a list of waiters

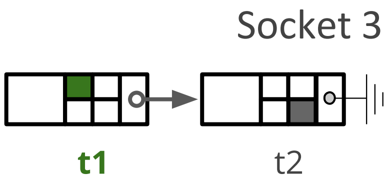
Sort waiters on the fly using socket ID

A waiting queue



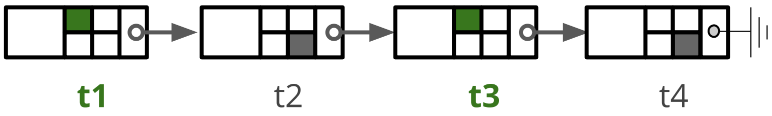
Sort waiters on the fly using socket ID

Another waiter is in a different socket



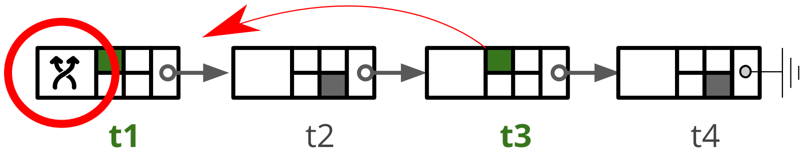
Sort waiters on the fly using socket ID

More waiters join



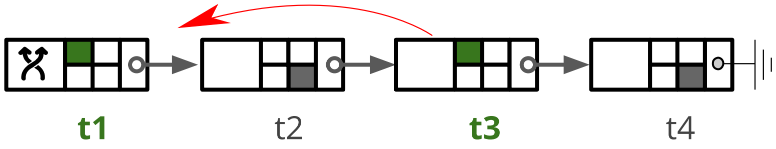
Sort waiters on the fly using socket ID

Shuffler (t1) sorts based on socket ID



Shuffling: Design methodology

A waiter (**shuffler** ⤲) reorders the queue of waiters



- A ***waiter***, otherwise spinning (i.e,. wasting), amortises the cost of lock ops
  1) By reordering (e.g., lock orders)
  2) By modifying waiters' states (e.g., waking-up/sleeping)

⟶ Shuffler **computes** NUMA-ness **on the fly without using memory**

Shuffling is generic!

Shuffling is generic!

A shuffler can modify the queue or a waiter's state with a **defined function/policy**!

Shuffling is generic!

A shuffler can modify the queue or a waiter’s state with a **defined function/policy**!

Blocking lock: wake up a nearby sleeping waiter

RWlock: Group writers together

Shuffling is generic!

A shuffler can modify the queue or a waiter's state with a **defined function/policy**!

Blocking lock: wake up a nearby sleeping waiter

RWlock: Group writers together

Incorporate **shuffling** in lock design

SHFLLOCKS

Minimal footprint locks
that handle any thread contention

ShflLocks

TAS (4B)
(test-and-set lock)

Queue tail (8B)
(waiters list)

ShflLocks

TAS (4B)
(test-and-set lock)

Queue tail (8B)
(waiters list)

- Decouples the lock holder and waiters
  - Lock holder holds the TAS lock
  - Waiters join the queue

SHFLLOCKS

TAS (4B) (test-and-set lock)
Queue tail (8B) (waiters list)

- Decouples the lock holder and waiters
  - Lock holder holds the TAS lock
  - Waiters join the queue

`lock()`: Try acquiring the TAS lock first; join the queue on failure

`unlock()`: Unlock the TAS lock (reset the TAS word to 0)

SHFLLOCKS

TAS (4B)
(test-and-set lock)

Queue tail (8B)
(waiters list)

TAS maintains single thread performance

SHFLLOCKS

TAS (4B)
(test-and-set lock)

Queue tail (8B)
(waiters list)

TAS maintains single thread performance

- Waiters use **shuffling** to improve application throughput
  - NUMA-awareness, efficient wake up strategy
  - Utilizing Idle/CPU wasting waiters

★ **Shuffling is off the critical path most of the time**

- Maintain long-term fairness:
  - Bound the number of shuffling rounds

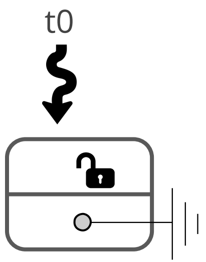
NUMA-aware SHFLLOCK in action

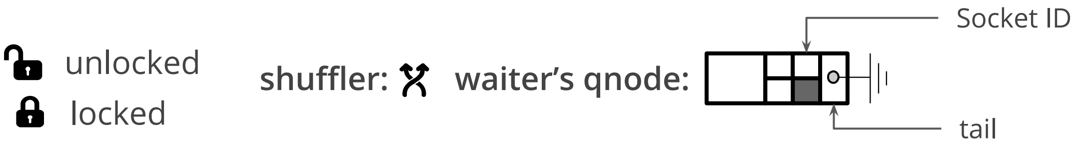
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





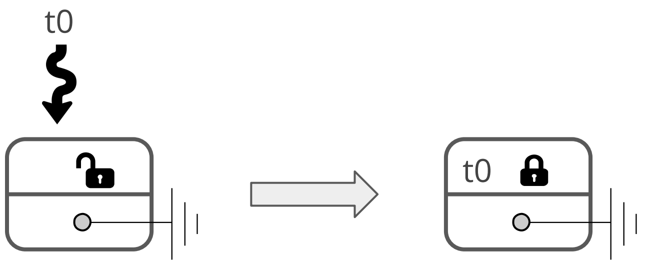
NUMA-aware SHFLLOCK in action

t0 (socket 0): lock()

Multiple threads join the queue

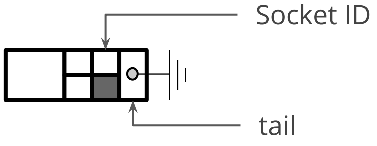
Shuffling in progress

t0: unlock()



unlocked
locked

shuffler: **waiter's qnode:**



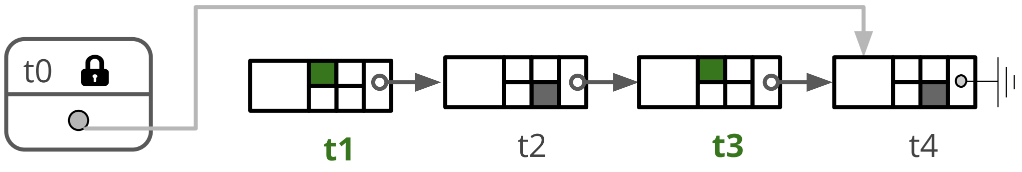
NUMA-aware SHFLLOCK in action

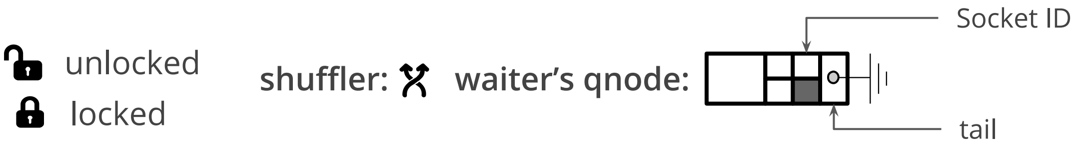
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





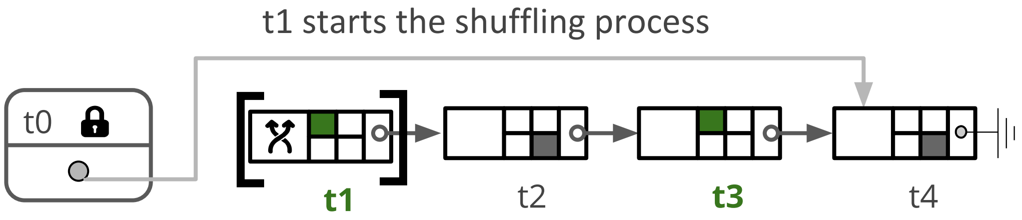
NUMA-aware SHFLLOCK in action

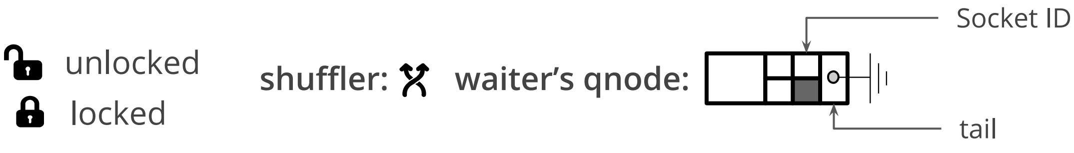
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





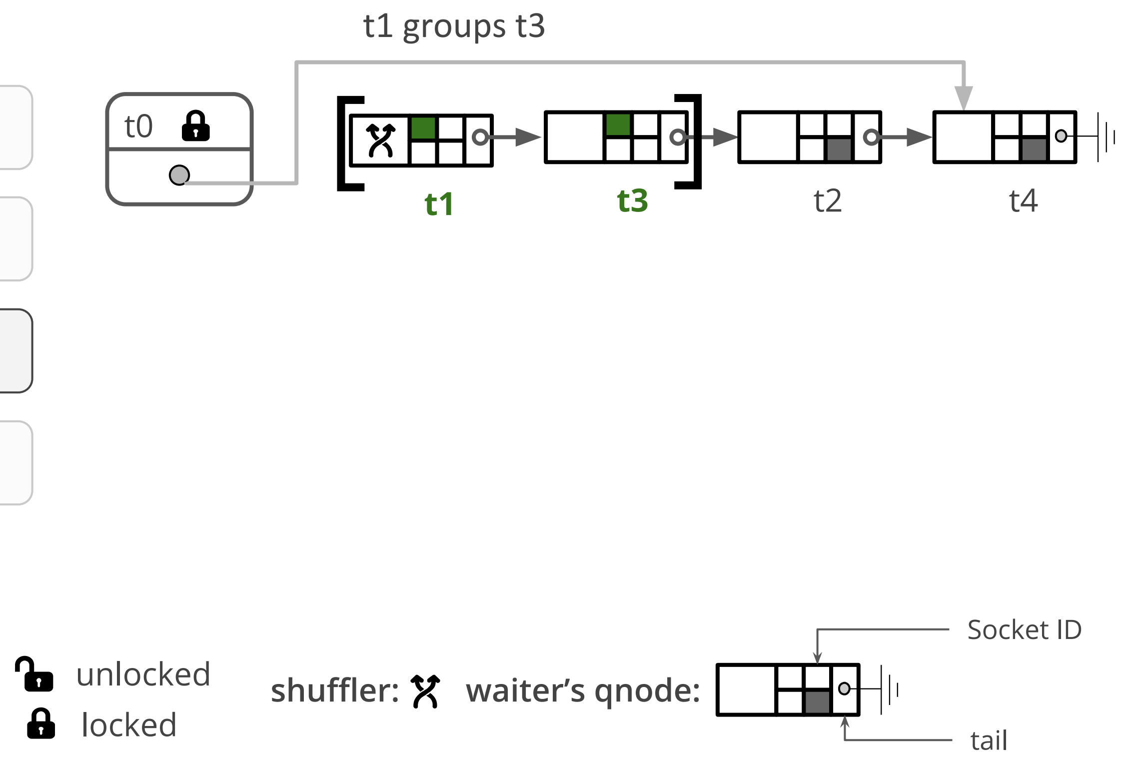
NUMA-aware SHFLLOCK in action

t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()



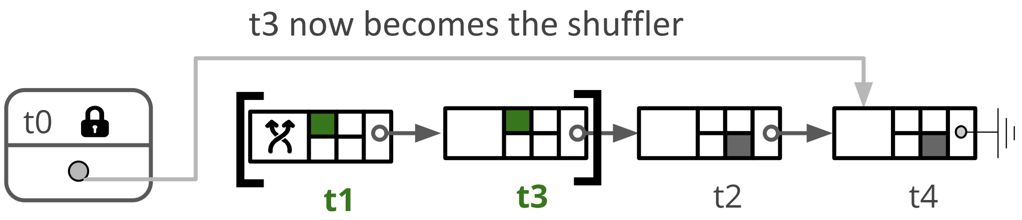
NUMA-aware SHFLLOCK in action

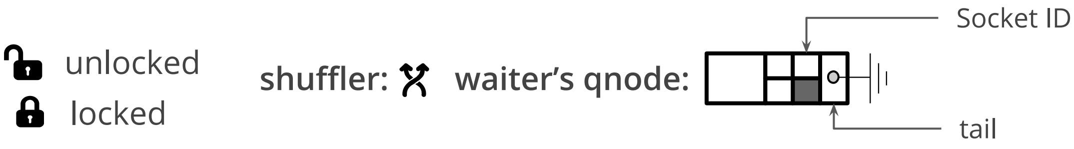
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





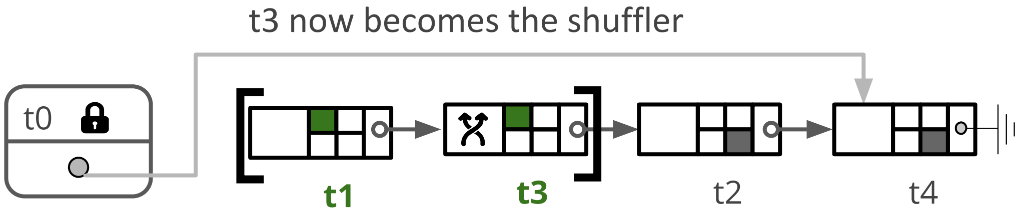
NUMA-aware SHFLLOCK in action

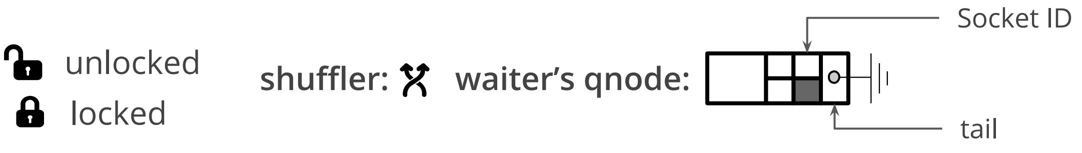
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





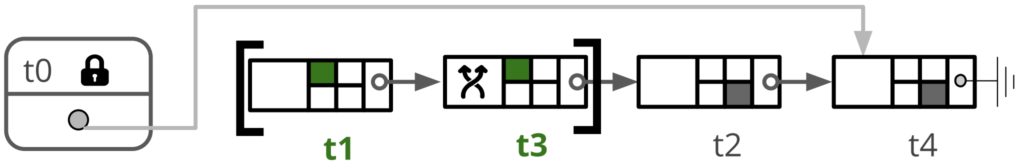
NUMA-aware SHFLLOCK in action

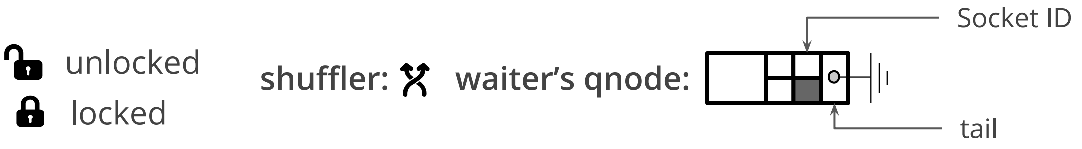
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





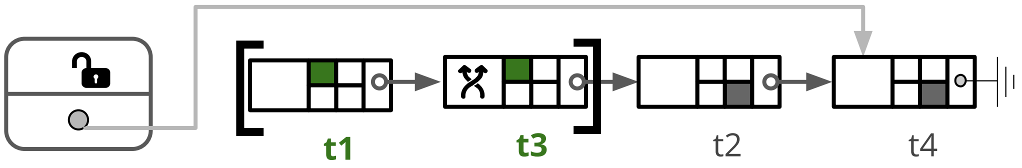
NUMA-aware SHFLLOCK in action

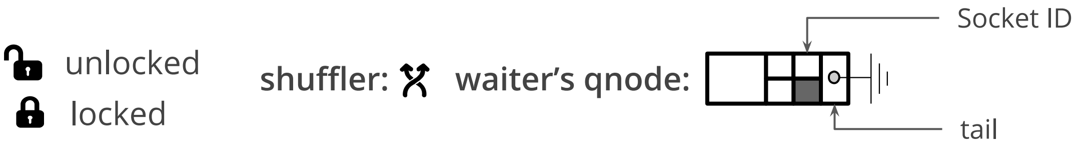
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





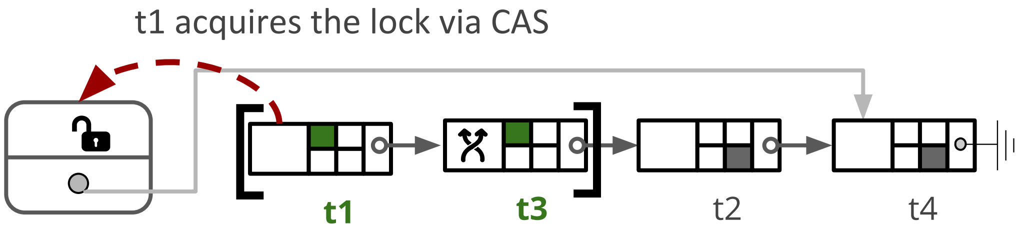
NUMA-aware SHFLLOCK in action

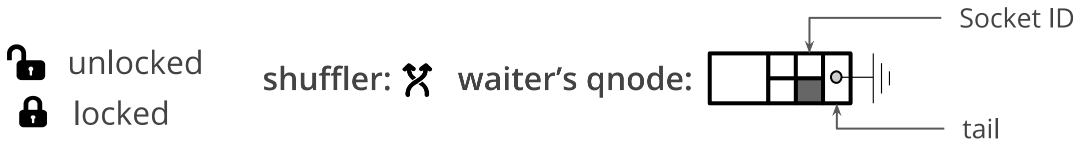
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





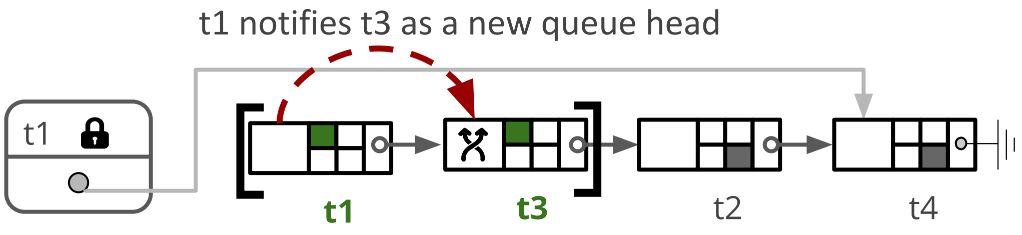
NUMA-aware SHFLLOCK in action

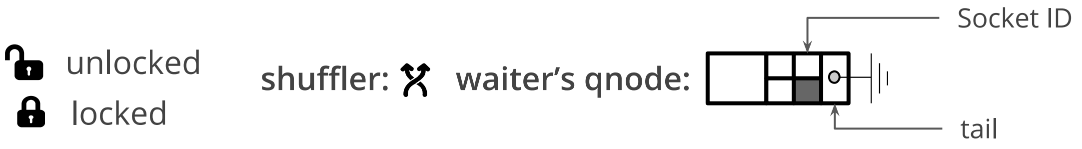
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





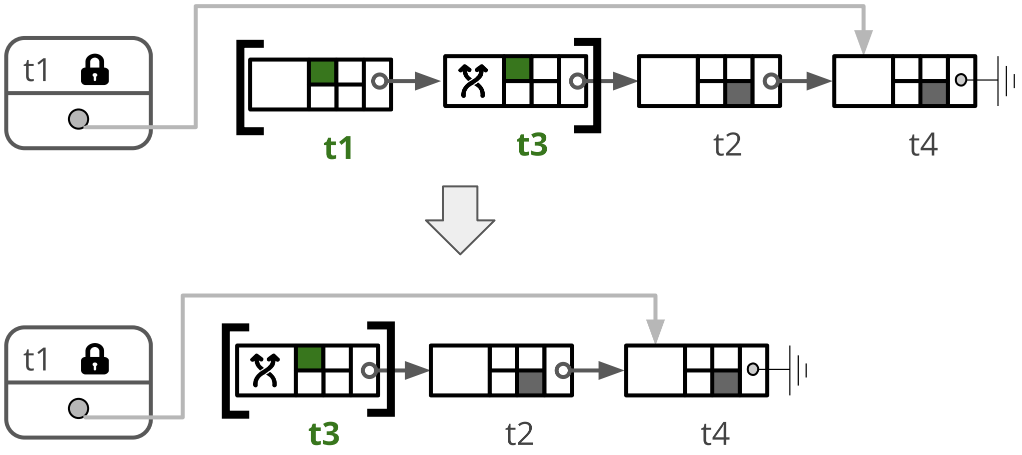
NUMA-aware SHFLLOCK in action

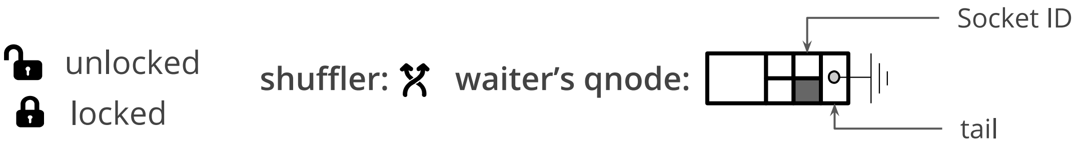
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





Implementation

- Kernel space:
  - Replaced ***all*** mutex and rwsem
  - Modified slowpath of the qspinlock
- User space:
  - Added to the LiTL library
- Please see our paper:
  - **Blocking lock**: Wake up nearby shuffled waiters
  - **Readers-writer lock**: Centralized rw-indicator + SHFLLOCK

https://github.com/sslab-gatech/shfllock

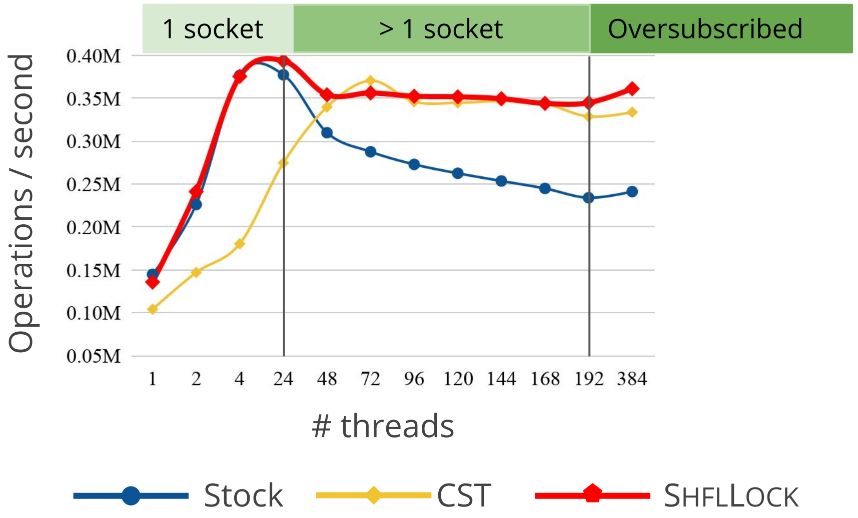
Evaluation

- SHFLLOCK performance:
  - Does shuffling maintains application's throughput?
  - What is the overall memory footprint?

Setup: 192-core/8-socket machine

Locks performance: Throughput

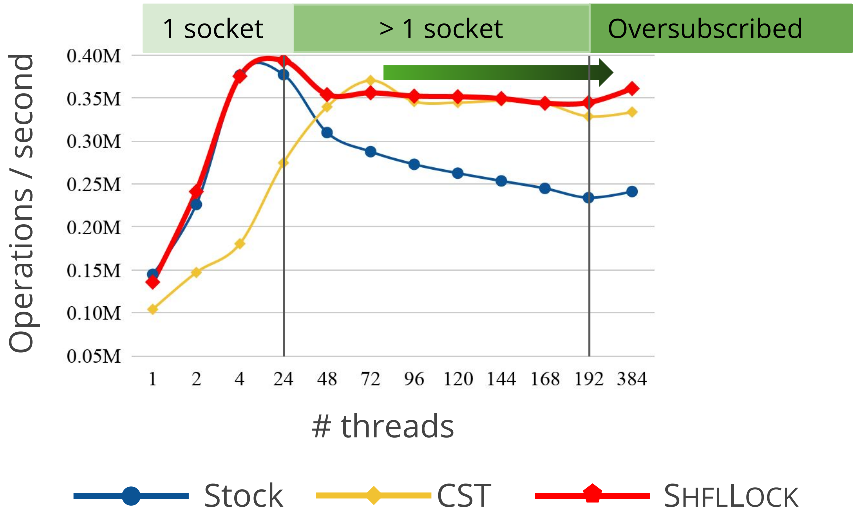
(e.g., each thread creates a file, a serial operation, in a shared directory)



- SHFLLOCKS maintain performance:

Locks performance: Throughput

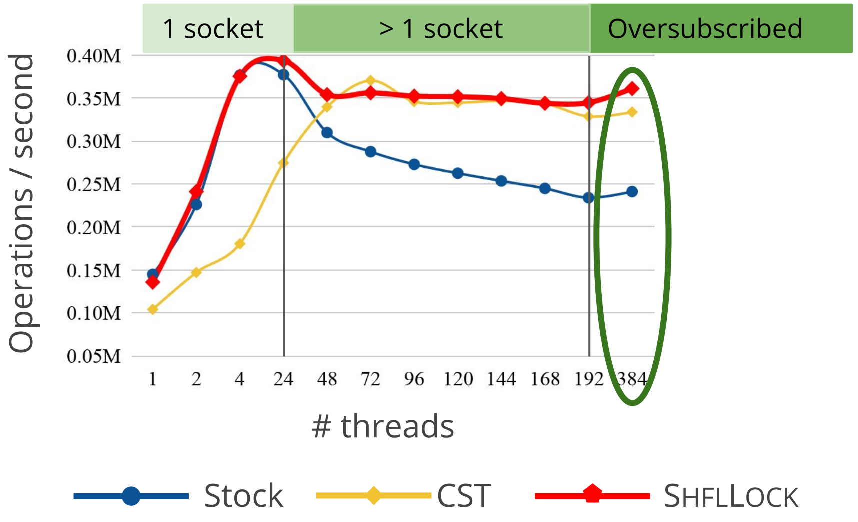
(e.g., each thread creates a file, a serial operation, in a shared directory)



- SHFLLOCKS maintain performance:
  - Beyond one socket
    - **NUMA-aware shuffling**

Locks performance: Throughput

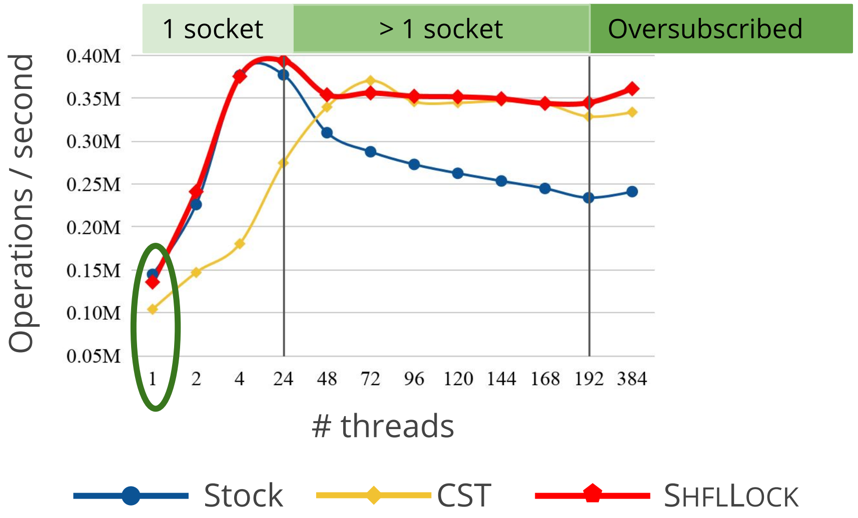
(e.g., each thread creates a file, a serial operation, in a shared directory)



- SHFLLOCKS maintain performance:
  - Beyond one socket
    - **NUMA-aware shuffling**
  - Core oversubscription
    - **NUMA-aware + wakeup shuffling**

Locks performance: Throughput

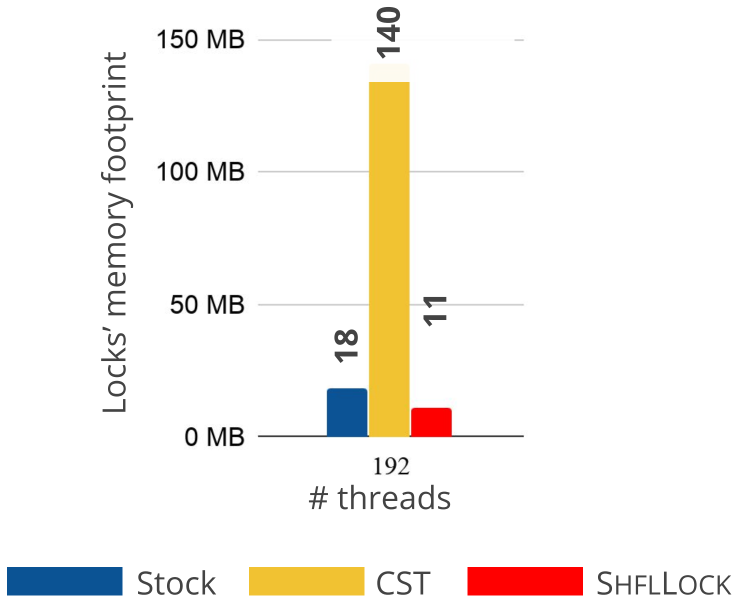
(e.g., each thread creates a file, a serial operation, in a shared directory)



- SHFLLOCKS maintain performance:
  - Beyond one socket
    - **NUMA-aware shuffling**
  - Core oversubscription
    - **NUMA-aware + wakeup shuffling**
  - Single thread
    - **TAS acquire and release**

Locks performance: Memory footprint

(e.g., each thread creates a file, a serial operation, in a shared directory)



- SHFLLOCK has least memory footprint

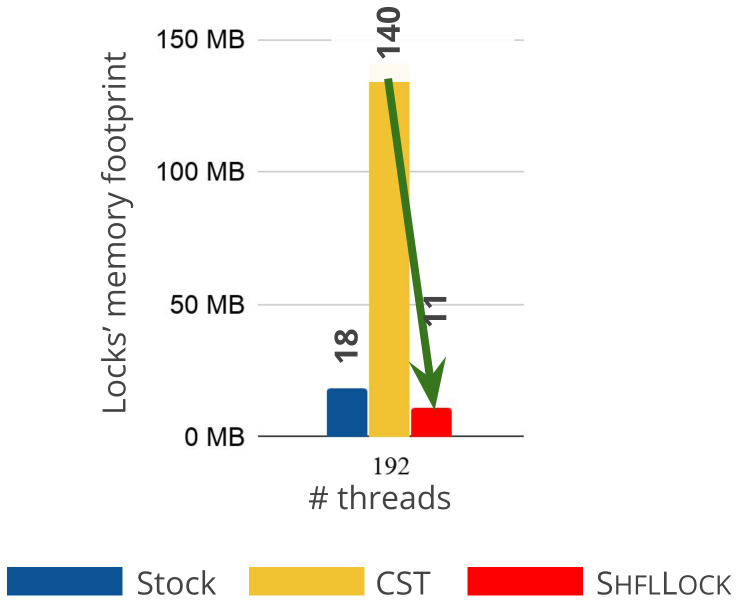
  Reason: No extra auxiliary data structure

  ➢ Stock: parking list structure + extra lock

  ➢ CST: per-socket structure

Locks performance: Memory footprint

(e.g., each thread creates a file, a serial operation, in a shared directory)



- SHFLLOCK has least memory footprint

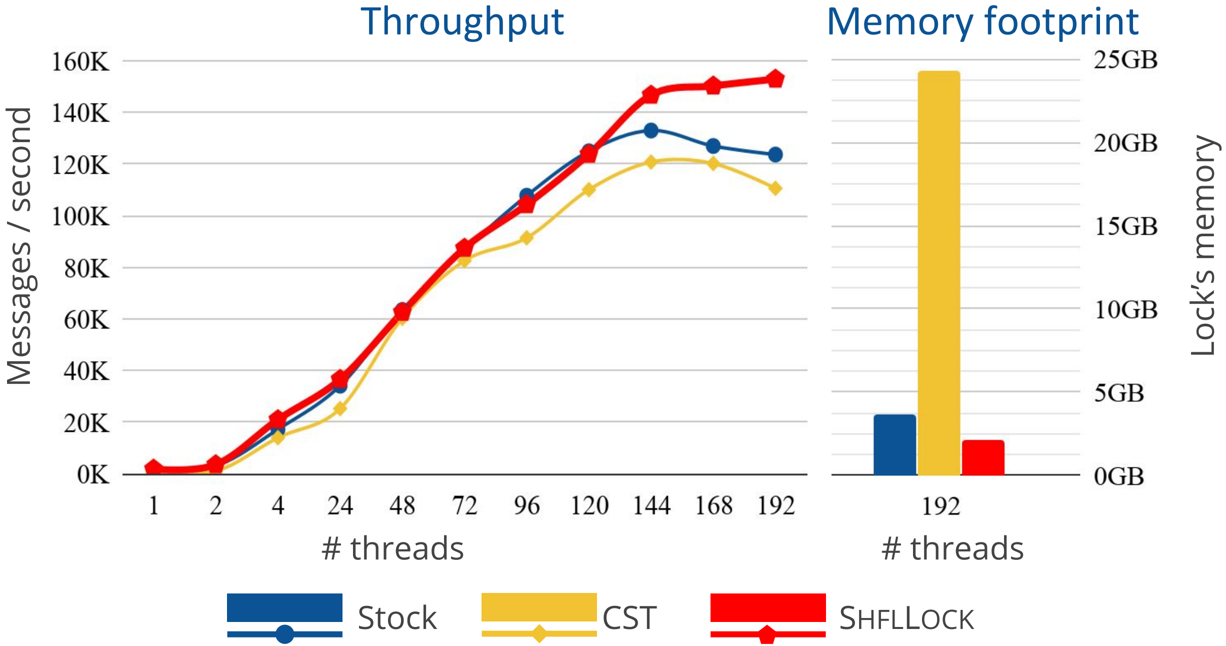
  Reason: No extra auxiliary data structure

  ➢ Stock: parking list structure + extra lock

  ➢ CST: per-socket structure

Case study: Exim mail server

It is fork intensive and stresses memory subsystem, file system and scheduler

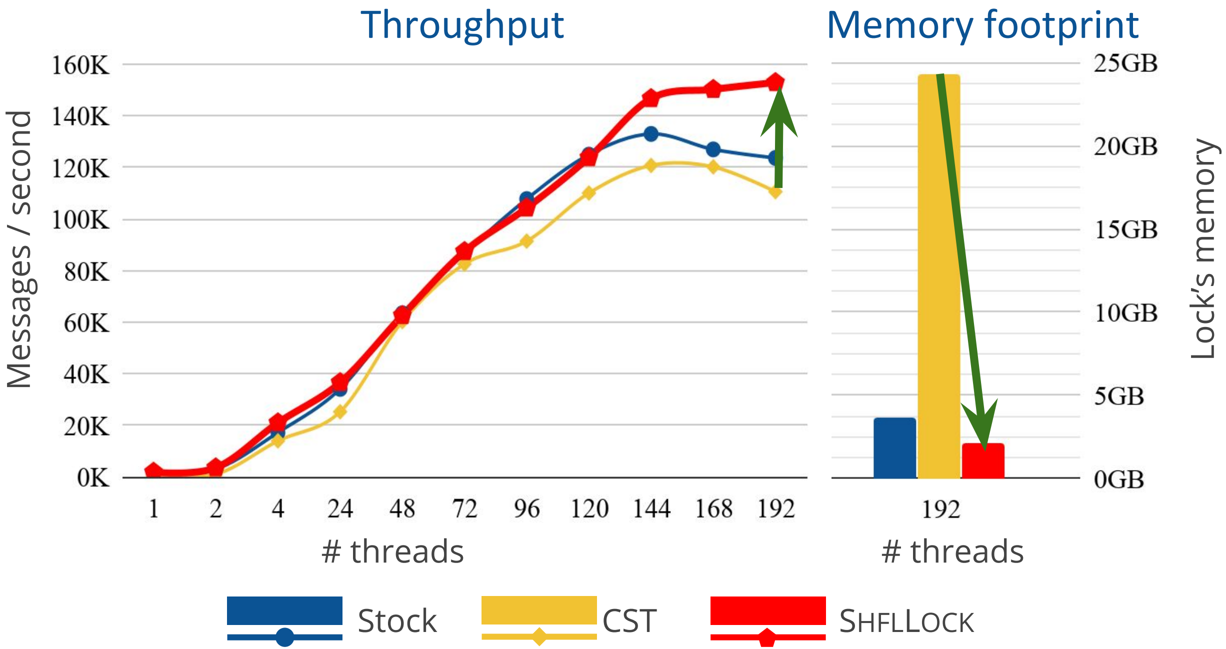


Improves throughput by up to 1.5x

Decreases memory footprint up to 93%

Case study: Exim mail server

It is fork intensive and stresses memory subsystem, file system and scheduler



Improves throughput by up to 1.5x

Decreases memory footprint up to 93%

Conclusion

- Current lock designs:
  - Do not maintain best throughput with varying threads
  - Have high memory footprint
- **Shuffling**: Reorder the list or modify a waiter's state on the fly
  - NUMA-awareness, waking up waiters
- **SHFLLOCKS**: Shuffling-based family of lock algorithms
  - NUMA-aware minimal memory footprint locks
  - Utilize waiters to amortize lock operations

Conclusion

- Current lock designs:
  - Do not maintain best throughput with varying threads
  - Have high memory footprint
- **Shuffling**: Reorder the list or modify a waiter's state on the fly
  - NUMA-awareness, waking up waiters
- **SHFLLOCKS**: Shuffling-based family of lock algorithms
  - NUMA-aware minimal memory footprint locks
  - Utilize waiters to amortize lock operations

Thank you!