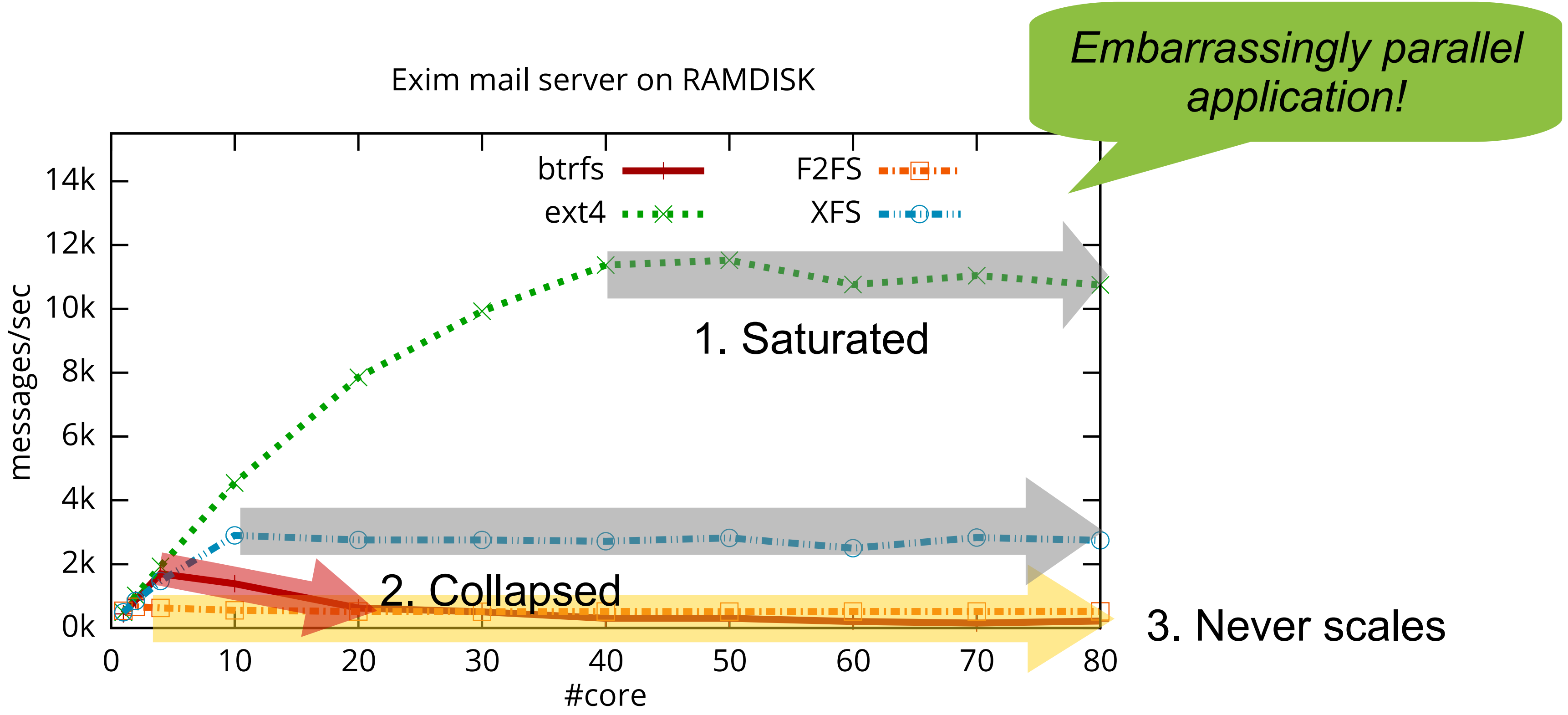
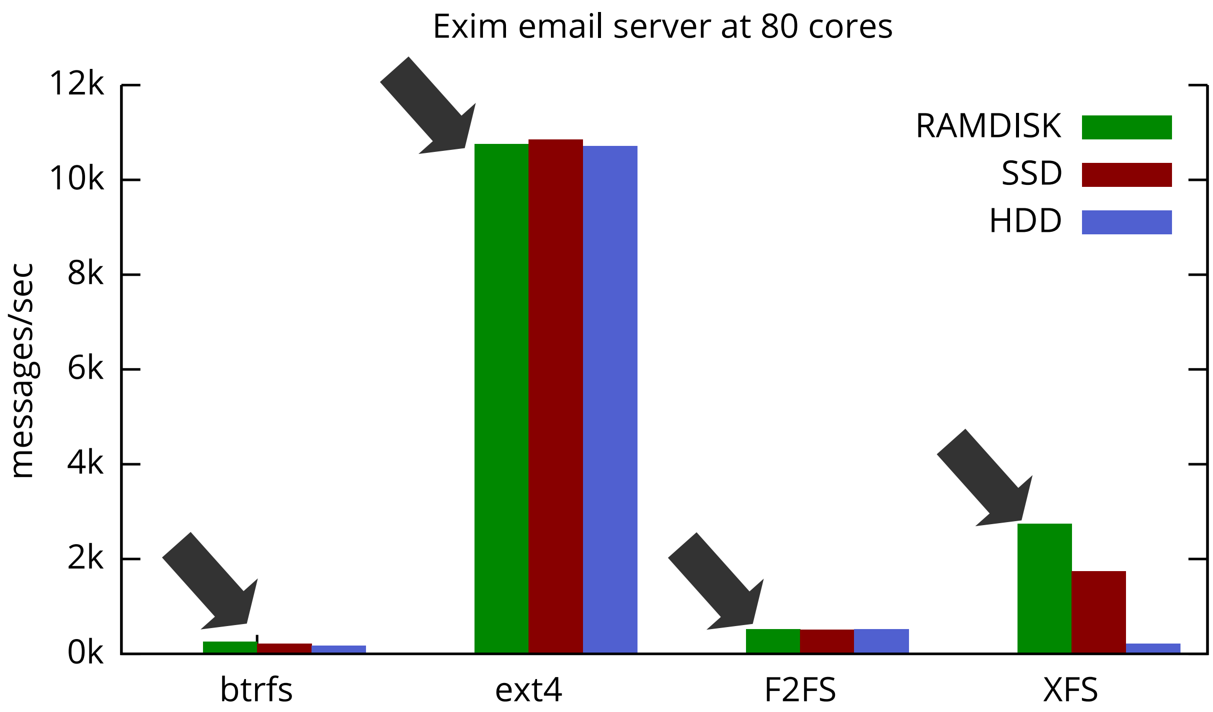
ShflLocks: Scalable and Practical Locking for Manycore Systems

Changwoo Min
COSMOSS Lab / ECE / Virginia Tech
https://cosmoss-vt.github.io/

File system becomes a bottleneck on manycore systems



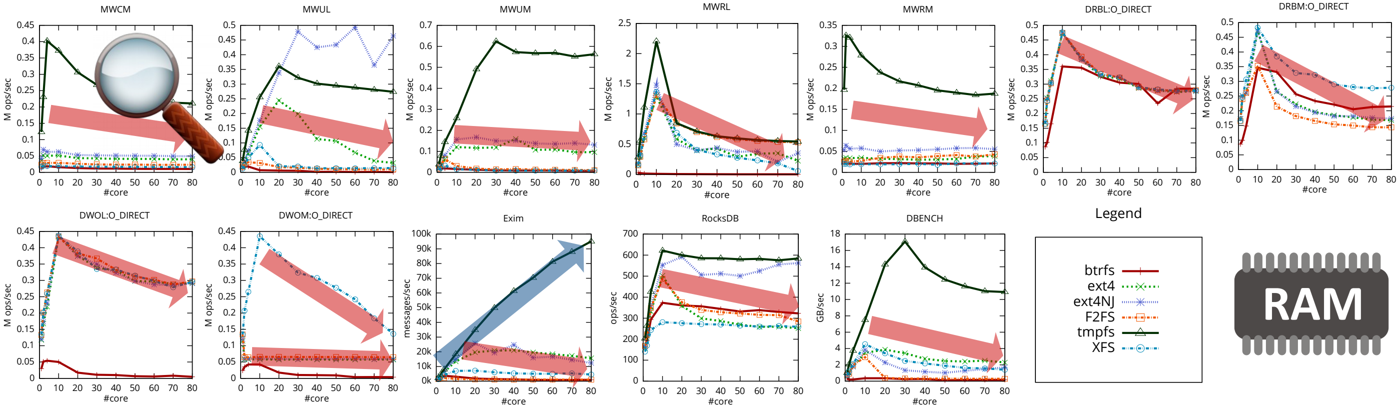
Even in slower storage medium file system becomes a bottleneck



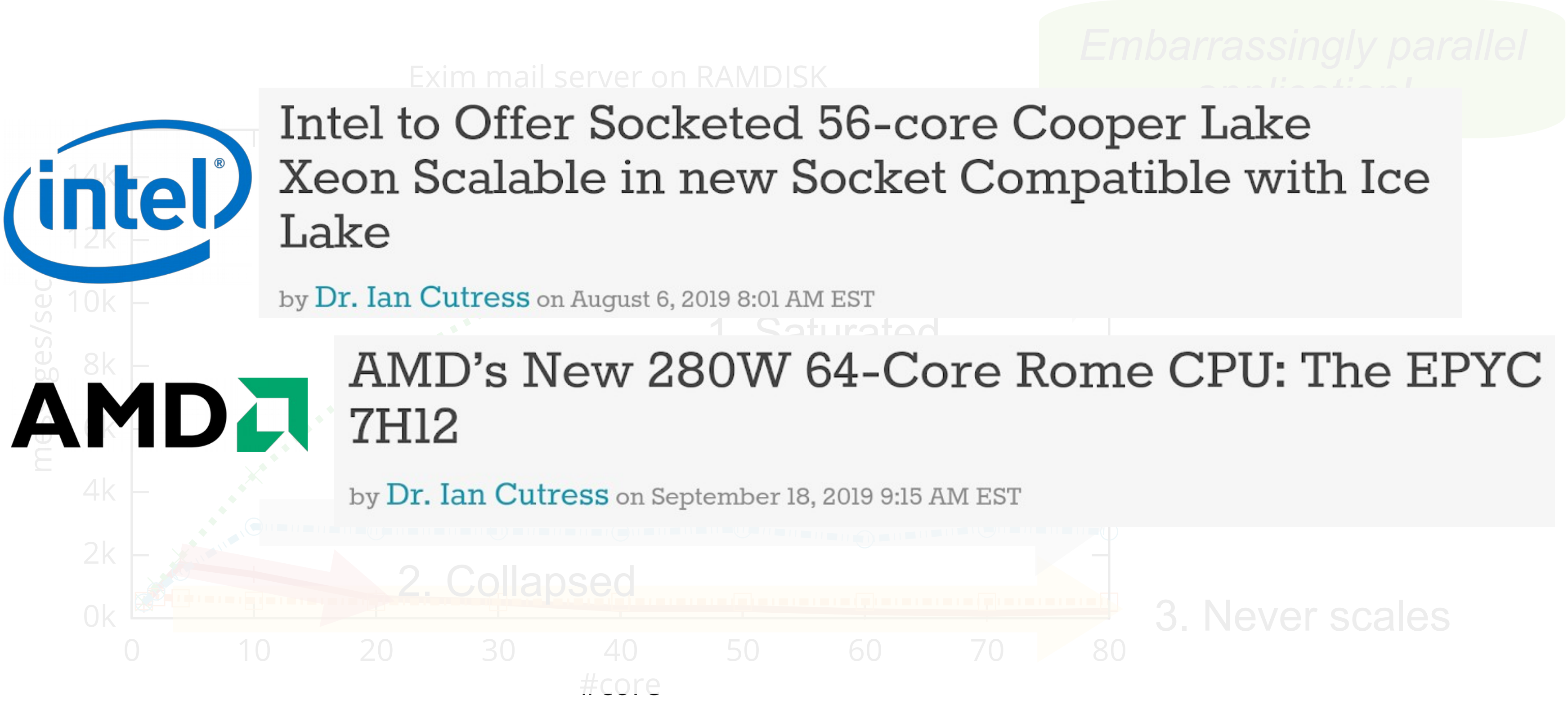
FxMark: File systems are not scalable in manycore systems

Create files on a shared directory

Locks are critical in performance and scalability



Future hardware further exacerbates the problem



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

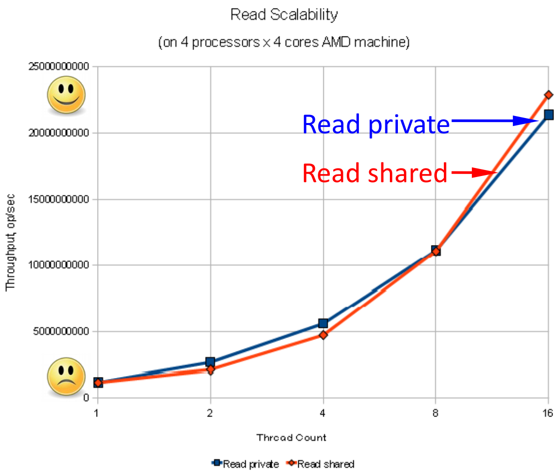
by Dr. Ian Cutress on August 6, 2019 8:01 AM EST

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

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

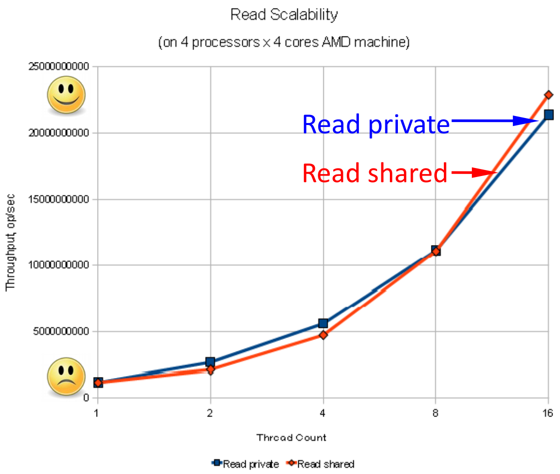
Why this happens?
: Memory access is NOT scalable

1. Read operations are scalable

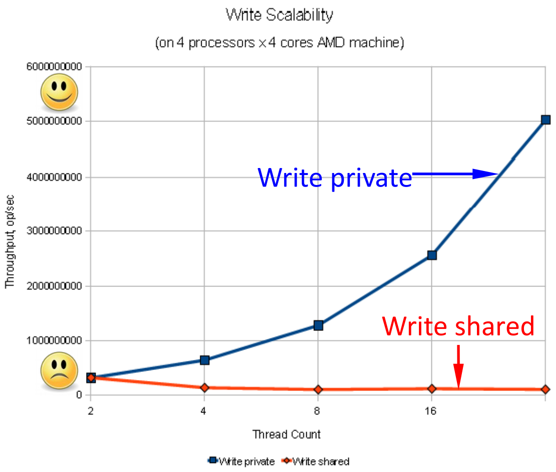


Why this happens?
: Memory access is NOT scalable

1. Read operations are scalable



2. Write operations are NOT scalable

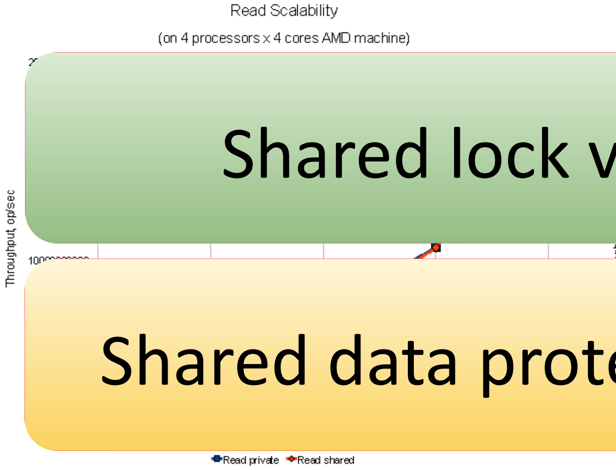


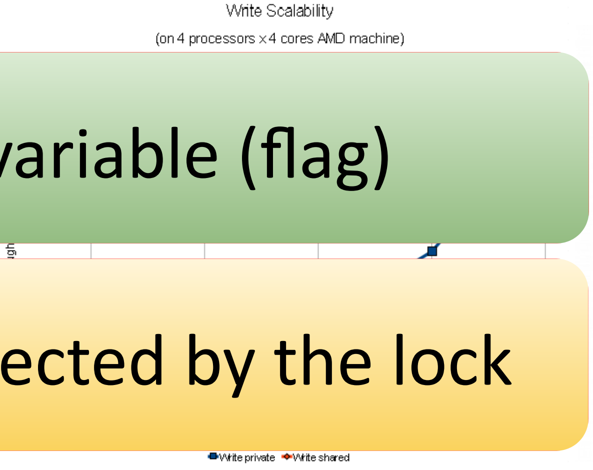
Why this happens?
: Memory access is NOT scalable

1. Read operations are scalable

2. Write operations are NOT scalable

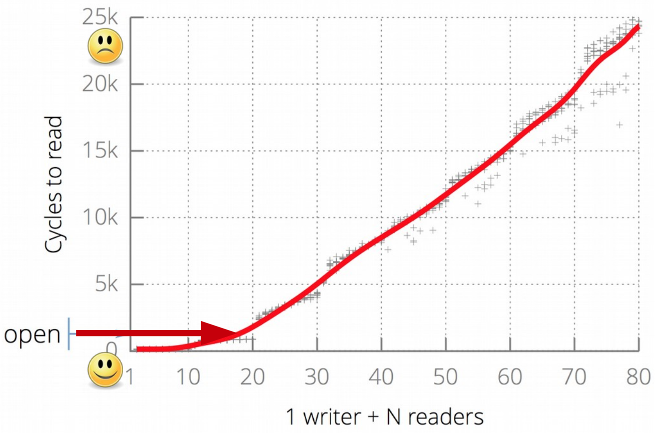
3. Write operations interfere read operations





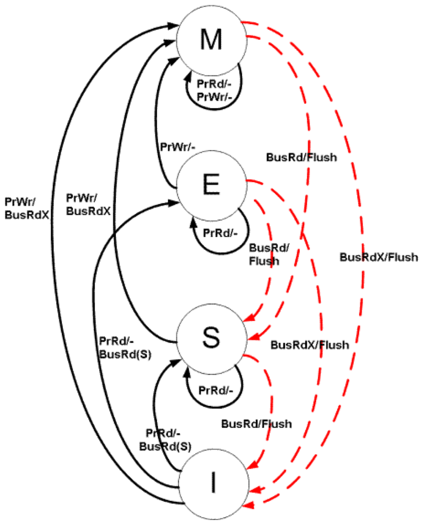
Shared lock variable (flag)

Shared data protected by the lock

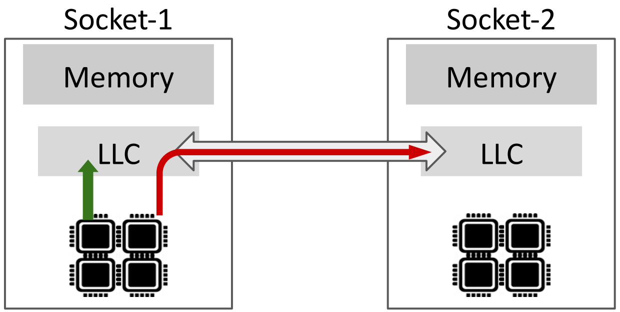


Why this happens?
: Cache coherence is not scalable

- Cache coherent traffic dominates!!!
- Writing a cache line in a popular MESI protocol:
  - Writer's cache: Shared → Exclusive
  - All readers' cache line: Shared → Invalidate



Should minimize contended cache lines and core-to-core communication traffic



Lock's research efforts and their use

Lock's research efforts

- Dekker's algorithm (1962)
- MCS lock (1991)
- HBO lock (2003)
- Hierarchical lock – HCLH (2006)
- Flat combining NUMA lock (2011)
- Remote Core locking (2012)
- Cohort lock (2012)
- RW cohort lock (2013)
- Malthusian lock (2014)
- HMCS lock (2015)
- AHMCS lock(2016)

NUMA-aware locks

Adopting new locks is necessary but it is not easy

Linux kernel lock adoption / modification

2011
- Spinlock → ticket (2.6)
- Mutex → TTAS + block (2.6)
- Rwsem → TTAS + block

2014
- Spinlock → ticket
- Mutex → TTAS + spin + block (3.16)
- Rwsem → TTAS + spin + block (3.16)

2016
- Spinlock → qspinlock (4.4)
- Mutex → TTAS + spin + block
- Rwsem → TTAS + spin + block

Two dimensions of lock design/goals

1) High throughput

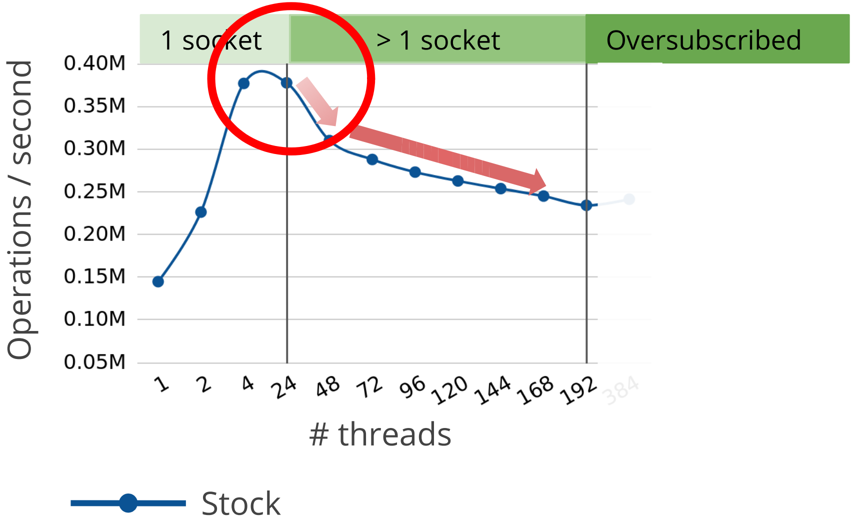
- In high thread count ⇨ Minimize lock contentions
- In single thread ⇨ No penalty when not contended
- In oversubscription ⇨ Avoid bookkeeping overheads

2) Minimal lock size

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

Locks performance: Throughput

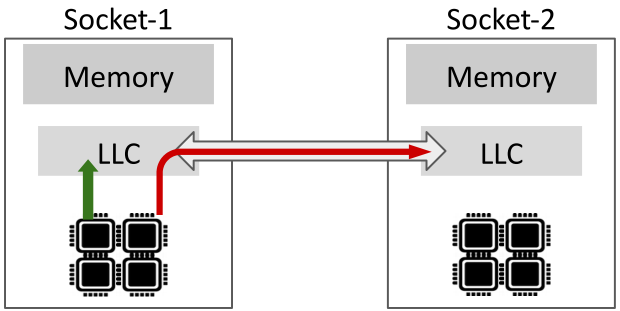
(e.g., each thread creates a file, a serial operation, in a shared directory)



- **Performance crashes after 1 socket.**

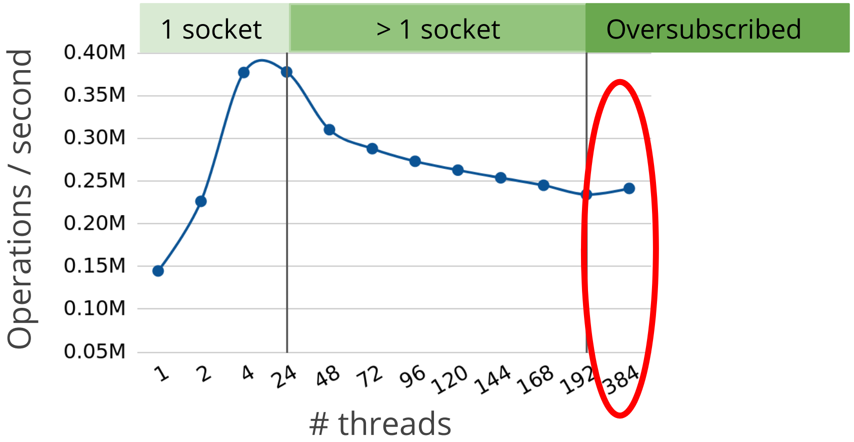
Due to non-uniform memory access (NUMA).

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



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).

↓

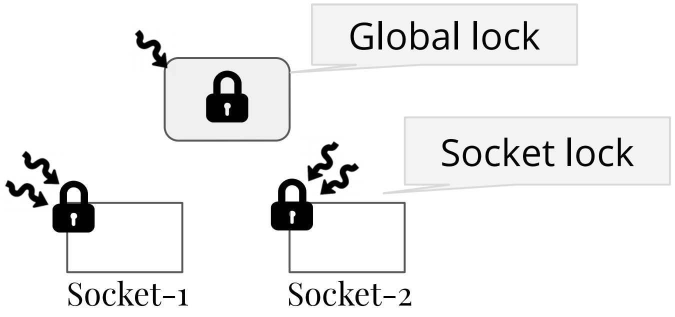
Accessing local socket memory is faster than the remote socket memory.

- **NUMA also affects oversubscription.**

Prevent throughput crash after **one socket**

Existing research efforts

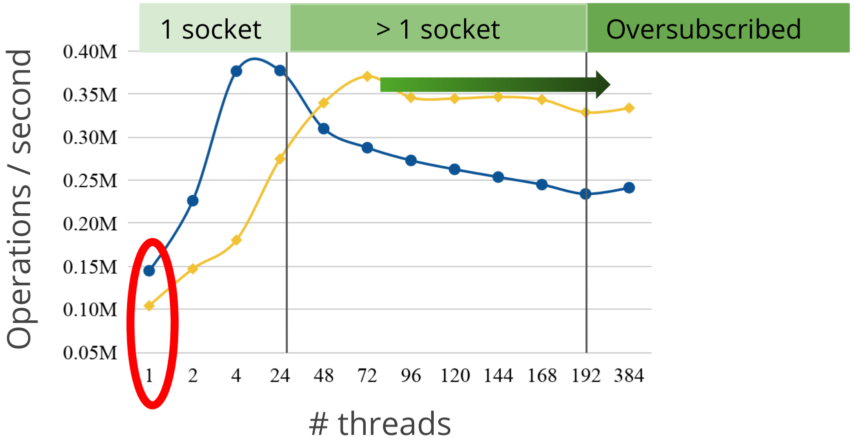
- Making locks NUMA-aware:
  - Two level locks: per-socket and global
  - Generally **hierarchical**
- Problems:
  - **Require extra memory allocation**
  - **Do not care about single thread throughput**
- Example: CST[1]



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)

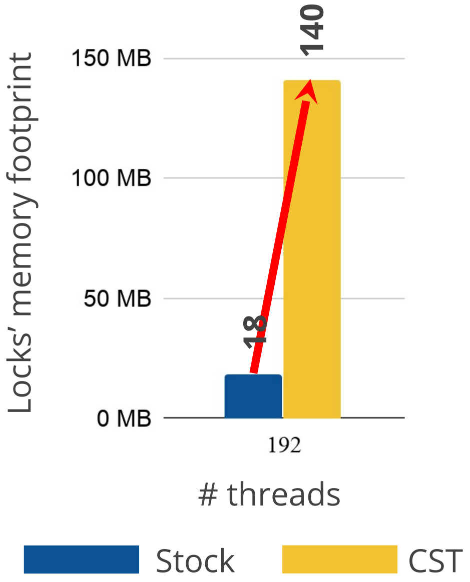


- **Maintains throughput:**
  Beyond one socket (high thread count).
  In oversubscribed case (384 threads).
- **Poor single thread throughput.**
  Multiple atomic instructions.

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)



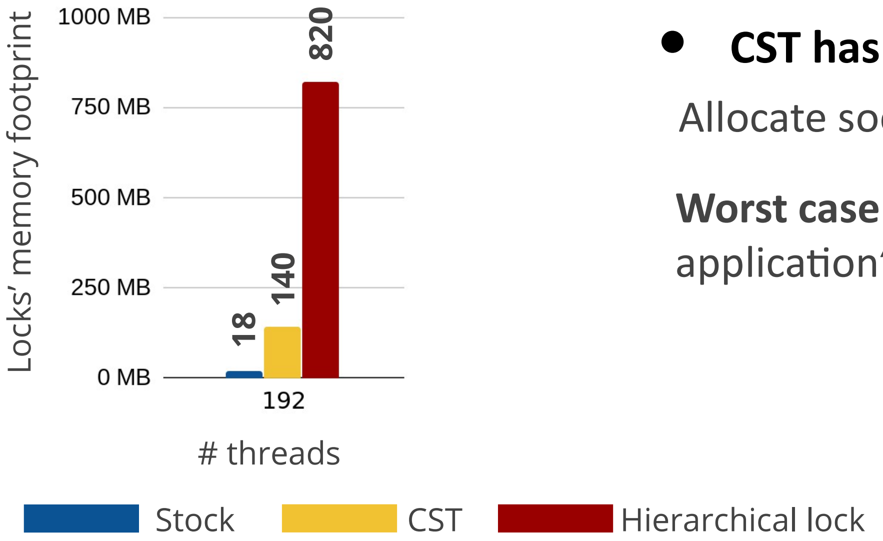
- **CST has large memory footprint.**

  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)



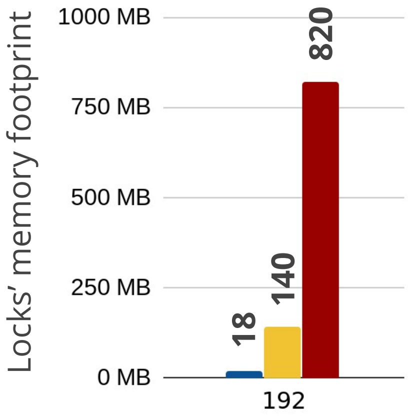
- **CST has large memory footprint.**

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:

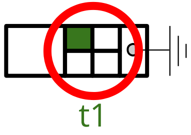
Hierarchical locks avoid NUMA by passing the lock within a socket

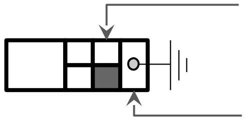
Queue-based locks already maintain a set of waiters

Shuffling: Design methodology

Representing a waiting queue

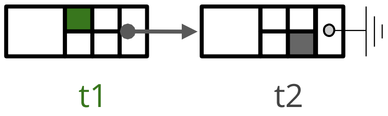
Socket id (e.g, socket 0)

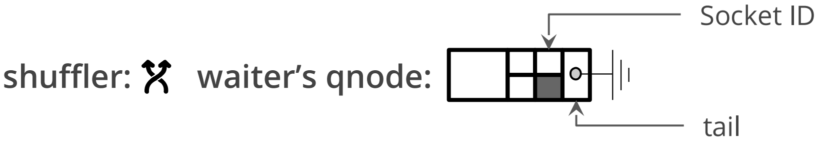


shuffler:  waiter's qnode: 

Shuffling: Design methodology

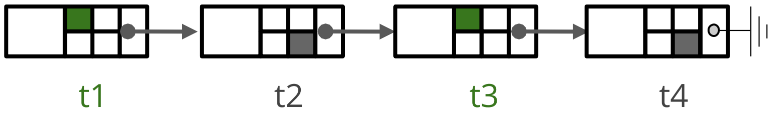
Another waiter is in a different socket



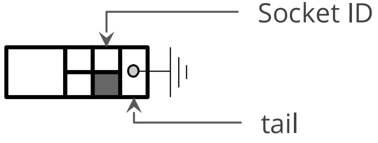


Shuffling: Design methodology

More waiters join

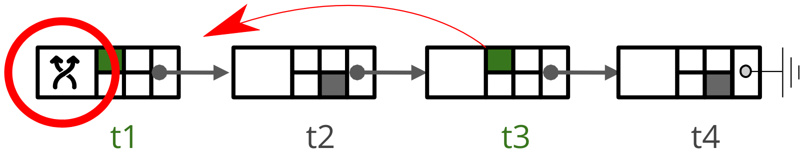


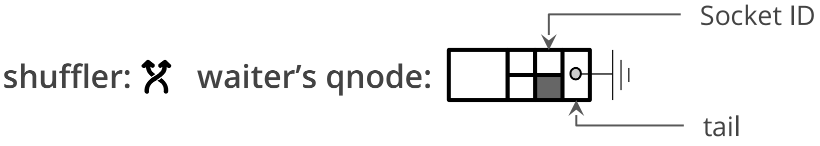
shuffler: waiter's qnode:



Shuffling: Design methodology

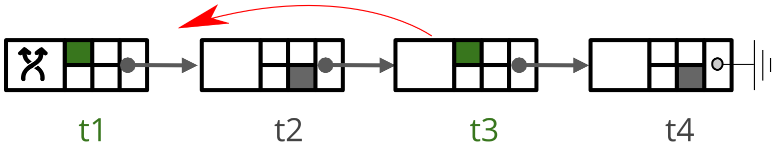
Shuffler (t1) sorts based on socket ID





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** unlike others

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)

- 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)

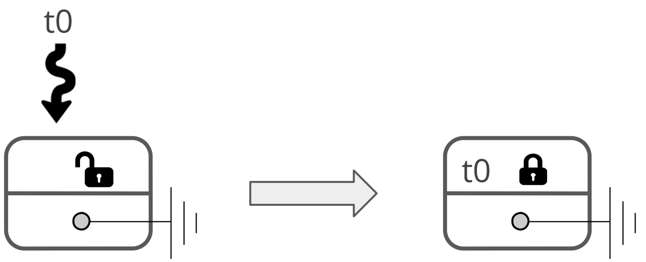
TAS maintains single thread performance

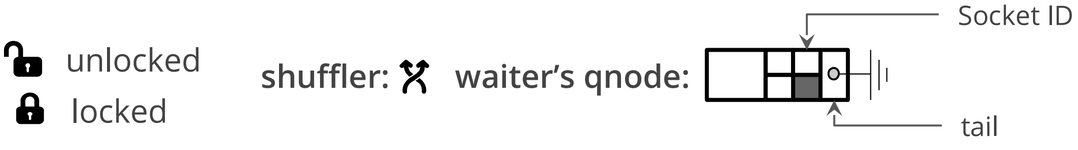
- Waiters use **shuffling** to improve application throughput
  - NUMA-awareness, efficient wake up strategy
  - Utilizing Idle/CPU wasting waiters
- Maintain long-term fairness:
  - Bound the number of shuffling rounds

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

NUMA-aware SHFLLOCK in action

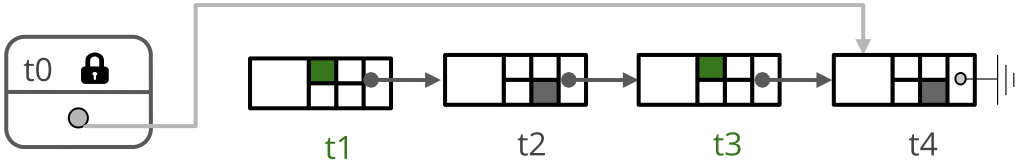
t0 (socket 1): lock()

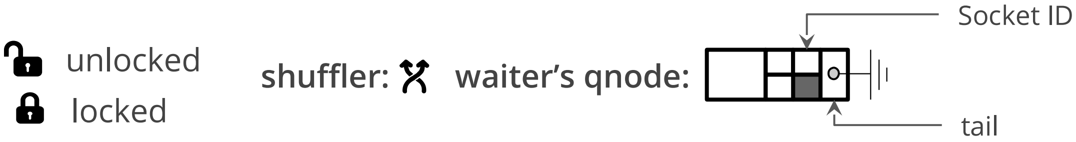




NUMA-aware SHFLLOCK in action

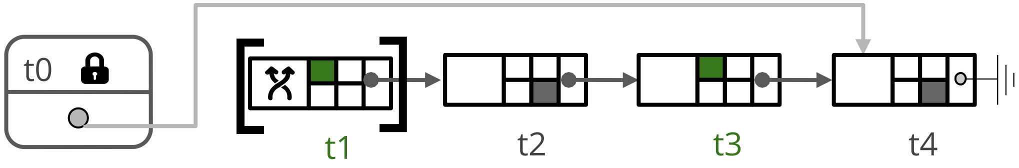
Multiple threads join the queue



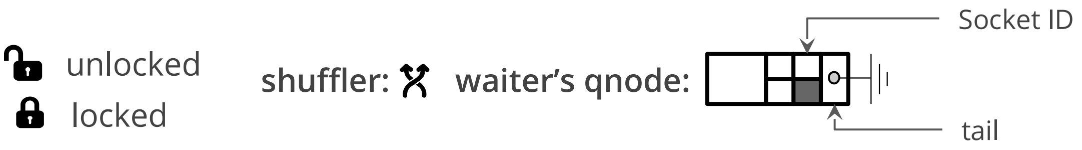


NUMA-aware SHFLLOCK in action

Shuffling in progress



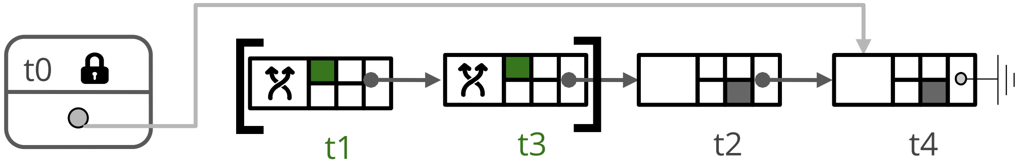
t1 starts the shuffling process

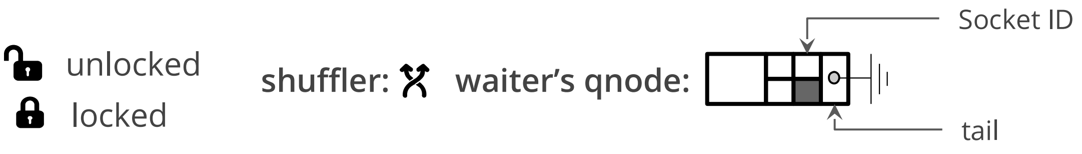


NUMA-aware SHFLLOCK in action

Shuffling in progress

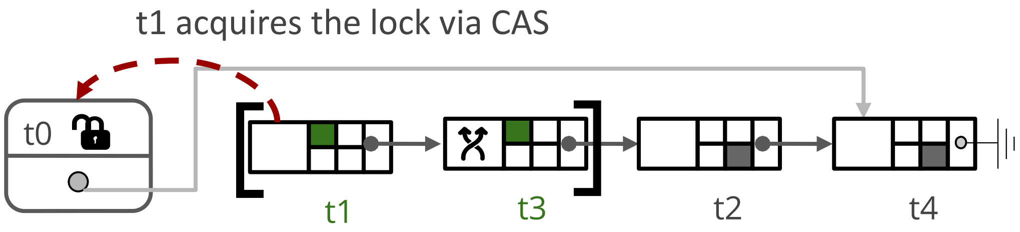
t3 now becomes the shuffler

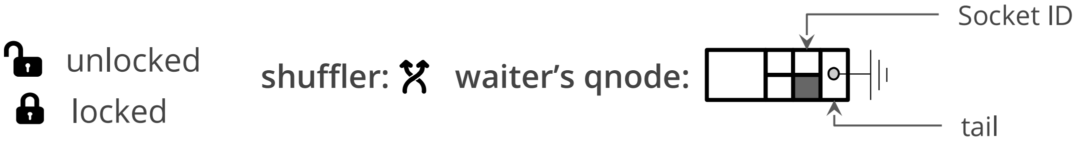




NUMA-aware SHFLLOCK in action

t0: unlock()

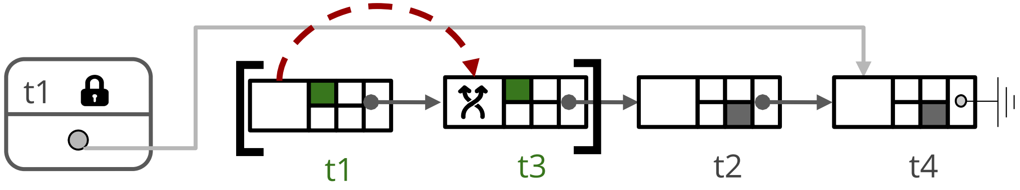


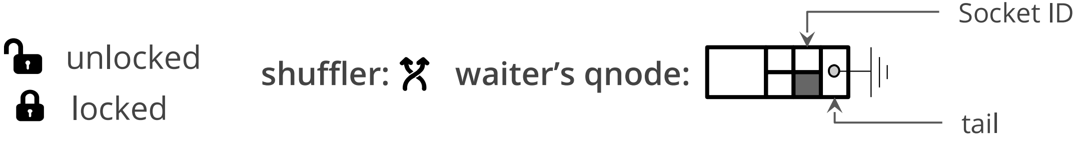


NUMA-aware SHFLLOCK in action

t0: unlock()

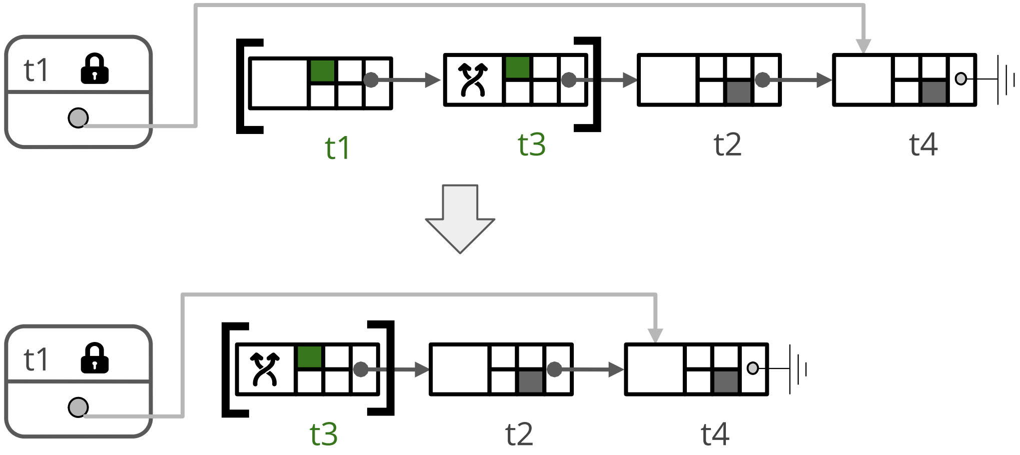
t1 notifies t3 as a new queue head

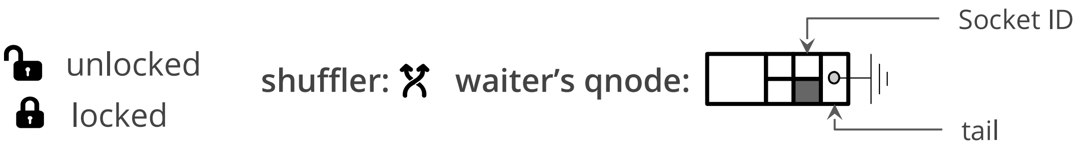




NUMA-aware SHFLLOCK in action

t0: unlock()

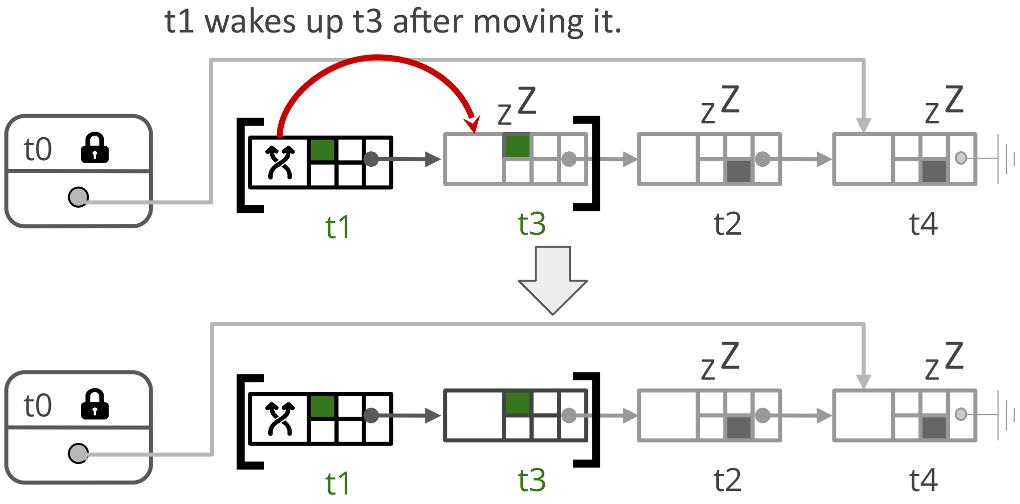




Other SHFLLOCKS: Blocking SHFLLOCK

- NUMA-aware blocking lock.
- Wake up shuffled waiters based on the socket ID.
  - Avoids the wakeup latency from the critical path.
- Lock is always passed to a spinning waiter.
  - Lock stealing: avoid lock-waiter preemption problem.
  - Shuffled waiters are already spinning.
- Guarantees forward progress of the system.

Blocking SHFLLOCK in action



Implementation

- Kernel space:
  - Replaced ***all*** mutex and rwsem
  - Modified slowpath of the qspinlock
- User space:
  - Added to the Litl library
- Please see our paper:
  - **Readers-writer lock**: Centralized rw-indicator + SHFLLOCK

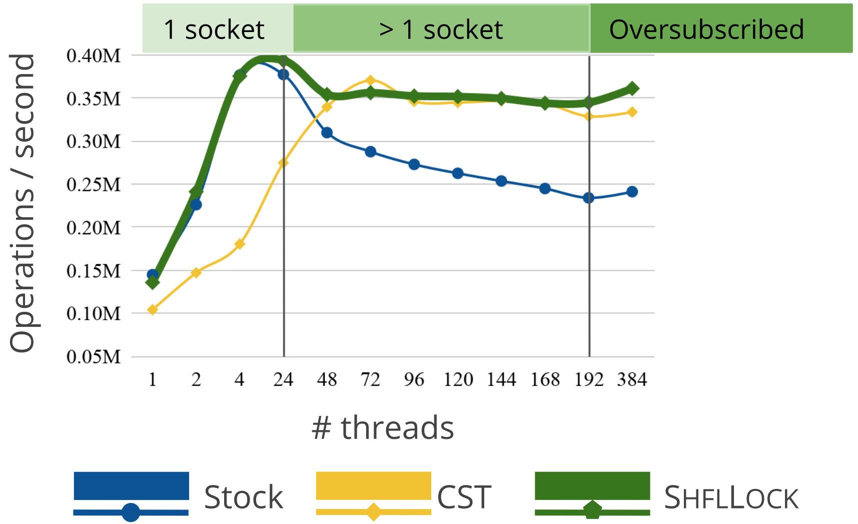
Evaluation

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

Setup: Eight socket 192-core/8-socket machine

Locks performance: Throughput

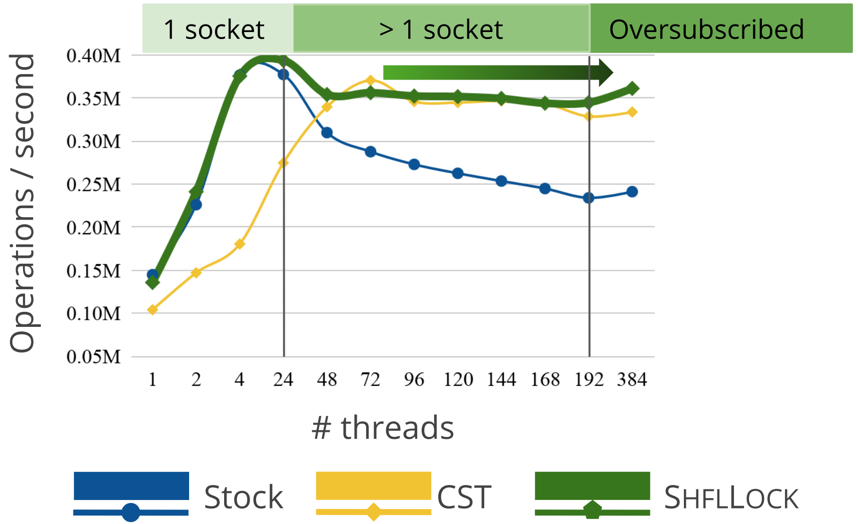
(e.g., each thread creates a file, a serial operation, in a shared directory)



- SHFLLOCKS maintain performance:

Locks performance: Throughput

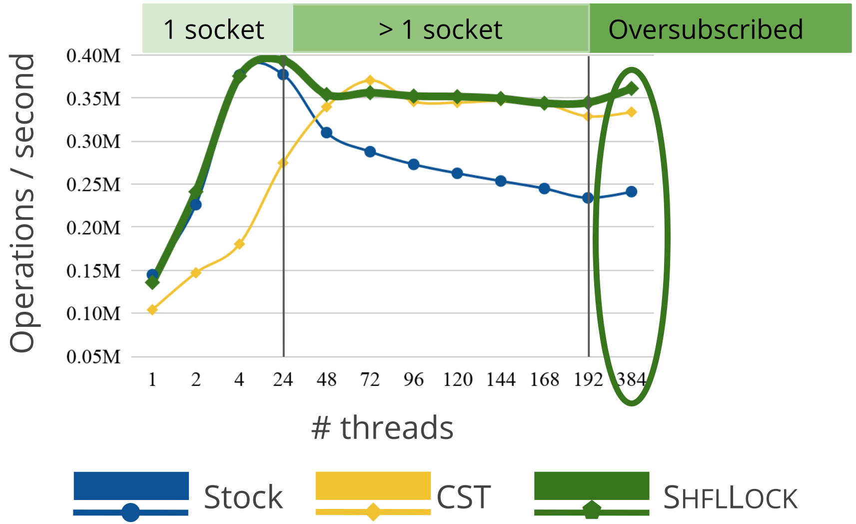
(e.g., each thread creates a file, a serial operation, in a shared directory)



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

Locks performance: Throughput

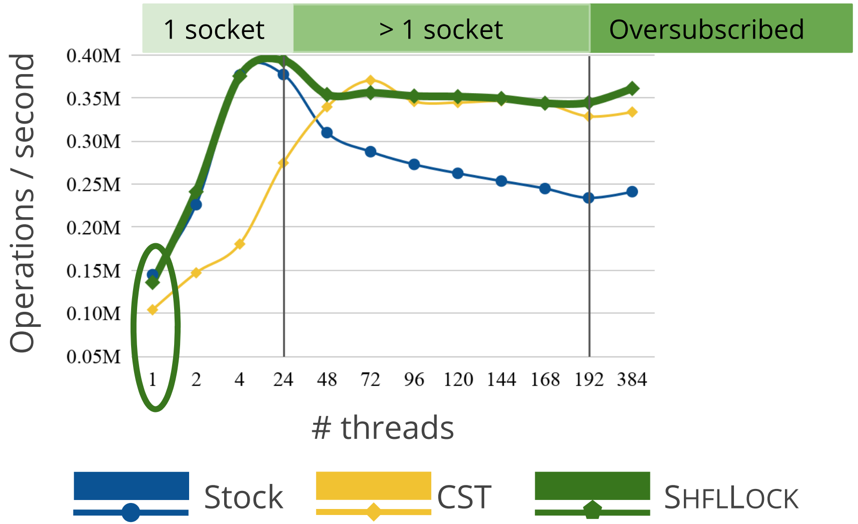
(e.g., each thread creates a file, a serial operation, in a shared directory)



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

Locks performance: Throughput

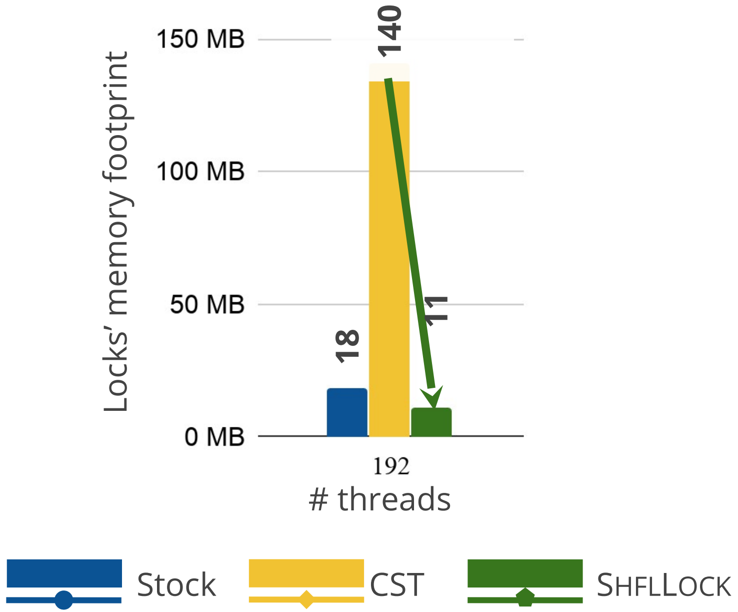
(e.g., each thread creates a file, a serial operation, in a shared directory)



- 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)



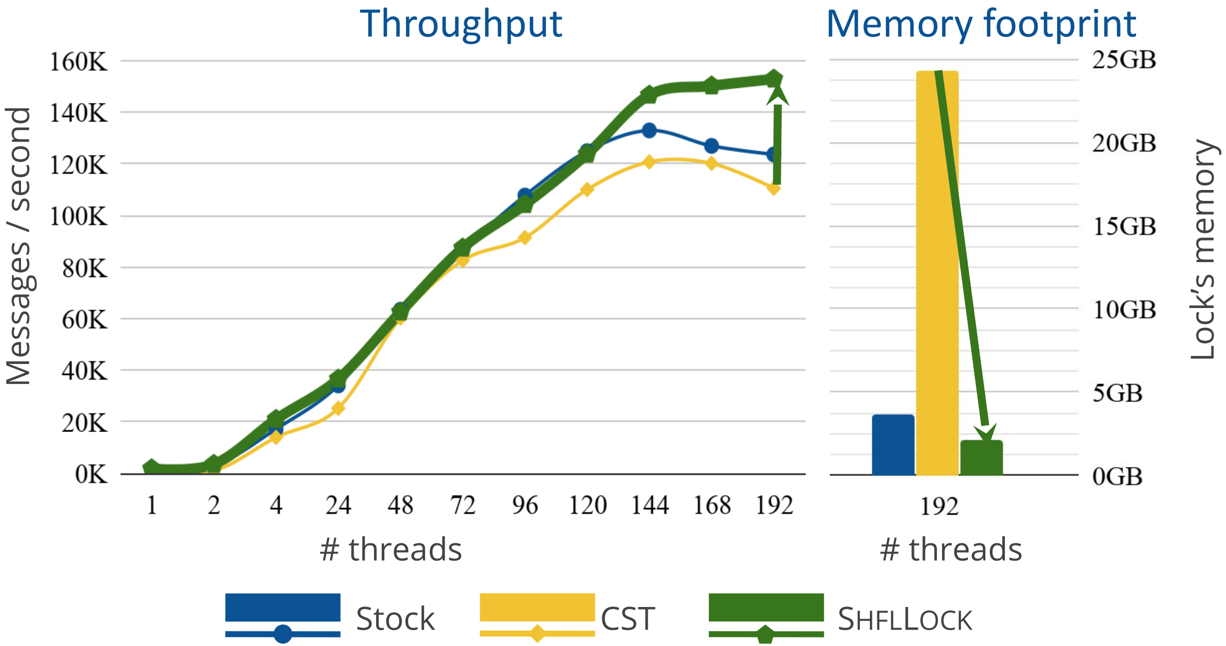
- SHFLLOCKS has least memory footprint

  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%

Discussion

- Another way to enforce these policies dynamically:
  - ***Lock holder*** splits the queue to provide:
    - E.g., NUMA-awareness: Compact NUMA-aware lock (CNA).
    - E.g., blocking lock: Malthusian lock.
- Shuffling can support other policies:
  - Non-inclusive cache (Skylake architecture).
  - Multi-level NUMA hierarchy (SGI machines).

Conclusion

- **Locks are critical for file system and application performance**
- **Current lock designs:**
  - Do not maintain best throughput with varying threads
  - Have high memory footprint
- **Shuffling**: Dynamically enforce policies
  - NUMA, blocking, etc
- **SHFLLOCKS**: Shuffling-based family of lock algorithms
  - NUMA-aware minimal memory footprint locks