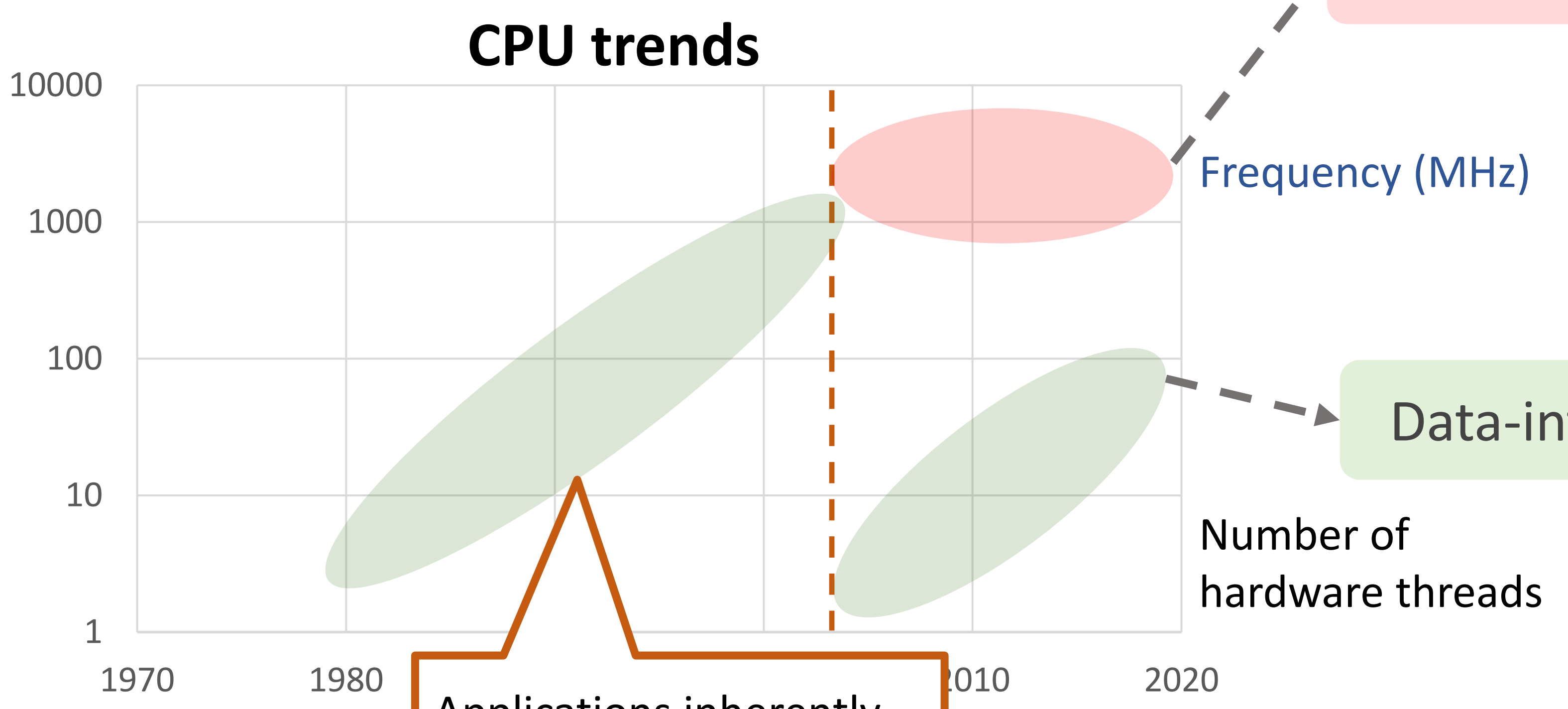
Scaling Synchronization Primitives

Ph.D. Defense of Dissertation

Sanidhya Kashyap

Rise of the multicore machines

Hardware limitation: frequency stagnated



Data-intensive applications performance

Applications inherently scaled with increasing

Machines have multiples of processors (multi-socket)

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten

Multicore machines → "The free lunch is over"

Today's de facto standard:
Concurrent applications that scale with increasing cores

Operating systems

Cloud services

Data processing systems

Databases

Synchronization primitives

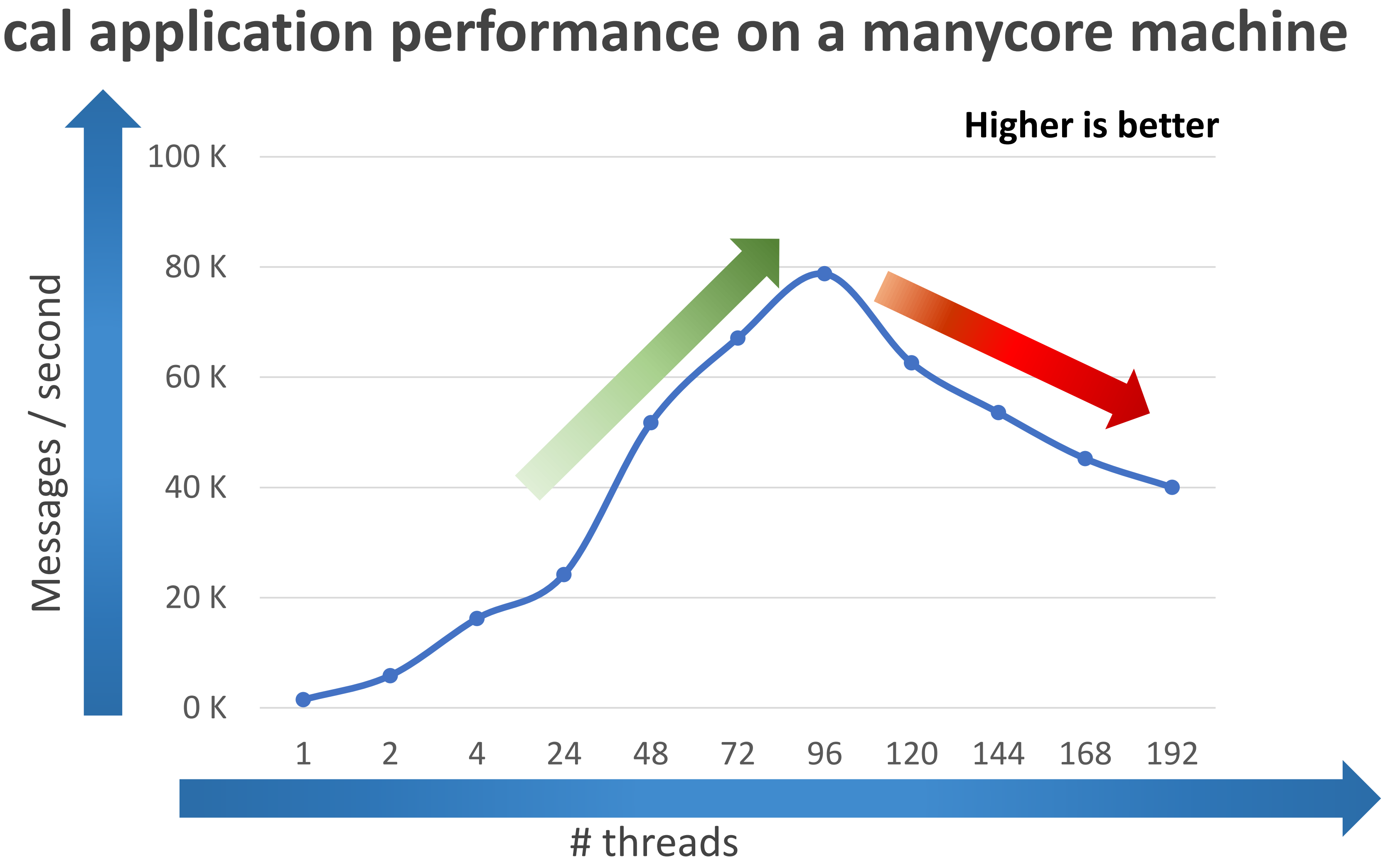
Basic building block for designing applications

Synchronization primitives

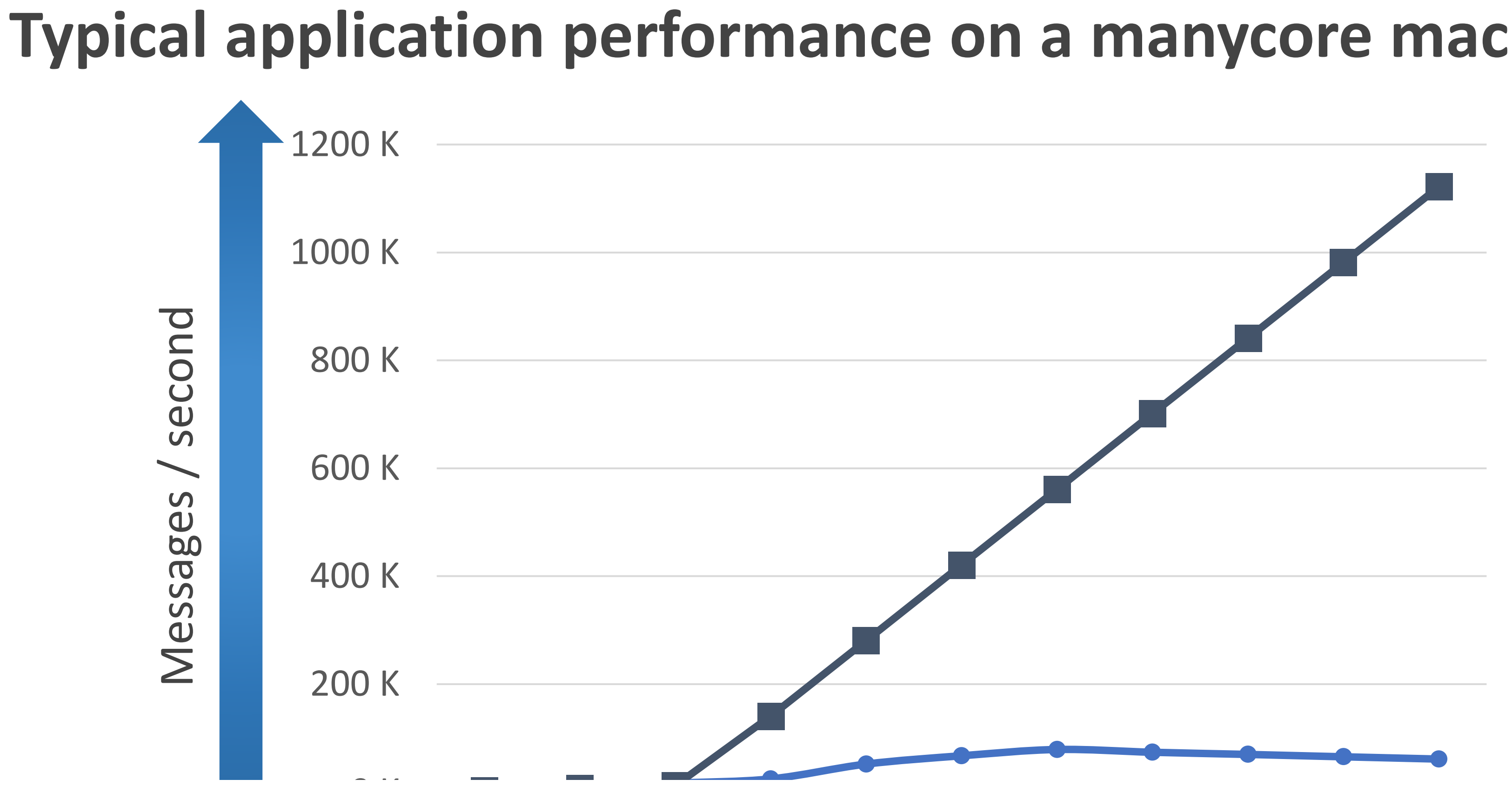
Provide some form of consistency required by applications

Determine the ordering/scheduling of concurrent events

Embarrassingly parallel application performance



Embarrassingly parallel application performance



Synchronization required at several places

Future hardware will exacerbate scalability

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

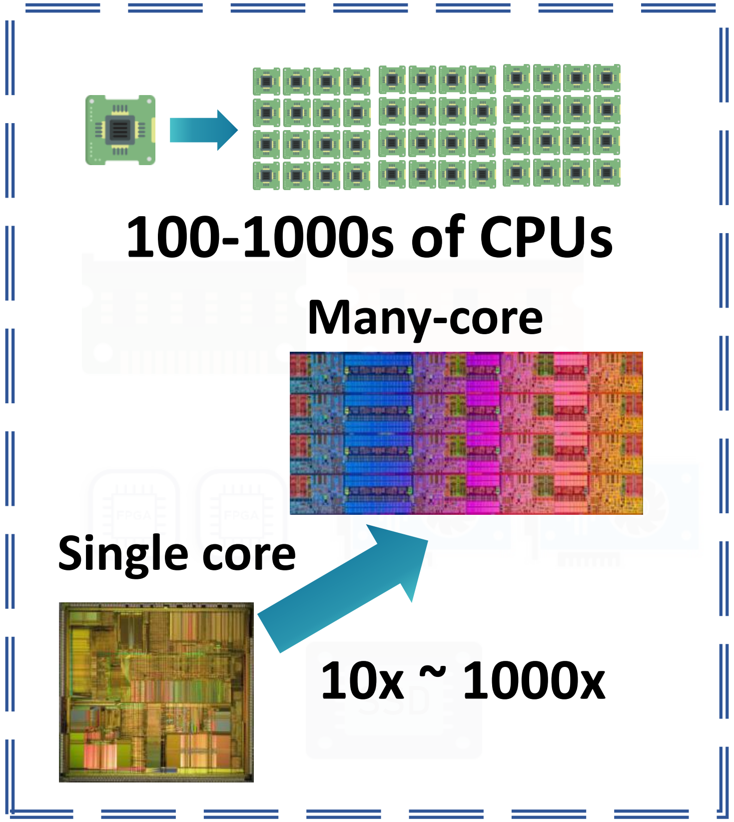
by Dr. Ian Cutress on August 6, 2019 8:01 AM EST

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

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

Arm Announces Neoverse N1 & E1 Platforms & CPUs: Enabling A Huge Jump In Infrastructure Performance

by Andrei Frumusanu on February 20, 2019 9:00 AM EST

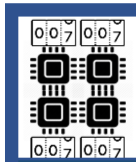


Challenge: Maintain application scalability

How can we minimize the overhead of synchronization primitives for large multicore machines?

Efficiently schedule events by leveraging HW/SW

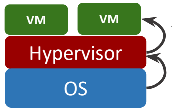
Thesis contributions



Hardware timestamping

Eurosys’18

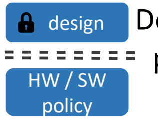
- Timestamping is costly on large multicore machines
- Cache contention due to atomic instructions
- **Approach**: Use per-core invariant hardware clock



VM double scheduling

ATC’18

- Double scheduling in a virtualized environment
- Introduce various types of preemption problems
- **Approach**: Expose semantic information across layers



Decouple lock policy from design

SOSP’19

- Discrepancy between lock design and use
- **Approach**: Decouple lock design from lock policy via shuffling mechanism

Thesis contributions

Hardware timestamping

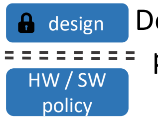
Eurosys'18

- Timestamping is costly on large multicore machines
- Cache contention due to atomic instructions
- **Approach**: Use per-core invariant hardware clock

VM double scheduling

ATC'18

- Double scheduling in a virtualized environment
- Introduce various types of preemption problems
- **Approach**: Expose semantic information across layers



Decouple lock policy from design

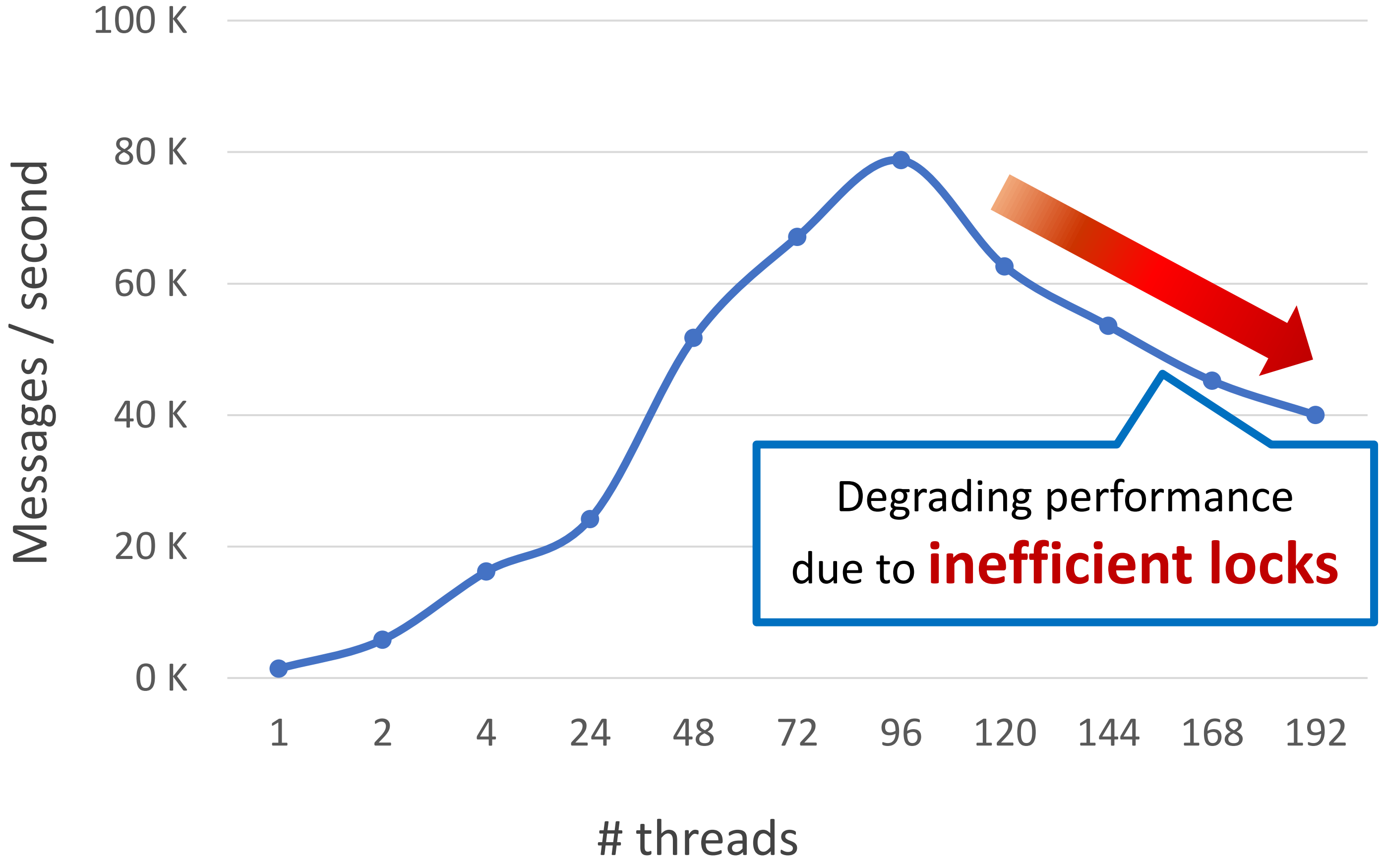
SOSP'19

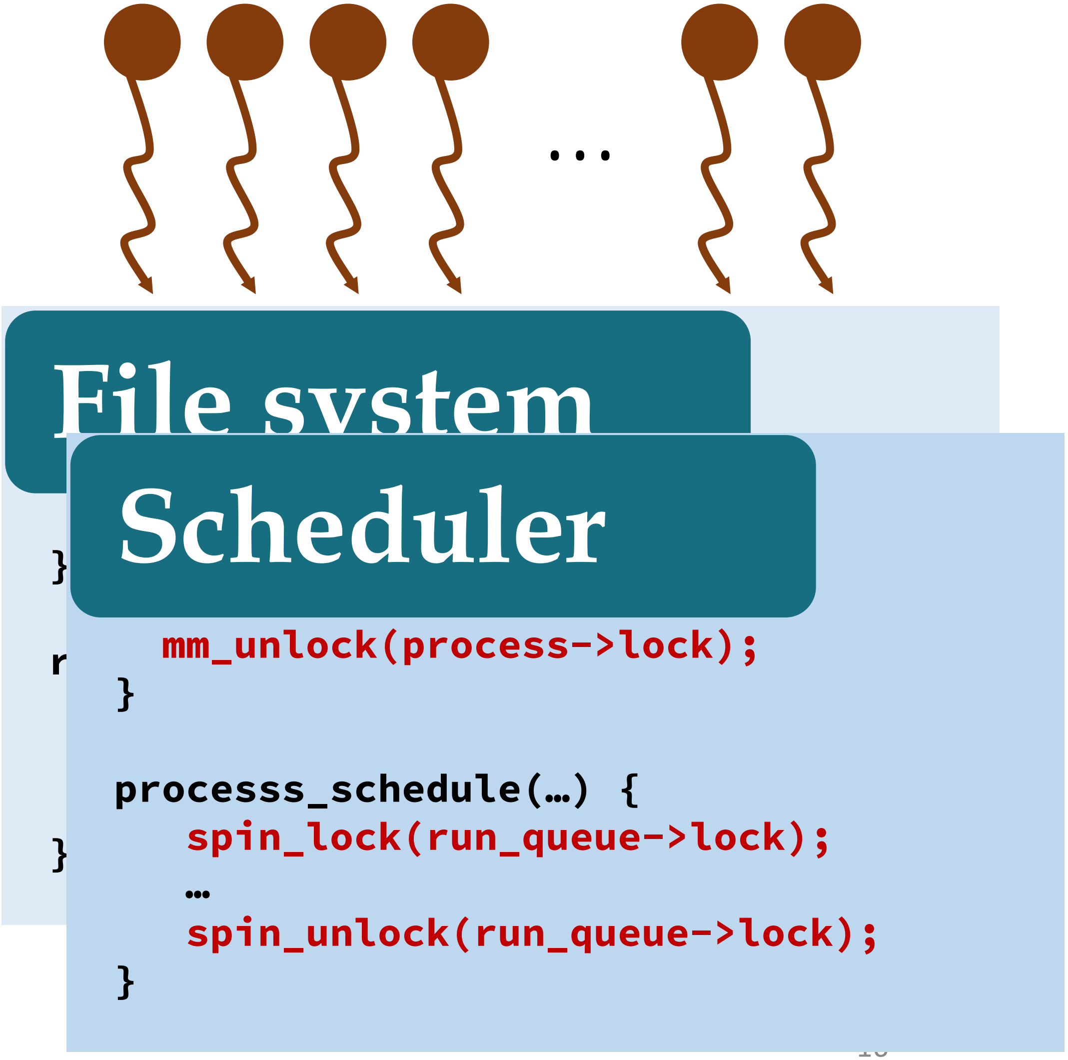
- Discrepancy between lock design and use
- **Approach**: Decouple lock design from lock policy via shuffling mechanism

Example: Email service

Embarrassingly parallel application performance

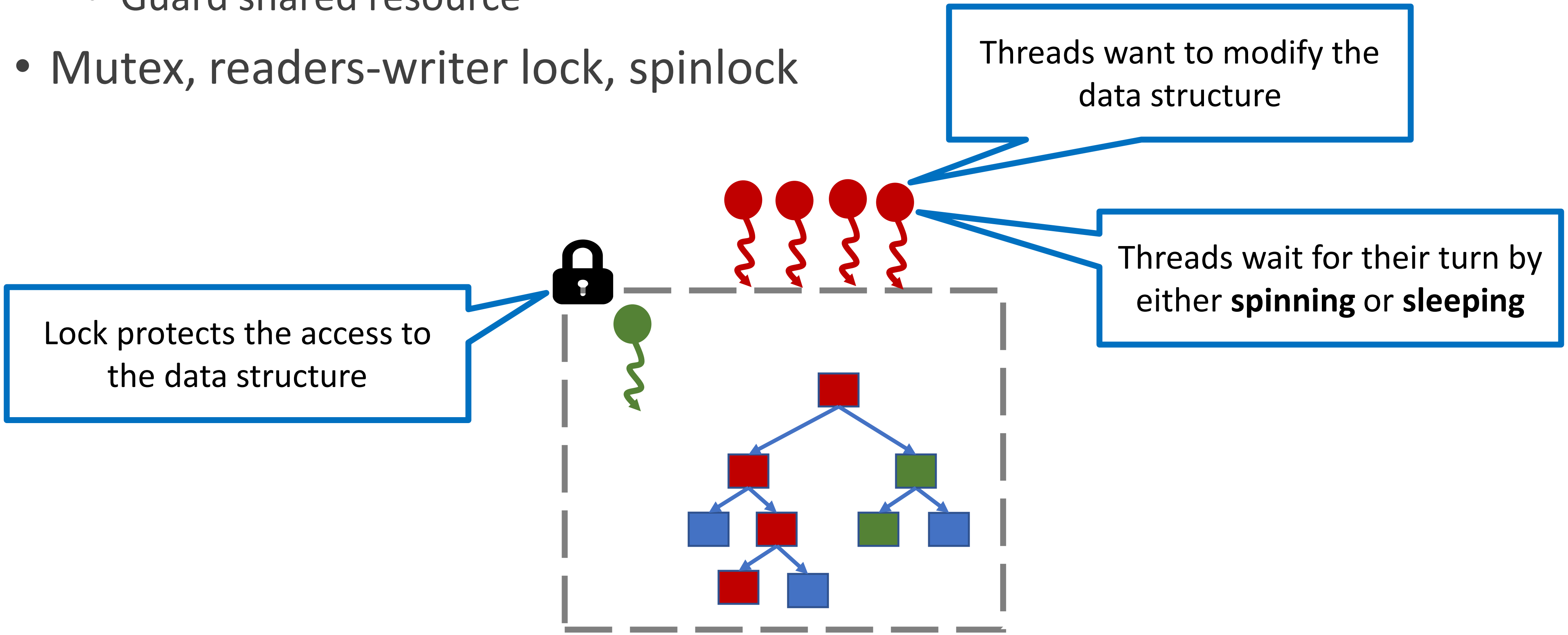
Process intensive and stresses memory subsystem, file system and scheduler



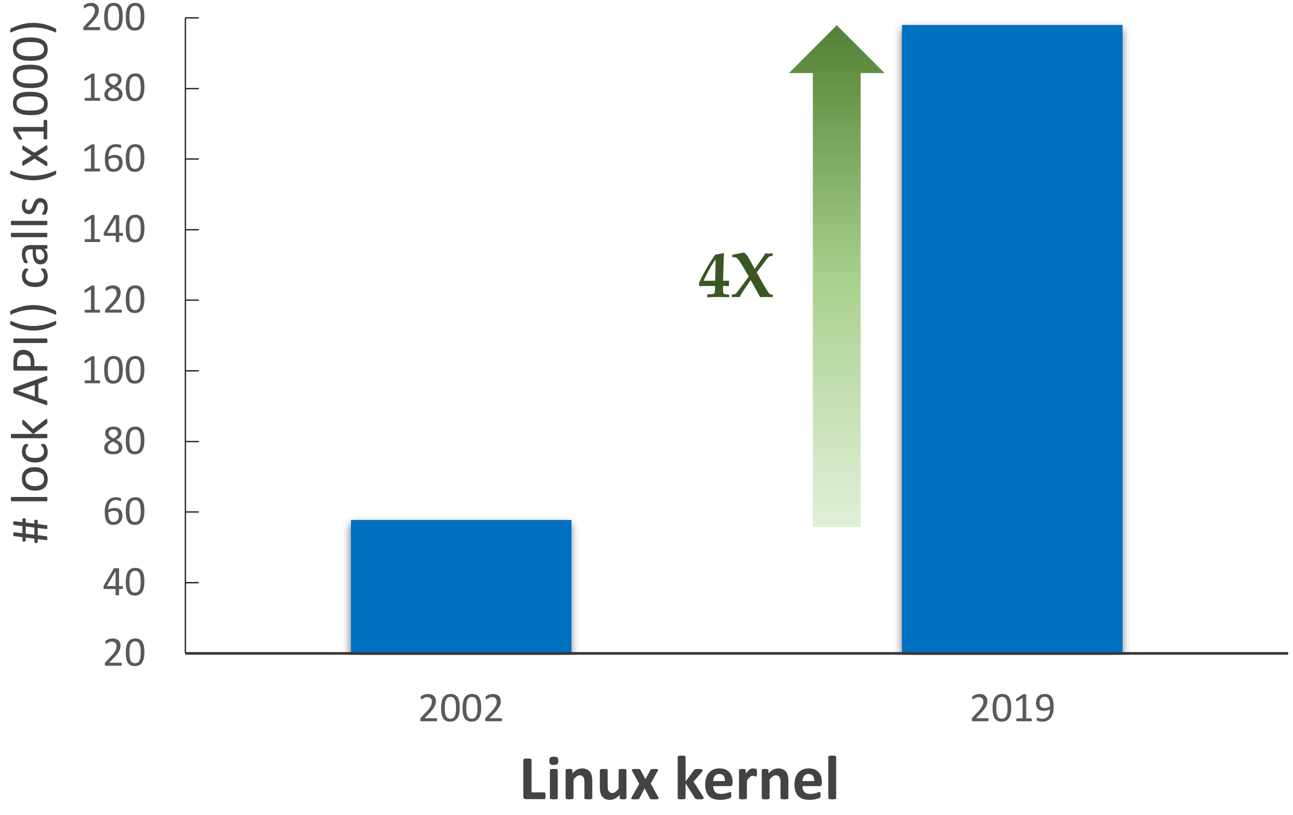


Synchronization primitive: Locks

- Provide mutual exclusion among tasks
  - Guard shared resource
- Mutex, readers-writer lock, spinlock

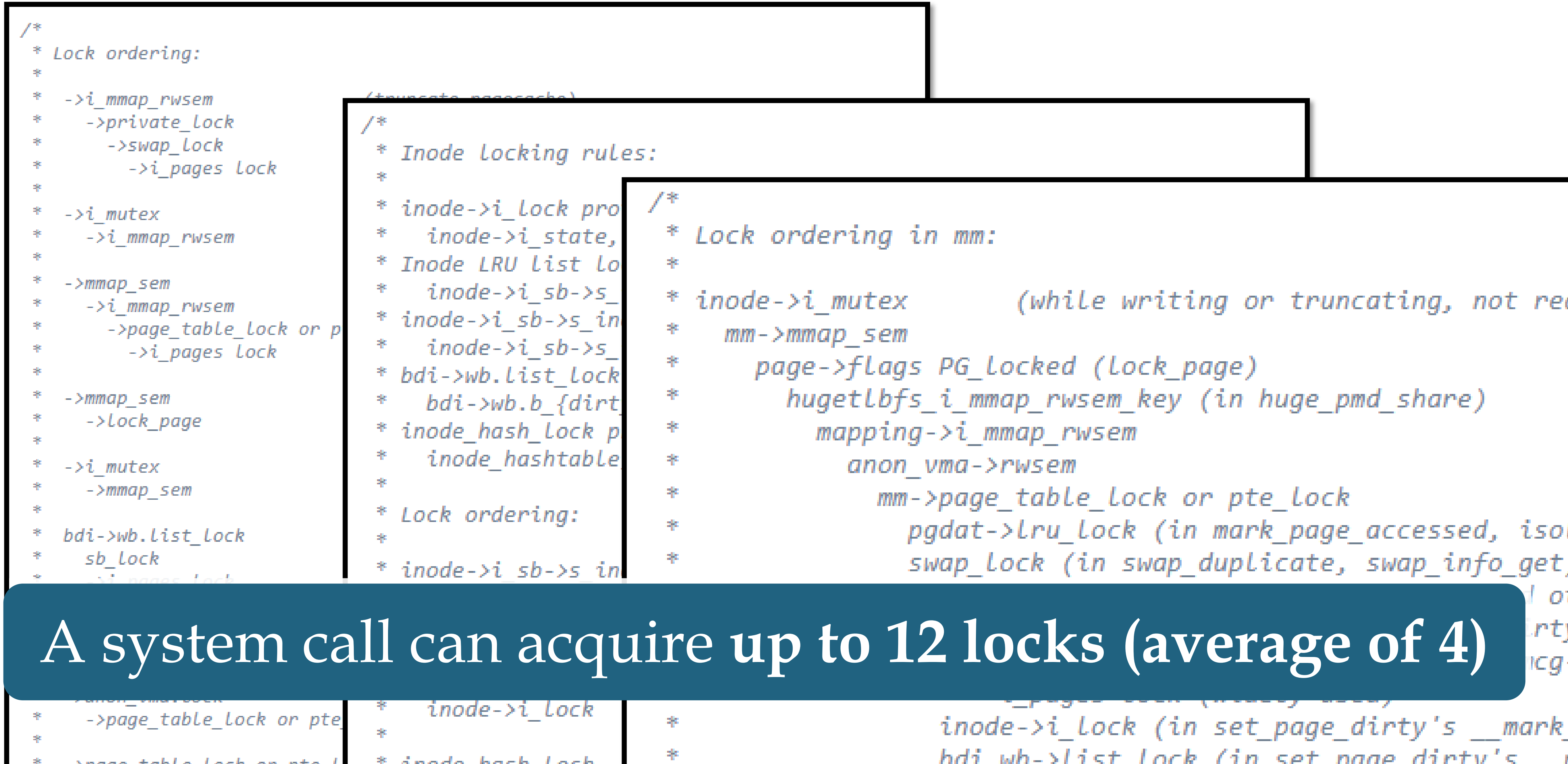


Locks: MOST WIDELY used primitive



More locks are in use to improve OS scalability

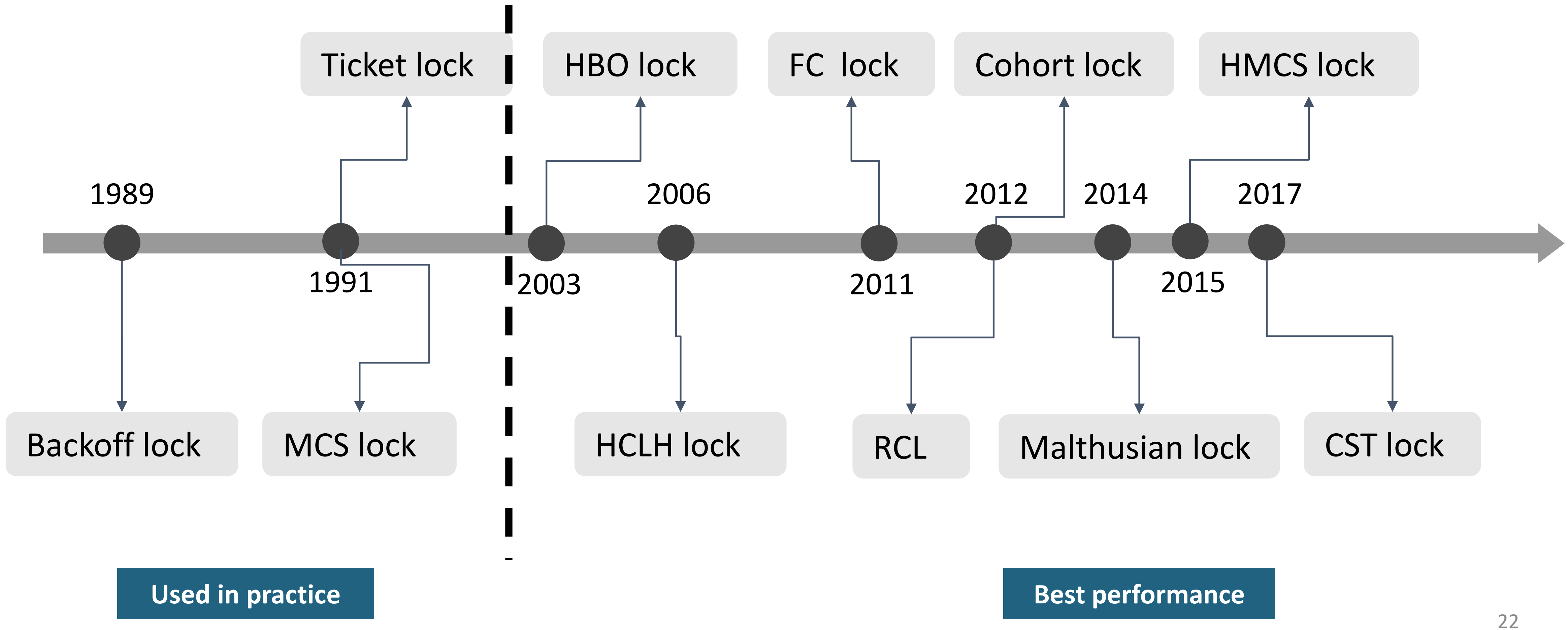
Locks are used in a complicated manner



A system call can acquire **up to 12 locks (average of 4)**

Issue with current lock designs

Design specific locks for hardware and software requirements



Issue with current lock designs

Design specific locks for hardware and software requirements

Locks in practice:

- Generic ⇨ Focus on simple lock, more applicability
- Forgo hardware characteristic ⇨ Worsening throughput with more cores

Locks in research:

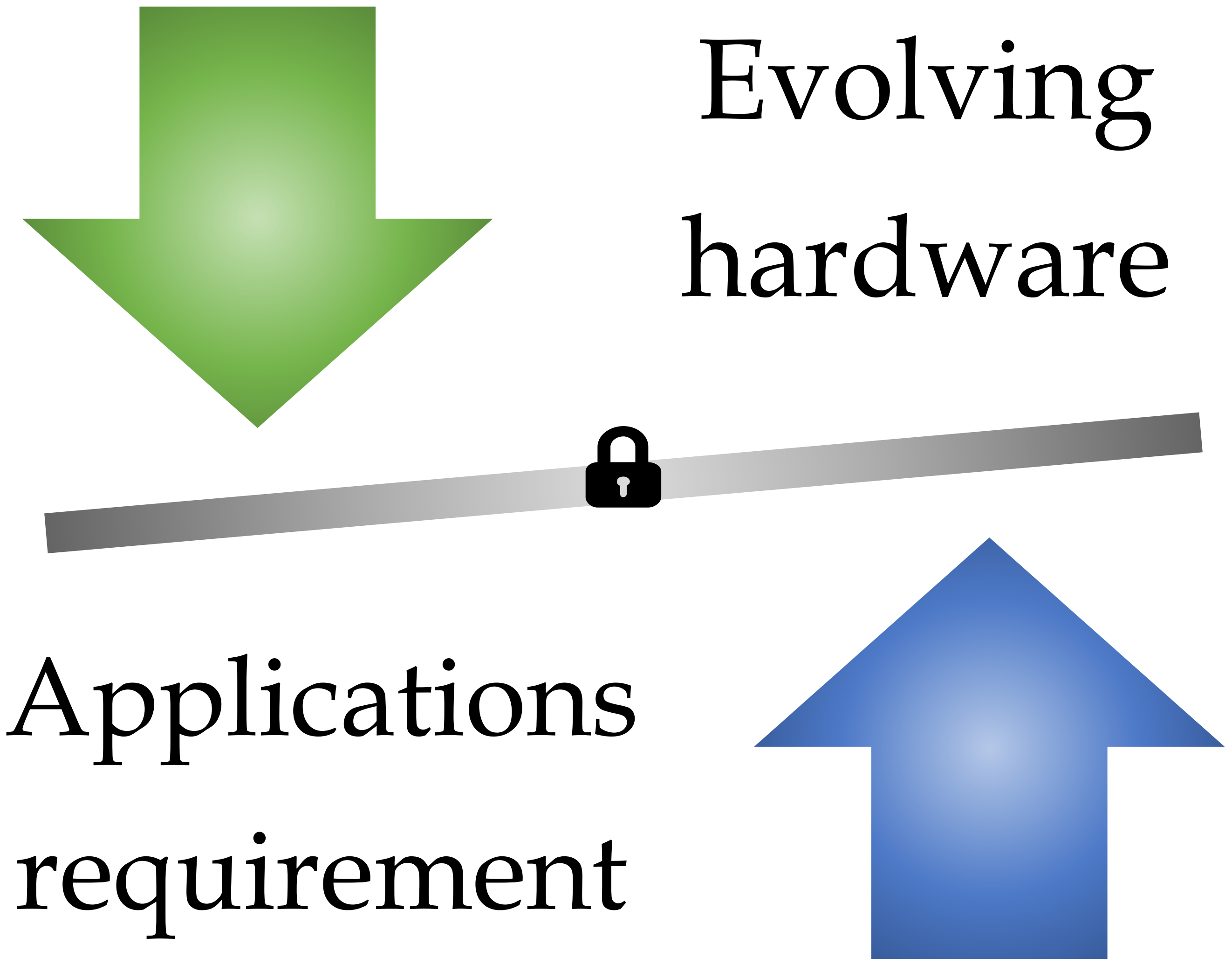
- Hardware specific design ⇨ High throughput for high thread count

HW/SW policies are statically tied together

Incorporating HW/SW policies dynamically

Scalable and practical locking algorithms

Two trends driving locks' innovation



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

Two dimensions of lock design / goals

1) High throughput

- In high thread count ⇒ Minimize lock contentions

Application: Multi-threaded to utilize cores to improve performance

Lock: Minimize lock contention while maintaining high throughput

2) Minimal lock size

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

Two dimensions of lock design / goals

1) High throughput

- In high thread count ⇨ Minimize lock contentions
- In single thread ⇨ No penalty when not contended

Application: Single thread to do an operation; fine-grained locking

Lock: Minimal or almost no lock/unlock overhead

- Memory footprint ⇨ Scales to millions of locks

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

Application: More threads than cores; common scenario; eg. I/O wait

Lock: Minimize scheduler overhead while waking or parking threads

Two dimensions of lock design / goals

1) High throughput

- In high thread count ⇨ Minimize lock contentions

Application: Locks are embedded in data structures; eg. file inodes

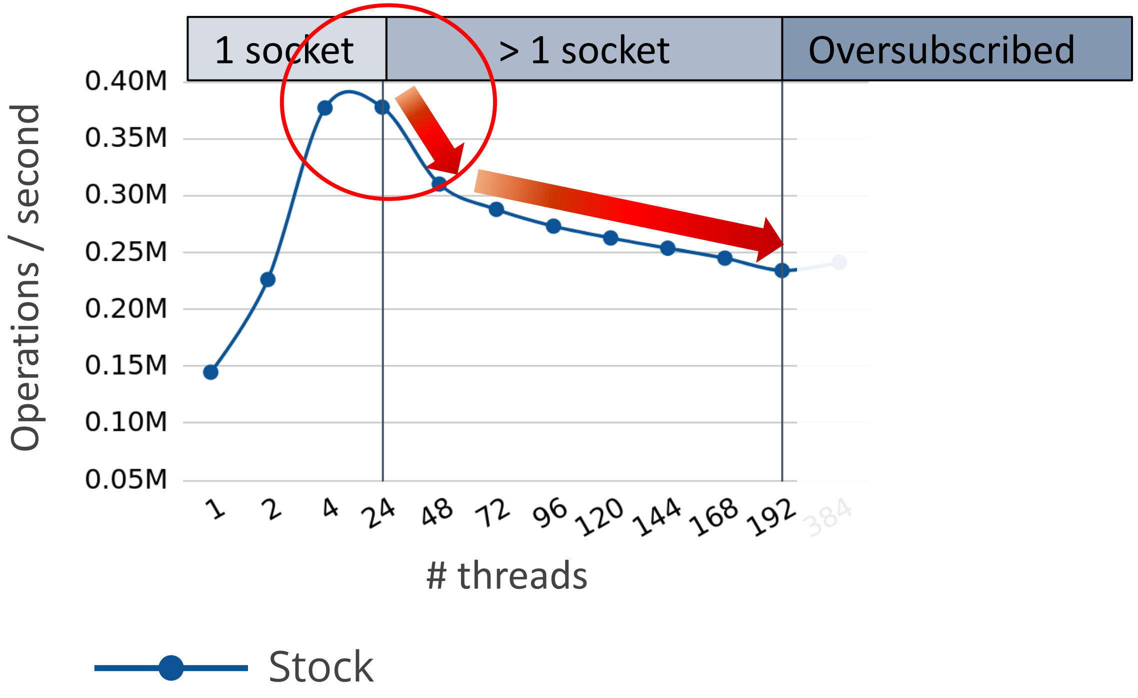
Lock: Can stress memory allocator or data structure alignment

2) Minimal lock size

- Memory footprint ⇨ Scales to millions of locks

Locks performance: Throughput

Benchmark: Each thread creates a file, a serial operation, in a shared directory[1]



- **Throughput collapses after one socket**

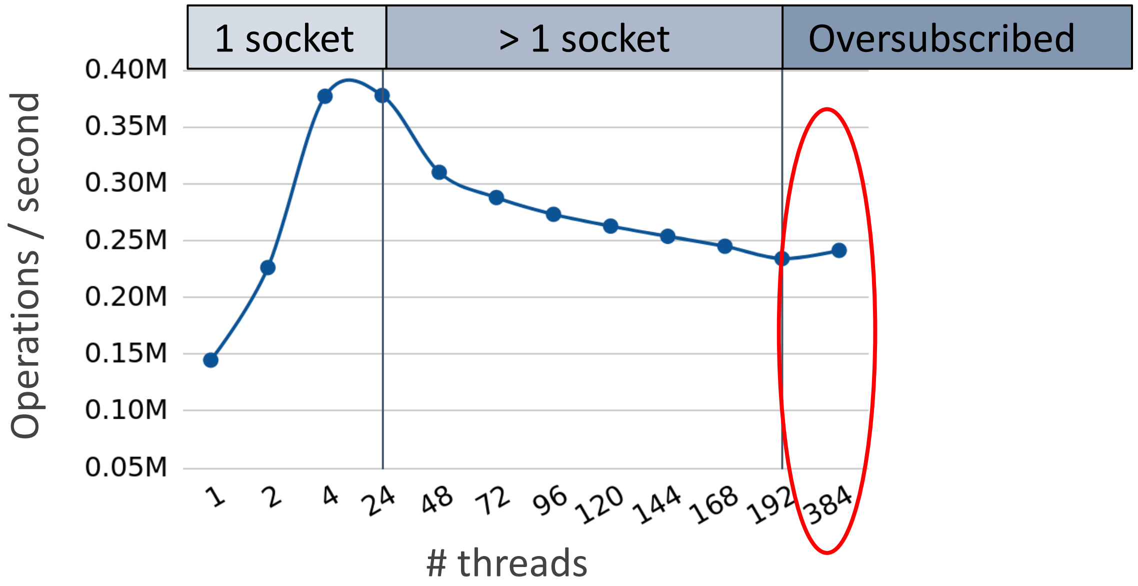
Due to **non-uniform memory access** (NUMA)

Setup: 192-core/8-socket machine

1. Understanding Manycore Scalability of File Systems [ATC'16]

Locks performance: Throughput

Benchmark: Each thread creates a file, a serial operation, in a shared directory[1]



- **Throughput collapses after one socket**

  Due to non-uniform memory access (NUMA)

- **NUMA also affects oversubscription**

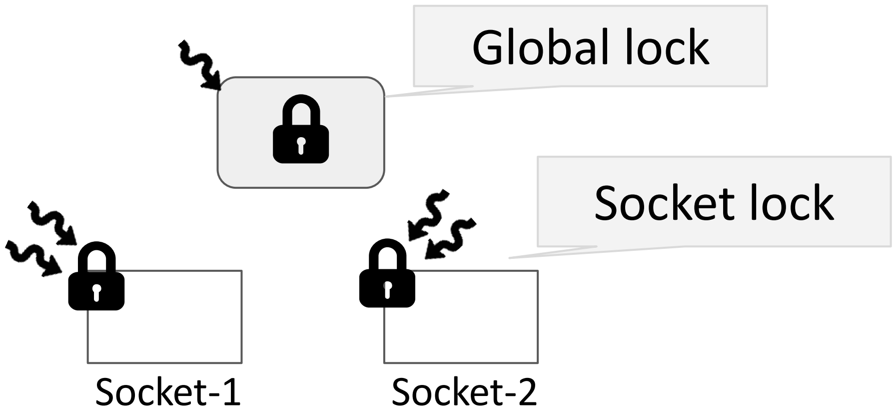
Prevent throughput collapse after **one socket**

1. Understanding Manycore Scalability of File Systems [ATC'16]

Existing research efforts: Hierarchical locks

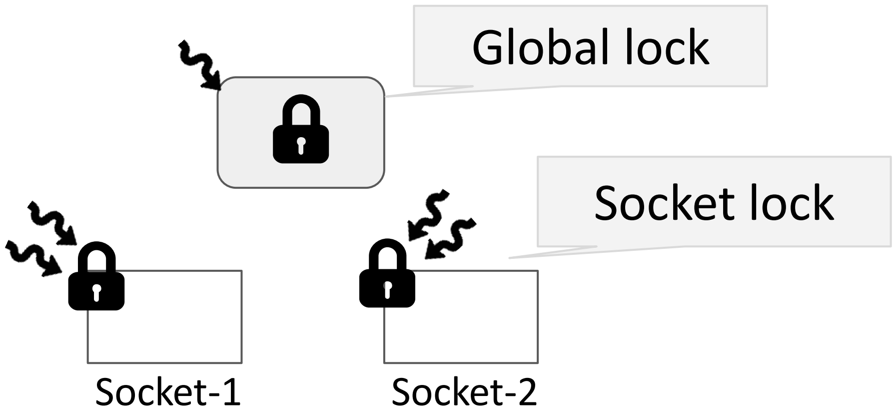
- Goal: high throughput at high thread count
- Making locks NUMA-aware:
  - **Use extra memory to improve throughput**
  - Two level locks: per-socket and the global
- Avoid NUMA overhead

→ Pass global lock within the same socket



Existing research efforts: Hierarchical locks

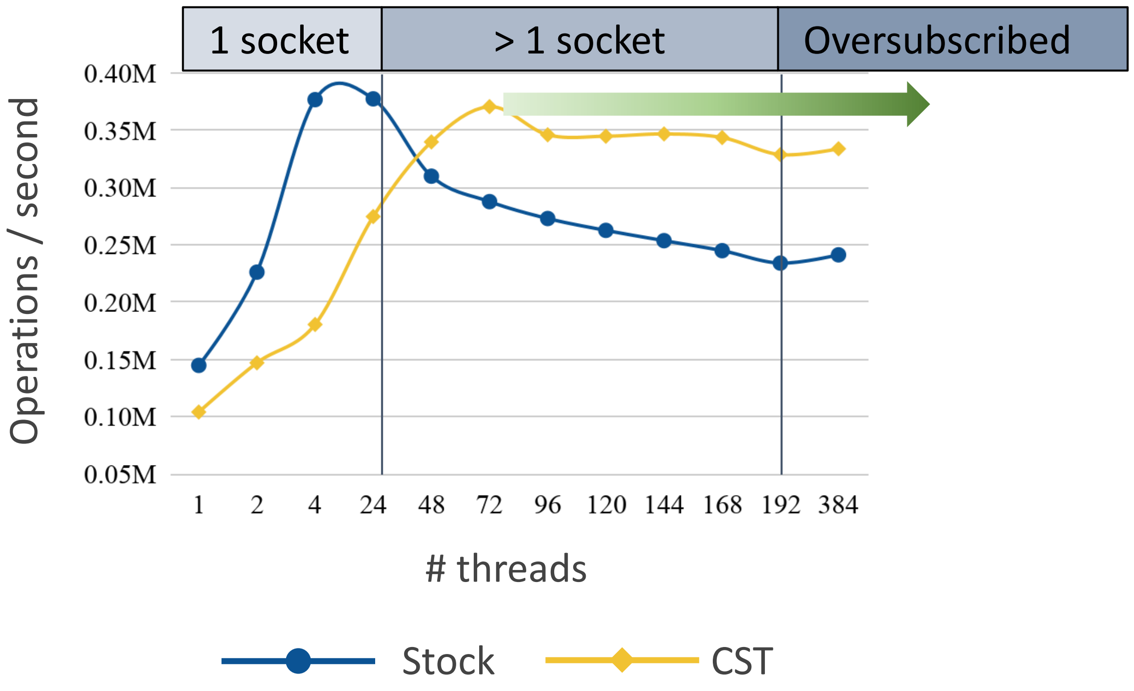
- Problems:
  - **Require extra memory allocation**
  - **Do not care about single thread throughput**
- Example: CST[2]
  - Allocates socket structure on first access
  - Handles oversubscription (# threads > # CPUS)



2. Scalable NUMA-aware Blocking Synchronization Primitives [ATC'17]

Locks performance: Throughput

Benchmark: Each thread creates a file, a serial operation, in a shared directory

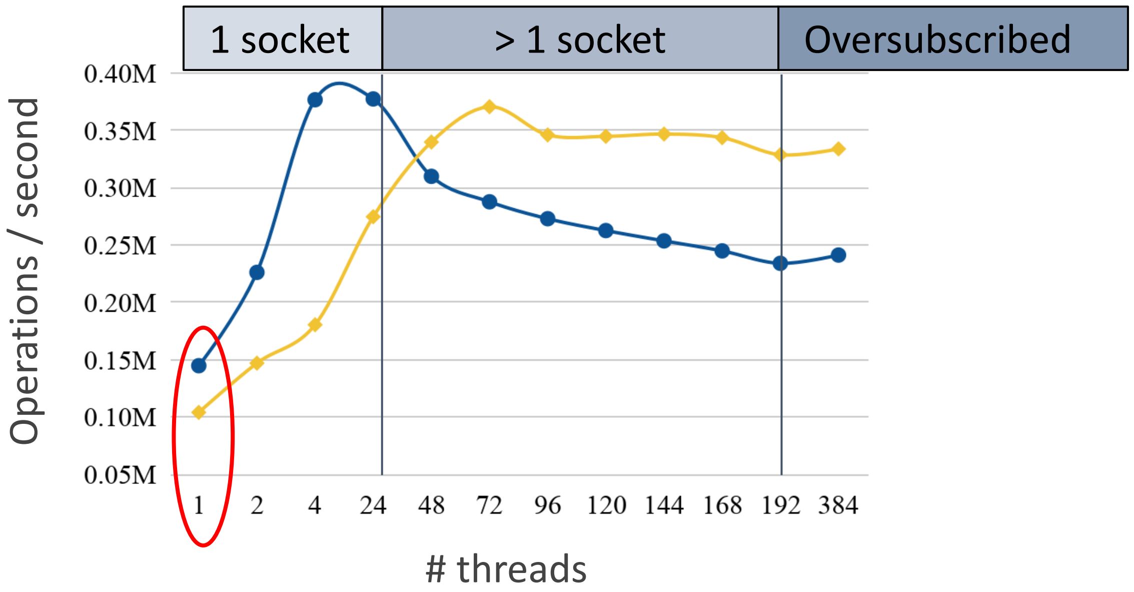


- **Maintains throughput:**
  Beyond one socket (high thread count)
  In oversubscribed case (384 threads)
- **Poor single thread throughput**
  Multiple atomic instructions

Setup: 192-core/8-socket machine

Locks performance: Throughput

Benchmark: Each thread creates a file, a serial operation, in a shared directory



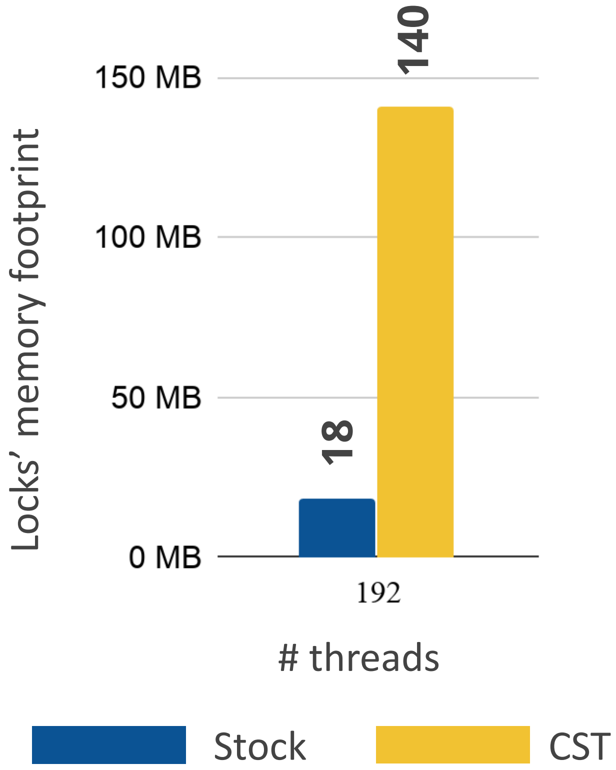
- **Maintains throughput:**
  Beyond one socket (high thread count)
  In oversubscribed case (384 threads)

- **Poor single thread throughput**
  Multiple atomic instructions

Non-contended case → single thread matters

Locks performance: Memory footprint

Benchmark: Each thread creates a file, a serial operation, in a shared directory



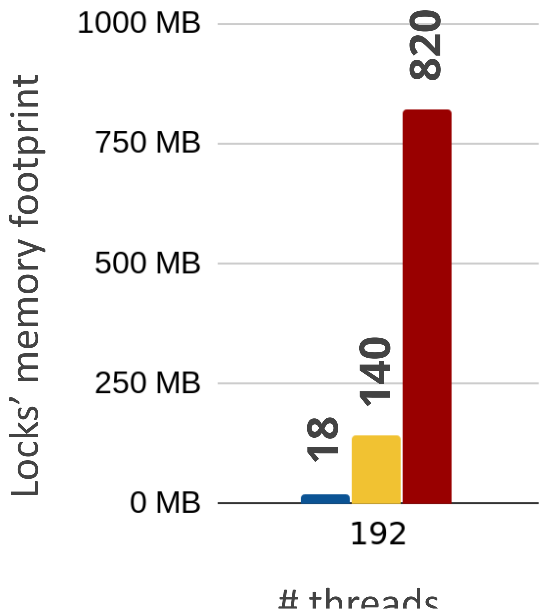
- **CST has large memory footprint**

Allocate socket structure and global lock

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

Locks performance: Memory footprint

Benchmark: 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

  → All per-socket locks are pre-allocated for the hierarchical lock

Lock's memory footprint affects its adoption

Two goals in our new lock design

1) **NUMA-aware** lock **without extra memory**

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

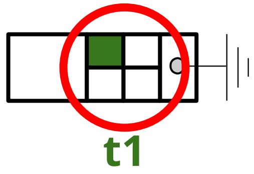
→ Queue-based locks already maintain a list of waiters

Sort waiters on the fly using socket ID

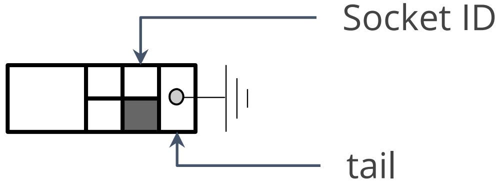
A waiting queue

Socket ID (e.g, socket 0)

Head →

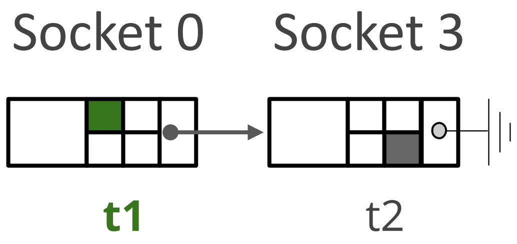


shuffler: waiter's qnode:

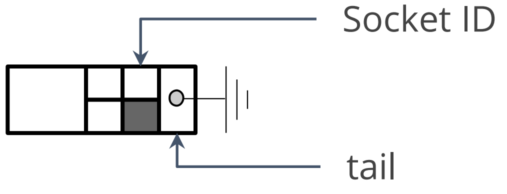


Sort waiters on the fly using socket ID

Another waiter is in a different socket

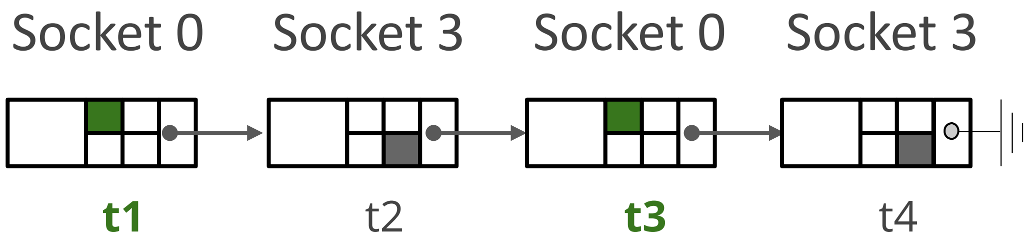


shuffler: waiter's qnode:



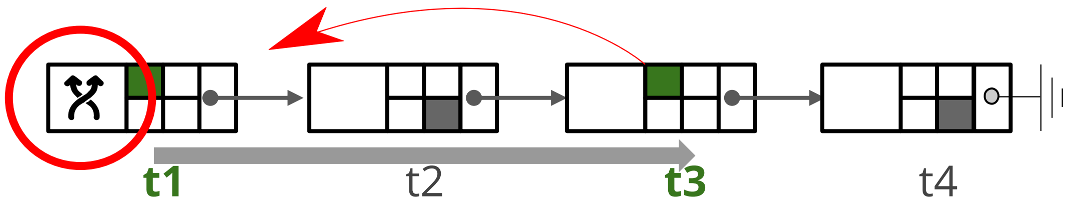
Sort waiters on the fly using socket ID

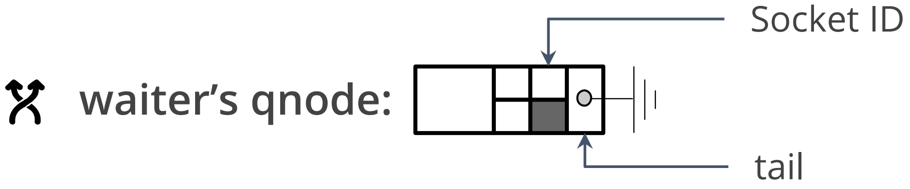
More waiters join



Sort waiters on the fly using socket ID

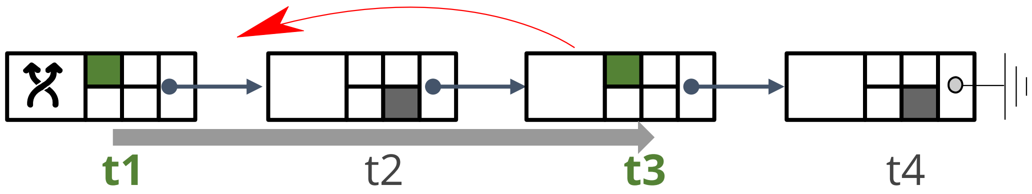
Shuffler (t1) sorts based on socket ID



shuffler: 

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 any additional memory**

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
- ★ **Shuffling is off the critical path most of the time**
- Maintain long-term fairness:
  - Bound the number of shuffling rounds

SHFLLOCKS: Family of lock algorithms

- NUMA-aware spinlock
- NUMA-aware blocking lock
- NUMA-aware writer preferred readers-writer lock

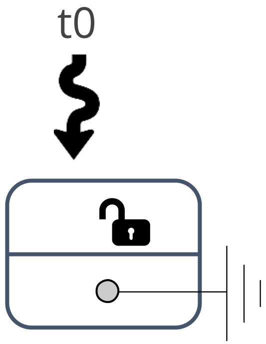
NUMA-aware SHFLLOCK in action

t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()

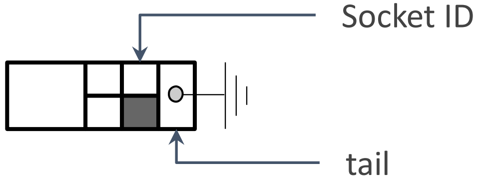


unlocked

locked

shuffler:

waiter's qnode:



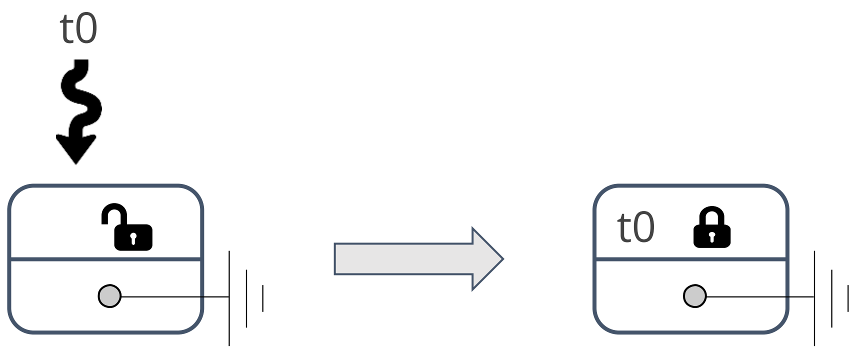
NUMA-aware SHFLLOCK in action

t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()

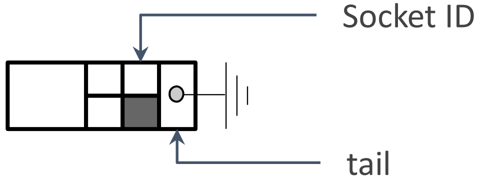


unlocked

locked

shuffler:

waiter's qnode:



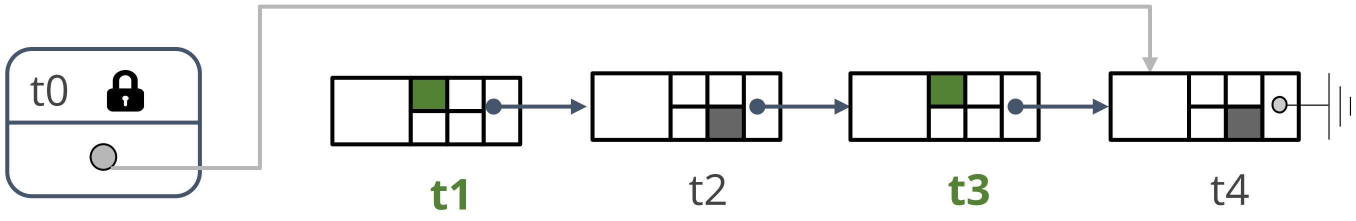
NUMA-aware SHFLLOCK in action

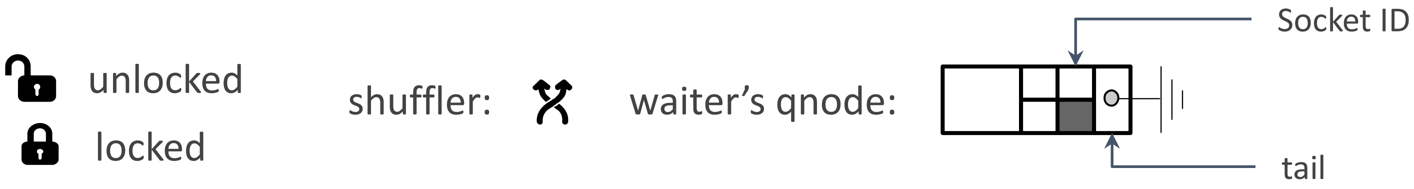
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





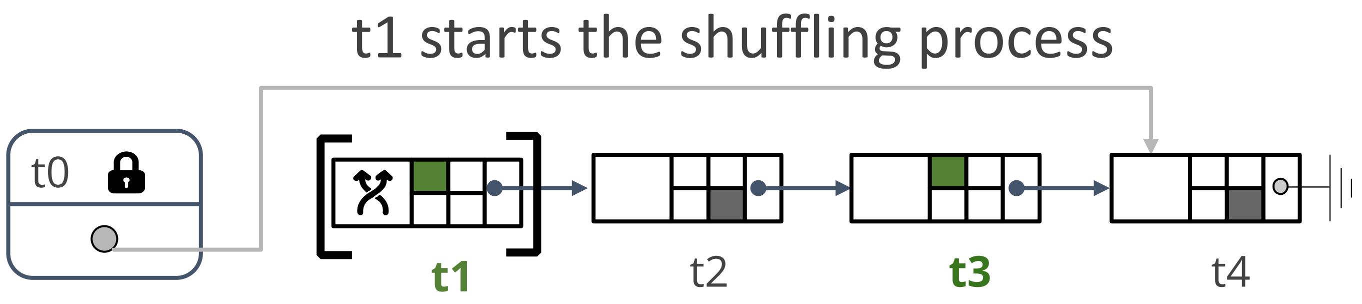
NUMA-aware SHFLLOCK in action

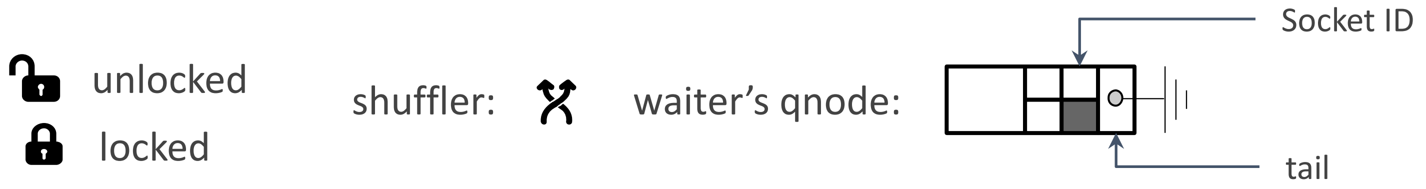
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





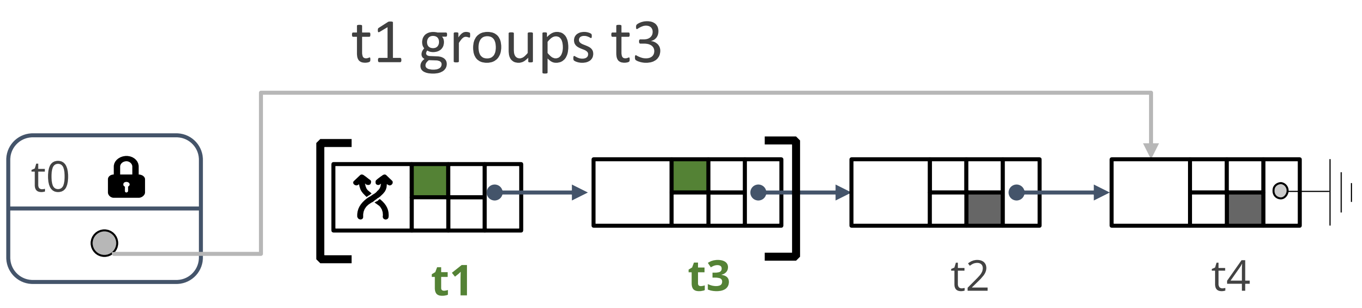
NUMA-aware SHFLLOCK in action

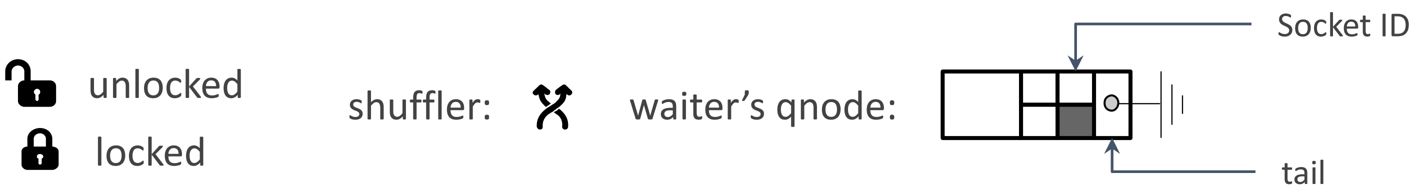
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





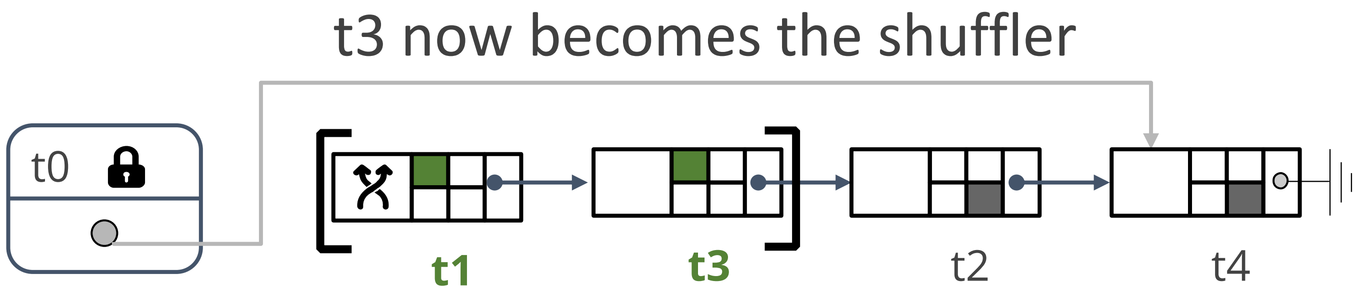
NUMA-aware SHFLLOCK in action

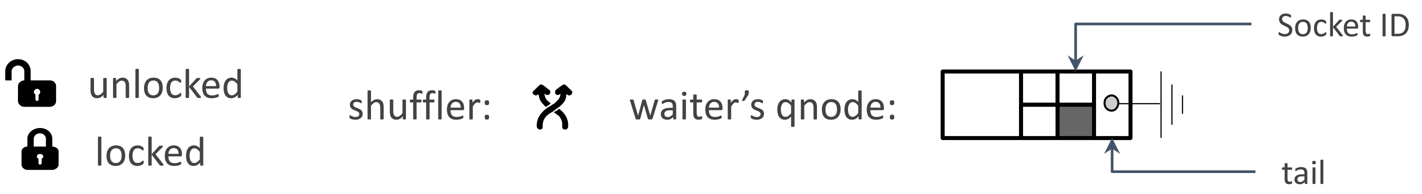
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





NUMA-aware SHFLLOCK in action

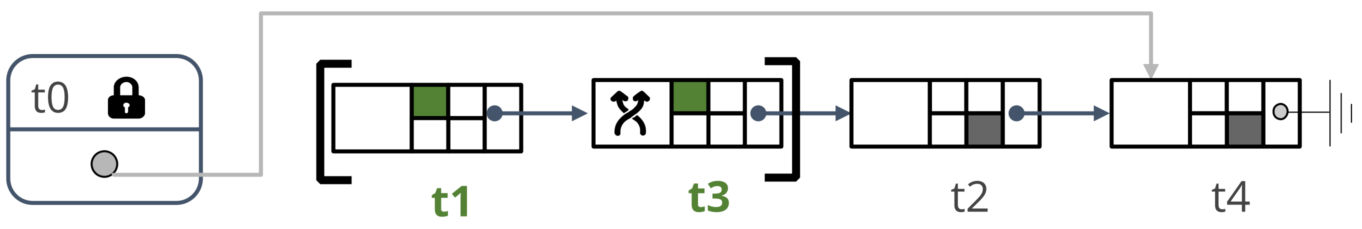
t0 (socket 0): lock()

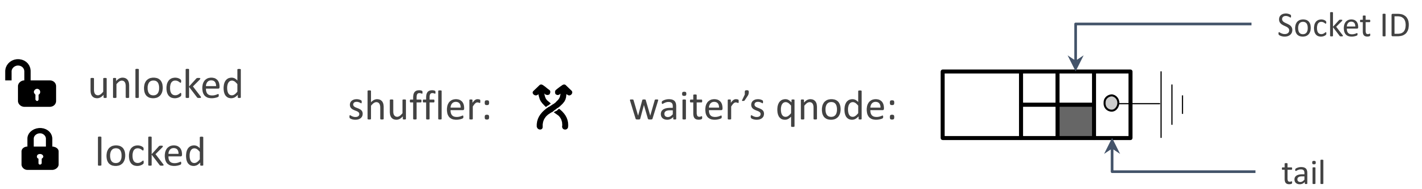
Multiple threads join the queue

Shuffling in progress

t0: unlock()

t3 now becomes the shuffler





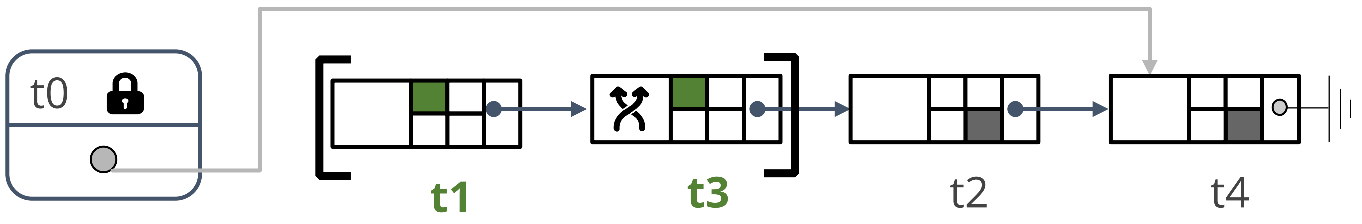
NUMA-aware SHFLLOCK in action

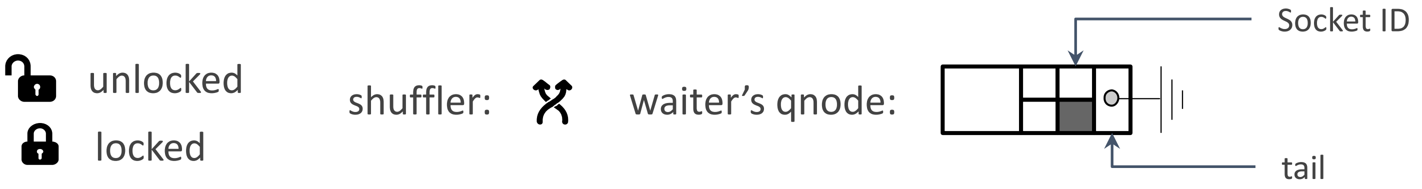
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





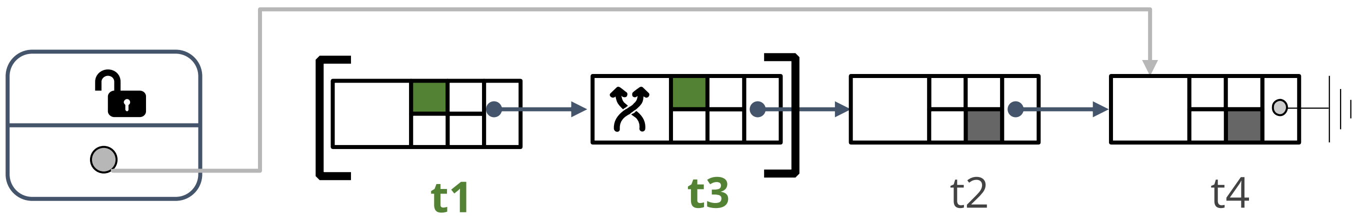
NUMA-aware SHFLLOCK in action

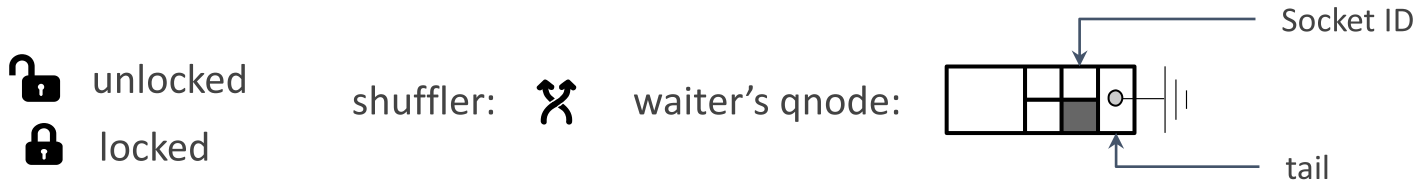
t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()





NUMA-aware SHFLLOCK in action

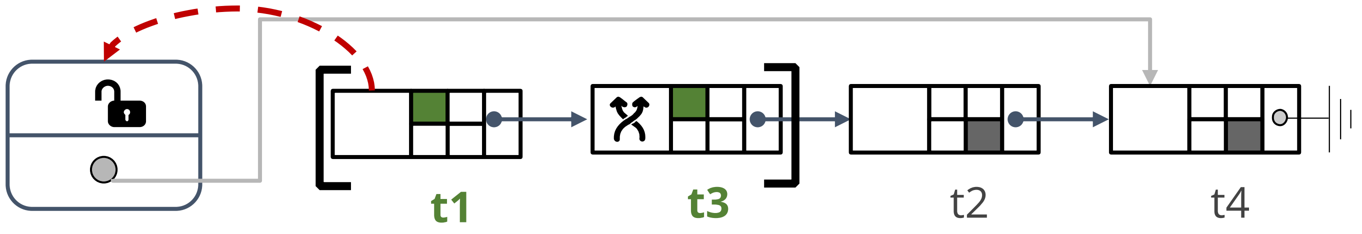
t0 (socket 0): lock()

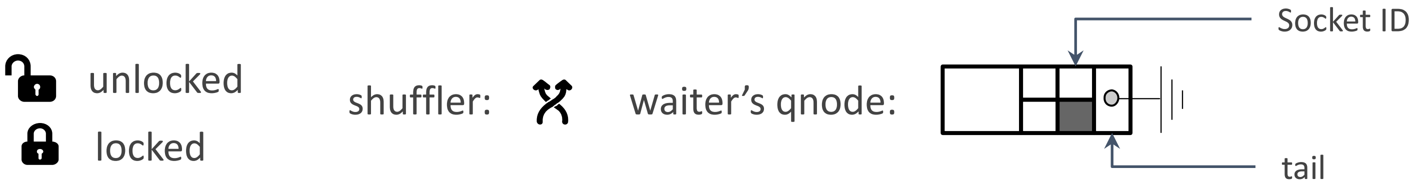
Multiple threads join the queue

Shuffling in progress

t0: unlock()

t1 acquires the lock via CAS





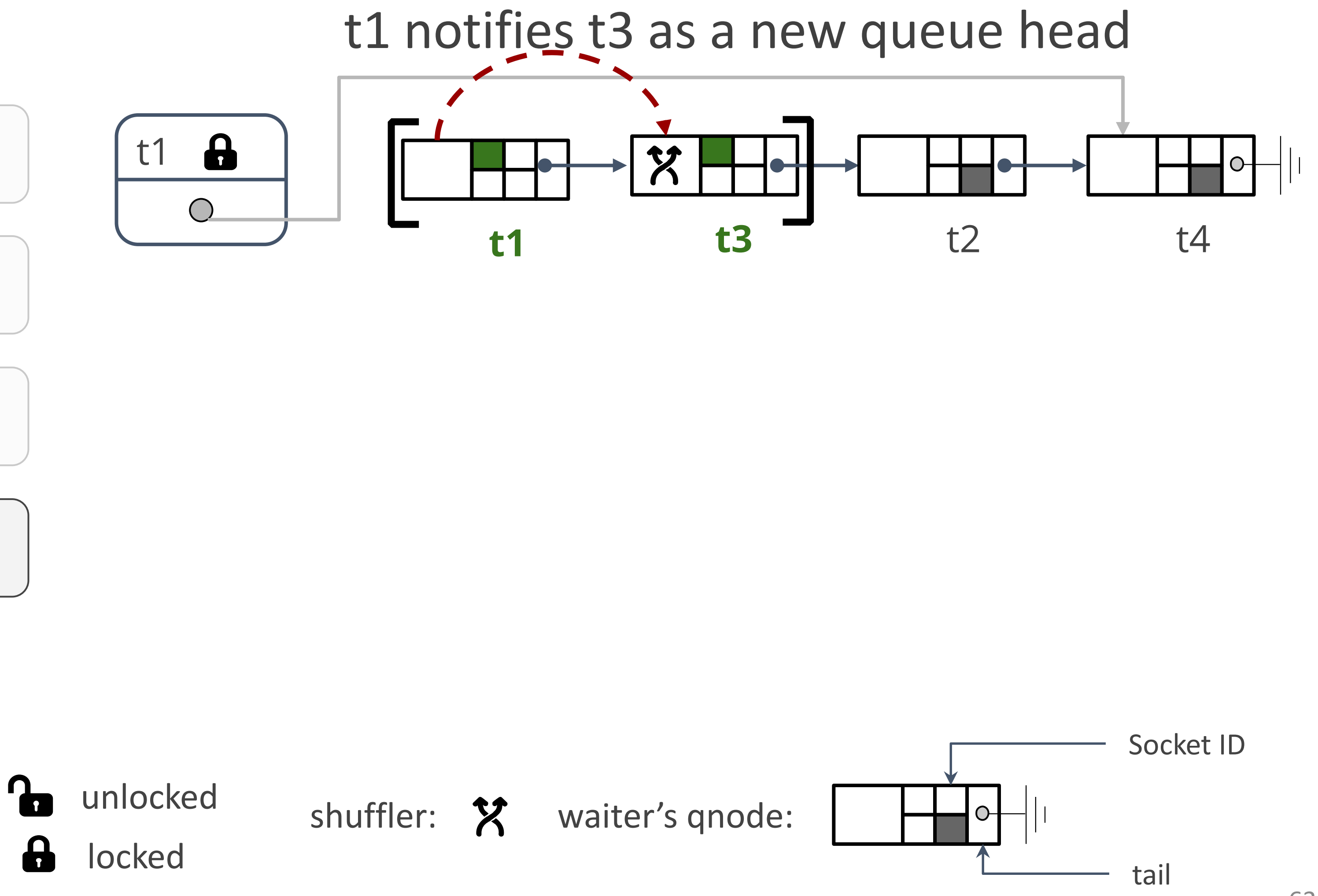
NUMA-aware SHFLLOCK in action

t0 (socket 0): lock()

Multiple threads join the queue

Shuffling in progress

t0: unlock()



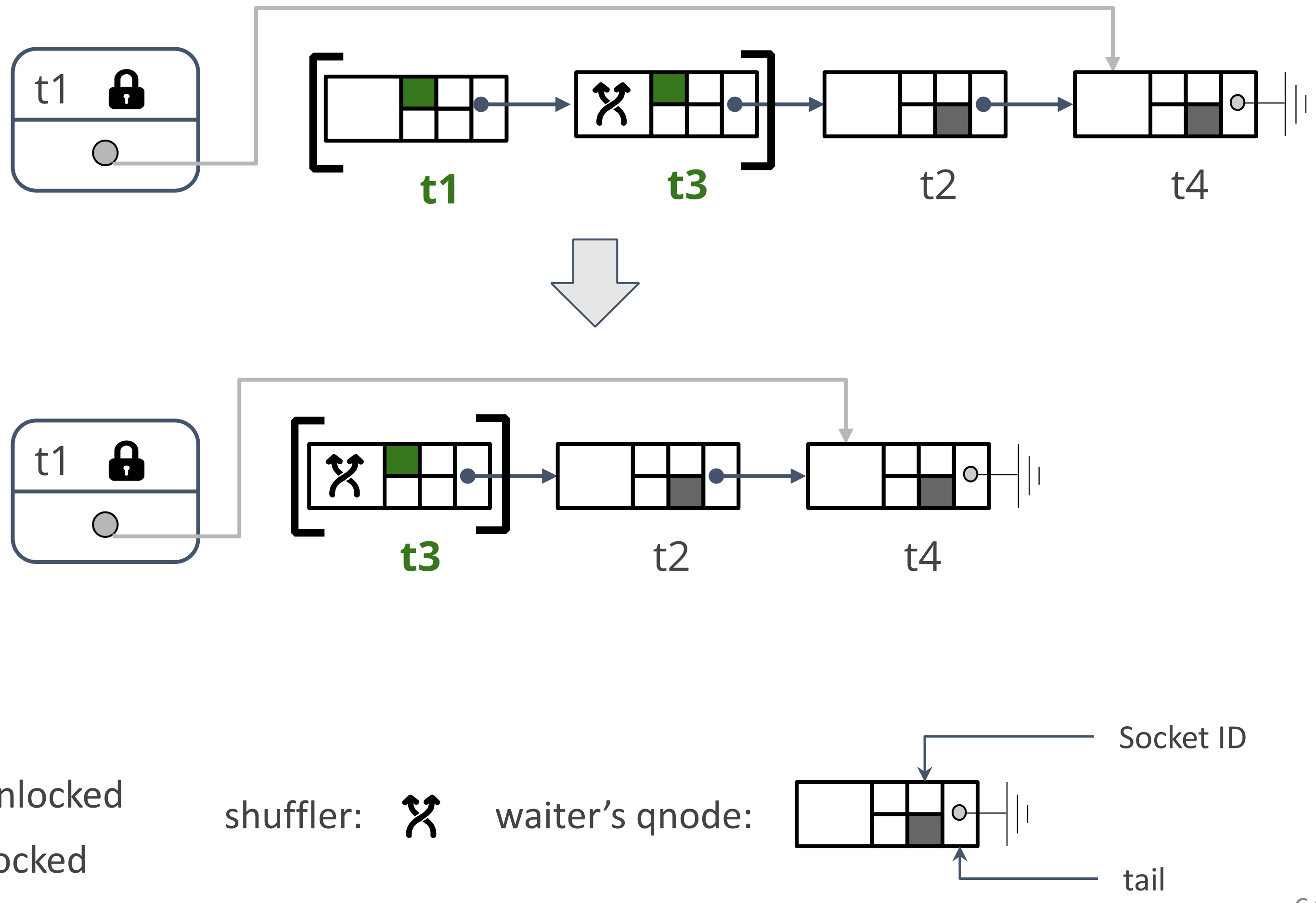
NUMA-aware SHFLLOCK in action

t0 (socket 0): lock()

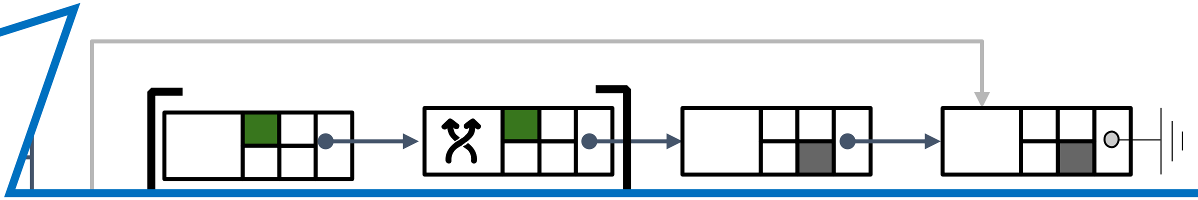
Multiple threads join the queue

Shuffling in progress

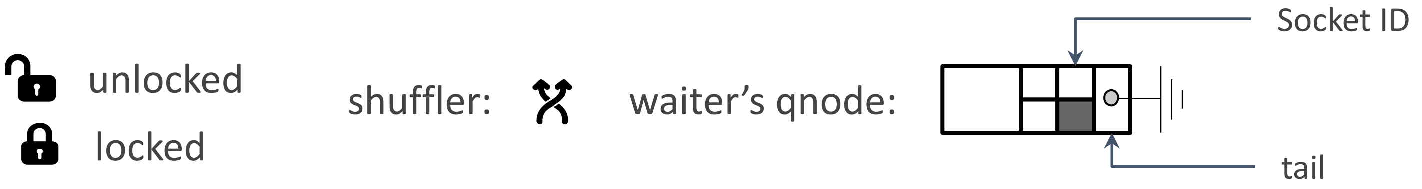
t0: unlock()



Shuffling invariants for correctness



→ Only **one waiter** is a shuffler at a time
→ Shuffling starts from the **head of the queue**
→ Shuffler can pass the shuffling role to any of its **successor**
→ After passing the shuffling role, the **waiter only spins**



SHFLLOCKS: Family of lock algorithms

NUMA-aware spinlock

NUMA-aware blocking lock

NUMA-aware writer preferred readers-writer lock

NUMA-aware blocking SHFLLOCK

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

Queue tail (8B)
(waiters list)

- Extension of NUMA-aware spinlock
- Handles core oversubscription
  - Shuffler wakes up sleeping waiter besides reordering
  - No extra data structure required for parking

Single parking list

- Spinning and parked waiters are in one list
  - Prior locks maintain a separate parking list
- Shuffler ensures:
  - NUMA-awareness by reordering queue
  - Shuffled waiters are always spinning

★ **Lock size remains intact**

★ **Shuffler ensures NUMA-awareness in both under- and over-subscribed case**

Minimal scheduler intervention

- Always pass the lock to a **spinning** waiter
  - Shuffler ensures by waking up shuffled waiters
  - Steal the global TAS lock
- Waiters only park if more than one tasks are running on a CPU (system load)

★ **Scheduler is mostly off the critical path**

★ **Guarantees forward progress of the system**

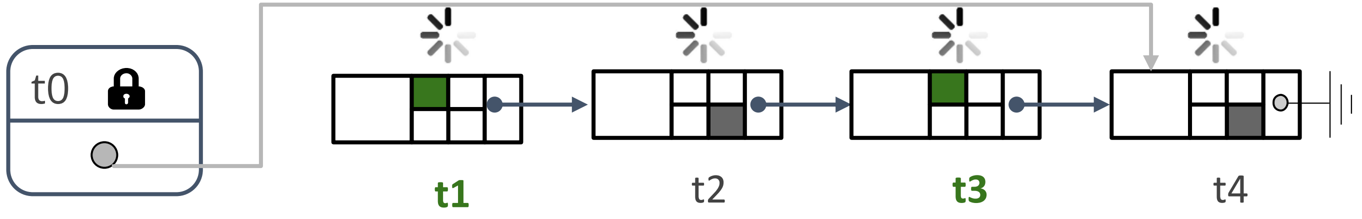
NUMA-aware blocking SHFLLOCK in action

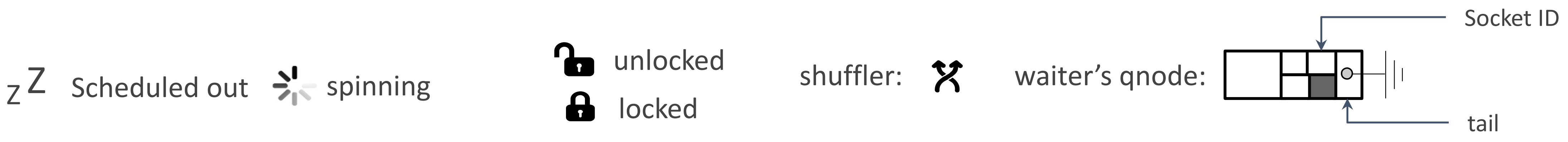
t0 (socket 0): lock()

Multiple threads join the queue

Threads go to sleep

Shuffling in progress





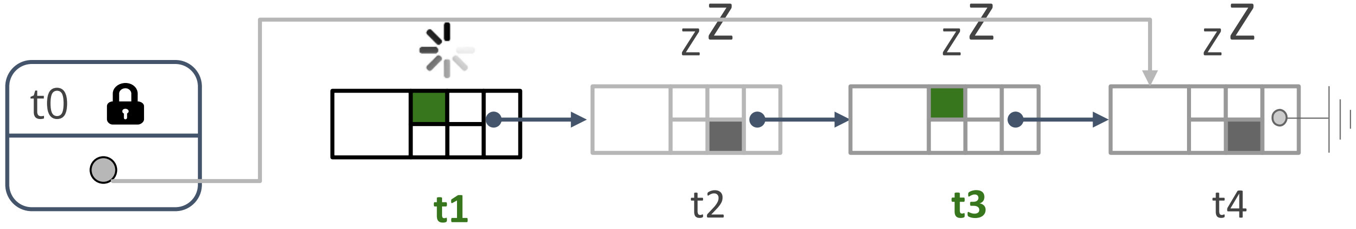
NUMA-aware blocking SHFLLOCK in action

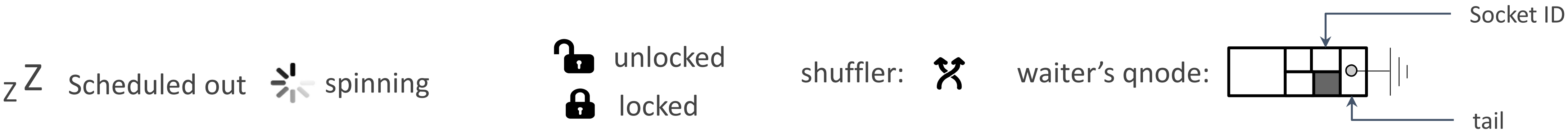
t0 (socket 0): lock()

Multiple threads join the queue

Threads go to sleep

Shuffling in progress





NUMA-aware blocking SHFLLOCK in action

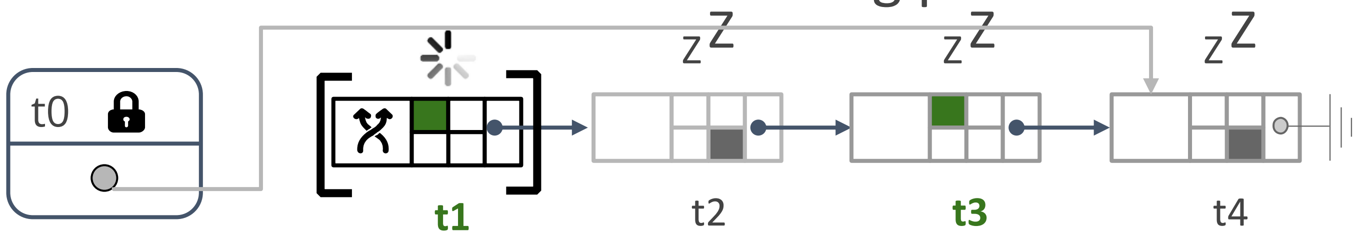
t0 (socket 0): lock()

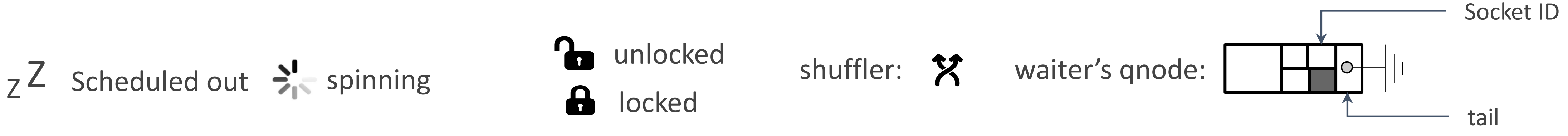
Multiple threads join the queue

Threads go to sleep

Shuffling in progress

t1 starts the shuffling process





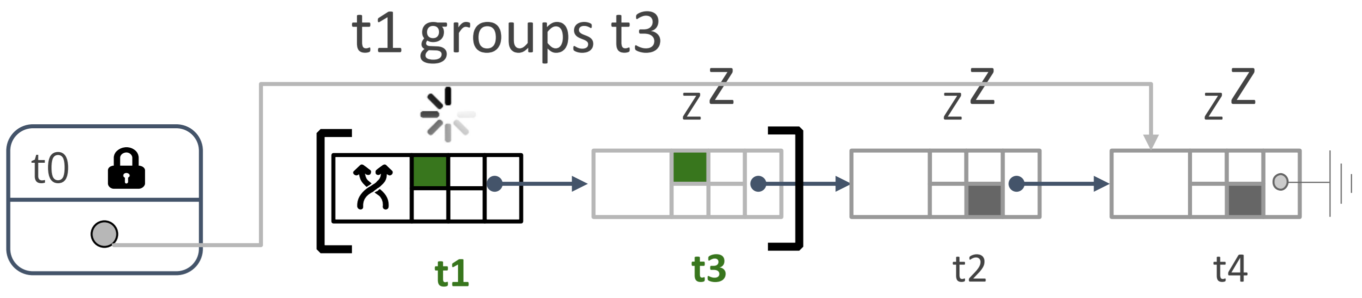
NUMA-aware blocking SHFLLOCK in action

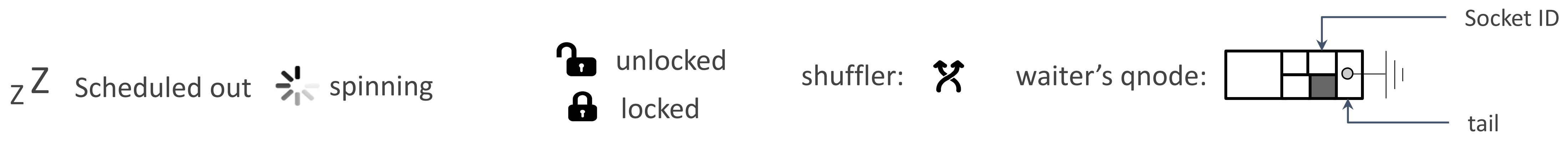
t0 (socket 0): lock()

Multiple threads join the queue

Threads go to sleep

Shuffling in progress





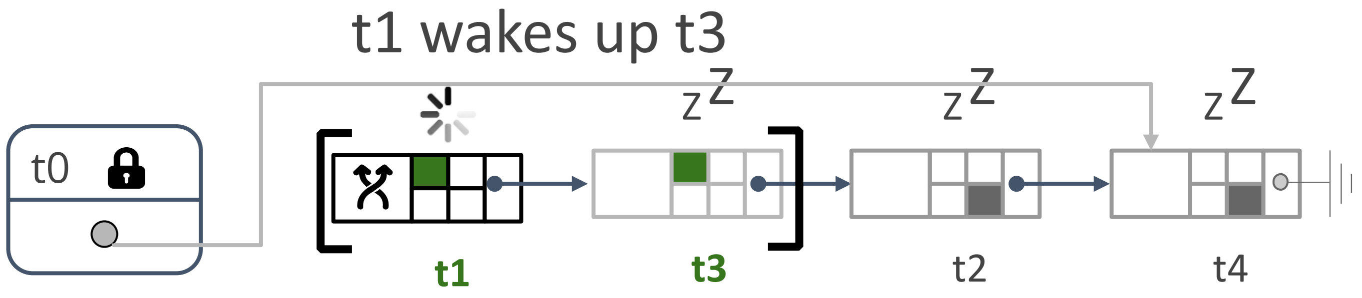
NUMA-aware blocking SHFLLOCK in action

t0 (socket 0): lock()

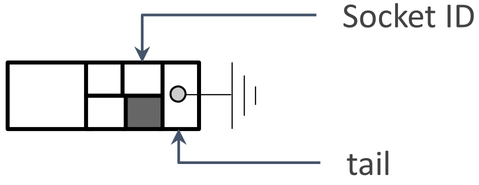
Multiple threads join the queue

Threads go to sleep

Shuffling in progress



zZ Scheduled out   spinning   unlocked   locked   shuffler: waiter's qnode:



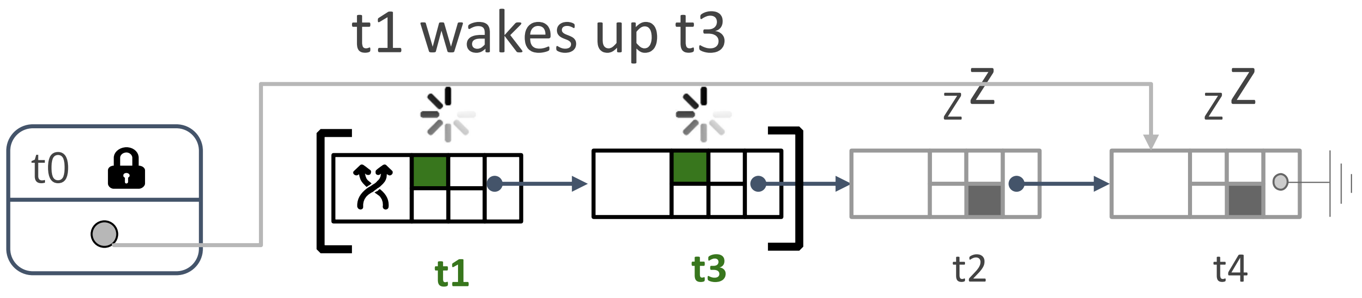
NUMA-aware blocking SHFLLOCK in action

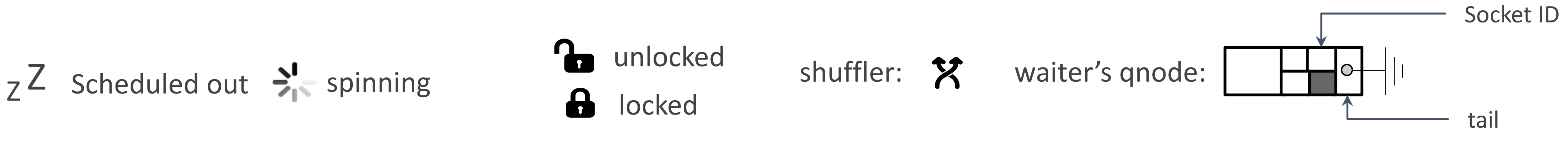
t0 (socket 0): lock()

Multiple threads join the queue

Threads go to sleep

Shuffling in progress





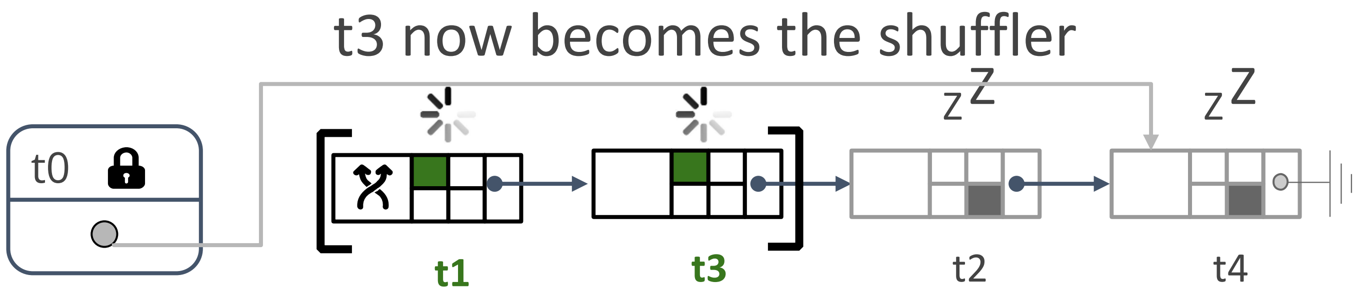
NUMA-aware blocking SHFLLOCK in action

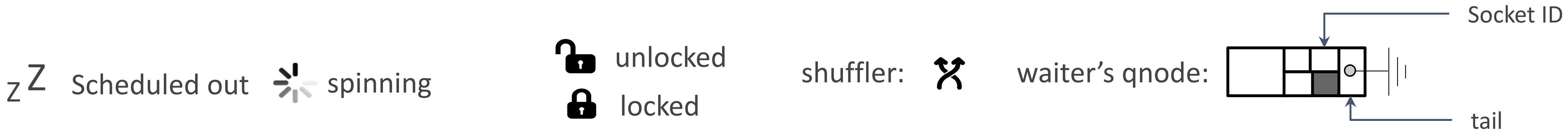
t0 (socket 0): lock()

Multiple threads join the queue

Threads go to sleep

Shuffling in progress





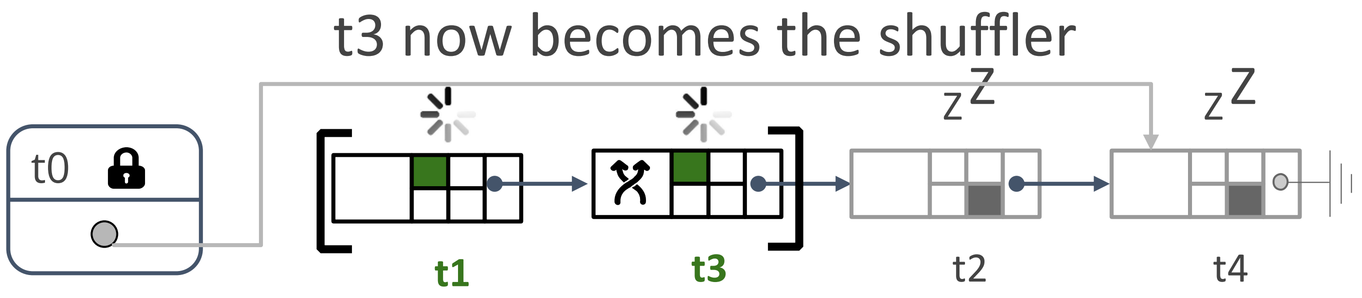
NUMA-aware blocking SHFLLOCK in action

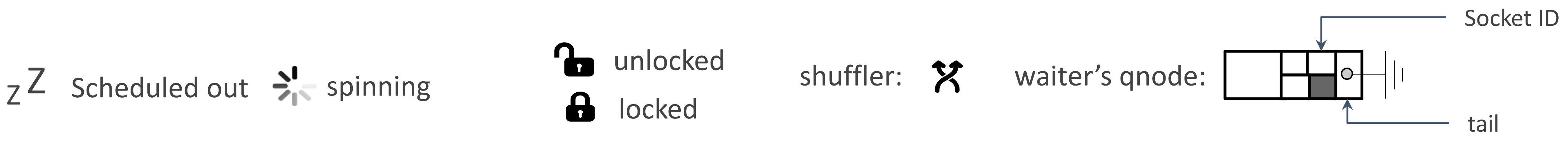
t0 (socket 0): lock()

Multiple threads join the queue

Threads go to sleep

Shuffling in progress





Current blocking locks

Locks	Handling threads contention			Over subscribe	NUMA aware	Memory footprint		Lock API
	Low	Medium	High			Lock size	Context	
mutex	✓	✗	✗	Bad	NO	40 B		
Malthusian	✓	✓	✗	Good*	NO	24 B	O(N*T)	Modified
CST	✗	✓	✓	Good	Yes	32 + 512 B		
SHFLLOCK	✓	✓	✓	Good	Yes	12 B		

- Extension of non-blocking locks

- T → Number of threads; N → number of locks

- Lock API modified → lock/unlock(L, ctx)

SHFLLOCKS: Family of lock algorithms

- NUMA-aware spinlock
- NUMA-aware blocking lock
- NUMA-aware writer preferred readers-writer lock

Readers-writer SHFLLOCK

Count (4/8B)
(readers indicator)

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

Queue tail (8B)
(waiters list)

- Extension of blocking lock
- Centralized counter encodes readers and writer
- Waiting readers and writer enqueued in one waiting list

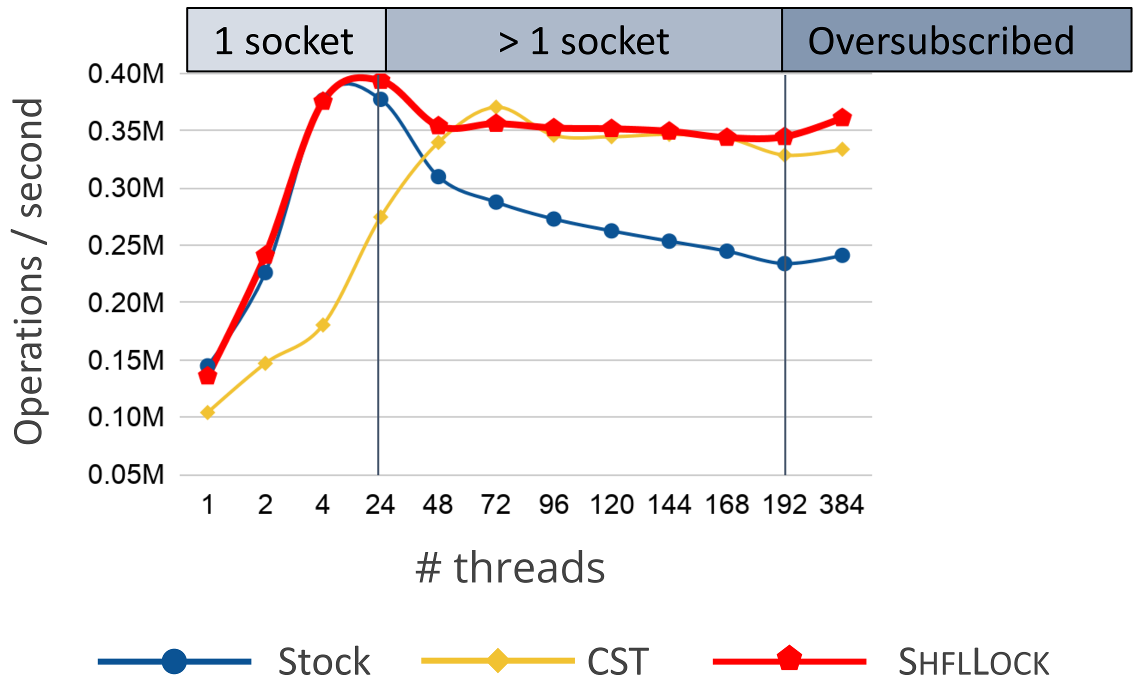
Analysis of SHFLLOCK

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

Setup: 192-core/8-socket machine

Locks performance: Throughput

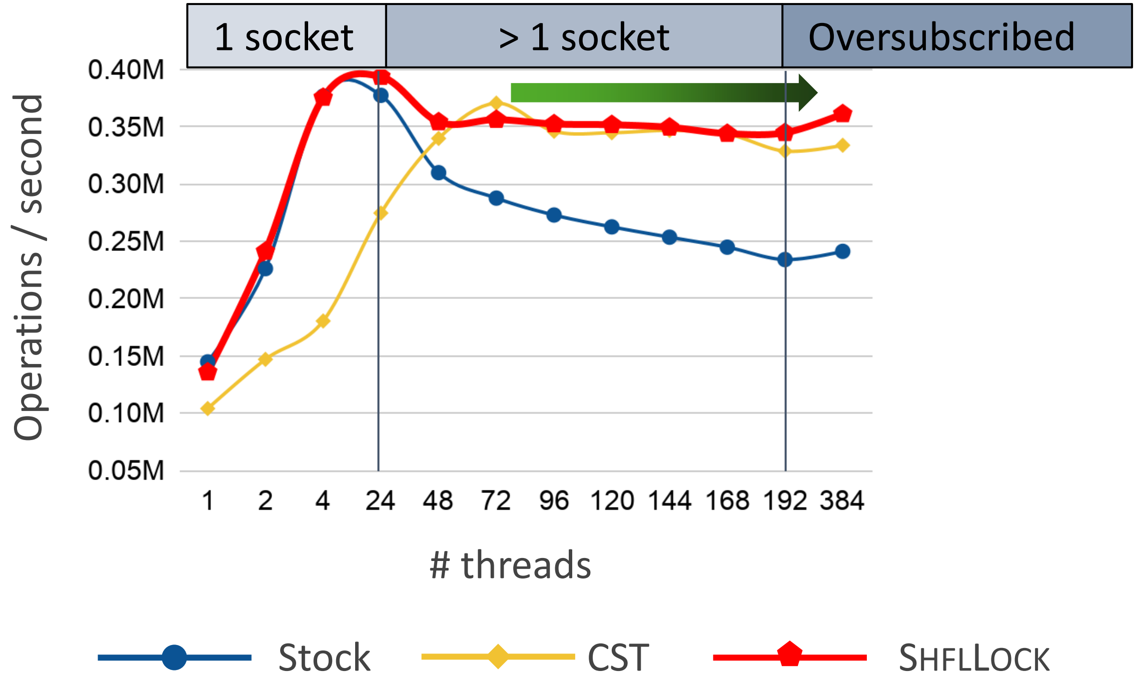
Benchmark: Each thread creates a file, a serial operation, in a shared directory



- SHFLLOCKS maintain performance:

Locks performance: Throughput

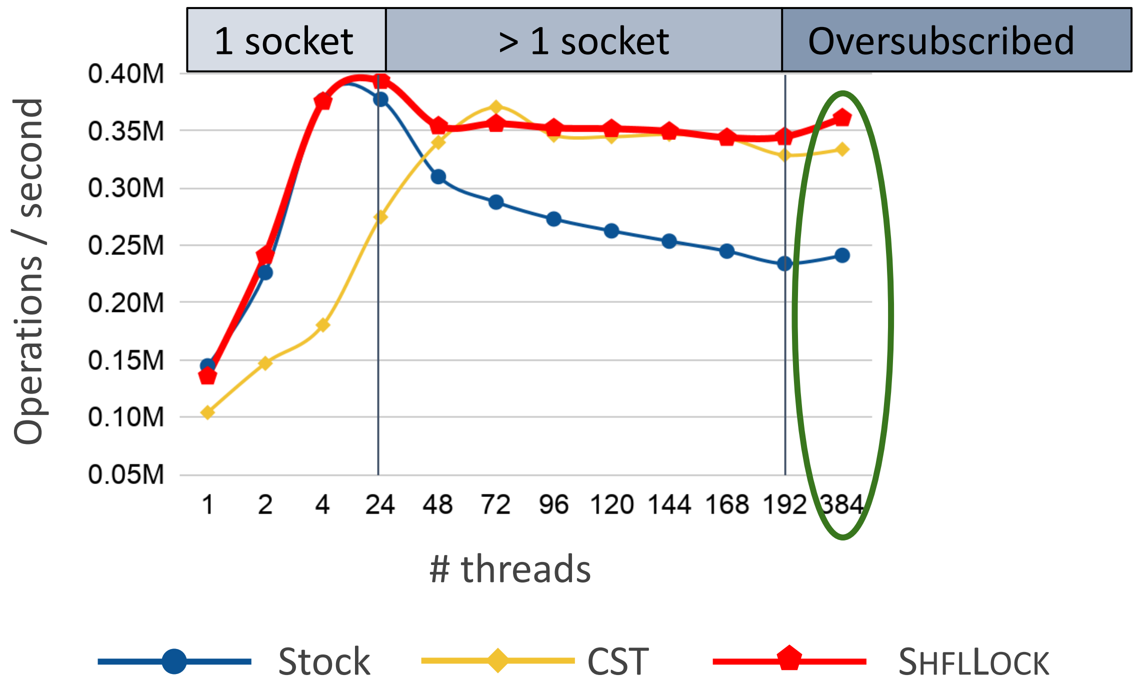
Benchmark: Each thread creates a file, a serial operation, in a shared directory



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

Locks performance: Throughput

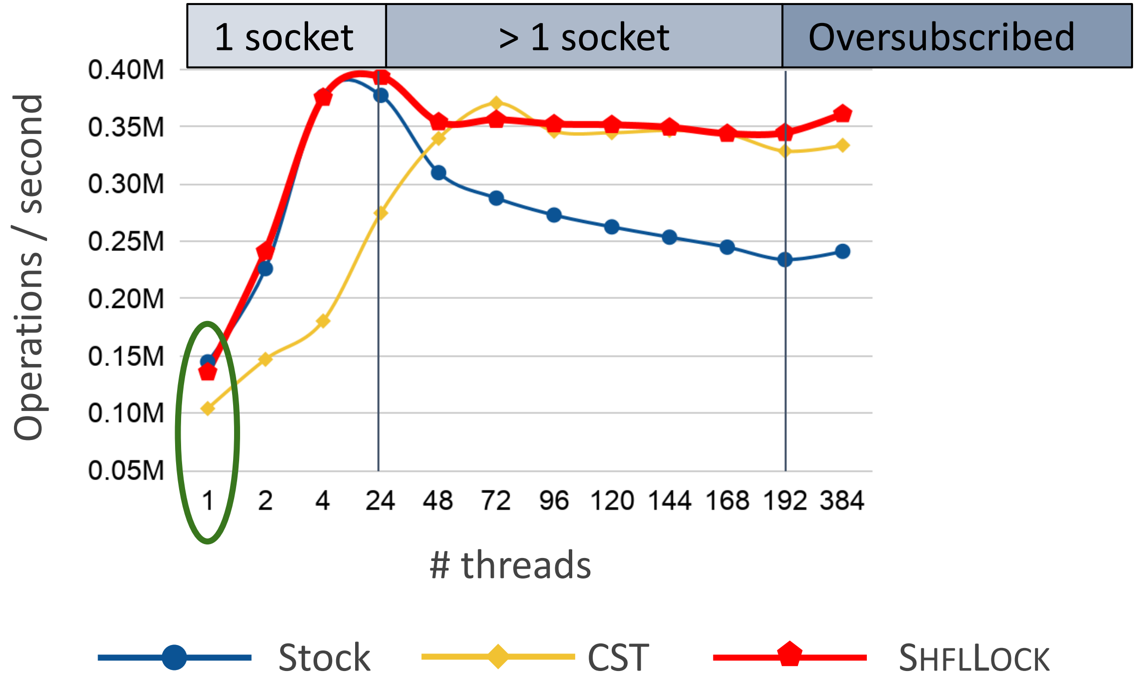
Benchmark: Each thread creates a file, a serial operation, in a shared directory



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

Locks performance: Throughput

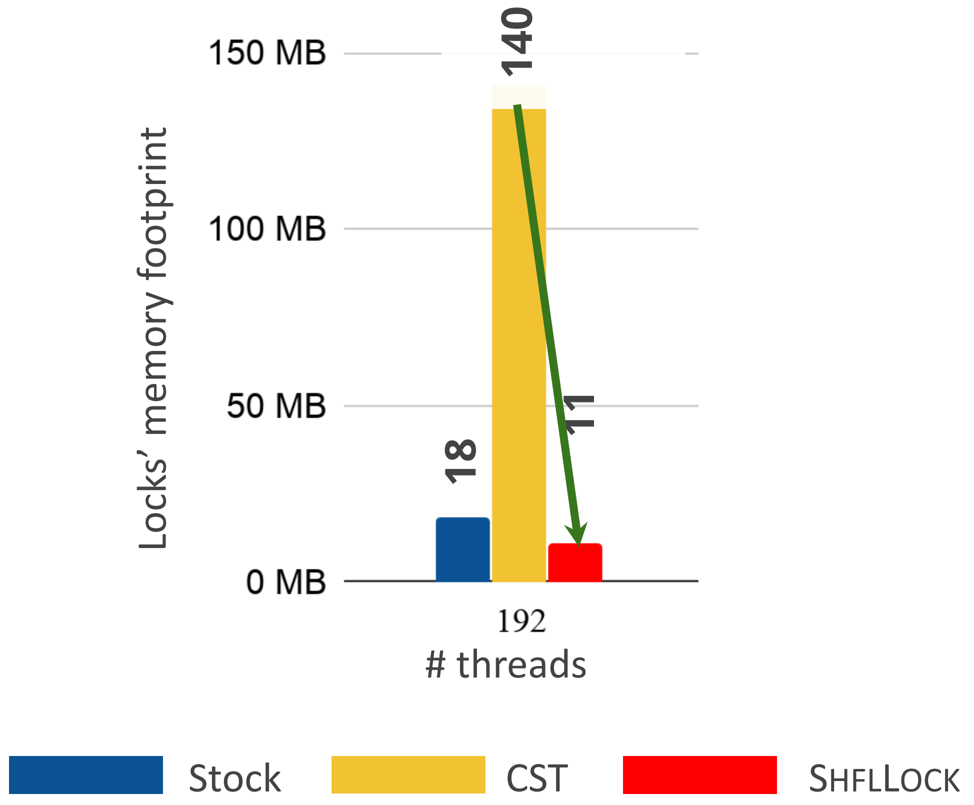
Benchmark: Each thread creates a file, a serial operation, in a shared directory



- 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

Benchmark: Each thread creates a file, a serial operation, in a shared directory



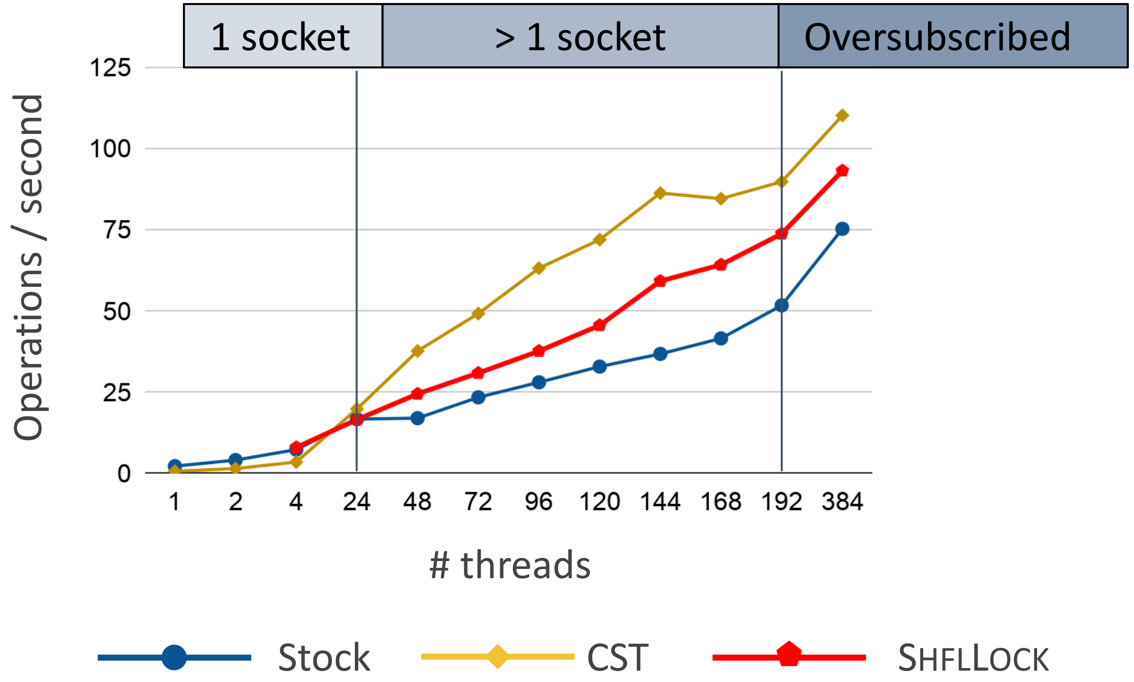
- SHFLLOCK has least memory footprint

  Reason: **No extra auxiliary data structure**

  - ➢ Stock: parking list structure + extra lock
  - ➢ CST: per-socket structure

RWLocks performance: Throughput

Benchmark: Each thread enumerates files in a shared directory: read-only workload

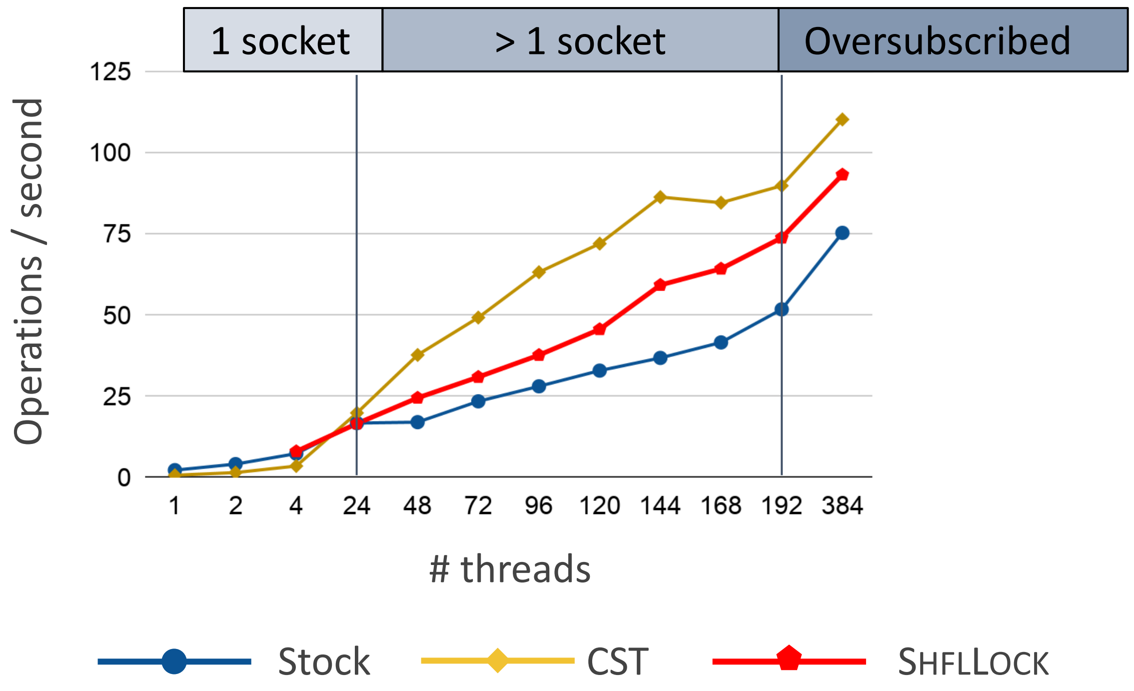


- SHFLLOCK is better than Stock:

  Few atomic instructions on the critical path

RWLocks performance: Throughput

Benchmark: Each thread enumerates files in a shared directory: read-only workload



- SHFLLOCK is better than Stock:

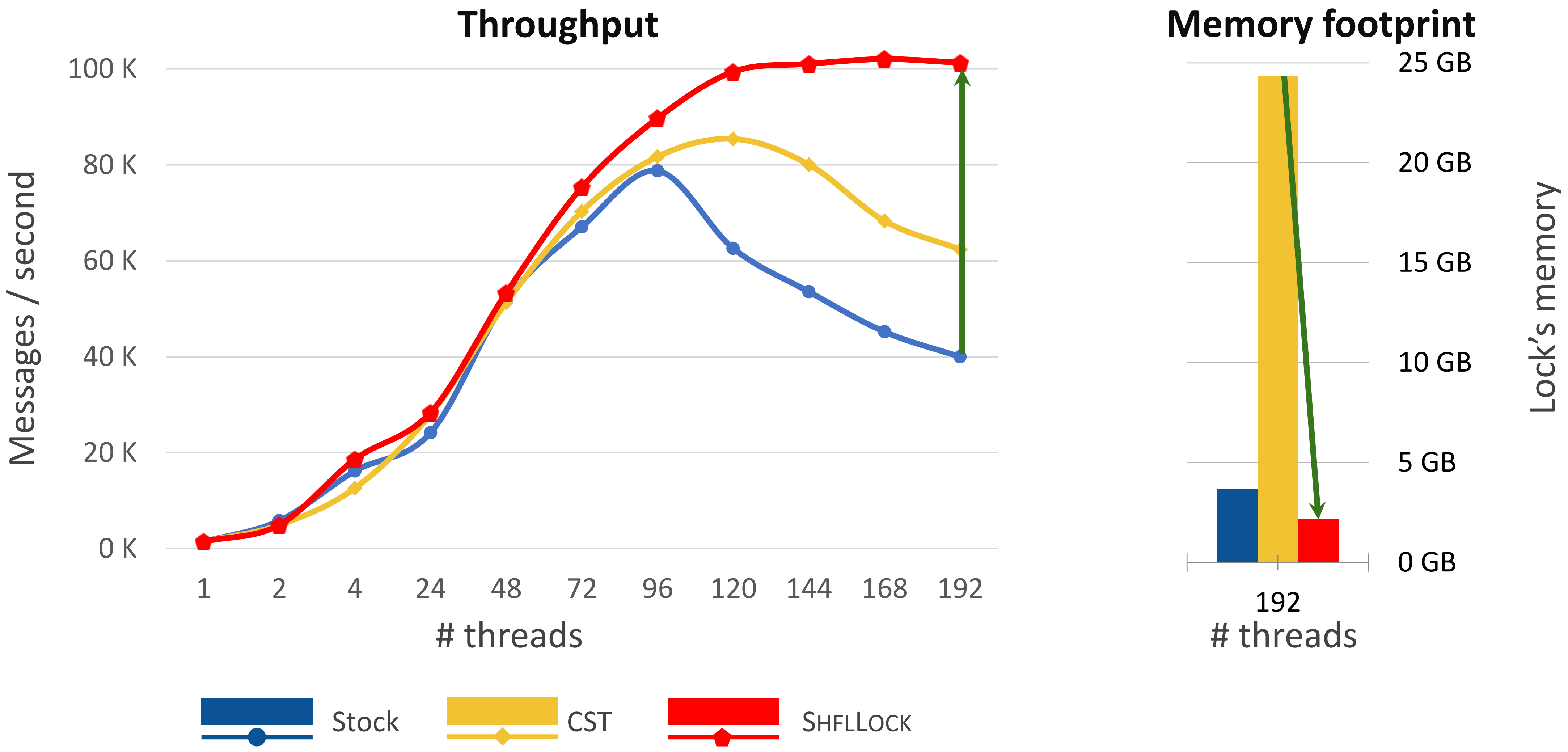
  Few atomic instructions on the critical path

- CST is better than SHFLLOCK:

  CST uses per-socket counters than a centralized counter
  → Minimizes coherence traffic

SHFLLOCK improves Exim's performance

Process intensive and stresses memory subsystem, file system and scheduler



Discussion

- Lock holder splits the queue:
  - NUMA-awareness: Compact NUMA-aware lock (CNA)
  - Blocking lock: Malthusian lock
- Shuffling can support other policies:
  - Non-inclusive cache (Skylake architecture)
  - Multi-level NUMA hierarchy (SGI machines)
  - Priority inheritance or boosting

SHFLLOCK: Conclusion

- Current lock designs:
    - Do not maintain best throughput with varying threads
    - Have high memory footprint
- **Shuffling**: Dynamically reorder the list or modify waiter's state
    - NUMA-awareness, waking up waiters
- **SHFLLOCKS**: Shuffling-based family of lock algorithms
    - Best throughput with no extra memory overhead
    - Utilize wasted CPU waiters to amortize lock operations

Acknowledgement

Taesoo Kim
Jaeho Kim
Byoungyoun Lee
Xiaohe Cheng
Seulbae Kim
Meng Xu
Junyeon Yoon
Wen Xu
Virendra Marathe
Dave Dice
Changwoo Min
Rohan Kadekodi
Yihe Huang
Yizhou Shan
Ajit Mathew
Madhava Krishnan
Woonhak Kang
Steffen Maass
Margo Seltzer
Alex Kogan
Irina Calciu
Pavel Emelyanov
Tushar Krishna
Vijay Chidambaram
Kangnyeon Kim
Jayashree Mohan
Mohan Kumar
Se Kwon Lee
Steve Byan
Jean Pierre Lozi

Conclusion

- **Designing scalable synchronization mechanisms is critical**
- **This thesis:**
  - **Lock algorithms** that decouple lock design from hardware and software policy
  - A **constant ordering primitive** that scales to 100s-1000s of CPUs
  - **Adding semantic information** to task schedulers to minimize double scheduling

Minimizing scheduling overhead of concurrent events that leverage both hardware and software efficiently

Thank you!