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
   - In single thread
   - In oversubscription
     ➡️ Minimize lock contentions
     ➡️ No penalty when not contended
     ➡️ 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)

- **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)

- **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**

Setup: 192-core/8-socket machine
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$^1$

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 affects 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

Socket ID (e.g., socket 0)

shuffler: × waiter's qnode:

Socket ID
tail
Sort waiters on the fly using socket ID

Another waiter is in a different socket

Socket 3

shuffler: waiter's qnode:
Sort waiters on the fly using socket ID

More waiters join

shuffler: X  waiter's qnode:
Sort waiters on the fly using socket ID

Shuffler (t1) sorts based on socket ID
Shuffling: Design methodology

A waiter (shuffler $\mathcal{X}$) 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
<table>
<thead>
<tr>
<th>SHFLLOCKS</th>
</tr>
</thead>
</table>
| **TAS (4B)**  
  (test-and-set lock) |
| **Queue tail (8B)**  
  (waiters list) |
SHFLLOCKS

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

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

.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()

Diagram:
- Unlock symbol
- Socket ID
- Locked
- Unlock
- Waiter's qnode
- Shuffler
NUMA-aware SHFLLOCK in action

- t0 (socket 0): lock()
- Multiple threads join the queue
- Shuffling in progress
- t0: unlock()

Socket ID

Unlocked

Locked

shuffler: waiter's qnode:

Socket ID

Tail
NUMA-aware SHFLLOCK in action

- **t0 (socket 0): lock()**
- **Multiple threads join the queue**
- **Shuffling in progress**
- **t0: unlock()**

![Diagram of shuffling process]
NUMA-aware SHFLLOCK in action

- t0 (socket 0): lock()
- Multiple threads join the queue
- Shuffling in progress
- t0: unlock()

Diagram:
- t1 starts the shuffling process
- shuffler: X
- waiter’s qnode: Socket ID
- unlocked
- locked
- tail
NUMA-aware SHFLLOCK in action

- t0 (socket 0): lock()
- Multiple threads join the queue
- Shuffling in progress
- t0: unlock()

Diagram:
- t0 (locked)
- t1 groups t3
- t1
- t3
- t2
- t4

Symbols:
- ![unlocked](image)
- ![locked](image)

Legend:
- Shuffler: ☘
- Waiter's qnode: ![image]
- Socket ID
- Tail
NUMA-aware SHFLLOCK in action

- t0 (socket 0): lock()
- Multiple threads join the queue
- Shuffling in progress
- t0: unlock()

![Diagram](image-url)

- t3 now becomes the shuffler
- Shuffler: unlock
- Waiter's qnode: locked
- Socket ID
- Tail
NUMA-aware SHFLLOCK in action

- t0 (socket 0): lock()
- Multiple threads join the queue
- Shuffling in progress
- t0: unlock()

Diagram:
- t3 now becomes the shuffler
- Socket ID
- unlocked
- locked
- shuffler: ✗  waiter’s qnode:tail
NUMA-aware SHFLLOCK in action

- **t0 (socket 0): lock()**
- Multiple threads join the queue
- Shuffling in progress
- **t0: unlock()**

**Diagram:**
- Unlocked and locked states for shuffler and waiter's qnode.
- Socket ID and tail indicators.
NUMA-aware SHFLLOCK in action

- **t0 (socket 0): lock()**
- **Multiple threads join the queue**
- **Shuffling in progress**
- **t0: unlock()**

Unlocked unlocked
Locked locked

Shuffler: ✗ Waiter’s qnode:

Socket ID
Tail
NUMA-aware SHFLLOCK in action

- t0 (socket 0): lock()
- Multiple threads join the queue
- Shuffling in progress
- t0: unlock()

Diagram:
- t1 acquires the lock via CAS
- Shuffler: ❌
- Waiter's qnode: Socket ID
- Tail
NUMA-aware SHFLLOCK in action

- t0 (socket 0): lock()
- Multiple threads join the queue
- Shuffling in progress
- t0: unlock()

Diagram:
- t1 notifies t3 as a new queue head
- Shuffler: \( \times \)  waiter's qnode:
- Unlocked: \( \bigcirc \)  Locked: \( \bigcirc \)
NUMA-aware SHFLLOCK in action

- **t0 (socket 0): lock()**
- **Multiple threads join the queue**
- **Shuffling in progress**
- **t0: unlock()**

Diagram:
- **Socket ID**
- **Locked: 🔐**
- **Unlocked: 🔓**
- **Shuffler:** ✗
- **Waiter's qnode:**
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 + SHFLOCK

[https://github.com/sslab-gatech/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)

- **SHFLOCK** 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

- **SHFLOCK**S: Shuffling-based family of lock algorithms
  - NUMA-aware minimal memory footprint locks
  - Utilize waiters to amortize lock operations

Thank you!