

Optimizing Spin Locks for Multi-core Processors



中正大學 資訊工程研究所

指導教授：羅習五 副教授

學生：林靖紳

Outline

- Introduction
- Related Work
- Implementation
- Evaluation
- Conclusion





Introduction

Introduction - Spinlocks Overview in Multi-Core CPUs

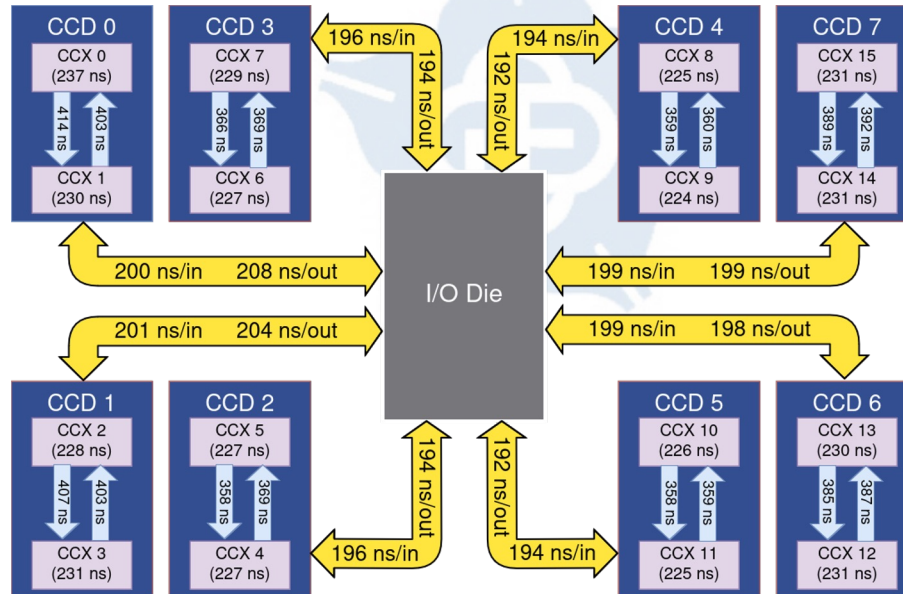
- Rise of Multi-core Processors
 - The advent of multi-core processors has revolutionized the computing field, offering unprecedented performance capabilities.
 - This architecture allows parallel processing but also introduces the challenge of ensuring synchronization mechanisms for concurrent access to shared resources.
- Why focus on spinlocks:
 - Spinlocks can be used independently, crucial for high-performance applications.
 - Other locks like mutexes and semaphores are implemented using spinlocks; for example, a mutex is a spinlock combined with a context switch.

Introduction - System Architecture Impact

- NUMA and ccNUMA Systems
 - Memory access times vary based on processor proximity.
 - Maintaining data coherence (cache coherence protocols) introduces overhead.
 - Spinlocks can increase interconnect traffic, reducing system performance.
- AMD Threadripper PRO 3995WX Case Study
 - Under the Multi-Chip Module (MCM) architecture, the system includes 4 NUMA nodes and multiple cores.
 - Uses snoop-based and directory-based cache coherence.
 - Serialization costs: Contention cost (next task determination) and Handoff cost (data passing).

Introduction - System Architecture Impact

- Core-to-Core Latency
 - Significant latency variations in ccNUMA environments impact performance.



Introduction - Optimizing Spinlock Performance

- NUMA-Aware Spinlocks and RON
 - Traditional grouping spinlocks often assume uniform latencies within a numa node.
 - RON reduces handoff costs using a routing table, optimizing data exchange paths.
- Potential Issues with RON:
 - **Memory Overhead:** Data structure increases linearly with the number of cores, leading to higher memory usage.
 - **Contention Cost:** RON handoff locks with the time complexity of $O(N)$.

Introduction - nxtRON ans shRON

- New Algorithms: next-RON (nxtRON) and shared-RON (shRON)
 - Developed to address limitations in RON.
 - Aim to enhance performance and scalability in multi-core systems.
- nxtRON:
 - Focuses on **optimizing contention costs**.
 - Aims to reduce the **time complexity** of RON
- shRON:
 - Targets **reducing memory overhead** and **cache contention** compared to traditional spinlock approaches.
 - Implements innovative data sharing and synchronization techniques tailored for modern multi-core CPUs.



Related Work

RON: One-Way Circular Shortest Routing

- Problem Description

- The number of cores on a single CPU is increasing
- Accessing shared data involves transferring data between different CPU cores
- Target: Minimize the data transfer between cores

- Problem Solution

- "How to allow threads on different cores to access shared data" is more like a **path planning problem**
- Pre-establish routing table using "Approximate Shortest Circular Path" to determine the order in which each core enters the critical section.

RON

Algorithm 1 The RON Algorithm

```
1  int TSP_ID_ARRAY[NUM_core]; /*per-process*/
2  thread_local TSP_ID; /*thread-local-storage*/
3  atomic_bool InUse=false; /*per-lock*/
4  atomic_int WaitArray[NUM_core]; /*per-lock*/
5  TSP_ID = TSP_ID_ARRAY[getcpu() ]
6  void spin_init()
7      for (each element in WaitArray)
8          element = 0;
9  void spin_lock()
10     WaitArray[TSP_ID]=1;
11     while(1)
12         if (WaitArray[TSP_ID]==0)
13             return;
14         if (cmp_xchg(&InUse, false, true)):
15             WaitArray[TSP_ID] = 0
16             return;
17 void spin_unlock()
18     for (int i=1; i<NUM_core; i++)
19         if (WaitArray[(i+TSP_ID)%NUM_core]==1)
20             WaitArray[(i+TSP_ID)%NUM_core]=0;
21         return;
22 InUse=false;
```

- *int TSP_ID_ARRAY[]*
 - Pre-establish routing table
- *thread_local TSP_ID*
 - The core's order in the routing table
 - Each thread has its own *TSP_ID*
- *atomic_bool InUse*
 - Someone gets the lock or not
 - If *InUse* = *false*: no thread in CS
- *atomic_int WaitArray[]*
 - Which core wants the lock (waiting to enter CS)

RON

Algorithm 1 The RON Algorithm

```
1  int TSP_ID_ARRAY[NUM_core]; /*per-process*/
2  thread_local TSP_ID; /*thread-local-storage*/
3  atomic_bool InUse=false; /*per-lock*/
4  atomic_int WaitArray[NUM_core]; /*per-lock*/
5  TSP_ID = TSP_ID_ARRAY[getcpu()]
6  void spin_init()
7      for (each element in WaitArray)
8          element = 0;
9  void spin_lock()
10     WaitArray[TSP_ID]=1;
11     while(1)
12         if (WaitArray[TSP_ID]==0)
13             return;
14         if (cmp_xchg(&InUse, false, true)):
15             WaitArray[TSP_ID] = 0
16             return;
17 void spin_unlock()
18     for (int i=1; i<NUM_core; i++)
19         if (WaitArray[(i+TSP_ID)%NUM_core]==1)
20             WaitArray[(i+TSP_ID)%NUM_core]=0;
21     return;
22     InUse=false;
```

- Lock procedure

- Goal: get the lock
- Register the *TSP_ID* wants the lock
- Wait until the *previous TSP_ID* hand the lock to *TSP_ID*
- Or wait until there is no thread in CS

■ *InUse* = *false*

RON

Algorithm 1 The RON Algorithm

```
1  int TSP_ID_ARRAY[NUM_core]; /*per-process*/
2  thread_local TSP_ID; /*thread-local-storage*/
3  atomic_bool InUse=false; /*per-lock*/
4  atomic_int WaitArray[NUM_core]; /*per-lock*/
5  TSP_ID = TSP_ID_ARRAY[getcpu()]
6  void spin_init()
7      for (each element in WaitArray)
8          element = 0;
9  void spin_lock()
10     WaitArray[TSP_ID]=1;
11     while(1)
12         if (WaitArray[TSP_ID]==0)
13             return;
14         if (cmp_xchg(&InUse, false, true)):
15             WaitArray[TSP_ID] = 0
16             return;
17 void spin_unlock()
18     for (int i=1; i<NUM_core; i++)
19         if (WaitArray[(i+TSP_ID)%NUM_core]==1)
20             WaitArray[(i+TSP_ID)%NUM_core]=0;
21         return;
22     InUse=false;
```

• Unlock procedure

- Goal: find the next and release the lock
- Find the next core wants to get the lock from TSP_ID+1
- If find someone want the lock :
 - Hand the lock
 - Set $WaitArray[TSP_ID]$ to 1
- else if cannot find someone want the lock
 - set $InUse$ into false

Related Work - Plock (GNU Pthread's Spinlock)

- Description:
 - Implements test-and-test-and-set (TTAS) mechanism for lock acquisition.
 - Simple and commonly used in POSIX environments like GNU.
 - Prone to fairness issues and scalability limitations under high contention.
- Implementation:
 - Threads repeatedly check and set the lock status using TTAS.
 - Can lead to increased contention and potential thread starvation.

Related Work - Ticket Spinlock

- Description:
 - Uses a ticket-based mechanism where threads wait for their ticket number to match the service number.
 - Ensures fairness by strictly adhering to the order of ticket issuance.
- Implementation:
 - Threads increment their ticket number when waiting to enter the critical section.
 - Waits until their ticket number matches the current service number for entry.

Related Work - RONPlock

- Description:
 - Integrates RON algorithm with pthreads spinlock to tackle oversubscription.
 - Uses optimized locking-unlocking order and routing table to reduce communication costs.
 - Manages thread contention and entry into critical sections efficiently.
- Implementation:
 - Utilizes *WaitArray* to track waiting threads per core.
 - Employs *atomic_fetch_add* for managing nWait variables locally.

Related Work - RONPlock

- Implementation:
 - Utilizes *WaitArray* to track waiting threads per core.
 - Employs *atomic_fetch_add* for managing numWait variables locally.

Algorithm 3 The RON-plock Algorithm

```
1  struct PLock {numWait=0, lock=MUST_WAIT;}  
2  atomic_bool InUse=false; //per-lock variable  
3  PLock WaitArray[NUM_core]; //per-lock variable  
4  void lock()  
5    atomic_inc(&WaitArray[TSP_ID].numWait);  
6    while(1)  
7      if (cmpxchg(&WaitArray[TSP_ID].lock, HAS_LOCK,  
8                MUST_WAIT))  
9        return;  
10     if (cmpxchg(&InUse, false, true))  
11       return;  
12 void unlock()  
13   atomic_dec(&WaitArray[TSP_ID].numWait);  
14   for(int i = 1; i < NUM_core+1; i++)  
15     if(WaitArray[(TSP_ID+i)%NUM_core].numWait>0)  
16       WaitArray[(TSP_ID+i)%NUM_core]=HAS_LOCK;  
17   return;  
18   InUse=false;
```

Related Work - RONTick

- Description:
 - Oversubscribed version of RON algorithm with ticket spinlock.
 - Enhances scalability under low contention by using ticket numbers for queuing.
 - Ensures fairness in critical section access across cores.
- Implementation:
 - Each core manages *grant* and *ticket* variables to control thread entry order.
 - Threads spin on a loop until their ticket matches the *grant* value for their core.

Related Work - MCS Spinlock

- Description:
 - Queue-based spinlock algorithm.
 - Each thread spins on a local flag and waits its turn in a queue structure.
- Implementation:
 - Known for its scalability and efficiency in NUMA architectures.
 - Supports multiple threads waiting for the lock in a FIFO order.

Related Work - C-BO-MCS Spinlock

- Description:
 - Grouping based Spinlock
 - Combines MCS locks for NUMA nodes with back-off locking strategy.
 - Prioritizes neighboring cores to reduce handover costs.
 - Uses test-and-test-and-set with back-off for inter-node competition.
- Implementation:
 - Manages two types of locks: local MCS and back-off for cross-node competition.
 - Enhances performance by minimizing cross-socket communication.

Related Work - Shuffle Lock (ShflLock)

- Description:
 - Dynamically reorders the queue of waiting threads based on a predefined policy.
 - Improves fairness and reduces contention, though may introduce short delays during reordering.
- Implementation:
 - Ensures efficient thread scheduling without additional data structures on the critical path.
 - Optimizes performance by managing thread order dynamically.



Implementation



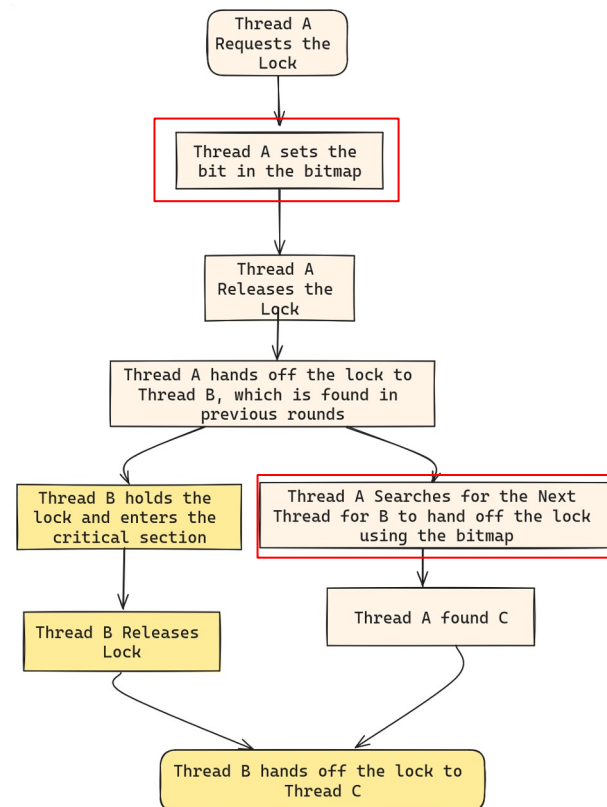
Implementation - nxtRON: Design Overview (1)

- RON-Plock Algorithm:
 - lock: Uses an integer array(WaitArray) to spin
 - unlock: Uses the same array to select the next thread by for loop traversing.
- nxtRON Introduction:
 - **Structure Separation:** nxtRON separates the "auxiliary variable" from the "spinning variable", keeping data consistent.
 - **Bitmap Usage:** Uses a bitmap as the auxiliary variable to track threads waiting for the lock.
- Bitmap Mechanism Advantages:
 - Reduces overhead during thread selection.
 - Reduces time complexity from $O(\#NUM_CORE)$ to $O(1)$.

Implementation - nxtRON: Design Overview (2)

- Thread Request and Lock Release:

- Thread A Requests the Lock
 - A sets the corresponding bit in the bitmap
- Thread A Releases the Lock
 - A hands off the lock to Thread B
- Thread B holds the lock
- Thread A Searches for the Next Thread
 - A finds the next thread for B to handing off the lock using the bitmap



Implementation - nxtRON: Design Overview (3)

- Observation:
 - Separation of **auxiliary variable** and **spinning variable** reduces the cost of *cmpxchg*.
 - **Spinning variable** is only changed during wait and wakeup.
- Key Advantages of nxtRON
 - **Reduced Time Complexity:**
 - Reduces time complexity of selecting the next thread from $O(N)$ to $O(1)$.
 - **Cache Coherence:**
 - By ensuring that the data within the auxiliary and spinning variable is consistent, the nxtRON algorithm benefits from improved cache coherence, further enhancing its performance.

Implementation - nxtRON: Implementation Details (1)

- **Data Structures:**

```
1  /*per-process*/
2  int TSP_ID_ARRAY[NUM_CORE];
3  /*thread-local-storage*/
4  thread_local TSP_ID;
5  TSP_ID = TSP_ID_ARRAY[getcpu()];
6
7  struct WAIT_t:
8      /*Number of waiting threads*/
9      int nWait=0;
10     /*Release(REL) & Acquire(ACQ)*/
11     int Lock=REL;
12
13     /*per-lock structure*/
14     struct nxtRONPlock_t:
15         WAIT_t WaitAry[NUM_CORE];
16         atomic_bool InUse=false;
17         /*#bits=NUM_CORE*/
18         long nxt_ary;
19         int record_next[NUM_CORE];
```

- **WaitArray[]: (spinning variable)**

- Array indicating the wait status of each core.

- **atomic_bool InUse:**

- if any thread is currently in the critical section.
- *InUse = false*: no thread in CS.

- **long nxt_ary: (auxiliary variable)**

- Bitmap where each bit corresponds to a core's wait status.

- **record_next[]:**

- Recording the next core scheduled to enter CS

Implementation - nxtRON: Implementation Details (2)

```
26 void spin_lock():
27     atomic_inc(&WaitAry[TSP_ID].nWait,1);
28     set_bit(TSP_ID, nxt_ary);
29     while(1):
30         while(WaitAry[TSP_ID].Lock!=0 &&
31             InUse==True):
32             CPU_PAUSE();
33             if(cmp_xchg(&WaitAry[TSP_ID].Lock,
34                 ACQ,REL)):
35                 return;
36             if(cmp_xchg(&InUse, false, true)):
37                 return;
```

spin_lock():

- Thread Request
 - Sets the corresponding bit in *nxt_ary*
- Busy-Wait Loop
 - Busy Waiting until its turn to acquire the lock
 - Or there is no thread in CS

Implementation - nxtRON: Implementation Details (3)

```
35 /*Find the first 1 bit set (FFS) after ID*/
36 int find_next(int ID):
37     long tmp = rotate_right(nxt_ary, TID)
38     return (ffs(tmp)+ID)%NUM_CORE;
39
40 void spin_unlock():
41     int next = record_next[TSP_ID];
42     if(next>-1 && WaitAry[next].nWait>0):
43         WaitAry[next].Lock = 0;
44         atomic_dec(&WaitAry[TSP_ID].nWait,1);
45         clr_bit(TSP_ID, nxt_ary);
46         record_next[next] = find_next(next+1);
47     return;
48     InUse=false;
49     atomic_dec(&WaitAry[TSP_ID].nWait,1);
50     clr_bit(TSP_ID, nxt_ary);
```

spin_unlock():

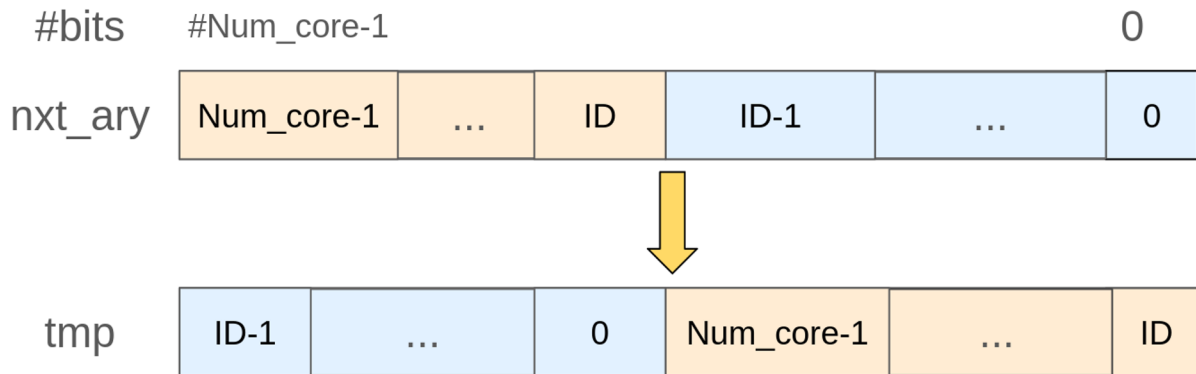
- Handoff the lock
 - We use an `int record_next` array to record the `TSP_ID`'s next
- If `next > -1`:

We had found the value in the last round

 - `TSP_ID` hand the lock to `next` (line 43)
 - `TSP_ID` put down the hand (line 45)
 - `TSP_ID` find the `next`'s next (line 46)
- else: set `InUse` into `false` (line 48)

Implementation - nxtRON: Implementation Details (4)

```
35 /*Find the first 1 bit set (FFS) after ID*/
36 int find_next(int ID):
37     long tmp = rotate_right(nxt_ary, TID)
38     return (ffs(tmp)+ID)%NUM_CORE;
```



- `find_next()`
 - `ffs()`:
 - `'__builtin_ctz'`
 - Provided by GCC
 - Time complexity: $O(1)$

Implementation - nxtRON: Conclusion

- Efficiency and Scalability:
 - Reduces thread selection overhead.
 - Improves system performance and scalability in multi-core environments.
- Limitations
 - Does not provide optimizations for space complexity.
 - Focuses on improving time complexity and reducing overhead without optimizing memory footprint.



Implementation - shRON: Design Overview (1)

- RON-Plock Algorithm:
 - Uses a **per-lock** `WaitArray[]`, which size increases linearly with the number of cores.
- ShRON introduction:
 - **Per-Process Shared** `WaitArray[]`: Reduces memory contention and overhead.
 - `WaitArray[]` Bitmap: Uses `atomic_long` to track cores waiting for the lock.
 - Lock Representation: Two bits represent a lock in `WaitArray`.
 - Capacity: Each `atomic_long` can track 32 locks in our system.

Implementation - shRON: Design Overview (2)

- Observation
 - Processor-Native Types:
 - Use types like `atomic_long` for hardware support.
 - Performance Impact:
 - Multiple cores accessing the same `atomic_long` leads to lower performance.
 - Bitmap Approach:
 - shRON uses a bitmap so that a core waits on multiple locks, minimizing performance issues.


Implementation - shRON: Design Overview (3)

- Key Advantages of shRON
 - Improved Cache Utilization:
 - Optimized data access patterns enhance cache memory usage.
 - Scalability:
 - Dynamic bitmap resizing ensures efficient resource use regardless of active locks.
 - Space Complexity Reduction:
 - Reduces space complexity by $2/(\text{sizeof}(\text{atomic_long}))$ times compared to RON-Plock.

Implementation - shRON: Implementation Details (1)

- Data Structure:

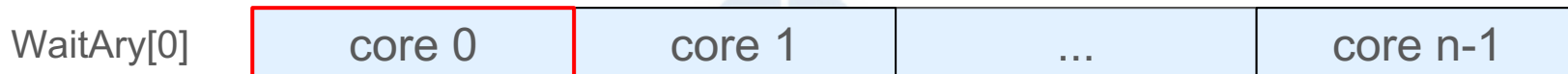
```
1  /*per-process*/
2  int TSP_ID_ARRAY[NUM_CORE];
3  /*PEND_BIT and LOCK_BIT denote one lock,
4  each row in WaitAry represents 32 locks*/
5  atomic_long WaitAry[NUM_LOCK/32][NUM_CORE];
```



- **WaitArray[]**: Array of *atomic_long* structures indicating each core's wait status.
 - LOCK_BIT: Shows lock status (ACQUIRE (ACQ) or RELEASE (REL)).
 - PEND_BIT: Shows waiting status (PENDING (PEN) or IDLING (IDL)).

Implementation - shRON: Implementation Details (2)

- Assume there are 32 locks to create and the #NUM_CORE is "n"
 - We will first malloc the WaitAry size as 1*n



- We zoom into the first core: WaitAry[0][0], which is a atomic_long variable with 64 bits.

#bits	63	62	...	3	2	1	0
Meaning	PEND_BIT	LOCK_BIT		PEND_BIT	LOCK_BIT	PEND_BIT	LOCK_BIT
Lock	Lock 31		...	Lock 1		Lock 0	

Implementation - shRON: Implementation Details (3)

- Data Structure:

```
7  /*thread-local-storage*/
8  thread_local TSP_ID;
9  TSP_ID = TSP_ID_ARRAY[getcpu()];
10
11 /*per-lock structure */
12 struct shRON_t:
13     /*The lock is acquired or not*/
14     int InUse:1;
15     /*The index bit in long WaitAry(0~63)*/
16     int WaitID:6;
17     /*The column index of WaitAry*/
18     int ColID:25;
```

- *int InUse:1*

- if any thread is currently in the CS
- *InUse = 0*: no thread in CS

- *int WaitID:6*

- Six-bit integer representing the LOCK_BIT index. (0, 2, 4, 6,..., 62)
- The third lock will get the WaitID as 4

- *int ColID: 25*

- 25-bit integer denoting the column index in *WaitAry[ColID][#CORE]*.
- The 34th lock will get the ColID as 1.

Implementation - shRON: Implementation Details (4)

```
20  /*Denotes "WaitAry[ColID][TSP_ID]" as WaitPos*/
21  void spin_lock():
22      while(1):
23          set_bit(PEND_BIT, WaitPos);
24          /*bitvl() means the bit value*/
25          while(bitvl(LOCK_BIT, WaitPos)==REL
26              && InUse==1)
27              CPU_PAUSE();
28          if(bitvl(LOCK_BIT, WaitPos)==ACQ):
29              /* cmp_xchg set PEND_BIT as IDL,
30                 LOCK_BIT as REL*/
31              if(putDownLock(LOCK_BIT,WaitPos)):
32                  return;
33          if (cmp_xchg(&InUse, false, true)):
34              clr_bit(PEND_BIT, WaitPos);
35              return;
```

- spin_lock()
 - Thread Request
 - Set the PEND_BIT as PEND
 - Busy-Wait Loop
 - Busy Waiting until its turn to acquire the lock
 - Or there is no thread in CS
 - Get the lock
 - Set PEND_BIT as IDL
 - Set the LOCK_BIT as REL (line 31)

Implementation - shRON: Implementation Details (5)

```
37  /*Denotes "WaitAry[ColID][nxt]" as NxtWaitPos*/
38  void spin_unlock():
39      for(int i=1; i<NUM_CORE; i++):
40          int nxt = (TSP_ID+i)%NUM_CORE;
41          if(bitvl(PEND_BIT, NxtWaitPos)):
42              set_bit(LOCK_BIT, NxtWaitPos);
43              return;
44      InUse=0;
45      clr_bit(PEND_BIT, WaitPos);
```

- spin_unlock()
 - Like RON-Plock
 - Using a for-loop traverse
 - Visit the bit value of *WaitAry*
 - Check the PEND_BIT as the thread want the lock or not. (line 42)
 - Release the lock by set the next's LOCK_BIT as ACQ.

Implementation - shRON: Conclusion

- Efficiency and Scalability:
 - Reduces the space complexity of RON-Plock and nxtRON.
 - Improves system performance and scalability in multi-lock environments.
- Limitations
 - Despite these advantages, shRON does incur increased computational overhead due to the additional bit operations involved in its design.



Evaluation

Evaluation - Testing Environment

- Model name: AMD Ryzen Threadripper PRO 3995WX
- Number of Cores: 64-Core Processor
- Virtual core: 128 (virtual core per core is 2)
- Architecture: x86_64
- Compile Environment: gcc version 10.5.0
- Target: AMD Zen+ and x86 64-linux-gnu
- Thread model: POSIX
- Linux Version: 5.4.0-177-generic

Evaluation - Testing Program - Userspace

- **Userspace:**

- We analyze each lock method in a quantitative manner through a **controlled microbenchmark**
- We construct our locking algorithms under a **public library: LiTL**
 - [Library for Transparent Lock interposition](#)
 - LiTL is a library that allows executing a program based on Pthread mutex locks with another locking algorithm

Author : Hugo Guiroux <hugo.guiroux at gmail dot com>

Related Publication: *Multicore Locks: the Case is Not Closed Yet*, Hugo Guiroux, Renaud Lachaize, Vivien Quéma, USENIX ATC'16.

Operating System Lab, National Chung-Cheng University

Evaluation - Testing Program - Microbenchmarks

```
while(1) {
```

```
    spinlock()
```

```
    //critical section  
    for i = 0.. sizeof(sharedData)  
        sharedData[i] += 1;
```

```
    spinunlock()
```

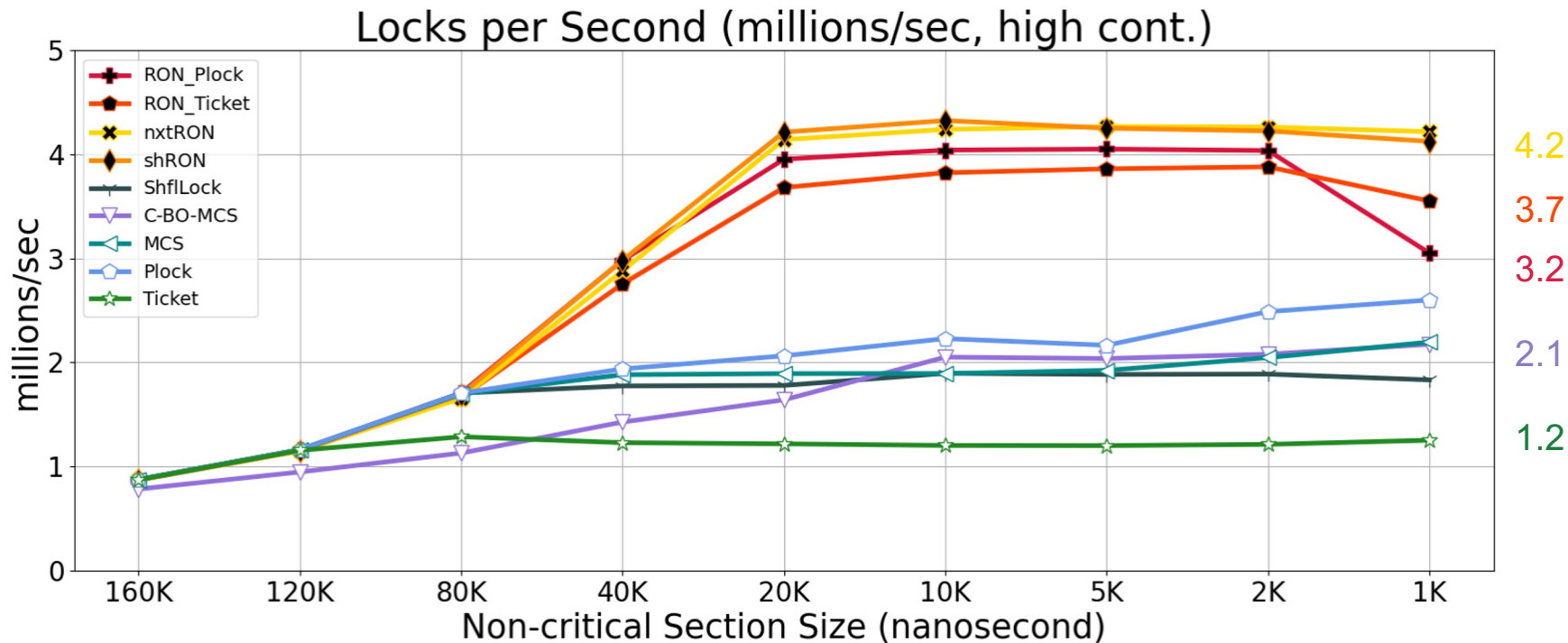
```
    //non-critical section  
    for_loop_sleep(nCS  $\pm$  15%);
```

```
}
```

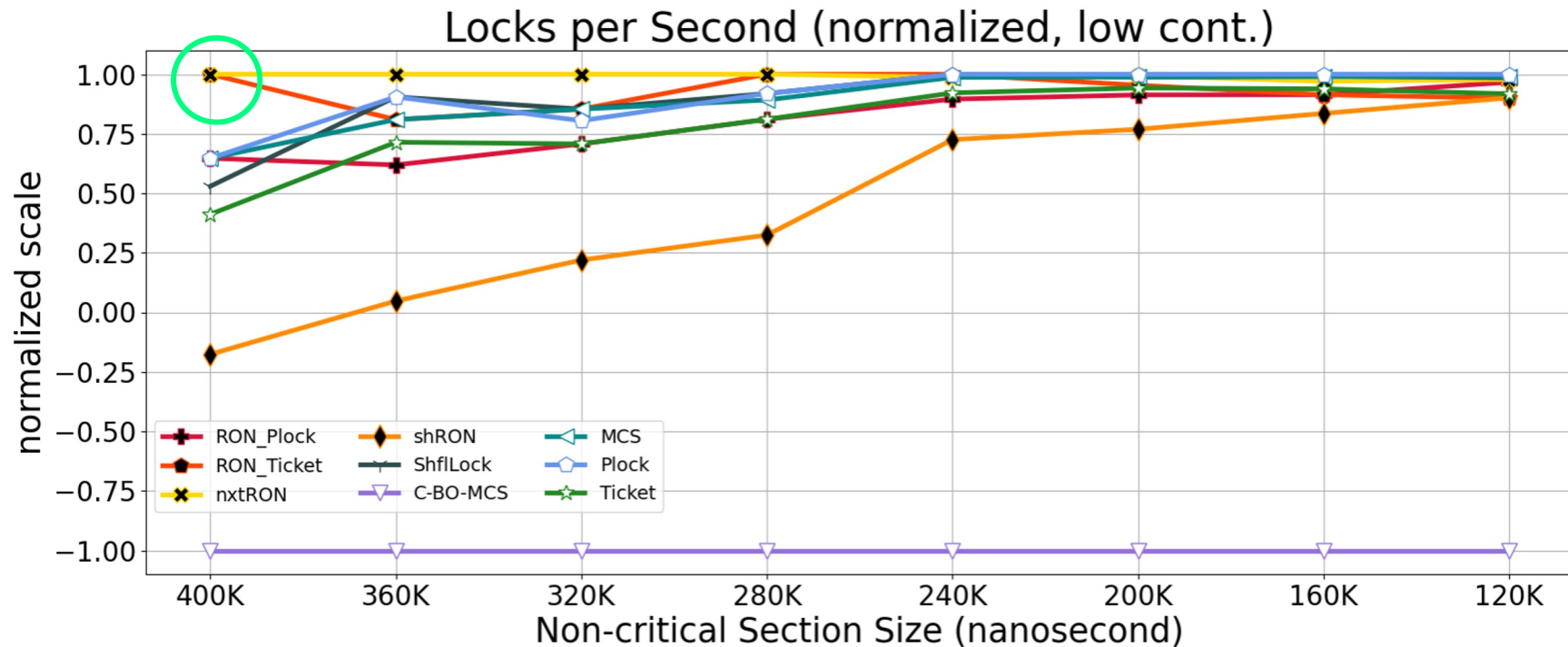
- nCS:

- Non-critical section / Remainder section
- It reflects the **contention** of the process
- The random variable is utilized to **simulate varying program loads**

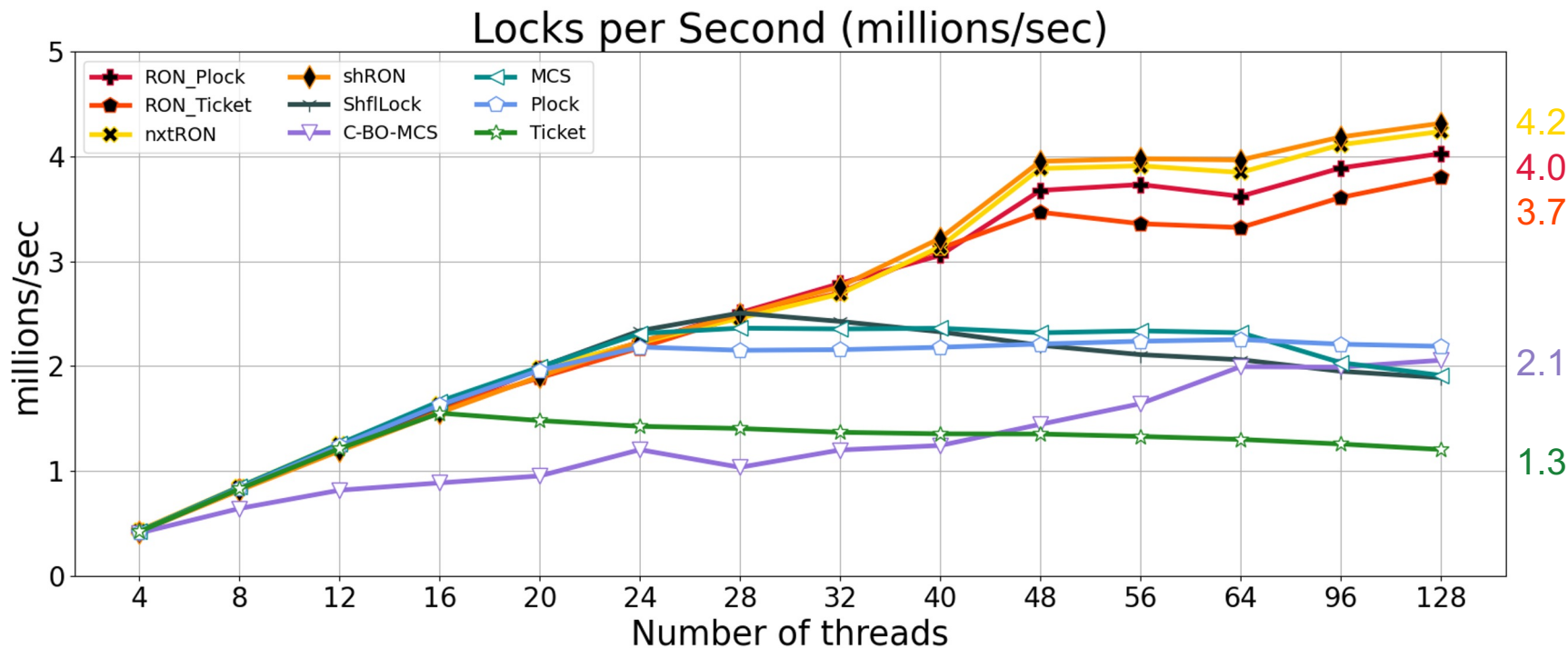
Evaluation - Throughput in High Contention



Evaluation - Normalized Throughput in Low Contention

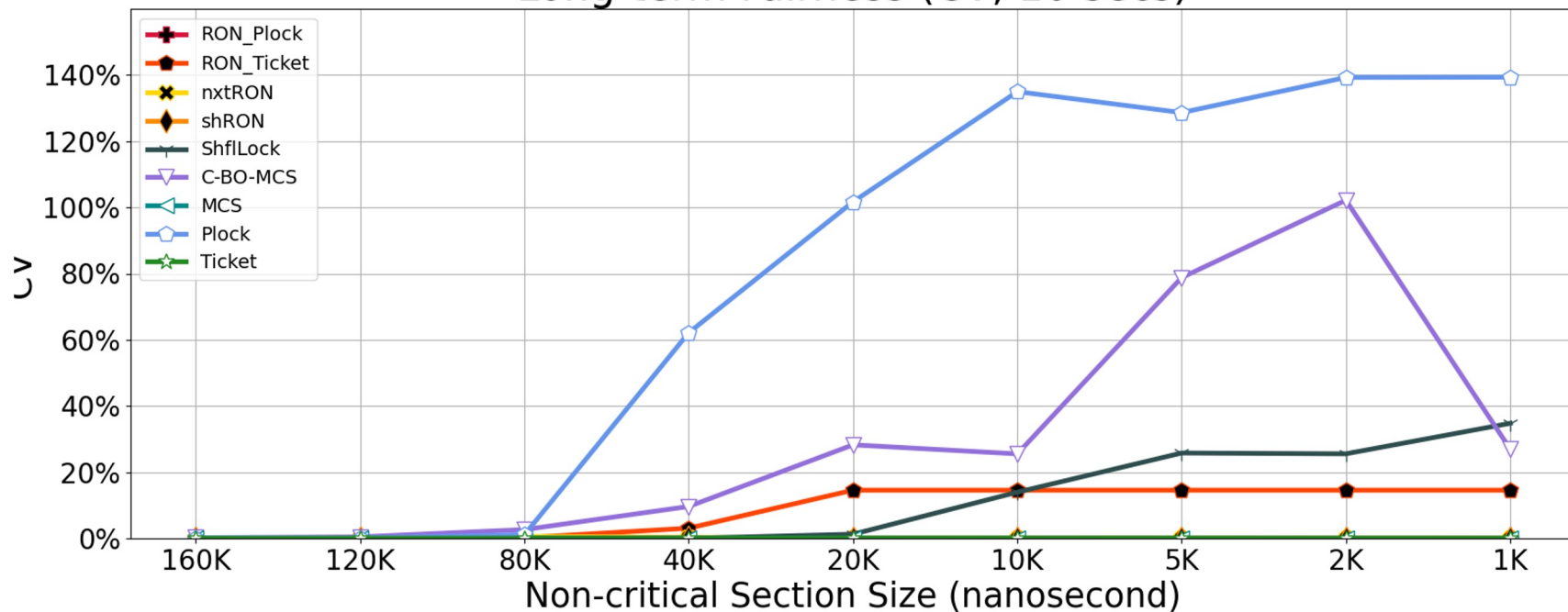


Evaluation - Scalability with RS=10000 ns

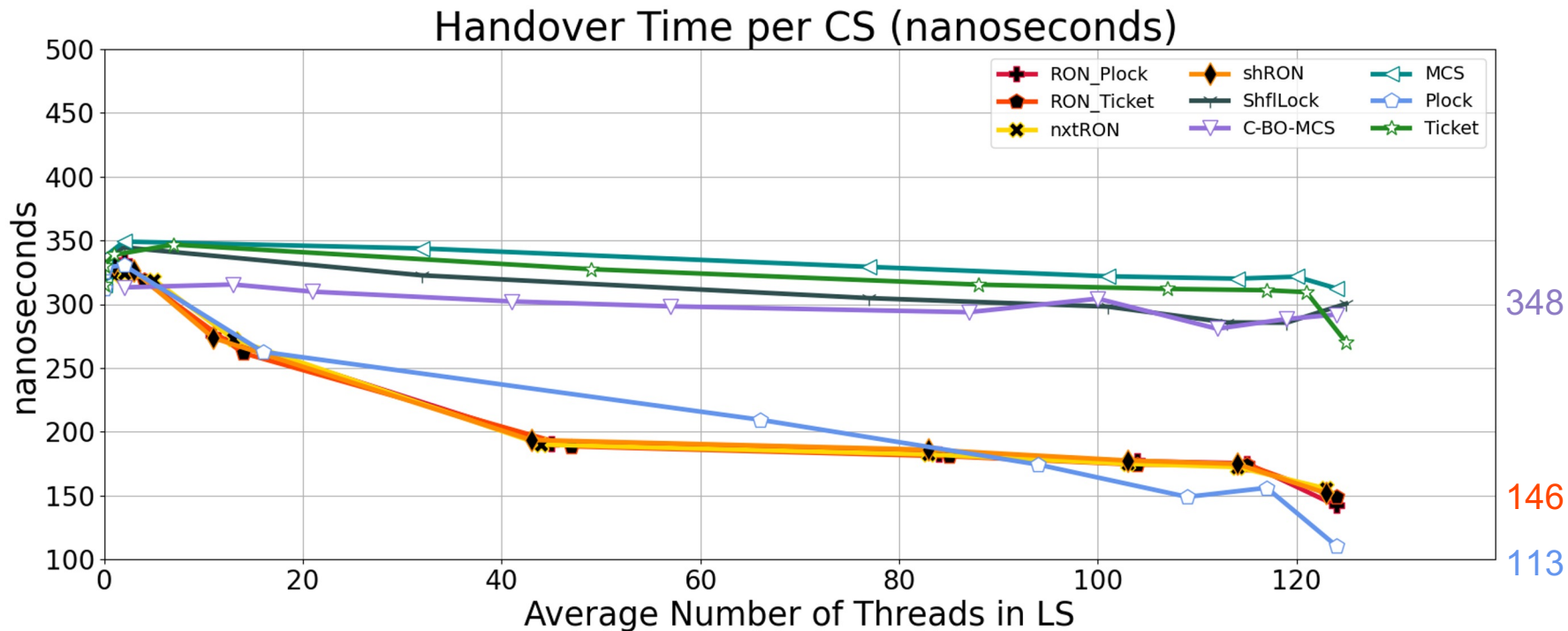


Evaluation - Long-Term Fairness

Long-term Fairness (CV, 10 secs)

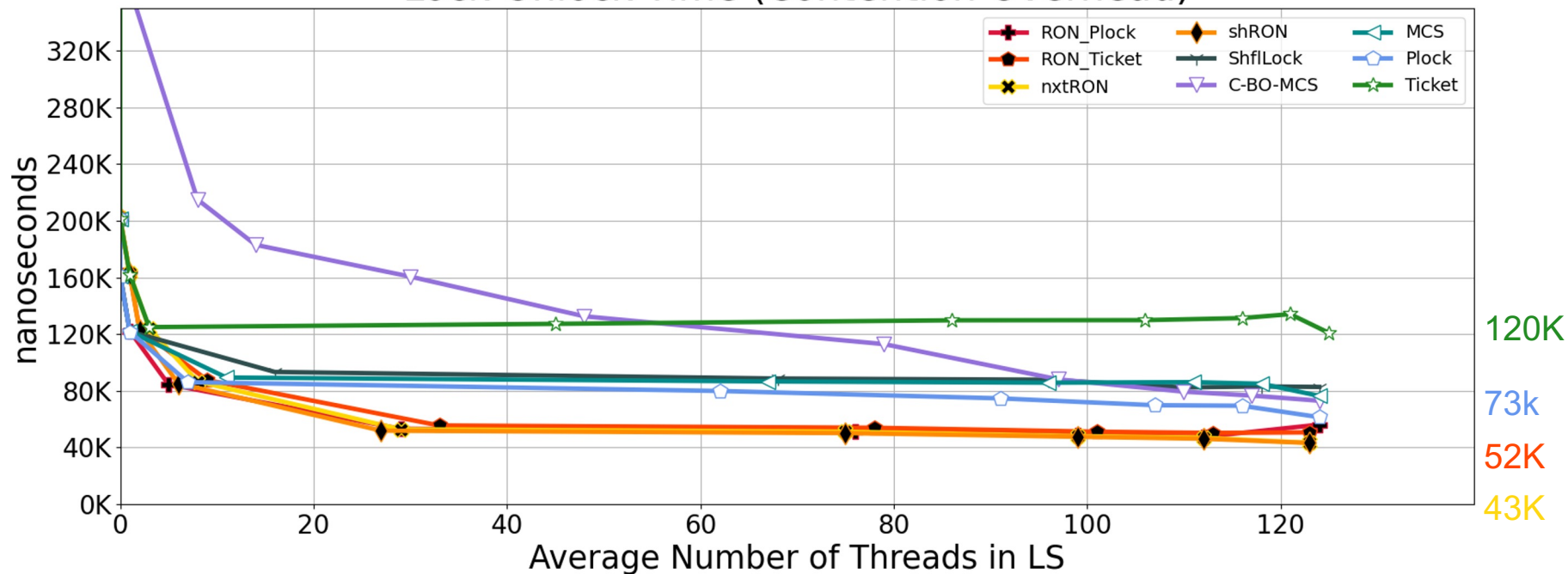


Evaluation - Handover Time per CS (ns)

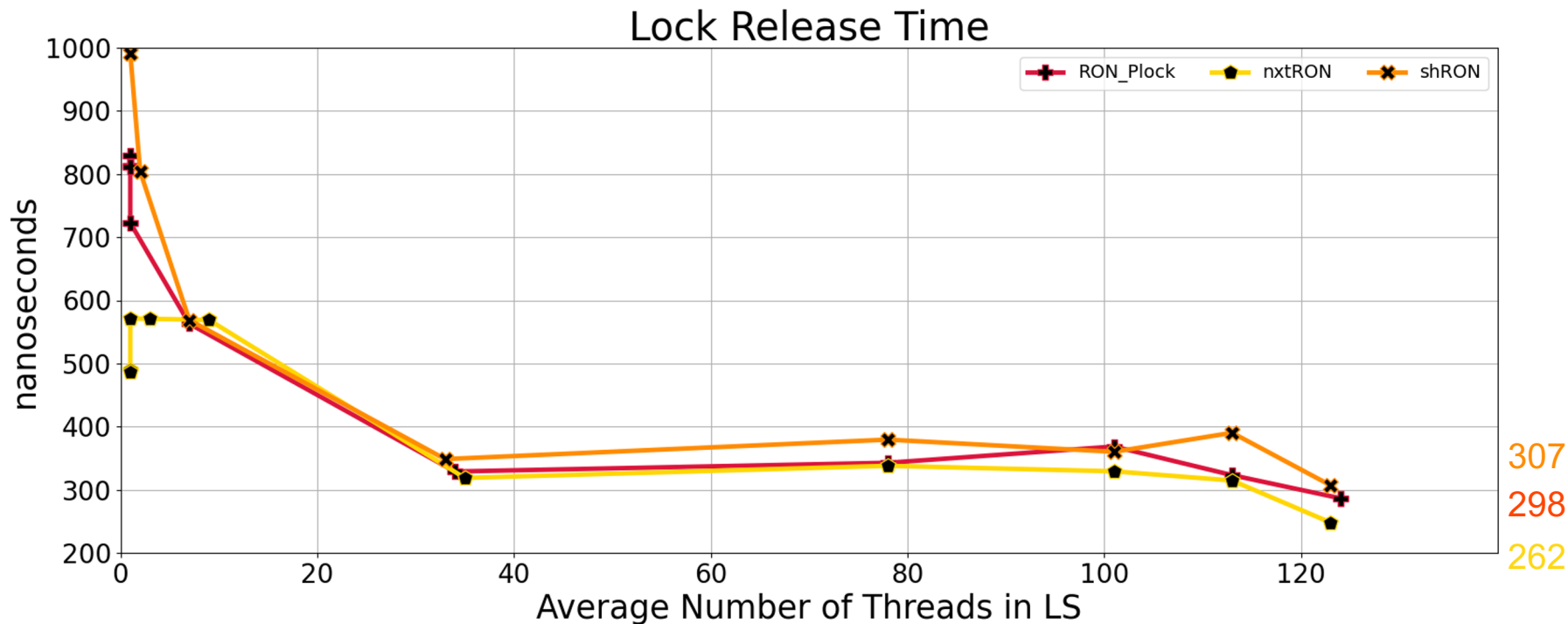


Evaluation - Contention Cost

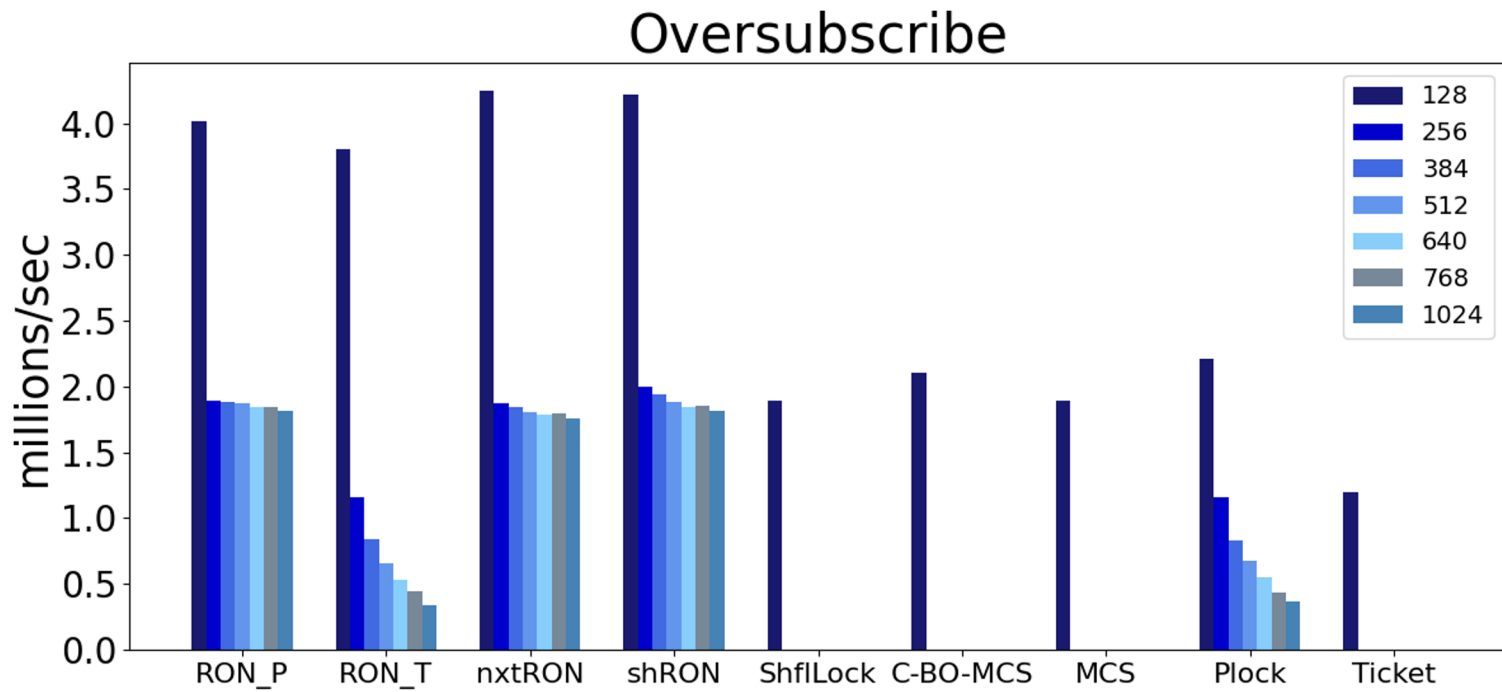
Lock-Unlock Time (Contention Overhead)



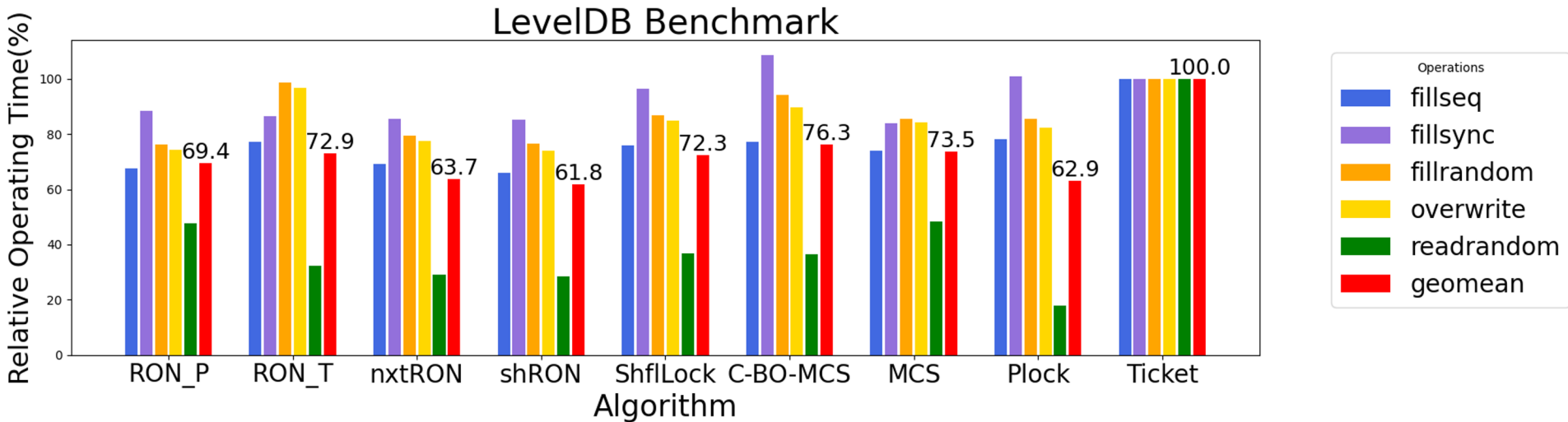
Evaluation - Lock Release Time (ns)



Evaluation - Oversubscribed



Evaluation - Google - leveldb (ms)



Evaluation - Testing Program - Linux Kernel

- Linux kernel locking algorithm
 - qspinlock
 - Fast path:
 - TTAS In low contention scenarios (few competitors), the lock is directly acquired through TTAS, like plock
 - Slow path:
 - MCS In high contention scenarios (many competitors), the slow path is entered

```
107 static __always_inline void queued_spin_lock(struct qspinlock *lock)
108 {
109     int val = 0;
110
111     if (likely(atomic_try_cmpxchg_acquire(&lock->val, &val, _Q_LOCKED_VAL)))
112         return;
113
114     queued_spin_lock_slowpath(lock, val);
115 }
```

Evaluation - Testing Program - Linux Kernel

- Linux kernel locking algorithm
 - By rewriting these two functions, we implement nxtRON in the Linux kernel.

```
80 extern void _spin_unlock(struct qspinlock *spin);
81 static __always_inline void queued_spin_lock(struct qspinlock *lock)
82 {
83     /*
84      * int val = 0;
85      *
86      * if (likely(atomic_try_cmpxchg_acquire(&lock->val, &val, _Q_LOCKED_VAL)))
87      *     return;
88      *
89      * queued_spin_lock_slowpath(lock, val);
90      */
91     _spin_lock(lock);
92 }
93 #endif
94
95 #ifndef queued_spin_unlock
96 /**
97  * queued_spin_unlock - release a queued spinlock
98  * @lock : Pointer to queued spinlock structure
99  */
100 static __always_inline void queued_spin_unlock(struct qspinlock *lock)
101 {
102     //smp_store_release(&lock->locked, 0);
103     _spin_unlock(lock);
104 }
105 #endif
```

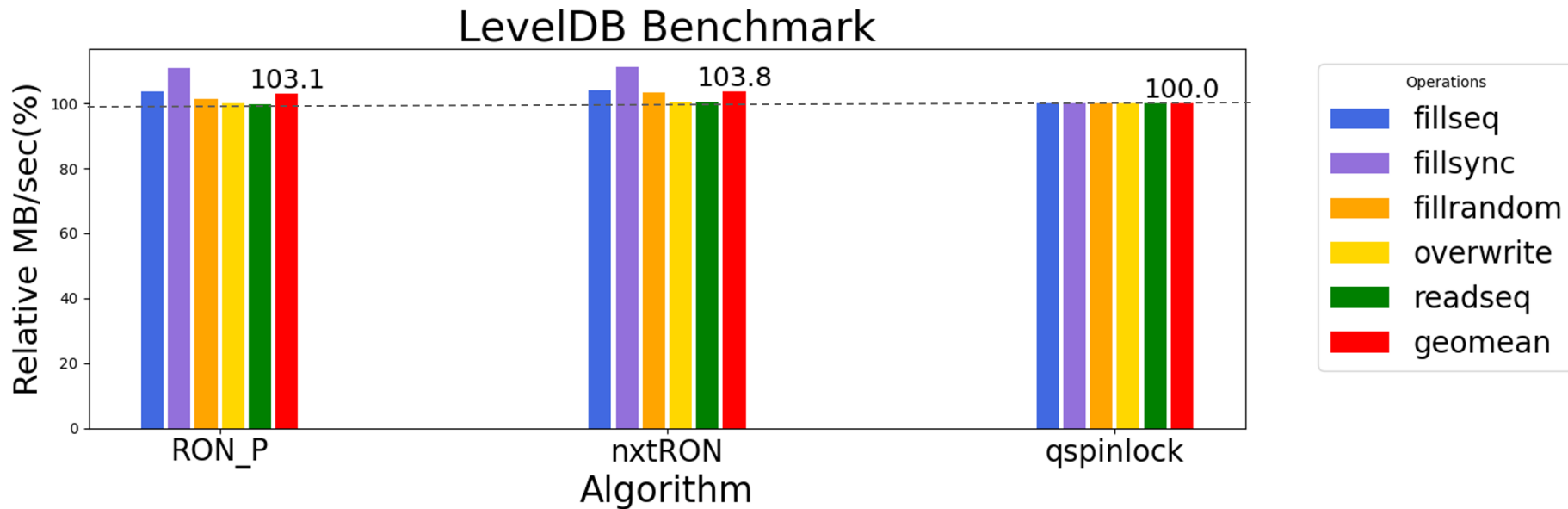

Testing Program - mmap

```
1  function thread()  
2      for i from 1 to 64  
3          addr = mmap(NULL, 1024*12, PROT_READ,  
4                      MAP_ANONYMOUS|MAP_SHARED, -1, 0)  
5          if addr == MAP_FAILED  
6              print "error"  
7          if addr != MAP_FAILED  
8              munmap(addr, 1024*12)
```

Linux Kernel - mmap (ms)

	mmap (ms)	clone (ms)	mprotect (ms)	munmap (ms)	geomean
qspinlock	1883.0	276.6	911.6	87.4	451.4
RON	2050.9	259.3	637.3	37.9	336.6
nxtRON	1961.8	280.7	609.0	37.9	335.7

Linux Kernel - leveldb (MB/sec)





Conclusion

Conclusion

- **nxtRON (next-RON):**
 - Uses bitmap to track waiting threads, reducing time complexity of RON.
 - Cuts lock acquisition time by up to 20%, and outperforms in Linux Kernel
- **shRON (shared-RON):**
 - Implements shared data structures, reducing memory and cache contention.
 - Enhance throughput by about 25% in high contention.
- **Performance Evaluations:**
 - Outperform existing solutions (RON-Plock, RON-ticket) by 10% to 15%.
 - Enhance system throughput by up to 20% across various operational modes.

Conclusion - Future Work

- Combine the two algorithms
- Integrate the combined algorithm into Linux kernel for broader optimization.
- Explore impact on energy efficiency and thermal management.

