SOURCE CODE

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

RON: One-Way Circular Shortest Routing to Achieve Efficient and Bounded-waiting Spinlocks

Shiwu Lo, Han-Ting Lin, Yao-Hong Xie, Chao-Ting Lin, Yu-Hsueh Fang, **Ching-Shen Lin**, Ching-Chun Huang, Kam Yiu Lam, Yuan-Hao Chang

Lock-unlock problems on NUMA machines

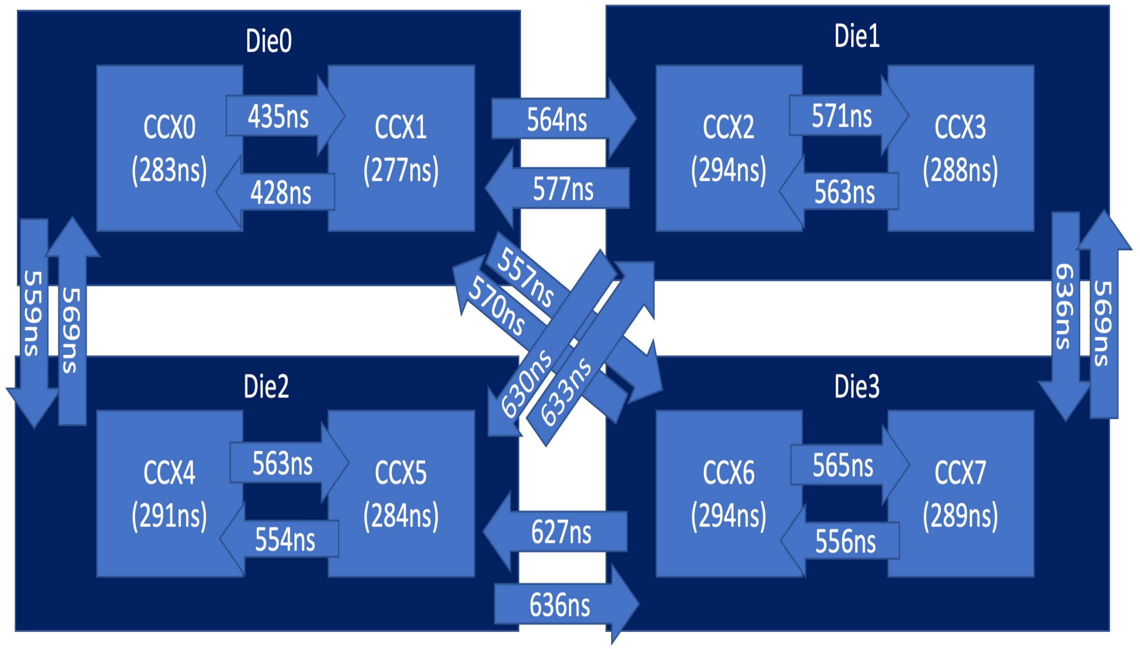
- Shared data among tasks is typically synchronized using critical sections to maintain data integrity.
- Accessing shared data involves transferring data between different CPU cores.
- The order in which threads lock-unlock the data corresponds to the sequence of data transmission across cores.

Background and Motivations
Multi-CPU NUMA

- Target: To minimize data transfer across CPUs
- Group the threads based on the CPUs they belong to and enable them to enter the critical section in batches.
- ***Grouping-based algorithms*** can lead to a trade-off between fairness and performance

Background and Motivations
Single-CPU NUMA (1/2)

1. Target: Minimize the data transfer between cores.
2. There is greater diversity in the interconnect speeds between cores.
3. For example:
   - Die1➔Die0 (577ns)
   - Die1➔Die2 (636ns)
   - Die1➔Die3 (630ns)
   - The cost of crossing boundaries differs.
4. As a result, "how to allow threads on different cores to access shared data" is more like a ***path planning problem***.



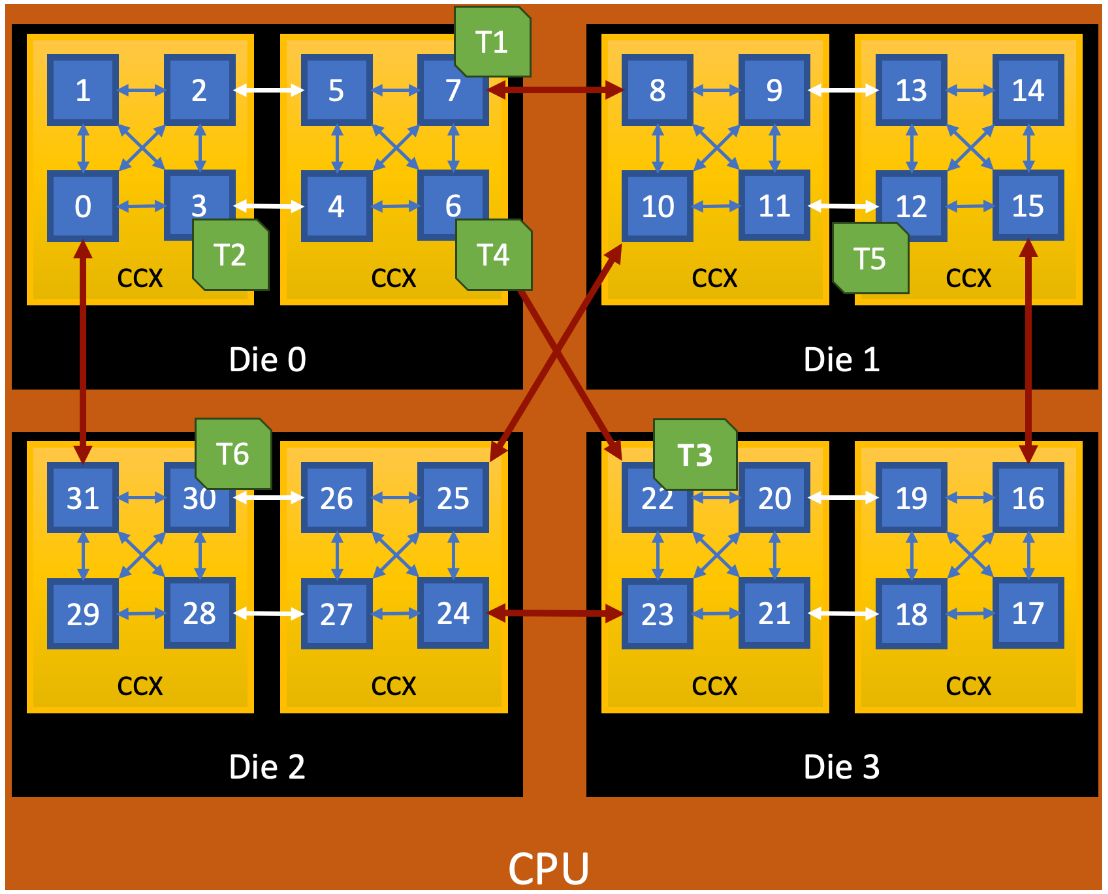
Background and Motivations Single-CPU NUMA (2/2)

1. Some cores have a higher chance of acquiring the lock based on their proximity to the core holding the lock or their higher execution frequency.
2. To ensure fairness between cores, specific mechanisms are indeed required.

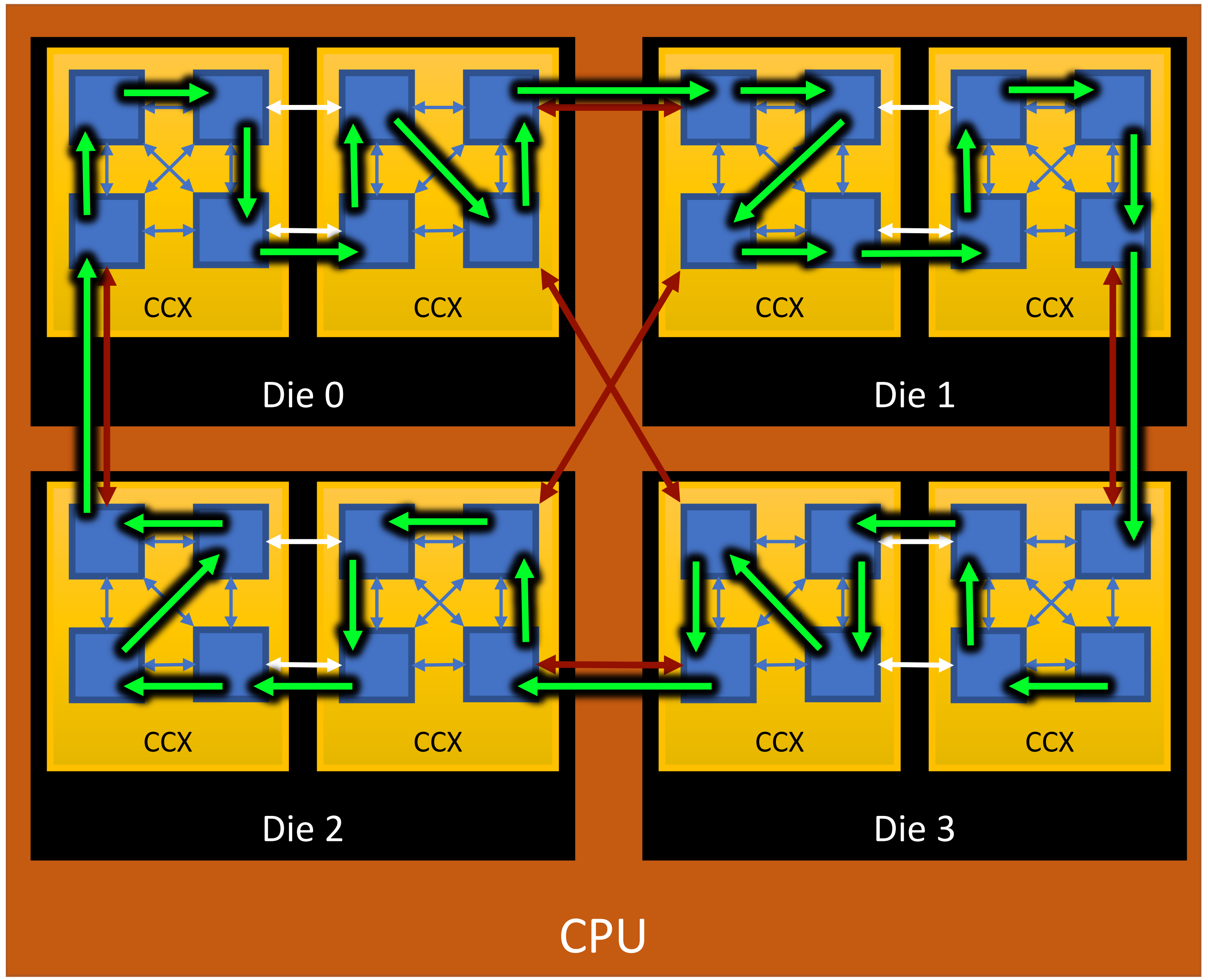
RON

The basic idea of RON

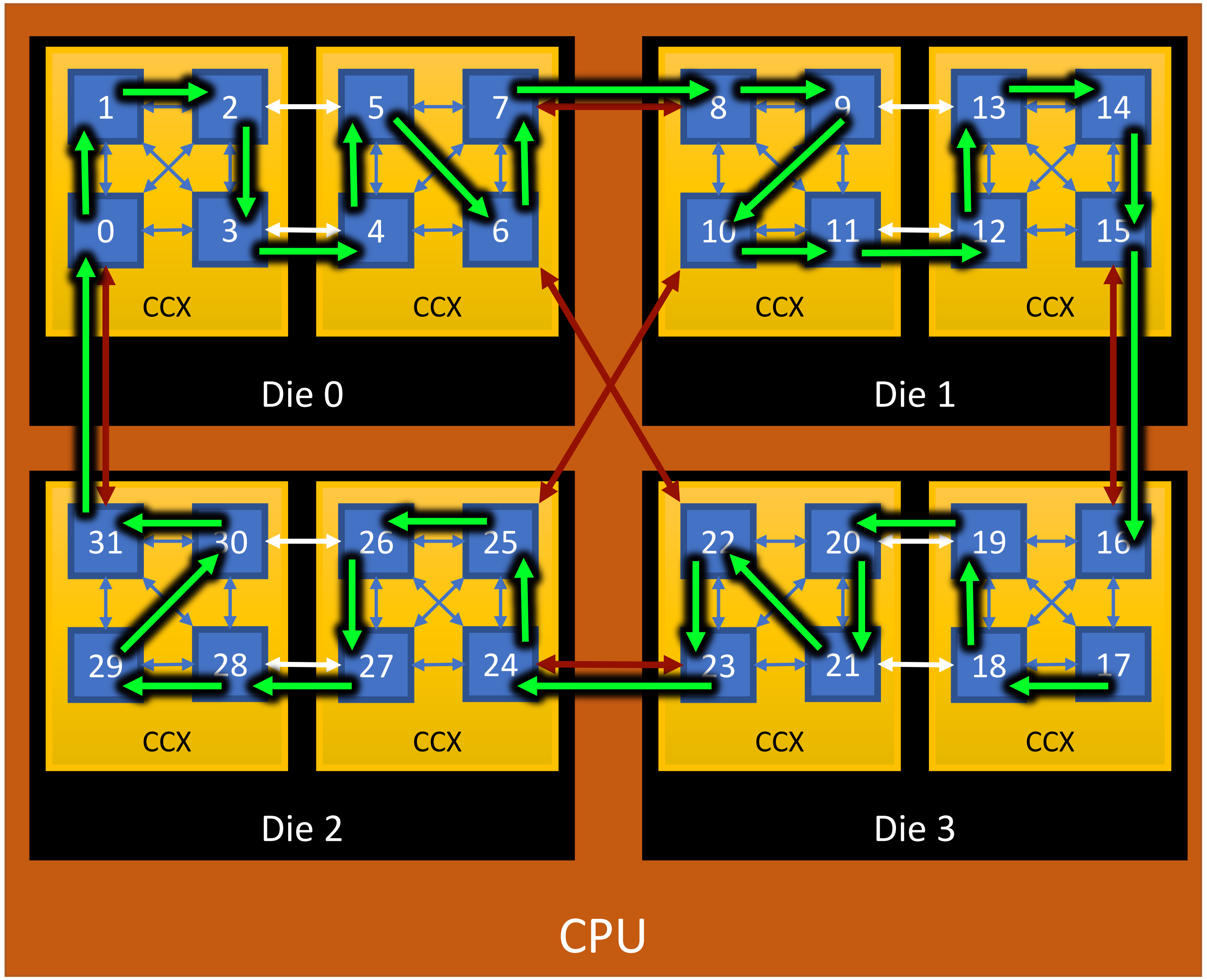
- Assume T1 holds the lock. When T1 leaves the CS, a spinlock algorithm should “find a thread to enter CS.”
- “Find a core to enter CS” is the “traveling salesman problem” (TSP), in terms of minimizing handover costs.
- RON is an efficient spinlock algorithm for solving TSP online.



Step 1:
Finding the shortest circular path (offline)



Step 2:
Give each core a TSP_ORDER (offline)

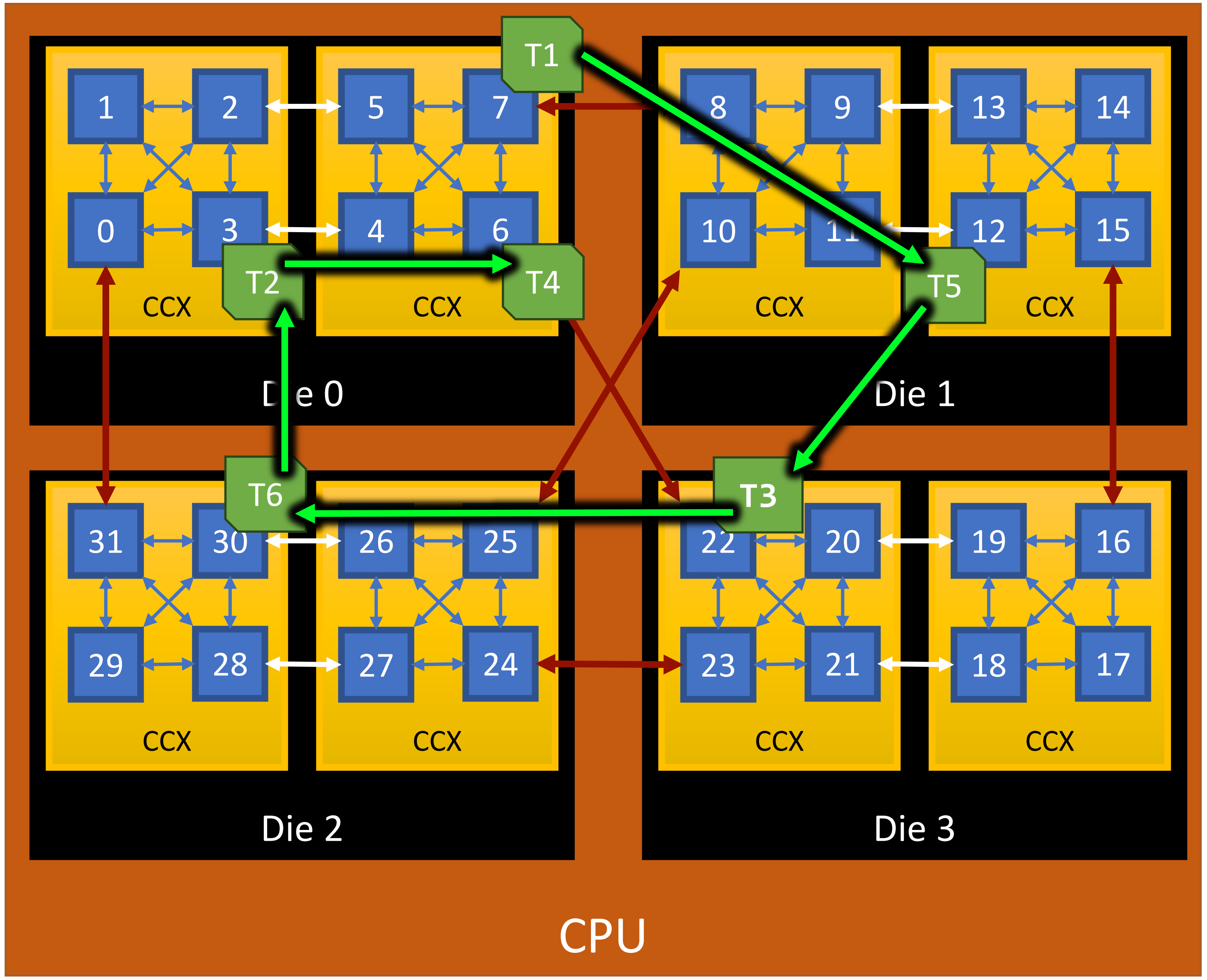


Step 3:
Arrange the order of entering the critical section based on the TSP_ORDER (online)

- RON: T1➔T5➔T3➔T6➔T2➔T4
- FIFO: T1➔T2➔T3➔T4➔T5➔T6

Note:
In this example, a thread can wait for a maximum of 31 threads. In terms of fairness, it satisfies bounded waiting.

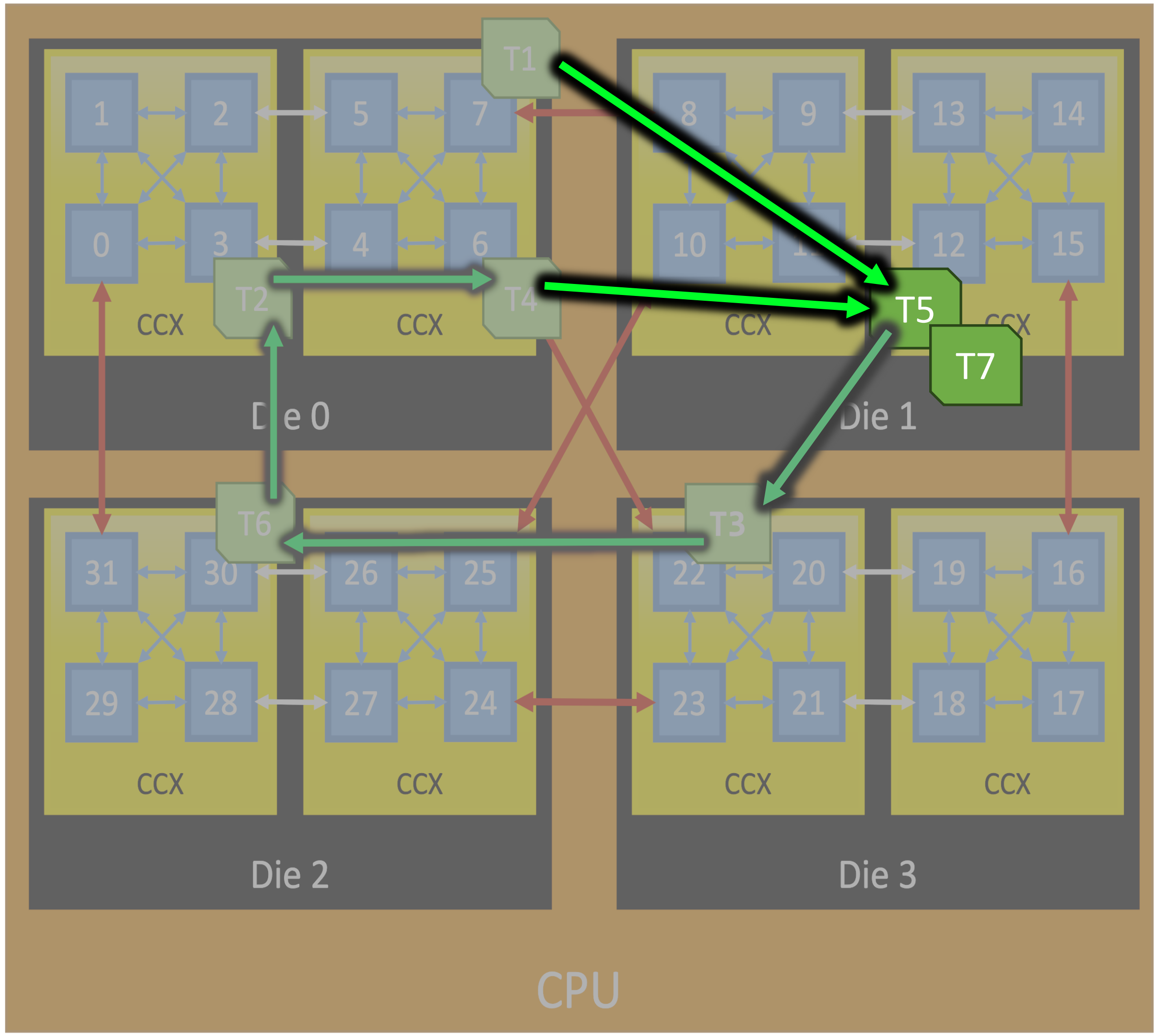
In the worst-case, each core has one thread waiting to enter the critical section.



RON + Oversubscription

Oversubscription
a trivial solution

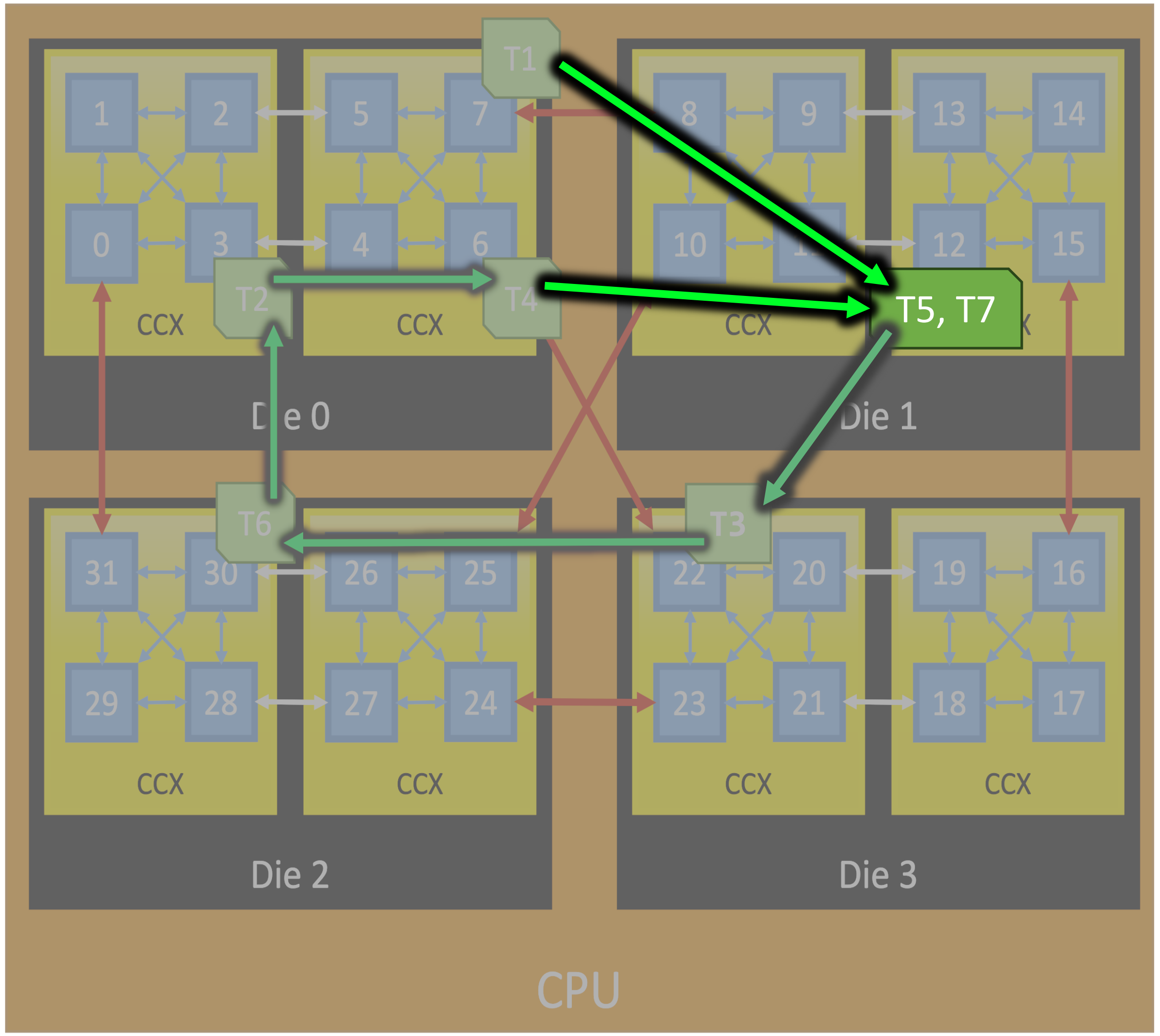
- Each thread has a lock entity and inserts it into the wait list of RON.
- Thread T7 requests to enter the critical section after thread T6.
- Since thread T7 and thread T5 are on the same core, only one of they can be "running" at a time.
- When T1 attempts to transfer the lock to T5, **T5 may be scheduled out**. That is the running thread on the core is T7.



Oversubscription

Threads on the same core share the same lock

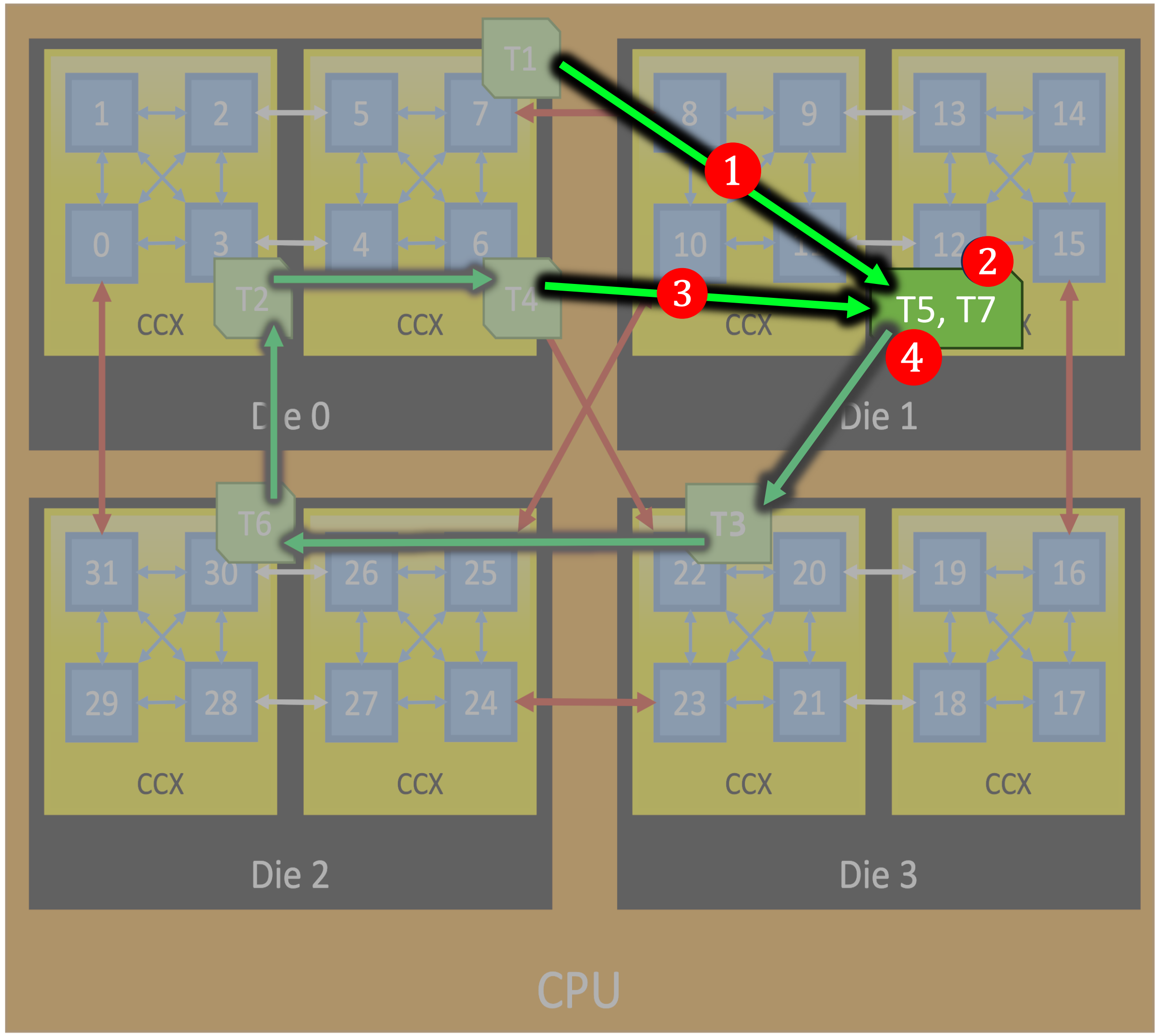
- Instead of passing the lock to the next thread based on TSP_ORDER, we'll now pass the lock to the next "**core**" based on TSP_ORDER.
- Threads (T5 and T7) on the same core compete for the "lock of core 12" using traditional spinlock algorithms,
- such as raw spinlock or ticket lock.



Oversubscription

Example of "RON + Plock"

1. When "T1" on "core 7" pass the lock to "core 12", the thread currently running on "core 12" will get the lock.
2. Due to "T7" currently running on "Core 12", as a result, "7" will successfully acquire the lock.
3. In the next round, "T4" will pass the lock to "core 12".
4. "T5" will successfully acquire the lock in this round.



Performance evaluation

Microbenchmarks - Quantitative Analysis

App-level benchmarks

Evaluation platforms

- CPU: AMD 2990WX with SMT
  - 32 physical cores
  - 64 SMT-cores
- Ubuntu 20.04
  - Linux kernel 5.4
  - gcc-9.3 with with the optimization parameter -march=znver1 -O3
- Temperature
  - Lower temperatures generally result in higher CPU performance.
  - To ensure accuracy in performance comparisons, all evaluations should start when the CPU temperature drops to 40 degrees.

Algorithms

We compared RON with the following algorithms.

- ShlfLock
    - The 'shuffler' can group threads belonging to the same node together and execute them consecutively.
- C-BO-MCS
    - If a core neighbors to the core that obtains the C-BO-MCS lock, it has a higher priority to enter the CS.
- Plock
    - test-test-and-set
- Ticket lock
    - The thread waits until its ticket number matches the system's grant number.

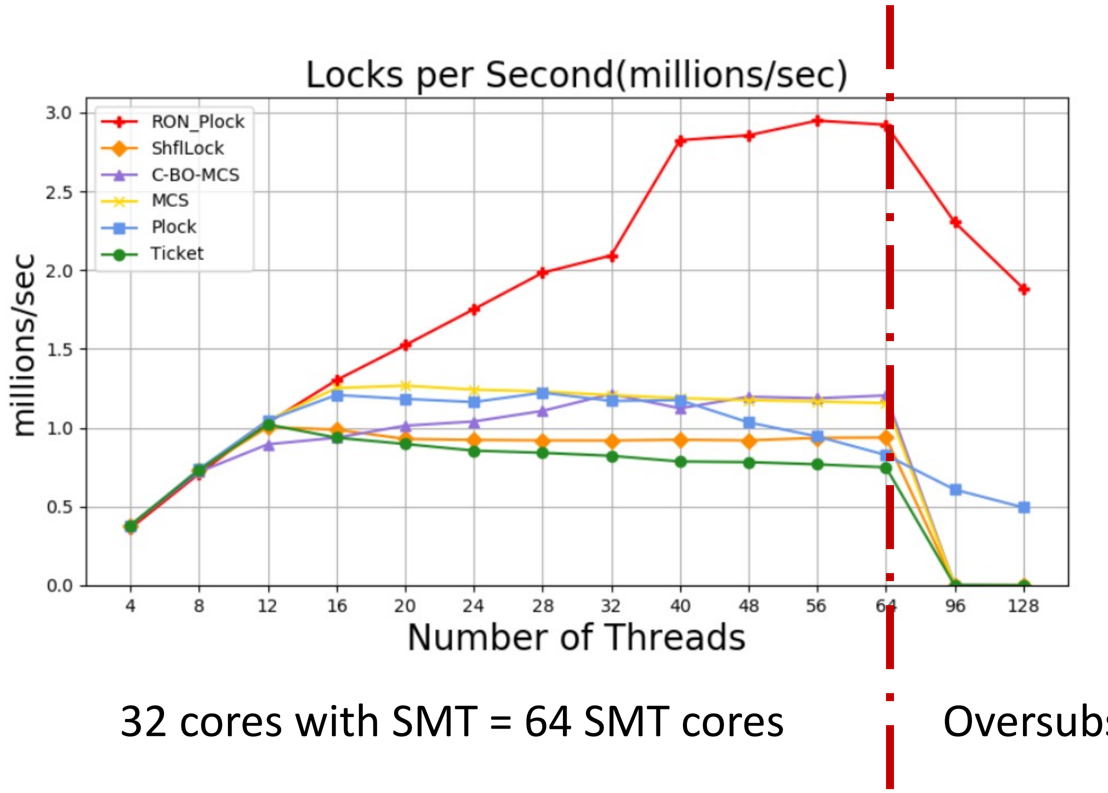
microbenchmarks

- Higher contention leads to more threads waiting at the spinlock() function.
- `sizeof(sharedData)` represents the size of the shared data that the thread accesses within the critical section.
- Higher values of nCS correspond to lower access frequency of `sharedData`, indicating lower contention.

```
while(1) {
    spinlock()

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

    spinunlock()

    //non-critical section
    for_loop_sleep(nCS ± 15%);
}
```

Microbenchmarks
Scalability

- RON has a clear advantage in high contention and oversubscription scenarios.



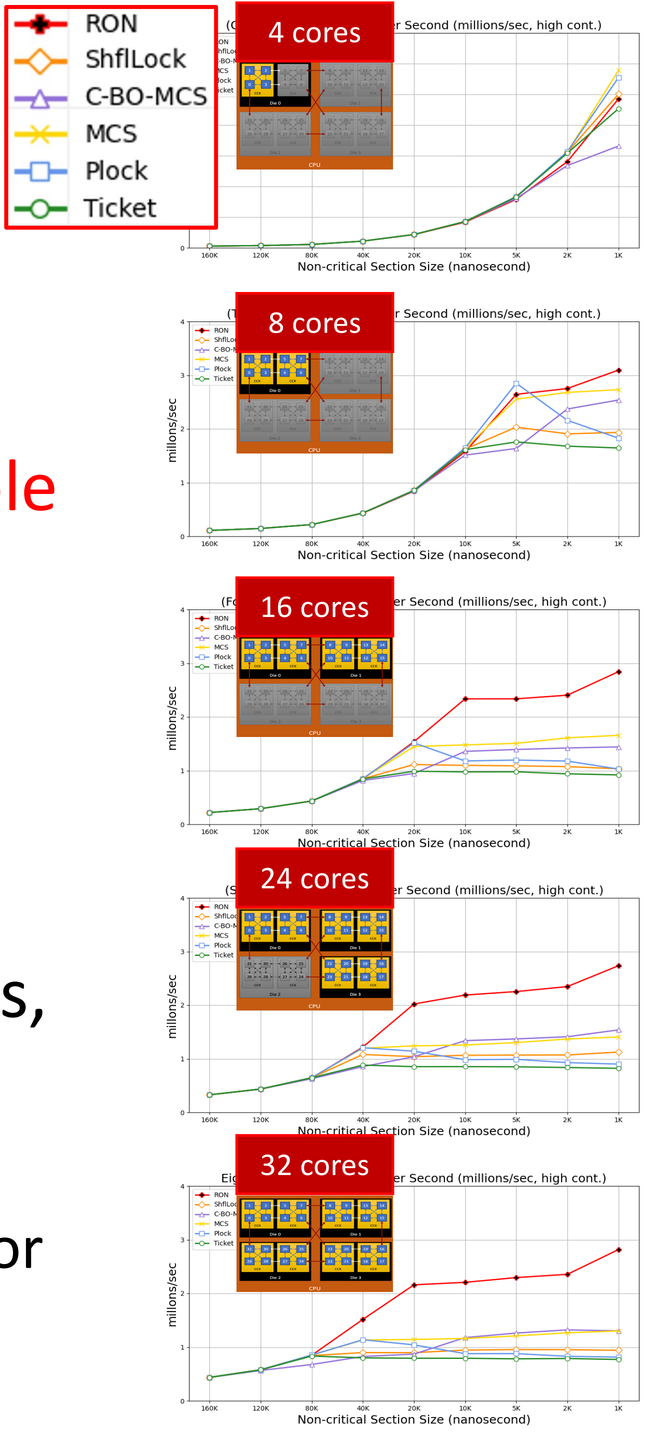
32 cores with SMT = 64 SMT cores

Oversubscription

Microbenchmarks

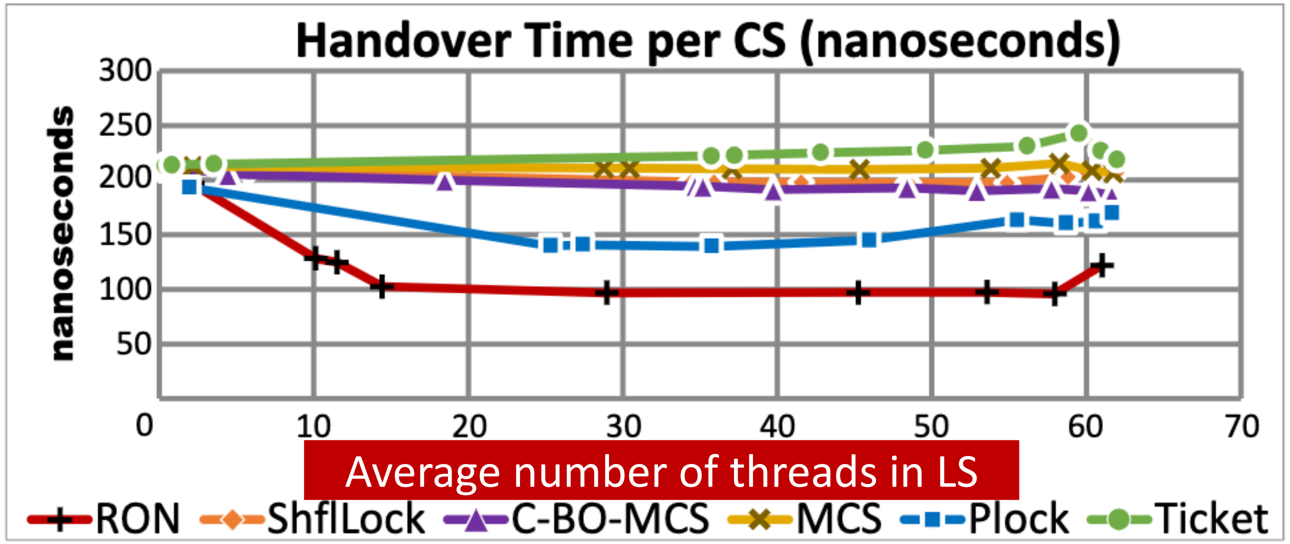
Assign apps to adjacent cores

- On multicore processors, it is possible to run multiple applications simultaneously.
  - Example: 8 cores for the database, 24 cores for the app server.
- When applications run on a subset of adjacent cores, RON performs well.
  - RON is a suitable choice when using more than 8 cores for an application.

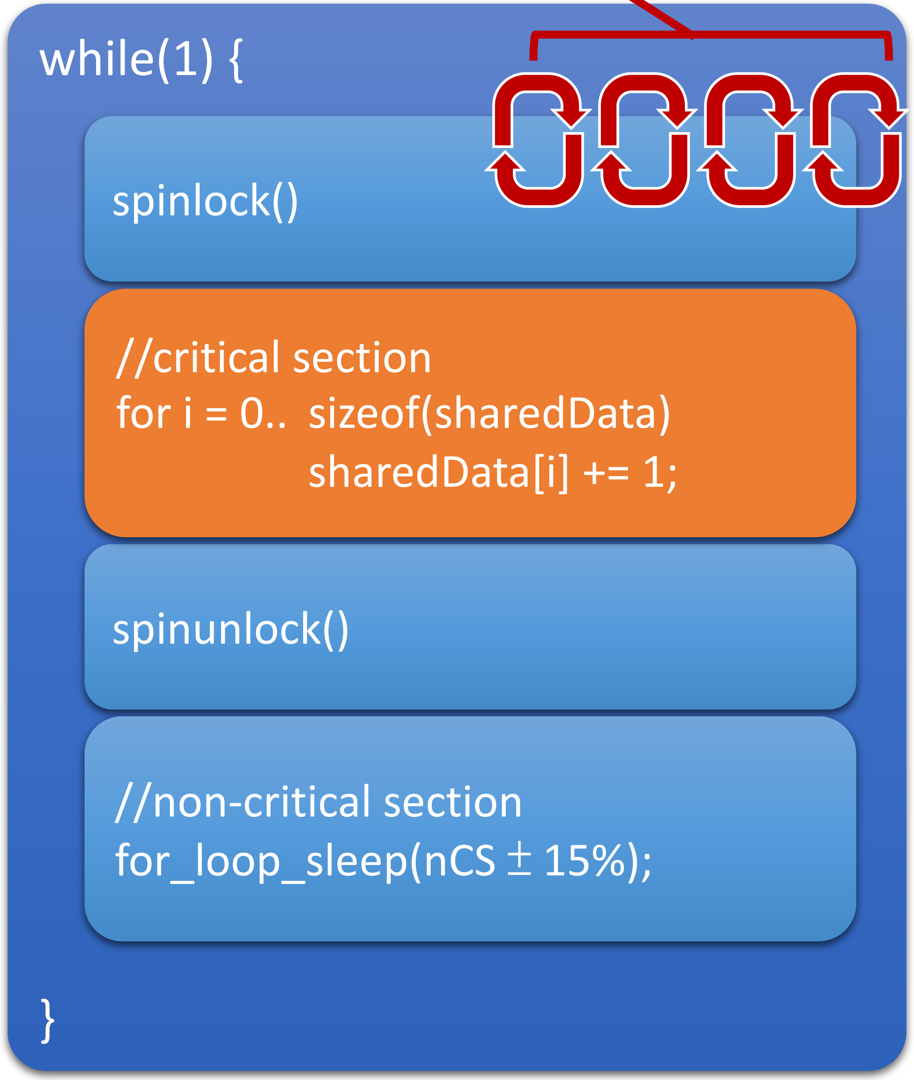


Handover time

- More threads waiting, more advantages for an algorithm with scheduled spinlocks.
- Under high load, RON has the lowest handover time.



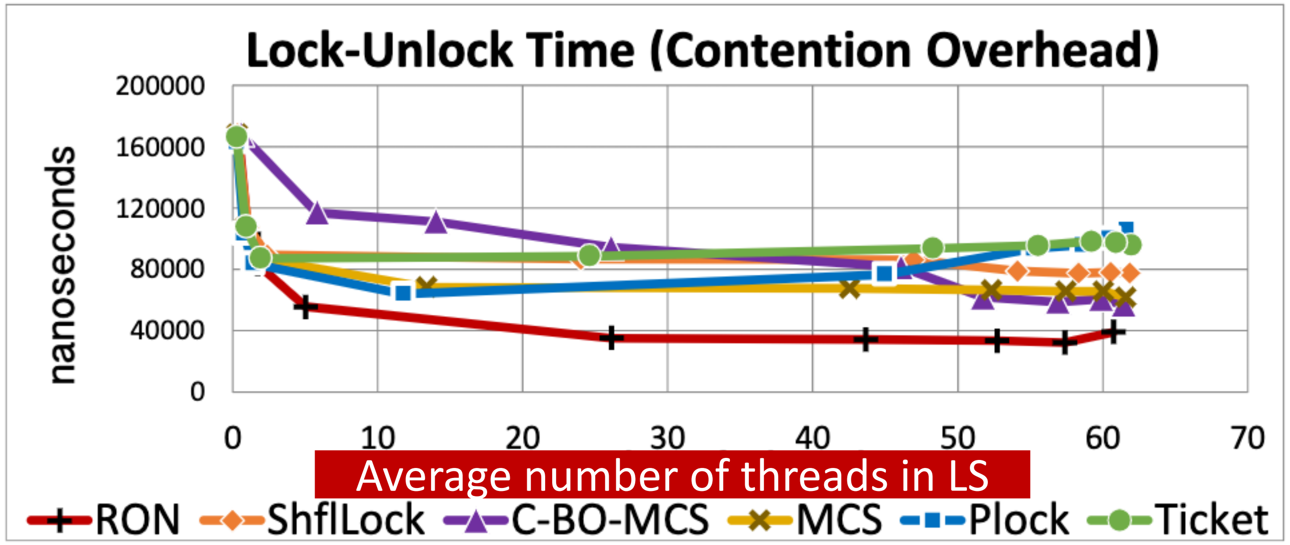
Average number of threads in LS

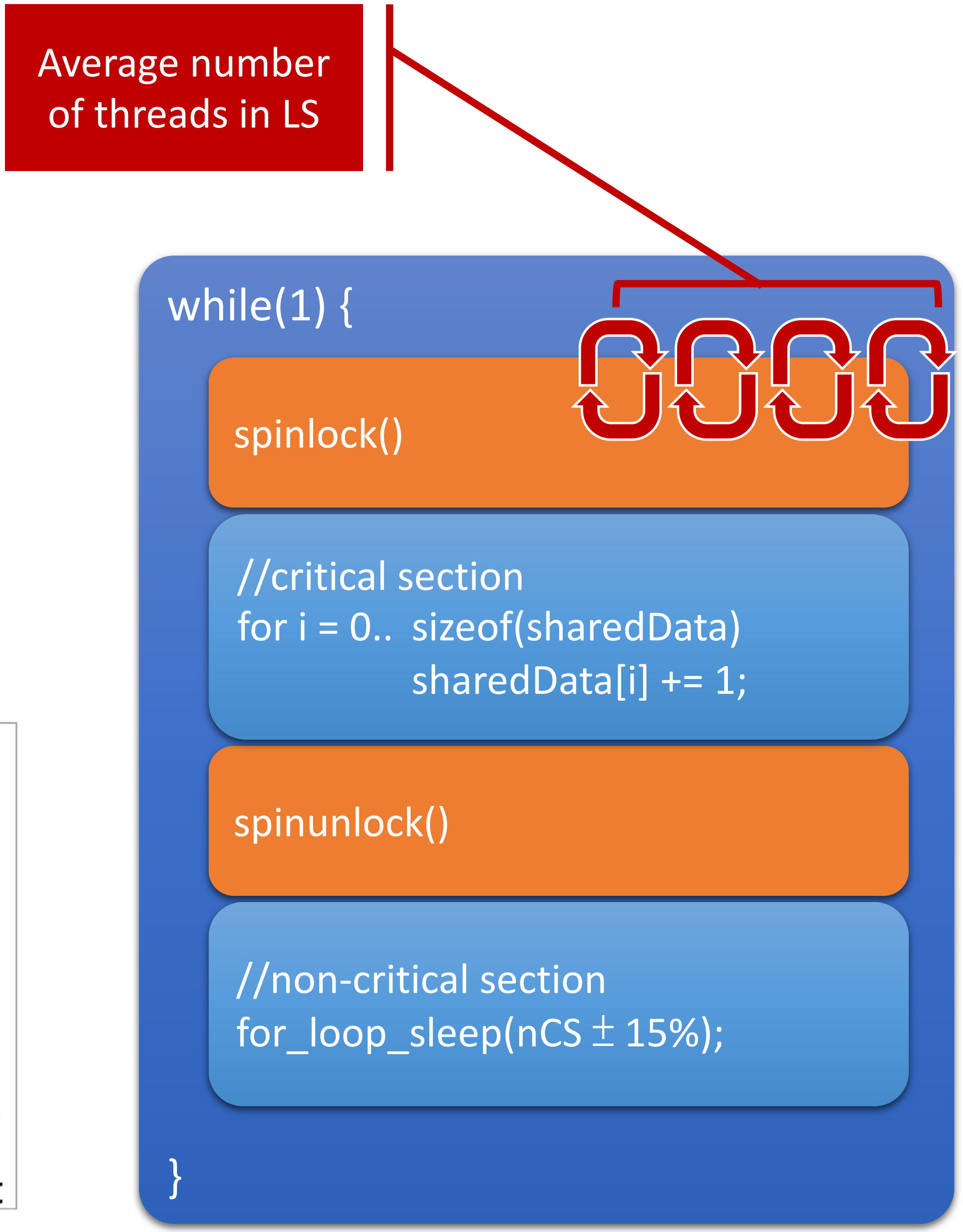


```
while(1) {
    spinlock()
    //critical section
    for i = 0..  sizeof(sharedData)
                 sharedData[i] += 1;
    spinunlock()
    //non-critical section
    for_loop_sleep(nCS ± 15%);
}
```

Lock-Unlock Time

- Lock-unlock time depends on algorithm complexity and atomic operation cost.
- The cost of atomic operations includes
  - cache coherency protocols and
  - executing data operations.



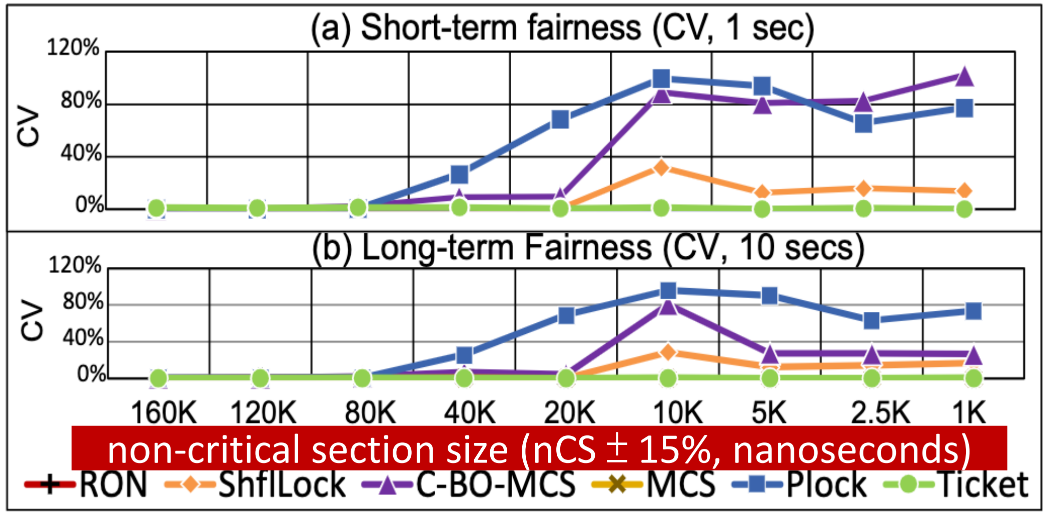


```
while(1) {
    spinlock()
    //critical section
    for i = 0.. sizeof(sharedData)
                sharedData[i] += 1;
    spinunlock()
    //non-critical section
    for_loop_sleep(nCS ± 15%);
}
```

Fairness

Performance anomaly prevention

- nCS ↓ ⟹ contention ↑
- Coefficient of variation (CV) ↓ ⟹ fairness ↑
- RON, MCS, Ticket is a near-perfect fairness algorithm.
- For certain algorithms, efficiency can be improved by sacrificing performance.
  - ShflLock
  - C-BO-MCS



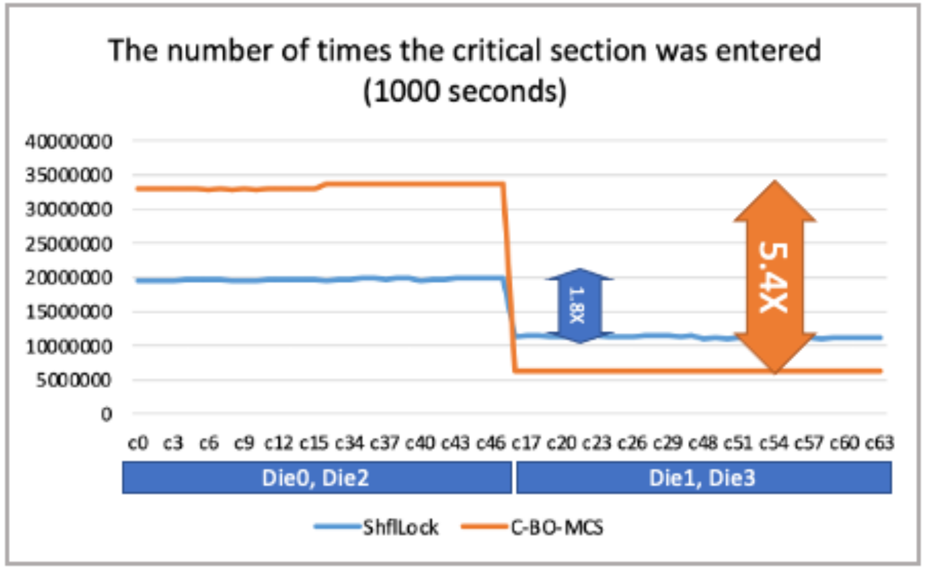
```
while(1) {
    spinlock()

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

    spinunlock()

    //non-critical section
    for_loop_sleep(nCS ± 15%);
}
```

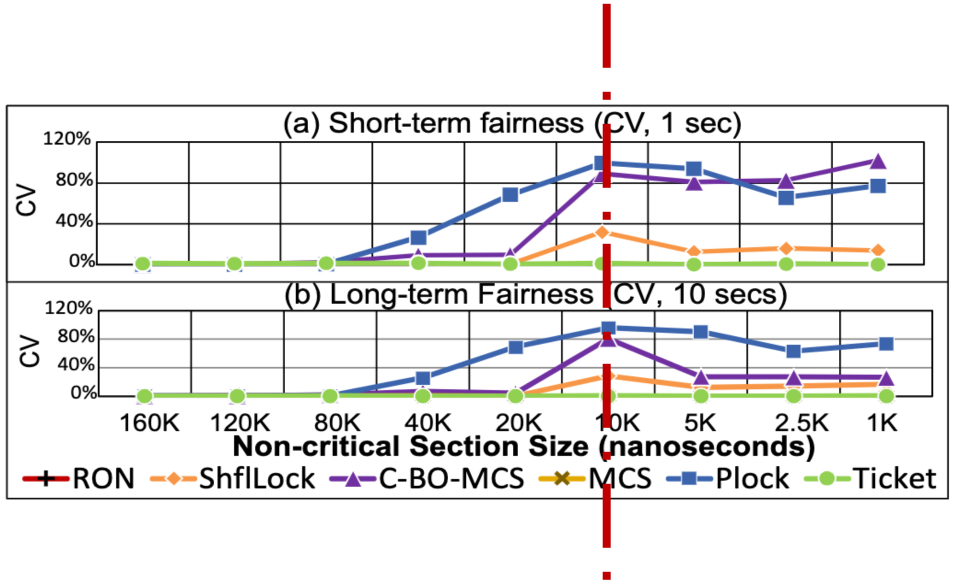
Non-critical section size is nCS ± 15%

Fairness

Performance anomaly prevention

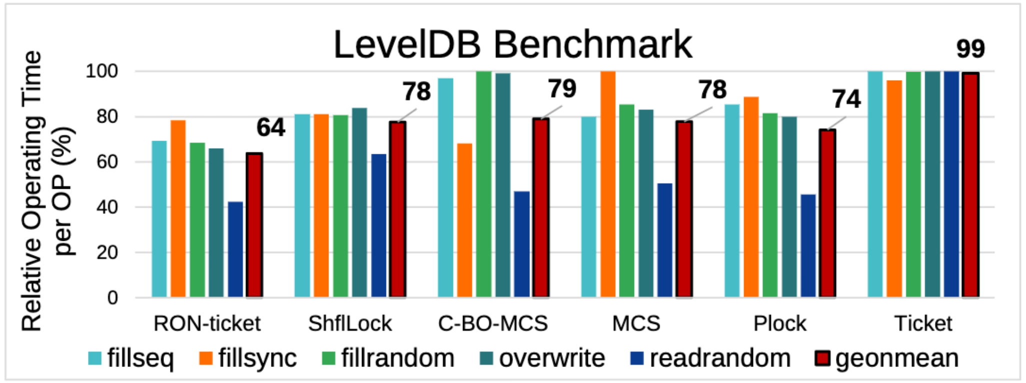
- Without well control, threads with the same code could have very different lock-unlock performance.
- In high contention scenarios (nCS = 10K), with shlflock and C-BO-MCS, die1 and die3 threads have a lower chance of entering the critical section, in addition to having a higher coefficient of variation.



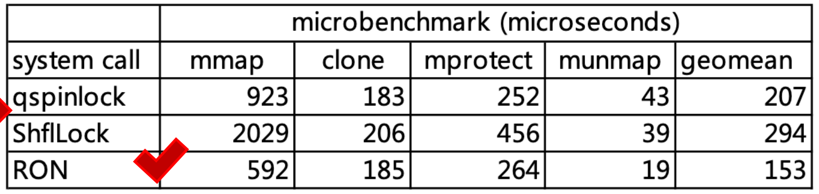


Application-level benchmarks

- As shown in the left figure, RON demonstrates the best relative operating time among the lock algorithms.
- The right figure illustrates the performance comparison of ShflLock, qspinlock, and RON in the Linux kernel.
  - In Figure (a), with 64 threads continuously issuing system calls, RON demonstrates the best performance under such high pressure.
  - Figure (b) depicts the execution of LevelDB, where RON and qspinlock exhibit nearly the same average performance.



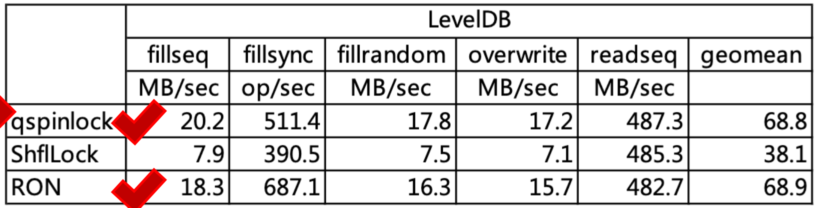
High contention



	microbenchmark (microseconds)				
system call	mmap	clone	mprotect	munmap	geomean
qspinlock	923	183	252	43	207
ShflLock	2029	206	456	39	294
RON ✓	592	185	264	19	153

(a)

Low contention



	LevelDB					
	fillseq	fillsync	fillrandom	overwrite	readseq	geomean
	MB/sec	op/sec	MB/sec	MB/sec	MB/sec	
qspinlock ✓	20.2	511.4	17.8	17.2	487.3	68.8
ShflLock	7.9	390.5	7.5	7.1	485.3	38.1
RON ✓	18.3	687.1	16.3	15.7	482.7	68.9

(b)

Conclusions

1. Due to the different connection speeds of the cores within a processor, optimizing spinlock performance can be likened to solving the shortest path problem.
2. Some cores are more powerful in acquiring locks. A more effective fairness mechanism is needed.
3. Configurability, which allows an application to utilize a group of adjacent cores, can be as important as scalability.
4. To prevent the next thread from being scheduled out, the oversubscription problem can be seen as "multithread-on-a-core share the same lock."