A Comparative Study of Locking Techniques: Fairness and Throughput Performance Evaluation

Ching-Shen Lin, Chao-Ting Lin and Shiwu Lo

Department of Computer Science

National Chung Cheng University

Outline

- **Introduction**
- **Synchronization Algorithms**
- **Testing Program**
- **Throughput Evaluation**
- **Fairness Evaluation**
- **Conclusion**

Introduction

Background

- Synchronization
- GNU/Linux's two common locking strategies
  - Spin Locks
  - Blocking Locks
- **Goal:** Examines the throughput and fairness of different locking methods in a shared resource access scenario

Synchronization Algorithms

Synchronization Algorithms

- Raw spinlock
- Pthread spinlock
- Ticket spinlock
- MCS spinlock
- Pthread Mutex
- Binary Semaphore

Raw Spinlock

- The Raw Spinlock algorithm employs the test-and-test-and-set (TTAS) technique to continuously examine and establish the lock status.
- This mechanism guarantees that only one thread can have exclusive access to the critical section at any given moment.

Pthread Spinlock

- The GNU Pthreads spinlocks (plock) provided by C library
- It also uses TTAS

```
 2  pthread_spinlock_t spin;
 3
 4  void spin_init(){
 5      pthread_spin_init(&spin, PTHREAD_PROCESS_PRIVATE);
 6  }
 7  void spin_lock(){
 8      pthread_spin_lock(&spin);
 9  }
10  void spin_unlock(){
11      pthread_spin_unlock(&spin);
12  }
```

Ticket Spinlock

- Threads wait until the ”service number” equals their ticket number before entering the critical section.
- This method can consume significant Network- on-Chip (NoC) bandwidth as waiting threads continuously query the ”grant” variable in a loop.

MCS Spinlock

- The MCS spinlock algorithm queues waiting threads in a linked list.
- Upon leaving the critical section, a thread sets the “locked” variable of the next thread to false to signal that it can enter the critical section.

Pthread Mutex

- We used the GNU/Linux's mutex function from the GNU pthread library.
- If the mutex is already locked, the calling thread will be blocked until it acquires the mutex.
- Upon completion of the function, the mutex object will be locked and owned by the calling thread

Binary Semaphore

- We used a binary semaphore with C library to restrict access to one thread at a time.
- sem_wait(): decreases the semaphore value if greater than zero, blocking the thread otherwise.
- sem_post(): reduces the semaphore by one and resumes waiting threads when the value increases.

Testing Program

Testing Environment

- Model name: AMD Ryzen Threadripper 2990WX
- Number of Cores: 32-Core Processor
- Virtual core: 64 (virtual core per core is 2)
- Architecture: x86_64
- Compile Environment: gcc version 10.3.0
- Target: AMD Zen+ and x86 64-linux-gnu
- Thread model: POSIX

Testing Program

- We analyze each lock method in a quantitative manner through a controlled microbenchmark.
- Each thread is assigned to a specific core for controlled testing.

Testing Program

Algorithm 1 The Microbenchmark

```
1  while(1):
2      spin_lock();
3      // critical section start
4      for(each element in sharedData):
5          if(with file I/O):
6              element += 1;
7              fprintf(test_file);
8          if(without file I/O):
9              element += 1;
10     // end of critcal section
11     spin_unlock();
12     // remainder section
13     rs_start = clock_gettime();
14     do{
15         t = clock_gettime();
16     }while(t-rs_start < RS_size*rand(0.85~1.15));
```

- The benchmark is run in two variations: one with file input/output operations and one without.

Testing Program

Algorithm 1 The Microbenchmark

```
1  while(1):
2      spin_lock();
3      // critical section start
4      for(each element in sharedData):
5          if(with file I/O):
6              element += 1;
7              fprintf(test_file);
8          if(without file I/O):
9              element += 1;
10     // end of critcal section
11     spin_unlock();
12     // remainder section
13     rs_start = clock_gettime();
14     do{
15         t = clock_gettime();
16     }while(t-rs_start < RS_size*rand(0.85~1.15));
```

- The random variable is used to simulate different loads of the program.

Throughput Evaluation

Throughput Evaluation Aspects

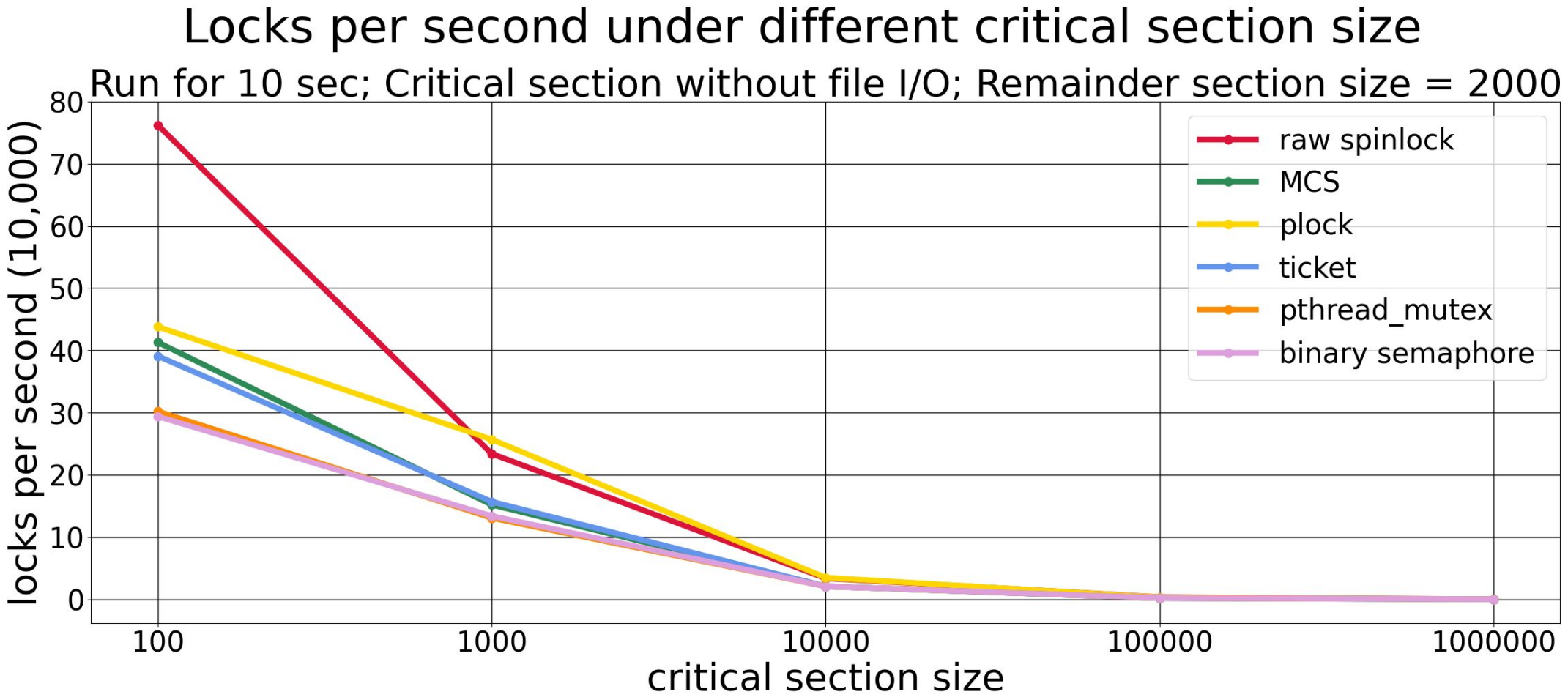
- Critical section size
- Remainder section size
- Oversubscription issue

Evaluation Aspects – (1)

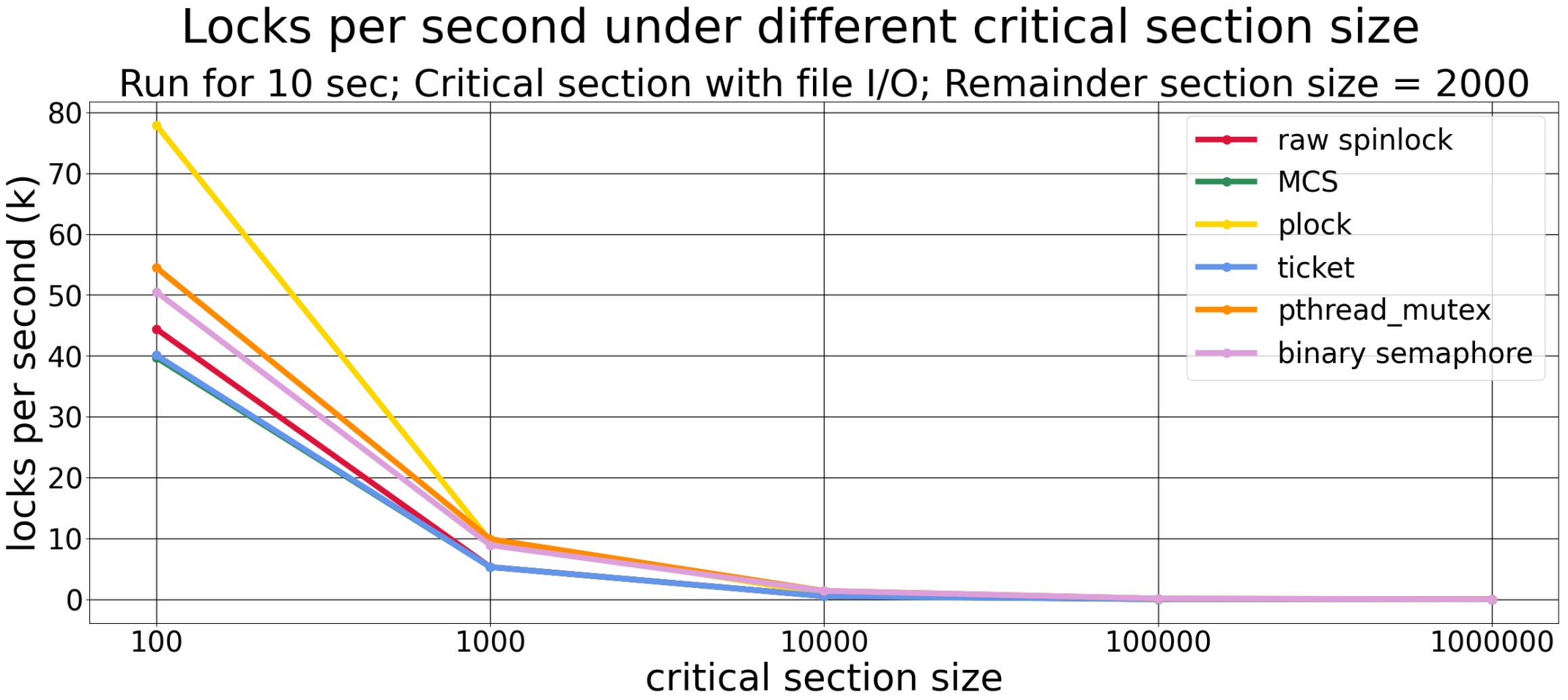
- **Critical section size**
  - The number of times the *for* loop execute in the critical section
  - For example *for(0 to 100)* means the critical section size = 100.

```
3       // critical section start
4       for(each element in sharedData):
5           if(with file I/O):
6               element += 1;
7               fprintf(test_file);
8           if(without file I/O):
9               element += 1;
10      // end of critcal section
```

Result - Critical Section Size (1)
without file I/O



Result - Critical Section Size (2)
with file I/O



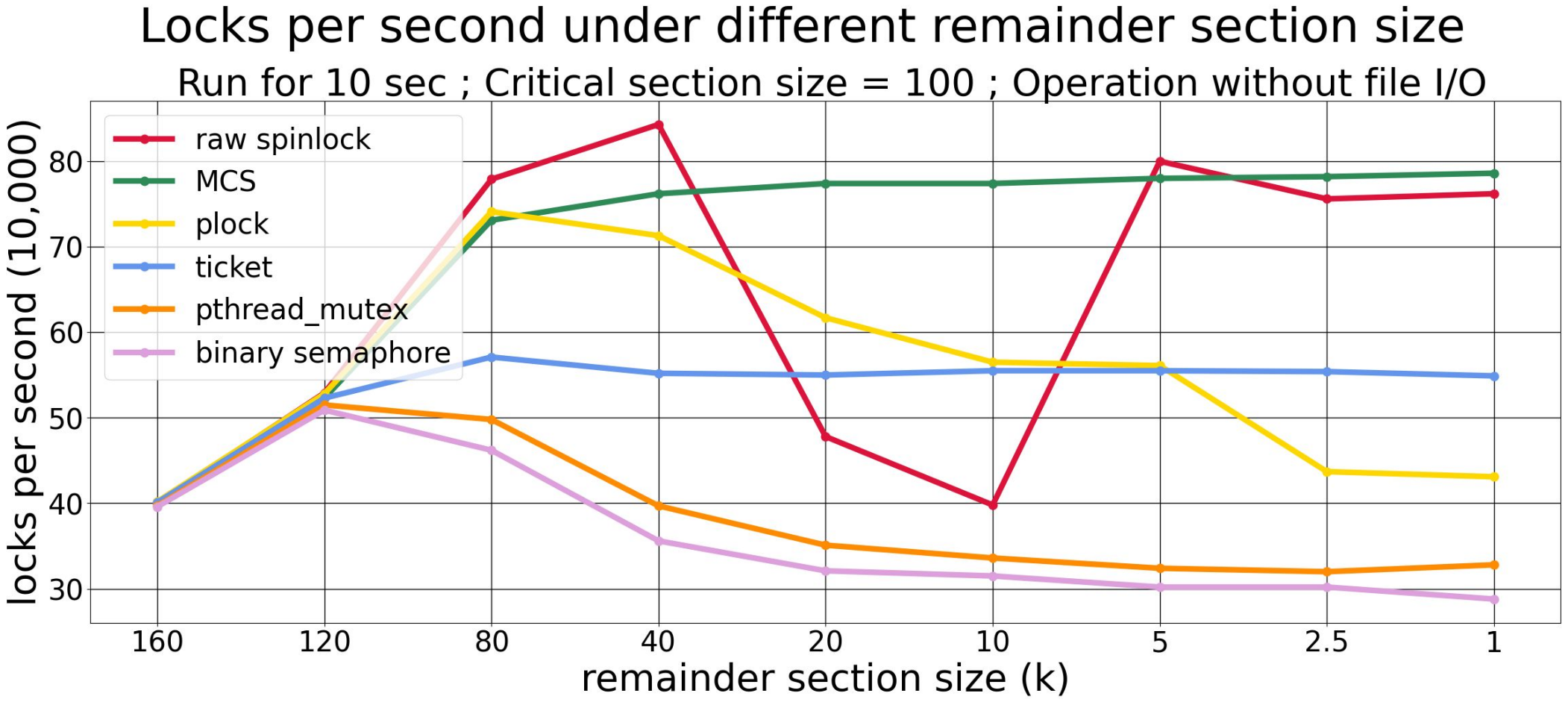
Evaluation Aspects – (2)

- **Remainder section size**
  - The time threads will wait after leaving the critical section
  - A larger remainder section size results in longer waiting times, indicating a low load condition. Conversely, a smaller remainder section size corresponds to a high load condition.

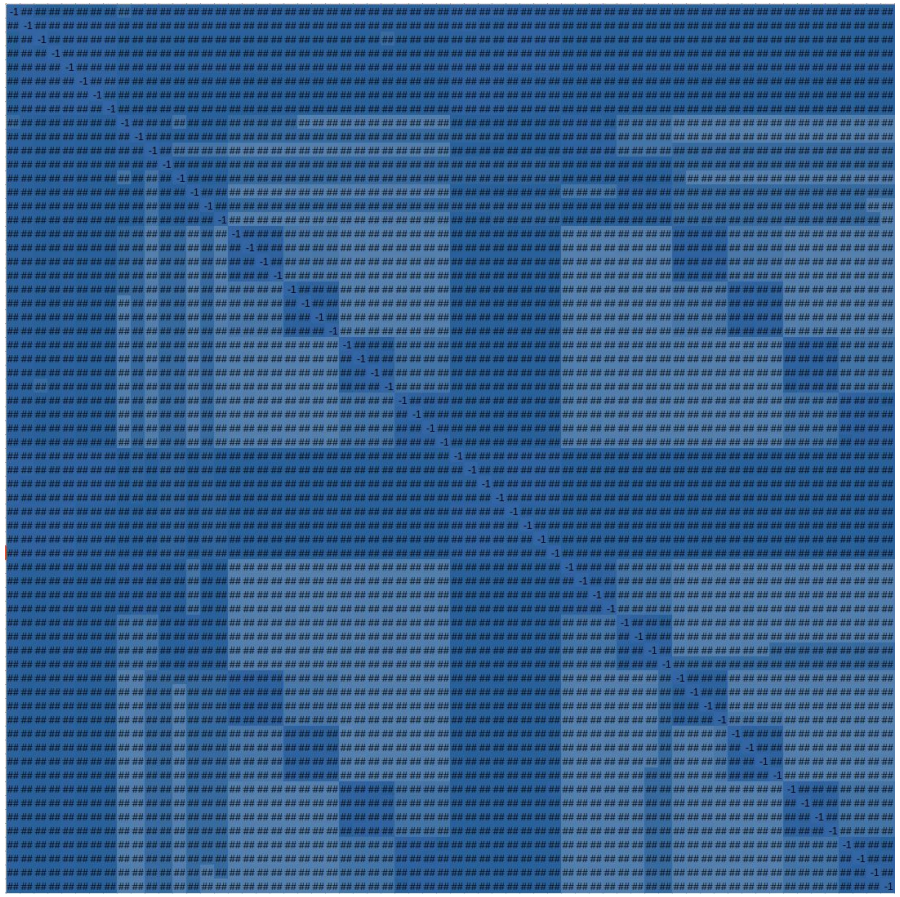
```
12        // remainder section
13        rs_start = clock_gettime();
14        do{
15             t = clock_gettime();
16        }while(t-rs_start < RS_size*rand(0.85~1.15));
```

Result - Remainder Section Size (1)

without file I/O



Why Raw Spinlock drops

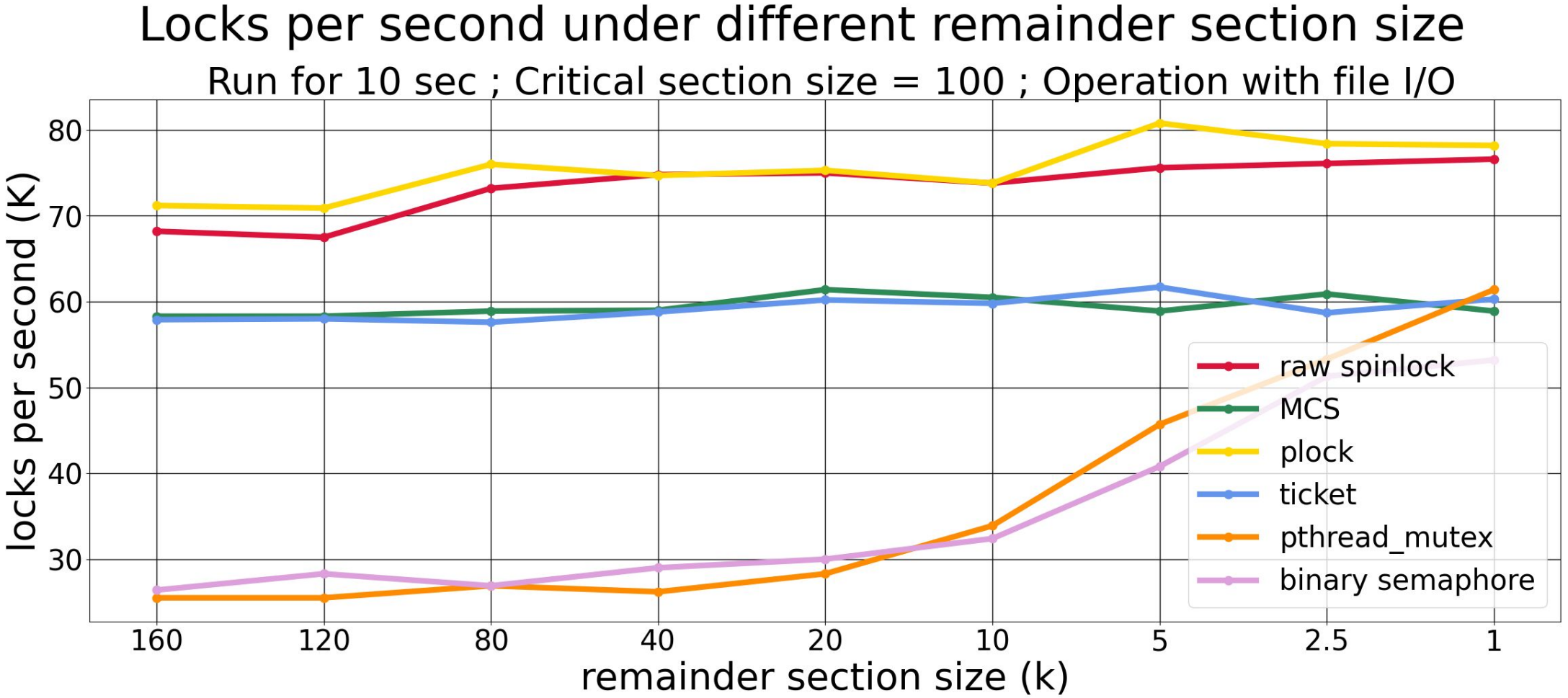


Core to core transmission time

- This graph shows the transmission time between cores
- The deeper the closer between two cores
- We use *atomic_compare_exchange()*
- Under this workload, the cores involved in the transfer happen to be farther apart, resulting in decreased performance

Result - Remainder Section Size (2)

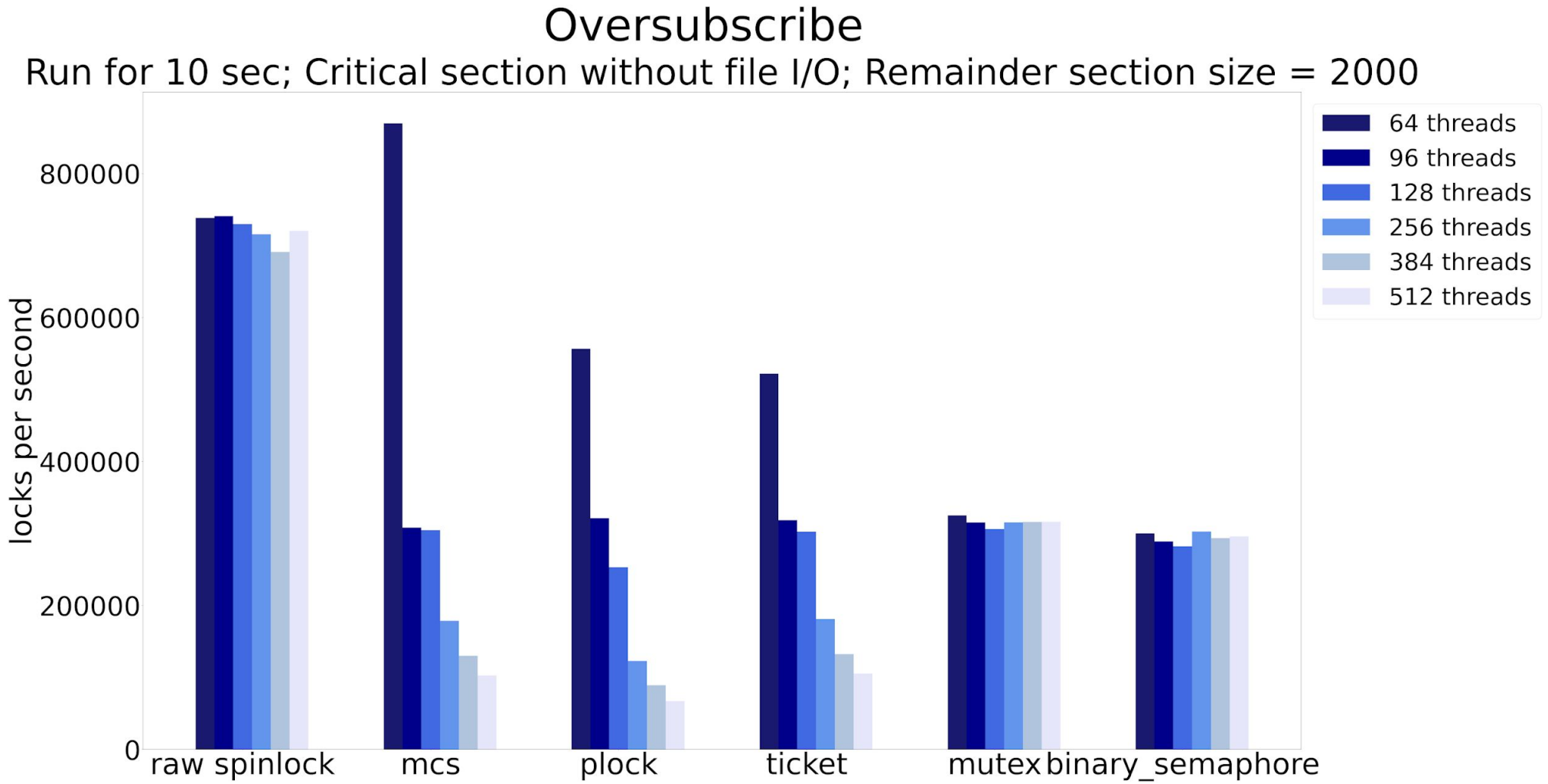
with file I/O



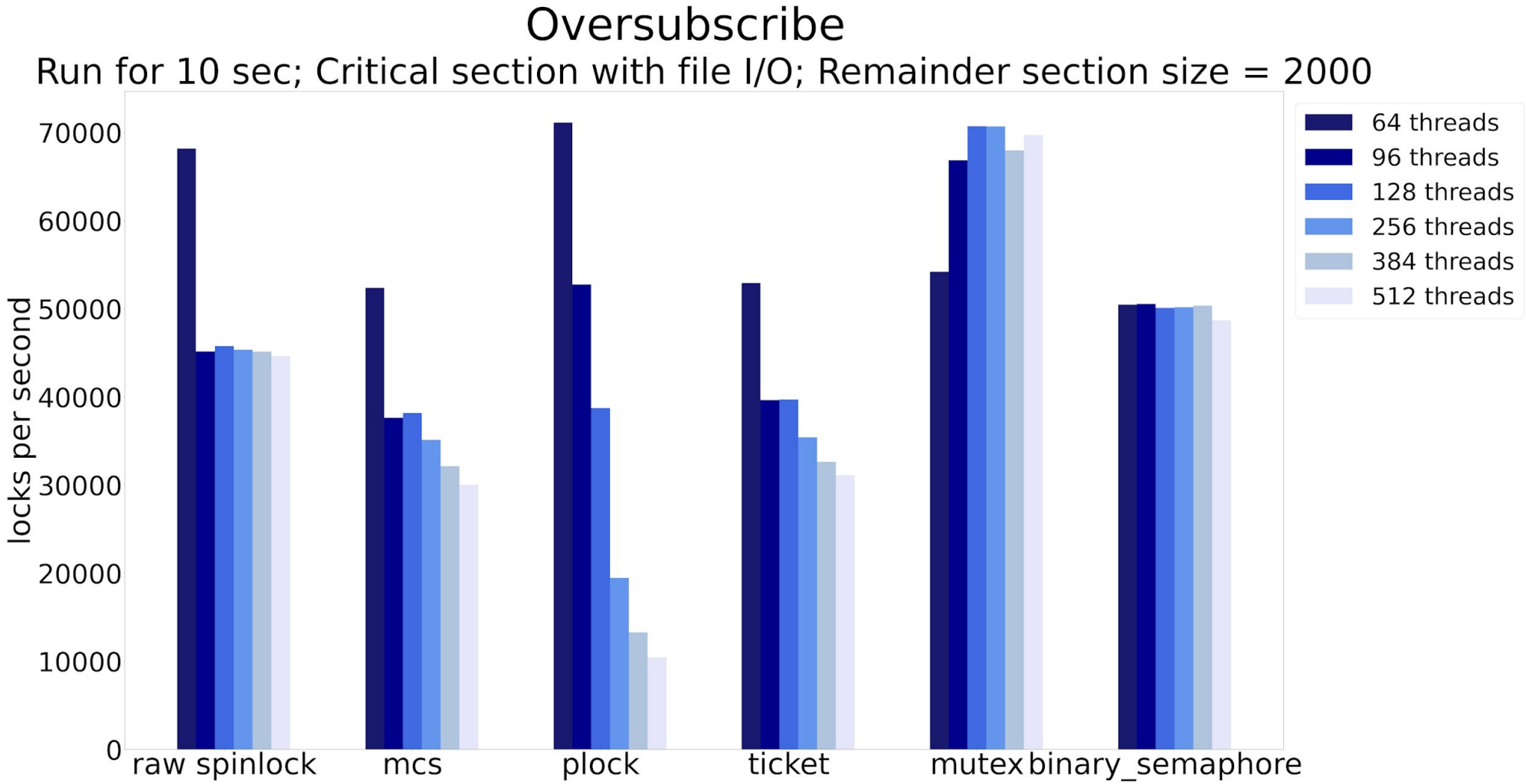
Evaluation Aspects – (3)

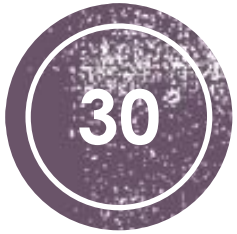
- **Oversubscription Issue**
  - In oversubscription, when there are more threads than available cores, it causes waiting, increased latency, and decreased performance.
  - We evaluated the locks for oversubscription by running the microbenchmark on a system with an excess of threads compared to cores, measuring their performance.

Result - Oversubscription Issue (1)
without file I/O



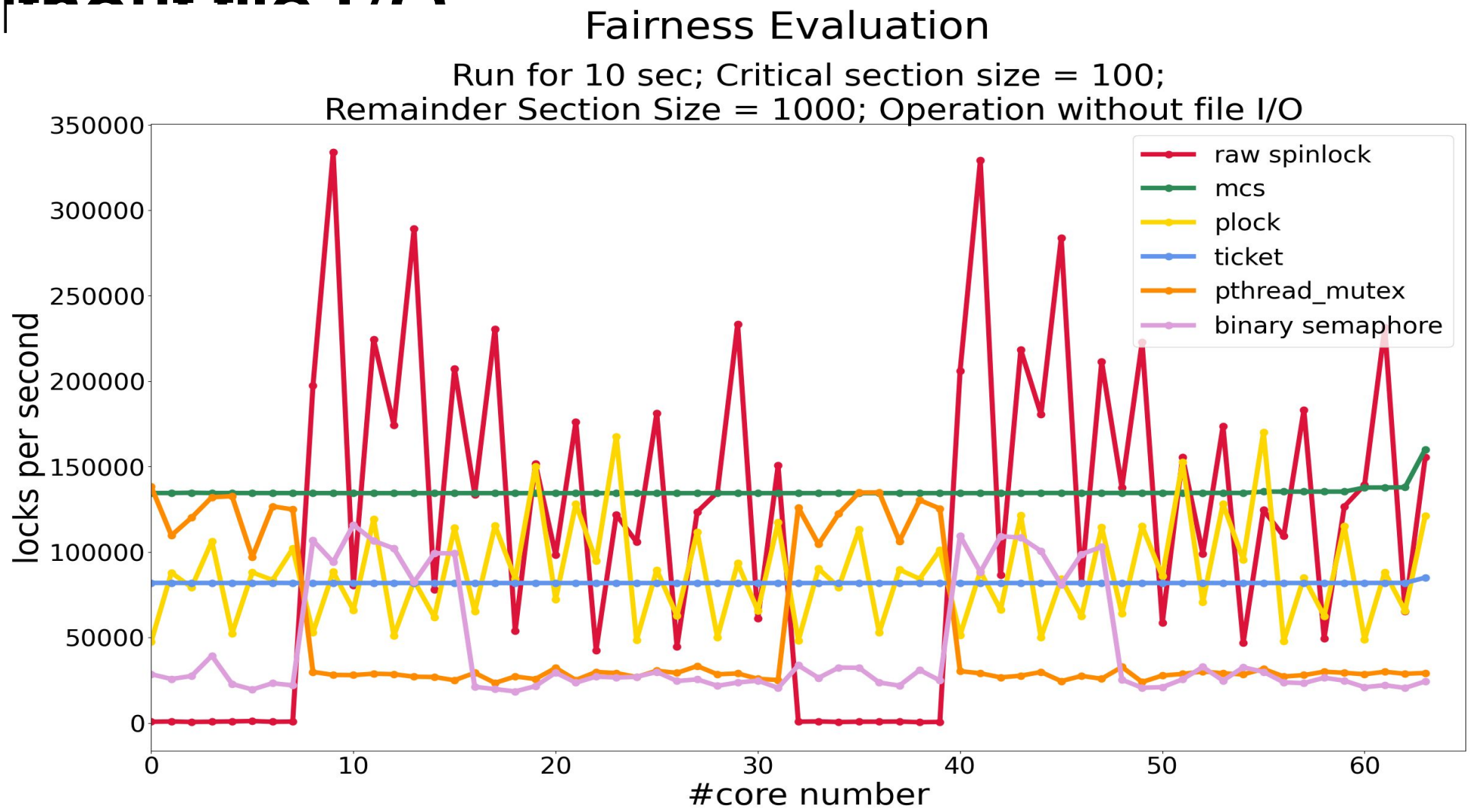
Result - Oversubscription Issue (2)
with file I/O



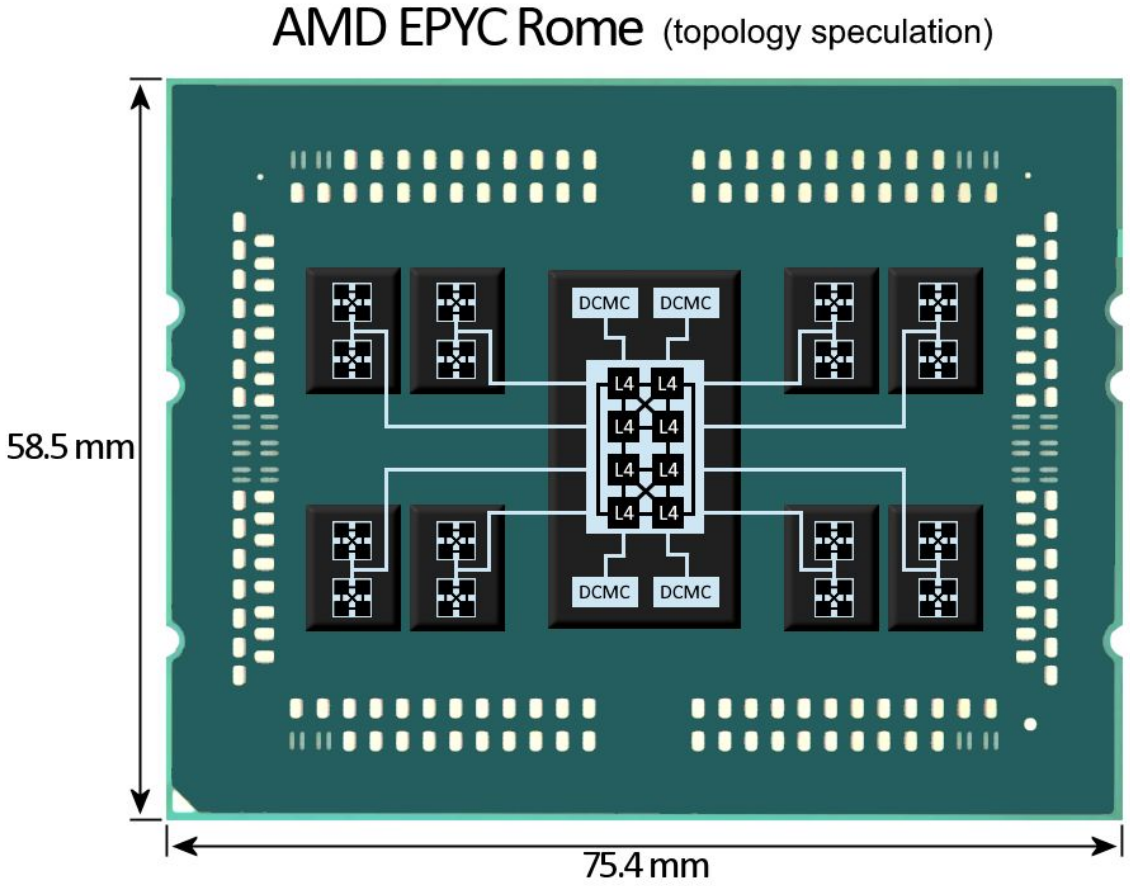


Fairness Evaluation

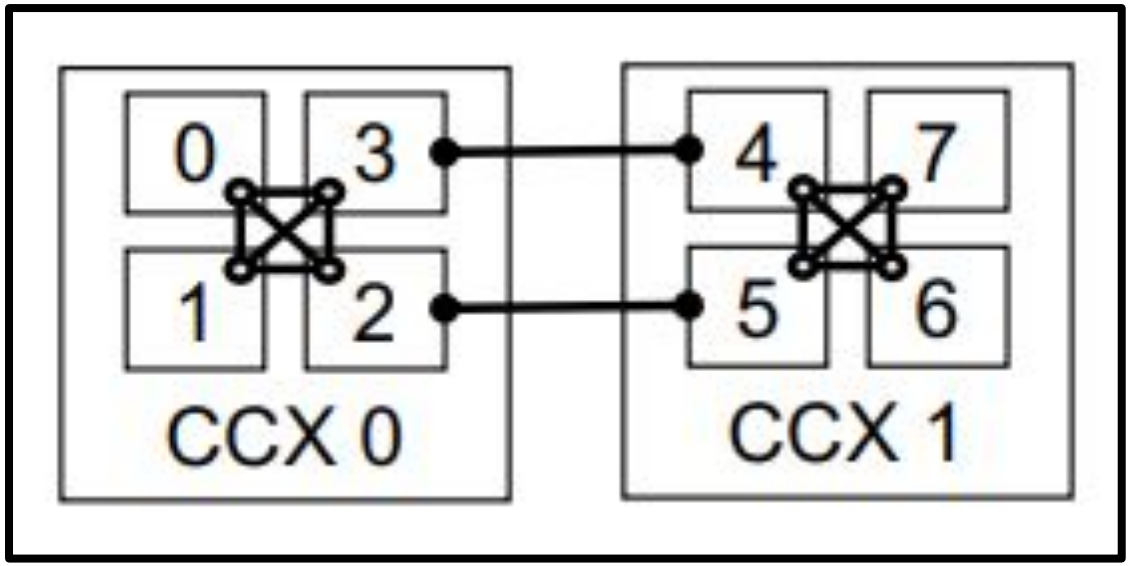
Result - Undersubscription Issue (1)
without file I/O



Why



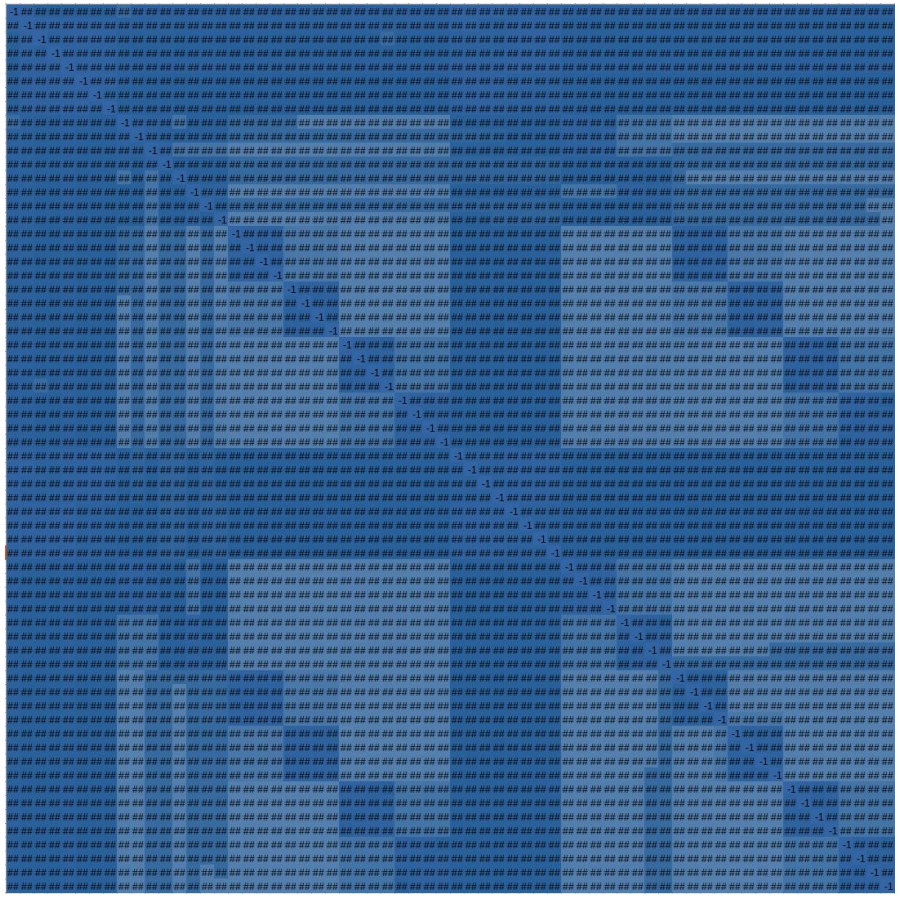
AMD EPYC processor architecture



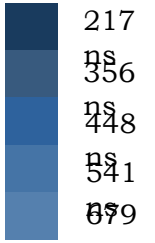
CPU Die

Resource

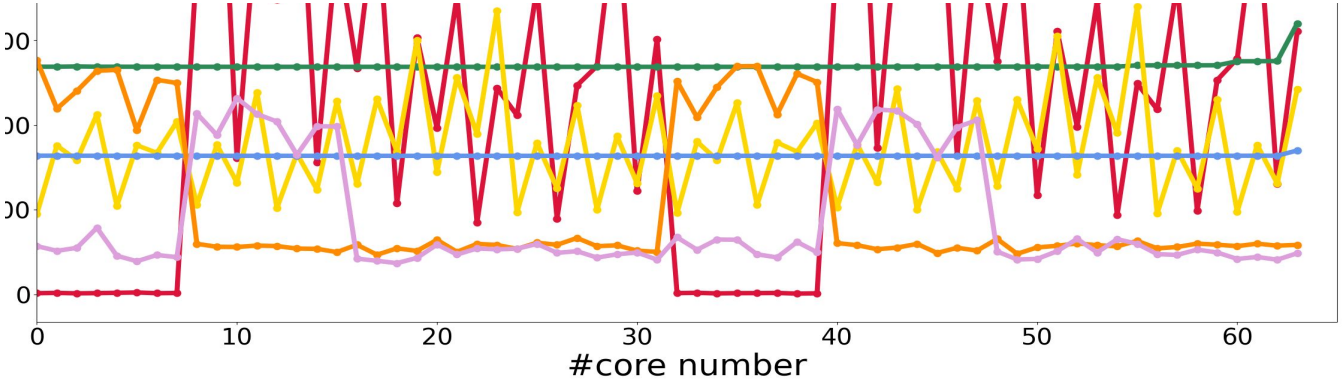
Why



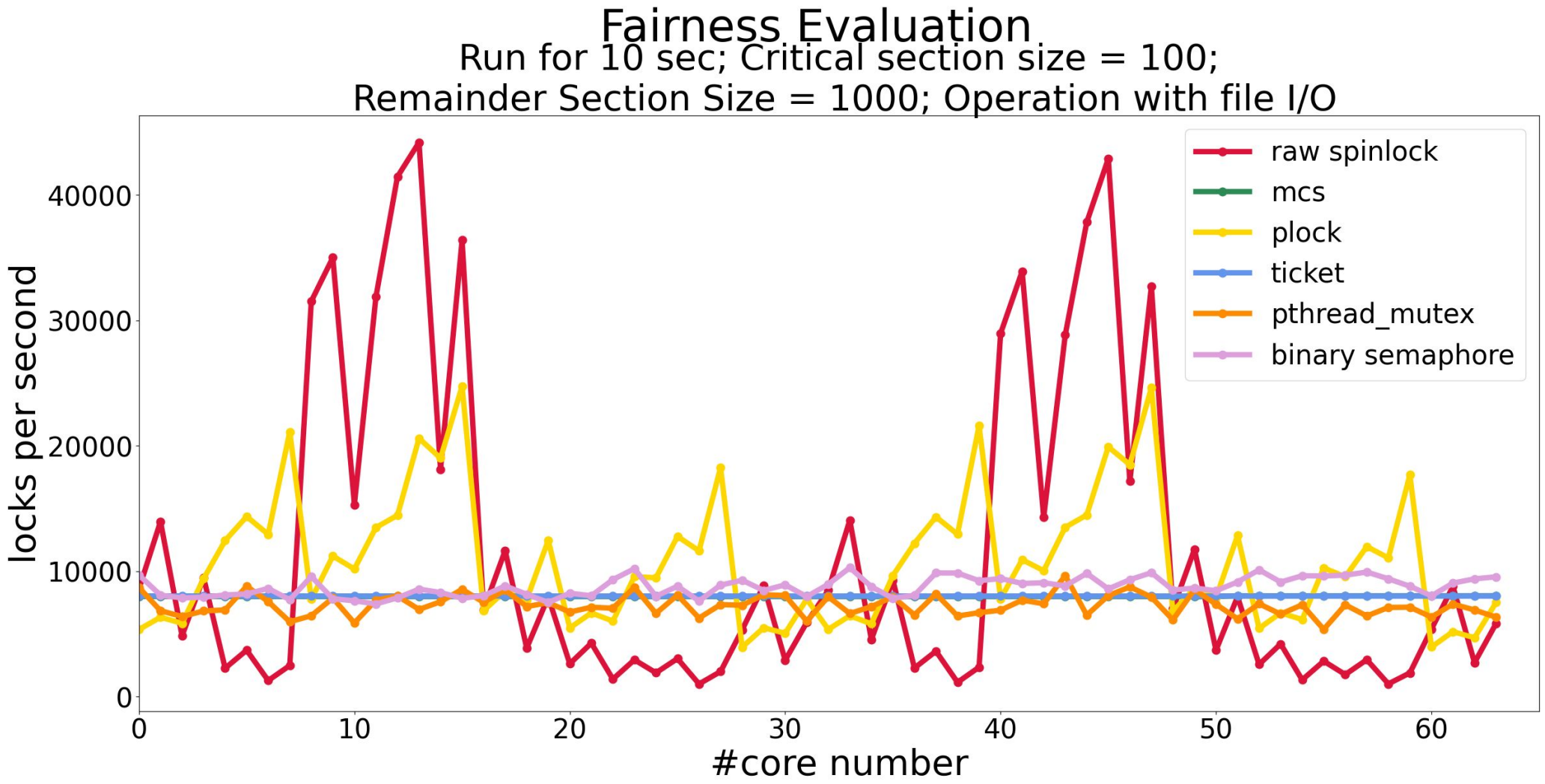
Core to core transmission time



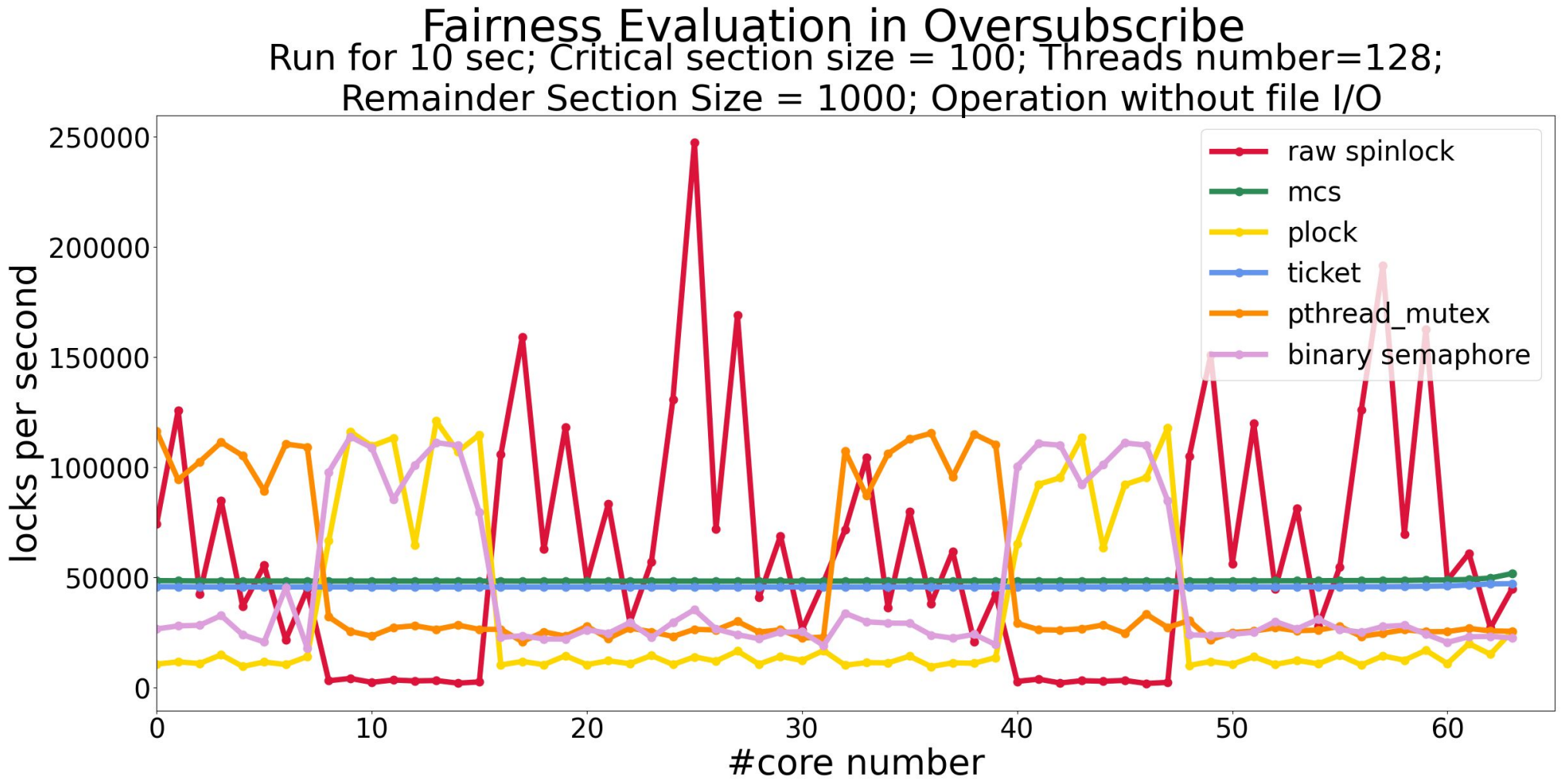
- This graph shows the transmission time between cores
- The deeper the closer between two cores
- The distances between cores can be divided into several major categories.



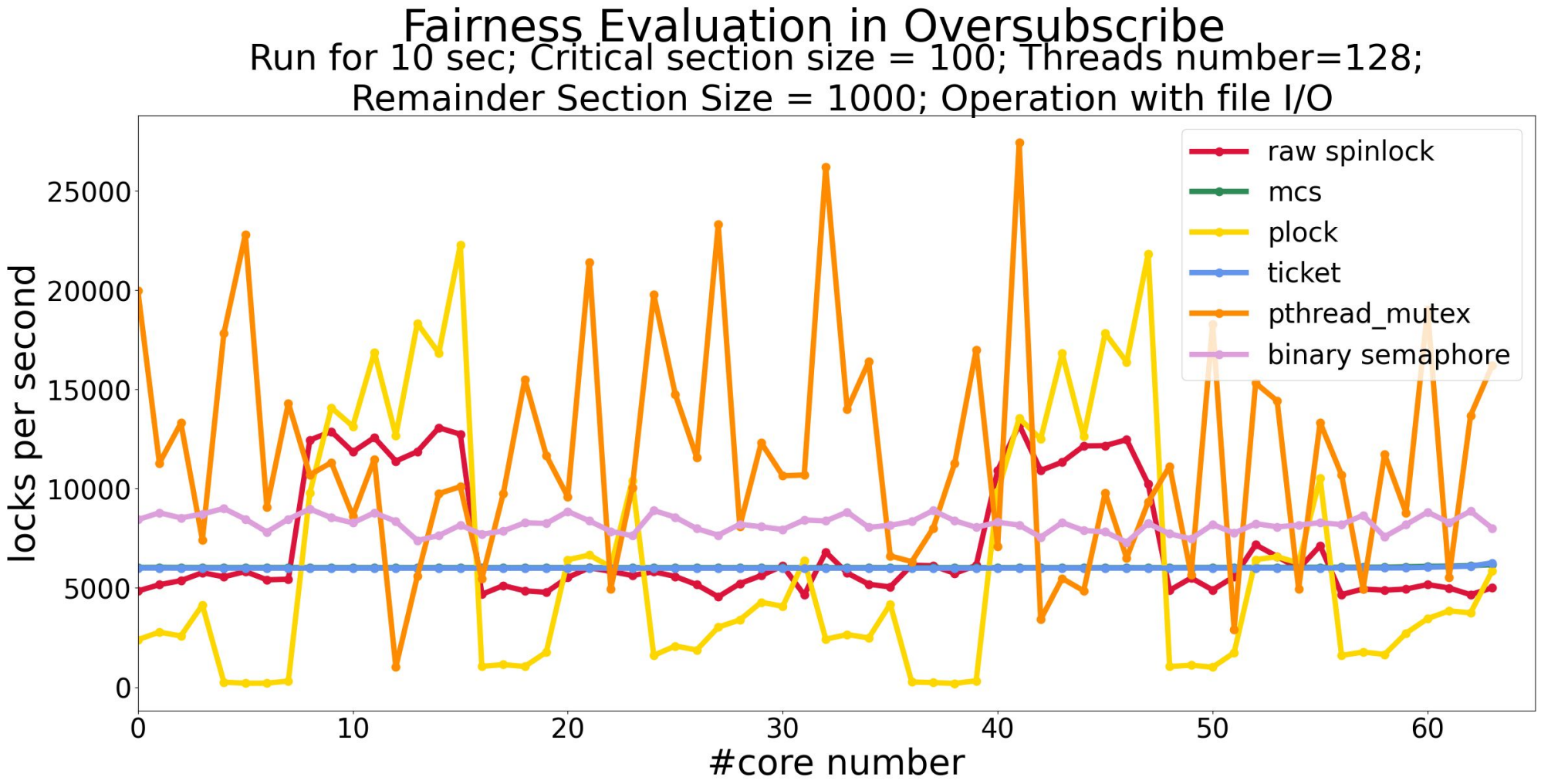
Result - Undersubscription Issue (2)
with file I/O



Result - Oversubscription Issue (1)
without file I/O



Result - Oversubscription Issue (2)



Conclusion

- Locking mechanism choice is critical for performance and fairness in shared resource access.
- Spinlocks perform well in smaller critical section sizes.
- Semaphore and mutex locks outperform ticket and MCS spinlocks as critical section size increases.
- The type of operation performed on the shared resource affects locking mechanism performance. Mutex and semaphore locks excel in file I/O operations.

Conclusion

- Oversubscription negatively impacts performance and fairness across all locking mechanisms.
- GNU/Linux's mutex exhibits extreme unfairness during file I/O operations in oversubscription.
- Optimal performance and fairness in shared resource access depend on carefully considering critical section size, operation type, and the potential for oversubscription when selecting a locking mechanism.

Q&A

Thank you for your attention