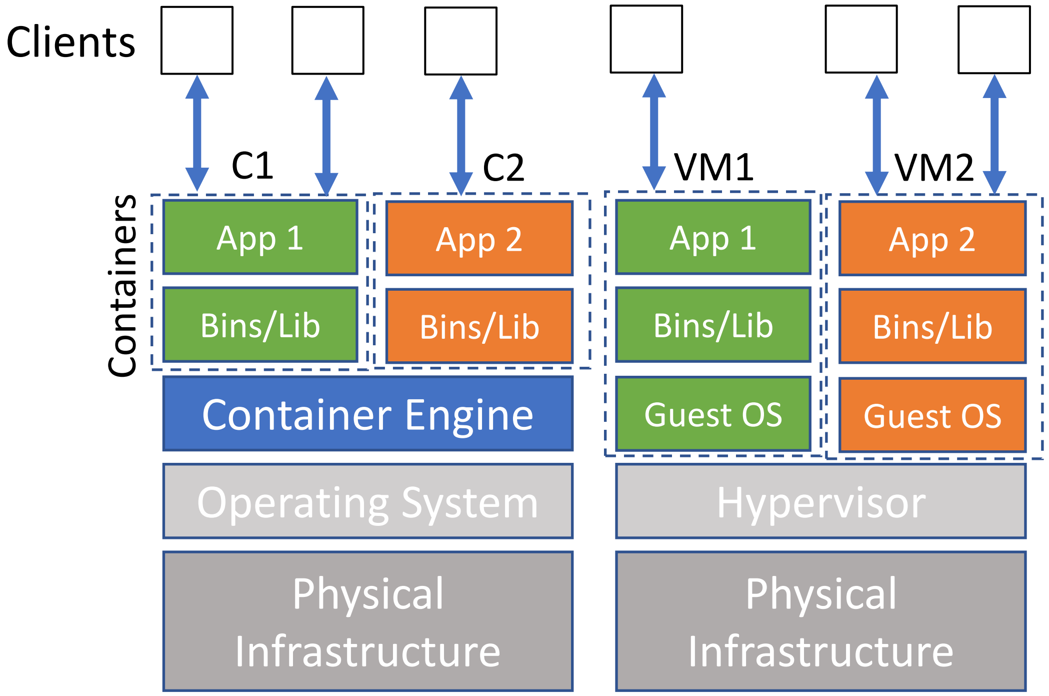
Avoiding Scheduler Subversion using Scheduler-Cooperative Locks

Yuvraj Patel, Leon Yang[*], Leo Arulraj[+],
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift
University of Wisconsin-Madison

[*] - Now at Facebook, [+] - Now at Cohesity

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Competitive environment



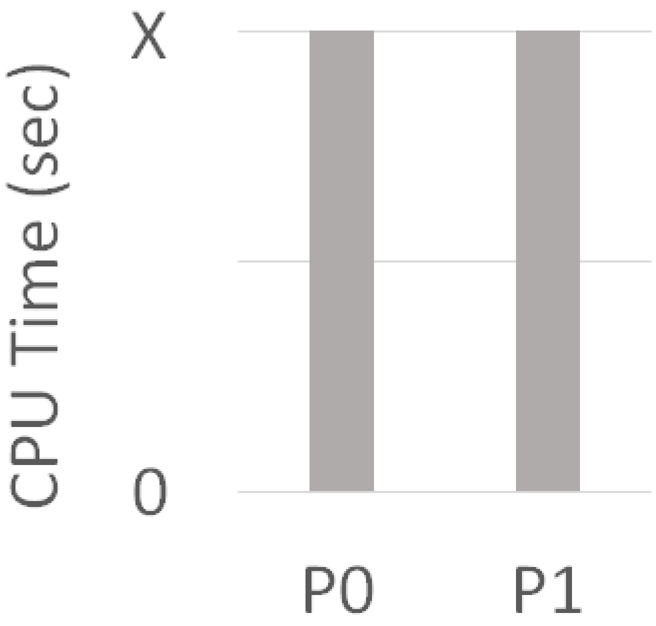
Example use-cases of modern data centers

- Every container/VM/user expects their desired share of resources
- Schedulers play an important role to fulfill the expectations
- CPU schedulers important for CPU allocation
- Majority of the systems are concurrent systems protected by locks

The problem – Scheduler Subversion

- Accessing locks can lead to new problem - “Scheduler subversion”
- Locks determine CPU allocation instead of the scheduler

- 2 Processes – P0 & P1
- Default priority
- P0 holds the lock twice as long as P1
- Ticket lock-acquisition fairness
- Linux CFS Scheduler

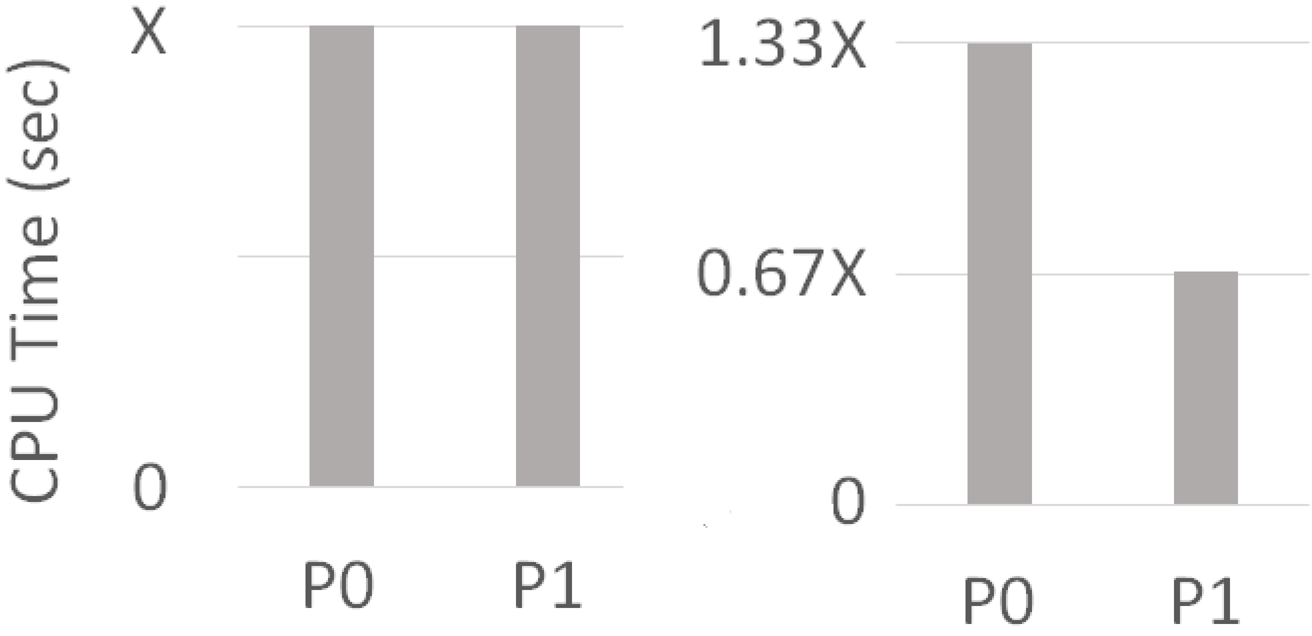


Expected

The problem – Scheduler Subversion

- Accessing locks can lead to new problem - “Scheduler subversion”
- Locks determine CPU allocation instead of the scheduler

- 2 Processes – P0 & P1
- Default priority
- P0 holds the lock twice as long as P1
- Ticket lock-acquisition fairness
- Linux CFS Scheduler



Expected

Observed

CPU allocation aligns with lock usage

The solution – Scheduler-Cooperative Locks

- Scheduler-Cooperative Locks (SCL) guarantee lock usage fairness by aligning with scheduling goals
- Three important design components to build SCLs
  - Track lock usage
  - Penalize dominant users
  - Provide dedicated window of opportunity to every user
- Implementation - Two user-space locks and one kernel lock
- Evaluation
  - Correctness - Allocate lock usage according to the scheduling goals even in extreme cases
  - Performance - Efficient and scalable
  - Useful – Apply SCLs to real-world systems – UpScaleDB, KyotoCabinet, Linux kernel

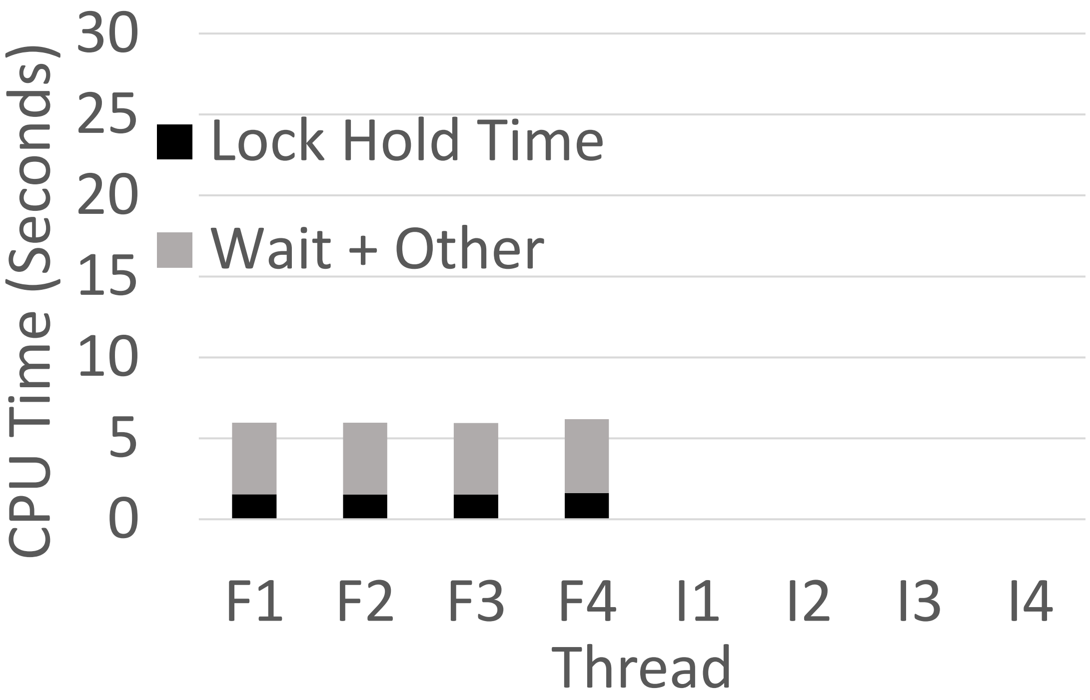
- Introduction
- The Problem – Scheduler Subversion
- The Solution – Scheduler-Cooperative Locks
- Evaluation
- Conclusion

Lock & CPU dominance

- UpScaleDB – embedded key-value database
- Global mutex lock
- Workload
  - 8 threads pinned on 4 CPU
    - 4 threads insert ops
    - 4 threads find ops
  - Default thread priority
    - Equal CPU allocation
  - Run for 120 seconds

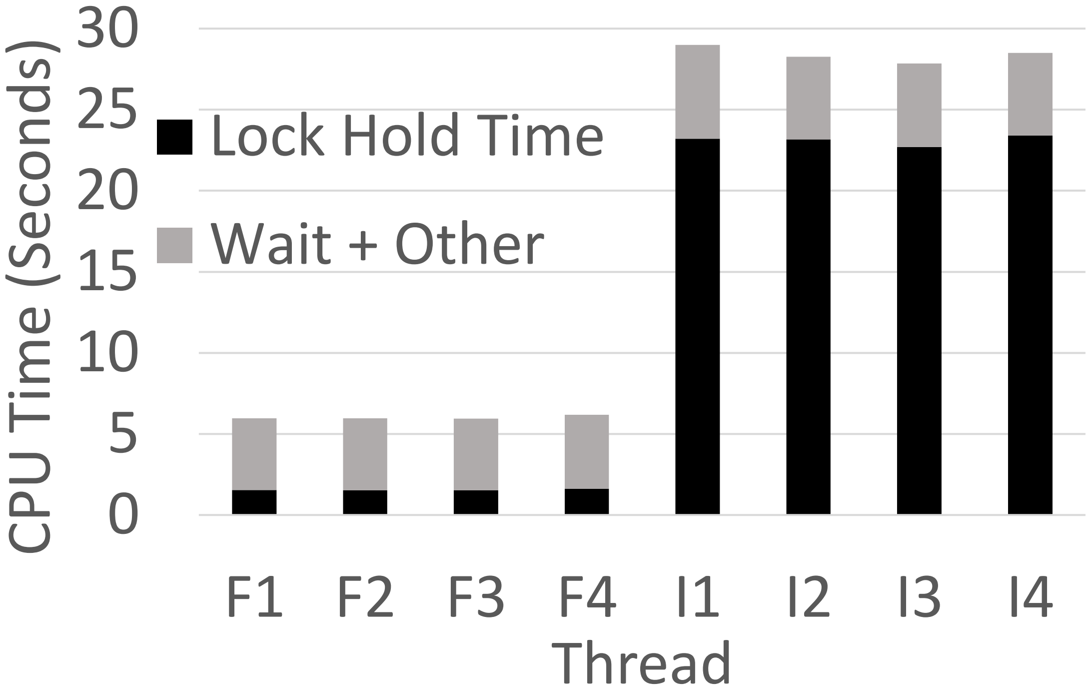
Lock & CPU dominance

- UpScaleDB – embedded key-value database
- Global mutex lock
- Workload
  - 8 threads pinned on 4 CPU
    - 4 threads insert ops
    - 4 threads find ops
  - Default thread priority
    - Equal CPU allocation
  - Run for 120 seconds



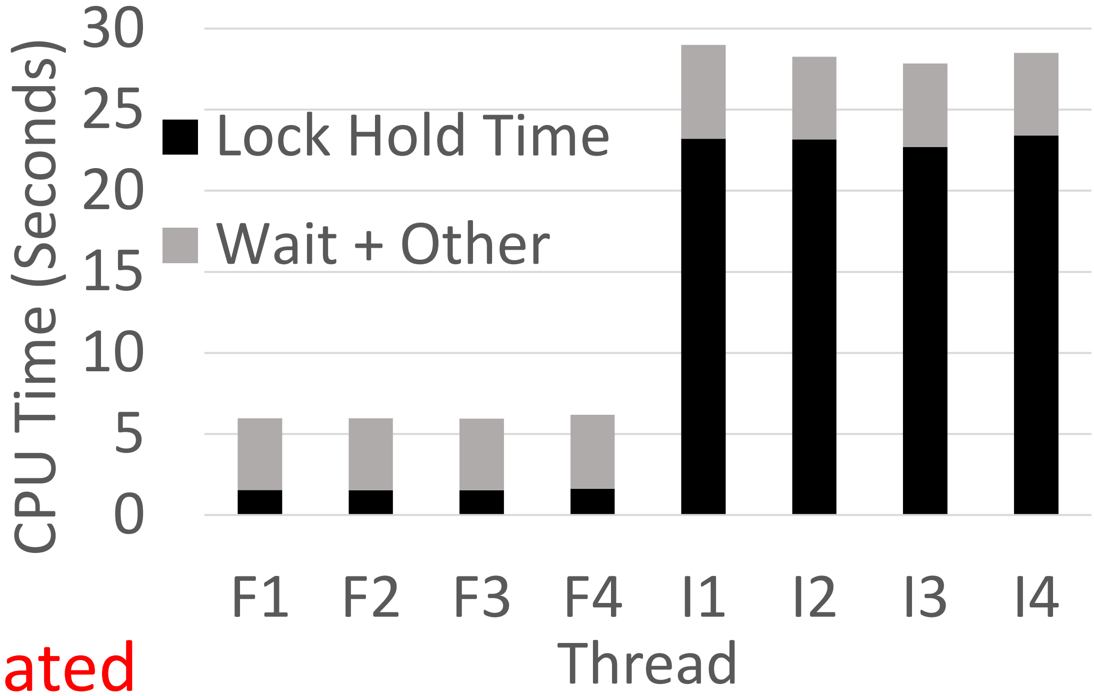
Lock & CPU dominance

- UpScaleDB – embedded key-value database
- Global mutex lock
- Workload
  - 8 threads pinned on 4 CPU
    - 4 threads insert ops
    - 4 threads find ops
  - Default thread priority
    - Equal CPU allocation
  - Run for 120 seconds



Lock & CPU dominance

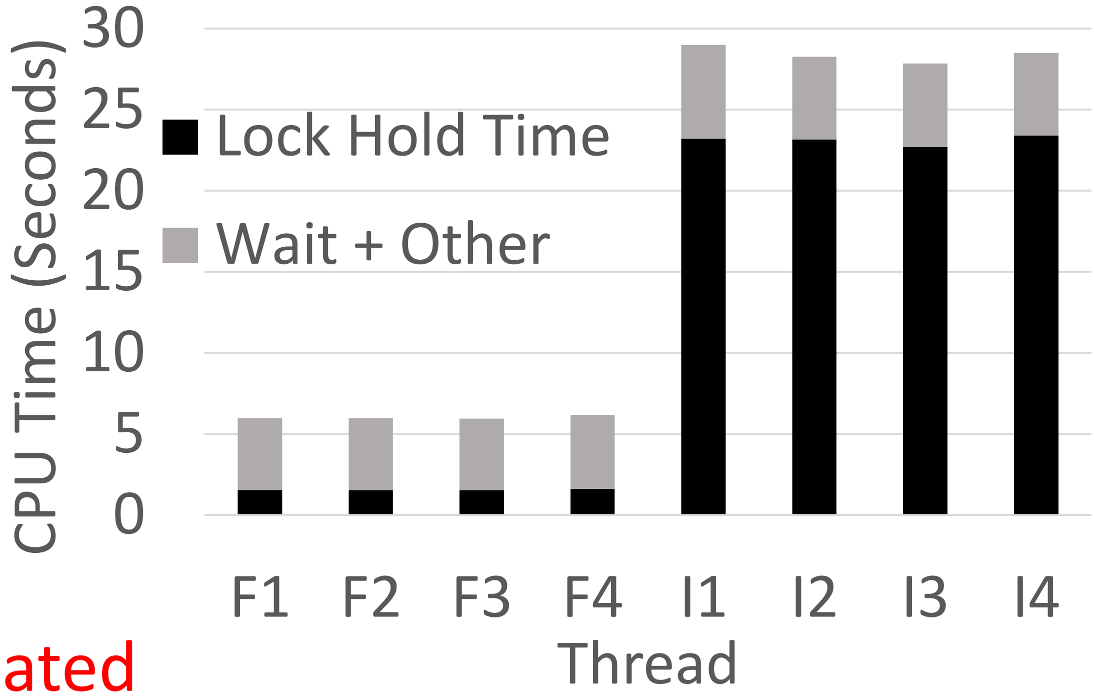
- UpScaleDB – embedded key-value database
- Global mutex lock
- Workload
  - 8 threads pinned on 4 CPU
    - 4 threads insert ops
    - 4 threads find ops
  - Default thread priority
    - Equal CPU allocation
  - Run for 120 seconds



Nearly six times more CPU allocated to insert threads than find threads

Lock & CPU dominance

- UpScaleDB – embedded key-value database
- Global mutex lock
- Workload
  - 8 threads pinned on 4 CPU
    - 4 threads insert ops
    - 4 threads find ops
  - Default thread priority
    - Equal CPU allocation
  - Run for 120 seconds



Nearly six times more CPU allocated to insert threads than find threads

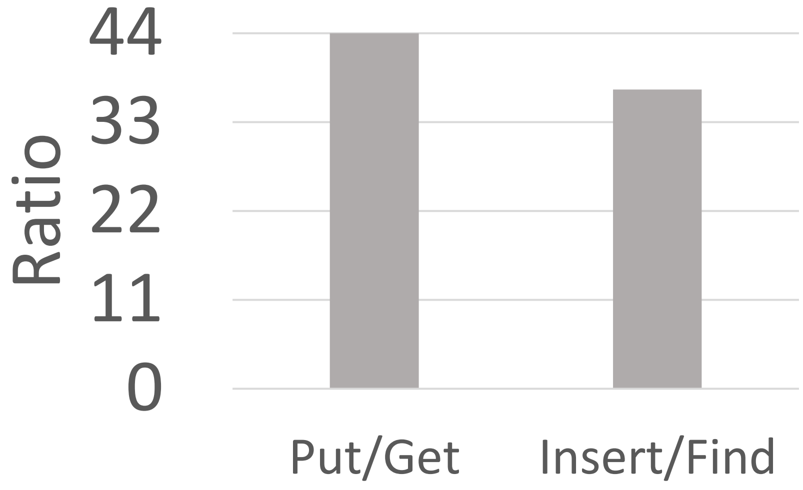
Insert threads dominate lock usage

Causes of scheduler subversion

- Two reasons

Reason #1 - Different critical section lengths

- Threads spend varied amount of time in critical section
- Thread dwelling longer in critical section becomes dominant user of CPU



LevelDB UpScaleDB

Ratio of median critical section times for various systems

Reason #2 - Majority locked run time

- Time spent in critical section is high -> contention
- Lock algorithm determines which threads scheduled
- Common case in many applications and OS [1,2,3,4]

1. Lock–Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems. ACM Trans. Comput. Syst.,36(1), March 2019
2. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. USENIX ATC 2012
3. Understanding Manycore Scalability of File Systems, USENIX ATC 2016
4. Non-scalable locks are dangerous. Linux Symposium, 2012

- Introduction
- The Problem – Scheduler Subversion
- The Solution – Scheduler-Cooperative Locks
- Evaluation
- Conclusion

Scheduler-Cooperative Locks (SCLs)

- Lock opportunity
  - Amount of time thread holds lock or could acquire lock when free
  - Important metric to measure lock usage fairness
- Philosophy
  - Prevent dominant users from acquiring lock
  - Ensure equal "lock opportunity" to every user
- Design locks that aligns with scheduling goals
- Three important design components

#1 - Track lock usage

- Track time spent in critical section

#1 - Track lock usage

- Track time spent in critical section

```
scl_lock()
{
    …..
    lock.start_cs = now()
}

scl_unlock()
{
    …..
    end_cs = now()
    cs_time = end_cs – lock.start_cs
    …..
}
```

#1 - Track lock usage

- Track time spent in critical section
- Tracking helps to identify dominant users

```
scl_lock()
{
    .....
    lock.start_cs = now()
}

scl_unlock()
{
    .....
    end_cs = now()
    cs_time = end_cs – lock.start_cs
    .....
}
```

#1 - Track lock usage

- Track time spent in critical section
- Tracking helps to identify dominant users
- Tracking flexible
  - Any schedulable entity such as threads, processes, containers
  - Type of work – readers or writers

```
scl_lock()
{
    .....
    lock.start_cs = now()
}

scl_unlock()
{
    .....
    end_cs = now()
    cs_time = end_cs – lock.start_cs
    .....
}
```

#2 – Penalize users

- Penalize dominant users

#2 – Penalize users

- Penalize dominant users
- Penalty calculated while releasing lock
- Penalty applied while acquiring lock
- Prevent user from acquiring lock

```
scl_lock()
{
    if (penalty) {
        sleep-until-penalty-time
    }
    …..
    lock.start_cs = now()
}

scl_unlock()
{
    …..
    end_cs = now()
    cs_time = end_cs – lock.start_cs
    calculate penalty, penalty-time
    …..
```

#2 – Penalize users

- Penalize dominant users
- Penalty calculated while releasing lock
- Penalty applied while acquiring lock
- Prevent user from acquiring lock
- Penalty based on scheduling goals

```
scl_lock()
{
    if (penalty) {
        sleep-until-penalty-time
    }
    …..
    lock.start_cs = now()
}

scl_unlock()
{
    …..
    end_cs = now()
    cs_time = end_cs – lock.start_cs
    calculate penalty, penalty-time
    …..
```

#3 – Dedicated window of opportunity

- Lock slice – dedicated window of opportunity to every user

#3 – Dedicated window of opportunity

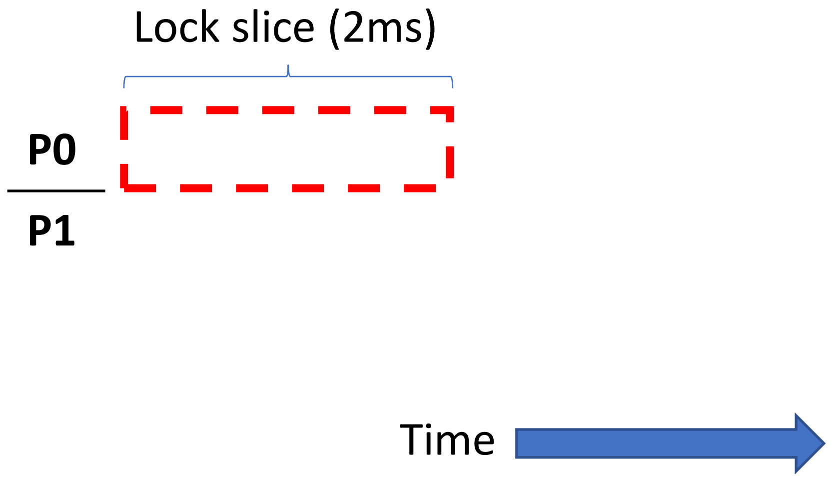
- Lock slice – dedicated window of opportunity to every user

P0

P1

#3 – Dedicated window of opportunity

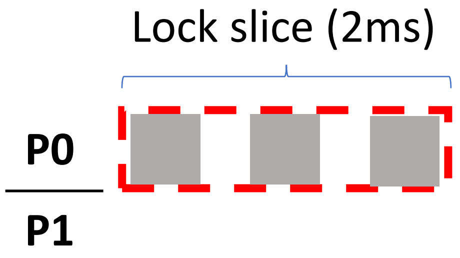
- Lock slice – dedicated window of opportunity to every user



Slice owner is lock owner

#3 – Dedicated window of opportunity

- Lock slice – dedicated window of opportunity to every user
- Owner can acquire lock multiple times within a slice without penalty

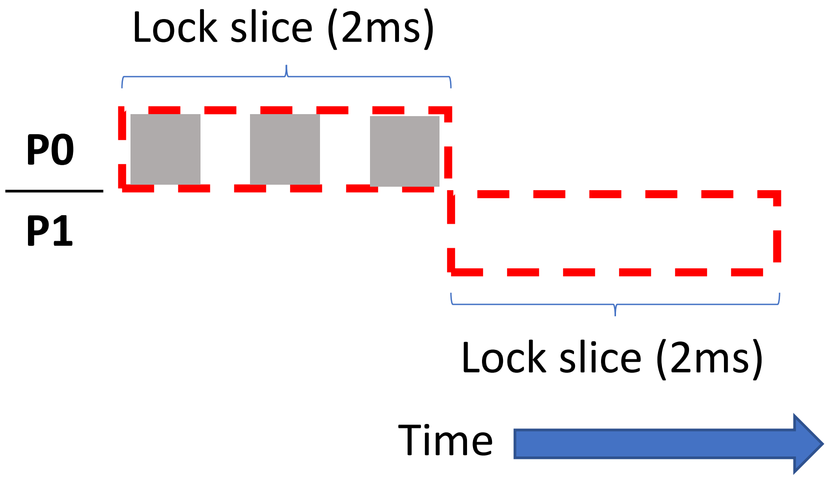


Lock acquisition is fast-pathed improving throughput

Slice owner is lock owner

#3 – Dedicated window of opportunity

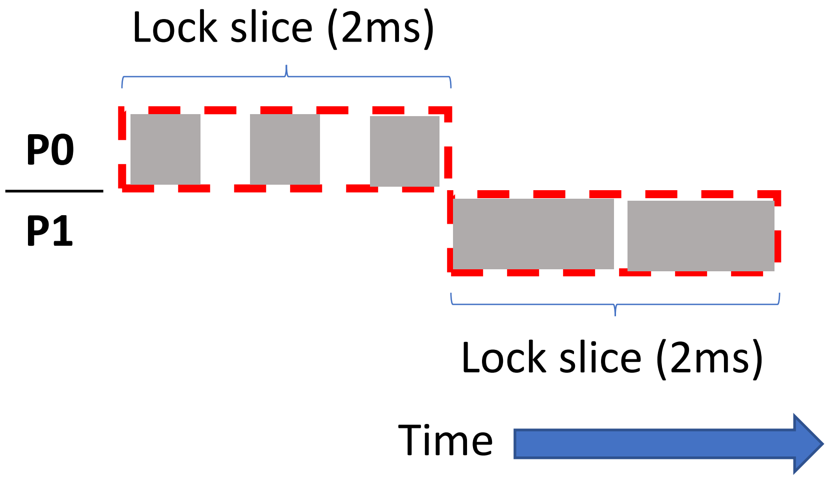
- Lock slice – dedicated window of opportunity to every user
- Owner can acquire lock multiple times within a slice without penalty



Slice ownership transferred to P1

#3 – Dedicated window of opportunity

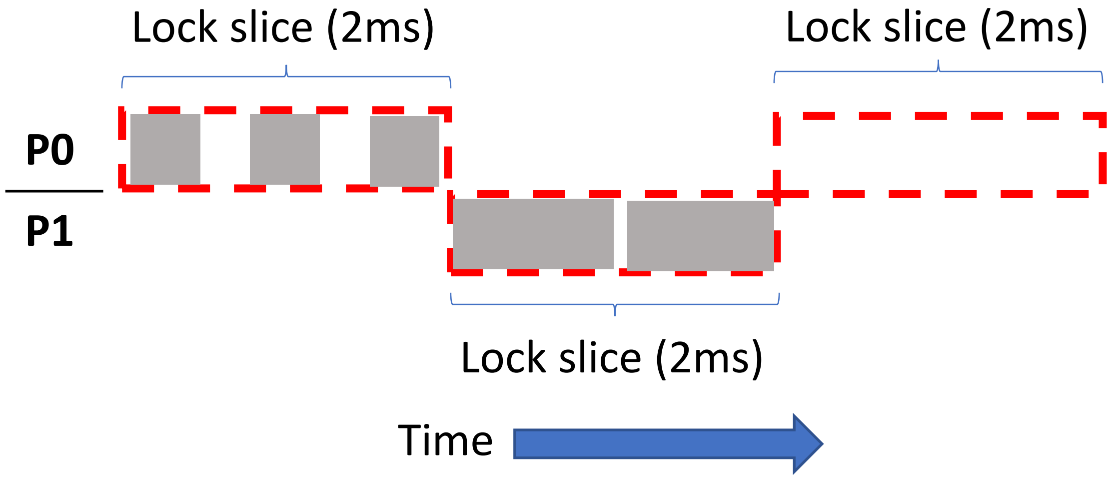
- Lock slice – dedicated window of opportunity to every user
- Owner can acquire lock multiple times within a slice without penalty



Size of individual critical section can vary

#3 – Dedicated window of opportunity

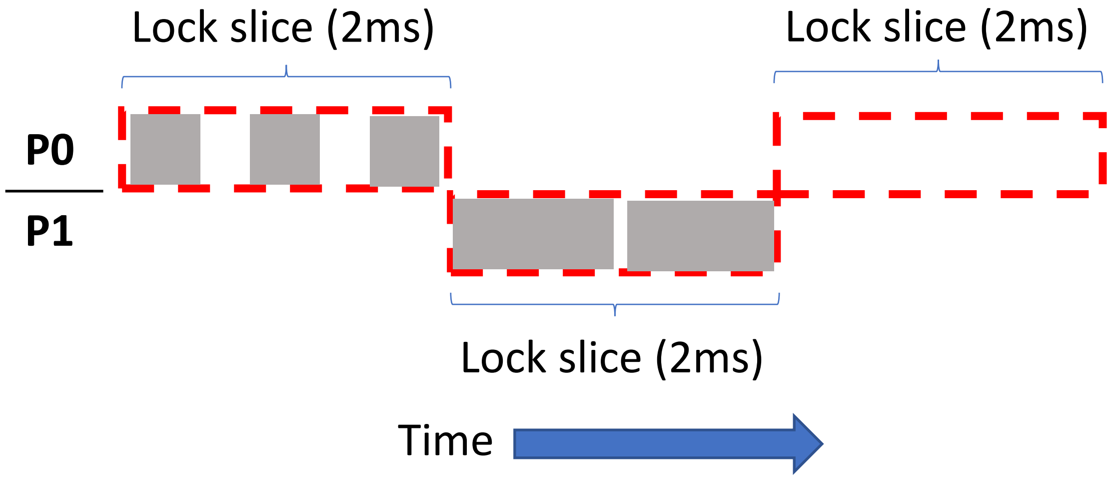
- Lock slice – dedicated window of opportunity to every user
- Owner can acquire lock multiple times within a slice without penalty
- Slice ownership alternates between users



Wait-times depends on lock slice size

#3 – Dedicated window of opportunity

- Lock slice – dedicated window of opportunity to every user
- Owner can acquire lock multiple times within a slice without penalty
- Slice ownership alternates between users



Lock slice

- Fixed-sized virtual critical section
- Transferred to next owner based on scheduling policy

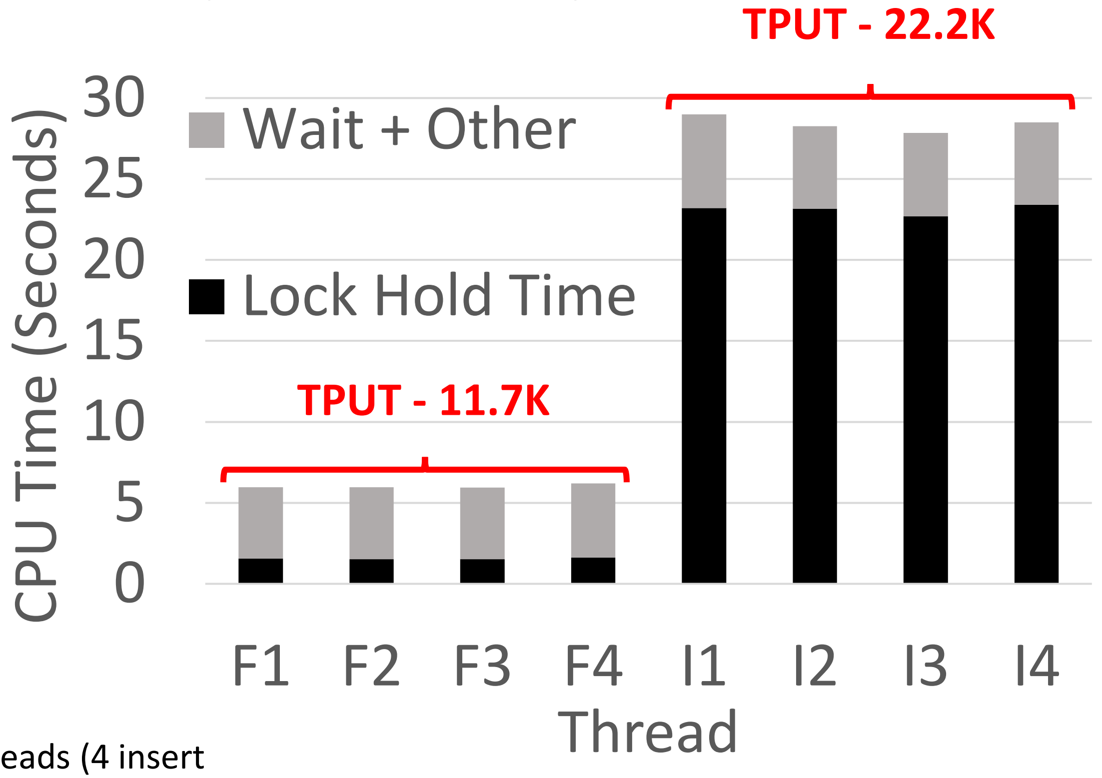
SCLs Implementation

- Three different implementations
  - u-SCL – User-space mutex replacement
  - RW-SCL – Reader-Writer Scheduler-Cooperative Lock
  - k-SCL – Kernel version of u-SCL
- New and existing optimization techniques
- u-SCL
  - Spin-and-park – To minimize CPU time spent while waiting
  - Next-thread prefetch – Next owner ready before slice ownership handoff
- RW-SCL
  - Per NUMA node counters
- More details in paper

- Introduction
- The Problem – Scheduler Subversion
- The Solution – Scheduler-Cooperative Locks
- Evaluation
- Conclusion

Evaluation

- Same UpScaleDB experiment

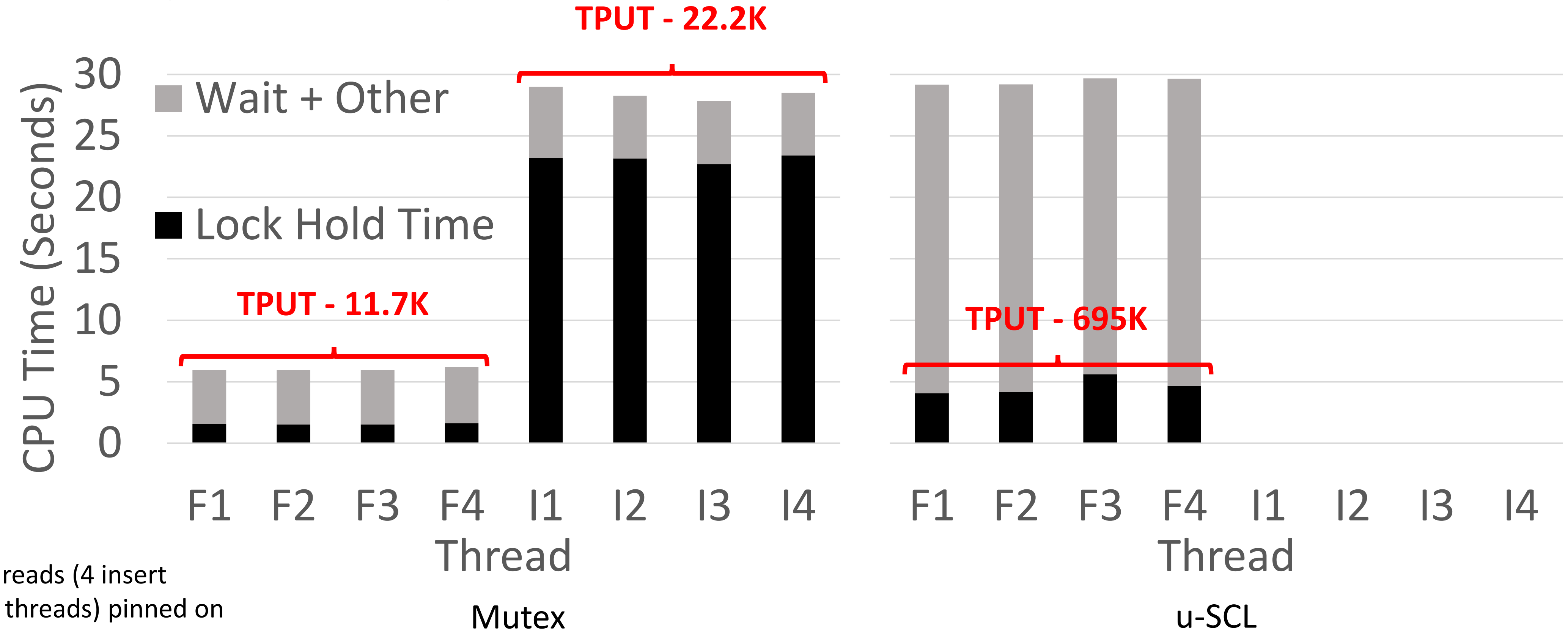


Mutex

Workload – 8 threads (4 insert threads + 4 find threads) pinned on 4 CPU, equal CPU allocation

Evaluation

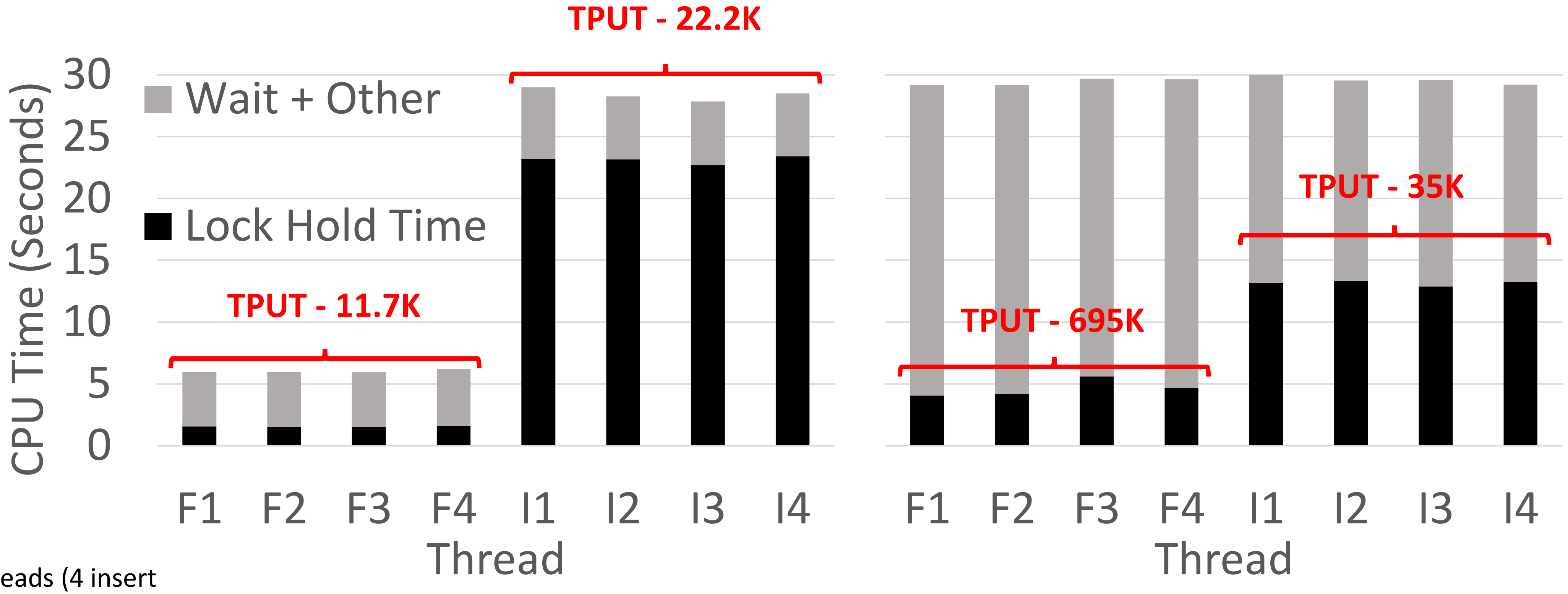
- Same UpScaleDB experiment



Workload – 8 threads (4 insert threads + 4 find threads) pinned on 4 CPU, equal CPU allocation

Evaluation

- Same UpScaleDB experiment



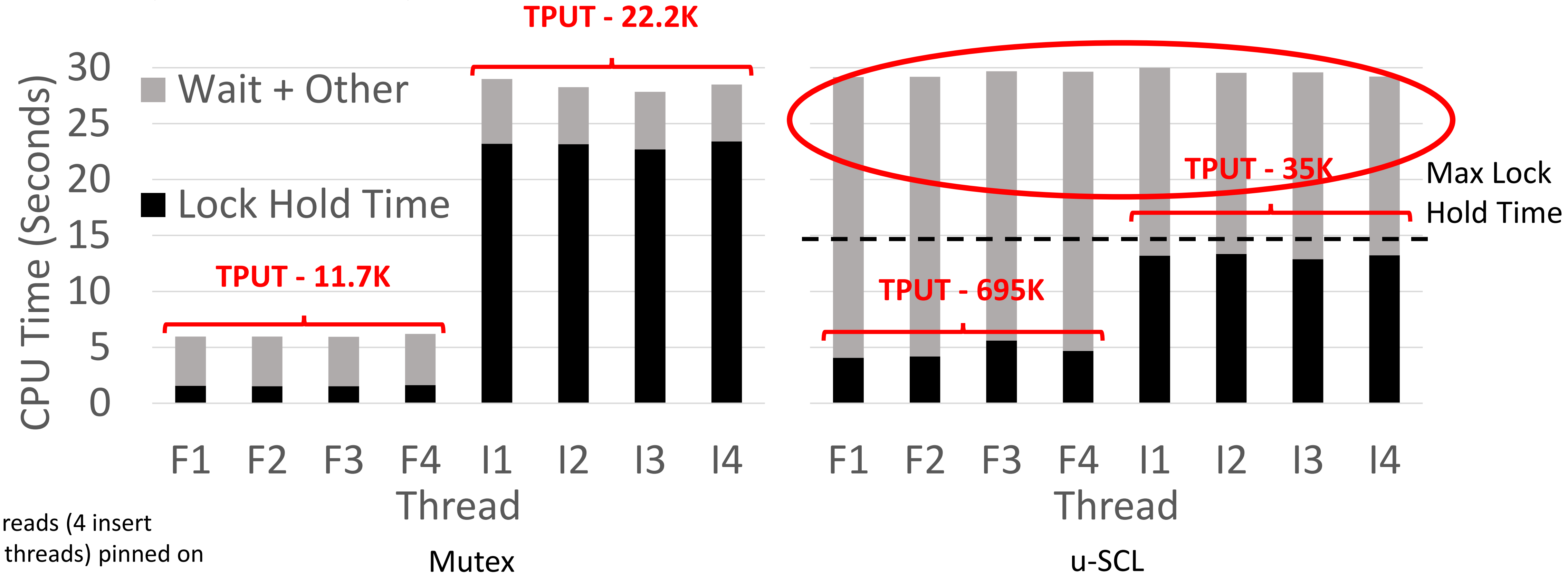
Mutex

u-SCL

Workload – 8 threads (4 insert threads + 4 find threads) pinned on 4 CPU, equal CPU allocation

Evaluation

- Same UpScaleDB experiment



Workload – 8 threads (4 insert threads + 4 find threads) pinned on 4 CPU, equal CPU allocation

Results summary

- Lock usage fairness – Allocate CPU proportionally even in extreme cases
- Lock overhead - Efficient and scales well up to 32 CPU
- Lock slice sizes vs. Performance
  - Large slice size – Higher throughput
  - Small slice size – Low Latency
- Demonstrate real-world utility of SCLs
  - Port RW-SCL to KyotoCabinet
  - Replace global file-system rename lock with k-SCL in Linux kernel

- Introduction
- The Problem – Scheduler Subversion
- The Solution – Scheduler-Cooperative Locks
- Evaluation
- Conclusion

Conclusion

- Lock usage determines CPU allocation subverting scheduling goals
- Introduce Scheduler-Cooperative Locks (SCL) to address the problem
- Evaluation shows the performance characteristics and versatility of SCLs
- Future work – Build SCLs that support other scheduling goals

Source - https://research.cs.wisc.edu/adsl/Software/

Thank you ☺