RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

FLEXGUARD: FAST MUTUAL EXCLUSION INDEPENDENT OF SUBSCRIPTION

→ → → TO APPEAR AT SOSP '25, SEOUL, SOUTH KOREA

Victor Laforet*, Sanidhya Kashyap†, Călin Iorgulescu‡, Julia Lawall*, Jean-Pierre Lozi*

* Inria, Paris, France
† EPFL, Lausanne, Switzerland
‡ Oracle Labs, Zürich, Switzerland

Background: blocking locks vs. spinlocks

- **Blocking locks:**
  - Most common locks, e.g., `pthread_mutex_lock()`

Background: blocking locks vs. spinlocks

- **Blocking locks:**
  - Most common locks, e.g., `pthread_mutex_lock()`
  - A thread fails to acquire the lock: it blocks with the FUTEX syscall

Background: blocking locks vs. spinlocks

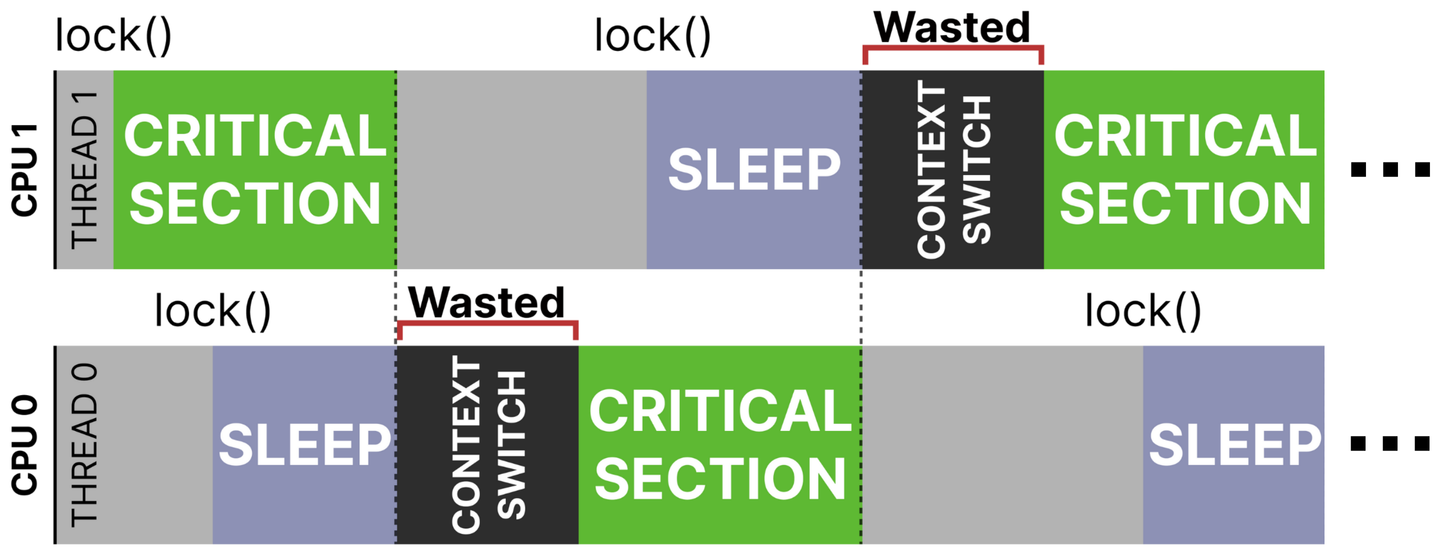
- **Blocking locks:**
  - Most common locks, e.g., `pthread_mutex_lock()`
  - A thread fails to acquire the lock: it blocks with the FUTEX syscall
  - A thread releases the lock: it wakes up the next one

Background: blocking locks vs. spinlocks

- **Blocking locks:**
  - Most common locks, e.g., `pthread_mutex_lock()`
  - A thread fails to acquire the lock: it blocks with the FUTEX syscall
  - A thread releases the lock: it wakes up the next one
  - **Problem:** slow, due to costly context switches on the critical path!
    - Not much you can do to speed them up…

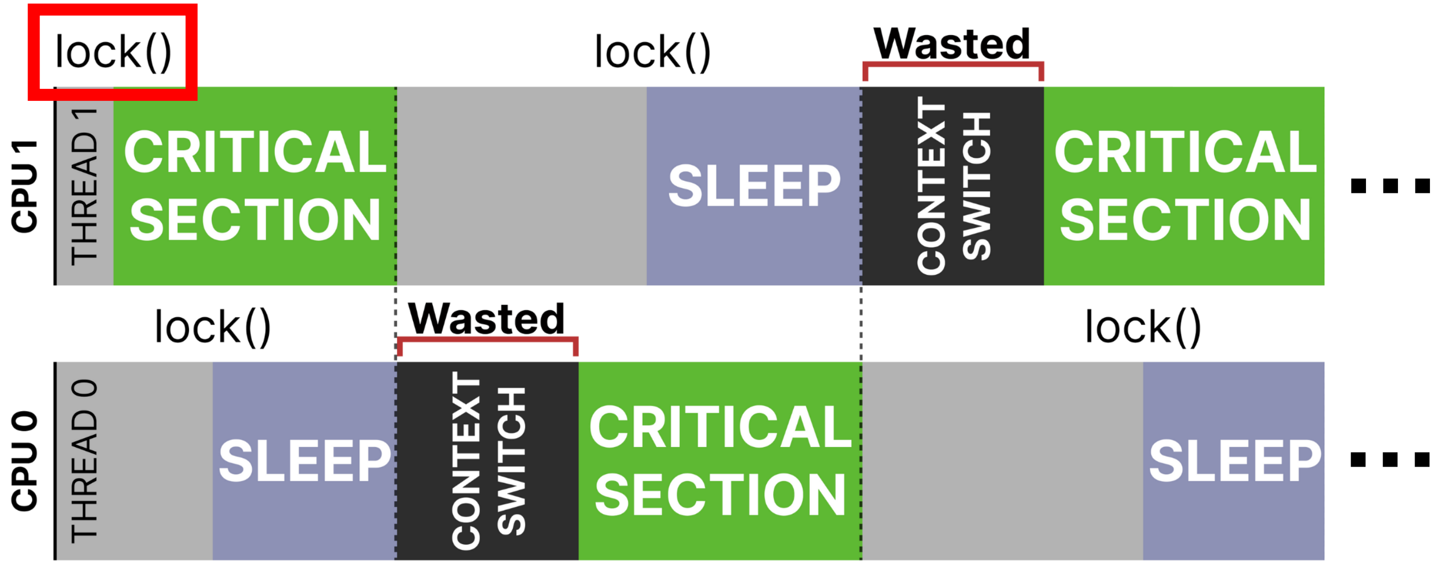
Background: blocking locks vs. spinlocks

- **Blocking locks:**
  - Most common locks, e.g., `pthread_mutex_lock()`
  - A thread fails to acquire the lock: it blocks with the `FUTEX` syscall
  - A thread releases the lock: it wakes up the next one
  - **Problem:** slow, due to costly context switches on the critical path!
    - Not much you can do to speed them up…



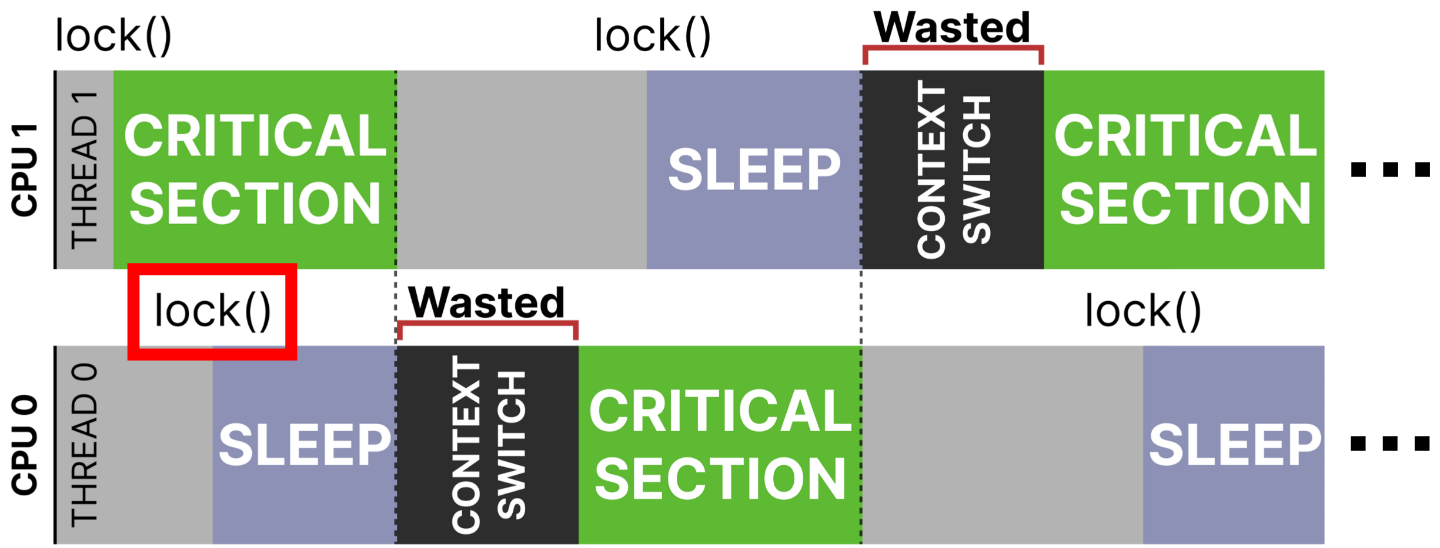
Background: blocking locks vs. spinlocks

- **Blocking locks:**
  - Most common locks, e.g., `pthread_mutex_lock()`
  - A thread fails to acquire the lock: it blocks with the FUTEX syscall
  - A thread releases the lock: it wakes up the next one
  - **Problem:** slow, due to costly context switches on the critical path!
    - Not much you can do to speed them up…



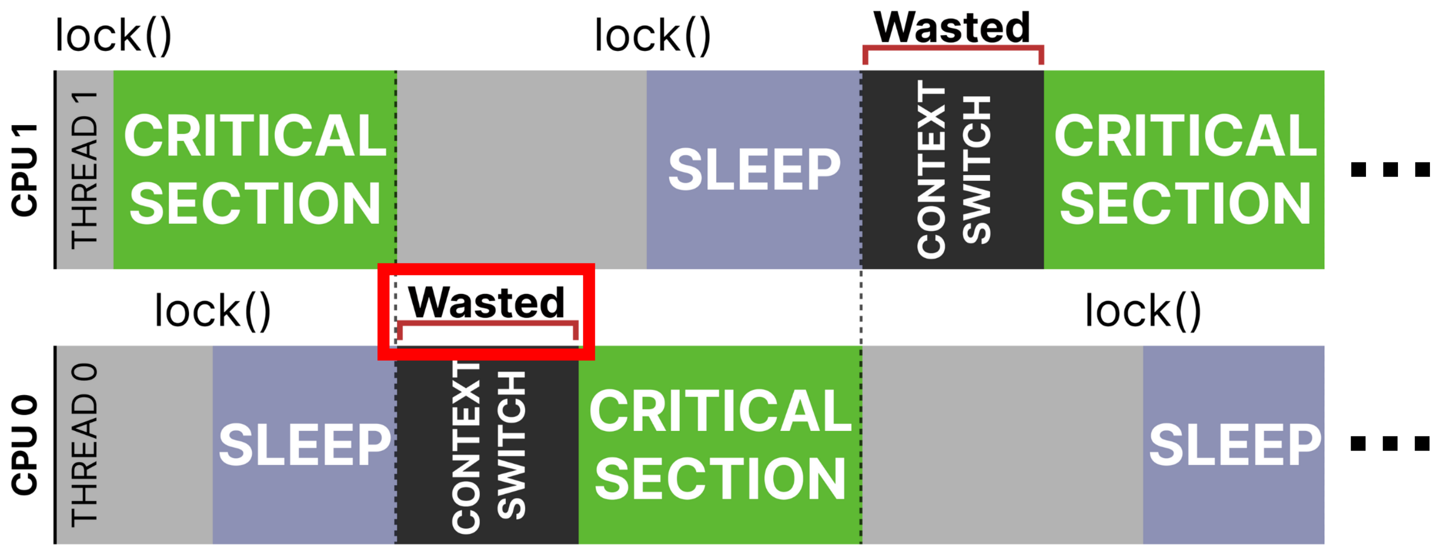
Background: blocking locks vs. spinlocks

- **Blocking locks:**
  - Most common locks, e.g., `pthread_mutex_lock()`
  - A thread fails to acquire the lock: it blocks with the FUTEX syscall
  - A thread releases the lock: it wakes up the next one
  - **Problem:** slow, due to costly context switches on the critical path!
    - Not much you can do to speed them up…



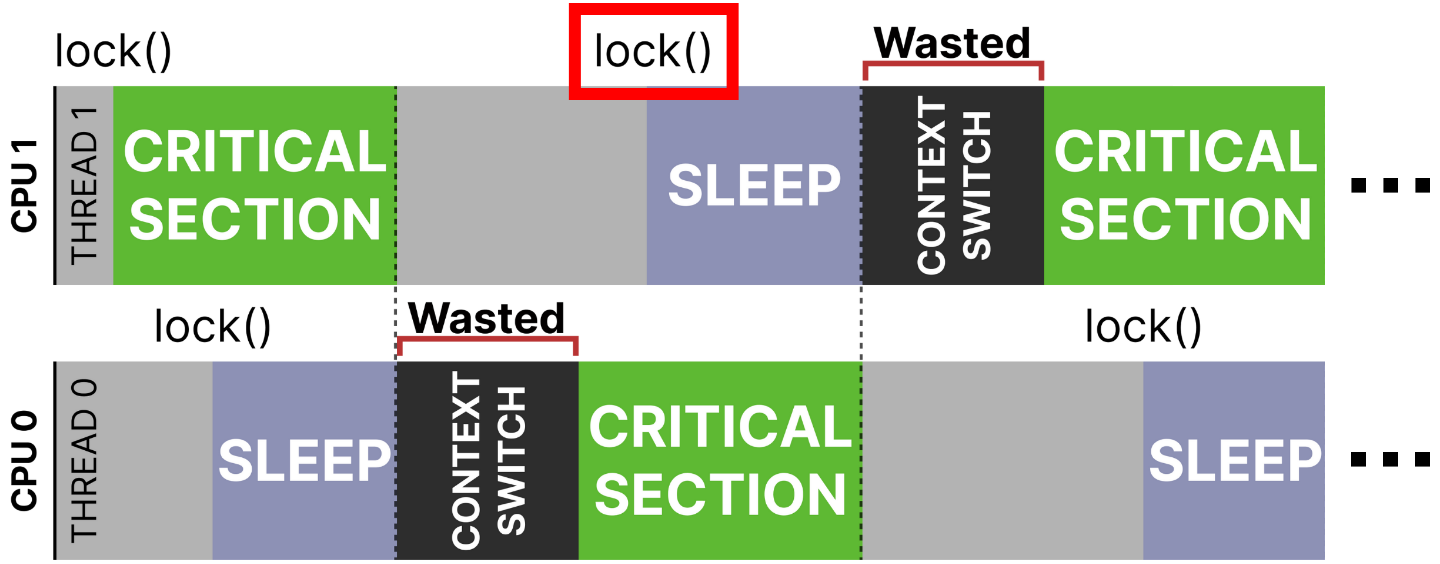
Background: blocking locks vs. spinlocks

- **Blocking locks:**
  - Most common locks, e.g., `pthread_mutex_lock()`
  - A thread fails to acquire the lock: it blocks with the FUTEX syscall
  - A thread releases the lock: it wakes up the next one
  - **Problem:** slow, due to costly context switches on the critical path!
    - Not much you can do to speed them up...



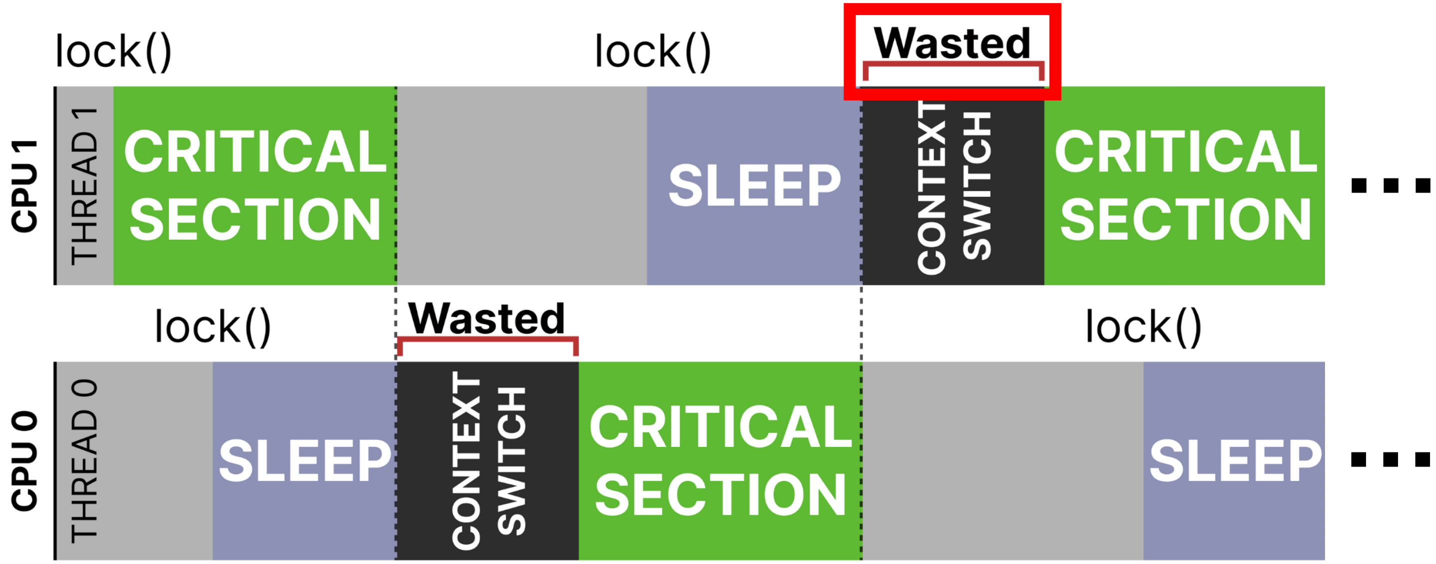
Background: blocking locks vs. spinlocks

- **Blocking locks:**
  - Most common locks, e.g., `pthread_mutex_lock()`
  - A thread fails to acquire the lock: it blocks with the FUTEX syscall
  - A thread releases the lock: it wakes up the next one
  - **Problem:** slow, due to costly context switches on the critical path!
    - Not much you can do to speed them up…



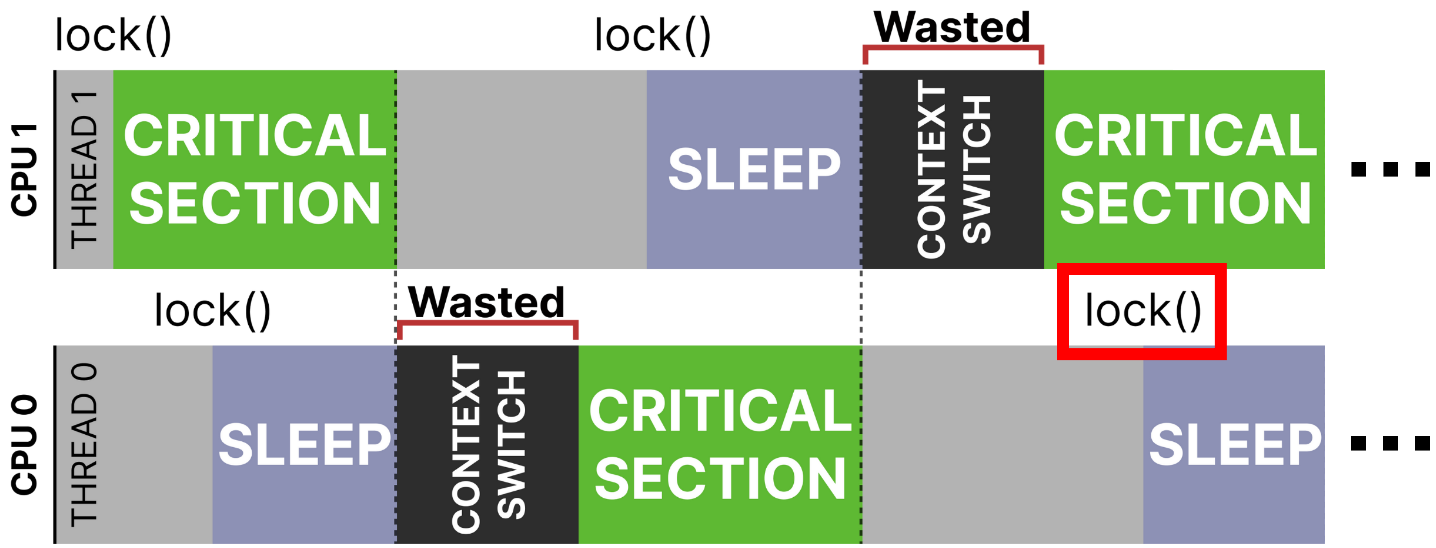
Background: blocking locks vs. spinlocks

- **Blocking locks:**
  - Most common locks, e.g., `pthread_mutex_lock()`
  - A thread fails to acquire the lock: it blocks with the `FUTEX` syscall
  - A thread releases the lock: it wakes up the next one
  - **Problem:** slow, due to costly context switches on the critical path!
    - Not much you can do to speed them up…



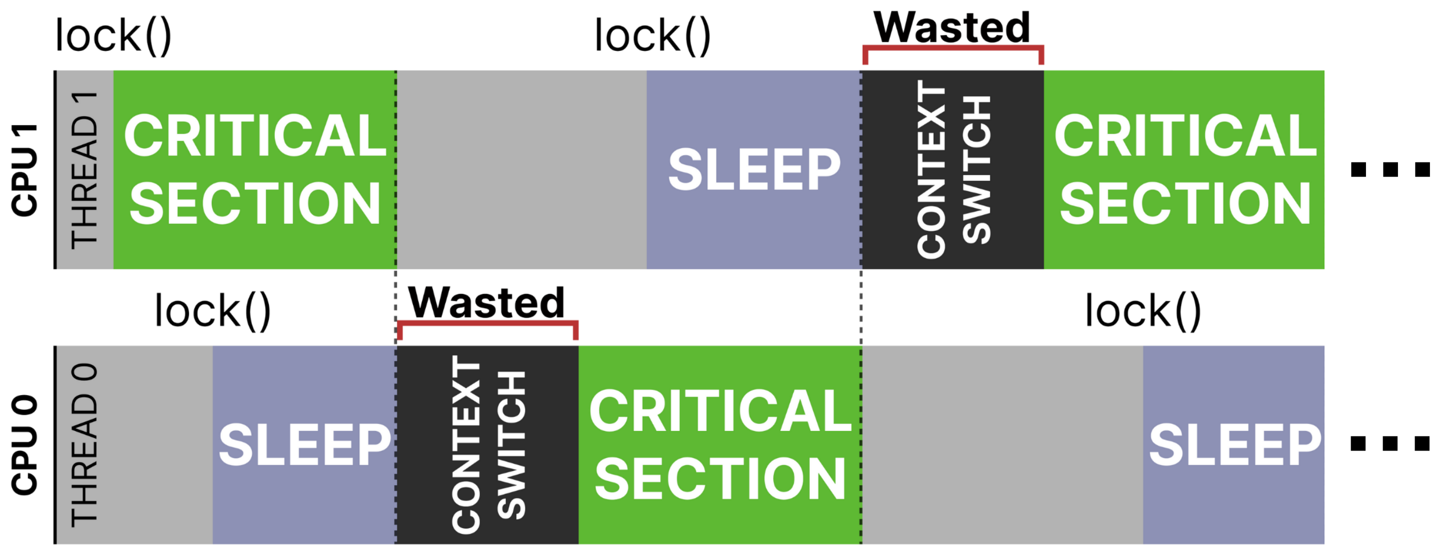
Background: blocking locks vs. spinlocks

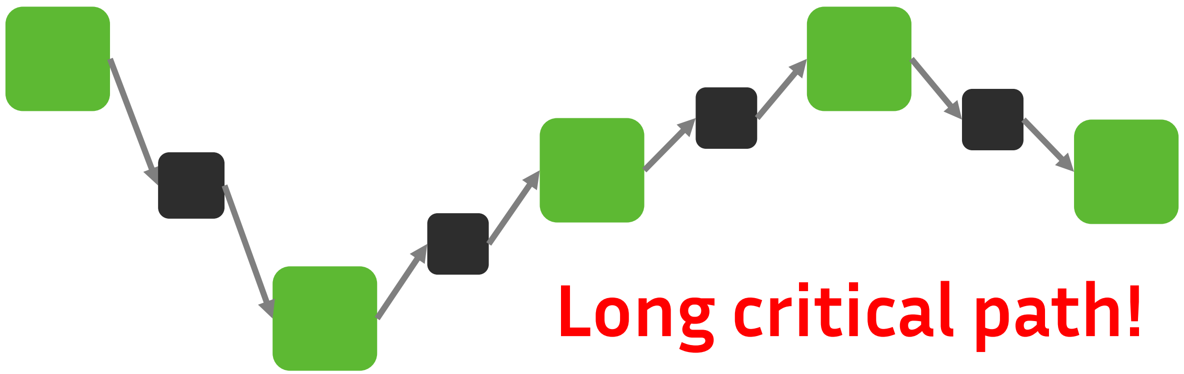
- **Blocking locks:**
  - Most common locks, e.g., `pthread_mutex_lock()`
  - A thread fails to acquire the lock: it blocks with the FUTEX syscall
  - A thread releases the lock: it wakes up the next one
  - **Problem:** slow, due to costly context switches on the critical path!
    - Not much you can do to speed them up…



Background: blocking locks vs. spinlocks

- **Blocking locks:**
  - Most common locks, e.g., `pthread_mutex_lock()`
  - A thread fails to acquire the lock: it blocks with the FUTEX syscall
  - A thread releases the lock: it wakes up the next one
  - **Problem:** slow, due to costly context switches on the critical path!
    - Not much you can do to speed them up…





Blocking locks vs. spinlocks

- **Spinlocks:**
  - Instead of blocking, spin (busy-wait)!

Blocking locks vs. spinlocks

```
lock() {
  while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
    PAUSE;
}

unlock() {
  lock = UNLOCKED;
}
```

- **Spinlocks:**
  - Instead of blocking, spin (busy-wait)!

Blocking locks vs. spinlocks

```
lock() {
  while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
    PAUSE;
}

unlock() {
  lock = UNLOCKED;
}
```

- **Spinlocks:**
  - Instead of blocking, spin (busy-wait)!
  - Transitions between critical sections much faster: one cache miss!

Blocking locks vs. spinlocks

```
lock() {
  while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
    PAUSE;
}

unlock() {
  lock = UNLOCKED;
}
```

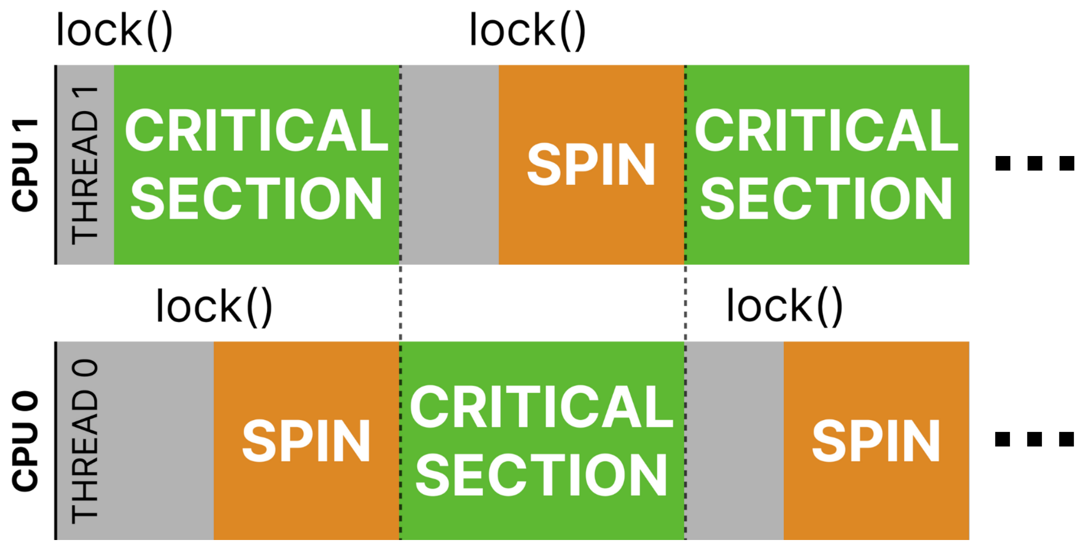
- **Spinlocks:**
  - Instead of blocking, spin (busy-wait)!
  - Transitions between critical sections much faster: one cache miss!
  - Lots of research in this area, many very fast spinlock algorithms!
    - TATAS locks, ticket locks, queue locks, NUMA locks, delegation locks…

Blocking locks vs. spinlocks

```
lock() {
  while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
    PAUSE;
}

unlock() {
  lock = UNLOCKED;
}
```

- **Spinlocks:**
  - Instead of blocking, spin (busy-wait)!
  - Transitions between critical sections much faster: one cache miss!
  - Lots of research in this area, many very fast spinlock algorithms!
    - TATAS locks, ticket locks, queue locks, NUMA locks, delegation locks…

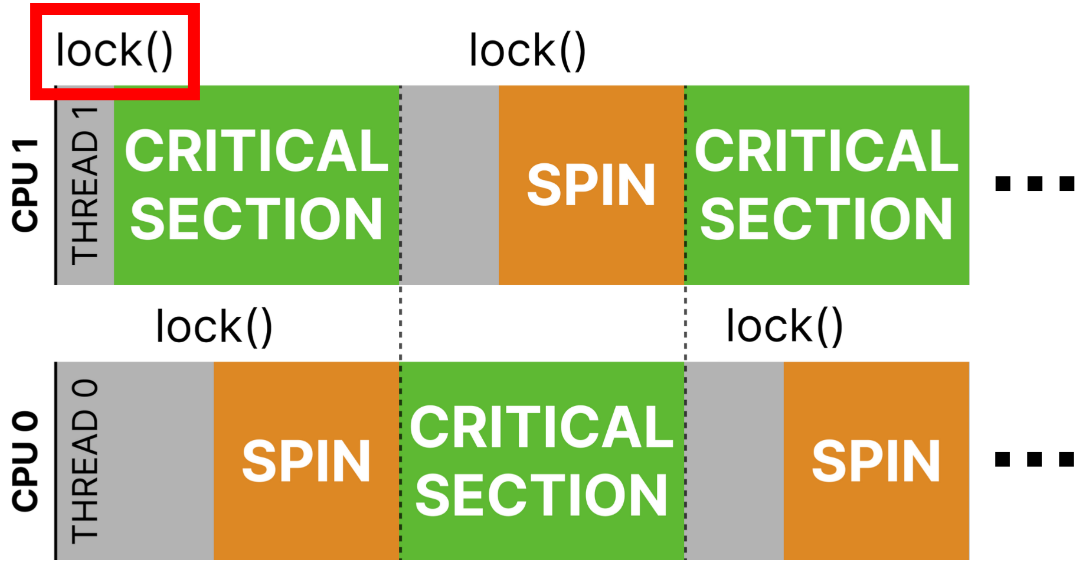


Blocking locks vs. spinlocks

```
lock() {
  while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
    PAUSE;
}

unlock() {
  lock = UNLOCKED;
}
```

- **Spinlocks:**
  - Instead of blocking, spin (busy-wait)!
  - Transitions between critical sections much faster: one cache miss!
  - Lots of research in this area, many very fast spinlock algorithms!
    - TATAS locks, ticket locks, queue locks, NUMA locks, delegation locks...

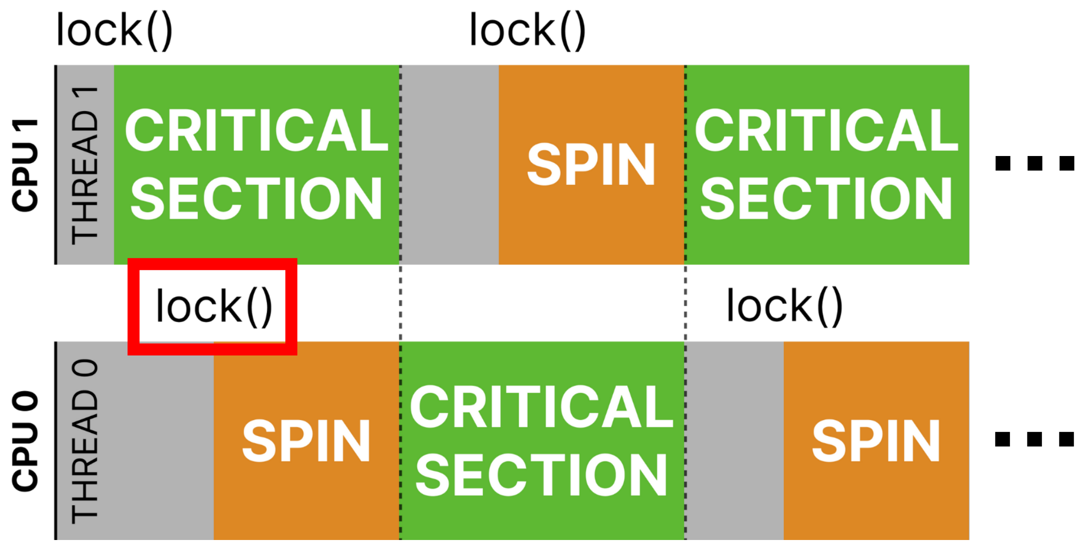


Blocking locks vs. spinlocks

```
lock() {
  while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
    PAUSE;
}

unlock() {
  lock = UNLOCKED;
}
```

- **Spinlocks:**
  - Instead of blocking, spin (busy-wait)!
  - Transitions between critical sections much faster: one cache miss!
  - Lots of research in this area, many very fast spinlock algorithms!
    - TATAS locks, ticket locks, queue locks, NUMA locks, delegation locks…

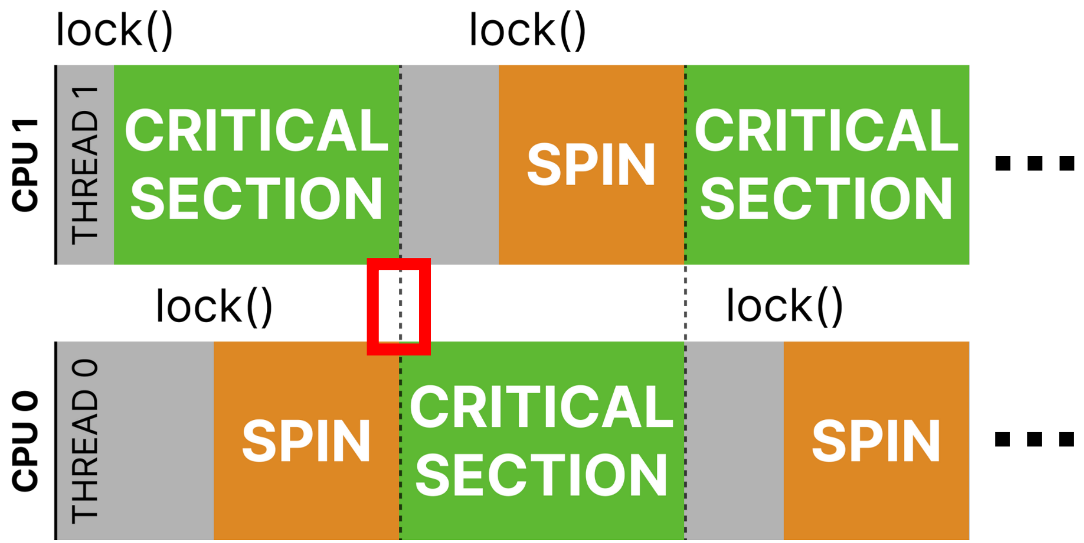


Blocking locks vs. spinlocks

```
lock() {
  while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
    PAUSE;
}

unlock() {
  lock = UNLOCKED;
}
```

- **Spinlocks:**
  - Instead of blocking, spin (busy-wait)!
  - Transitions between critical sections much faster: one cache miss!
  - Lots of research in this area, many very fast spinlock algorithms!
    - TATAS locks, ticket locks, queue locks, NUMA locks, delegation locks…

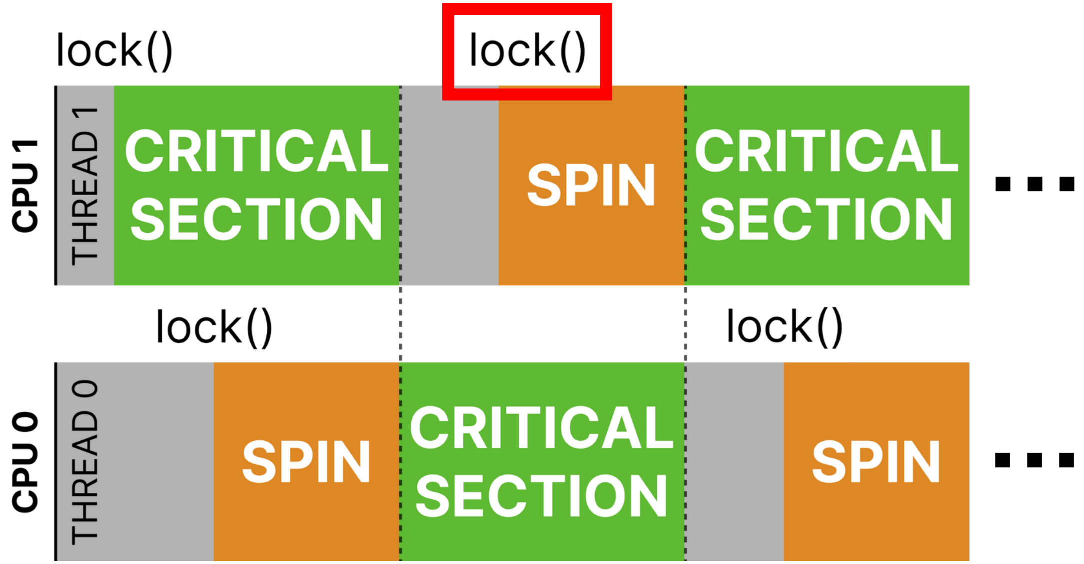


Blocking locks vs. spinlocks

```
lock() {
  while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
    PAUSE;
}

unlock() {
  lock = UNLOCKED;
}
```

- **Spinlocks:**
  - Instead of blocking, spin (busy-wait)!
  - Transitions between critical sections much faster: one cache miss!
  - Lots of research in this area, many very fast spinlock algorithms!
    - TATAS locks, ticket locks, queue locks, NUMA locks, delegation locks...

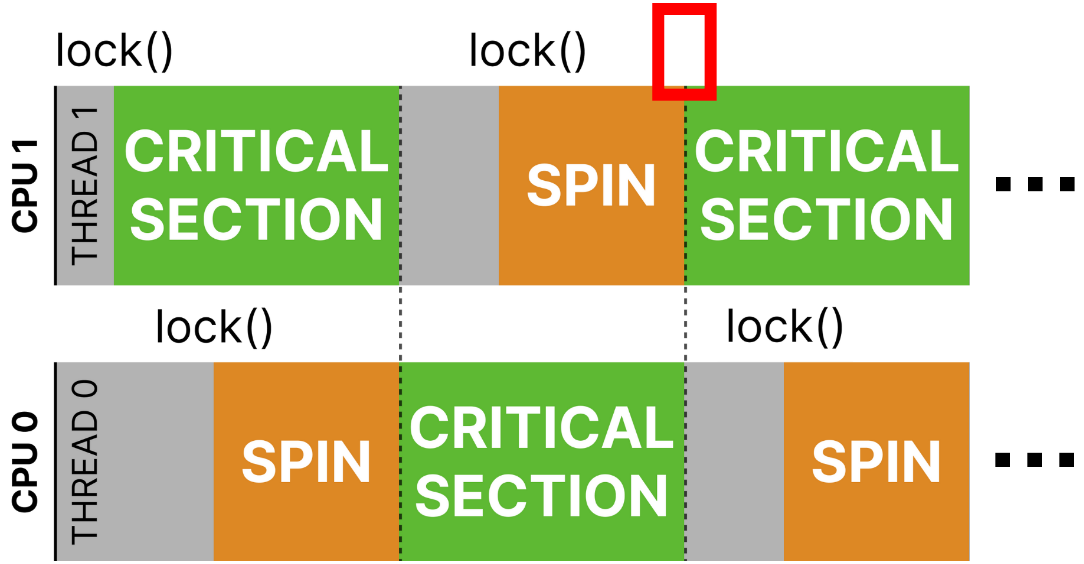


Blocking locks vs. spinlocks

```
lock() {
  while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
    PAUSE;
}

unlock() {
  lock = UNLOCKED;
}
```

- **Spinlocks:**
  - Instead of blocking, spin (busy-wait)!
  - Transitions between critical sections much faster: one cache miss!
  - Lots of research in this area, many very fast spinlock algorithms!
    - TATAS locks, ticket locks, queue locks, NUMA locks, delegation locks…

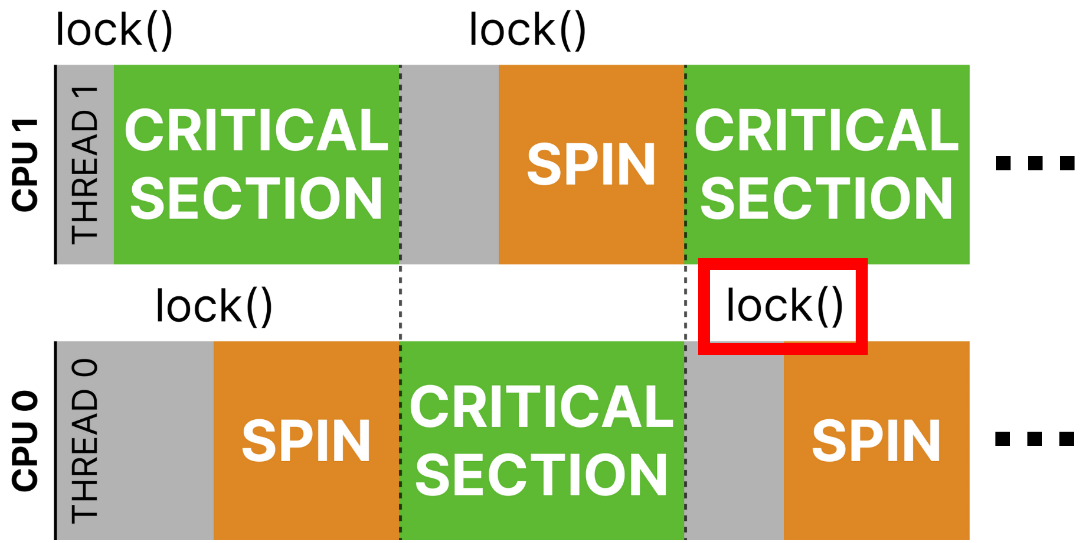


Blocking locks vs. spinlocks

```
lock() {
  while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
    PAUSE;
}

unlock() {
  lock = UNLOCKED;
}
```

- **Spinlocks:**
  - Instead of blocking, spin (busy-wait)!
  - Transitions between critical sections much faster: one cache miss!
  - Lots of research in this area, many very fast spinlock algorithms!
    - TATAS locks, ticket locks, queue locks, NUMA locks, delegation locks…

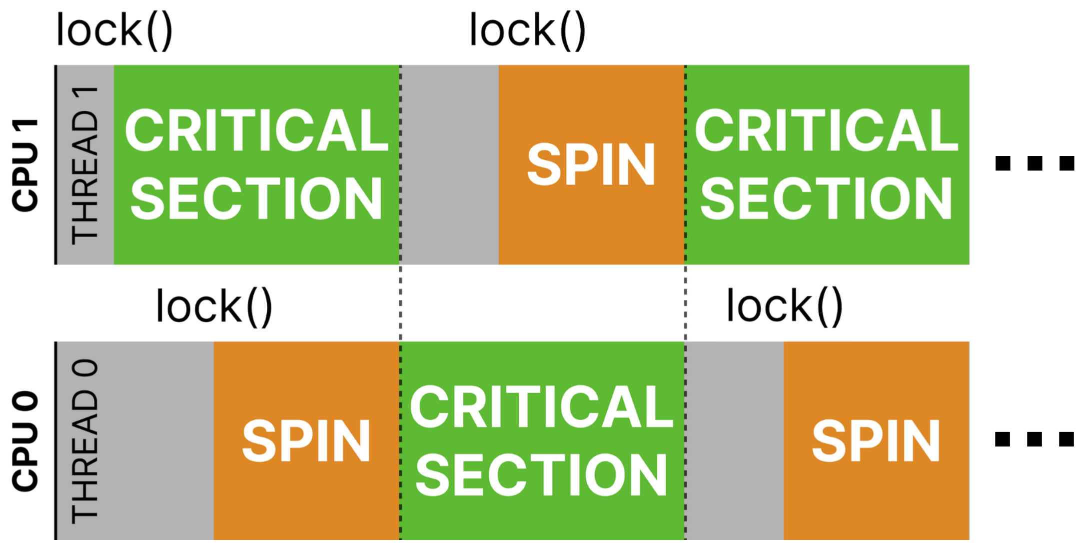


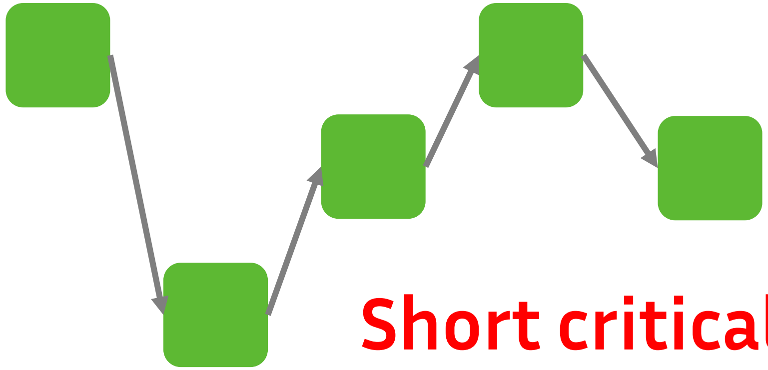
Blocking locks vs. spinlocks

```
lock() {
  while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
    PAUSE;
}

unlock() {
  lock = UNLOCKED;
}
```

- **Spinlocks:**
  - Instead of blocking, spin (busy-wait)!
  - Transitions between critical sections much faster: one cache miss!
  - Lots of research in this area, many very fast spinlock algorithms!
    - TATAS locks, ticket locks, queue locks, NUMA locks, delegation locks…





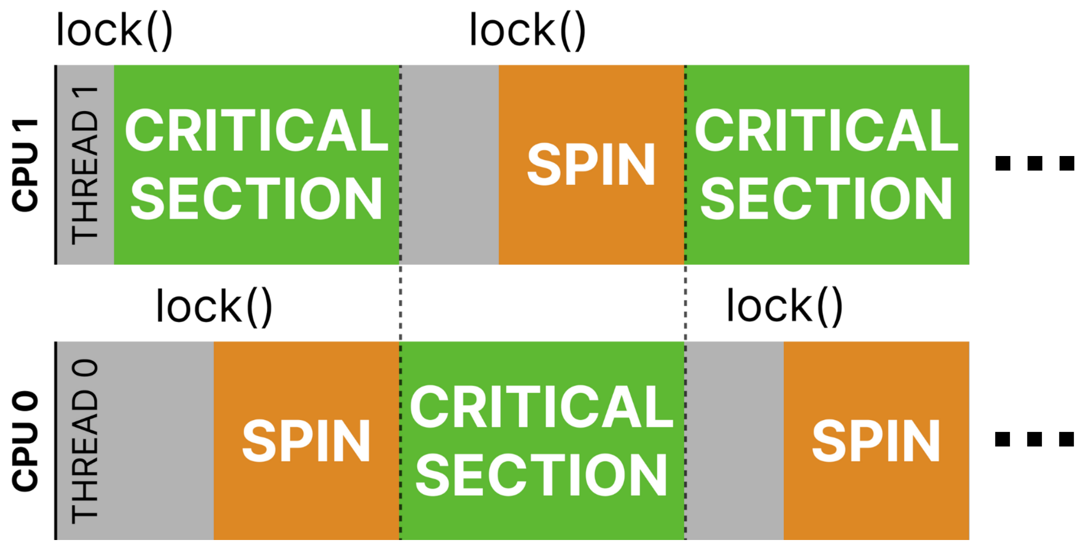
Short critical path!

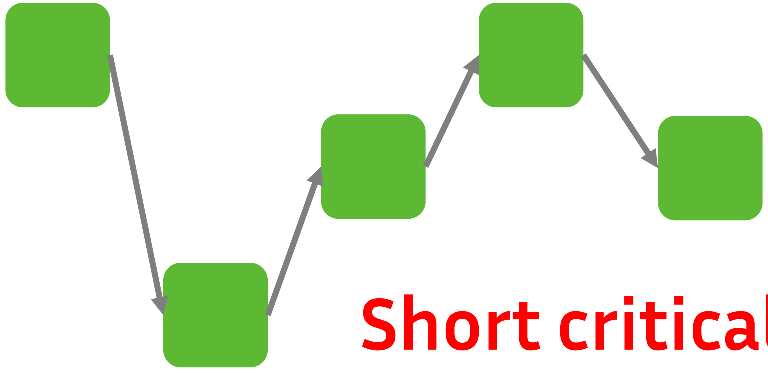
Blocking locks vs. spinlocks

```
lock() {
  while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
    PAUSE;
}

unlock() {
  lock = UNLOCKED;
}
```

- **Spinlocks:**
  - Instead of blocking, spin (busy-wait)!
  - Transitions between critical sections much faster: one cache miss!
  - Lots of research in this area, many very fast spinlock algorithms!
    - TATAS locks, ticket locks, queue locks, NUMA locks, delegation locks…





Short critical path!

- Spinning wastes energy? A few, but faster applications = lower energy consumption!

Blocking locks vs. spinlocks

- Why do standard libraries (e.g., POSIX) use blocking locks?

Blocking locks vs. spinlocks

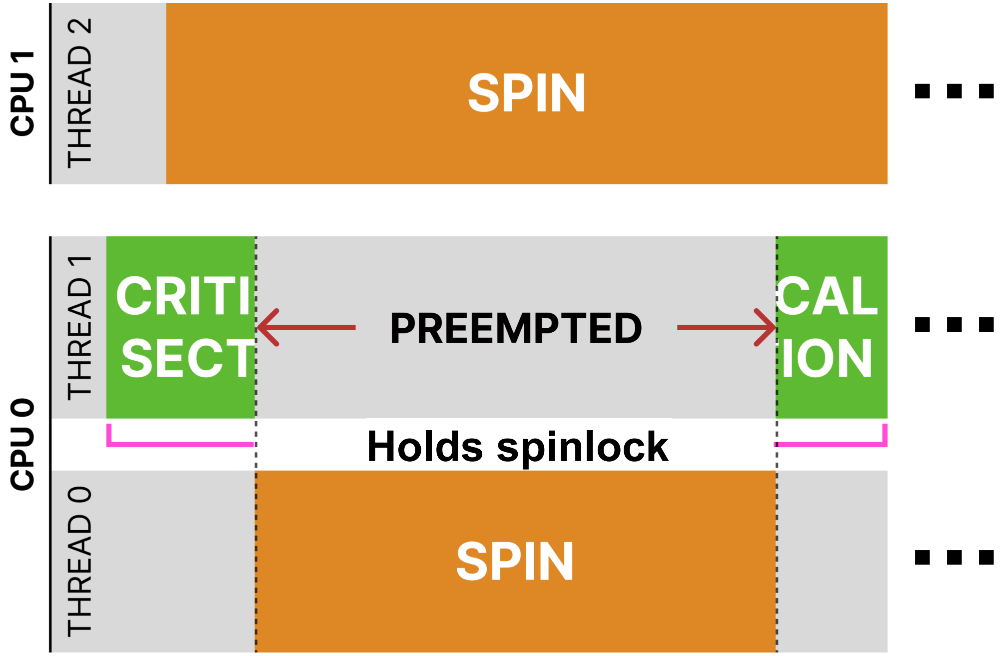
- **Why do standard libraries (e.g., POSIX) use blocking locks?**
  - **Answer:** stability!
    - Spinlocks perform great when # threads ≤ # hardware contexts
    - But when # threads > # hardware contexts, performance collapses!

Blocking locks vs. spinlocks

- **Why do standard libraries (e.g., POSIX) use blocking locks?**
  - **Answer:** stability!
    - Spinlocks perform great when # threads ≤ # hardware contexts
    - But when # threads > # hardware contexts, performance collapses!
  - **Reason:** spinners preempt the critical sections, stopping all progress on the critical path!

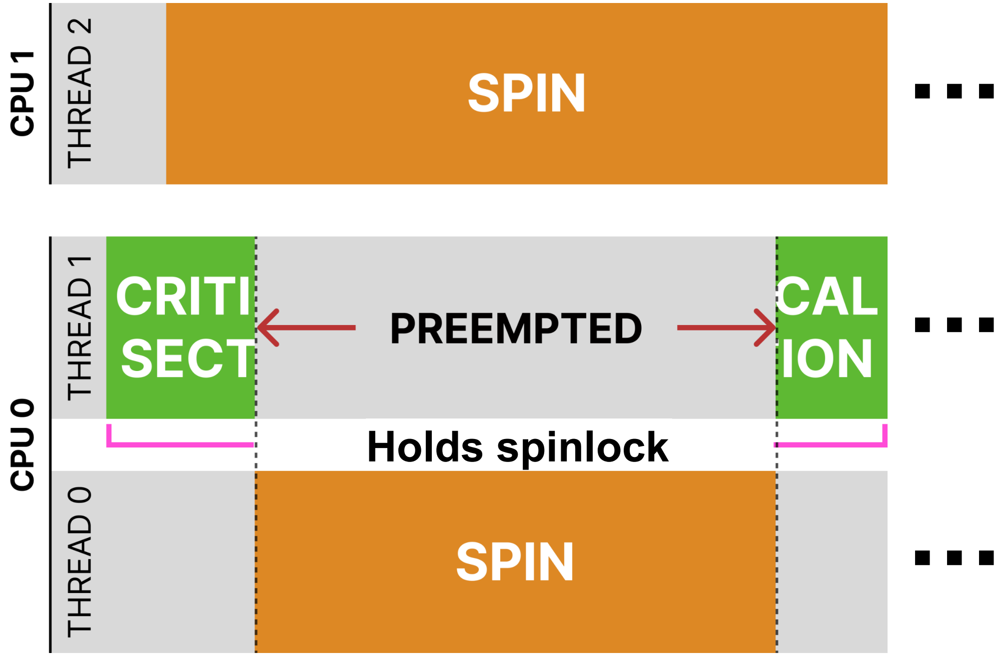
Blocking locks vs. spinlocks

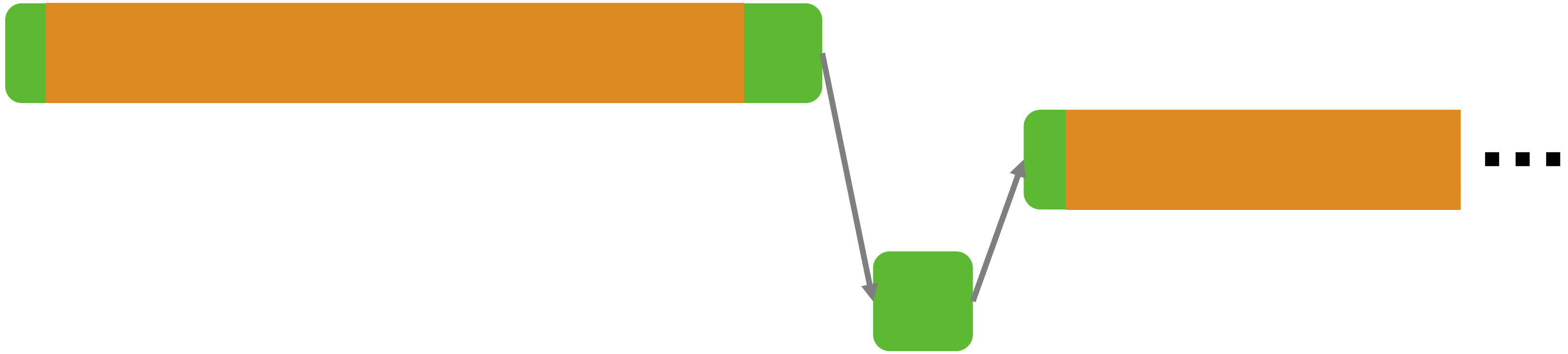
- Why do standard libraries (e.g., POSIX) use blocking locks?
  - **Answer:** stability!
    - Spinlocks perform great when # threads ≤ # hardware contexts
    - But when # threads > # hardware contexts, performance collapses!
  - **Reason:** spinners preempt the critical sections, stopping all progress on the critical path!



Blocking locks vs. spinlocks

- Why do standard libraries (e.g., POSIX) use blocking locks?
  - Answer: stability!
    - Spinlocks perform great when # threads ≤ # hardware contexts
    - But when # threads > # hardware contexts, performance collapses!
  - Reason: spinners preempt the critical sections, stopping all progress on the critical path!





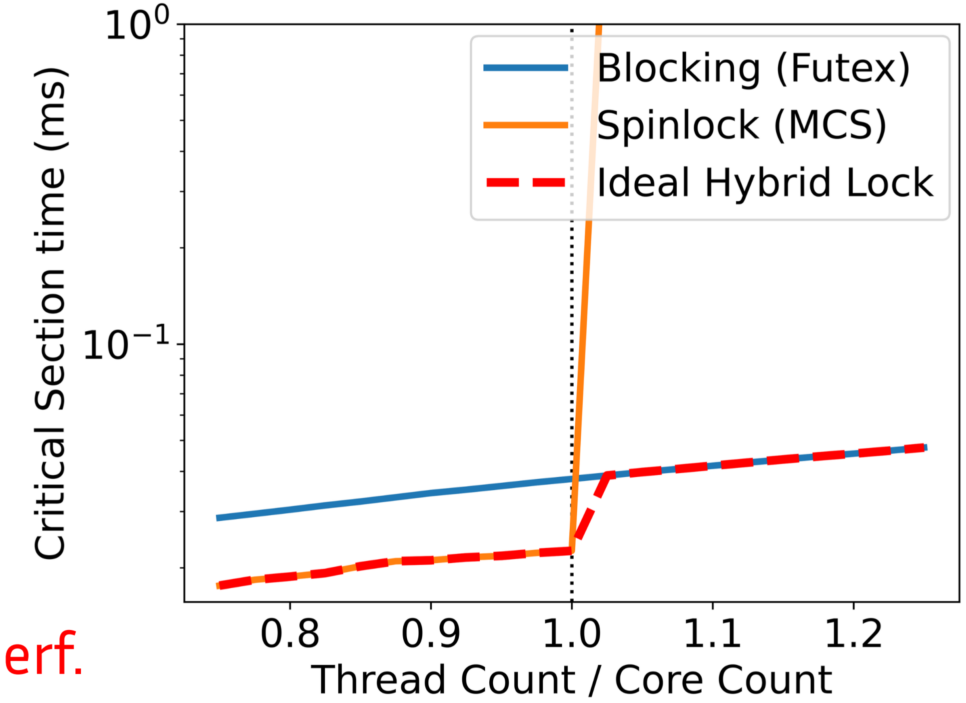
Extremely long critical path!

FlexGuard: the best of both worlds!

- **Goal:** get the best of both worlds!
  - When # threads ≤ available # hw ctxts, spinlock perf.
  - When # threads > available # hw ctxts, blocking lock perf.

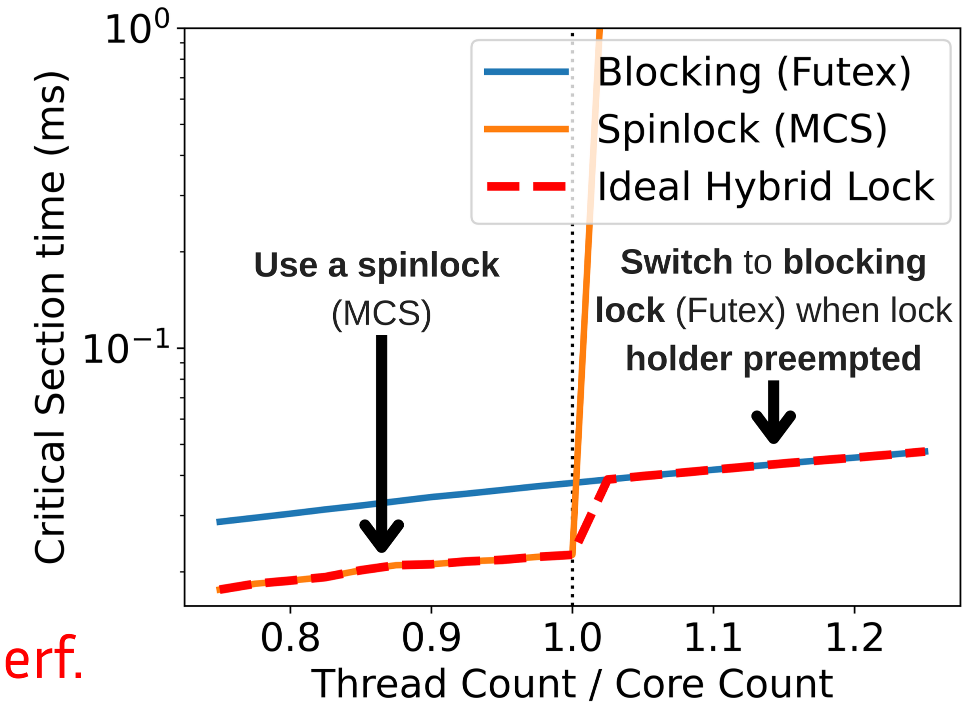
FlexGuard: the best of both worlds!

- **Goal:** get the best of both worlds!
  - When # threads ≤ available # hw ctxts, spinlock perf.
  - When # threads > available # hw ctxts, blocking lock perf.



Lower is better

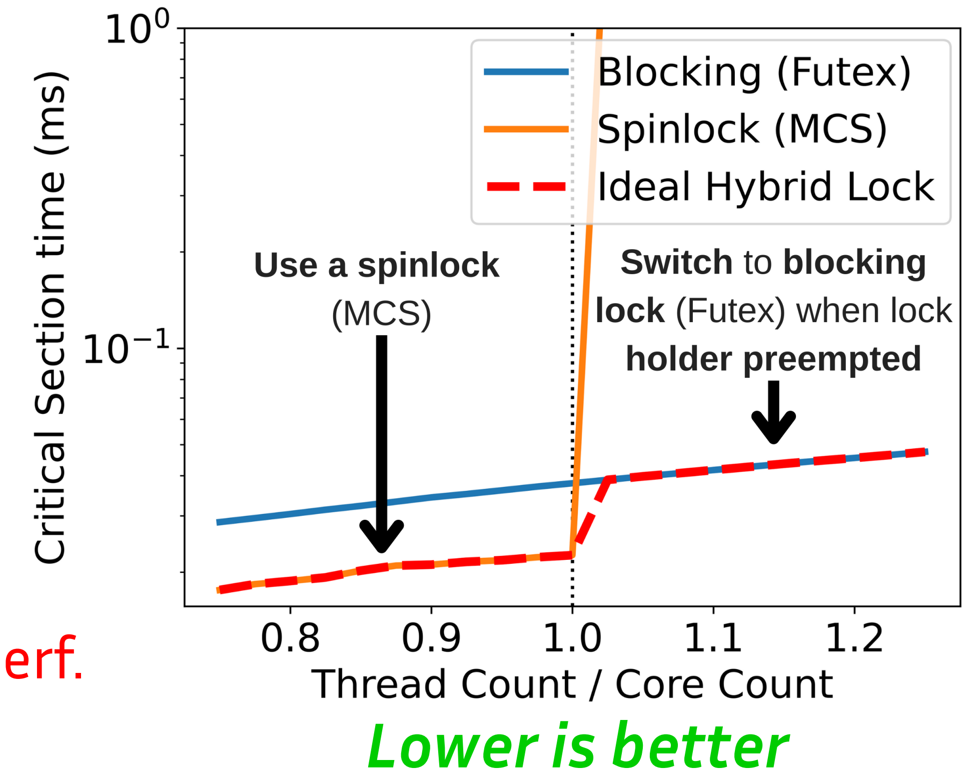
FlexGuard: the best of both worlds!



Lower is better

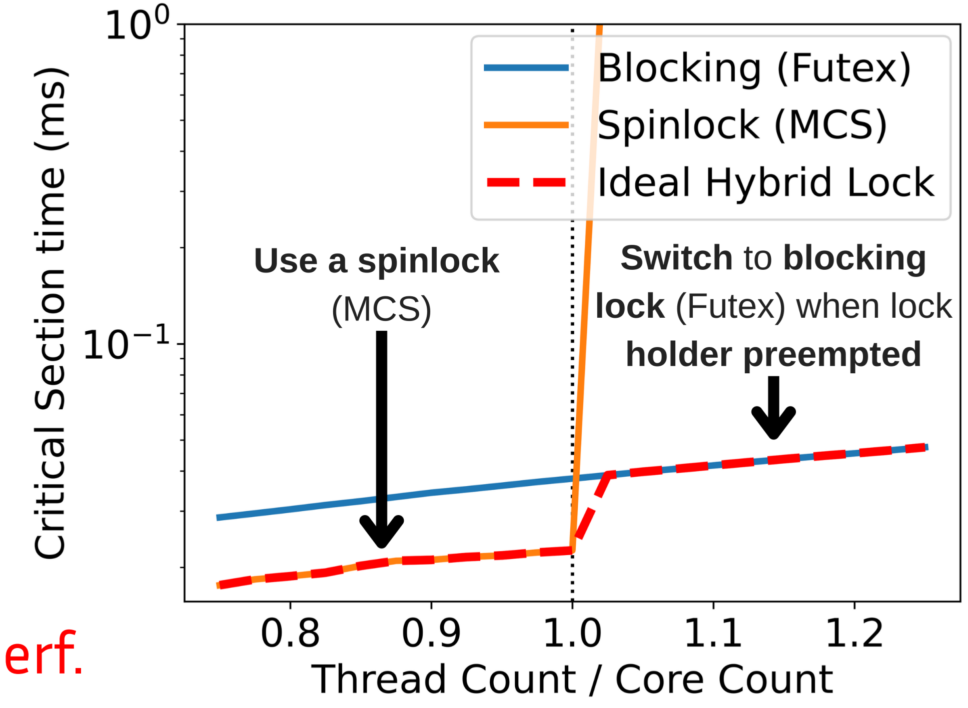
- **Goal**: get the best of both worlds!
  - When # threads ≤ available # hw ctxts, spinlock perf.
  - When # threads > available # hw ctxts, blocking lock perf.
- **Idea**: use a spinlock, when critical section preempted, switch to a blocking lock!

FlexGuard: the best of both worlds!



- **Goal:** get the best of both worlds!
  - When # threads ≤ available # hw ctxts, spinlock perf.
  - When # threads > available # hw ctxts, blocking lock perf.
- **Idea:** use a spinlock, when critical section preempted, switch to a blocking lock!
  - Can we do this?

FlexGuard: the best of both worlds!



Lower is better

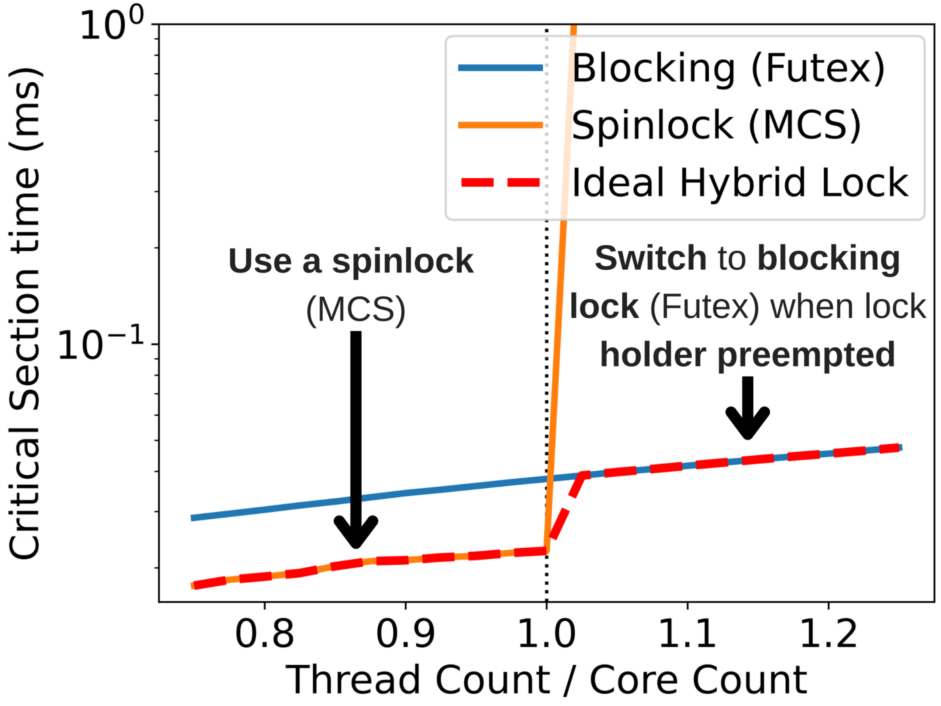
- **Goal:** get the best of both worlds!
  - When # threads ≤ available # hw ctxts, spinlock perf.
  - When # threads > available # hw ctxts, blocking lock perf.
- **Idea:** use a spinlock, when critical section preempted, switch to a blocking lock!
  - Can we do this?
- **Insight:** nowadays, with eBPF we can!
  - We can instrument context switches to see all preemptions
  - We can view the full state of the thread: preemption address + register contents

⇒ We can 100% tell whether we are in a critical section!

eBPF

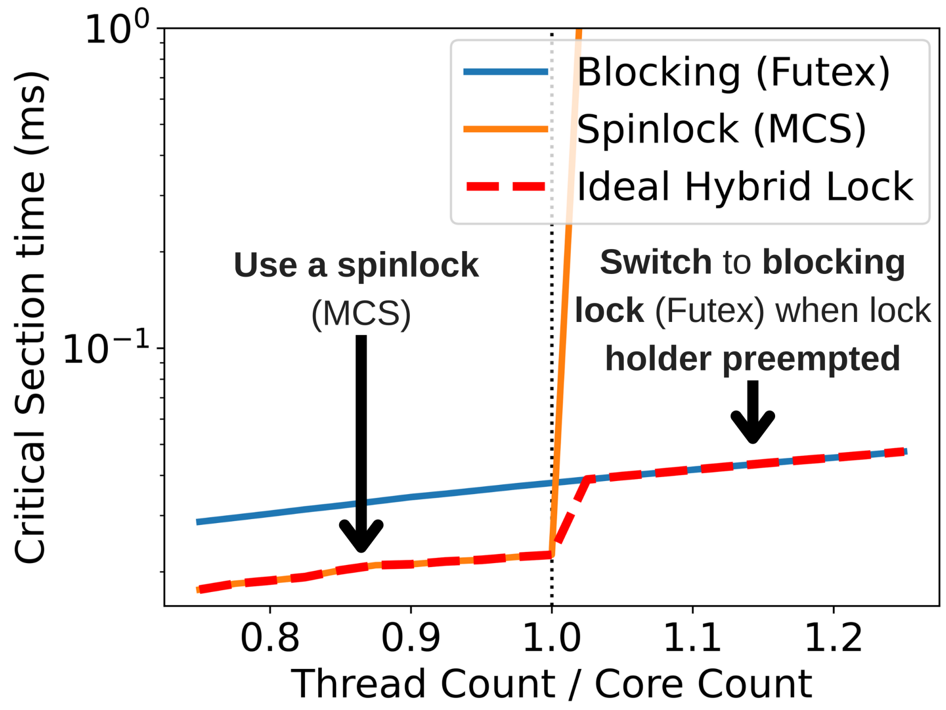
FlexGuard: the best of both worlds!

- Wait… Didn't others try to do this before?!
  - I.e., switch between spinning and blocking?

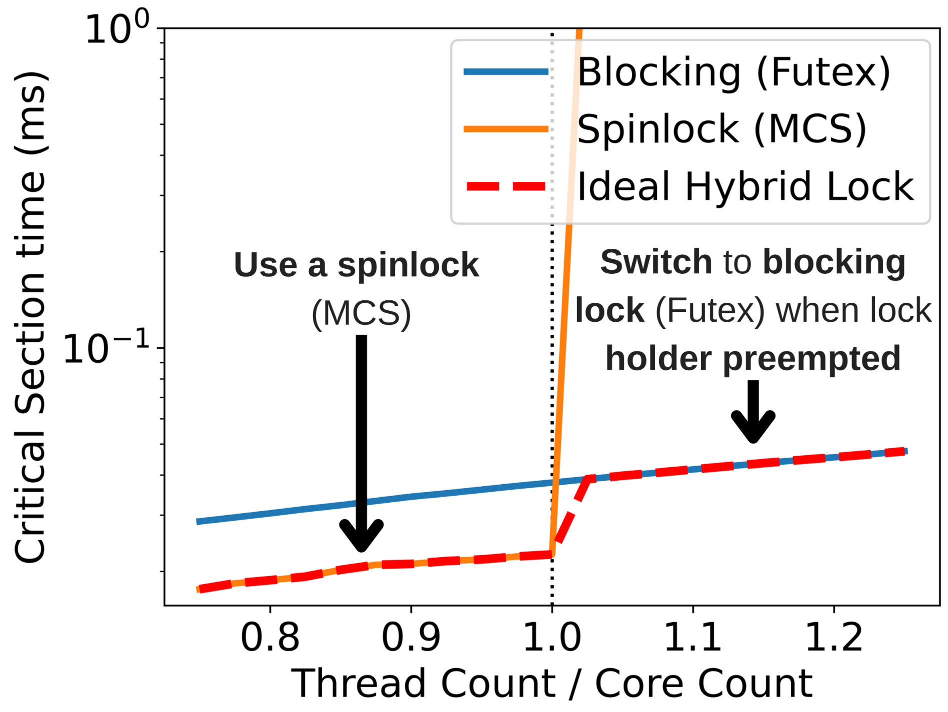


FlexGuard: the best of both worlds!

- Wait… Didn't others try to do this before?!
  - I.e., switch between spinning and blocking?
- **Answer:** yes, but they used unreliable heuristics!

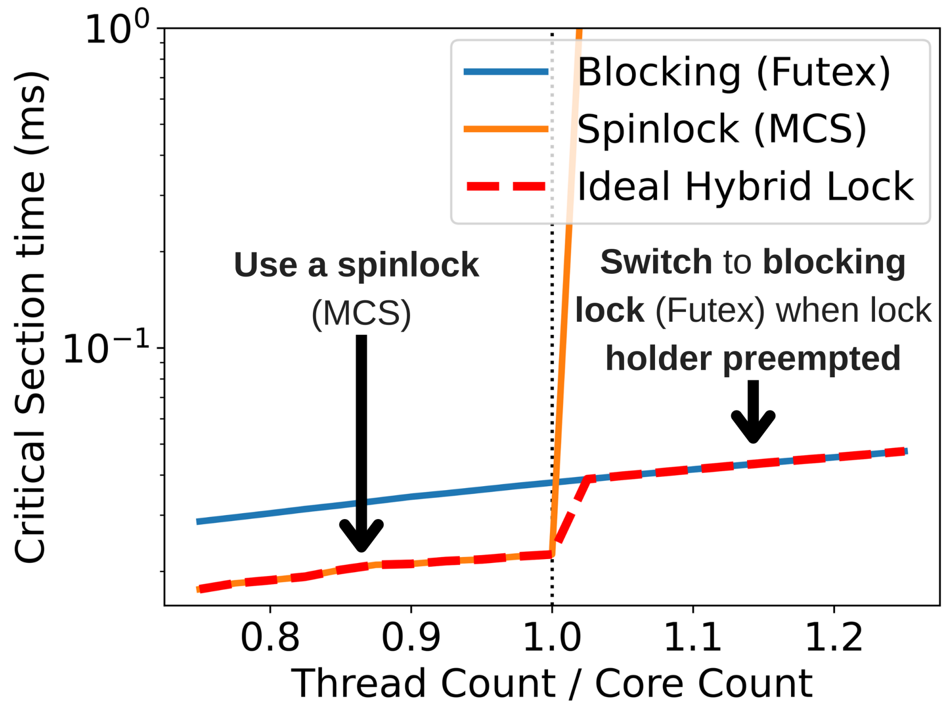


FlexGuard: the best of both worlds!



- Wait… Didn't others try to do this before?!
  - I.e., switch between spinning and blocking?
- **Answer:** yes, but they used unreliable heuristics!
  - **Spin-then-park:** spin a little before blocking
    - Actually POSIX uses this, sometimes worse than just blocking in our experiments
    - **Heuristic:** how long do you spin?

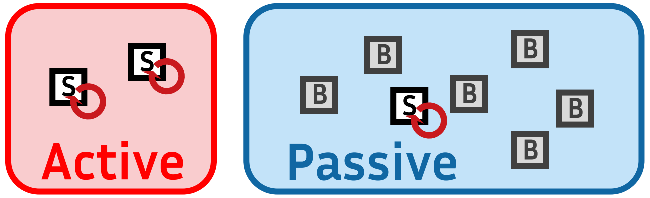
FlexGuard: the best of both worlds!



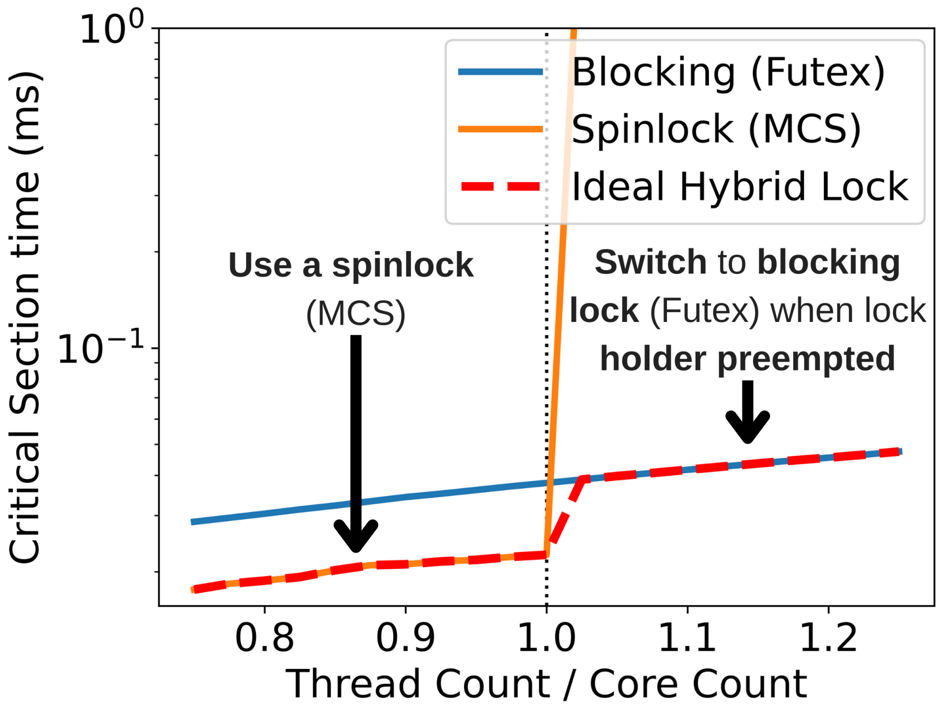
- Wait… Didn't others try to do this before?!
  - I.e., switch between spinning and blocking?
- **Answer:** yes, but they used unreliable heuristics!
  - **Spin-then-park:** spin a little before blocking
    - Actually POSIX uses this, sometimes worse than just blocking in our experiments
    - **Heuristic:** how long do you spin?

[Dice, 2017]

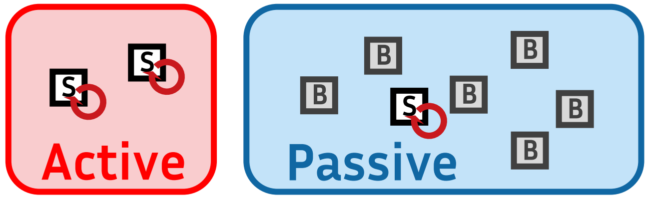
  - **Malthusian locks:** spin-then-park + some threads in a "passive" list
    - Few active threads in the "spin" phase (fairness tradeoff)
    - **Heuristic:** how long do you spin?

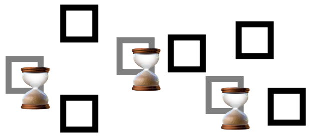


FlexGuard: the best of both worlds!

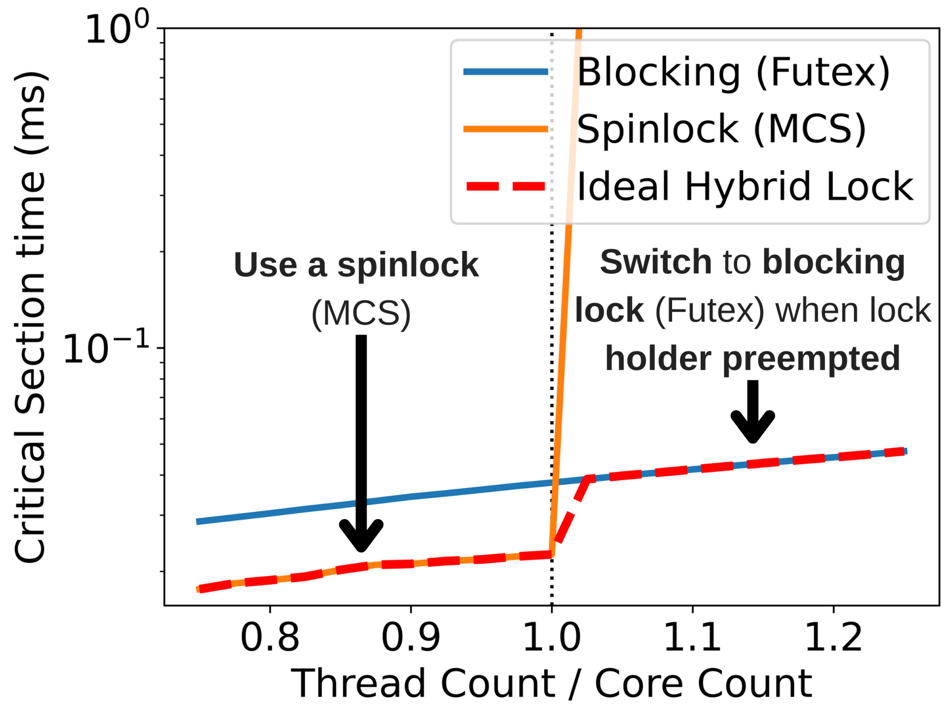


- Wait… Didn't others try to do this before?!
  - I.e., switch between spinning and blocking?
- **Answer:** yes, but they used unreliable heuristics!
  - **Spin-then-park:** spin a little before blocking
    - Actually POSIX uses this, sometimes worse than just blocking in our experiments
    - **Heuristic:** how long do you spin?
  - [Dice, 2017] **Malthusian locks:** spin-then-park + some threads in a "passive" list
    - Few active threads in the "spin" phase (fairness tradeoff)
    - **Heuristic:** how long do you spin?
  - [He et al., 2005] **Time-published locks:** store timestamps, guess preemption if "stale" timestamp
    - **Heuristic:** what timeout do you pick?

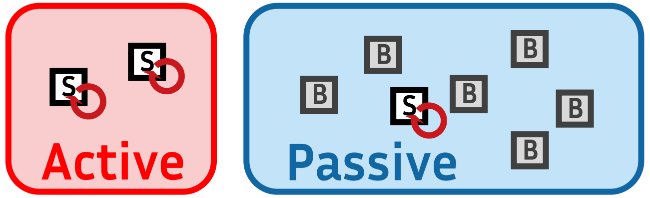




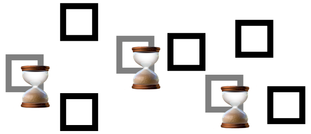
FlexGuard: the best of both worlds!



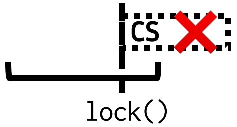
- Wait… Didn't others try to do this before?!
  - I.e., switch between spinning and blocking?
- **Answer:** yes, but they used unreliable heuristics!
  - **Spin-then-park:** spin a little before blocking
    - Actually POSIX uses this, sometimes worse than just blocking in our experiments
    - **Heuristic:** how long do you spin?
  - [Dice, 2017] **Malthusian locks:** spin-then-park + some threads in a "passive" list
    - Few active threads in the "spin" phase (fairness tradeoff)
    - **Heuristic:** how long do you spin?



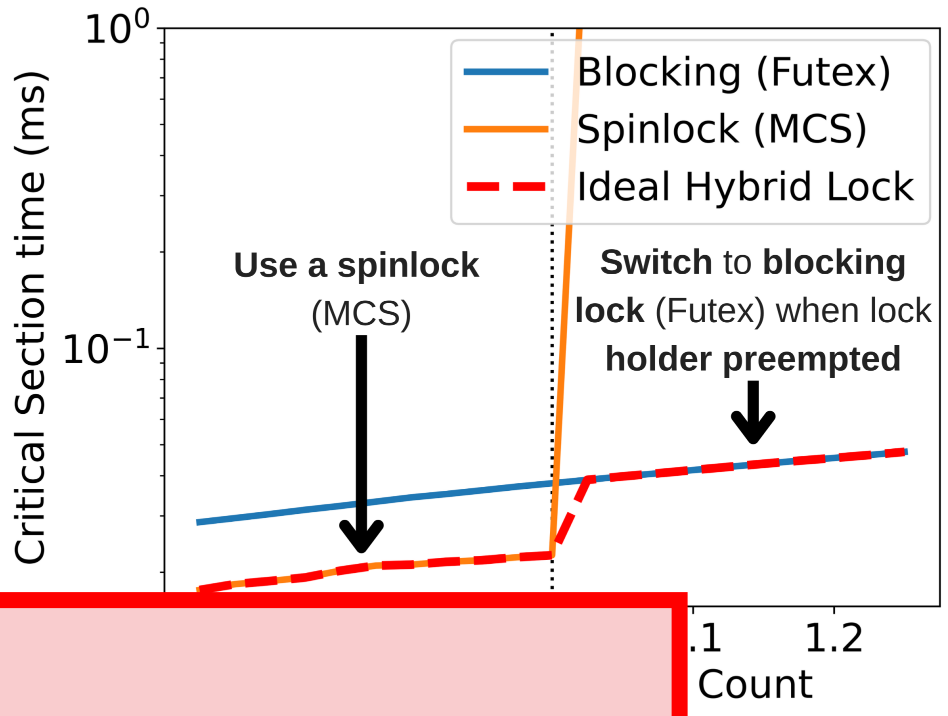
  - [He et al., 2005] **Time-published locks:** store timestamps, guess preemption if "stale" timestamp
    - **Heuristic:** what timeout do you pick?



  - [Teabe et al., 2017] **I-Spinlocks:** only take the lock if enough time left in Xen timeslice (kernel locks in VMs)
    - **Heuristic:** how much is "enough time"?



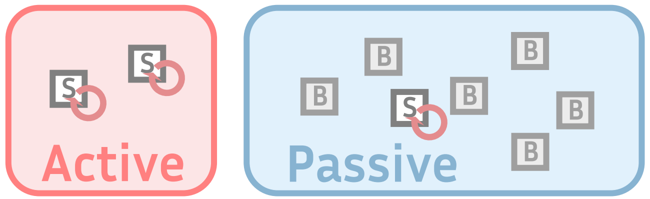
FlexGuard: the best of both worlds!



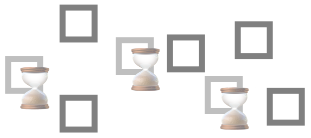
- Wait… Didn't others try to do this before?!
  - I.e., switch between spinning and blocking?

FlexGuard: first completely deterministic approach!
- Switches to blocking ***precisely*** when a critical section preemption happens
- Thanks to eBPF!

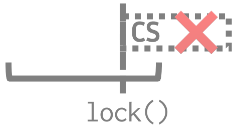
[Dice, 2017]
- **Malthusian locks:** spin-then-park + some threads in a "passive" list
  - Few active threads in the "spin" phase (fairness tradeoff)
  - **Heuristic:** how long do you spin?



[He et al., 2005]
- **Time-published locks:** store timestamps, guess preemption if "stale" timestamp
  - **Heuristic:** what timeout do you pick?



[Teabe et al., 2017]
- **I-Spinlocks:** only take the lock if enough time left in Xen timeslice (kernel locks in VMs)
  - **Heuristic:** how much is "enough time"?



FlexGuard's Preemption Monitor

- FlexGuard's Preemption Monitor detects critical section (CS) preemptions
  - eBPF handler that hooks to the `sched_switch` event
  - **How to detect thread in a critical section?**

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break

def unlock(L):

label at_store
  L = UNLOCKED
```

FlexGuard's Preemption Monitor

- FlexGuard's Preemption Monitor detects critical section (CS) preemptions
  - eBPF handler that hooks to the `sched_switch` event
  - **How to detect thread in a critical section?**
- Example with a simple TATAS spinlock

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break

def unlock(L):

label at_store
  L = UNLOCKED
```

FlexGuard's Preemption Monitor

- FlexGuard's Preemption Monitor detects critical section (CS) preemptions
  - eBPF handler that hooks to the `sched_switch` event
  - **How to detect thread in a critical section?**
- Example with a simple TATAS spinlock
  - **Idea:** use a flag!
    - Set it at the end of `lock()`
    - Unset it at the beginning of `unlock()`
    - If flag set, we're in a critical section!

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break

def unlock(L):

label at_store
  L = UNLOCKED
```

CS

FlexGuard's Preemption Monitor

- FlexGuard's Preemption Monitor detects critical section (CS) preemptions
  - eBPF handler that hooks to the `sched_switch` event
  - **How to detect thread in a critical section?**
- Example with a simple TATAS spinlock
  - **Idea:** use a flag!
    - Set it at the end of `lock()`
    - Unset it at the beginning of `unlock()`
    - If flag set, we're in a critical section!
  - **Actually, need to use a counter** for nested CSs
    - If `cs_counter > 0`, we're in a critical section

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break
  cs_counter += 1
def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

FlexGuard's Preemption Monitor

- FlexGuard's Preemption Monitor detects critical section (CS) preemptions
  - eBPF handler that hooks to the `sched_switch` event
  - **How to detect thread in a critical section?**

eBPF

- Example with a simple TATAS spinlock
  - **Idea:** use a flag!
    - Set it at the end of `lock()`
    - Unset it at the beginning of `unlock()`
    - If flag set, we're in a critical section!
  - **Actually, need to use a counter** for nested CSs
    - If `cs_counter > 0`, we're in a critical section
  - **Is that enough to be accurate?**

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break
  cs_counter += 1   <-
def unlock(L):
  cs_counter -= 1   <-
label at_store
  L = UNLOCKED
```

CS

FlexGuard's Preemption Monitor

- **Answer:** no, the counter is not enough.

CS

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break
  cs_counter += 1
def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

FlexGuard's Preemption Monitor

- **Answer:** no, the counter is not enough.
- **`lock()` function:** when are we in the critical section?

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break
  cs_counter += 1
def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS

FlexGuard's Preemption Monitor

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break
  cs_counter += 1
def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS

- **Answer:** no, the counter is not enough.
- **`lock()` function:** when are we in the critical section?
  - Right after `XCHG` succeeded in changing the lock value, already in the CS!
    - *There could be instructions until the actual* `cs_counter` *increment!*

FlexGuard's Preemption Monitor

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break
  cs_counter += 1
def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS

CS

- **Answer:** no, the counter is not enough.
- **`lock()` function:** when are we in the critical section?
  - Right after `XCHG` succeeded in changing the lock value, already in the CS!
    - *There could be instructions until the actual* `cs_counter` *increment!*
  - I.e., if we've been preempted between `at_break` and the end of the lock function

FlexGuard's Preemption Monitor

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break
  cs_counter += 1

def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS

CS

- **Answer:** no, the counter is not enough.

- **`lock()` function:** when are we in the critical section?
  - Right after `XCHG` succeeded in changing the lock value, already in the CS!
    - *There could be instructions until the actual* `cs_counter` *increment!*
  - I.e., if we've been preempted between `at_break` and the end of the lock function

- **`unlock()` function:** when are we in the critical section?

FlexGuard's Preemption Monitor

```
5  def lock(L):
6    while (True):
7      if L == UNLOCKED:
8  label at_xchg
9        if XCHG(&L, LOCKED) == UNLOCKED:
10 label at_break
11         break
12   cs_counter += 1
13 def unlock(L):
14   cs_counter -= 1
15 label at_store
16   L = UNLOCKED
```

CS

CS

- **Answer:** no, the counter is not enough.
- **`lock()` function:** when are we in the critical section?
  - Right after `XCHG` succeeded in changing the lock value, already in the CS!
    - *There could be instructions until the actual* `cs_counter` *increment!*
  - I.e., if we've been preempted between `at_break` and the end of the lock function
- **`unlock()` function:** when are we in the critical section?
  - Until the store at line 16 actually completed, still in the CS!
    - *There could be instructions between the cs_counter decrement and that!*

FlexGuard's Preemption Monitor

```
5  def lock(L):
6    while (True):
7      if L == UNLOCKED:
8  label at_xchg
9        if XCHG(&L, LOCKED) == UNLOCKED:
10 label at_break
11         break
12   cs_counter += 1
13 def unlock(L):
14   cs_counter -= 1
15 label at_store
16   L = UNLOCKED
```

CS

CS

CS

- **Answer:** no, the counter is not enough.
- **`lock()` function:** when are we in the critical section?
  - Right after `XCHG` succeeded in changing the lock value, already in the CS!
    - *There could be instructions until the actual* `cs_counter` *increment!*
  - I.e., if we've been preempted between `at_break` and the end of the lock function
- **`unlock()` function:** when are we in the critical section?
  - Until the store at line 16 actually completed, still in the CS!
    - *There could be instructions between the cs_counter decrement and that!*
  - I.e., if we've been preempted between the beginning of the unlock function and `at_store`
    - Assuming `at_store` is the final `MOV` that changes the lock variable's value

FlexGuard's Preemption Monitor

```
5  def lock(L):
6    while (True):
7      if L == UNLOCKED:
8  label at_xchg
9        if XCHG(&L, LOCKED) == UNLOCKED:
10 label at_break
11         break
12   cs_counter += 1
13 def unlock(L):
14   cs_counter -= 1
15 label at_store
16   L = UNLOCKED
```

CS

CS

CS

- **Answer:** no, the counter is not enough.
- **`lock()` function:** when are we in the critical section?
  - Right after `XCHG` succeeded in changing the lock value, already in the CS!
    - *There could be instructions until the actual* `cs_counter` *increment!*
  - I.e., if we've been preempted between `at_break` and the end of the lock function
- **`unlock()` function:** when are we in the critical section?
  - Until the store at line 16 actually completed, still in the CS!
    - *There could be instructions between the `cs_counter` decrement and that!*
  - I.e., if we've been preempted between the beginning of the unlock function and `at_store`
    - Assuming `at_store` is the final `MOV` that changes the lock variable's value
- Can we take care of these cases?

FlexGuard's Preemption Monitor

```
5  def lock(L):
6    while (True):
7      if L == UNLOCKED:
8  label at_xchg
9        if XCHG(&L, LOCKED) == UNLOCKED:
10 label at_break
11         break
12   cs_counter += 1
13 def unlock(L):
14   cs_counter -= 1
15 label at_store
16   L = UNLOCKED
```

CS

CS

CS

- **Answer:** no, the counter is not enough.
- **`lock()` function:** when are we in the critical section?
  - Right after `XCHG` succeeded in changing the lock value, already in the CS!
    - *There could be instructions until the actual* `cs_counter` *increment!*
  - I.e., if we've been preempted between `at_break` and the end of the lock function
- **`unlock()` function:** when are we in the critical section?
  - Until the store at line 16 actually completed, still in the CS!
    - *There could be instructions between the `cs_counter` decrement and that!*
  - I.e., if we've been preempted between the beginning of the unlock function and `at_store`
    - Assuming `at_store` is the final `MOV` that changes the lock variable's value
- Can we take care of these cases?
  - **Yes, since the eBPF handler has access to the preemption address!**

FlexGuard's Preemption Monitor

- Is it finally accurate?

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break
  cs_counter += 1

def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS

CS

CS

FlexGuard's Preemption Monitor

- Is it finally accurate?
- **No, still one problematic case:**
  - What if preeemption right after the XCHG?
  - Then we are in a critical section iff the return value is UNLOCKED!

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break
  cs_counter += 1
def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS?
CS
CS
CS

FlexGuard's Preemption Monitor

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break
  cs_counter += 1

def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS? → if XCHG(&L, LOCKED) == UNLOCKED:

CS (label at_break … cs_counter += 1)

CS (cs_counter += 1 … cs_counter -= 1)

CS (def unlock(L) … label at_store)

- Is it finally accurate?
- No, still one problematic case:
  - What if preeemption right after the XCHG?
  - Then we are in a critical section iff the return value is UNLOCKED!
- Can we take care of this case?
  - Yes, we can force the return value of XCHG to be in a specific register (w/ `asm volatile`)
  - In the eBPF handler, we can access dumped register value (through the `task_struct`)!

  ⇒ Preemptions detected with 100% accuracy!

FlexGuard's Preemption Monitor

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break
  cs_counter += 1

def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS?

CS

CS

CS

- **Is it finally accurate?**

- **No, still one problematic case:**
  - What if preeemption right after the XCHG?
  - Then we are in a critical section iff the return value is UNLOCKED!

- **Can we take care of this case?**
  - Yes, we can force the return value of XCHG to be in a specific register (w/ `asm volatile`)
  - In the eBPF handler, we can access dumped register value (through the `task_struct`)!

  ⇒ Preemptions detected with 100% accuracy!

- **Is it important to be fully accurate?**
  - Yes application critical sections only a few lines long, preemptions likely in `lock()`/`unlock()`
  - Sufficient to cause performance collapse!

FlexGuard's Preemption Monitor

```
__thread cs_counter = 0 # Per-thread critical section (CS)
                        # counter (# of CSs a thread is in)
__thread bool is_cs_preempted = False # Thread in CS?
num_preempted_cs = 0 # System-wide preemption counter
```

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
CS? → if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break
  cs_counter += 1

def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS

CS

CS

```
def sched_switch_btf(prev, next):
  # If next was previously preempted
  if next.is_cs_preempted:
    next.is_cs_preempted = False
    atomic_dec(num_preempted_cs)

  prev_in_cs = False # Will be set to true if prev in CS code
  if prev.cs_counter > 0: # values > 1 indicate nesting
    prev_in_cs = True # prev holding at least one lock; in CS
  else # prev.cs_counter == 0
    # Addr. of next instruction to execute after preemption
    preemption_addr = bpf_get_task_stack(prev)[0]
    if at_xchg < preemption_addr <= at_break:
      registers = bpf_get_task_registers(prev)
      if registers.rcx == UNLOCKED:
        prev_in_cs = True # lock acquired; already in CS code
    elif at_break < preemption_addr <= lock$end or
           unlock <= preemption_addr <= at_store:
      prev_in_cs = True # prev in already/still in CS code

  if prev_in_cs:
    prev.is_cs_preempted = True
    atomic_inc(num_preempted_cs)
```

FlexGuard's Preemption Monitor

```
__thread cs_counter = 0 # Per-thread critical section (CS)
                        # counter (# of CSs a thread is in)
__thread bool is_cs_preempted = False # Thread in CS?
num_preempted_cs = 0 # System-wide preemption counter
```

For communication w/ the lock

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
CS?──► if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
          break
  cs_counter += 1

def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS

CS

CS

```
def sched_switch_btf(prev, next):
  # If next was previously preempted
  if next.is_cs_preempted:
    next.is_cs_preempted = False
    atomic_dec(num_preempted_cs)

  prev_in_cs = False # Will be set to true if prev in CS code
  if prev.cs_counter > 0: # values > 1 indicate nesting
    prev_in_cs = True # prev holding at least one lock; in CS
  else # prev.cs_counter == 0
    # Addr. of next instruction to execute after preemption
    preemption_addr = bpf_get_task_stack(prev)[0]
    if at_xchg < preemption_addr <= at_break:
      registers = bpf_get_task_registers(prev)
      if registers.rcx == UNLOCKED:
        prev_in_cs = True # lock acquired; already in CS code
    elif at_break < preemption_addr <= lock$end or
         unlock <= preemption_addr <= at_store:
      prev_in_cs = True # prev in already/still in CS code

  if prev_in_cs:
    prev.is_cs_preempted = True
    atomic_inc(num_preempted_cs)
```

FlexGuard's Preemption Monitor

```
__thread cs_counter = 0 # Per-thread critical section (CS)
                        # counter (# of CSs a thread is in)
__thread bool is_cs_preempted = False # Thread in CS?
num_preempted_cs = 0 # System-wide preemption counter
```

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break
  cs_counter += 1

def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS? → `if XCHG(&L, LOCKED) == UNLOCKED:`

CS

CS

CS

```
def sched_switch_btf(prev, next):
  # If next was previously preempted
  if next.is_cs_preempted:
    next.is_cs_preempted = False
    atomic_dec(num_preempted_cs)

  prev_in_cs = False # Will be set to true if prev in CS code
  if prev.cs_counter > 0: # values > 1 indicate nesting
    prev_in_cs = True # prev holding at least one lock; in CS
  else # prev.cs_counter == 0
    # Addr. of next instruction to execute after preemption
    preemption_addr = bpf_get_task_stack(prev)[0]
    if at_xchg < preemption_addr <= at_break:
      registers = bpf_get_task_registers(prev)
      if registers.rcx == UNLOCKED:
        prev_in_cs = True # lock acquired; already in CS code
    elif at_break < preemption_addr <= lock$end or
         unlock <= preemption_addr <= at_store:
      prev_in_cs = True # prev in already/still in CS code

  if prev_in_cs:
    prev.is_cs_preempted = True
    atomic_inc(num_preempted_cs)
```

next rescheduled in CS

Inria

FlexGuard's Preemption Monitor

```
__thread cs_counter = 0 # Per-thread critical section (CS)
                        # counter (# of CSs a thread is in)
__thread bool is_cs_preempted = False # Thread in CS?
num_preempted_cs = 0 # System-wide preemption counter
```

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
        break
  cs_counter += 1

def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS? → if XCHG(&L, LOCKED) == UNLOCKED:

CS

CS

CS

```
def sched_switch_btf(prev, next):
  # If next was previously preempted
  if next.is_cs_preempted:
    next.is_cs_preempted = False
    atomic_dec(num_preempted_cs)

  prev_in_cs = False # Will be set to true if prev in CS code
  if prev.cs_counter > 0: # values > 1 indicate nesting
    prev_in_cs = True # prev holding at least one lock; in CS
  else # prev.cs_counter == 0
    # Addr. of next instruction to execute after preemption
    preemption_addr = bpf_get_task_stack(prev)[0]
    if at_xchg < preemption_addr <= at_break:
      registers = bpf_get_task_registers(prev)
      if registers.rcx == UNLOCKED:
        prev_in_cs = True # lock acquired; already in CS code
    elif at_break < preemption_addr <= lock$end or
         unlock <= preemption_addr <= at_store:
      prev_in_cs = True # prev in already/still in CS code

  if prev_in_cs:
    prev.is_cs_preempted = True
    atomic_inc(num_preempted_cs)
```

FlexGuard's Preemption Monitor

```
__thread cs_counter = 0 # Per-thread critical section (CS)
                        # counter (# of CSs a thread is in)
__thread bool is_cs_preempted = False # Thread in CS?
num_preempted_cs = 0 # System-wide preemption counter
```

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
CS? → if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
          break
  cs_counter += 1

def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS CS CS

```
def sched_switch_btf(prev, next):
  # If next was previously preempted
  if next.is_cs_preempted:
    next.is_cs_preempted = False
    atomic_dec(num_preempted_cs)

  prev_in_cs = False # Will be set to true if prev in CS code
  if prev.cs_counter > 0: # values > 1 indicate nesting
    prev_in_cs = True # prev holding at least one lock; in CS
  else # prev.cs_counter == 0
    # Addr. of next instruction to execute after preemption
    preemption_addr = bpf_get_task_stack(prev)[0]
    if at_xchg < preemption_addr <= at_break:
      registers = bpf_get_task_registers(prev)
      if registers.rcx == UNLOCKED:
        prev_in_cs = True # lock acquired; already in CS code
    elif at_break < preemption_addr <= lock$end or
         unlock <= preemption_addr <= at_store:
      prev_in_cs = True # prev in already/still in CS code

  if prev_in_cs:
    prev.is_cs_preempted = True
    atomic_inc(num_preempted_cs)
```

FlexGuard's Preemption Monitor

```
__thread cs_counter = 0 # Per-thread critical section (CS)
                        # counter (# of CSs a thread is in)
__thread bool is_cs_preempted = False # Thread in CS?
num_preempted_cs = 0 # System-wide preemption counter
```

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
CS?→ if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
         break
  cs_counter += 1

def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS

CS

CS

```
def sched_switch_btf(prev, next):
  # If next was previously preempted
  if next.is_cs_preempted:
    next.is_cs_preempted = False
    atomic_dec(num_preempted_cs)

  prev_in_cs = False # Will be set to true if prev in CS code
  if prev.cs_counter > 0: # values > 1 indicate nesting
    prev_in_cs = True # prev holding at least one lock; in CS
  else # prev.cs_counter == 0
    # Addr. of next instruction to execute after preemption
    preemption_addr = bpf_get_task_stack(prev)[0]
    if at_xchg < preemption_addr <= at_break:
      registers = bpf_get_task_registers(prev)
      if registers.rcx == UNLOCKED:
        prev_in_cs = True # lock acquired; already in CS code
    elif at_break < preemption_addr <= lock$end or
         unlock <= preemption_addr <= at_store:
      prev_in_cs = True # prev in already/still in CS code

  if prev_in_cs:
    prev.is_cs_preempted = True
    atomic_inc(num_preempted_cs)
```

FlexGuard's Preemption Monitor

```
__thread cs_counter = 0 # Per-thread critical section (CS)
                        # counter (# of CSs a thread is in)
__thread bool is_cs_preempted = False # Thread in CS?
num_preempted_cs = 0 # System-wide preemption counter
```

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
      if XCHG(&L, LOCKED) == UNLOCKED:   <- CS?
label at_break
        break
  cs_counter += 1

def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS (brackets: from at_break to cs_counter += 1; from cs_counter += 1 to cs_counter -= 1; from cs_counter -= 1 to L = UNLOCKED)

```
def sched_switch_btf(prev, next):
  # If next was previously preempted
  if next.is_cs_preempted:
    next.is_cs_preempted = False
    atomic_dec(num_preempted_cs)

  prev_in_cs = False # Will be set to true if prev in CS code
  if prev.cs_counter > 0: # values > 1 indicate nesting
    prev_in_cs = True # prev holding at least one lock; in CS
  else # prev.cs_counter == 0
    # Addr. of next instruction to execute after preemption
    preemption_addr = bpf_get_task_stack(prev)[0]
    if at_xchg < preemption_addr <= at_break:
      registers = bpf_get_task_registers(prev)
      if registers.rcx == UNLOCKED:
        prev_in_cs = True # lock acquired; already in CS code
    elif at_break < preemption_addr <= lock$end or
        unlock <= preemption_addr <= at_store:
      prev_in_cs = True # prev in already/still in CS code

  if prev_in_cs:
    prev.is_cs_preempted = True
    atomic_inc(num_preempted_cs)
```

FlexGuard's Preemption Monitor

```
__thread cs_counter = 0 # Per-thread critical section (CS)
                        # counter (# of CSs a thread is in)
__thread bool is_cs_preempted = False # Thread in CS?
num_preempted_cs = 0 # System-wide preemption counter
```

```
def lock(L):
  while (True):
    if L == UNLOCKED:
label at_xchg
CS?──►  if XCHG(&L, LOCKED) == UNLOCKED:
label at_break
          break
  cs_counter += 1

def unlock(L):
  cs_counter -= 1
label at_store
  L = UNLOCKED
```

CS

CS

CS

```
def sched_switch_btf(prev, next):
  # If next was previously preempted
  if next.is_cs_preempted:
    next.is_cs_preempted = False
    atomic_dec(num_preempted_cs)

  prev_in_cs = False # Will be set to true if prev in CS code
  if prev.cs_counter > 0: # values > 1 indicate nesting
    prev_in_cs = True # prev holding at least one lock; in CS
  else # prev.cs_counter == 0
    # Addr. of next instruction to execute after preemption
    preemption_addr = bpf_get_task_stack(prev)[0]
    if at_xchg < preemption_addr <= at_break:
      registers = bpf_get_task_registers(prev)
      if registers.rcx == UNLOCKED:
        prev_in_cs = True # lock acquired; already in CS code
    elif at_break < preemption_addr <= lock$end or
         unlock <= preemption_addr <= at_store:
      prev_in_cs = True # prev in already/still in CS code

  if prev_in_cs:
    prev.is_cs_preempted = True
    atomic_inc(num_preempted_cs)
```

prev preempted in CS

FlexGuard's lock algorithm

- We now have a reliable way to detect critical section preemptions

FlexGuard's lock algorithm

- We now have a reliable way to detect critical section preemptions
- We need an efficient hybrid spin/blocking lock algorithm to go with it

FlexGuard's lock algorithm

- We now have a reliable way to detect critical section preemptions
- We need an efficient hybrid spin/blocking lock algorithm to go with it
- **For this, we need a bit of background on efficient lock algorithms**

FlexGuard's lock algorithm

- We now have a reliable way to detect critical section preemptions
- We need an efficient hybrid spin/blocking lock algorithm to go with it
- **For this, we need a bit of background on efficient lock algorithms**
- **Focus:** efficient spinlock algorithms
  - Blocking locks simply call the FUTEX syscall, can't be improved
  - *Unless you spin…*

Previous work on spinlocks

- **Basic spinlock**:

```
lock() {
    while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
        PAUSE; // Spinloop hint
}

unlock() { lock = UNLOCKED; }
```

Previous work on spinlocks

- **Basic spinlock**:

```
lock() {
    while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
        PAUSE; // Spinloop hint
}

unlock() { lock = UNLOCKED; }
```

- In theory, transitions between critical sections fast: one cache miss!
  - `lock = UNLOCKED` invalidates `lock`'s cache line

Previous work on spinlocks

- **Basic spinlock**:

```
lock() {
    while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
        PAUSE; // Spinloop hint
}

unlock() { lock = UNLOCKED; }
```

- In theory, transitions between critical sections fast: one cache miss!
  - `lock = UNLOCKED` invalidates `lock`'s cache line
  - Another thread fetches it and instantly executes a successful CAS
    - *Much faster than waking up a thread*

Previous work on spinlocks

- **Basic spinlock**:

```
lock() {
    while (compare_and_swap(&lock, UNLOCKED, LOCKED) != UNLOCKED)
        PAUSE; // Spinloop hint
}

unlock() { lock = UNLOCKED; }
```

- In theory, transitions between critical sections fast: one cache miss!
  - `lock = UNLOCKED` invalidates `lock`'s cache line
  - Another thread fetches it and instantly executes a successful CAS
    - *Much faster than waking up a thread*
- In practice, spinlocks can be very fast, but you need smarter algorithms than that…
  - Lots of write contention on the `lock` variable!

Previous work on spinlocks

- **Optimisation 1: spin in read mode on the lock variable**
  - **Test-and-Test-and-Set (TATAS) lock**: test the lock value without an atomic instruction first

```
while (lock == UNLOCKED && XCHG(&lock, LOCKED) != UNLOCKED)
```

Previous work on spinlocks

- **Optimisation 1: spin in read mode on the lock variable**
  - **Test-and-Test-and-Set (TATAS) lock**: test the lock value without an atomic instruction first

```
while (lock == UNLOCKED && XCHG(&lock, LOCKED) != UNLOCKED)
```

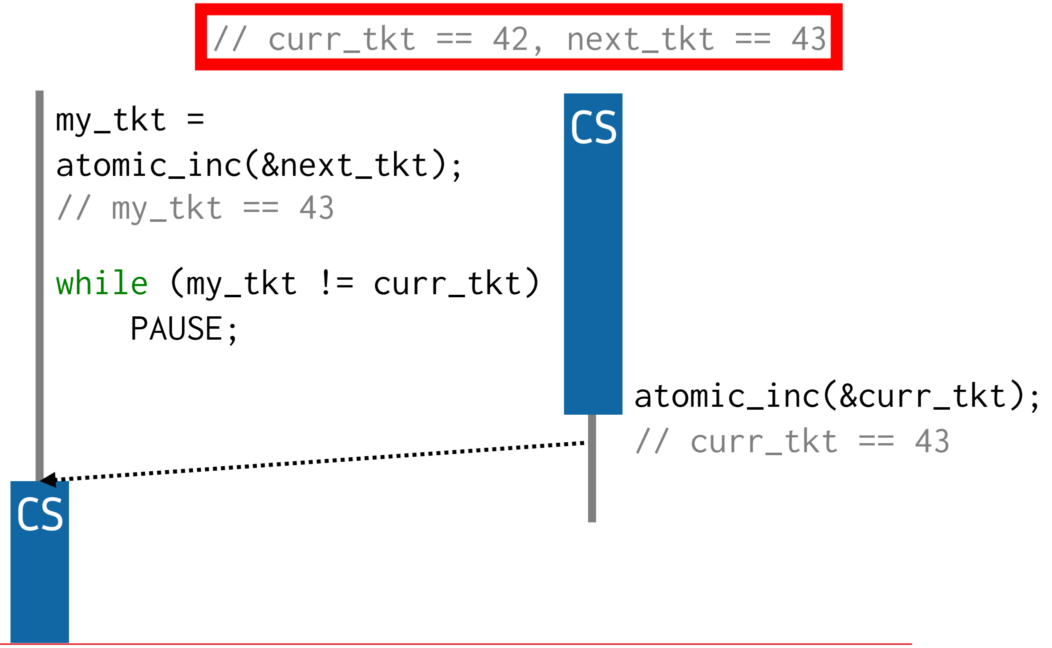
  - Not 100% in read mode, nothing ensures `lock` is still `UNLOCKED` when you do the `XCHG`...

Previous work on spinlocks

- **Optimisation 1: spin in read mode on the lock variable**
  - **Test-and-Test-and-Set (TATAS) lock**: test the lock value without an atomic instruction first

```
while (lock == UNLOCKED && XCHG(&lock, LOCKED) != UNLOCKED)
```

  - Not 100% in read mode, nothing ensures `lock` is still `UNLOCKED` when you do the `XCHG`...
- **Ticket lock**: current ticket defines who's in CS
  - Like at the post office ✉️ *(in some countries 🇨🇭)*

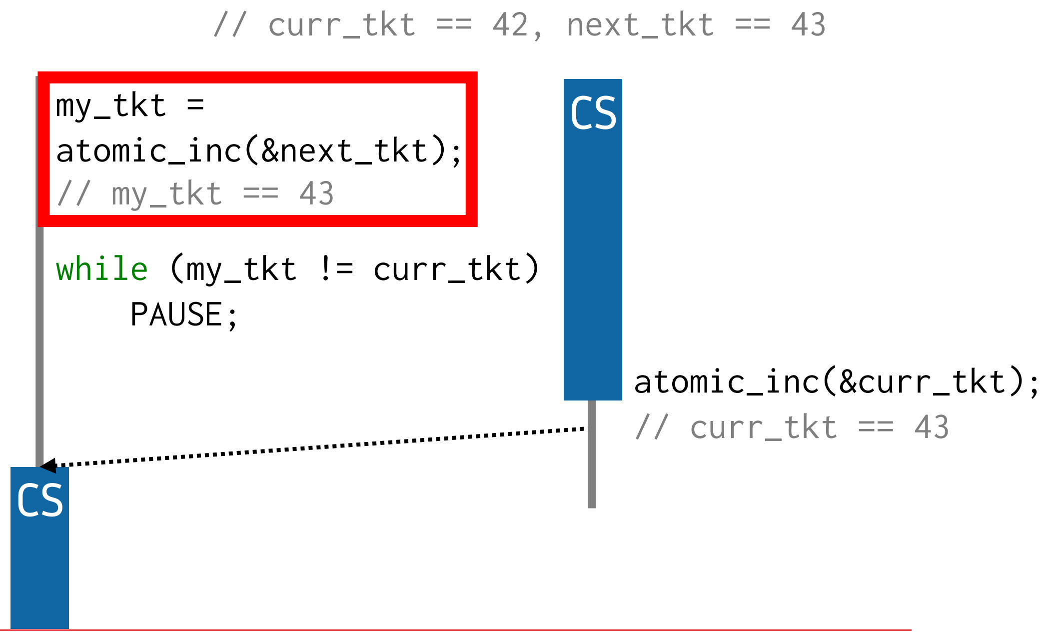


Previous work on spinlocks

- **Optimisation 1: spin in read mode on the lock variable**
  - **Test-and-Test-and-Set (TATAS) lock**: test the lock value without an atomic instruction first

```
while (lock == UNLOCKED && XCHG(&lock, LOCKED) != UNLOCKED)
```

    - Not 100% in read mode, nothing ensures `lock` is still `UNLOCKED` when you do the `XCHG`...
  - **Ticket lock**: current ticket defines who's in CS
    - Like at the post office ✉️ *(in some countries* 🇨🇭 *)*
    - **Before acquiring the lock:** get your ticket
      - Atomic but not on the critical path

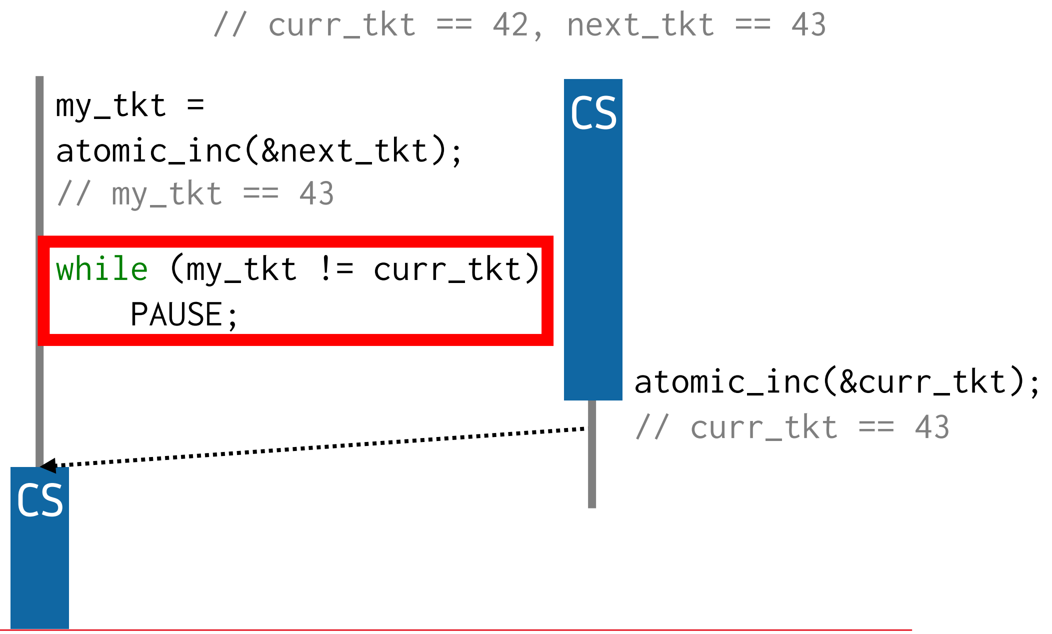


Previous work on spinlocks

- **Optimisation 1: spin in read mode on the lock variable**
  - **Test-and-Test-and-Set (TATAS) lock**: test the lock value without an atomic instruction first

```
while (lock == UNLOCKED && XCHG(&lock, LOCKED) != UNLOCKED)
```

  - Not 100% in read mode, nothing ensures `lock` is still `UNLOCKED` when you do the `XCHG`…
  - **Ticket lock**: current ticket defines who's in CS
    - Like at the post office ✉️ *(in some countries 🇨🇭)*
    - **Before acquiring the lock:** get your ticket
      - Atomic but not on the critical path
    - **Lock acquisition:**
      - Spin until the current ticket == your ticket value
      - 100% in read mode!

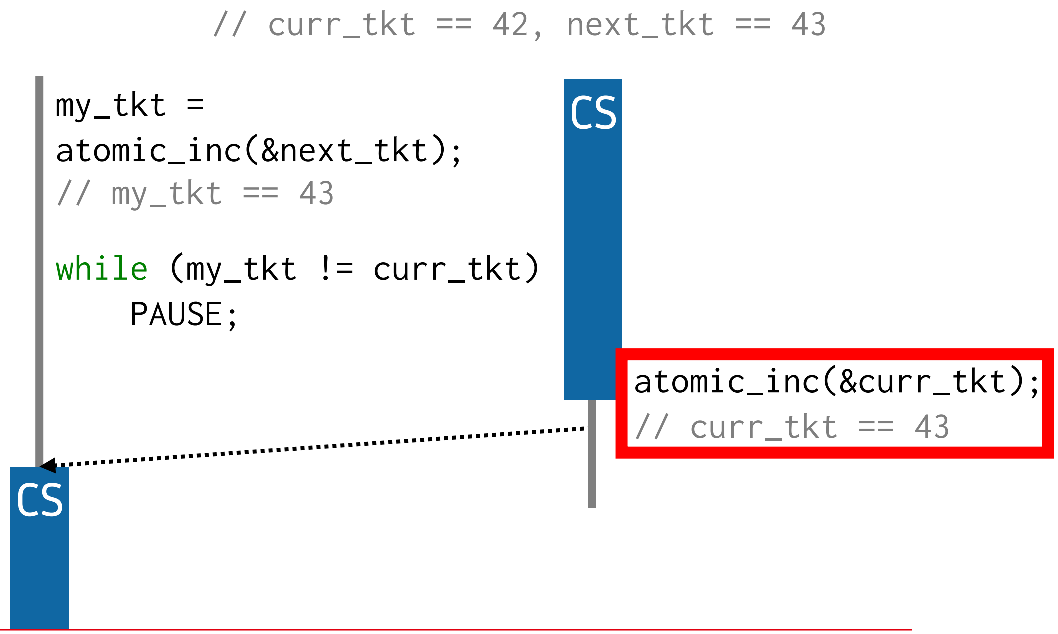


Previous work on spinlocks

- **Optimisation 1: spin in read mode on the lock variable**
  - **Test-and-Test-and-Set (TATAS) lock**: test the lock value without an atomic instruction first

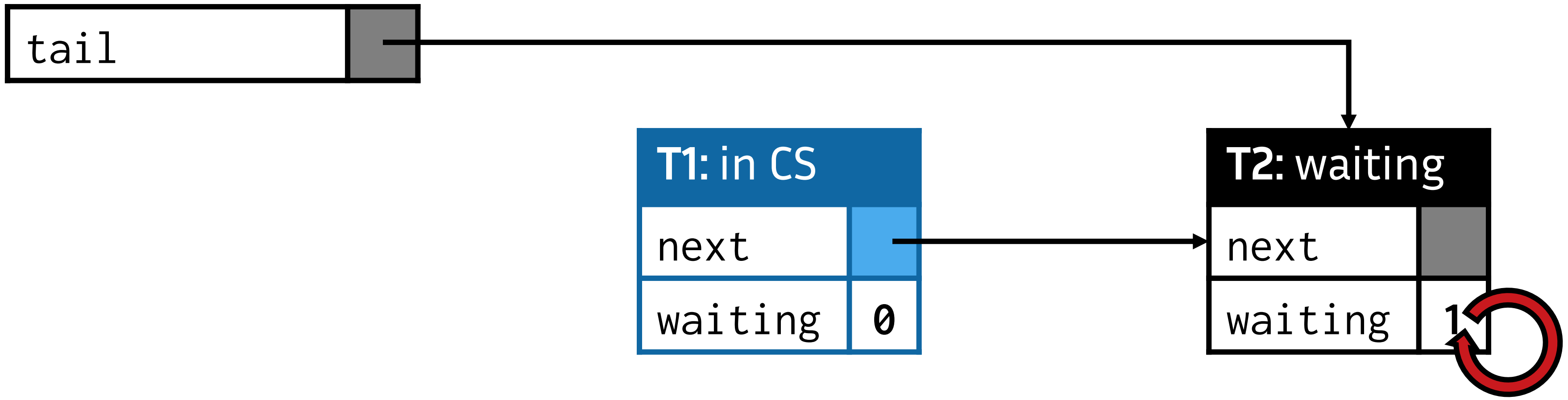
```
while (lock == UNLOCKED && XCHG(&lock, LOCKED) != UNLOCKED)
```

  - Not 100% in read mode, nothing ensures `lock` is still `UNLOCKED` when you do the `XCHG`…
  - **Ticket lock**: current ticket defines who's in CS
    - Like at the post office ✉️ *(in some countries 🇨🇭)*
    - **Before acquiring the lock:** get your ticket
      - Atomic but not on the critical path
    - **Lock acquisition:**
      - Spin until the current ticket == your ticket value
      - 100% in read mode!
    - **On CS exit:** atomically increment the current ticket



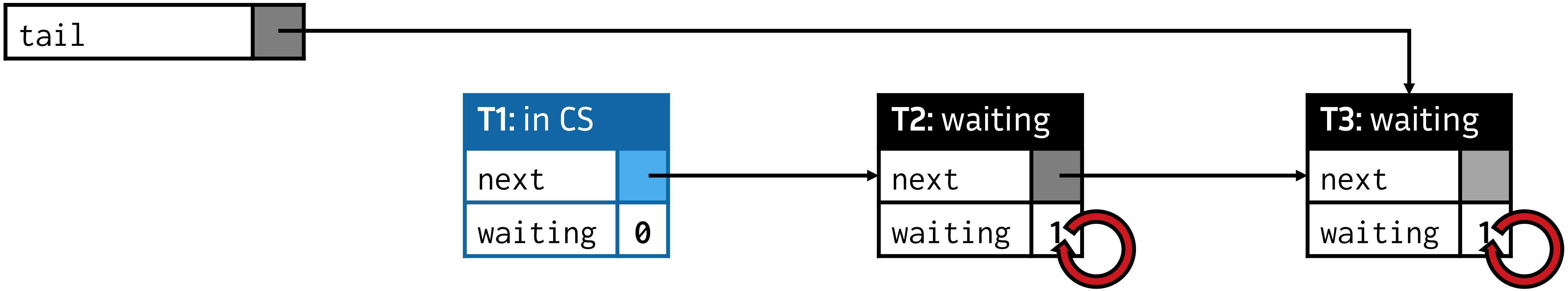
Previous work on spinlocks

- **Optimisation 2:** use multiple lock variables
  - **Queue locks (MCS, CLH)**: [Mellor-Crummey et all., 1991] [Craig et al. 1993; Magnussen et al. 1994]
    - One queue node/lock variable per thread



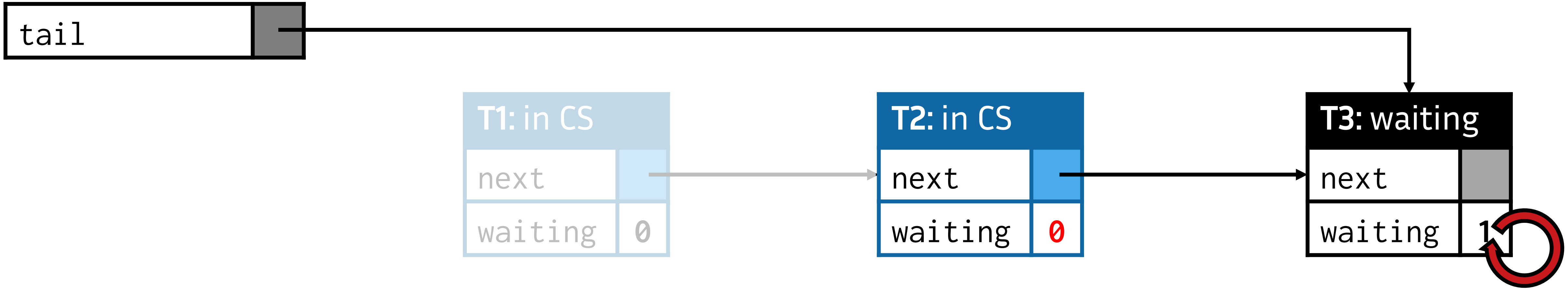
Previous work on spinlocks

- **Optimisation 2:** use multiple lock variables
  - **Queue locks (MCS, CLH)**: [Mellor-Crummey et all., 1991] [Craig et al. 1993; Magnussen et al. 1994]
    - One queue node/lock variable per thread
    - **Lock acquisition:** enqueue the thread's node (atomic, outside the critical path)



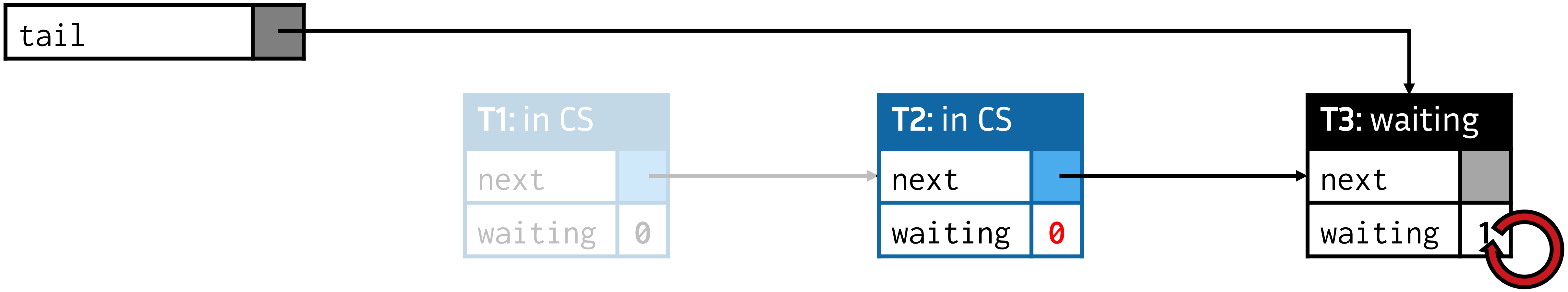
Previous work on spinlocks

- **Optimisation 2:** use multiple lock variables
  - **Queue locks (MCS, CLH)**: [Mellor-Crummey et all., 1991] [Craig et al. 1993; Magnussen et al. 1994]
    - One queue node/lock variable per thread
    - **Lock acquisition:** enqueue the thread's node (atomic, outside the critical path)
    - **On critical section exit:** write local lock variable to signal the next thread we're done



Previous work on spinlocks

- **Optimisation 2:** use multiple lock variables
  - **Queue locks (MCS, CLH)**: [Mellor-Crummey et all., 1991] [Craig et al. 1993; Magnussen et al. 1994]
    - One queue node/lock variable per thread
    - **Lock acquisition:** enqueue the thread's node (atomic, outside the critical path)
    - **On critical section exit:** write local lock variable to signal the next thread we're done
    - **Difference between MCS and CLH:** direction of the queue



Previous work on spinlocks

- **Optimisation 3:** NUMA-awareness
  - Modern machines often have Non-Uniform Memory Architectures (NUMA)
    - *E.g., one NUMA node = one processor*

Previous work on spinlocks

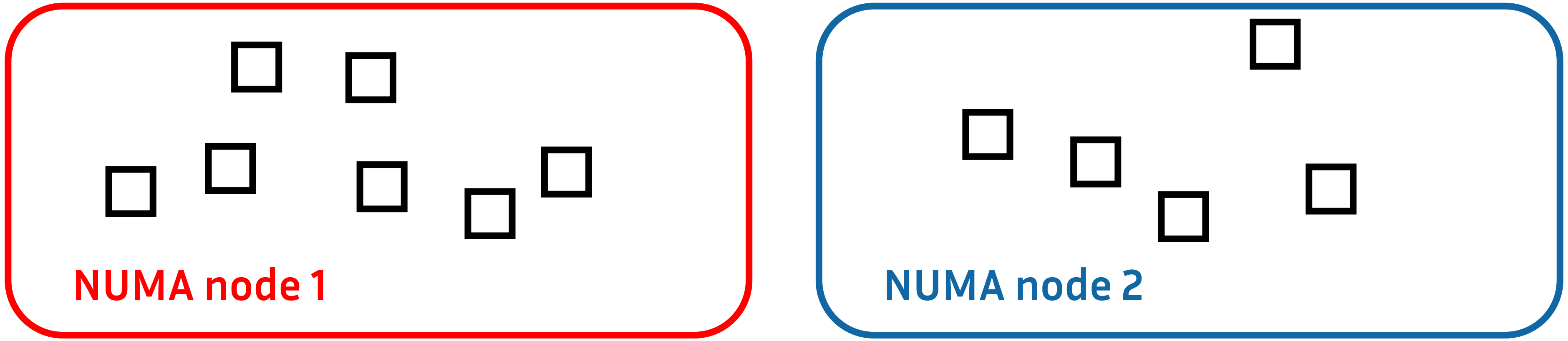
- **Optimisation 3:** NUMA-awareness
  - Modern machines often have Non-Uniform Memory Architectures (NUMA)
    - *E.g., one NUMA node = one processor*
  - Faster to hand over the lock on the same NUMA node than to a remote NUMA node

Previous work on spinlocks

- **Optimisation 3:** NUMA-awareness
  - Modern machines often have Non-Uniform Memory Architectures (NUMA)
    - *E.g., one NUMA node = one processor*
  - Faster to hand over the lock on the same NUMA node than to a remote NUMA node
  - **Idea:** hand over the lock locally for a while before handing it over remotely
    - Trades fairness for performance

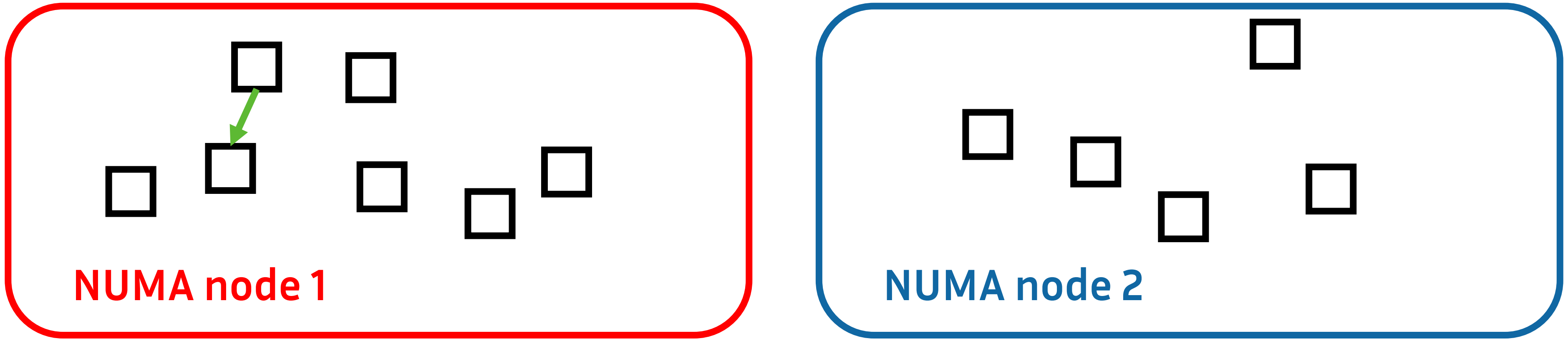
Previous work on spinlocks

- **Optimisation 3:** NUMA-awareness
  - Modern machines often have Non-Uniform Memory Architectures (NUMA)
    - *E.g., one NUMA node = one processor*
  - Faster to hand over the lock on the same NUMA node than to a remote NUMA node
  - **Idea:** hand over the lock locally for a while before handing it over remotely
    - Trades fairness for performance



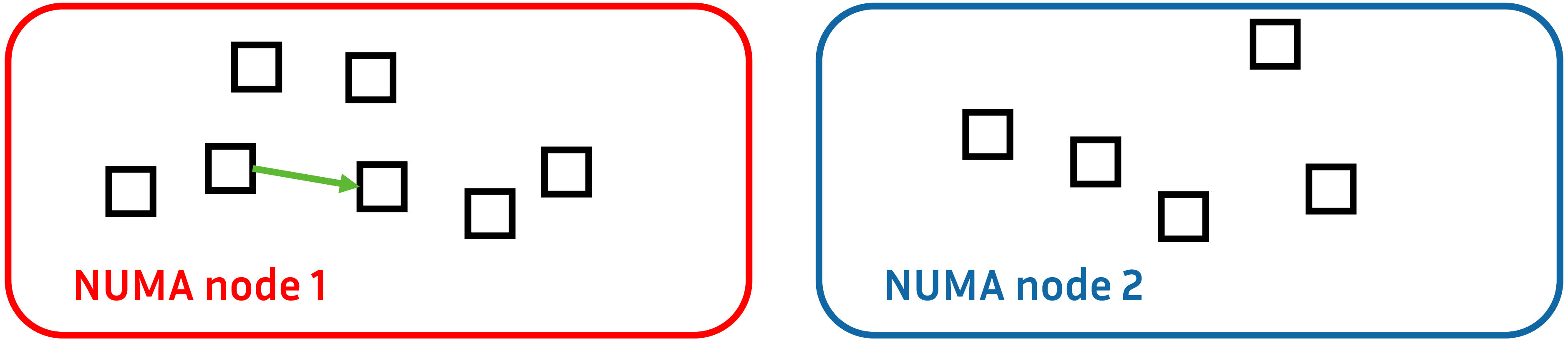
Previous work on spinlocks

- **Optimisation 3:** NUMA-awareness
  - Modern machines often have Non-Uniform Memory Architectures (NUMA)
    - *E.g., one NUMA node = one processor*
  - Faster to hand over the lock on the same NUMA node than to a remote NUMA node
  - **Idea:** hand over the lock locally for a while before handing it over remotely
    - Trades fairness for performance



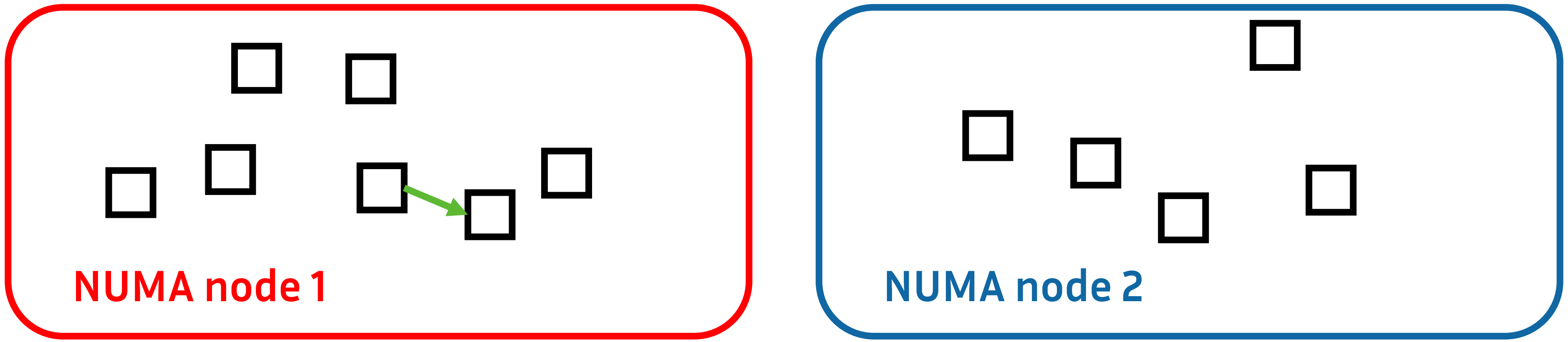
Previous work on spinlocks

- **Optimisation 3:** NUMA-awareness
  - Modern machines often have Non-Uniform Memory Architectures (NUMA)
    - *E.g., one NUMA node = one processor*
  - Faster to hand over the lock on the same NUMA node than to a remote NUMA node
  - **Idea:** hand over the lock locally for a while before handing it over remotely
    - Trades fairness for performance



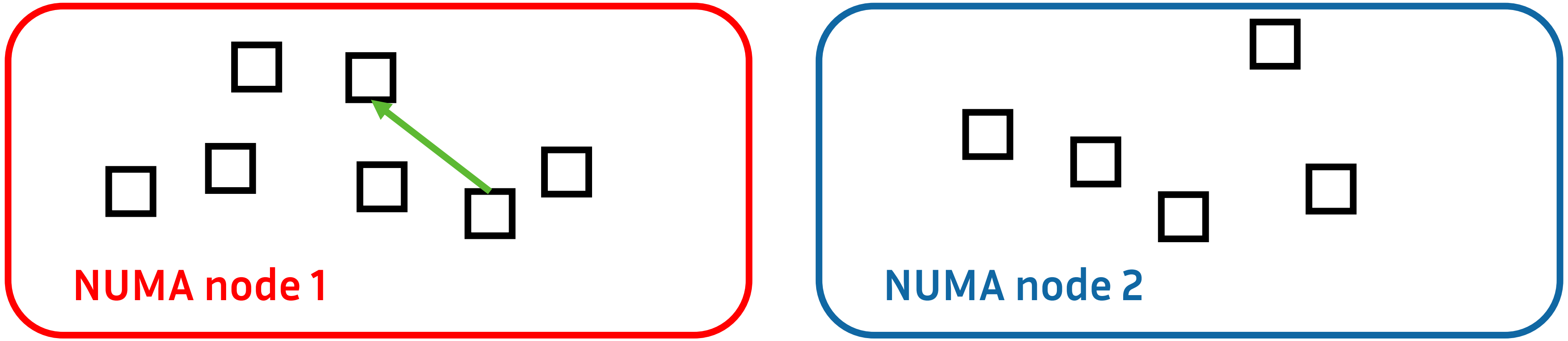
Previous work on spinlocks

- **Optimisation 3:** NUMA-awareness
  - Modern machines often have Non-Uniform Memory Architectures (NUMA)
    - *E.g., one NUMA node = one processor*
  - Faster to hand over the lock on the same NUMA node than to a remote NUMA node
  - **Idea:** hand over the lock locally for a while before handing it over remotely
    - Trades fairness for performance



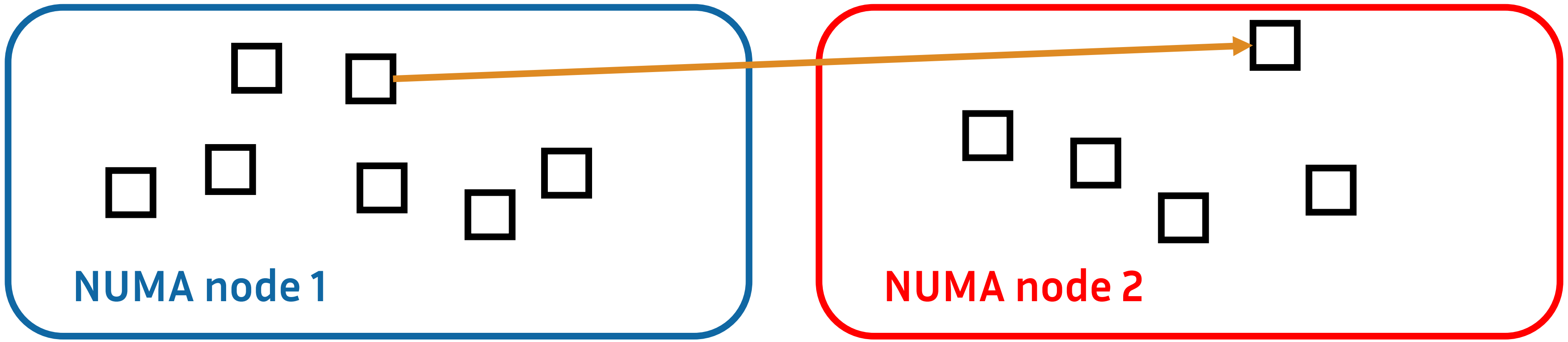
Previous work on spinlocks

- **Optimisation 3:** NUMA-awareness
  - Modern machines often have Non-Uniform Memory Architectures (NUMA)
    - *E.g., one NUMA node = one processor*
  - Faster to hand over the lock on the same NUMA node than to a remote NUMA node
  - **Idea:** hand over the lock locally for a while before handing it over remotely
    - Trades fairness for performance



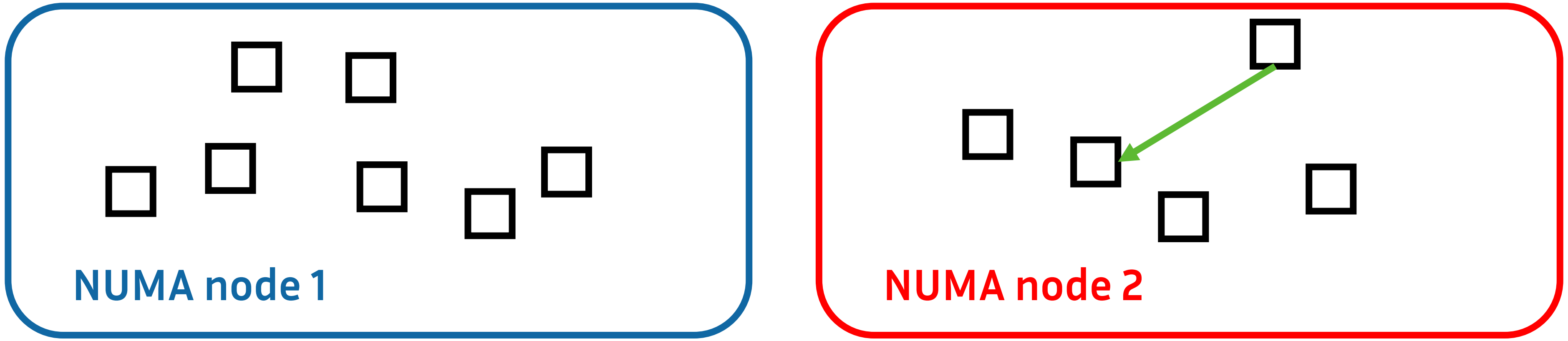
Previous work on spinlocks

- **Optimisation 3:** NUMA-awareness
  - Modern machines often have Non-Uniform Memory Architectures (NUMA)
    - *E.g., one NUMA node = one processor*
  - Faster to hand over the lock on the same NUMA node than to a remote NUMA node
  - **Idea:** hand over the lock locally for a while before handing it over remotely
    - Trades fairness for performance



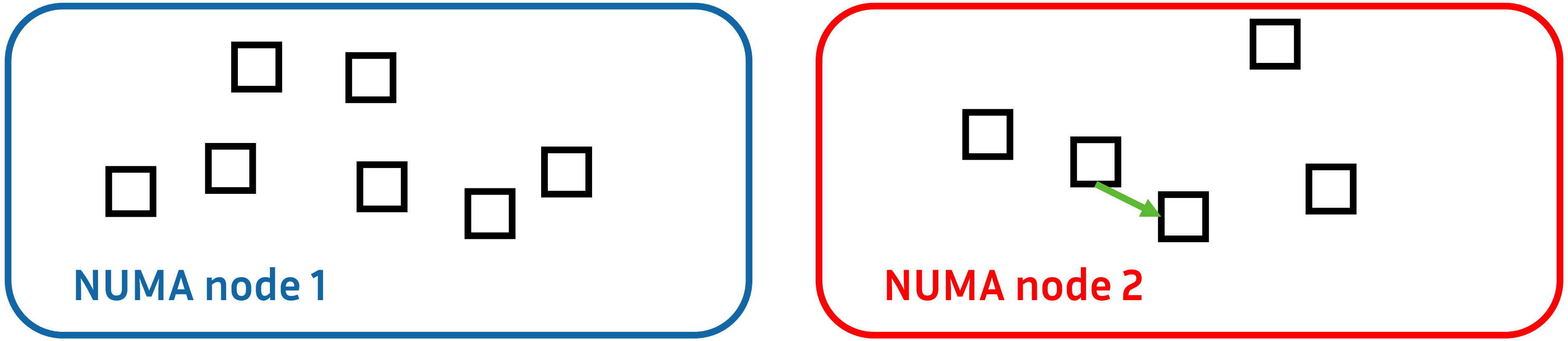
Previous work on spinlocks

- **Optimisation 3:** NUMA-awareness
  - Modern machines often have Non-Uniform Memory Architectures (NUMA)
    - *E.g., one NUMA node = one processor*
  - Faster to hand over the lock on the same NUMA node than to a remote NUMA node
  - **Idea:** hand over the lock locally for a while before handing it over remotely
    - Trades fairness for performance



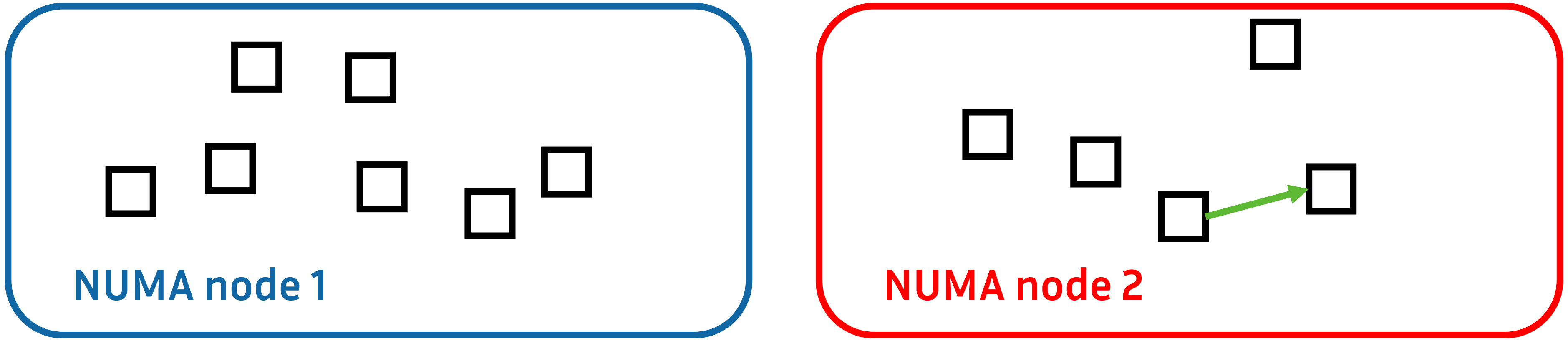
Previous work on spinlocks

- **Optimisation 3:** NUMA-awareness
  - Modern machines often have Non-Uniform Memory Architectures (NUMA)
    - *E.g., one NUMA node = one processor*
  - Faster to hand over the lock on the same NUMA node than to a remote NUMA node
  - **Idea:** hand over the lock locally for a while before handing it over remotely
    - Trades fairness for performance



Previous work on spinlocks

- **Optimisation 3:** NUMA-awareness
  - Modern machines often have Non-Uniform Memory Architectures (NUMA)
    - *E.g., one NUMA node = one processor*
  - Faster to hand over the lock on the same NUMA node than to a remote NUMA node
  - **Idea:** hand over the lock locally for a while before handing it over remotely
    - Trades fairness for performance



Previous work on spinlocks

- **Optimisation 3:** NUMA-awareness

[Dice et al., 2012]

- **Lock cohorting**: use a pair of spinlock algorithms (from TATAS, ticket, MCS, CLH…)
  - One for local nodes, one to switch between nodes

Previous work on spinlocks

```
shuffle_lock() {
    if (lock == UNLOCKED)
        locked = XCHG(&lock, LOCKED);
    if (locked != UNLOCKED)
        mcs_lock(&mcs_lock);
        while (XCHG(&lock, LOCKED) != UNLOCKED)
            PAUSE;
        mcs_unlock(&mcs_lock);
    ...
```

- **Optimisation 3:** NUMA-awareness

[Dice et al., 2012]
- **Lock cohorting**: use a pair of spinlock algorithms (from TATAS, ticket, MCS, CLH…)
  - One for local nodes, one to switch between nodes

[Kashyap et al., 2019]
- **Shuffle lock**: queue lock, move threads so that local waiters are next
  - With one extra optimisation (unrelated to NUMA): TAS + MCS

Previous work on spinlocks

```
1 shuffle_lock() {
2     if (lock == UNLOCKED)
3         locked = XCHG(&lock, LOCKED);
4     if (locked != UNLOCKED)
5         mcs_lock(&mcs_lock);
6         while (XCHG(&lock, LOCKED) != UNLOCKED)
7             PAUSE;
8         mcs_unlock(&mcs_lock);
9     ...
```

- **Optimisation 3:** NUMA-awareness

[Dice et al., 2012]
- **Lock cohorting**: use a pair of spinlock algorithms (from TATAS, ticket, MCS, CLH…)
  - One for local nodes, one to switch between nodes

[Kashyap et al., 2019]
- **Shuffle lock**: queue lock, move threads so that local waiters are next
  - With one extra optimisation (unrelated to NUMA): TAS + MCS
  - **Fast path:** just acquire the TAS lock if free (L1-2)
  - **Slow path:** acquire the MCS, acquire the TAS lock, release the MCS (L4-7)

Previous work on spinlocks

```
1 shuffle_lock() {
2     if (lock == UNLOCKED)
3         locked = XCHG(&lock, LOCKED);
4     if (locked != UNLOCKED)
5         mcs_lock(&mcs_lock);
6         while (XCHG(&lock, LOCKED) != UNLOCKED)
7             PAUSE;
8         mcs_unlock(&mcs_lock);
9     ...
```

- **Optimisation 3:** NUMA-awareness

[Dice et al., 2012]
- **Lock cohorting**: use a pair of spinlock algorithms (from TATAS, ticket, MCS, CLH…)
  - One for local nodes, one to switch between nodes

[Kashyap et al., 2019]
- **Shuffle lock**: queue lock, move threads so that local waiters are next
  - With one extra optimisation (unrelated to NUMA): TAS + MCS
  - **Fast path:** just acquire the TAS lock if free (L1-2)
  - **Slow path:** acquire the MCS, acquire the TAS lock, release the MCS (L4-7)
  - **Advantages:**
    - Fast acquisition when lock free; at most one spinner on the TAS lock
    - At most one MCS acquired at a time, lower memory consumption for nested locks
      - *Only one MCS node per thread needed, instead of one per thread per lock*

```
# Single-variable lock states
# LOCKED_WITH_BLOCKED_WAITERS = at least one thread is blocking,
# the holder should call futex_wake when releasing the lock
UNLOCKED = 0, LOCKED = 1, LOCKED_WITH_BLOCKED_WAITERS = 2
# CS preemption counter updated by the eBPF Preemption Monitor
num_preempted_cs = 0

class Lock:
  val = UNLOCKED, queue = None # Single-variable lock, MCS tail
class QNode:
  next = None, waiting = False

def mcs_exit(lock: Lock, qnode: QNode):
  if qnode.next is None:
    if CAS(&lock.queue, qnode, None) == qnode:
      return
    while qnode.next is None:
      PAUSE()
  qnode.next.waiting = False

def flexguard_unlock(lock: Lock, qnode: QNode):
  qnode.cs_counter -= 1
label at_unlock
  if XCHG(&lock.val, UNLOCKED) == LOCKED_WITH_BLOCKED_WAITERS:
    futex_wake(&lock.val, 1) # Wake one of the waiting threads

def flexguard_lock(lock: Lock, qnode: QNode):
label at_fastpath # Try to steal the single-variable lock if free
  if lock.val == UNLOCKED and CAS(&lock.val, UNLOCKED, LOCKED):
    qnode.cs_counter += 1
    return
  # There are waiters in the queue, enter the slow path
  flexguard_slow_path(lock, qnode)
```

```
def flexguard_slow_path(lock, qnode):
  enqueued = False
  if num_preempted_cs == 0: # If spinning, begin Phase 1
    enqueued = True
    qnode.next = None
    qnode.waiting = True
label at_xchg
    pred = XCHG(&lock.queue, qnode)
    if pred is not None:
      pred.next = qnode
      while qnode.waiting and num_preempted_cs == 0:
        PAUSE()
label at_phase2 # Begin Phase 2
  state = CAS(&lock.val, UNLOCKED, LOCKED)
  while state != UNLOCKED:
    if num_preempted_cs == 0: # Busy-waiting mode
      PAUSE()
      state = CAS(&lock.val, UNLOCKED, LOCKED)
    else: # Blocking mode
      if enqueued:
        mcs_exit(lock, qnode)
        enqueued = False
      if state != LOCKED_WITH_BLOCKED_WAITERS:
        state = XCHG(&lock.val, LOCKED_WITH_BLOCKED_WAITERS)
      if state != UNLOCKED:
        futex_wait(&lock.val, LOCKED_WITH_BLOCKED_WAITERS)
        state = XCHG(&lock.val, LOCKED_WITH_BLOCKED_WAITERS)
        if state != UNLOCKED and num_preempted_cs == 0:
          # Back to spin mode, restart slow path (using MCS)
          return flexguard_slow_path(lock, qnode)
  if enqueued: # Exit the queue if still enqueued
    mcs_exit(lock, qnode)
  qnode.cs_counter += 1
```

FlexGuard’s lock algorithm

- Similar TAS+MCS optimization as the **Shuffle lock**
  - But the TAS lock variable can also be used as the FUTEX lock variable

```
# Single-variable lock states
# LOCKED_WITH_BLOCKED_WAITERS = at least one thread is blocking,
# the holder should call futex_wake when releasing the lock
UNLOCKED = 0, LOCKED = 1, LOCKED_WITH_BLOCKED_WAITERS = 2
# CS preemption counter updated by the eBPF Preemption Monitor
num_preempted_cs = 0

class Lock:
  val = UNLOCKED, queue = None # Single-variable lock, MCS tail
class QNode:
  next = None, waiting = False

def mcs_exit(lock: Lock, qnode: QNode):
  if qnode.next is None:
    if CAS(&lock.queue, qnode, None) == qnode:
      return
    while qnode.next is None:
      PAUSE()
  qnode.next.waiting = False

def flexguard_unlock(lock: Lock, qnode: QNode):
  qnode.cs_counter -= 1
label at_unlock
  if XCHG(&lock.val, UNLOCKED) == LOCKED_WITH_BLOCKED_WAITERS:
    futex_wake(&lock.val, 1) # Wake one of the waiting threads

def flexguard_lock(lock: Lock, qnode: QNode):
label at_fastpath # Try to steal the single-variable lock if free
  if lock.val == UNLOCKED and CAS(&lock.val, UNLOCKED, LOCKED):
    qnode.cs_counter += 1
    return
  # There are waiters in the queue, enter the slow path
  flexguard_slow_path(lock, qnode)
```

```
def flexguard_slow_path(lock, qnode):
  enqueued = False
  if num_preempted_cs == 0: # If spinning, begin Phase 1
    enqueued = True
    qnode.next = None
    qnode.waiting = True
label at_xchg
    pred = XCHG(&lock.queue, qnode)
    if pred is not None:
      pred.next = qnode
      while qnode.waiting and num_preempted_cs == 0:
        PAUSE()
label at_phase2 # Begin Phase 2
  state = CAS(&lock.val, UNLOCKED, LOCKED)
  while state != UNLOCKED:
    if num_preempted_cs == 0: # Busy-waiting mode
      PAUSE()
      state = CAS(&lock.val, UNLOCKED, LOCKED)
    else: # Blocking mode
      if enqueued:
        mcs_exit(lock, qnode)
        enqueued = False
      if state != LOCKED_WITH_BLOCKED_WAITERS:
        state = XCHG(&lock.val, LOCKED_WITH_BLOCKED_WAITERS)
      if state != UNLOCKED:
        futex_wait(&lock.val, LOCKED_WITH_BLOCKED_WAITERS)
        state = XCHG(&lock.val, LOCKED_WITH_BLOCKED_WAITERS)
        if state != UNLOCKED and num_preempted_cs == 0:
          # Back to spin mode, restart slow path (using MCS)
          return flexguard_slow_path(lock, qnode)
  if enqueued: # Exit the queue if still enqueued
    mcs_exit(lock, qnode)
  qnode.cs_counter += 1
```

FlexGuard's lock algorithm

- Similar TAS+MCS optimization as the **Shuffle lock**
  - But the TAS lock variable can also be used as the FUTEX lock variable

⇒ Possible to acquire the lock as spinning or blocking

FlexGuard's lock algorithm

- Similar TAS+MCS optimization as the Shuffle lock
  - But the TAS lock variable can also be used as the FUTEX lock variable

⇒ Possible to acquire the lock as spinning or blocking

⇒ **No atomicity issues**

FlexGuard's lock algorithm

- Similar TAS+MCS optimization as the Shuffle lock
  - But the TAS lock variable can also be used as the FUTEX lock variable

⇒ Possible to acquire the lock as spinning or blocking

⇒ **No atomicity issues**

- **Spin mode ⇔ `num_preempted_cs == 0`**

FlexGuard's lock algorithm

- Similar TAS+MCS optimization as the Shuffle lock
  - But the TAS lock variable can also be used as the FUTEX lock variable

  ⇒ Possible to acquire the lock as spinning or blocking

  ⇒ **No atomicity issues**

- **Spin mode ⇔ `num_preempted_cs == 0`**
- **In spin mode:** similar behavior as the Shuffle lock, except no NUMA reshuffling

FlexGuard’s lock algorithm

- Similar TAS+MCS optimization as the Shuffle lock
  - But the TAS lock variable can also be used as the FUTEX lock variable

  ⇒ Possible to acquire the lock as spinning or blocking

  ⇒ **No atomicity issues**

- **Spin mode ⇔ `num_preempted_cs == 0`**
- **In spin mode:** similar behavior as the Shuffle lock, except no NUMA reshuffling
  - *NUMA has little impact on recent x86 machines*

FlexGuard's lock algorithm

- Similar TAS+MCS optimization as the Shuffle lock
  - But the TAS lock variable can also be used as the FUTEX lock variable

  ⇒ Possible to acquire the lock as spinning or blocking

  ⇒ **No atomicity issues**

- **Spin mode ⇔ `num_preempted_cs == 0`**
- **In spin mode:** similar behavior as the Shuffle lock, except no NUMA reshuffling
  - *NUMA has little impact on recent x86 machines*
- **In blocking mode:** MCS queue bypassed!

FlexGuard's lock algorithm

- Similar TAS+MCS optimization as the Shuffle lock
  - But the TAS lock variable can also be used as the FUTEX lock variable

  ⇒ Possible to acquire the lock as spinning or blocking

  ⇒ **No atomicity issues**

- **Spin mode ⇔ `num_preempted_cs == 0`**
- **In spin mode:** similar behavior as the Shuffle lock, except no NUMA reshuffling
  - *NUMA has little impact on recent x86 machines*
- **In blocking mode:** MCS queue bypassed!

- **Spinning→blocking transition:** spin waiters exit the MCS queue

FlexGuard's lock algorithm

- Similar TAS+MCS optimization as the Shuffle lock
  - But the TAS lock variable can also be used as the FUTEX lock variable

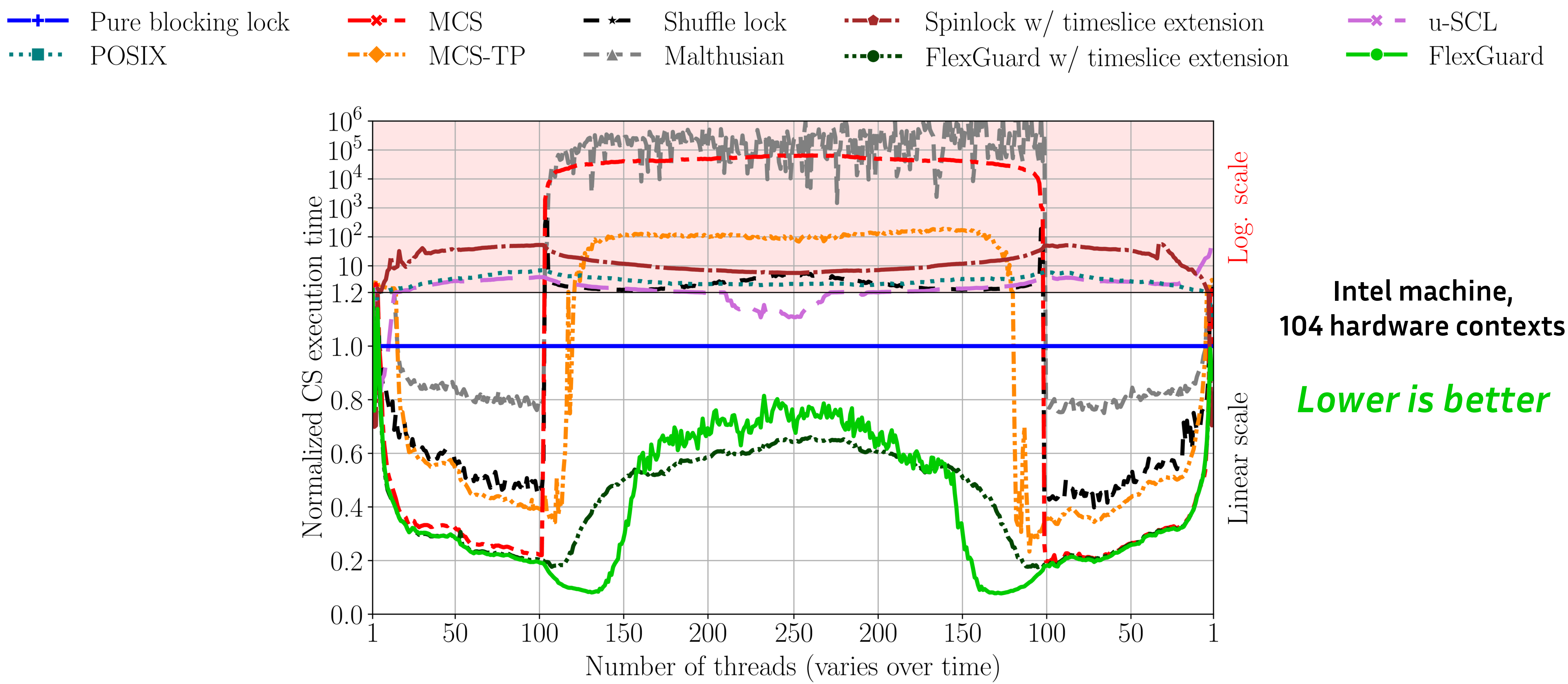
  ⇒ Possible to acquire the lock as spinning or blocking

  ⇒ **No atomicity issues**

- **Spin mode ⇔ `num_preempted_cs == 0`**
- **In spin mode:** similar behavior as the Shuffle lock, except no NUMA reshuffling
  - *NUMA has little impact on recent x86 machines*
- **In blocking mode:** MCS queue bypassed!

- **Spinning→blocking transition:** spin waiters exit the MCS queue
- **Blocking→spinning transition:** blocking waiters reenqueue themselves in the MCS queue
  - *One woken up at each `unlock()` if there are blocked waiters, TAS attempt then reenqueuing*

Evaluation: microbenchmark (Intel)

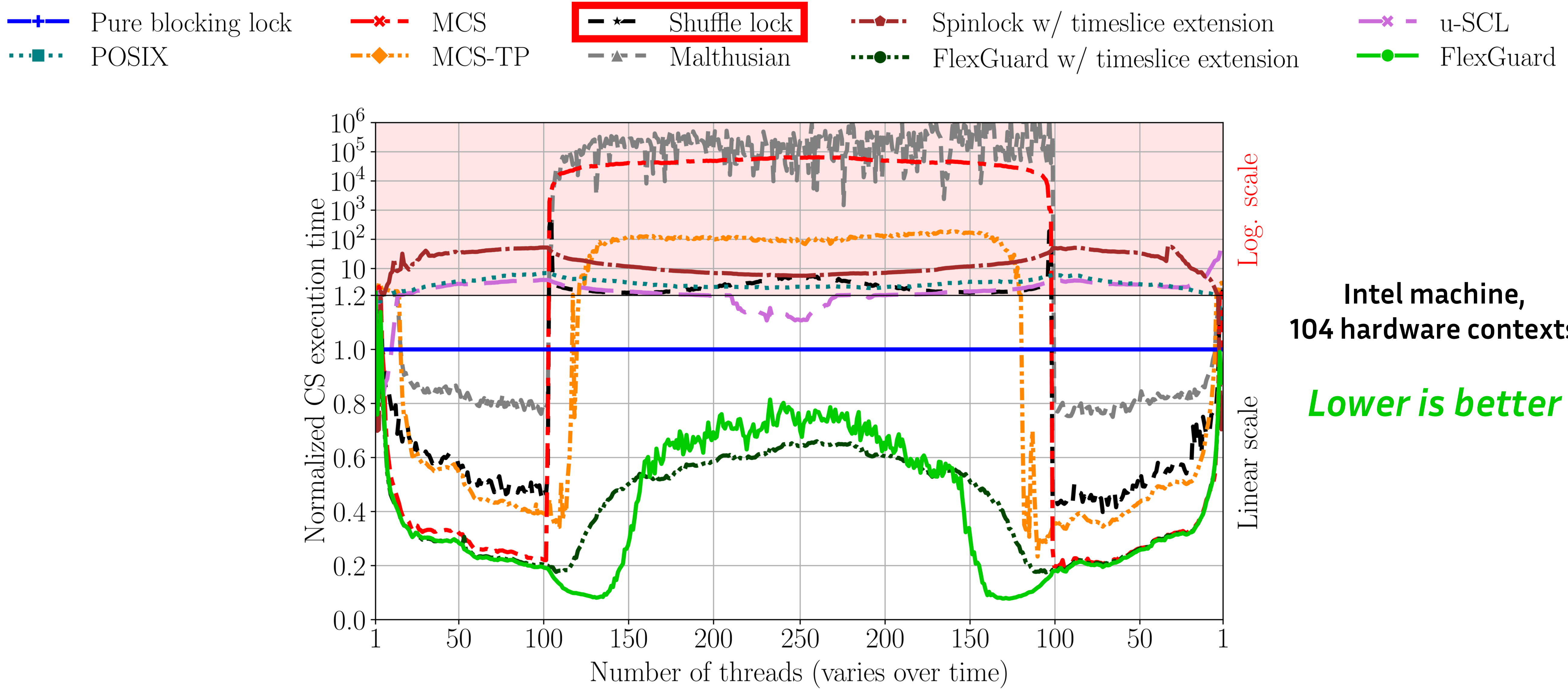


Intel machine,
104 hardware contexts

Lower is better

Evaluation: microbenchmark (Intel)

With spin-then-park

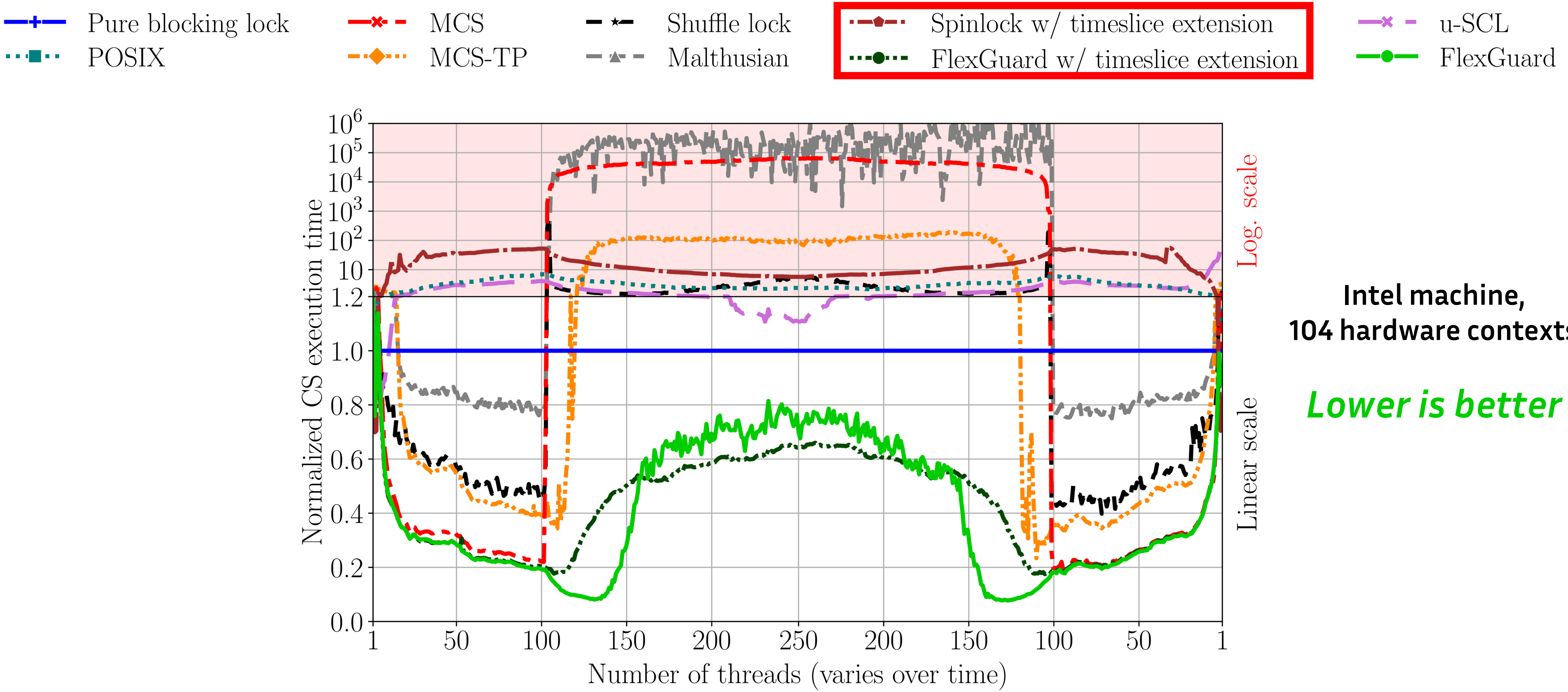


Intel machine, 104 hardware contexts

Lower is better

Evaluation: microbenchmark (Intel)

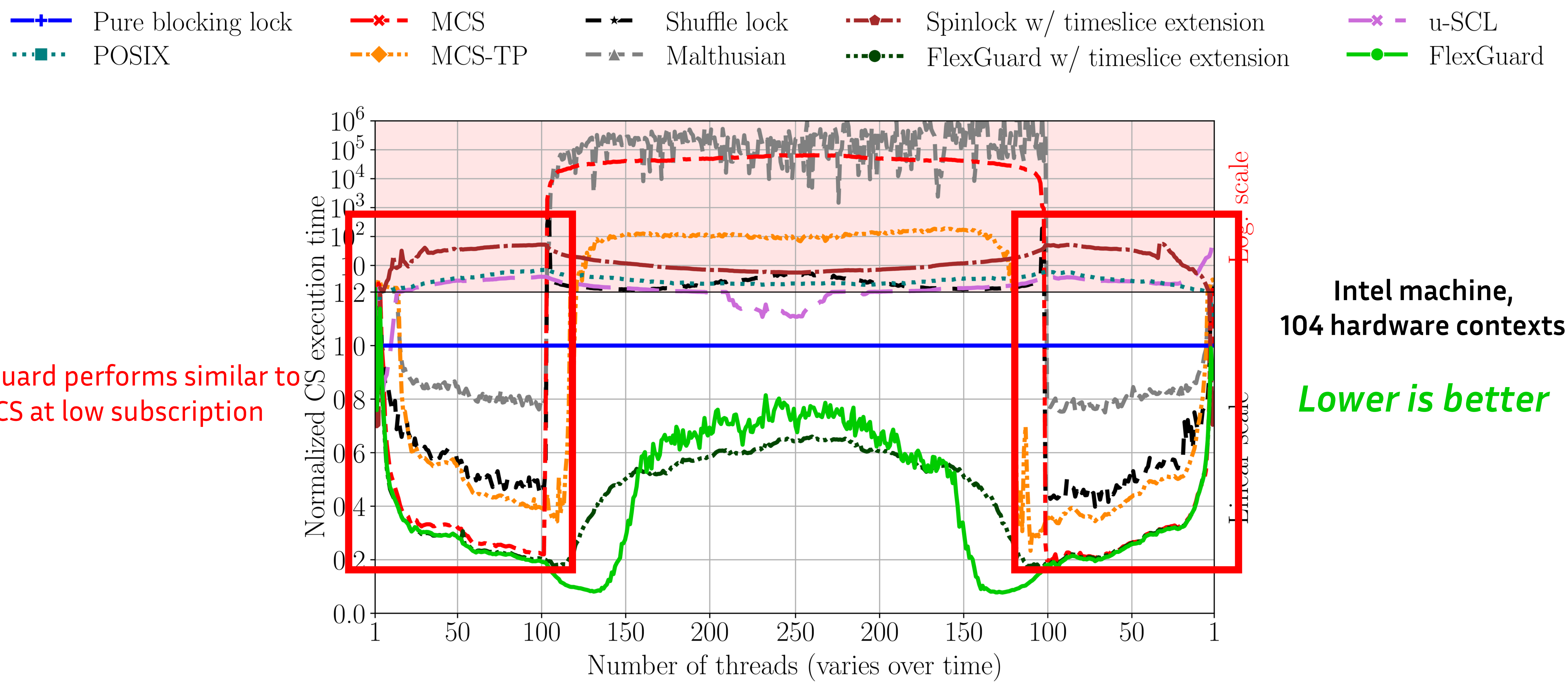
Recently proposed patch for Linux
Applicable to our approach



Intel machine,
104 hardware contexts

Lower is better

Evaluation: microbenchmark (Intel)

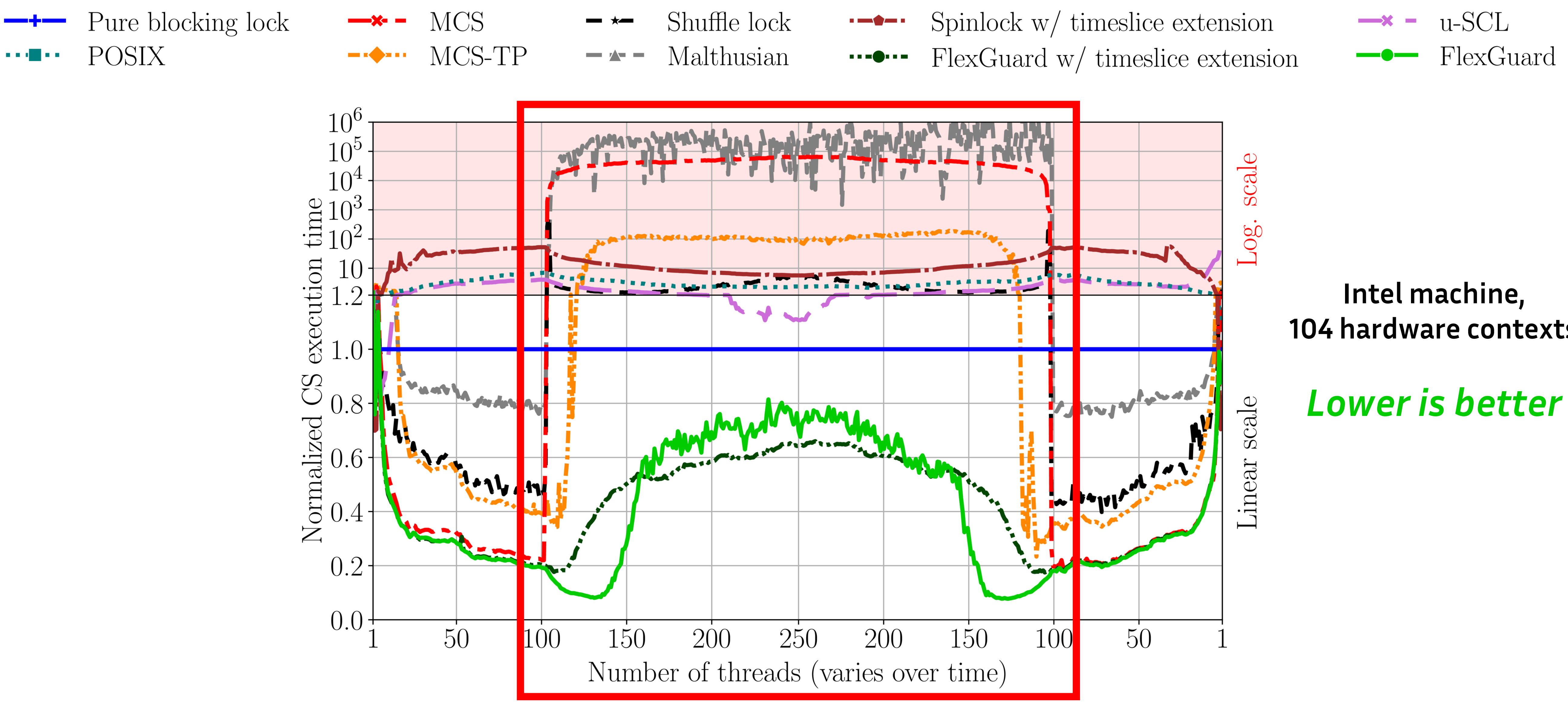


FlexGuard performs similar to MCS at low subscription

Intel machine,
104 hardware contexts

Lower is better

Evaluation: microbenchmark (Intel)



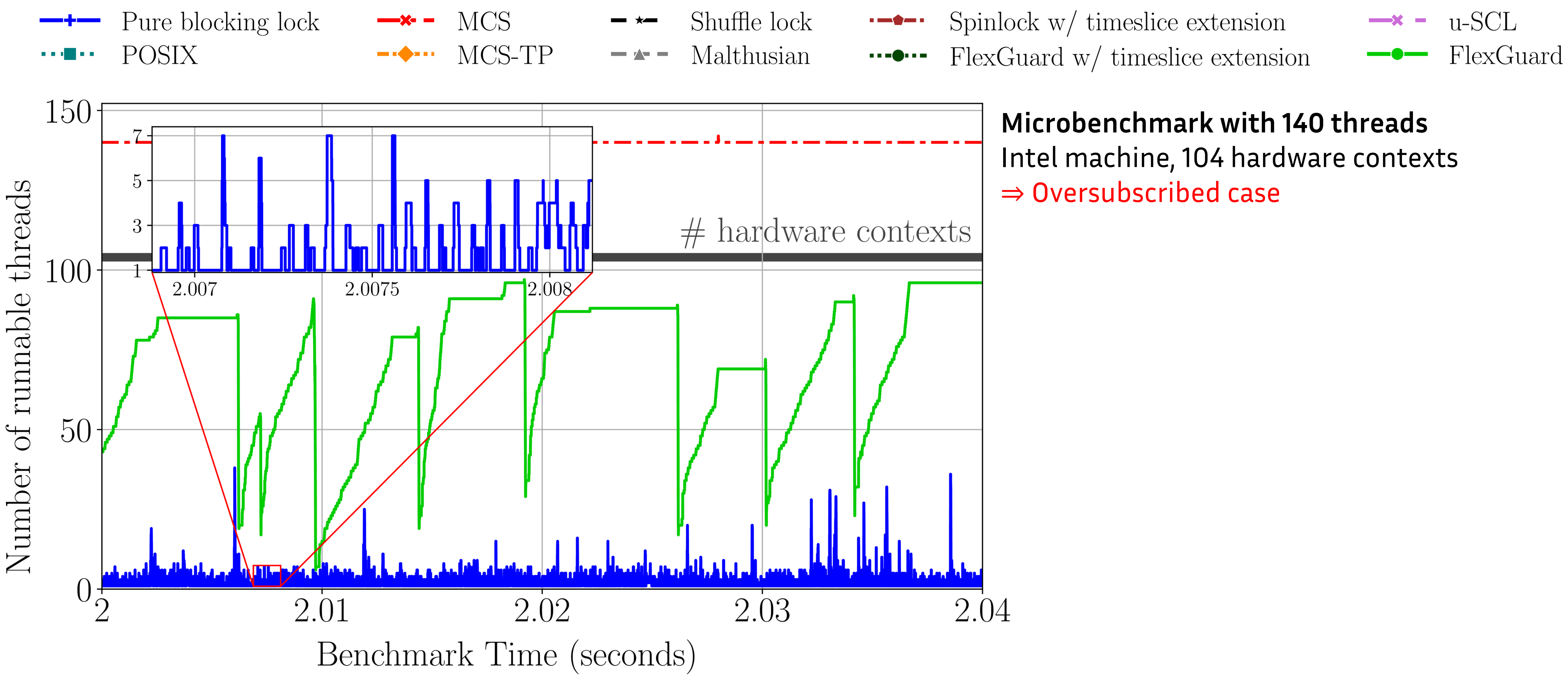
Intel machine, 104 hardware contexts

Lower is better

FlexGuard avoids the performance collapse at high subscription

Even greatly outperforms the blocking locks… but why?

Runnable threads

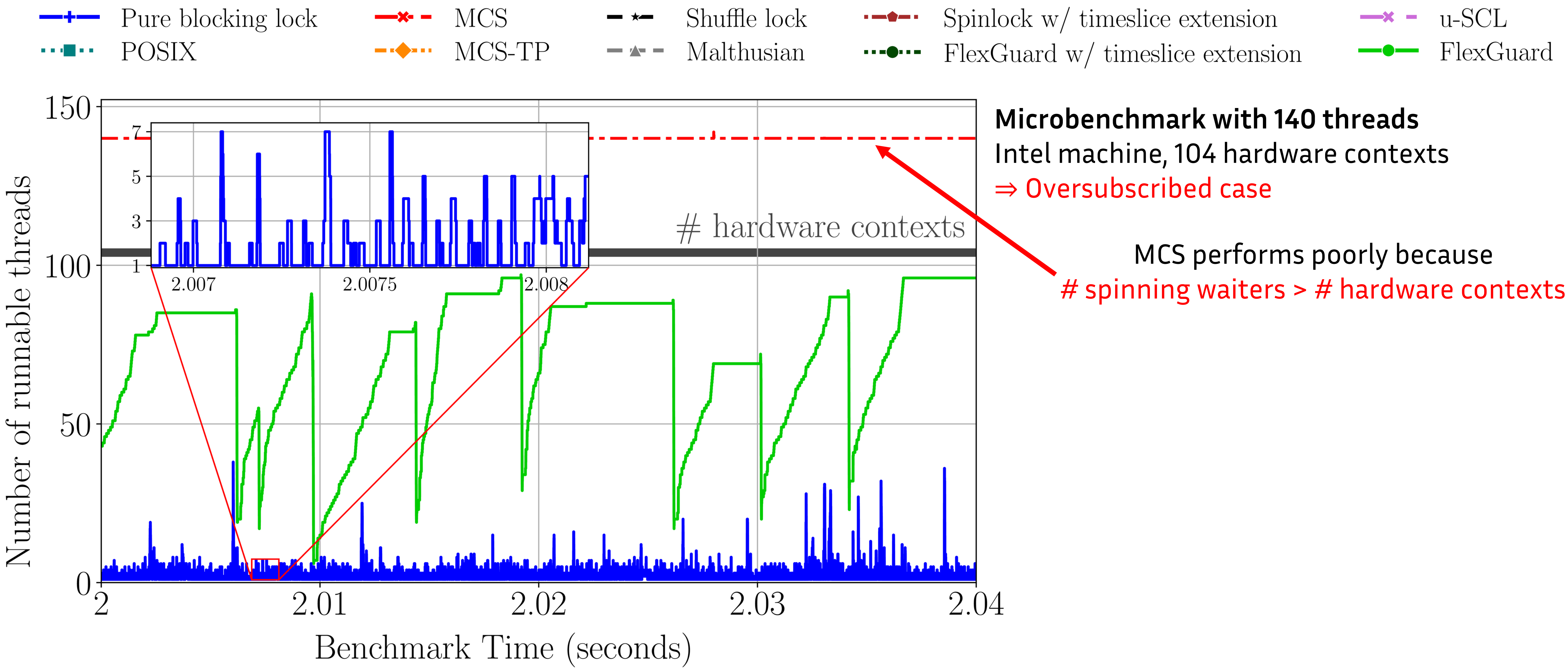


Microbenchmark with 140 threads

Intel machine, 104 hardware contexts

⇒ Oversubscribed case

Runnable threads



Microbenchmark with 140 threads

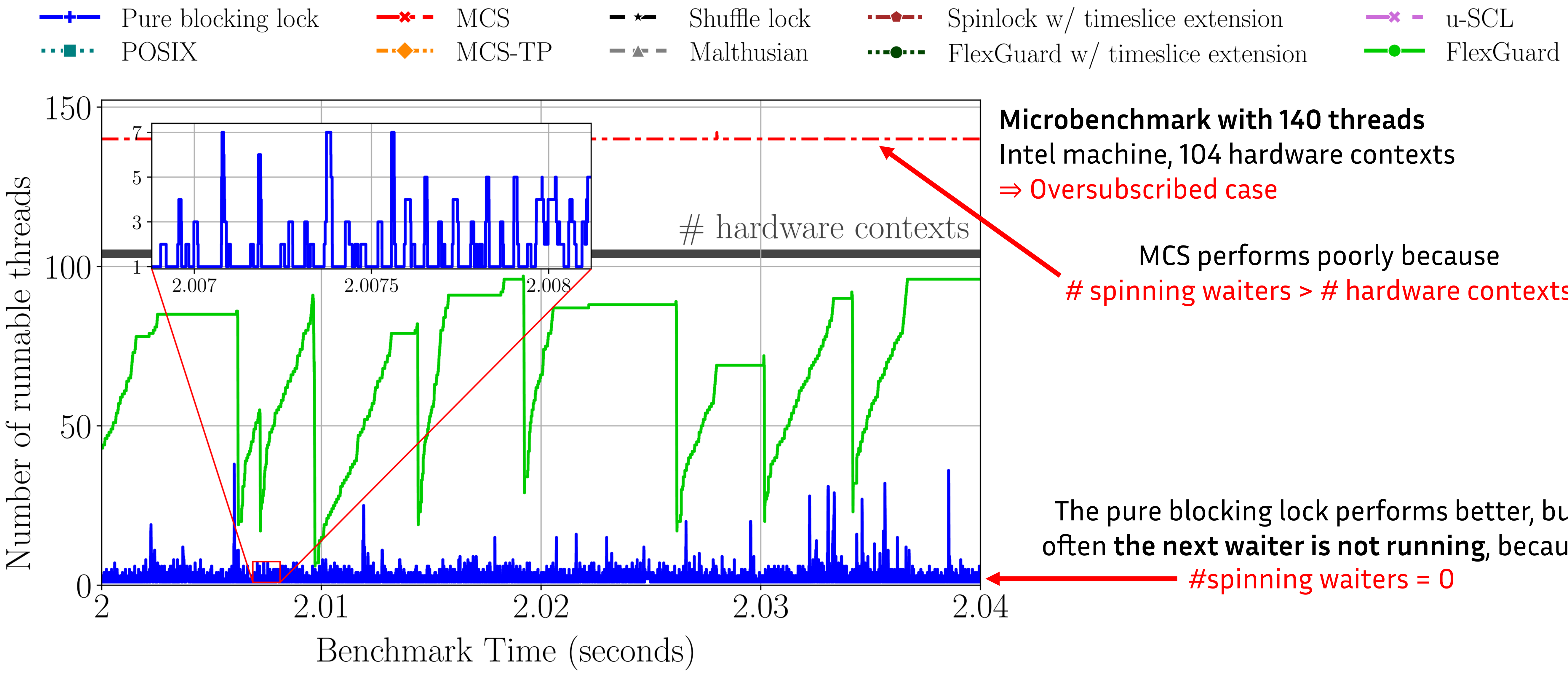
Intel machine, 104 hardware contexts

⇒ Oversubscribed case

MCS performs poorly because

spinning waiters > # hardware contexts

Runnable threads



Microbenchmark with 140 threads

Intel machine, 104 hardware contexts

⇒ Oversubscribed case

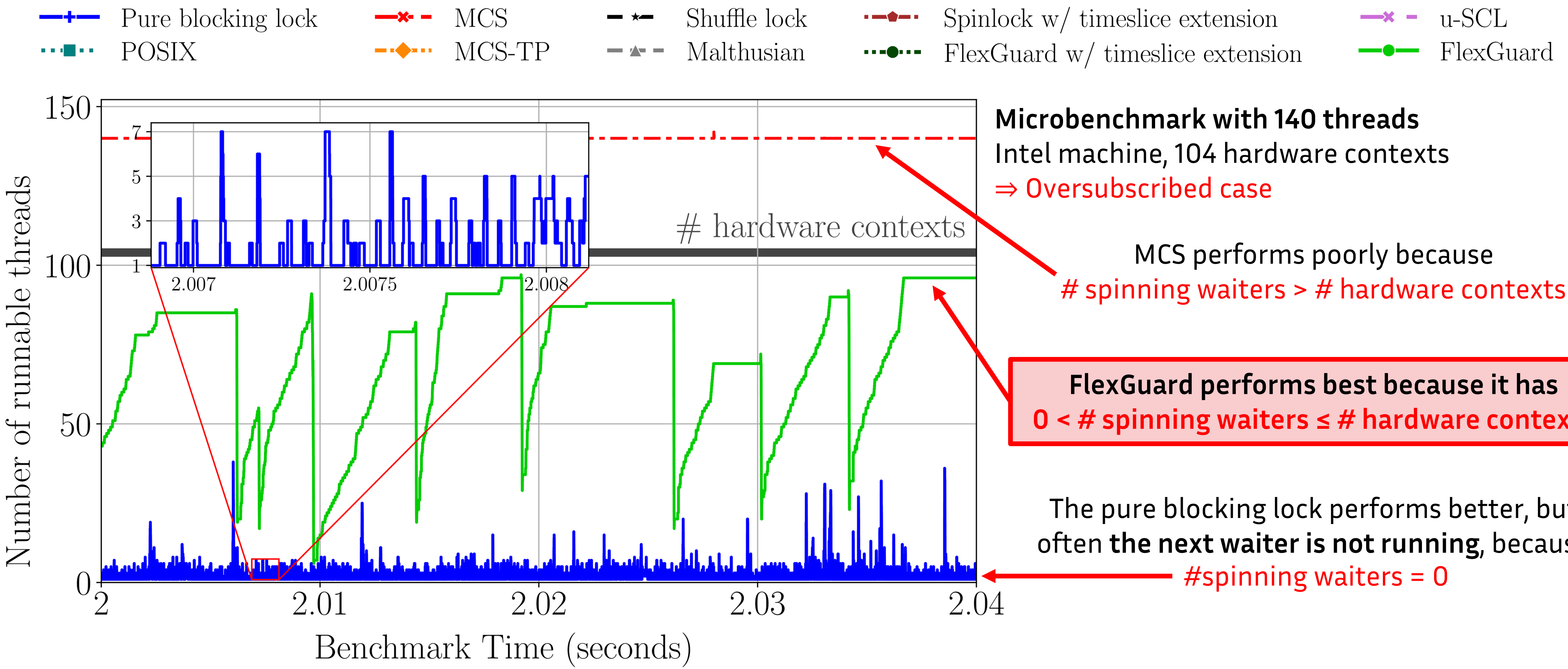
MCS performs poorly because

spinning waiters > # hardware contexts

The pure blocking lock performs better, but often **the next waiter is not running**, because

#spinning waiters = 0

Runnable threads



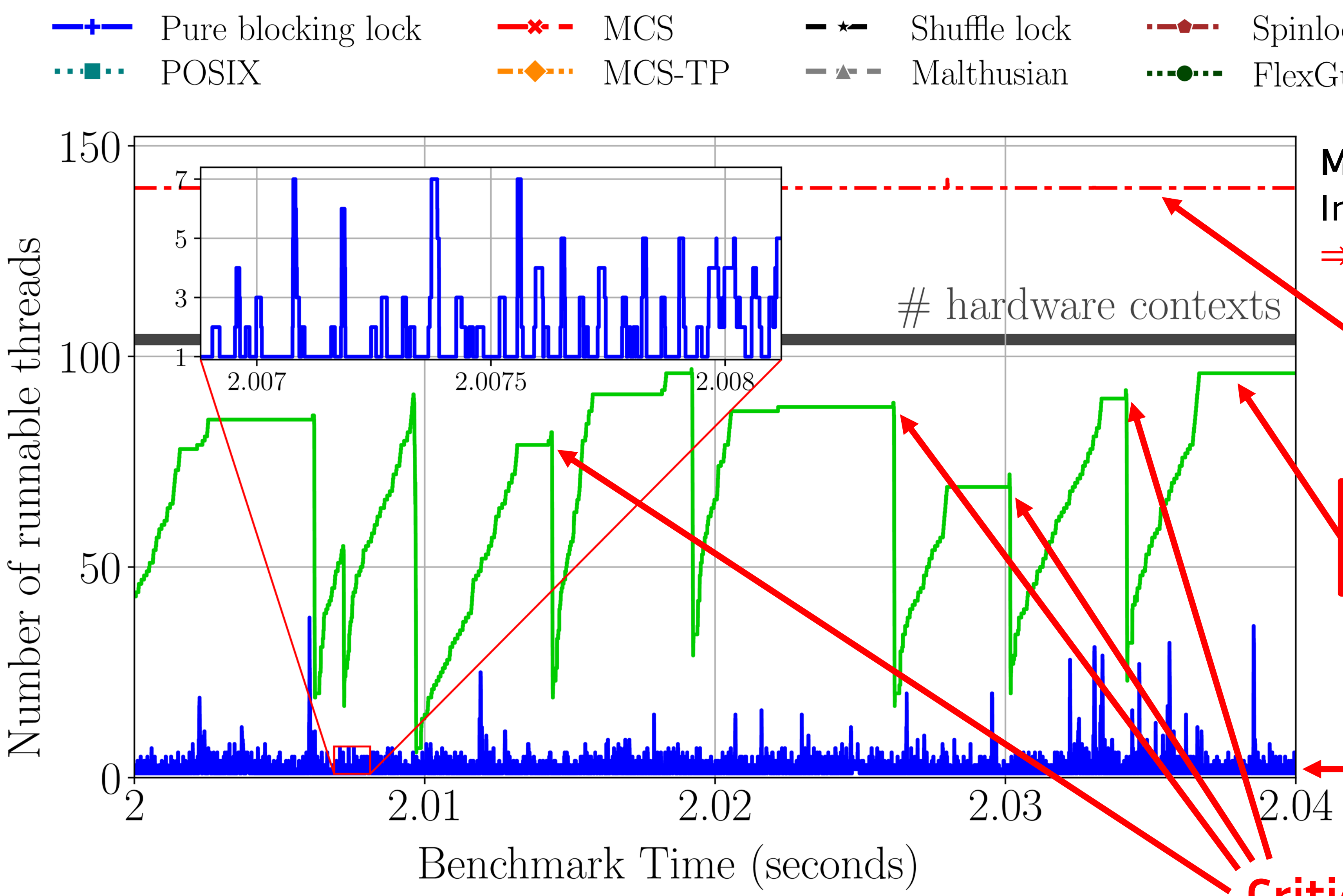
Microbenchmark with 140 threads
Intel machine, 104 hardware contexts
⇒ Oversubscribed case

MCS performs poorly because
spinning waiters > # hardware contexts

FlexGuard performs best because it has
0 < # spinning waiters ≤ # hardware contexts

The pure blocking lock performs better, but often **the next waiter is not running**, because
#spinning waiters = 0

Runnable threads



Microbenchmark with 140 threads
Intel machine, 104 hardware contexts
⇒ Oversubscribed case

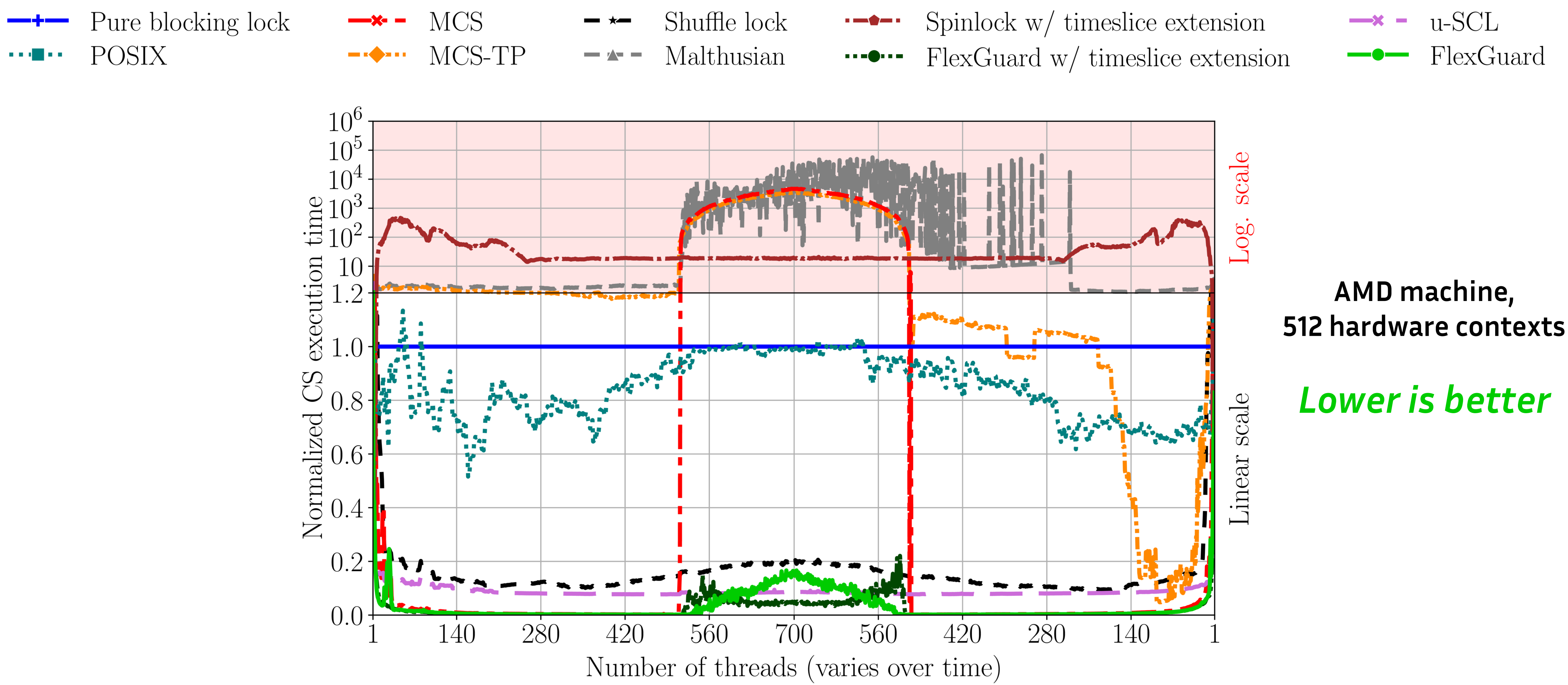
MCS performs poorly because
spinning waiters > # hardware contexts

FlexGuard performs best because it has
0 < # spinning waiters ≤ # hardware contexts

The pure blocking lock performs better, but often **the next waiter is not running**, because
#spinning waiters = 0

Critical section preemptions

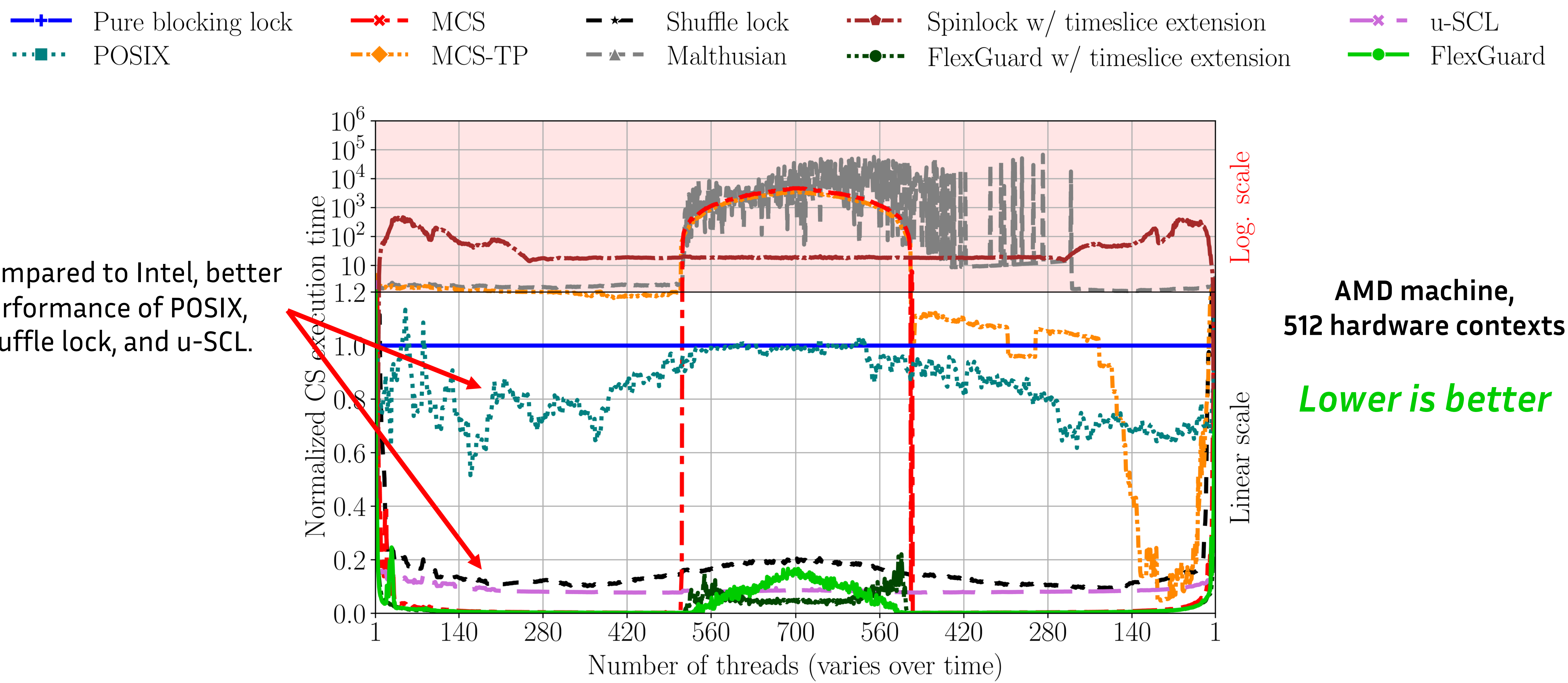
Evaluation: microbenchmark (AMD)



AMD machine, 512 hardware contexts

Lower is better

Evaluation: microbenchmark (AMD)

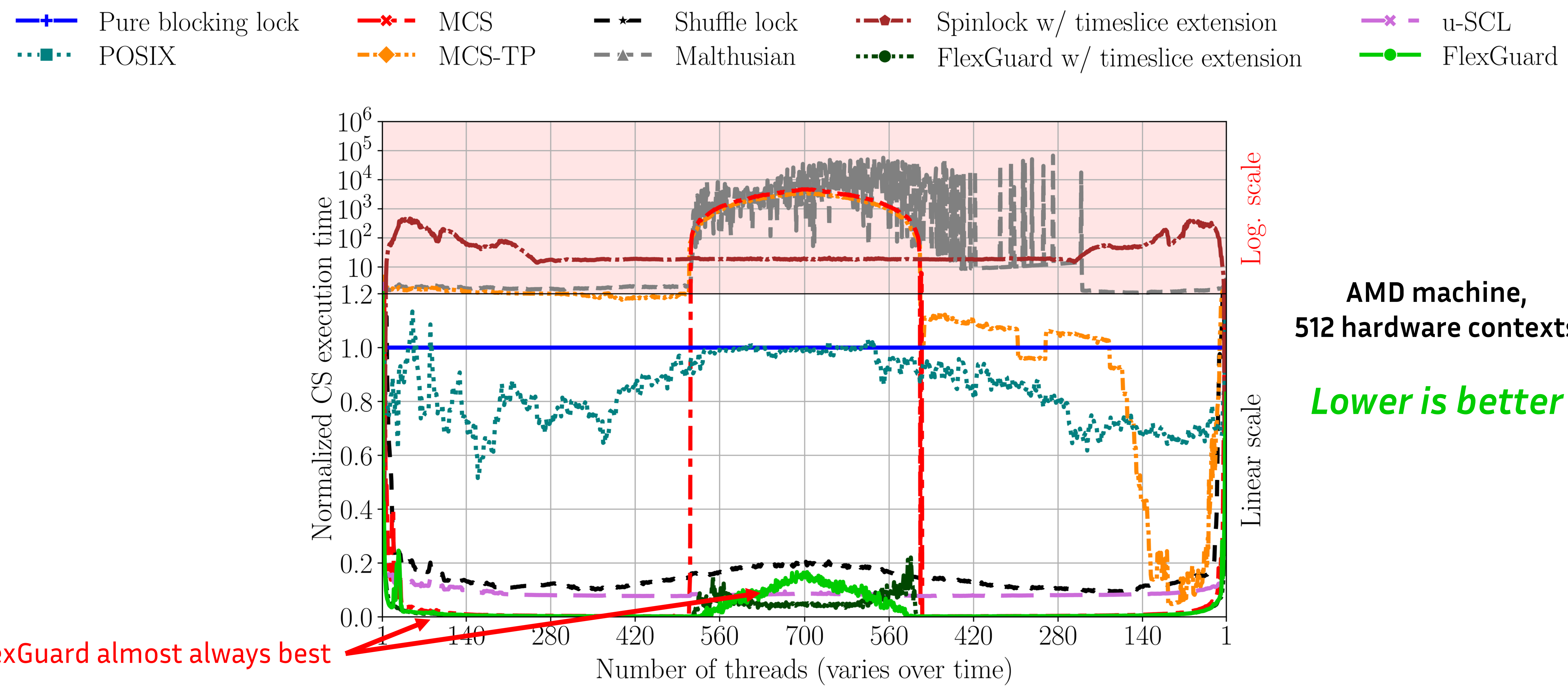


As compared to Intel, better performance of POSIX, Shuffle lock, and u-SCL.

AMD machine, 512 hardware contexts

Lower is better

Evaluation: microbenchmark (AMD)

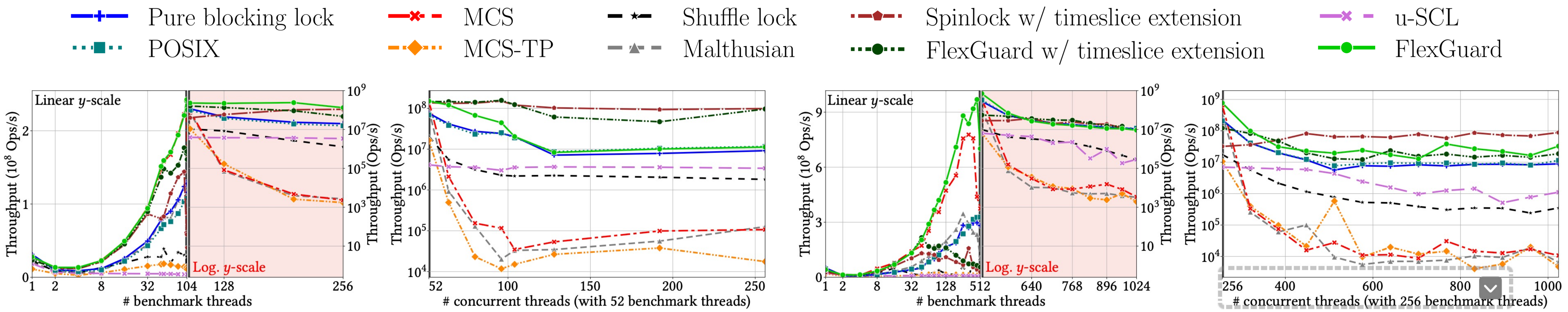


FlexGuard almost always best

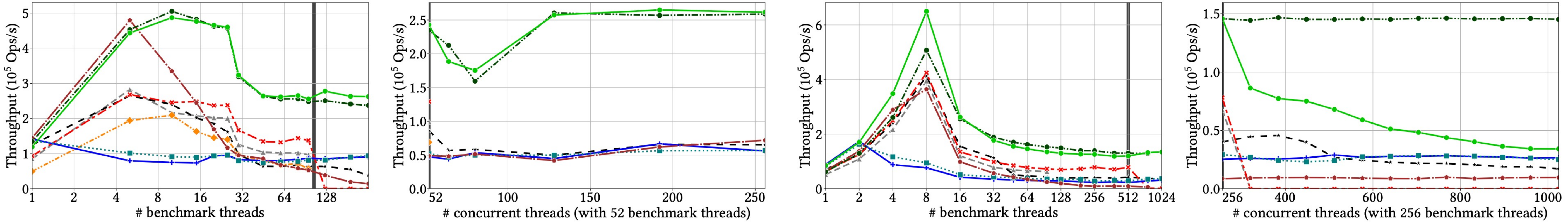
AMD machine,
512 hardware contexts

Lower is better

Evaluation: benchmarks



(a) Hash table on Intel. **(b)** Hash table on Intel w/ conc. **(c)** Hash table on AMD. **(d)** Hash table on AMD w/ conc.

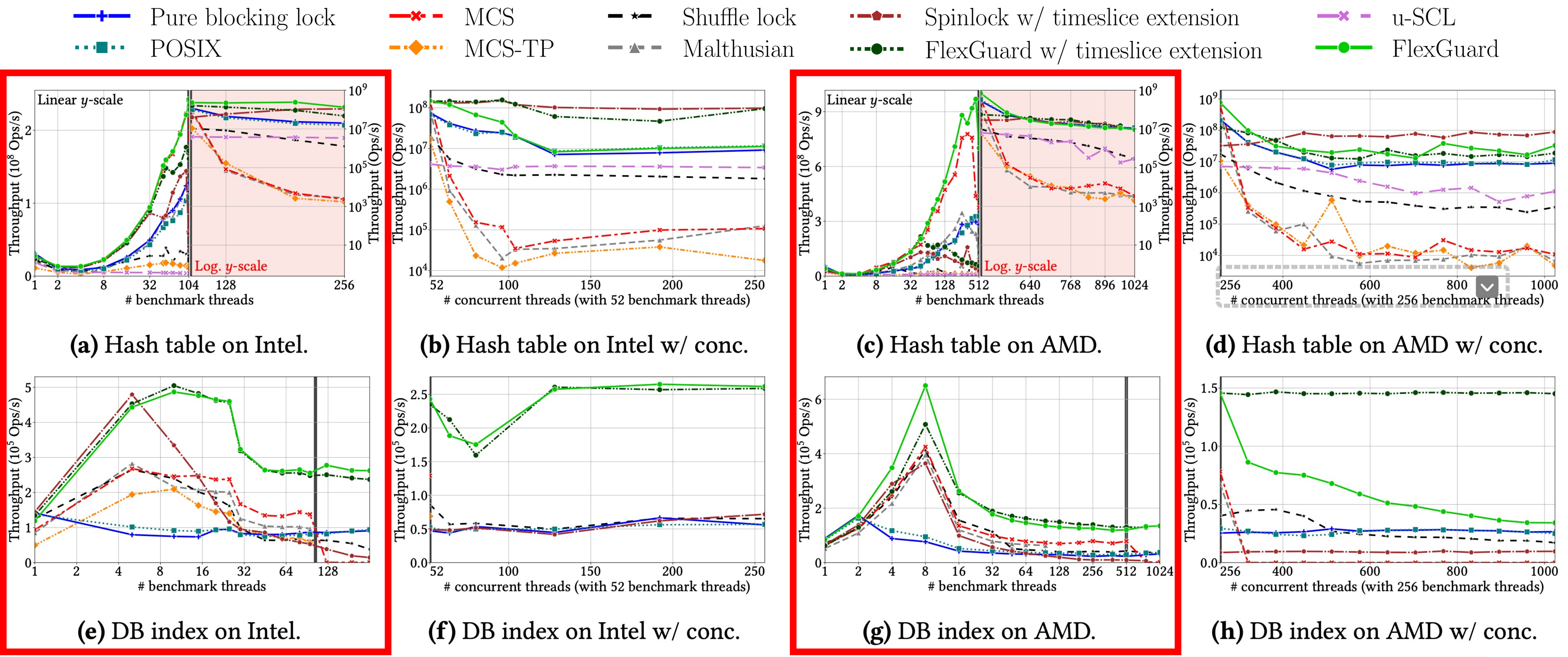


(e) DB index on Intel. **(f)** DB index on Intel w/ conc. **(g)** DB index on AMD. **(h)** DB index on AMD w/ conc.

Higher is better

Evaluation: benchmarks

FlexGuard matches or outperforms other locks on both low and high subscription

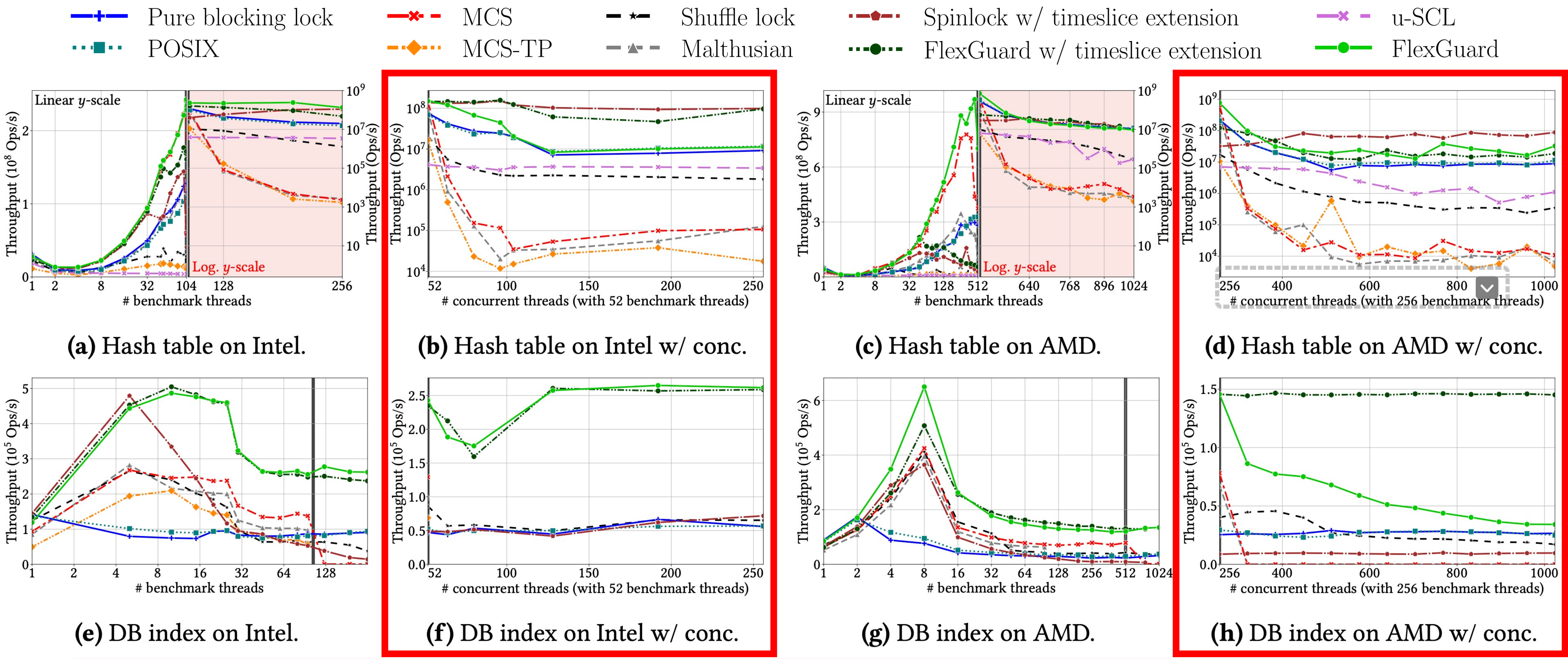


Higher is better

Evaluation: benchmarks

Benchmarks w/ concurrent workload: fixed number of benchmark threads, varying number of concurrent threads

In this case, timeslice extension helps because the concurrent workload's threads cannot be blocked

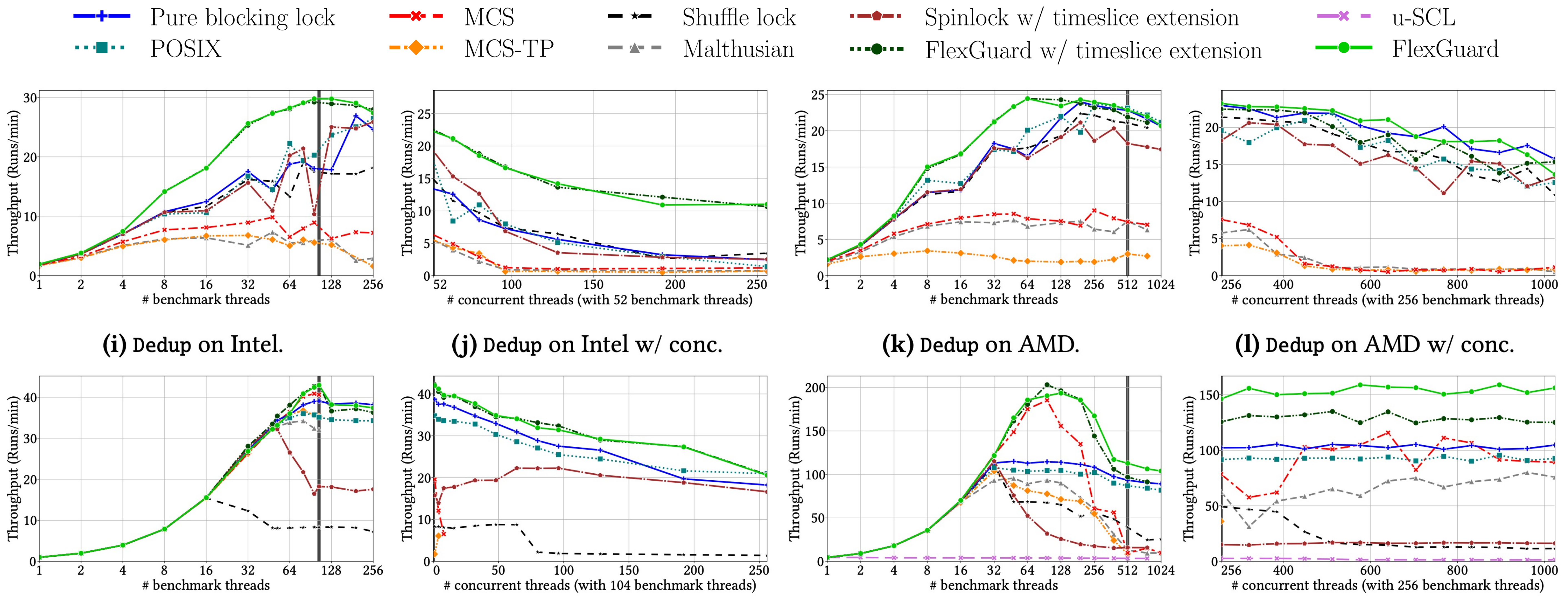


(a) Hash table on Intel. **(b)** Hash table on Intel w/ conc. **(c)** Hash table on AMD. **(d)** Hash table on AMD w/ conc.

(e) DB index on Intel. **(f)** DB index on Intel w/ conc. **(g)** DB index on AMD. **(h)** DB index on AMD w/ conc.

Higher is better

Evaluation: benchmarks



(i) Dedup on Intel.

(j) Dedup on Intel w/ conc.

(k) Dedup on AMD.

(l) Dedup on AMD w/ conc.

(m) Raytrace on Intel.

(n) Raytrace on Intel w/ conc.

(o) Raytrace on AMD.

(p) Raytrace on AMD w/ conc.

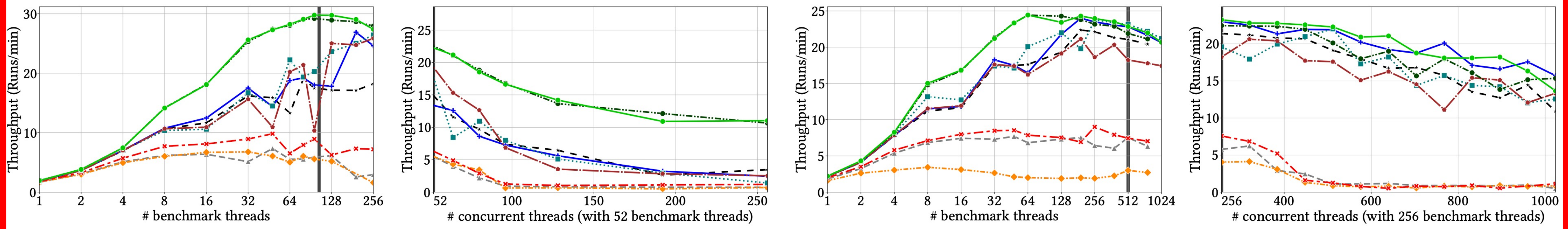
Higher is better

Evaluation: benchmarks

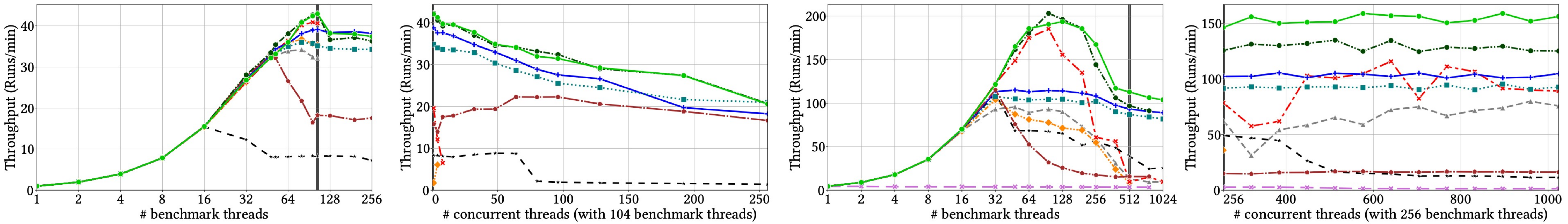
Poor performance of MCS, MCS-TP, and Malthusian, due to the high number of locks (266K): one queue node per thread **and per lock**, many cache misses

FlexGuard performs well due to the Shuffle lock optimization

Pure blocking lock · MCS · Shuffle lock · Spinlock w/ timeslice extension · u-SCL
POSIX · MCS-TP · Malthusian · FlexGuard w/ timeslice extension · FlexGuard

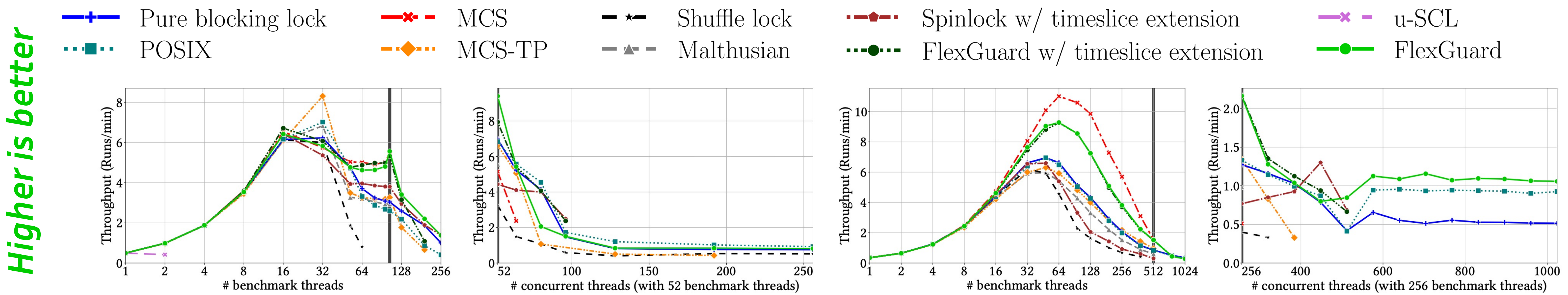


(i) Dedup on Intel. **(j)** Dedup on Intel w/ conc. **(k)** Dedup on AMD. **(l)** Dedup on AMD w/ conc.

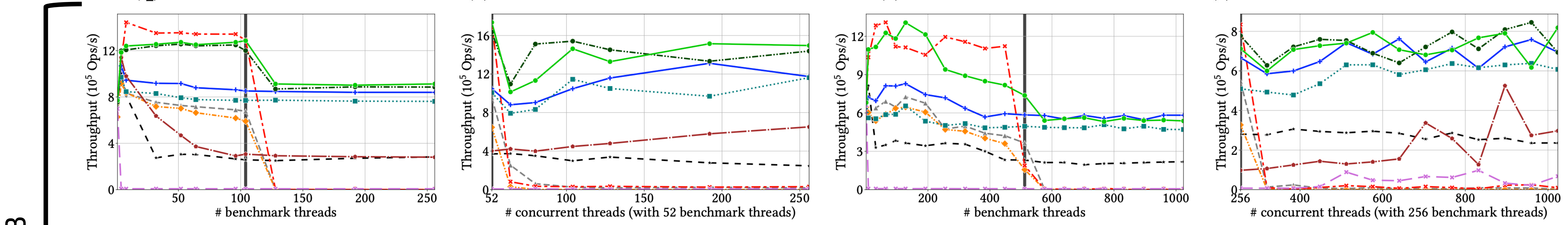


(m) Raytrace on Intel. **(n)** Raytrace on Intel w/ conc. **(o)** Raytrace on AMD. **(p)** Raytrace on AMD w/ conc.

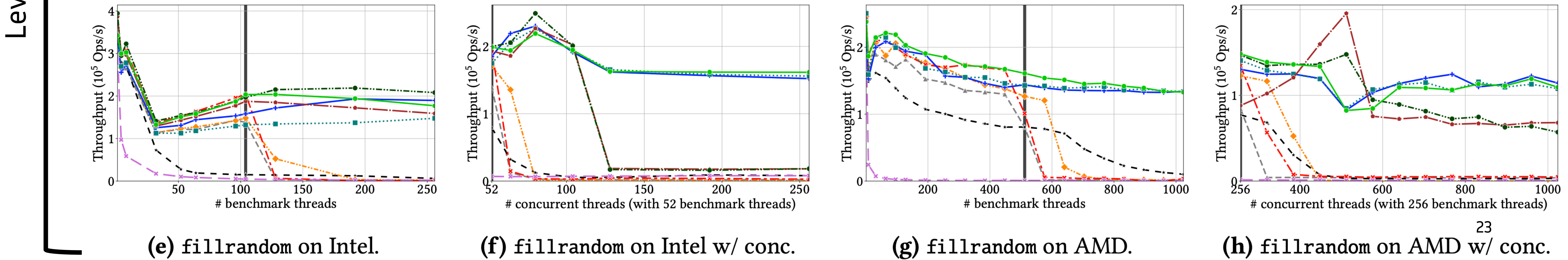
Higher is better



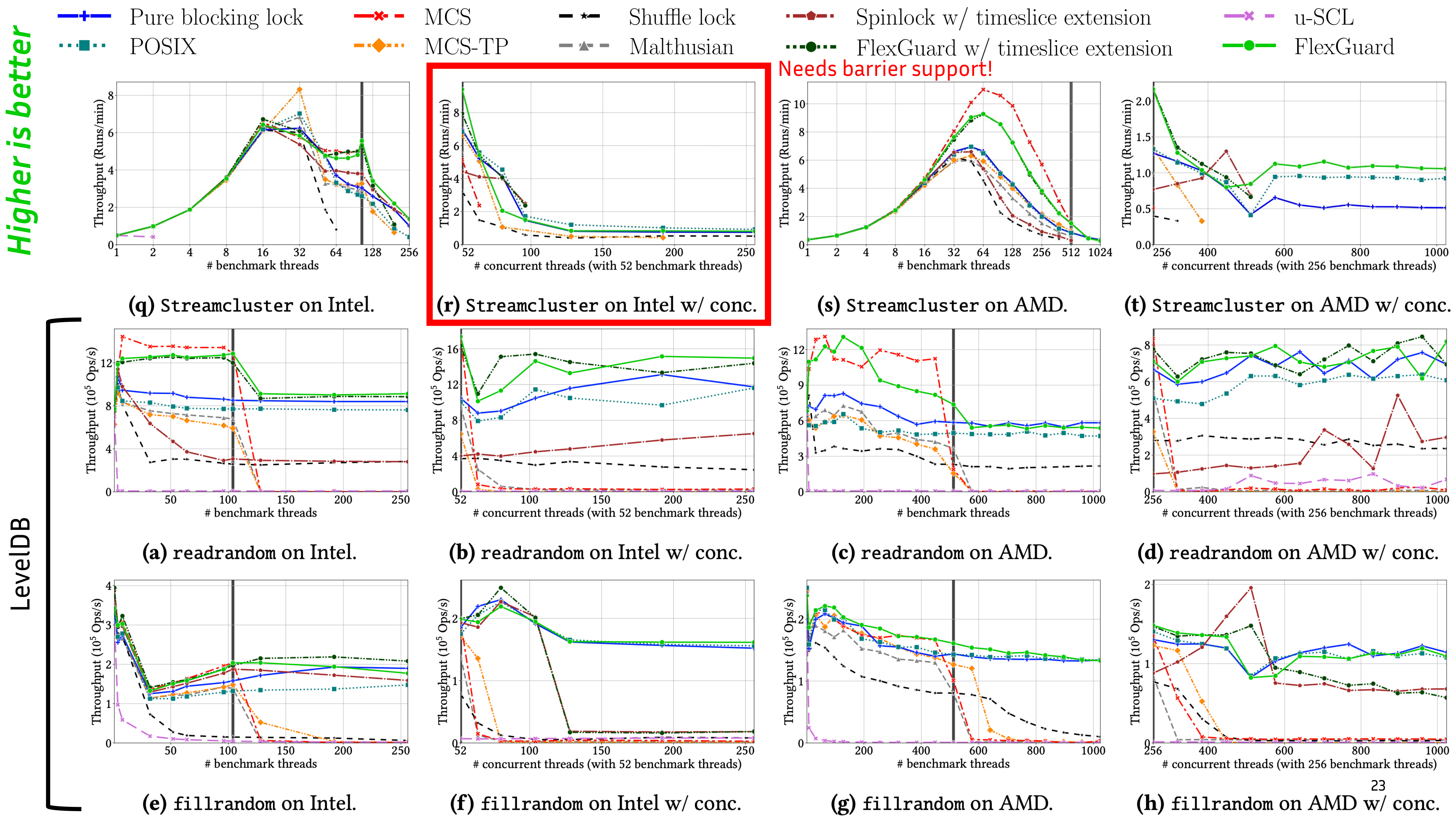
(q) `Streamcluster` on Intel. **(r)** `Streamcluster` on Intel w/ conc. **(s)** `Streamcluster` on AMD. **(t)** `Streamcluster` on AMD w/ conc.



(a) `readrandom` on Intel. **(b)** `readrandom` on Intel w/ conc. **(c)** `readrandom` on AMD. **(d)** `readrandom` on AMD w/ conc.



(e) `fillrandom` on Intel. **(f)** `fillrandom` on Intel w/ conc. **(g)** `fillrandom` on AMD. **(h)** `fillrandom` on AMD w/ conc.



(q) `Streamcluster` on Intel.

(r) `Streamcluster` on Intel w/ conc.

(s) `Streamcluster` on AMD.

(t) `Streamcluster` on AMD w/ conc.

(a) `readrandom` on Intel.

(b) `readrandom` on Intel w/ conc.

(c) `readrandom` on AMD.

(d) `readrandom` on AMD w/ conc.

(e) `fillrandom` on Intel.

(f) `fillrandom` on Intel w/ conc.

(g) `fillrandom` on AMD.

(h) `fillrandom` on AMD w/ conc.

Conclusion

- **FlexGuard = best of both worlds between spinning and blocking**

```
# Single-variable lock states
# LOCKED_WITH_BLOCKED_WAITERS = at least one thread is blocking,
# t… all futex_wake when releasing the lock
UNLOCKED = 0, LOCKED = 1, LOCKED_WITH_BLOCKED_WAITERS = 2
# CS preemption counter updated by the eBPF Preemption Monitor
num_preempted_cs = 0

class Lock:
  val = UNLOCKED, queue = None # Single-variable lock, MCS tail
class QNode:
  next = None, waiting = False

def mcs_exit(lock: Lock, qnode: QNode):
  if qnode.next is None:
    if CAS(&lock.queue, qnode, None) == qnode:
      return
    while qnode.next is None:
      PAUSE()
  qnode.next.waiting = False

def flexguard_unlock(lock: Lock, qnode: QNode):
  qnode.cs_counter -= 1
label at_unlock
  if XCHG(&lock.val, UNLOCKED) == LOCKED_WITH_BLOCKED_WAITERS:
    futex_wake(&lock.val, 1) # Wake one of the waiting threads

def flexguard_lock(lock: Lock, qnode: QNode):
label at_fastpath # Try to steal the single-variable lock if free
  if lock.val == UNLOCKED and CAS(&lock.val, UNLOCKED, LOCKED):
    qnode.cs_counter += 1
    return
  # There are waiters in the queue, enter the slow path
  flexguard_slow_path(lock, qnode)
```

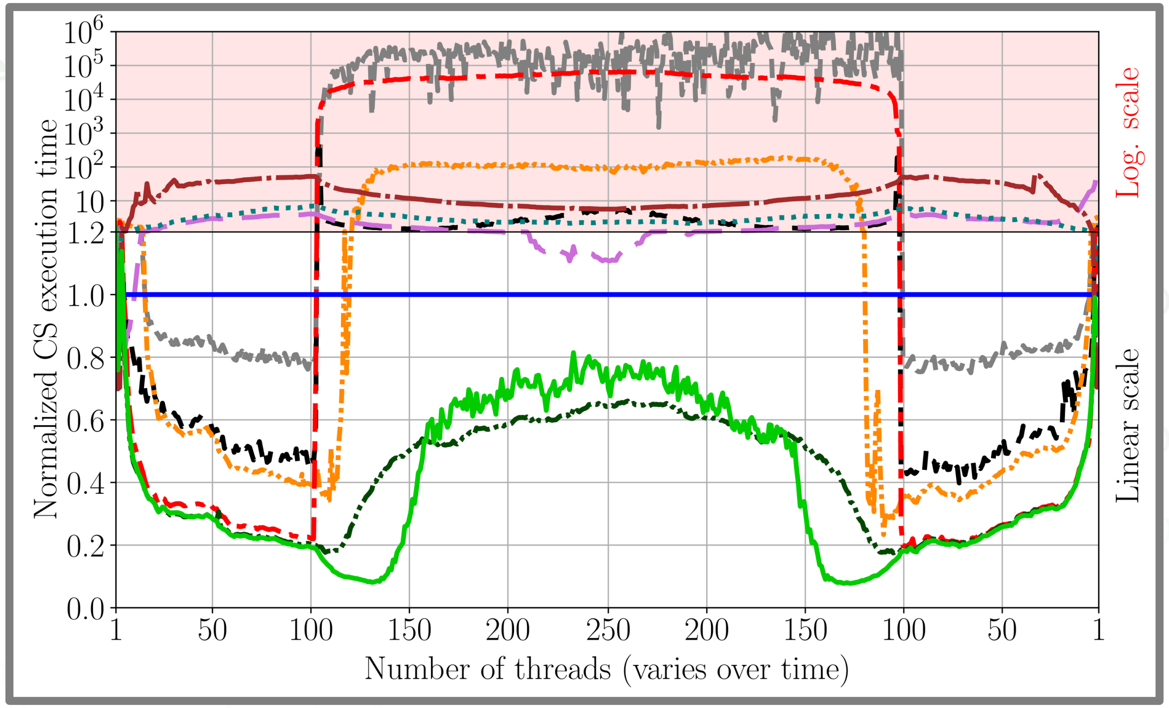
```
def flexguard_slow_path(lock, qnode):
  enqueued = False
  if num_preempted_cs == 0: # If spinning, begin Phase 1
    enqueued = True
    qnode.next = None
    qnode.waiting = True
label at_xchg
    pred = XCHG(&lock.queue, qnode)
    if pred is not None:
      pred.next = qnode
      while qnode.waiting and num_preempted_cs == 0:
        PAUSE()
label at_phase2 # Begin Phase 2
  state = CAS(&lock.val, UNLOCKED, LOCKED)
  while state != UNLOCKED:
    if num_preempted_cs == 0: # Busy-waiting mode
      PAUSE()
      state = CAS(&lock.val, UNLOCKED, LOCKED)
    else: # Blocking mode
      if enqueued:
        mcs_exit(lock, qnode)
        enqueued = False
      if state != LOCKED_WITH_BLOCKED_WAITERS:
        state = XCHG(&lock.val, LOCKED_WITH_BLOCKED_WAITERS)
      if state != UNLOCKED:
        futex_wait(&lock.val, LOCKED_WITH_BLOCKED_WAITERS)
        state = XCHG(&lock.val, LOCKED_WITH_BLOCKED_WAITERS)
        if state != UNLOCKED and num_preempted_cs == 0:
          # Back to spin mode, restart slow path (using MCS)
          return flexguard_slow_path(lock, qnode)
  if enqueued: # Exit the queue if still enqueued
    mcs_exit(lock, qnode)
  qnode.cs_counter += 1
```

Conclusion

- FlexGuard = best of both worlds between spinning and blocking
- Preemption Monitor: accurate critical section preemption detection
  - First non-heuristic approach, thanks to eBPF!

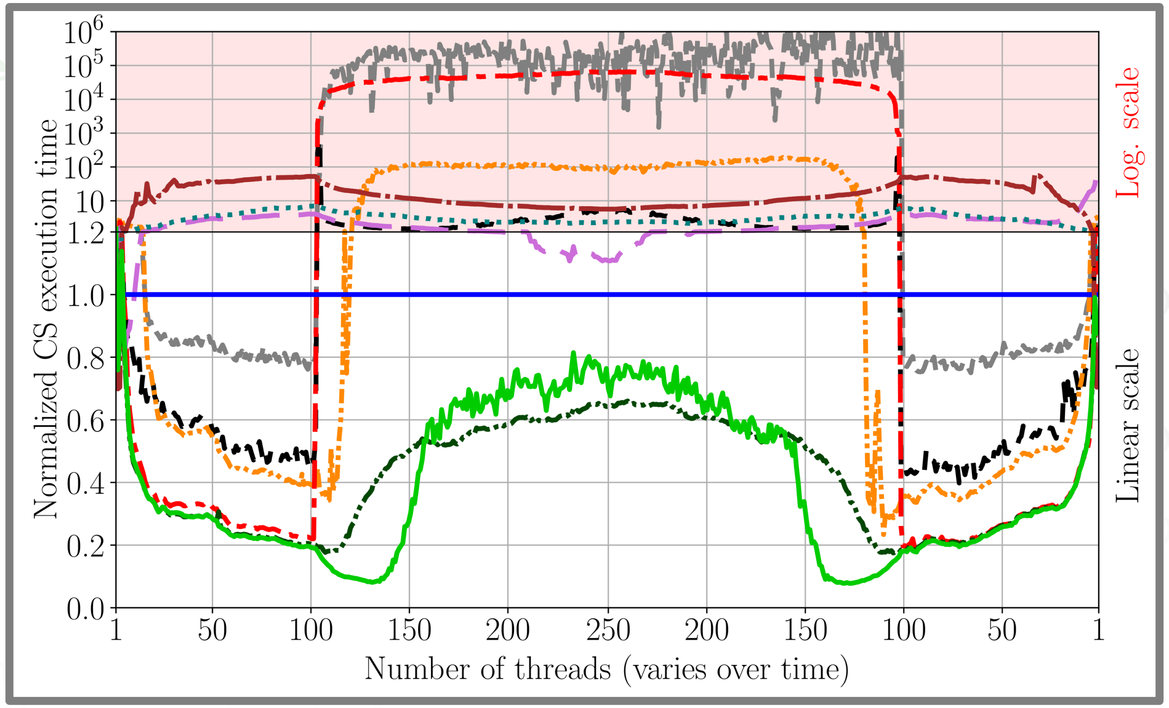
Conclusion

- **FlexGuard** = **best of both worlds between spinning and blocking**
  - **Preemption Monitor:** accurate critical section preemption detection
    - First non-heuristic approach, thanks to eBPF!
  - **FlexGuard's lock algorithm:** outperforms blocking locks when oversubscribed
    - Good amount of spinning waiters

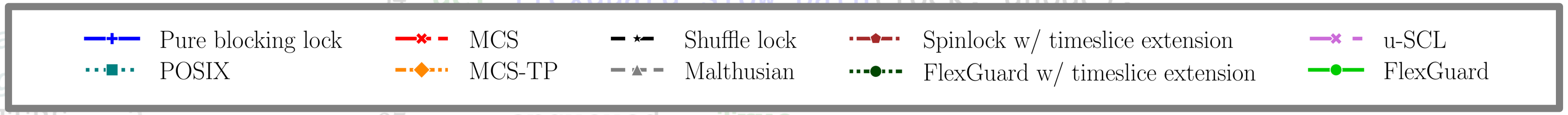


Conclusion

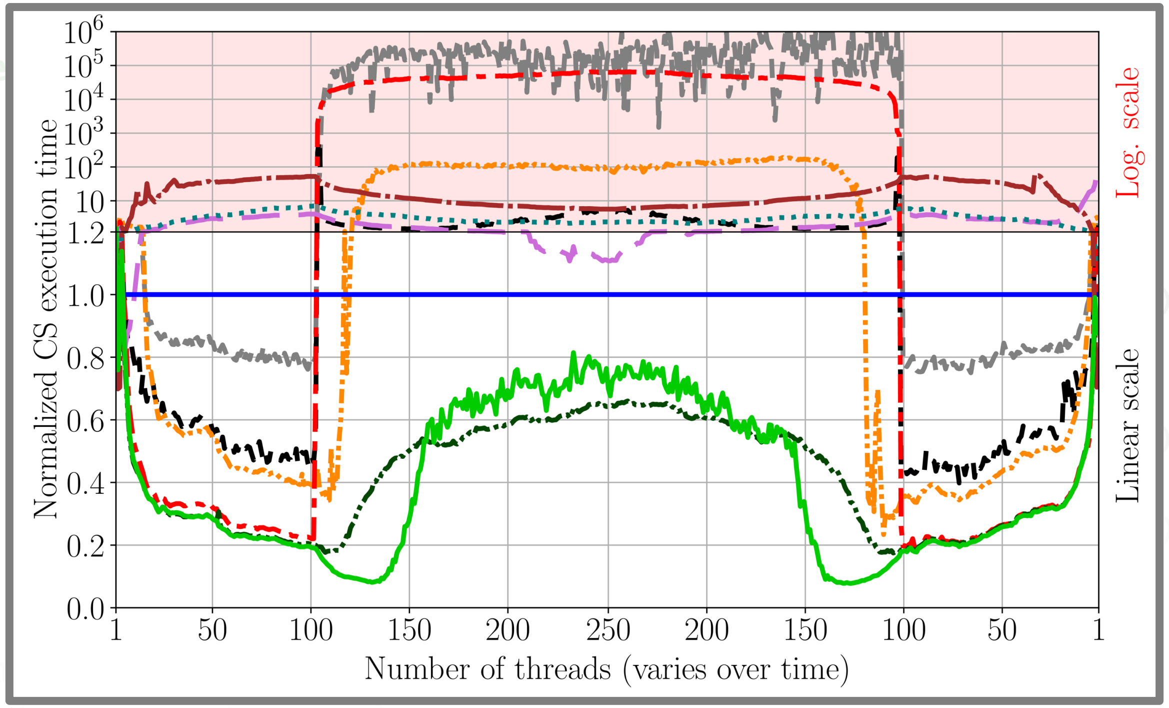
- **FlexGuard** = **best of both worlds between spinning and blocking**
  - **Preemption Monitor:** accurate critical section preemption detection
    - First non-heuristic approach, thanks to eBPF!
  - **FlexGuard's lock algorithm:** outperforms blocking locks when oversubscribed
    - Good amount of spinning waiters
  - Recently proposed Linux timeslice extension: complementary



Conclusion



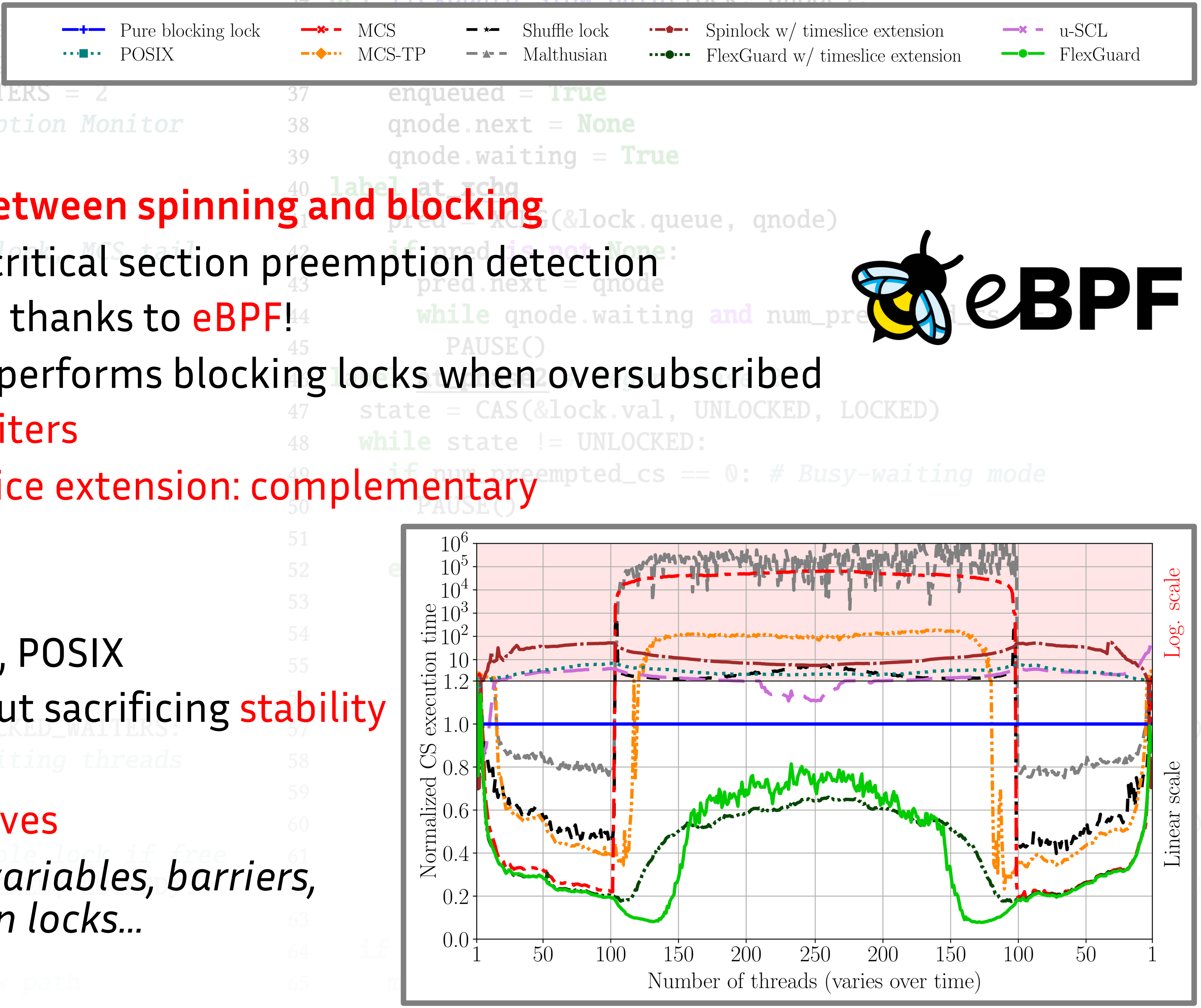
- **FlexGuard = best of both worlds between spinning and blocking**
  - **Preemption Monitor:** accurate critical section preemption detection
    - First non-heuristic approach, thanks to eBPF!
  - **FlexGuard's lock algorithm:** outperforms blocking locks when oversubscribed
    - Good amount of spinning waiters
  - Recently proposed Linux timeslice extension: complementary
- **Where could FlexGuard be used?**
  - In standard libraries such as e.g., POSIX
    - Spinlock performance, without sacrificing stability
      - *No performance collapse!*



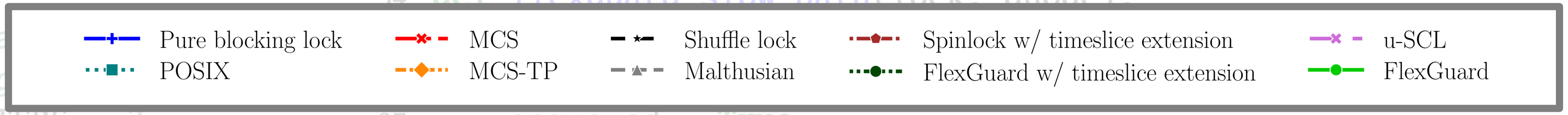
Conclusion

- **FlexGuard = best of both worlds between spinning and blocking**
  - **Preemption Monitor:** accurate critical section preemption detection
    - First non-heuristic approach, thanks to eBPF!
  - **FlexGuard's lock algorithm:** outperforms blocking locks when oversubscribed
    - Good amount of spinning waiters
  - Recently proposed Linux timeslice extension: complementary

- **Where could FlexGuard be used?**
  - In standard libraries such as e.g., POSIX
    - Spinlock performance, without sacrificing stability
      - *No performance collapse!*
  - In more synchronization primitives
    - *Read-write locks, condition variables, barriers, optimistic locking, delegation locks…*



Conclusion



- **FlexGuard = best of both worlds between spinning and blocking**
  - **Preemption Monitor:** accurate critical section preemption detection
    - First non-heuristic approach, thanks to eBPF!
  - **FlexGuard's lock algorithm:** outperforms blocking locks when oversubscribed
    - Good amount of spinning waiters
  - Recently proposed Linux timeslice extension: complementary
- **Where could FlexGuard be used?**
  - In standard libraries such as e.g., POSIX
    - Spinlock performance, without sacrificing stability
      - *No performance collapse!*
  - In more synchronization primitives
    - *Read-write locks, condition variables, barriers, optimistic locking, delegation locks…*
  - In the virtualized case (vCPU preemptions)