

Outline

- CPU caches
- Cache coherence
- Placement of data
- **Hardware synchronization instructions**
- Correctness: Memory model & compiler
- Performance: Programming techniques

The Programmer's Toolbox: Hardware Synchronization Instructions

- Depends on the processor;
- CAS generally provided;
- Test-and-Set and Fetch-and-Increment etc. may or may not be provided;
- x86:
 - Atomic exchange, increment, decrement provided
 - Memory barrier also available
- New Intels (Haswell) provide transactional memory

Example: Atomic Ops in GCC

```
type __sync_fetch_and_OP(type *ptr, type value);  
type __sync_OP_and_fetch(type *ptr, type value);  
// OP in {add,sub,or,and,xor,nand}
```

```
type __sync_val_compare_and_swap(type *ptr, type  
oldval, type newval);
```

```
bool __sync_bool_compare_and_swap(type *ptr, type  
oldval, type newval);
```

```
__sync_synchronize(); // memory barrier
```

Intel's Transactional Synchronization Extensions (TSX)

I. Hardware lock elision (HLE)

- Instruction prefixes:

XACQUIRE

XRELEASE

Ex:

```
__hle_{acquire,release}_compare_exchange_n{1,2,4,8}
```

- Try to execute critical sections without acquiring/releasing the lock.
- If conflict detected, abort and acquire the lock before re-doing the work

Intel's Transactional Synchronization Extensions (TSX)

2. Restricted Transactional Memory (RTM)

```
_xbegin();  
_xabort();  
_xtest();  
_xend();
```

Not starvation free!

Transactions can be aborted for a variety of reasons.

Should have a non-transactional back-up.

Limited transaction size.

Intel's Transactional Synchronization Extensions (TSX)

2. Restricted Transactional Memory (RTM)

Example:

```
if (_xbegin() == _XBEGIN_STARTED){
    counter = counter + 1;
    _xend();
} else {
    __sync_fetch_and_add(&counter,1);
}
```

Outline

- CPU caches
- Cache coherence
- Placement of data
- Hardware synchronization instructions
- **Correctness: Memory model & compiler**
- Performance: Programming techniques

Concurrent Algorithm Correctness

- Designing correct concurrent algorithms:
 1. Theoretical part
 - 2. Practical part**
- The processor and compiler optimize assuming no concurrency!

The Memory Consistency Model

```
//A, B shared variables, initially 0;  
//r1, r2 – local variables;
```

P1

```
A = 1;  
r1 = B;
```

P2

```
B = 1;  
r2 = A;
```

What values can r1 and r2 take?

(assume x86 processor)

Answer:

(0,1), (1,0), (1,1) and (0,0)

The Memory Consistency Model

- The order in which memory instructions appear to execute
 - What would the programmer like to see?
- **Sequential consistency**
 - All operations executed in some sequential order;
 - Memory operations of each thread in program order;
 - Intuitive, but limits performance;


The Memory Consistency Model

How can the processor reorder instructions to different memory addresses?

x86 (Intel,AMD):TSO variant

- Reads not reordered w.r.t. reads
- Writes not reordered w.r.t writes
- Writes not reordered w.r.t. reads
- Reads may be reordered w.r.t. writes to different memory addresses

```
//A,B,C
//globals
...
int x,y,z;
x = A;
y = B;
B = 3;
A = 2;
y = A;
C = 4;
z = B;
...
```



The Memory Consistency Model

- Single thread – reorderings transparent;
- Avoid reorderings: memory barriers
 - x86 – implicit in atomic ops;
 - “volatile” in Java;
 - Expensive - use only when really necessary;
- Different processors – different memory consistency models
 - e.g., ARM – relaxed memory model (anything goes!);
 - VMs (e.g. JVM, CLR) have their own memory models;

Beware of the Compiler

```
void lock(int * some_lock) {
    while (CAS(some_lock,0,1) != 0) {}
    asm volatile("" ::: "memory"); //compiler barrier
}
void unlock(int * some_lock) {
    asm volatile("" ::: "memory"); //compiler barrier
    *some_lock = 0;
}
```

```
volatile int the_lock=0;
```

```
lock(&the_lock);
```

```
...
```

```
unlock(&the_lock);
```

**C "volatile" !=
Java "volatile"**

- The compiler can:
 - reorder
 - remove instructions
 - not write values to memory

Outline

- CPU caches
- Cache coherence
- Placement of data
- Hardware synchronization instructions
- Correctness: Memory model & compiler
- **Performance: Programming techniques**

Concurrent Programming Techniques

- What techniques can we use to speed up our concurrent application?
- Main idea: minimize contention on cache lines
- Use case: Locks
 - acquire()
 - release()

Let's start with a simple lock...

Test-and-Set Lock

```
typedef volatile uint lock_t;

void acquire(lock_t * some_lock) {
    while (TAS(some_lock) != 0) {}
    asm volatile("" ::: "memory");
}

void release(lock_t * some_lock) {
    asm volatile("" ::: "memory");
    *some_lock = 0;
}
```


How good is this lock?

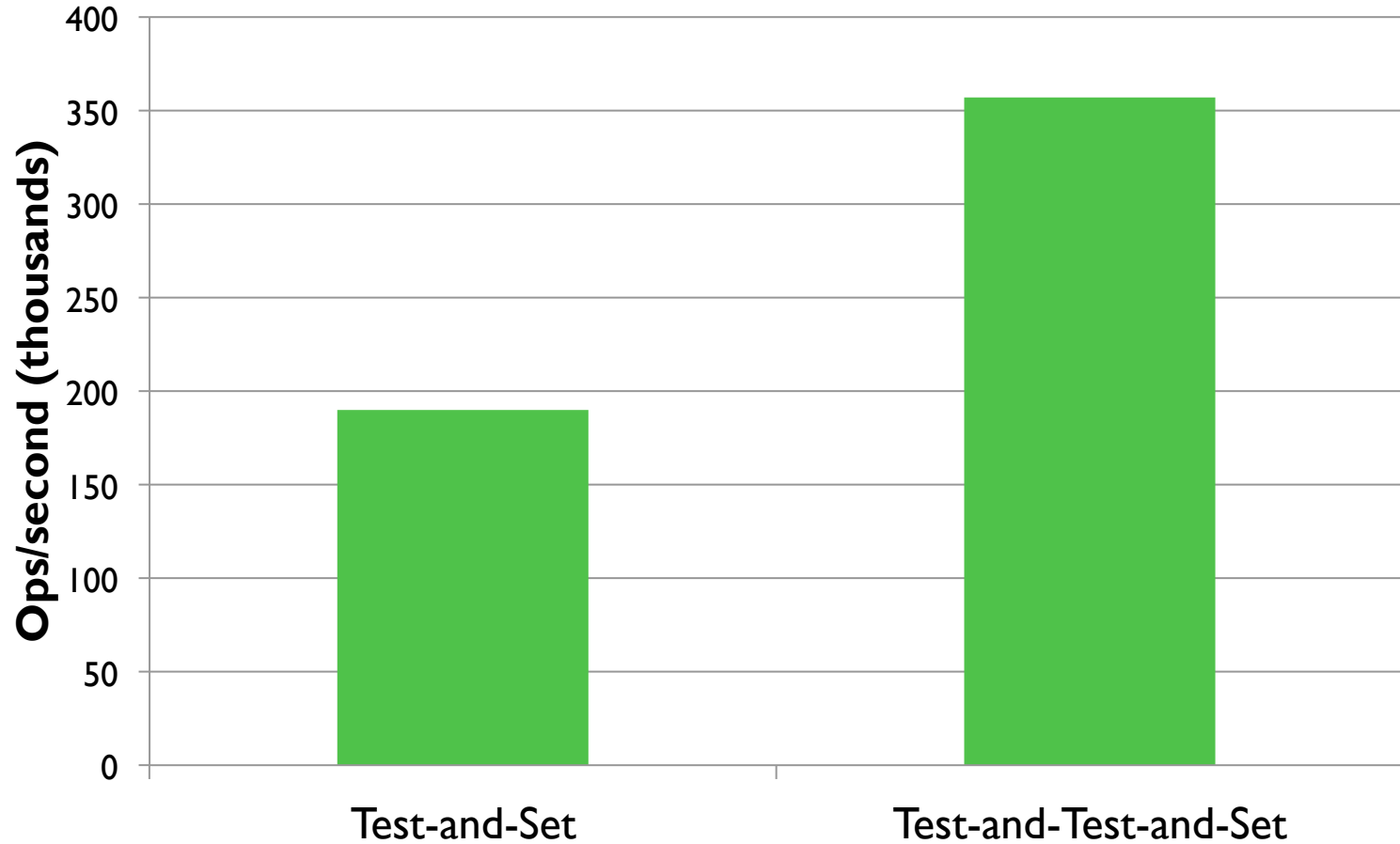
- A simple benchmark
- Have 48 threads continuously acquire a lock, update some shared data, and unlock
- Measure how many operations we can do in a second
- Test-and-Set lock: 190K operations/second

How can we improve things?

Avoid cache-line ping-pong: Test-and-Test-and-Set Lock

```
void acquire(lock_t * some_lock) {
    while(1) {
        while (*some_lock != 0) {}
        if (TAS(some_lock) == 0) {
            return;
        }
    }
    asm volatile("" ::: "memory");
}
void release(lock_t * some_lock) {
    asm volatile("" ::: "memory");
    *some_lock = 0;
}
```

Performance comparison



But we can do even better

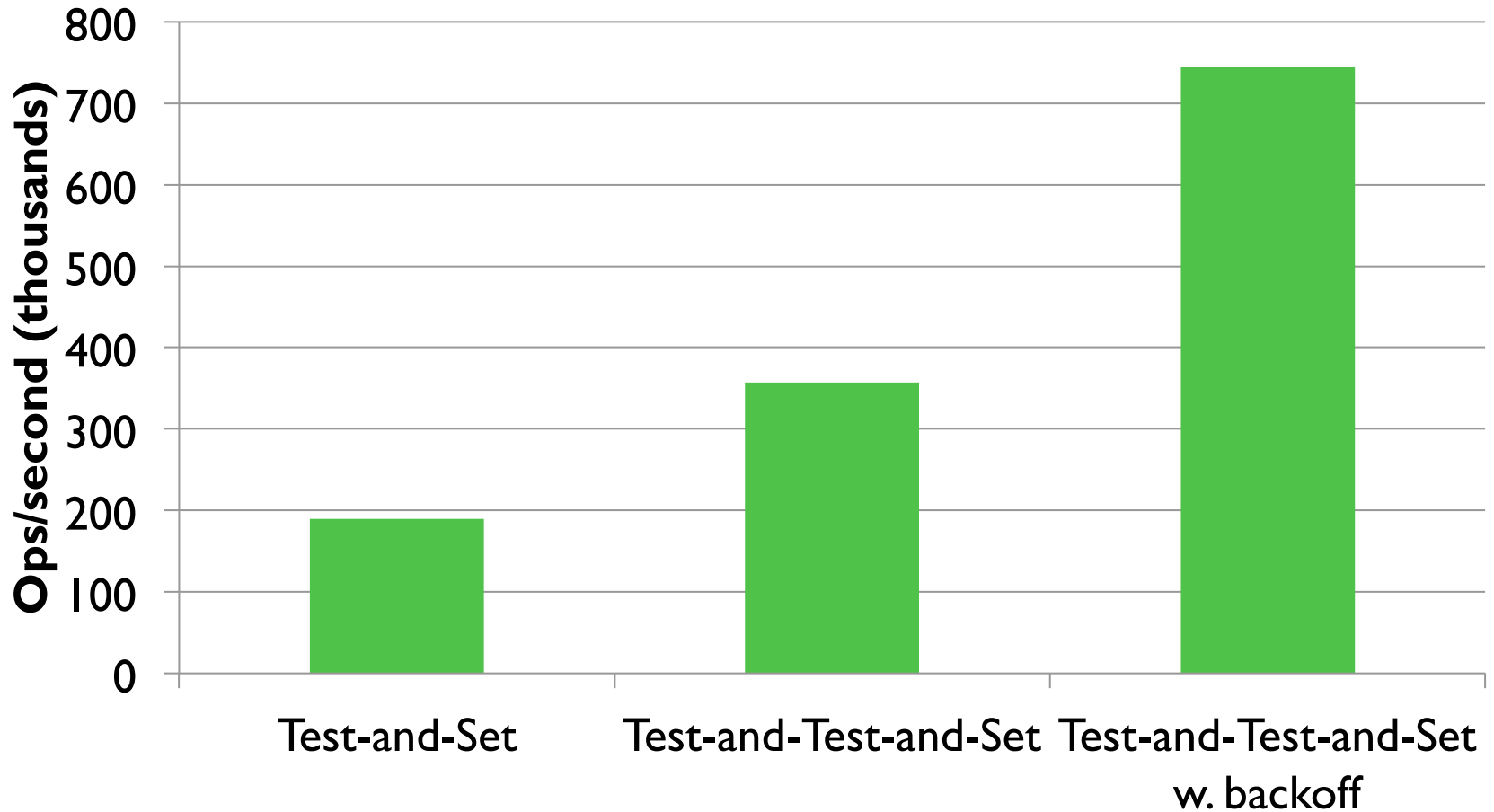
Avoid thundering herd:

Test-and-Test-and-Set with Back-off

```
void acquire(lock_t * some_lock) {
    uint backoff = INITIAL_BACKOFF;
    while(1) {
        while (*some_lock != 0) {}
        if (TAS(some_lock) == 0) {
            return;
        } else {
            lock_sleep(backoff);
            backoff=min(backoff*2,MAXIMUM_BACKOFF);
        }
    }
    asm volatile("" ::: "memory");
}

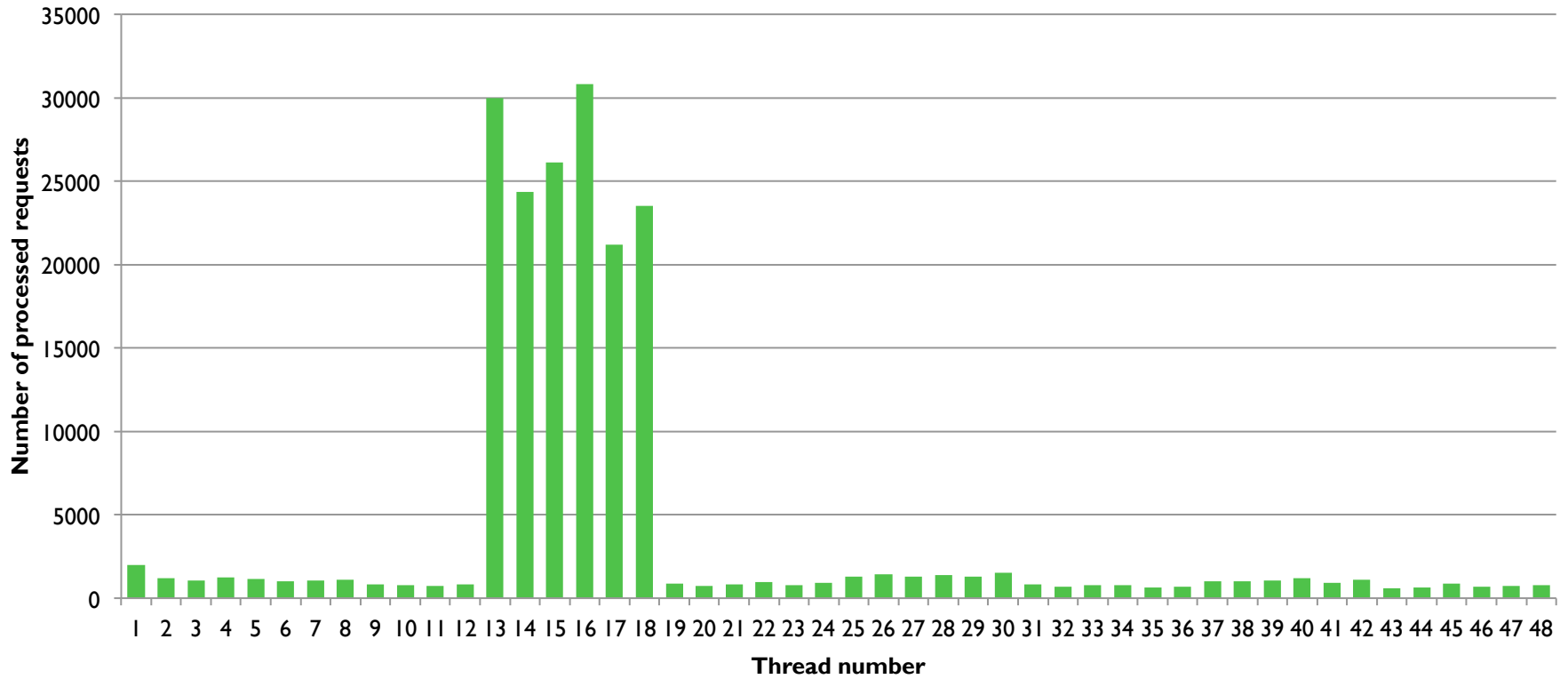
void release(lock_t * some_lock) {
    asm volatile("" ::: "memory");
    *some_lock = 0;
}
```

Performance comparison



Are these locks fair?

Processed requests per thread, Test-and-Set lock



What if we want fairness?

Use a FIFO mechanism:
Ticket Locks

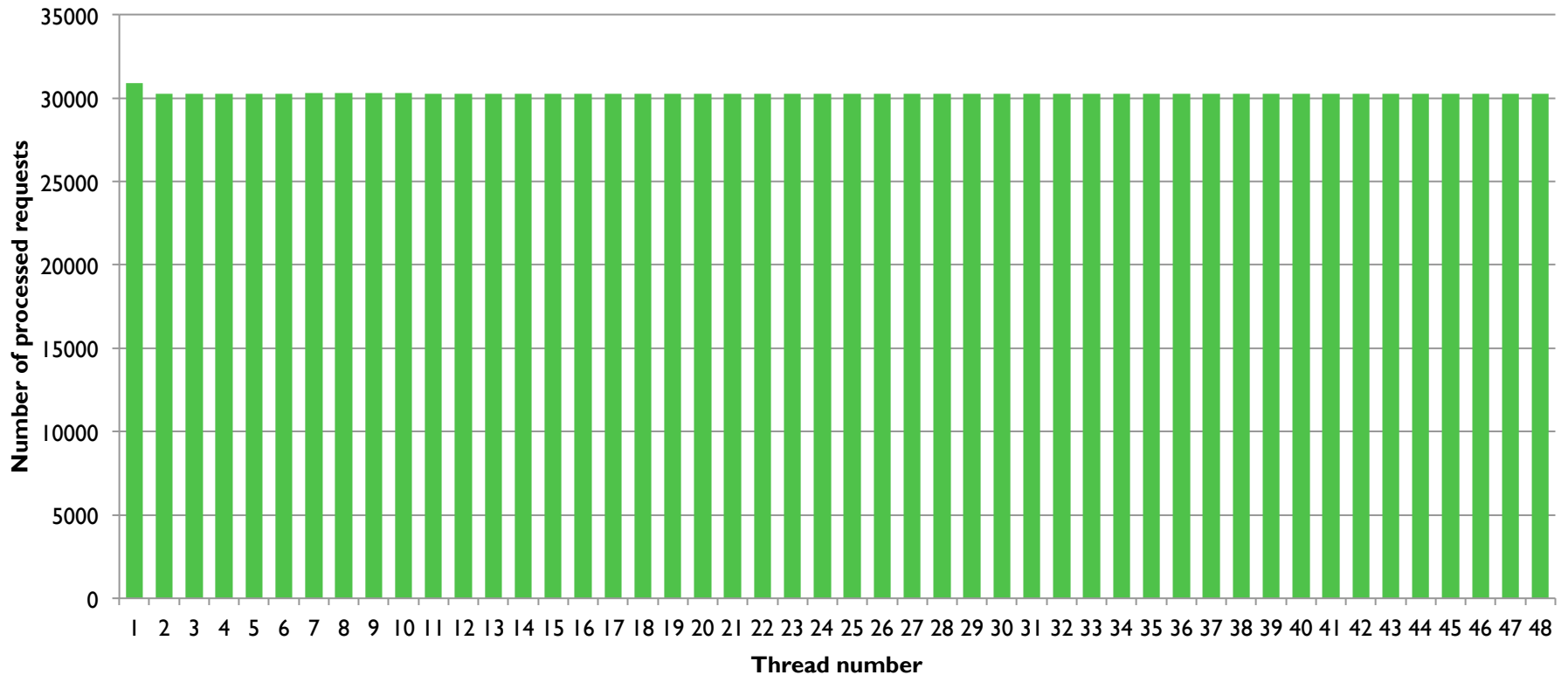
```
typedef ticket_lock_t {
    volatile uint head;
    volatile uint tail;
} ticket_lock_t;

void acquire(ticket_lock_t * a_lock) {
    uint my_ticket = fetch_and_inc(&(a_lock->tail));
    while (a_lock->head != my_ticket) {}
    asm volatile("" ::: "memory");
}

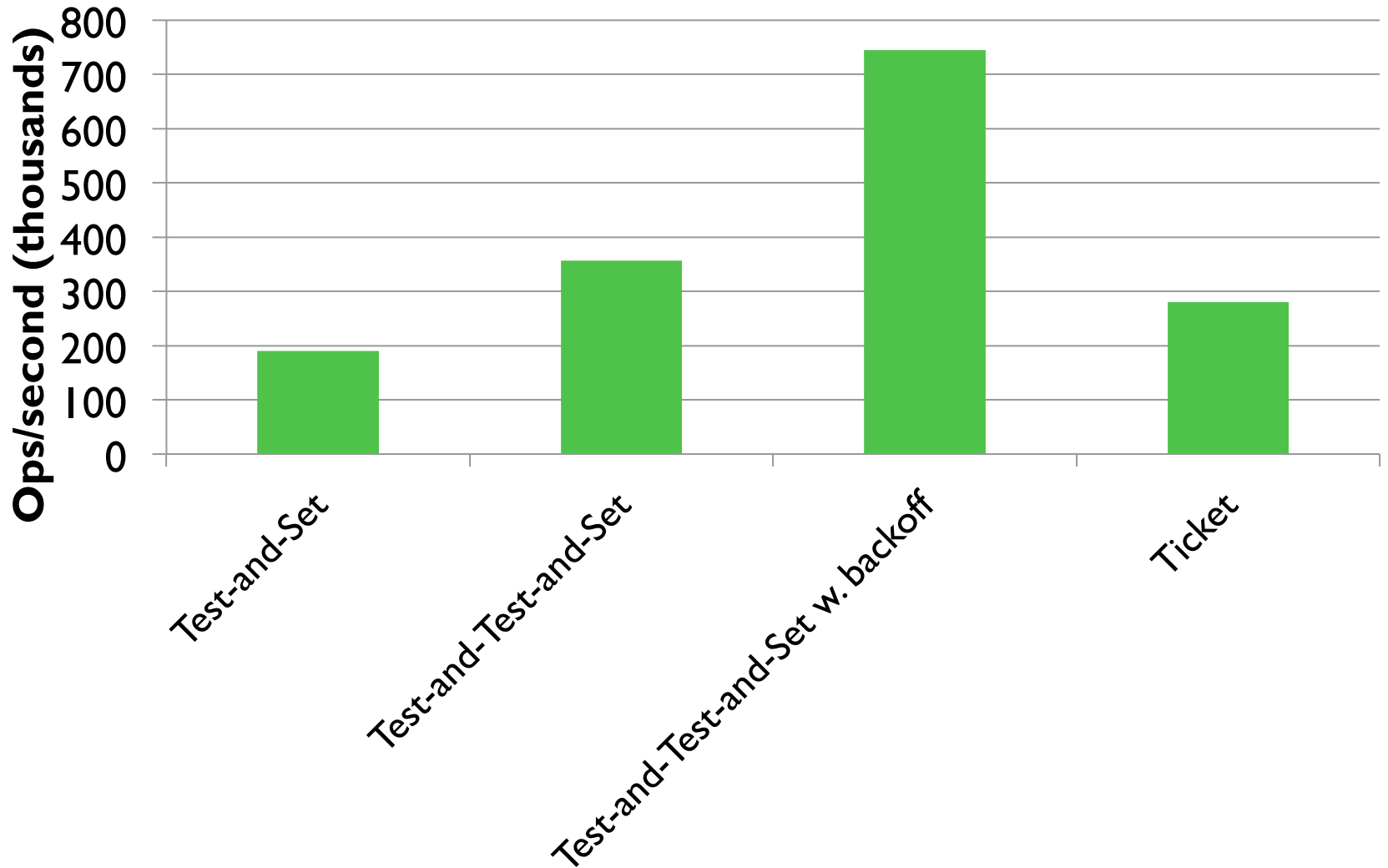
void release(ticket_lock_t * a_lock) {
    asm volatile("" ::: "memory");
    a_lock->head++;
}
```

What if we want fairness?

Processed requests per thread, Ticket Locks



Performance comparison



Can we back-off here as well?

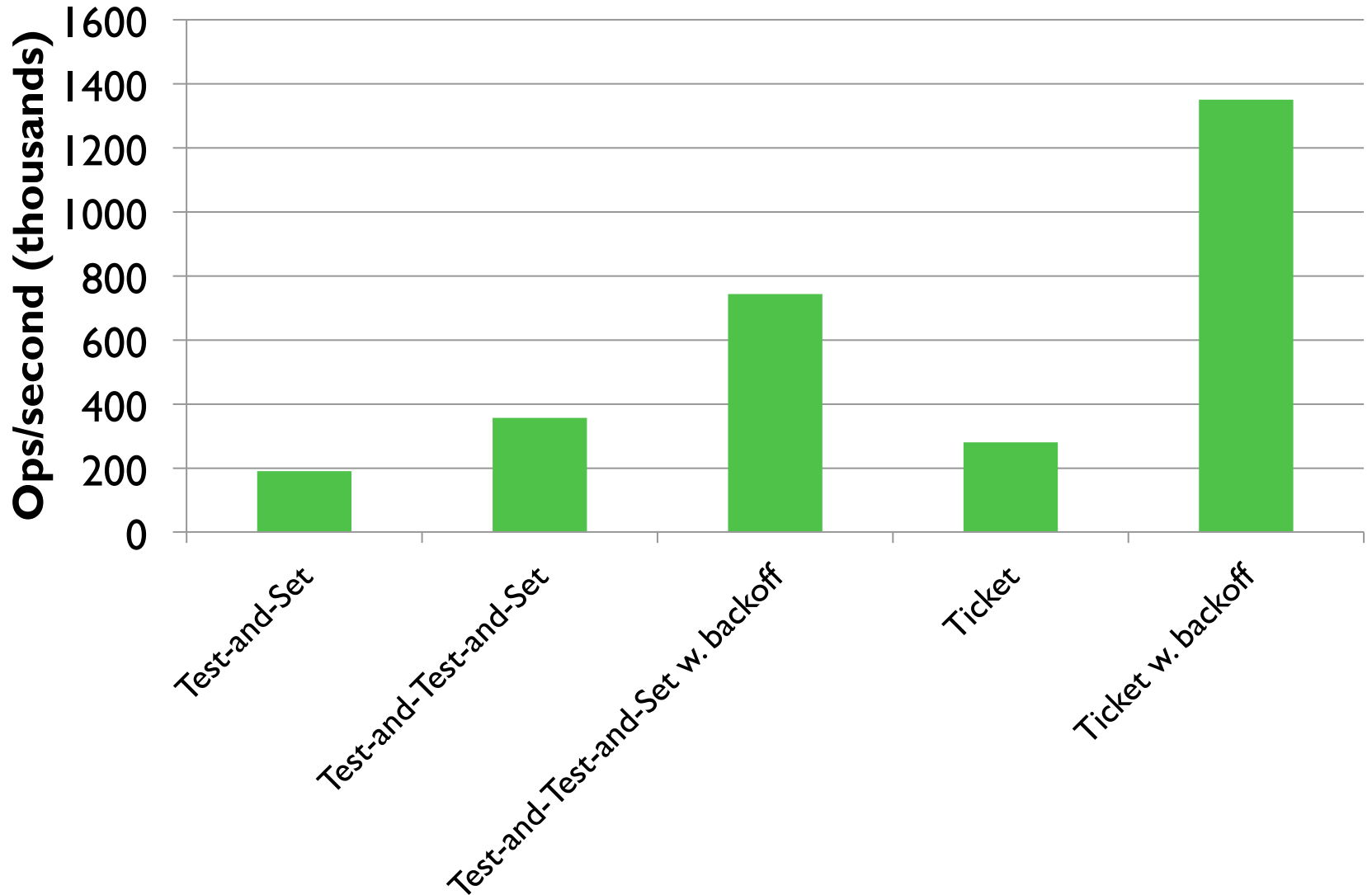
Yes, we can:

Proportional back-off

```
void acquire(ticket_lock_t * a_lock) {
    uint my_ticket = fetch_and_inc(&(a_lock->tail));
    uint distance, current_ticket;
    while (1) {
        current_ticket = a_lock->head;
        if (current_ticket == my_ticket) break;
        distance = my_ticket - current_ticket;
        if (distance > 1)
            lock_sleep(distance * BASE_SLEEP);
    }
    asm volatile("" ::: "memory");
}

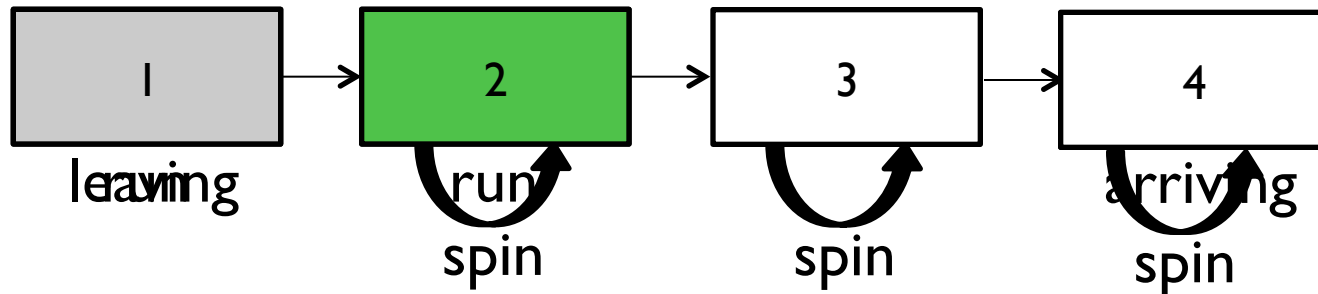
void release(ticket_lock_t * a_lock) {
    asm volatile("" ::: "memory");
    a_lock->head++;
}
```

Performance comparison



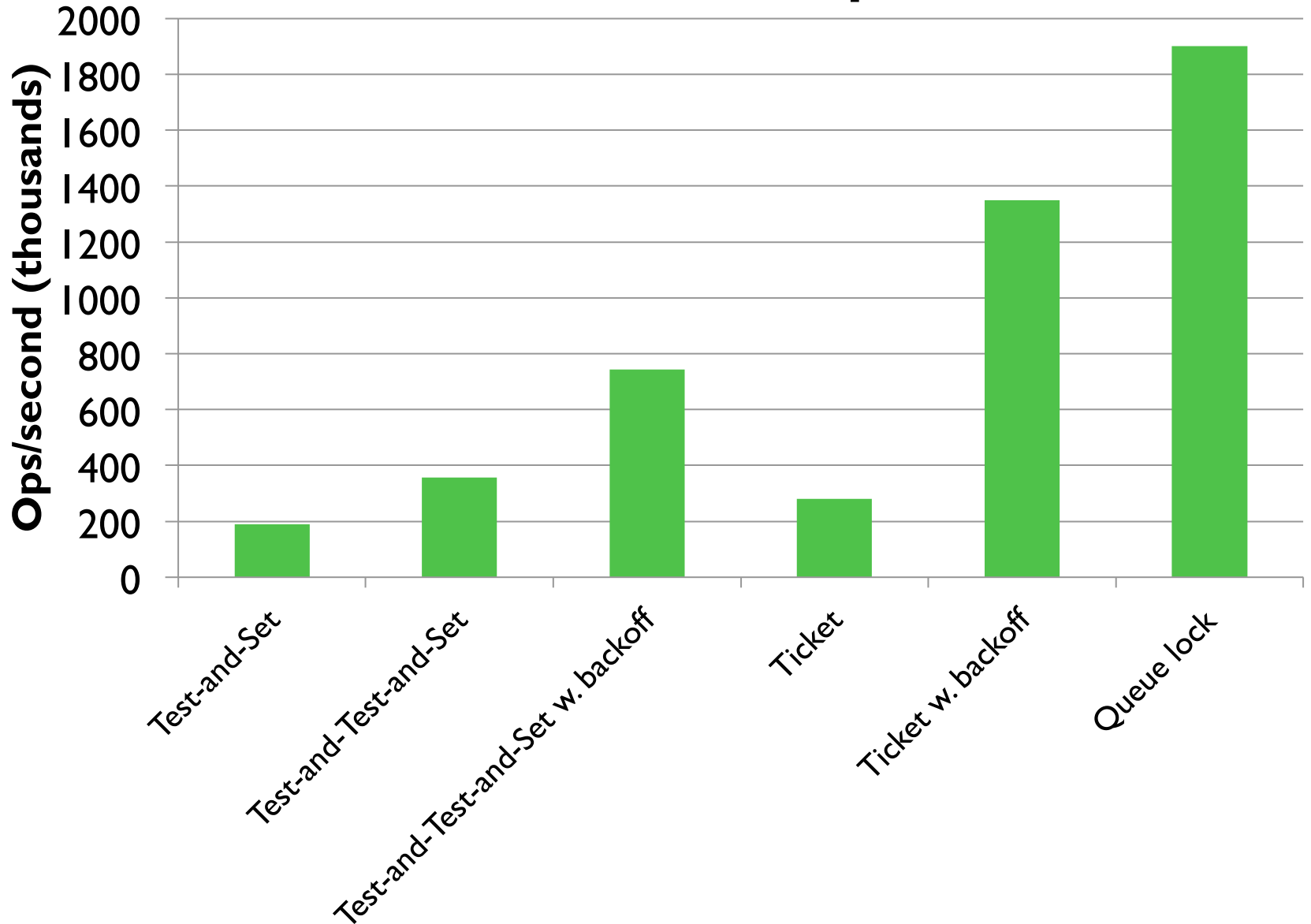
Still, everyone is spinning on the same variable....

Use a different address for each thread:
Queue Locks



Use with moderation: storage overheads

Performance comparison



To sum it up

- Reading before trying to write
- Pausing when it's not our turn
- Ensuring fairness
- Accessing disjoint addresses (cache lines)

More than 10x performance gain!

Conclusion

- Concurrent algorithm design:
 - Theoretical design
 - Practical design (may be just as important)
- You need to know your hardware
 - For correctness
 - For performance

Reminder

Programming assignments due next Tuesday!

If you have any questions,
attend today's exercise session