MULTICORE PROGRAMMING

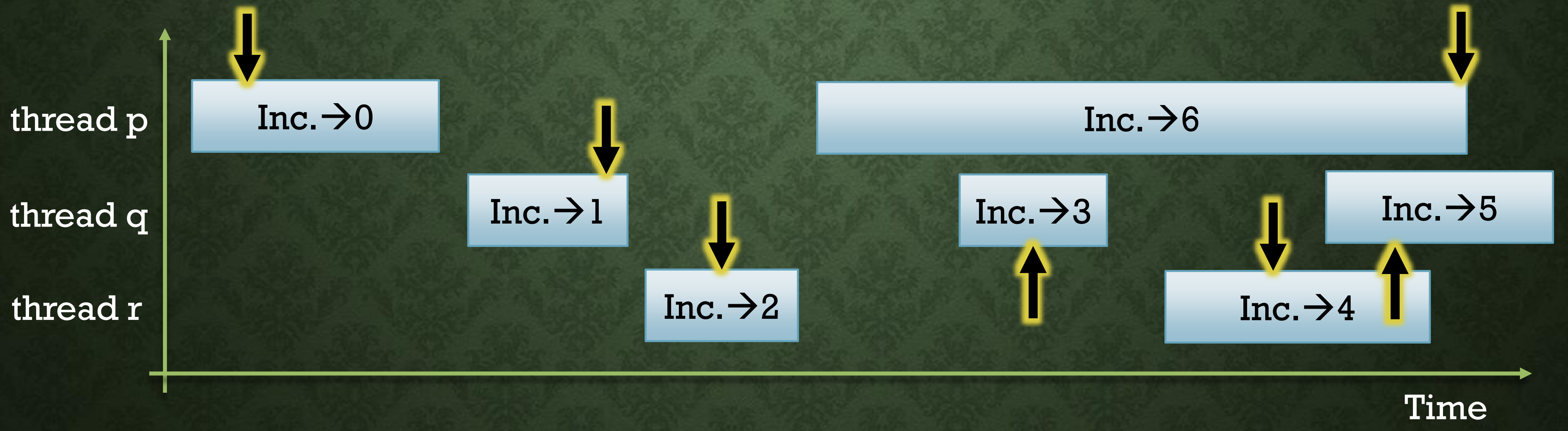
Implementing a counter: what could go wrong?

Lecture 2

Trevor Brown

RECALL: WHAT IT MEANS FOR AN **EXECUTION** TO BE **LINEARIZABLE**

- Must be possible to choose **linearization points** during each operation such that all operations return **the same values** that they would if they were executed **instantly** at their linearization points



- Non-linearizable if it is **not possible** to pick such linearization points

RECALL: WHAT IT MEANS FOR AN **OBJECT** TO BE **LINEARIZABLE**

- For **every possible execution E** of the object
  - E must be a linearizable **execution**

- An **object** is non-linearizable
  - if **any** execution of the object is non-linearizable
  - (i.e., the object can possibly behave “badly”)

RECALL: THIS EXECUTION OF THE **NAÏVE COUNTER**

thread p	thread q	thread r
Read(c) → 0		
	Read(c) → 0 Write(c, 1) Return 0	
		Read(c) → 1 Write(c, 2) Return 1
	Read(c) → 2 Write(c, 3) Return 2	
Write(c, 1)		

Time

Any changes between this Read and Write are **overwritten**!

IMPLEMENTING A LINEARIZABLE COUNTER

- Intuition: **increment** must **atomically** Read and Write ("at the same time")
  - Otherwise a thread can always Write a "stale" value (overwriting "fresh" values)
- Need stronger tools!
  - How about using a **lock?**

A LOCK / MUTEX

- **Guards** an object: allows only **one thread at a time** to access it
- Operations
  - Acquire / lock
    - Blocks until the calling thread has acquired the lock (then returns)
    - (Thread is then **<u>allowed</u>** to access the object)
  - Release / unlock
    - Releases the lock so other threads can acquire it
    - (Thread is **<u>no longer</u>** allowed to access the object)

```
Java: synchronized
C: pthread_spin_lock
C++: std::mutex
```

LOCK-BASED COUNTER

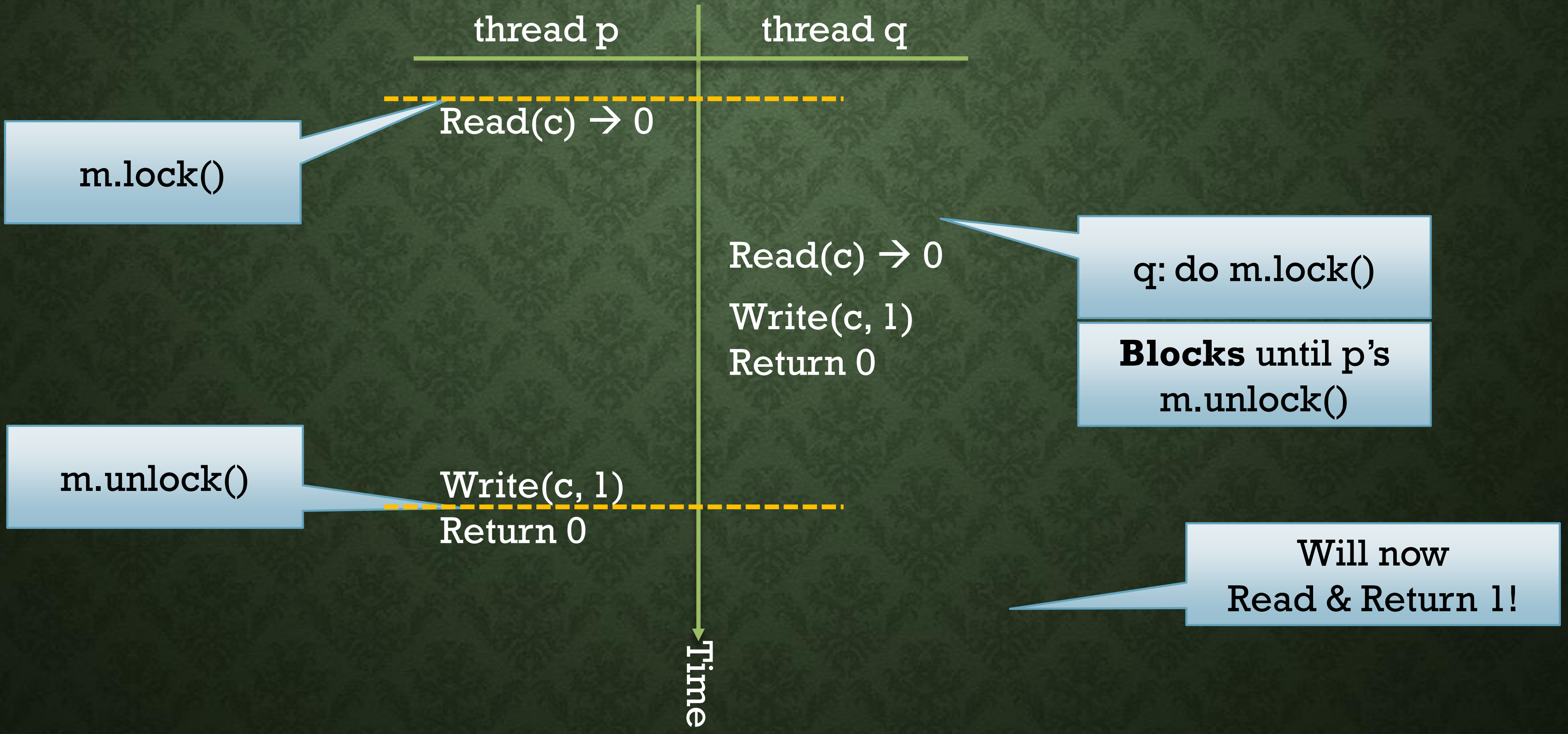
```cpp
class counter2 {
private:
    std::mutex m;
    int v;
public:
    counter2() { v = 0; }
    int increment(int threadID) {
        m.lock();
        auto ret = v++;
        m.unlock();
        return ret;
    }
    int get() {
        m.lock();
        auto ret = v;
        m.unlock();
        return ret;
    }
};
```

This is really a **read** followed by a **write**…

How does this help?

Both **read** and **write** are done while holding the lock

CAN THIS PROBLEM HAPPEN NOW?



WHERE DO WE LINEARIZE EACH OPERATION?

```
class counter2 {
private:
    std::mutex m;
    int v;
public:
    counter2() { v = 0; }
    int increment(int threadID) {
        m.lock();
        auto ret = v++;
        m.unlock();
        return ret;
    }
    int get() {
        m.lock();
        auto ret = v;
        m.unlock();
        return ret;
    }
};
```

Intuition behind why these linearization points work:
to all threads, operations appear to happen instantly at these lines

Linearize increment at the WRITE
(really, *any time* when the lock is held would work)

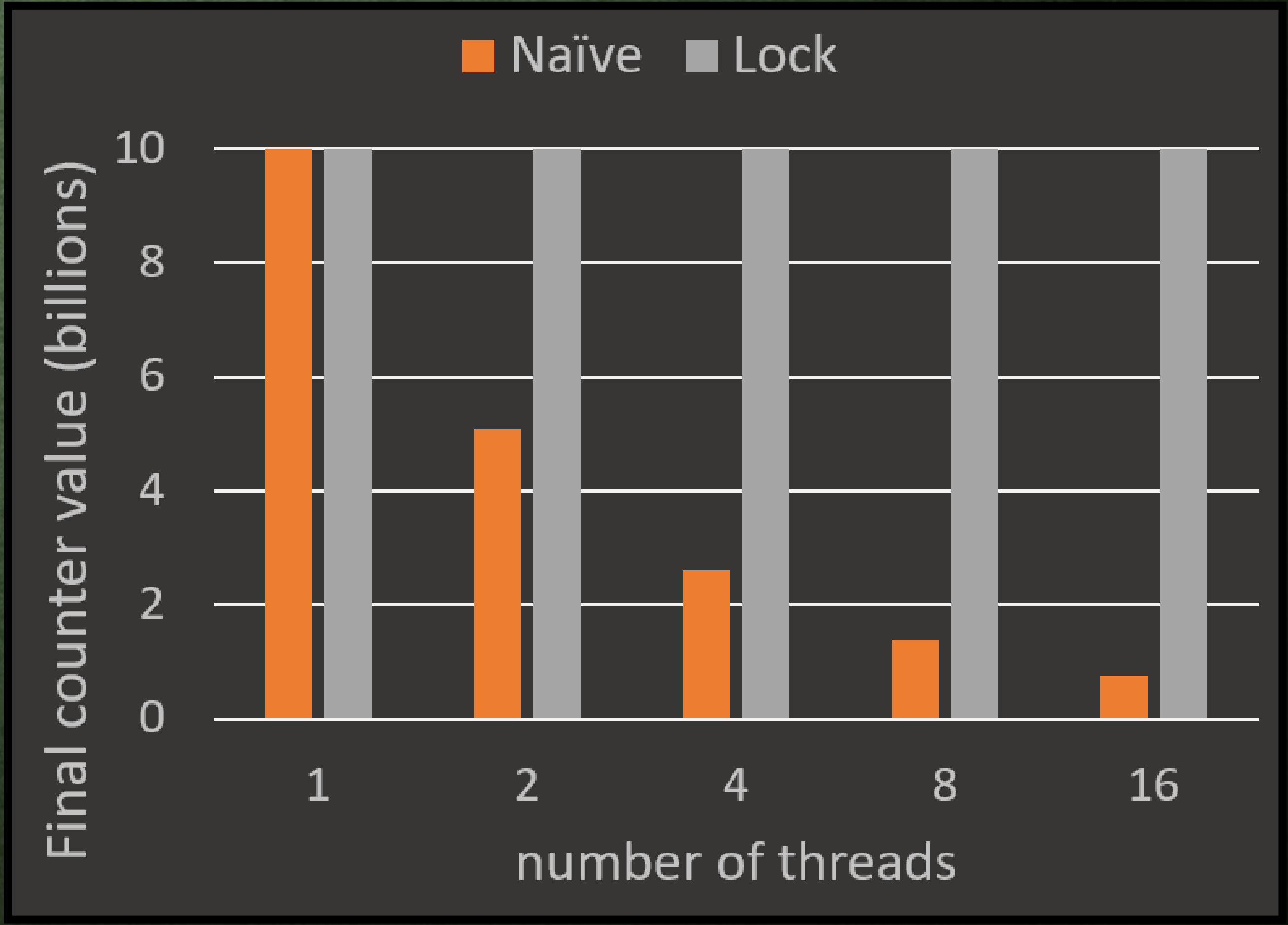
(So, anything that happens while the counter is locked is **effectively atomic**)

(… **because**, from the perspective of other threads, the counter cannot be accessed while it is locked)

Linearize get at the READ of v

OUTPUT AFTER ADDING A LOCK

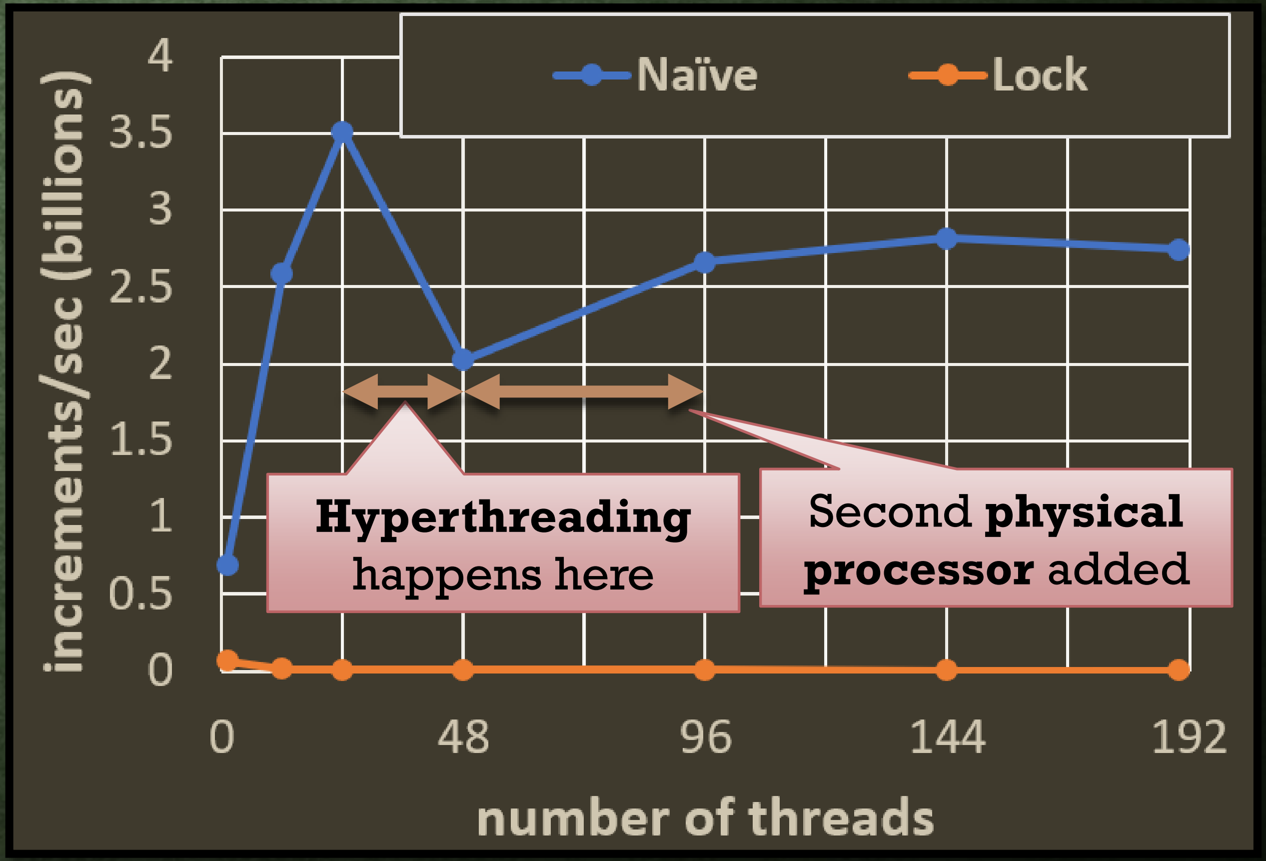
- Same as the previous "**accuracy experiment**"
  - Comparing final counter value of **naïve** and **lock-based**
- Output is now **correct**!
- (Of course, this experiment is **not** a correctness proof)
- What about **performance**?



PERFORMANCE COMPARISON

- Simple **timed** experiment
- Each data point = average of 5 trials
- In each trial, for 3 seconds,
  - threads repeatedly perform Increment,
  - and we measure increments/second
- What is the overhead of locking?
  - 10x slower with 1 thread
  - **450x slower** with 190 threads
- Is there a better tool than locking?

Machine with 4 physical processors (each with 24 cores + hyperthreading)



FETCH AND ADD (FAA)

- Instruction implemented modern Intel and AMD systems:
  - `lock xadd`
- FAA(addr, val) does the following **atomically** (all at once)
  - old = Read(addr)
  - temp = old + val
  - Write(addr, temp)
  - Return old

EASY FAA-BASED COUNTER

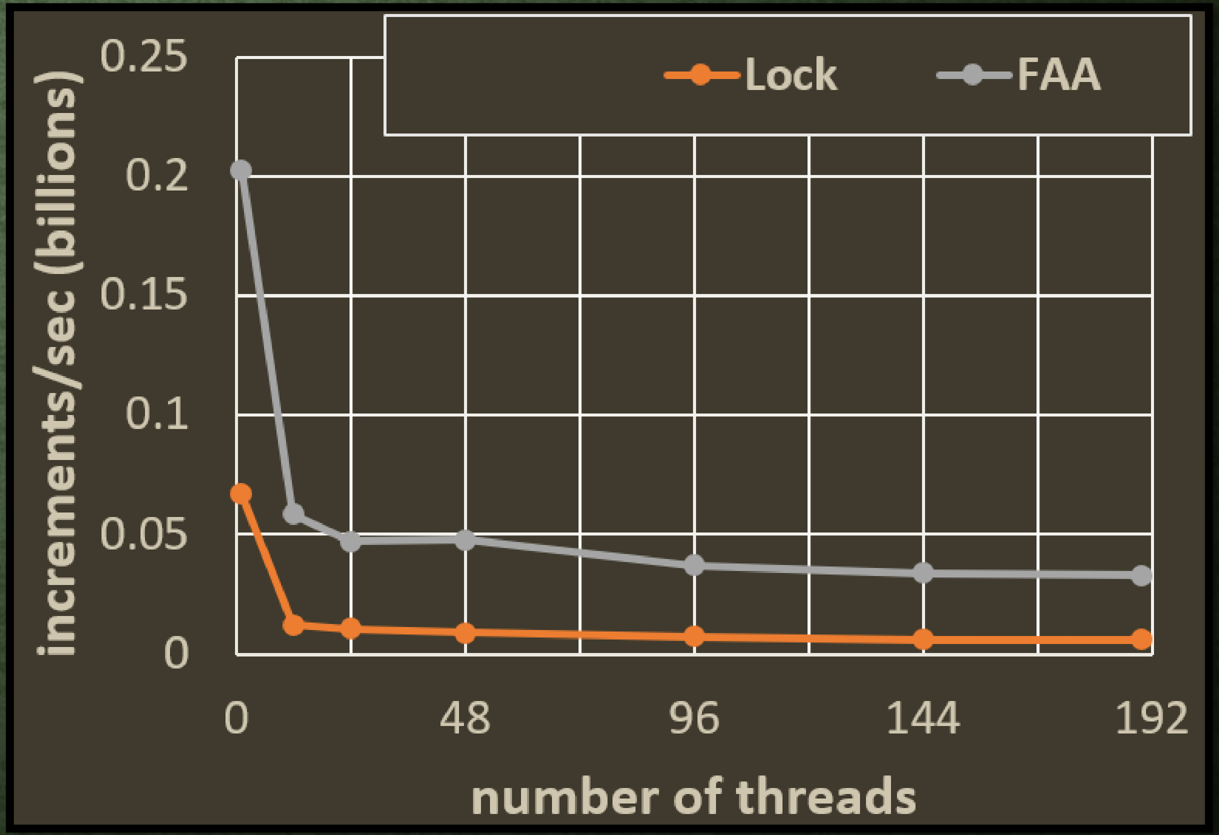
```cpp
class counter3 {
private:
    atomic<int> v;
public:
    counter3() : v(atomic<int>(0)) {}
    int increment(int threadID) {
        return v++;
    }
    int get() {
        return v;
    }
};
```

```asm
        mov     DWORD PTR [rsp-4], 0
        mov     eax, 1
        lock xadd       DWORD PTR [rsp-4], eax
        ret
```

Because **v** is **atomic<int>**, this is really a **FAA**!

HOW DOES THIS PERFORM IN PRACTICE?

- Same timed experiment
- Excluding Naïve from the graph (to zoom in on Lock and FAA)
- Compared to Lock
  - FAA is up to **5.4x faster**
- Compared to Naïve (incorrect)
  - FAA is up to **83x slower**
  - (much better than Lock's **450x**)



PROBLEM: TOO MUCH CONTENTION

- Accessing a single counter creates a **contention bottleneck**
- What if we **shard (partition)** the counter into multiple **sub counters**
  - Increment: pick one sub counter and increment it
  - What about Get?
    - Counter value is ***distributed*** over the sub counters
    - Trade-off
      - Single counter → slow Increment, fast Get
      - Sharded counter → fast Increment, slower/more complex Get?
    - We are going to **ignore** these complications and **only think about increment…**

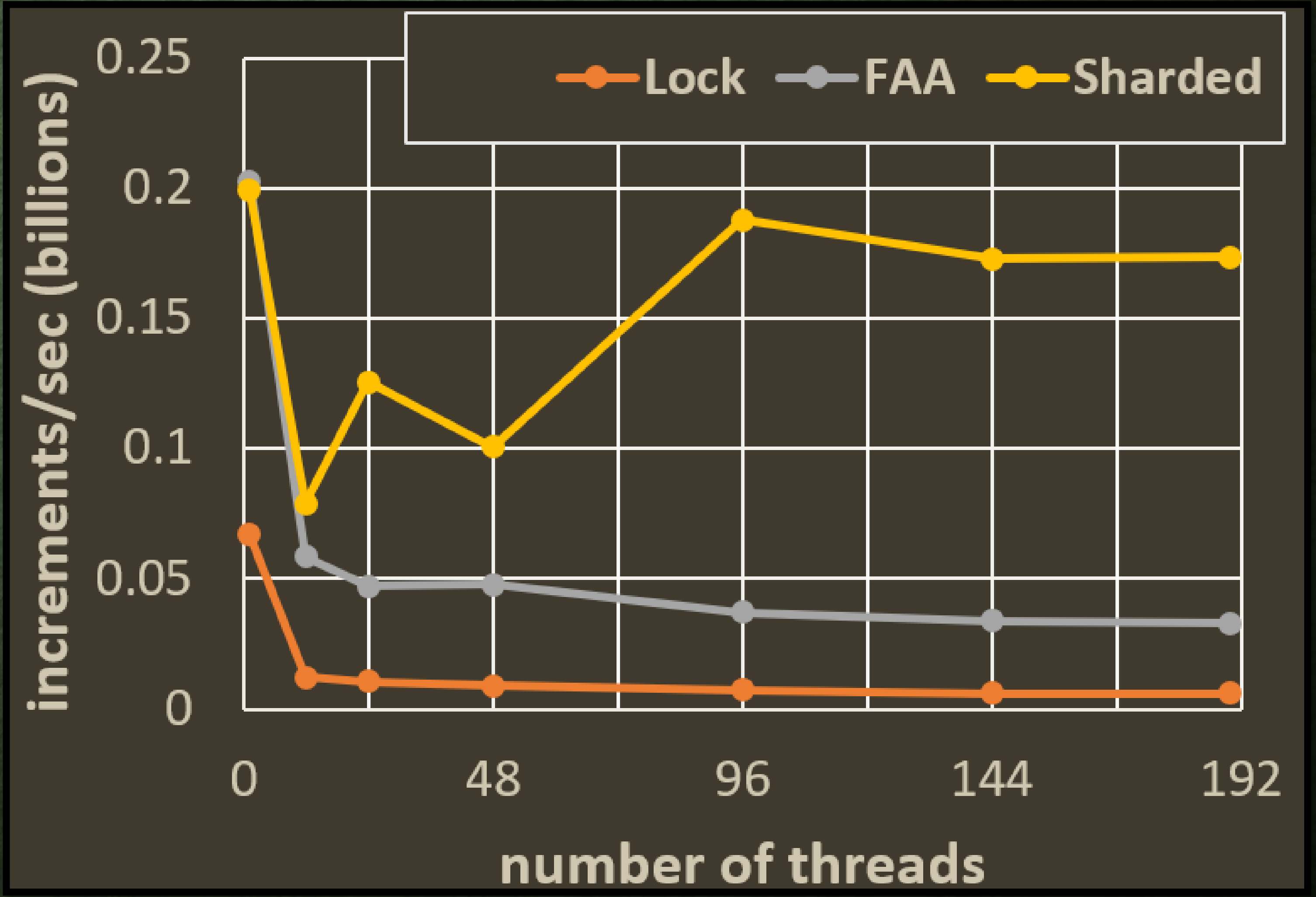
NAÏVE SHARDED COUNTER

- Each thread uses its own sub counter
- **No data sharing,** should scale perfectly

```
class counter4 {
private:
    atomic<int> data[MAX_THREADS];
public:
    counter4() {
        for (int threadID=0; threadID<MAX_THREADS; ++threadID)
            new (&data[threadID]) atomic<int>(0);
    }
    int increment(int threadID) {
        return data[threadID]++; // atomic
    }
    int get() {
        int sum = 0;
        for (int threadID=0; threadID<MAX_THREADS; ++threadID)
            sum += data[threadID];
        return sum;
    }
};
```

HOW DOES THIS PERFORM?

- Same timed experiment
- Why is the scaling so poor?
  - **No shared data**, right?
- Answer: **cache coherence**



HOW CACHE COHERENCE WORKS

Thread 1's cache

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	S

Thread 2's cache

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	X

Thread 1 reads w_2

Thread 2 reads w_7

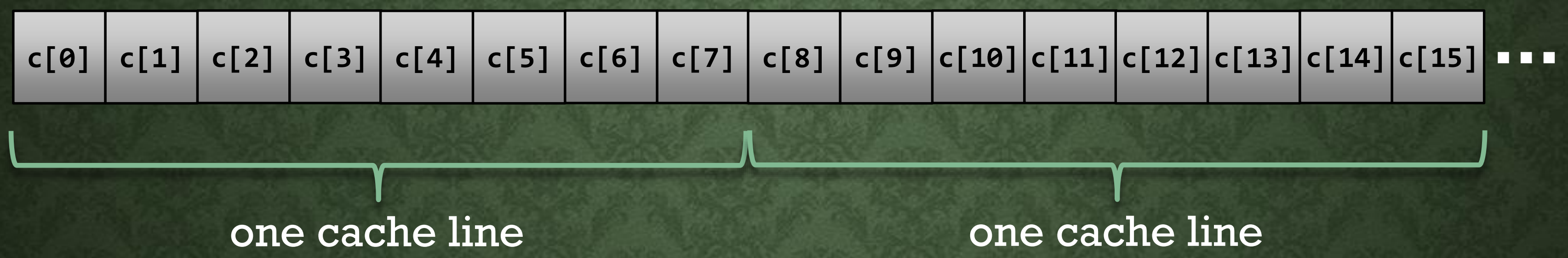
Cache line **invalidated** and **evicted!**

Thread 2 **writes** w_7

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8

64 byte (8 word) cache line

MEMORY LAYOUT OF SUB COUNTERS

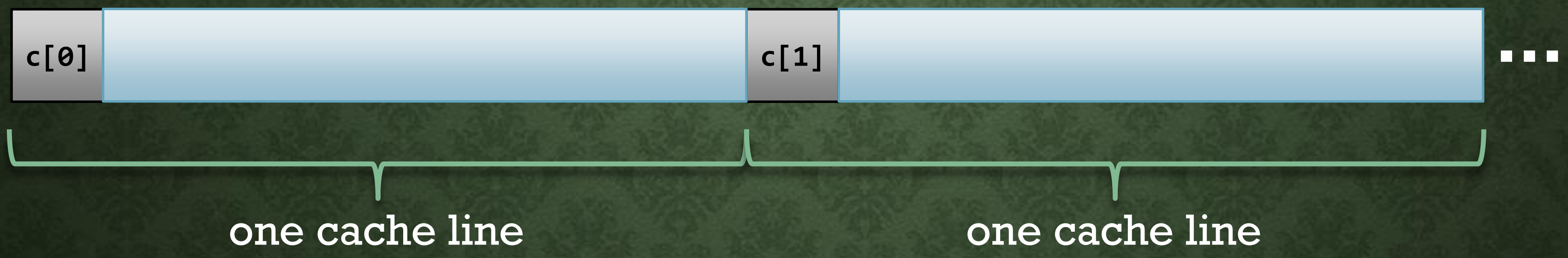


Incrementing **one of these** will invalidate **all of them** (causing huge contention)

This is called **false sharing**

SOLUTION: PADDING

- Add **empty space** to each sub counter
  - To make it **cache line sized**

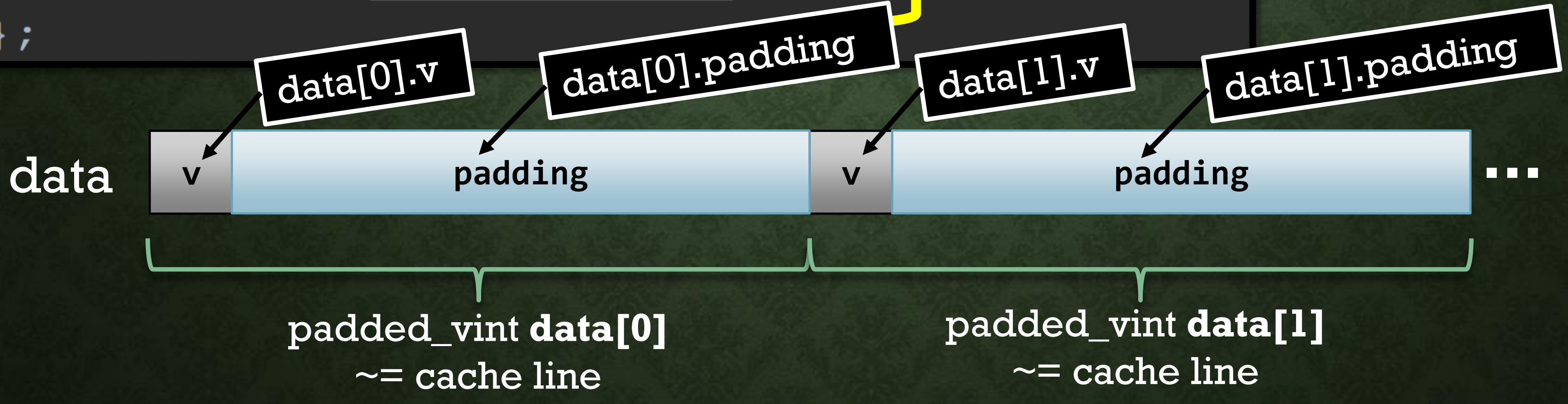


PADDED SHARDED COUNTER

```
class counter5 {
private:
    struct padded_vint {
        atomic<int> v;
        char padding[64-sizeof(atomic<int>)];
    };
    padded_vint data[MAX_THREADS];
public:
    counter5() {...4 lines }
    int increment(int threadID) { .
    int get() {...6 lines }
};
```

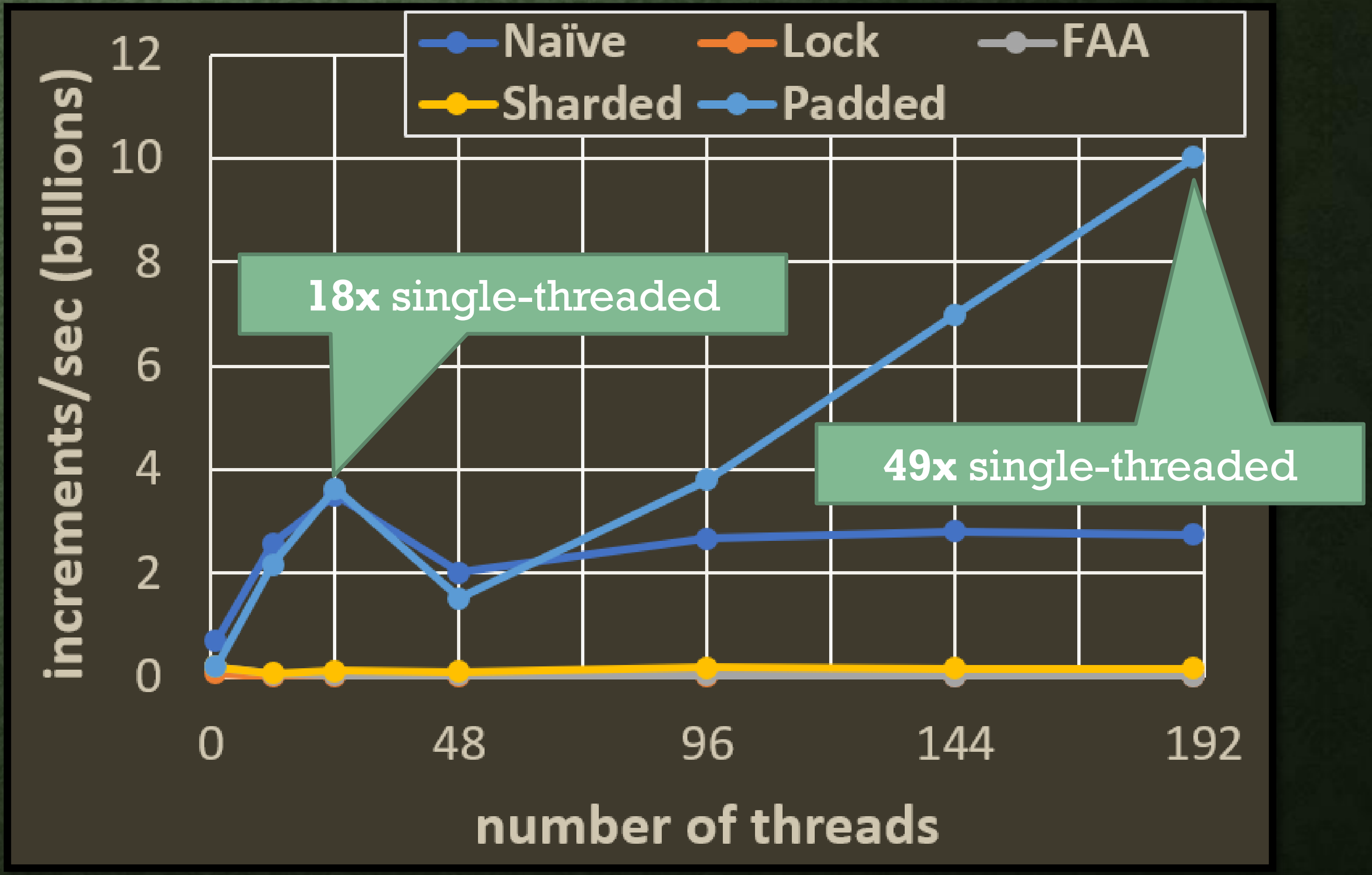
This is where the magic happens

Identical to naïve sharded counter



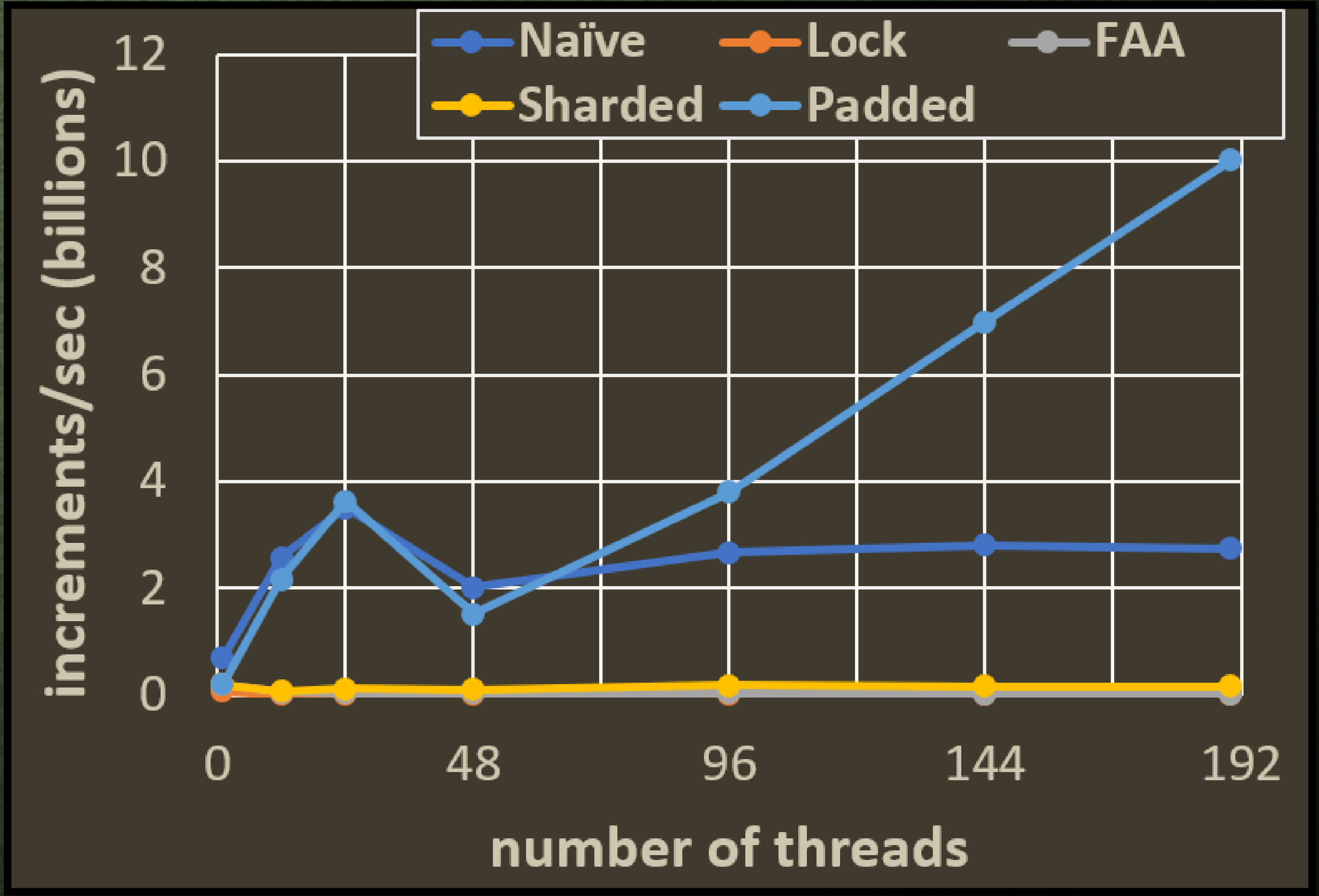
HOW DOES THIS PERFORM?

- Same experiment, but comparing **naïve sharding** with a **padded counter**
- **Pretty good scaling**
  - **18x vs optimal 24x**
  - **49x vs optimal 190x**



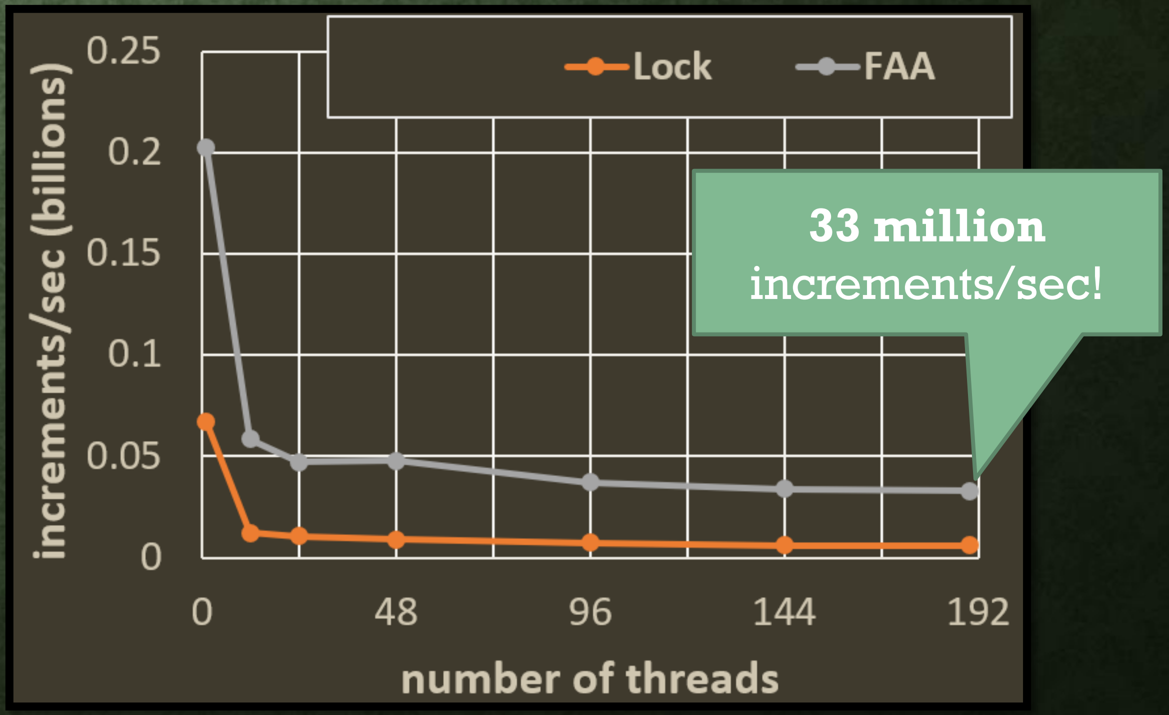
SPEED VS SIMPLICITY

- But… **reading** is hard!
  - Solving this will add complexity
- Simplicity is valuable!
  - Do we **need** a complex solution?
  - Sometimes… but not always…



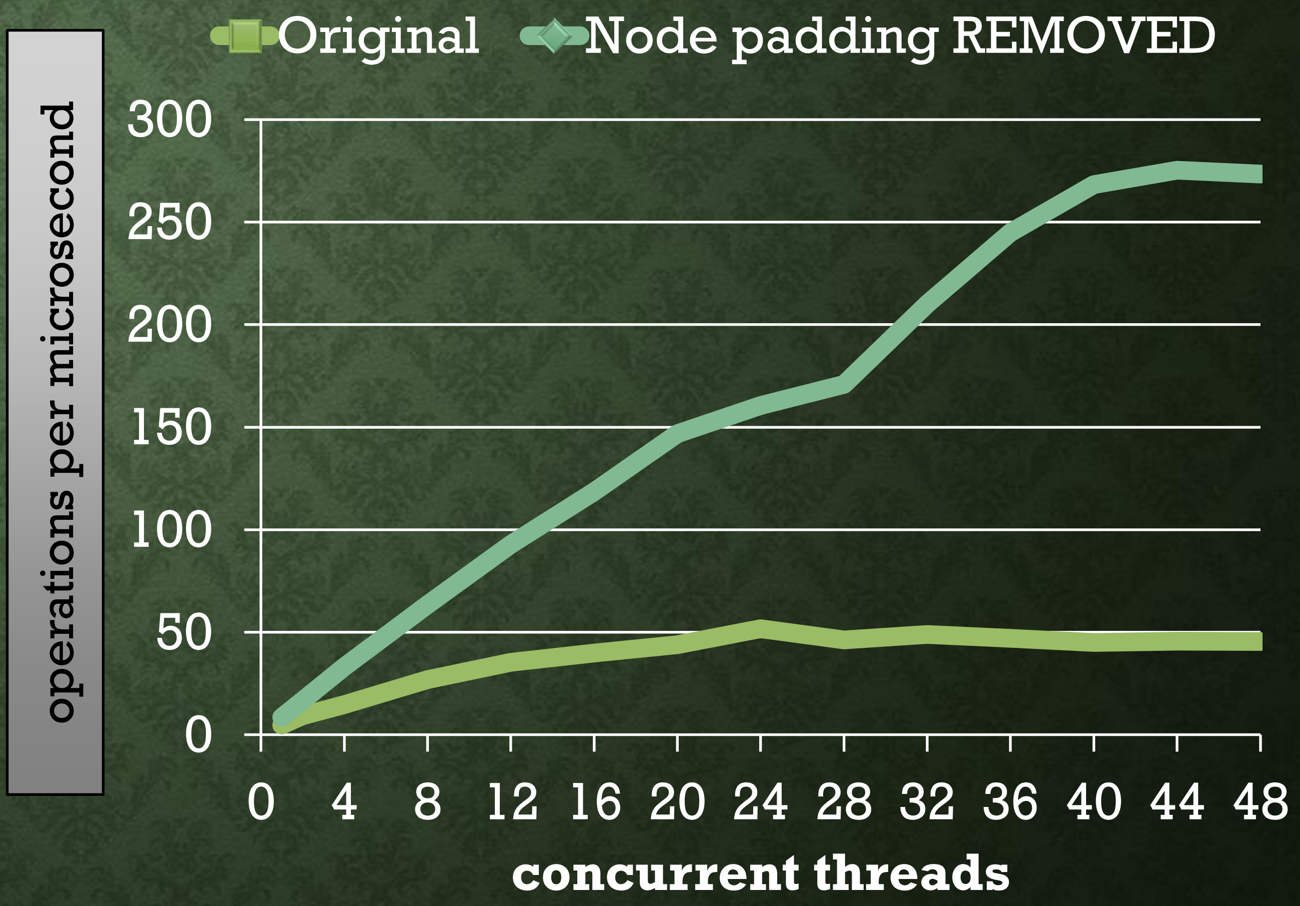
CONSIDERING USE CASES: A FAA-COUNTER MIGHT BE GOOD ENOUGH

- FAA-based counter does not truly scale
- But, its absolute throughput might be **high enough** for your application
- Real applications do more than just increment a single counter
  - Avoid unnecessary optimization
  - Figure out **if it's a bottleneck** first



A WORD OF WARNING: PADDING CAN HURT

- Union-find data structure
- Each 8b node was padded to 64b
- **Removing** padding → 5x faster!
- Why?
  - Many nodes, uniformly accessed
    → contention is rare
    → false sharing is rare
    → padding can't help much
  - Padding wastes space
    → $1/8^{th}$ as many nodes in cache!



Important principle!

WHEN TO PAD?

- When the number of objects being padded is O(# threads) for a small constant
- AND threads frequently **write** to these objects

- Try and see if it helps…

SUMMARY

- **Cache coherence**, shared and exclusive modes, cache invalidations, contention
- Sharding (partitioning data to reduce contention)
- False sharing and padding (principle: when to pad)
- Locks, fetch-and-add
- Implementing linearizable counters
    - Lock-based counter
    - Fetch-and-add counter
    - Sharded counter
    - Padded sharded counter