wCQ: A Fast Wait-Free Queue with Bounded Memory Usage

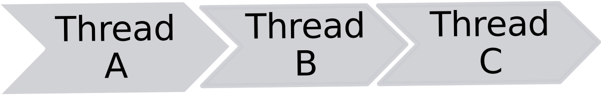
Ruslan Nikolaev *, rnikola@psu.edu, Penn State University, USA

Binoy Ravindran, binoy@vt.edu, Virginia Tech, USA

** Most of the work was done while the author was at Virginia Tech*

Concurrent Data Structures

- Many-core systems require efficient access to data
  - Concurrent data structures
- Multiple threads need to ***safely*** manipulate data structures (similar to sequential data structures)
  - "nothing bad will happen"

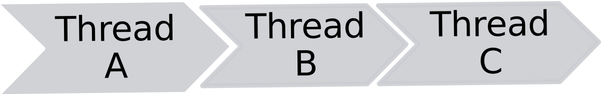

Thread A

Thread B

Thread C

Concurrent Data Structures

- Many-core systems require efficient access to data
  - Concurrent data structures
- Multiple threads need to ***safely*** manipulate data structures (similar to sequential data structures)
  - "nothing bad will happen"

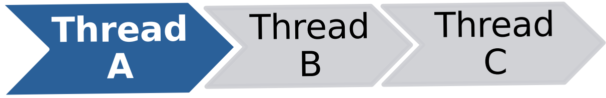


Thread A

Thread B

Thread C

- Concurrency also adds a ***liveness*** property, which stipulates how threads will be able to make progress
  - "something good will happen eventually"



Thread A

Thread B

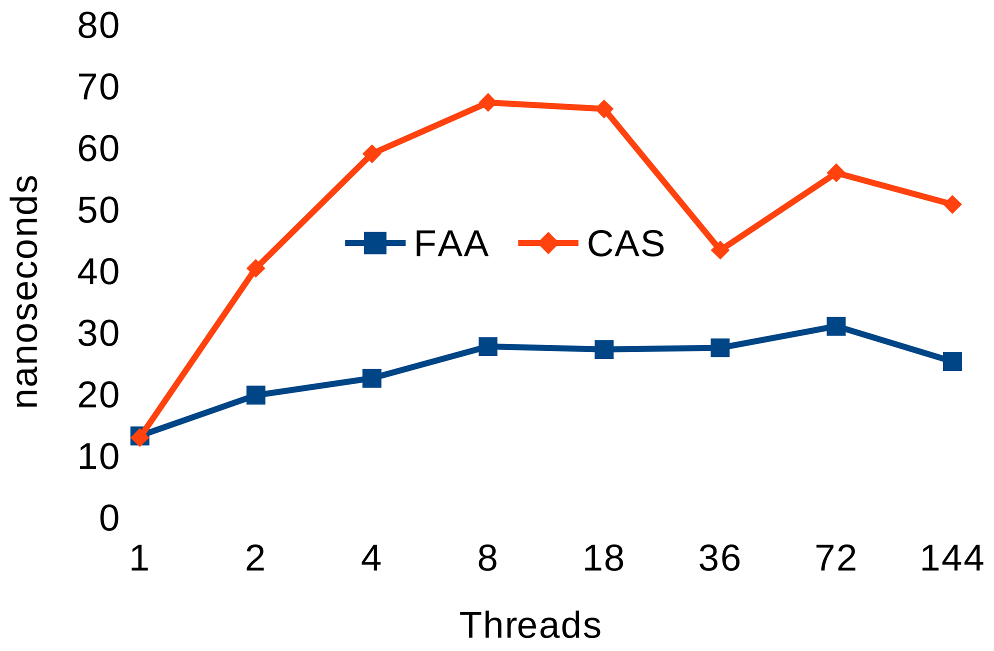
Thread C

Wait-Freedom

- **Non-blocking** data structures
  - **Lock-free** data structures require that at least ***one*** thread completes an operation after a ***finite*** number of steps
  - **Wait-free** data structures require that ***all*** threads complete ***any*** operation after a ***finite*** number of steps
- Wait-free algorithms have increasingly gained more attention due to their strongest non-blocking progress property
  - But building wait-free queues is challenging

F&A: Hardware-based vs. CAS-emulated

- F&A (fetch-and-add) generally scales better than CAS (compare-and-set)
  - Used by LCRQ [PPoPP’13], YMC [PPoPP’16], SCQ [DISC’19]



Xeon E7-8880 v3 2.3 GHz, 4x18 cores

Existing Approaches

- There are quite a few concurrent queues but there is no ***truly* wait-free** queue which has performance on par with state-of-the-art **lock-free** queues
- Kogan-Petrank's queue [PPoPP'11]
  - Wait-free but slow
- CRTurn queue [PPoPP'17]
  - Wait-free but is still slow
- Yang and Mellor-Crummey (YMC) queue [PPoPP'16]
  - Fast but has flawed memory reclamation => not truly wait-free
  - Uses ring buffers

Existing Approaches

- LCRQ [PPoPP’13]
  - Uses ring buffers
  - Fast and memory reclamation is correct but is only lock-free
  - Always needs a slower (M&S) queue as an outer layer for lock-free progress
- Scalable Circular Queue (SCQ) [DISC’19]
  - Uses ring buffers
  - Fast but is only lock-free
  - Unlike LCRQ, does not need M&S queue for lock-free progress
- **We present a wait-free circular queue (wCQ) which extends SCQ**

Background: Infinite Array Queue (livelock-prone)

```
int Tail = 0, Head = 0;

void enqueue(void *p) {
    while (true) {
        T = F&A(&Tail, 1);
        if (SWAP(&Array[T], p) = ⊥)
            break;
    }
}
```

```
void *dequeue() {
    while (true) {
        H = F&A(&Head, 1);
        p = SWAP(&Array[H], T);
        if (p ≠ ⊥) return p;
        if (Load(Head) ≤ H + 1)
            return nullptr;
    }
}
```

Background: Infinite Array Queue (livelock-prone)

```
int Tail = 0, Head = 0;

void enqueue(void *p) {
    while (true) {
        T = F&A(&Tail, 1);
        if (SWAP(&Array[T], p) = ⊥)
            break;
    }
}
```

```
void *dequeue() {
    while (true) {
        H = F&A(&Head, 1);
        p = SWAP(&Array[H], T);
        if (p ≠ ⊥) return p;
        if (Load(Head) ≤ H + 1)
            return nullptr;
    }
}
```

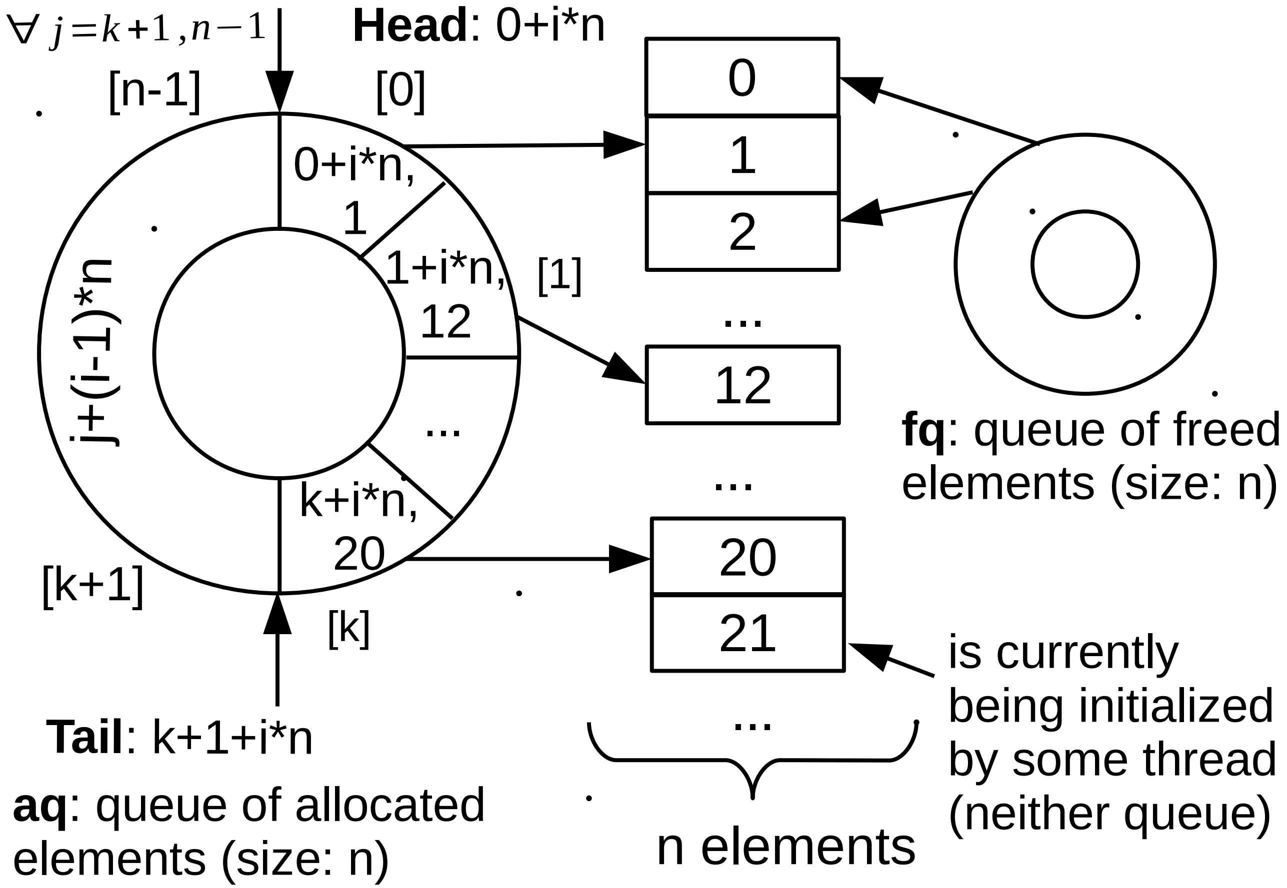
Background: Infinite Array Queue (livelock-prone)

```
int Tail = 0, Head = 0;

void enqueue(void *p) {
    while (true) {
        T = F&A(&Tail, 1);
        if (SWAP(&Array[T], p) = ⊥)
            break;
    }
}
```

```
void *dequeue() {
    while (true) {
        H = F&A(&Head, 1);
        p = SWAP(&Array[H], T);
        if (p ≠ ⊥) return p;
        if (Load(Head) ≤ H + 1)
            return nullptr;
    }
}
```

Background: SCQ's Data Structure



$\forall j = k+1, n-1$

[n-1]

Head: 0+i*n

[0]

0+i*n, 1

1+i*n, 12

[1]

...

k+i*n, 20

j+(i-1)*n

[k+1]

[k]

Tail: k+1+i*n

aq: queue of allocated elements (size: n)

0

1

2

...

12

...

20

21

...

n elements

fq: queue of freed elements (size: n)

is currently being initialized by some thread (neither queue)

- Two queues
  - **aq** and **fq** store indices
  - A data array contains fixed-size elements (or arbitrary pointers)
  - Uses only a single word and avoids ABA
    - Crucial for wCQ!

Challenges

- Memory reclamation is tough when also considering wait-free progress properties
  - Not impossible but is error-prone
  - Better to avoid altogether if possible
- Kogan-Petrank's ***fast-path-slow-path*** method [PPoPP'12] does not support specialized instructions such as ***fetch-and-add*** (F&A)
  - F&A scales better and is the key instruction in SCQ
    - Unclear how to leverage F&A with Kogan-Petrank's method
  - Uses dynamic memory allocation
    - Implicitly assumes memory reclamation

wCQ’s Key Idea

- ***Key insight***: avoid memory reclamation altogether
  - Allocate fixed-size ring buffers and one descriptor per each thread during initialization

wCQ's Key Idea

- ***Key insight***: avoid memory reclamation altogether
  - Allocate fixed-size ring buffers and one descriptor per each thread during initialization
- We design our own fast-path-slow-path method for SCQ that also supports F&A
  - The fast path is almost identical to SCQ
  - No memory reclamation is needed: all descriptors are static
  - The slow path is used as a fall-back if no progress is being made after several iterations

wCQ’s Key Idea

- ***Key Requirement*:** a double-width CAS, available on x86-64 and AArch64
  - Also possible to implement via single-width LL/SC on certain architectures such as PowerPC and MIPS
  - Keeps an additional cycle for entries to avoid inconsistencies when multiple threads modify the same element (slow path)
  - Also used with head and tail to keep a special helpee request (slow path)
  - **But fast paths still use a regular CAS and hardware F&A for head and tail!**

wCQ’s Key Idea

- ***Key Requirement*:** a double-width CAS, available on x86-64 and AArch64
  - Also possible to implement via single-width LL/SC on certain architectures such as PowerPC and MIPS
  - Keeps an additional cycle for entries to avoid inconsistencies when multiple threads modify the same element (slow path)
  - Also used with head and tail to keep a special helpee request (slow path)
  - **But fast paths still use a regular CAS and hardware F&A for head and tail!**
- ***Slow path*** does not take advantage of F&A anymore (for the most part)
  - It must be compatible with F&A in the fast path though

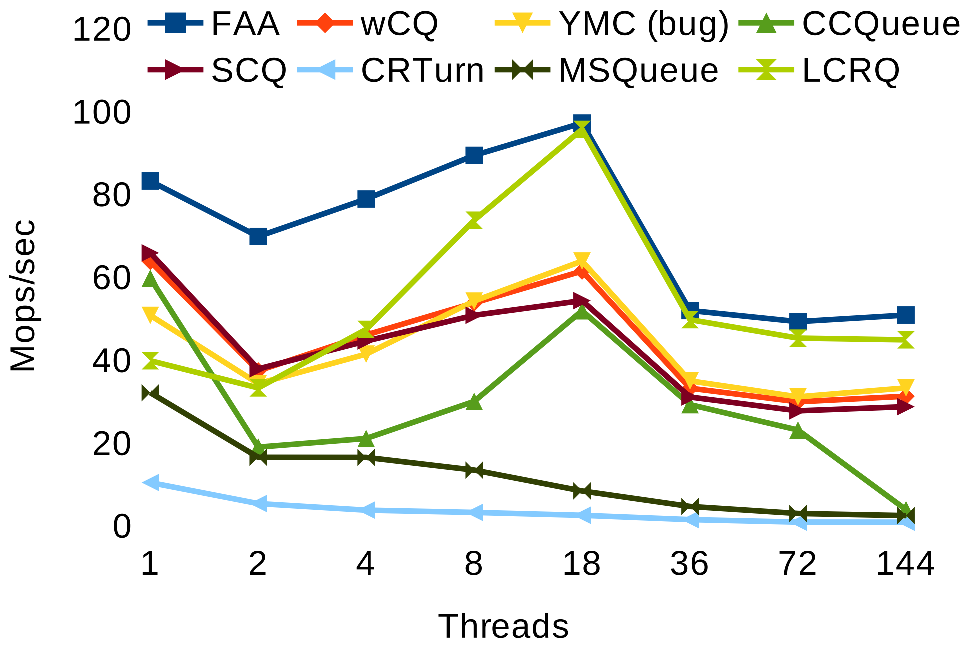
wCQ’s Slow Path

- In the **slow path**, F&A is substituted with a more complex operation slow_F&A, which allows coordinated increments of head/tail counters across multiple helpers and the helpee
- All active threads eventually converge to help a thread that is stuck
  - One of these threads will eventually succeed due to the underlying SCQ’s ***lock-free*** guarantees (i.e., at least one thread always succeeds)
  - All helpers must repeat exactly the same procedure as the helpee

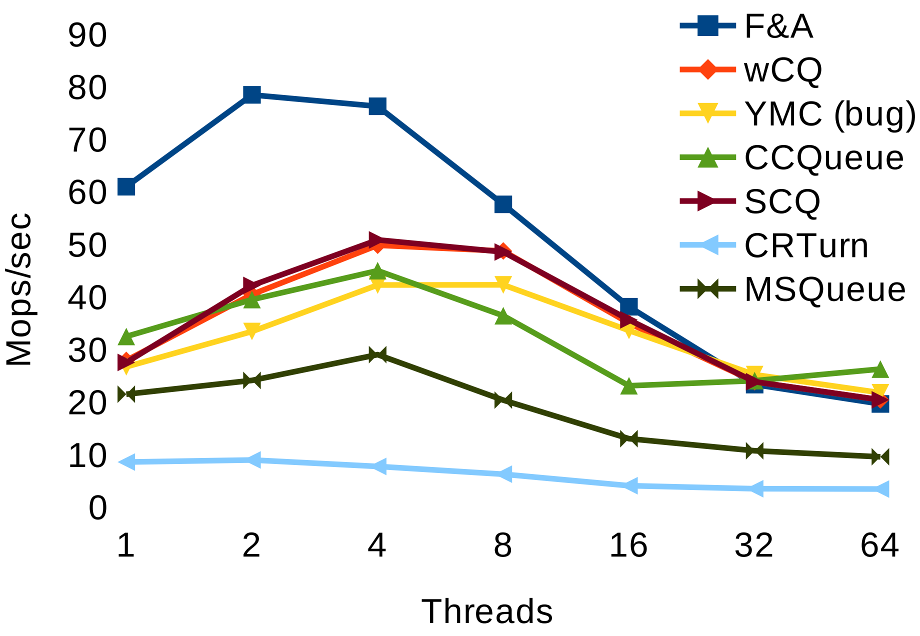
Evaluation

- **wCQ** is the fastest wait-free queue
  - wCQ generally outperforms YMC, for which memory usage can be unbounded
  - LCRQ can yield better performance but lacks wait-freedom
- **wCQ**’s performance is close to the SCQ algorithm

Evaluation: 50% Enqueue, 50% Dequeue

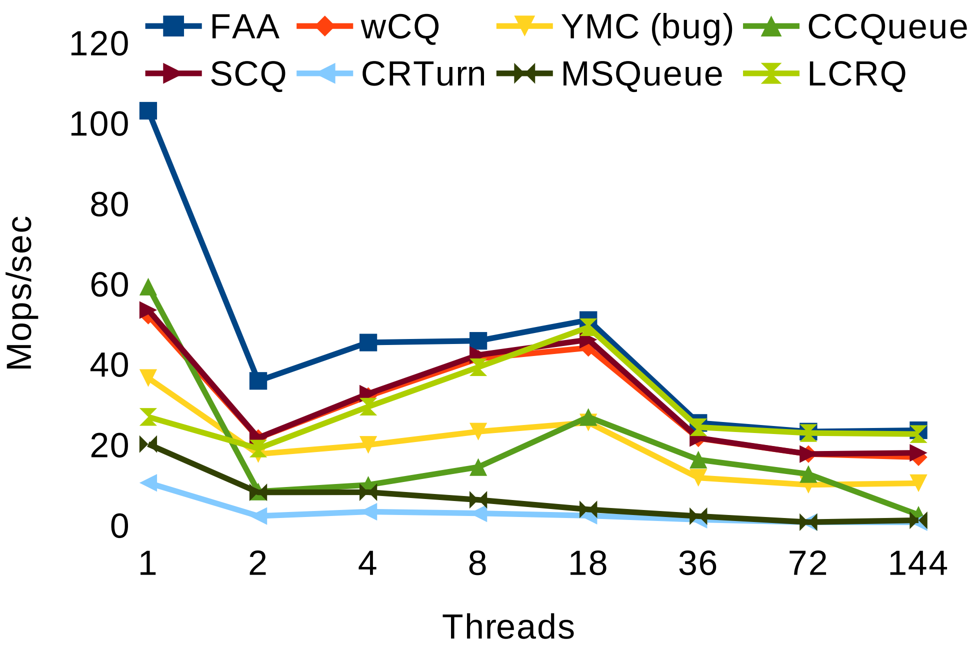


Xeon E7-8880 v3 2.3 GHz, 4x18 cores

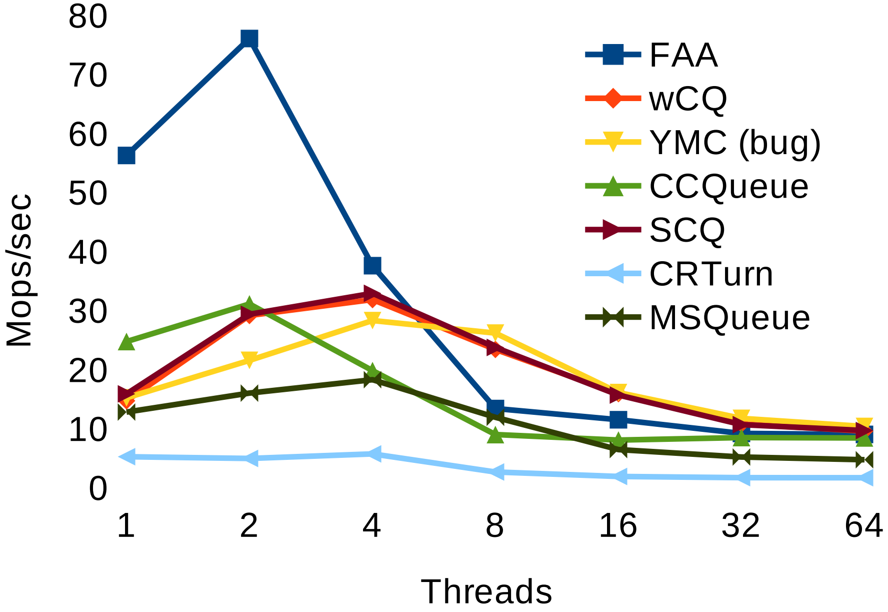


POWER8 3.0 GHz, 8x8 cores

Evaluation: Pairwise Enqueue-Dequeue

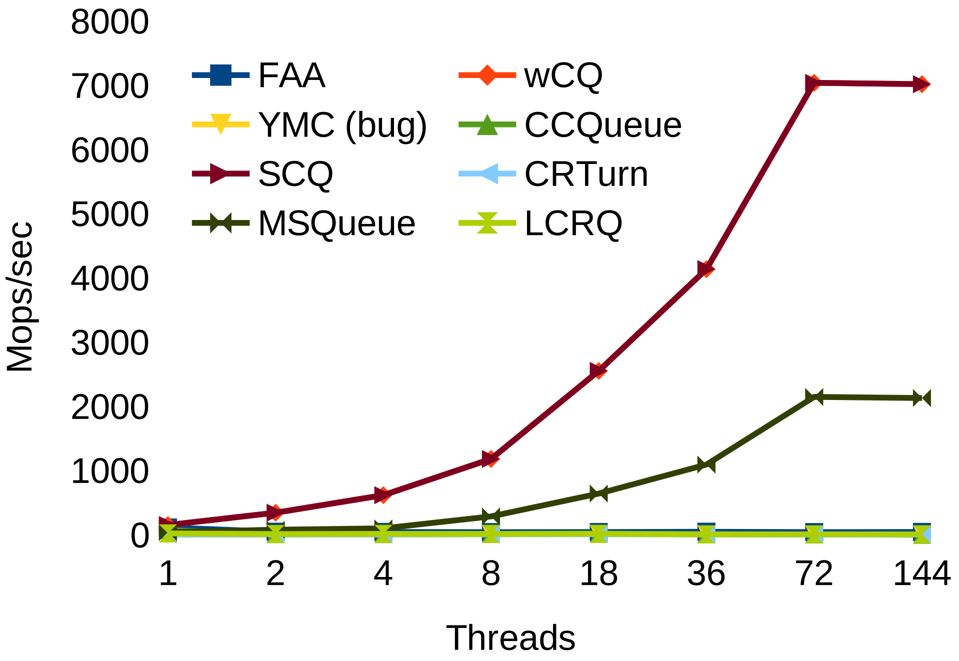


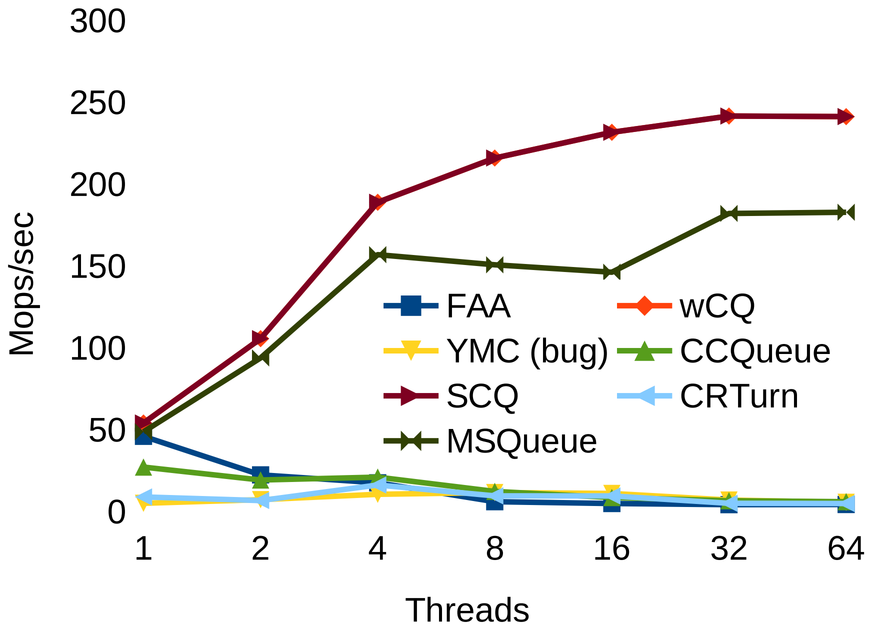
Xeon E7-8880 v3 2.3 GHz, 4x18 cores



POWER8 3.0 GHz, 8x8 cores

Evaluation: Empty Dequeue





Xeon E7-8880 v3 2.3 GHz,
4x18 cores

POWER8 3.0 GHz,
8x8 cores

Remarks

- **wCQ** implements a **bounded** queue (ring buffer)
- LCRQ and SCQ link ring buffers together to create an **unbounded** queue
  - The outer layer does not need to be scalable
    - LCRQ and SCQ use (lock-free) M&S queue
- The same approach can be taken with wCQ
  - Use a slower wait-free queue as an outer layer (e.g., CRTurn)

Source Code

- Code is open-source and available at:
  - https://github.com/rusnikola/wfqueue

Source Code

- Code is open-source and available at:
  - https://github.com/rusnikola/wfqueue

THANK YOU!