A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue

Ruslan Nikolaev
Systems Software Research Group
Virginia Tech, USA

Motivation

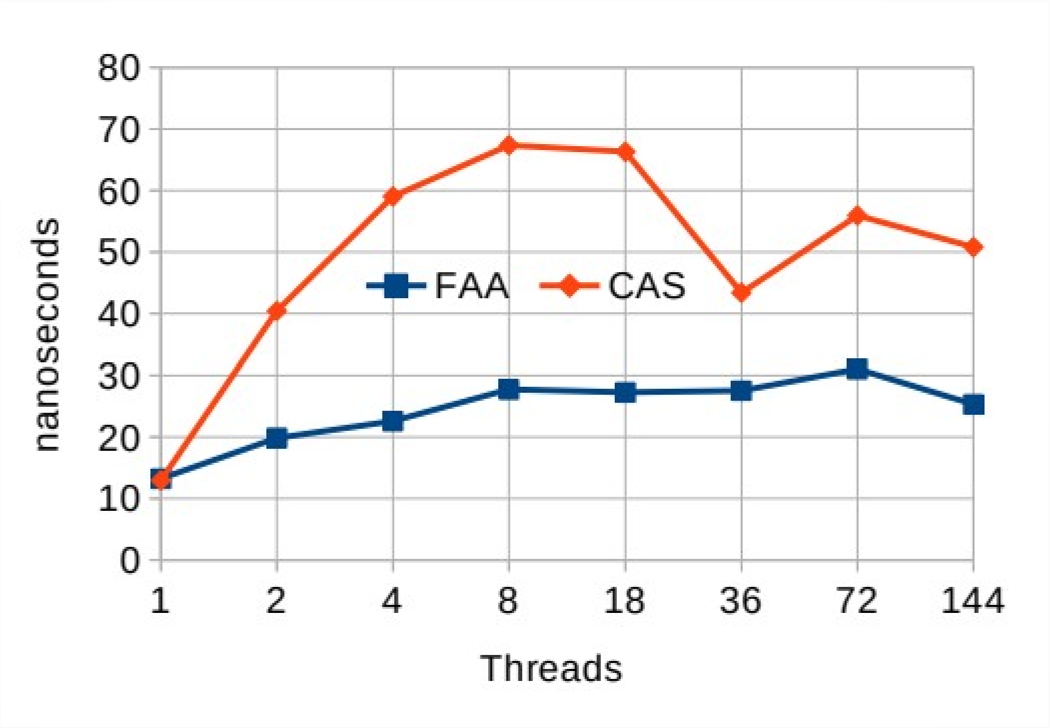
- Efficient concurrent FIFO queues are hard
  - Elimination techniques and relaxed FIFO queues are typically specialized
- Desirable properties
  - *Scalability*: leveraging many cores efficiently
  - *Portability*: using standard atomic primitives (e.g., single-width CAS)
  - *Memory Efficiency*: high memory utilization, avoiding reallocation due to livelocks

Existing Approaches

- Classical Michael & Scott's (M&S) FIFO queue: not very scalable [*PODC'96*]
- Various "lockless" ring buffers (circular queues). They are typically either not lock-free or linearizable, or both
- Lock-free ring buffers. They are not that scalable [*Tsigas et al: SPAA'01, Feldman et al.: SIGAPP'15*]
- LCRQ: a M&S list of scalable (but livelock-prone) ring buffers. Requires double-width CAS [*Morrison et al: PPoPP'13*]
- WFQUEUE: a wait-free design, the fast-path-slow-path methodology workarounds livelocks. More complex API and per-thread state [*Yang et al: PPoPP'16*]

FAA vs. CAS

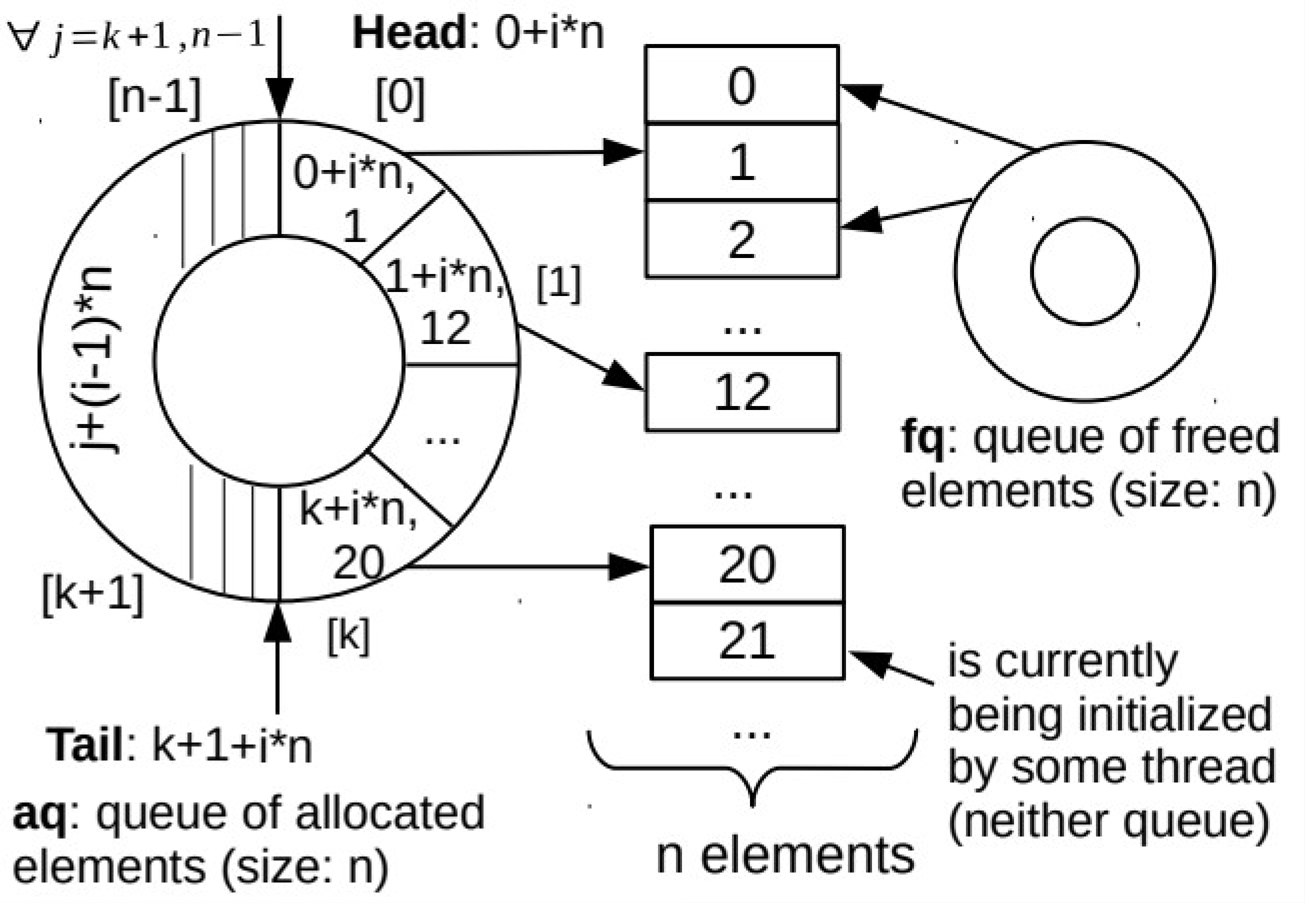
- FAA (fetch-and-add) generally scales better than CAS (compare-and-set)
  - Can be leveraged for ring buffers (LCRQ, WFQUEUE)



Xeon E7-8880 v3 2.3 GHz, 4x18 cores

Proposed Data Structure

- Two queues
  - **aq** and **fq** store indices
  - A data array contains elements
  - Single-width CAS is sufficient!



Infinite Array Queue (livelock-prone)

- The original design described for LCRQ

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

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

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

Infinite Array Queue (livelock-prone)

- The original design described for LCRQ

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

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

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

Infinite Array Queue (livelock-prone)

- The original design described for LCRQ

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

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

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

Infinite Array Queue (livelock-prone)

- The original design described for LCRQ

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

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

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

Infinite Array Queue (livelock-free)

- We use our data structure and introduce a “threshold”

```
int Tail = 0, Head = 0;
signed int Threshold = -1;

void enqueue(size_t idx) {
   while (true) {
      T = FAA(&Tail, 1);
      if (SWAP(&Ent[T], idx) = ⊥) {
         Store(&Threshold, 2n-1);
         break;
      }
   }
}
```

```
size_t dequeue() {
   if (Load(&Threshold) < 0)
      return <empty>;
   while (true) {
      H = FAA(&Head, 1);
      idx = SWAP(&Ent[H], T);
      if (idx != ⊥) return idx;
      if (FAA(&Threshold, -1) ≤ 0)
         return <empty>;
      if (Load(Head) ≤ H + 1)
         return <empty>;
   }
}
```

Infinite Array Queue (livelock-free)

- We use our data structure and introduce a "threshold"

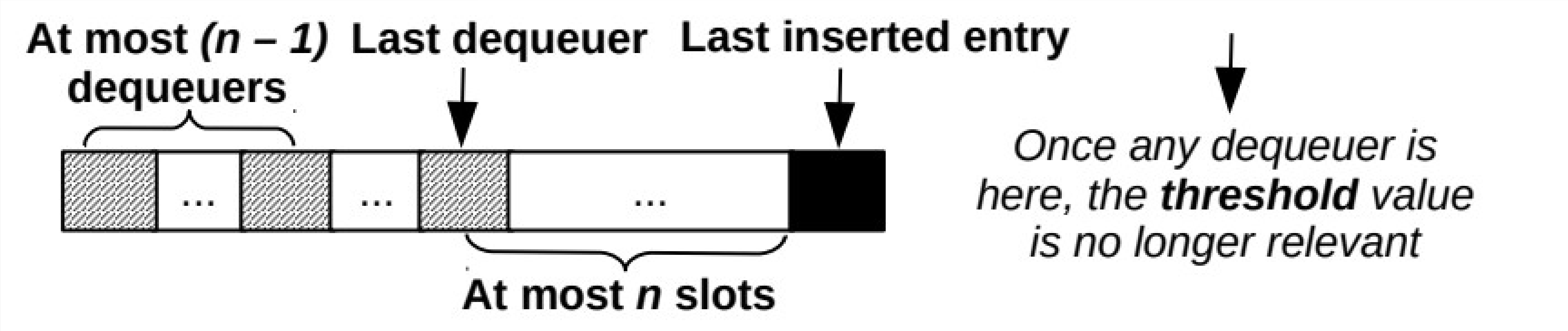
```
int Tail = 0, Head = 0;
signed int Threshold = -1;

void enqueue(size_t idx) {
   while (true) {
      T = FAA(&Tail, 1);
      if (SWAP(&Ent[T], idx) = ⊥) {
         Store(&Threshold, 2n-1);
         break;
      }
   }
}
```

```
size_t dequeue() {
   if (Load(&Threshold) < 0)
      return <empty>;
   while (true) {
      H = FAA(&Head, 1);
      idx = SWAP(&Ent[H], T);
      if (idx != ⊥) return idx;
      if (FAA(&Threshold, -1) ≤ 0)
         return <empty>;
      if (Load(Head) ≤ H + 1)
         return <empty>;
   }
}
```

Threshold Bound

- Consider two cases
  - The last dequeuer is ahead of the last enqueuer (the threshold value does not matter)
  - The last dequeuer is not ahead of the last enqueuer

```
Number of threads ≤ n
```

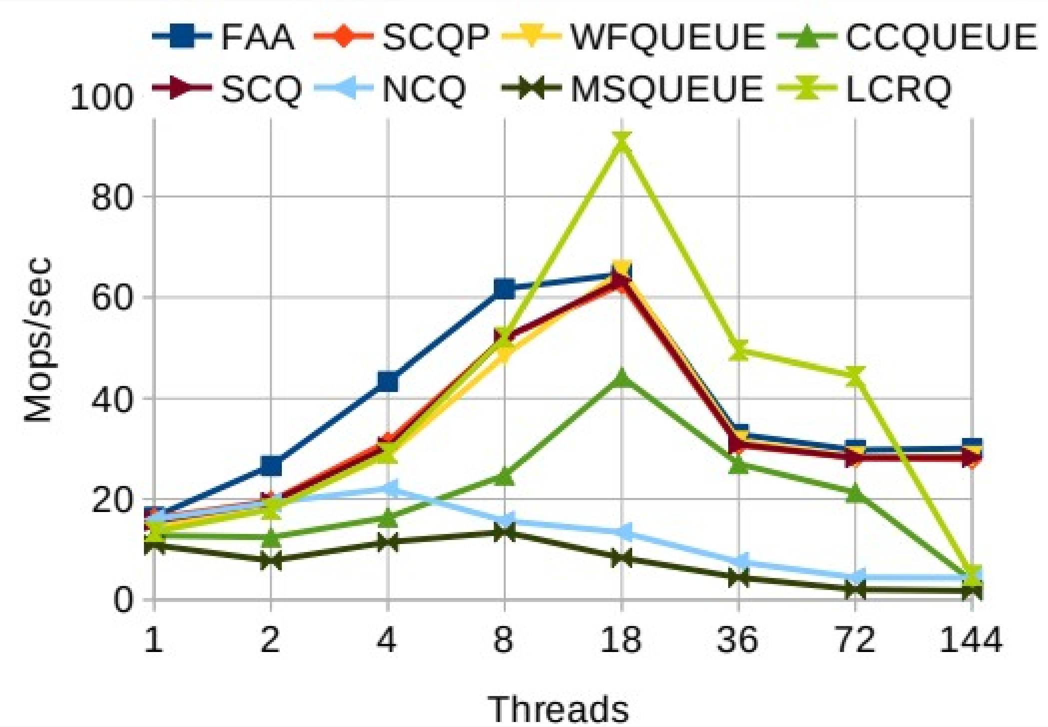


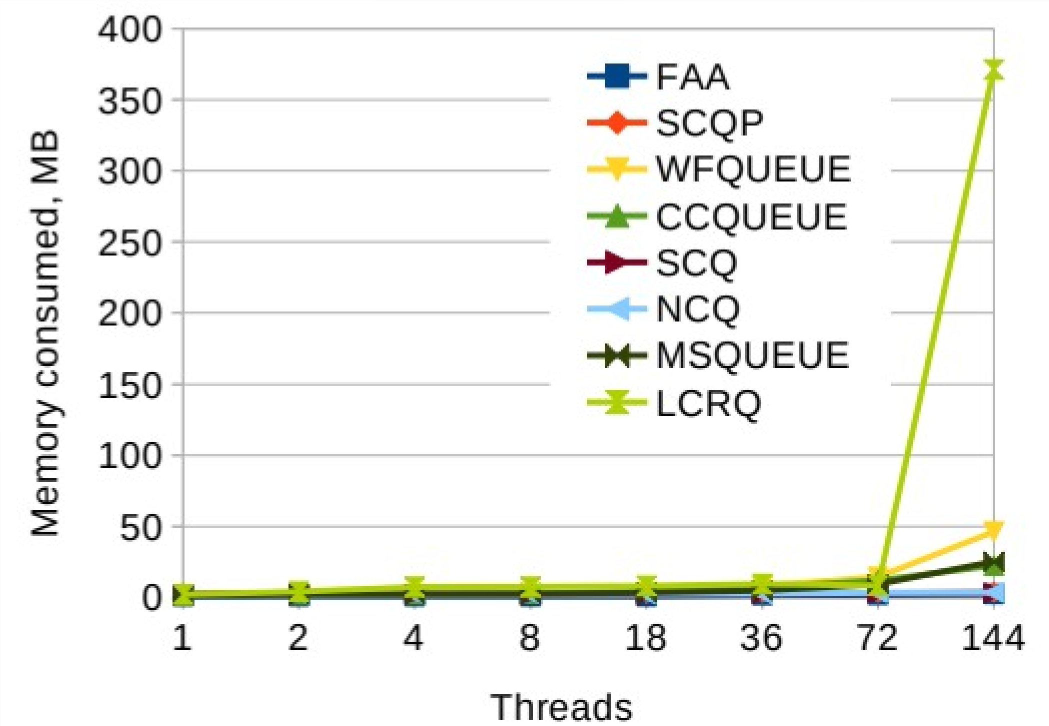
Scalable Circular Queue (SCQ)

- We **double the capacity** of the queue and set the threshold value to (3n-1)
- Some other differences (e.g., cycle management) with LCRQ
- (Unbounded) LSCQ: more memory efficient than LCRQ
- A specialized version of SCQ for double-width CAS

Evaluation: Memory Usage

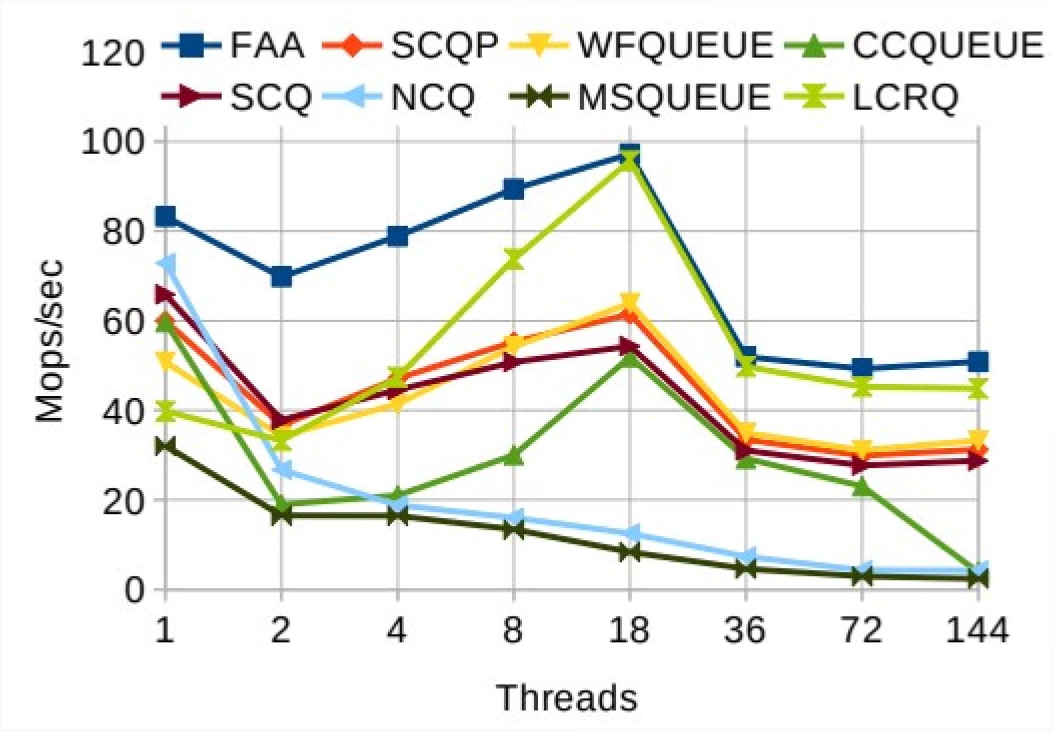
Xeon E7-8880 v3 2.3 GHz, 4x18 cores



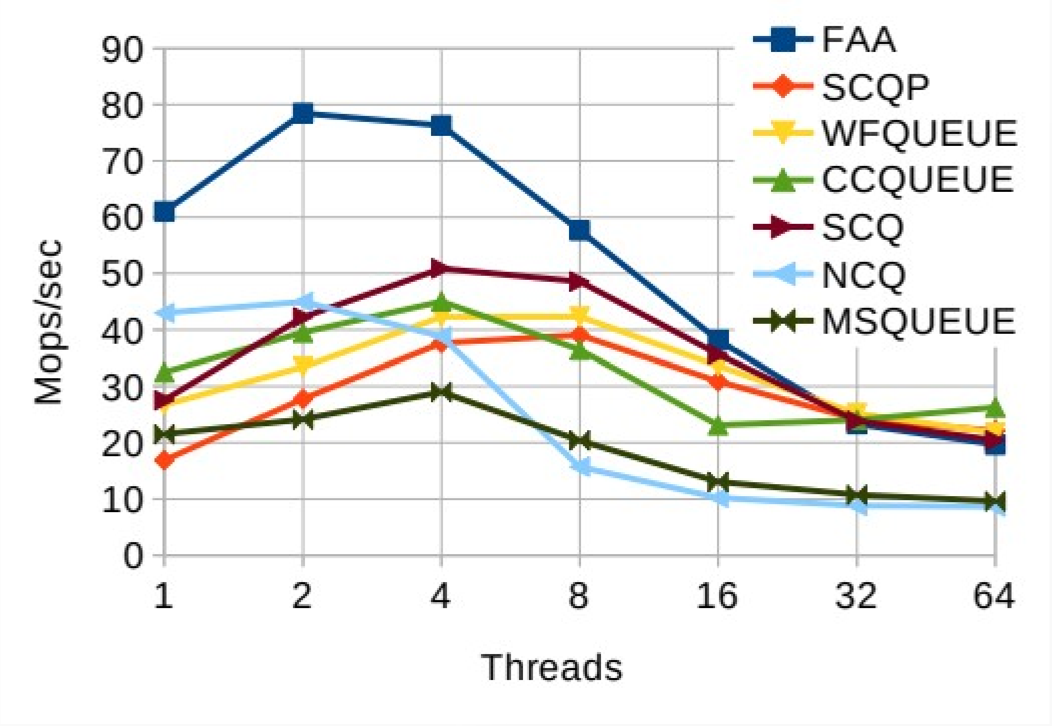


Evaluation: 50% Enq, 50% Deq

Xeon E7-8880 v3 2.3 GHz, 4x18 cores

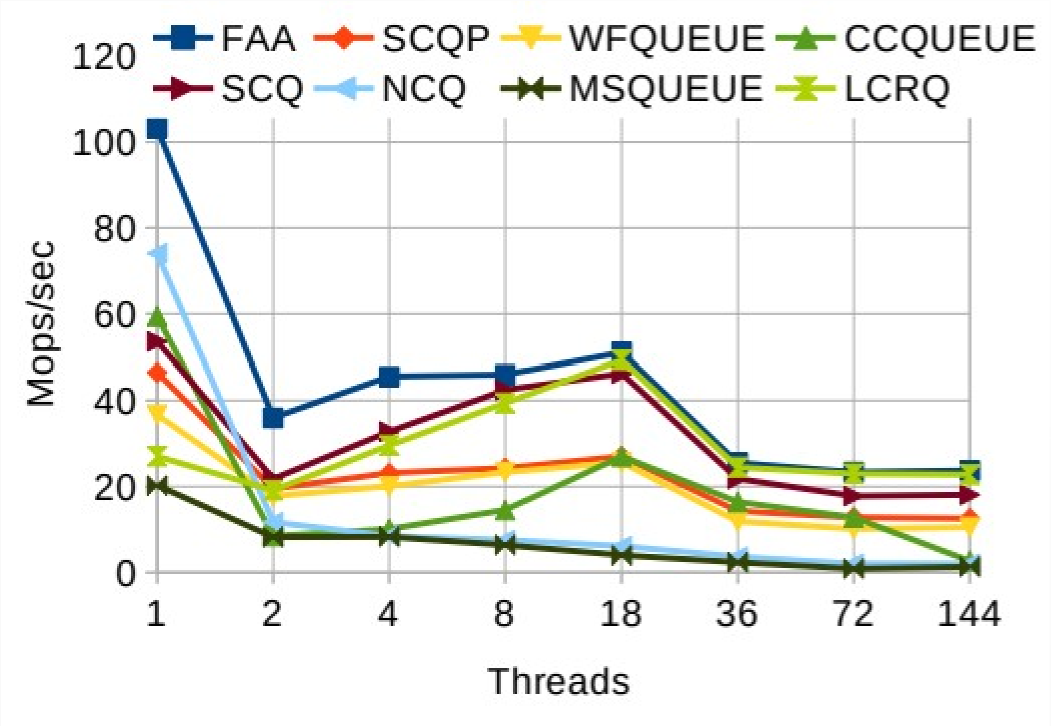


POWER8 3.0 GHz, 8x8 cores

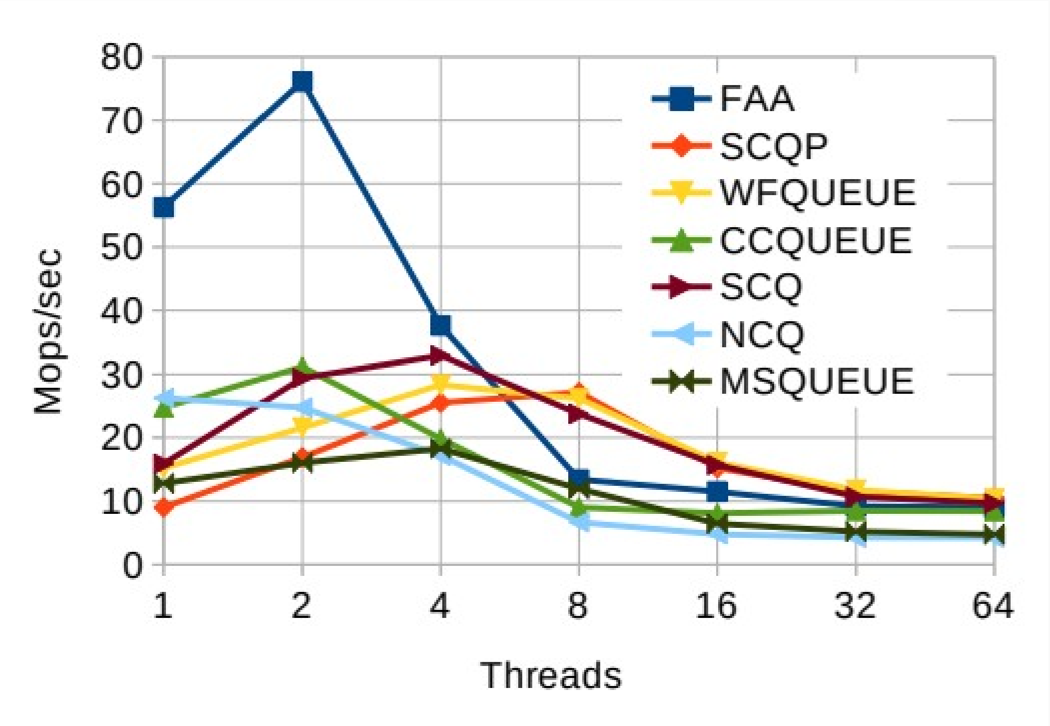


Evaluation: Pairwise Enq-Deq

Xeon E7-8880 v3 2.3 GHz, 4x18 cores



POWER8 3.0 GHz, 8x8 cores



More details

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

Thank you!