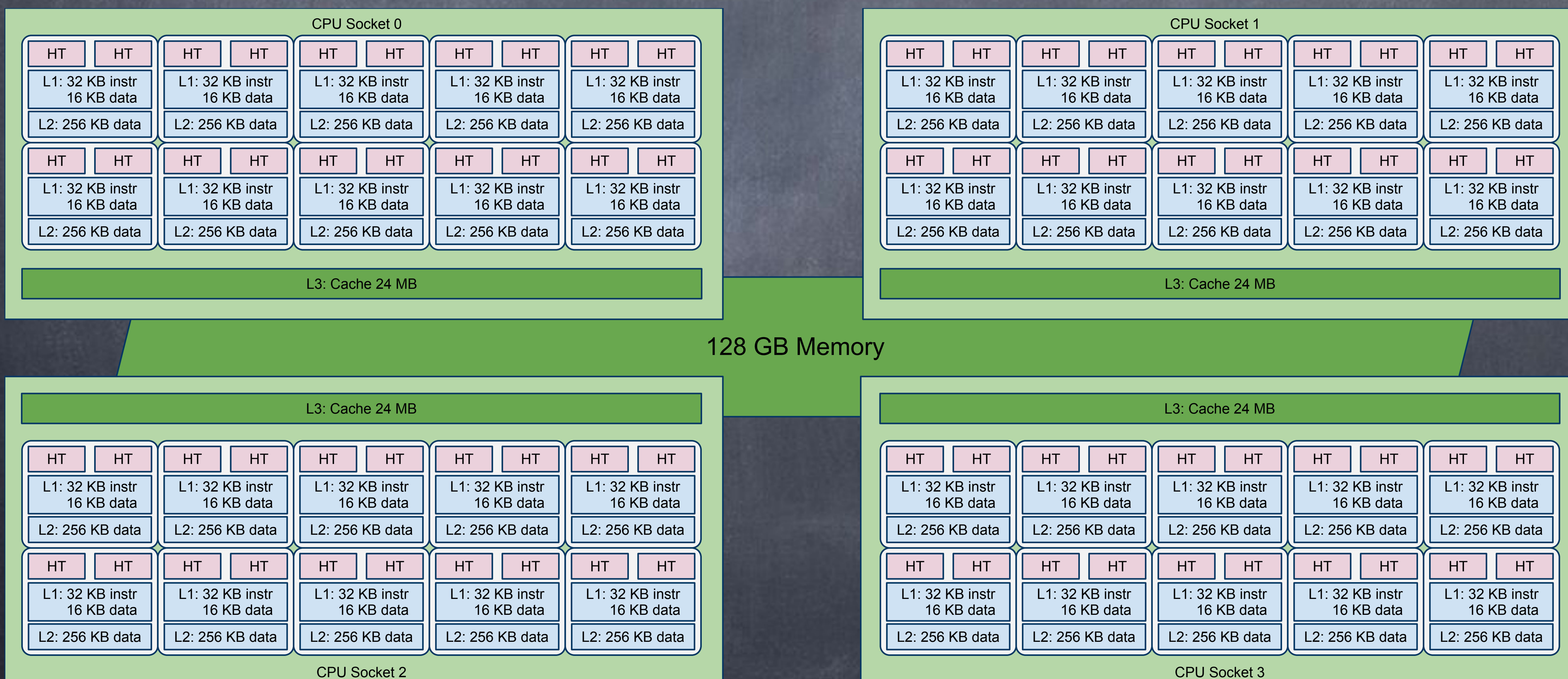
Distributed Queues: Faster Pools and Better Queues

Christoph Kirsch
Universität Salzburg

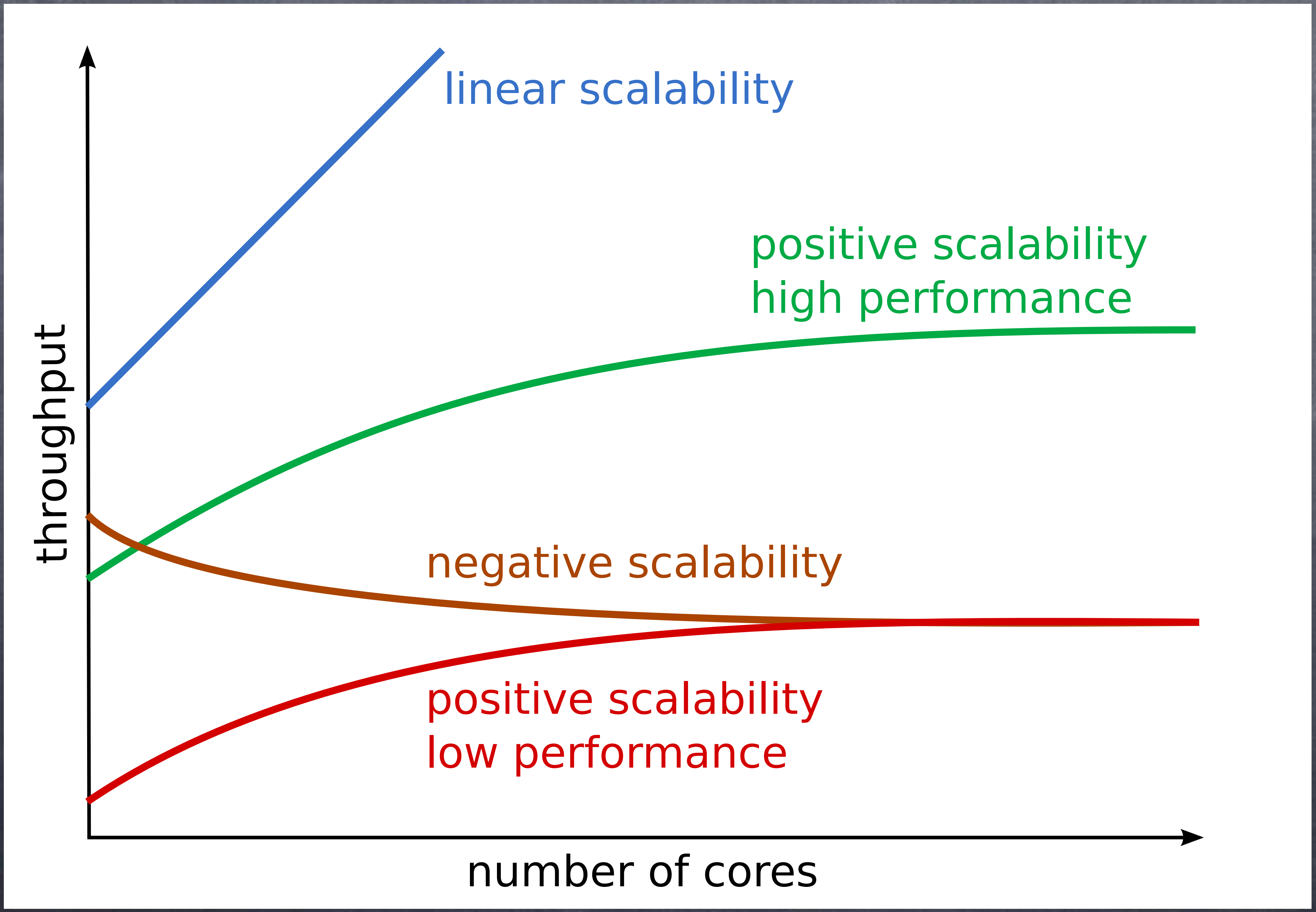
Stanford, December 2012

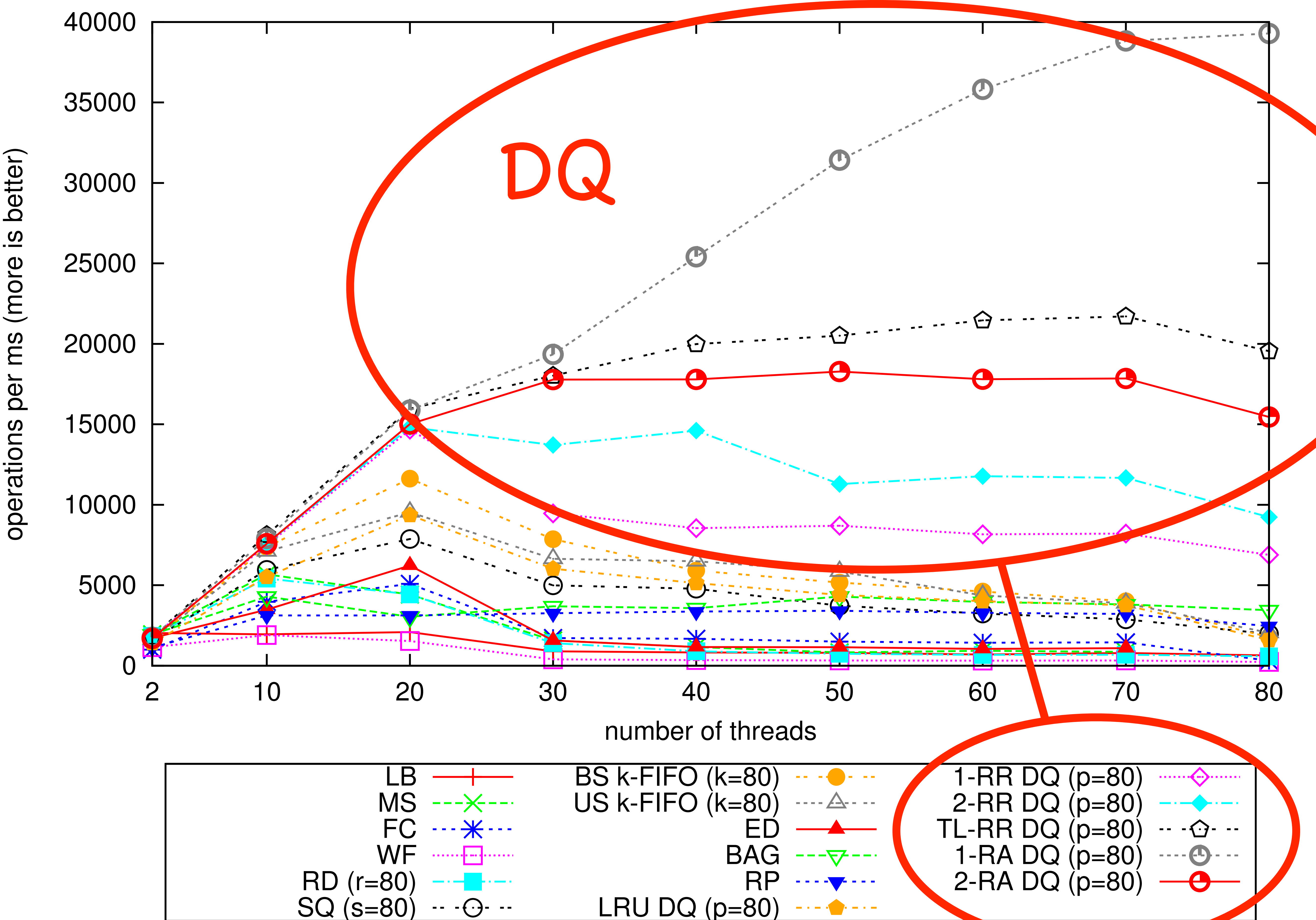
Joint work w/ A. Haas,
M. Lippautz, H. Payer,
A. Sokolova and our
collaborators at IST Austria
T. Henzinger, A. Sezgin

4 processors x 10 cores x 2 hardware threads = 80 hardware threads



Performance & Scalability

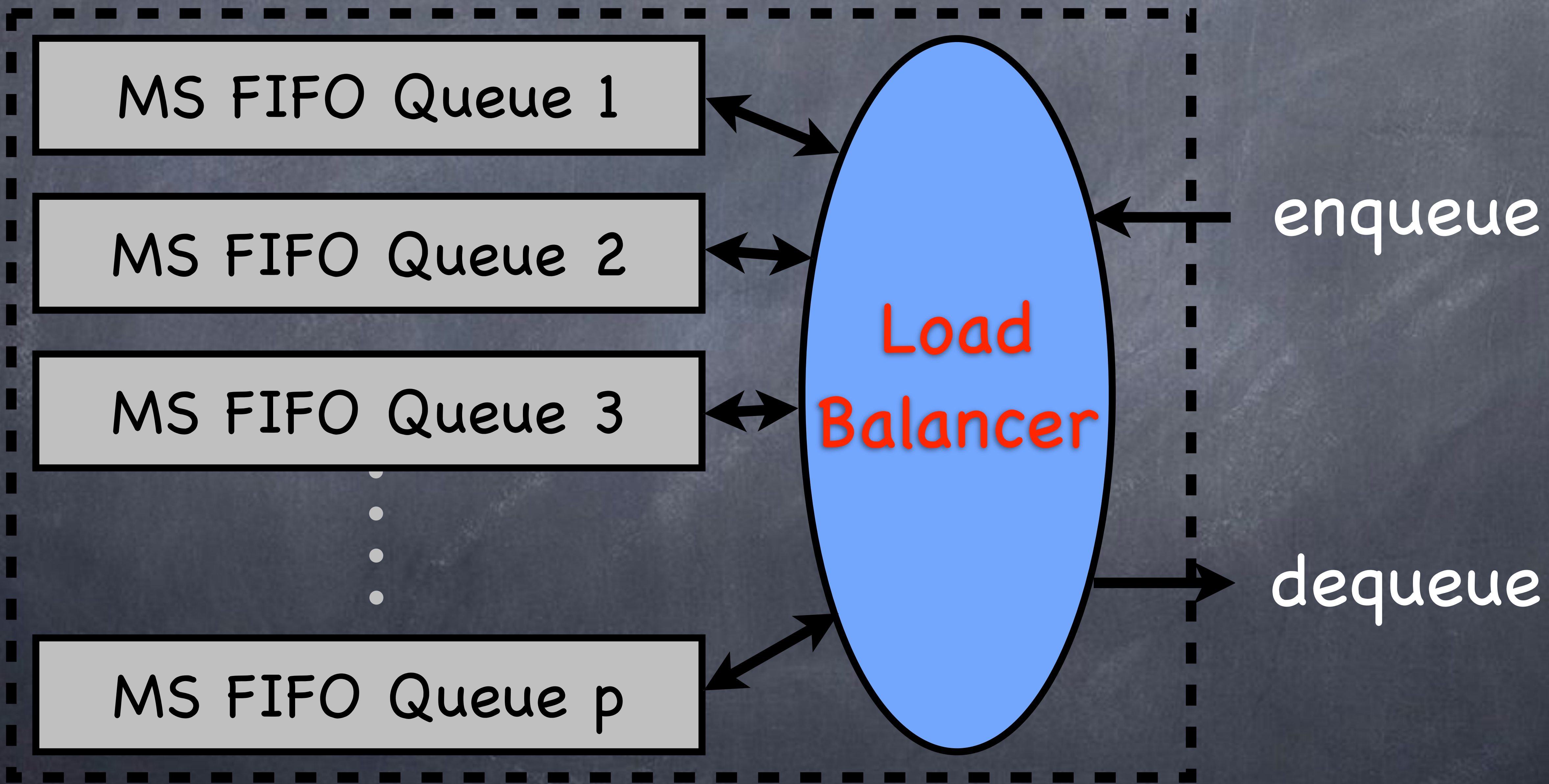




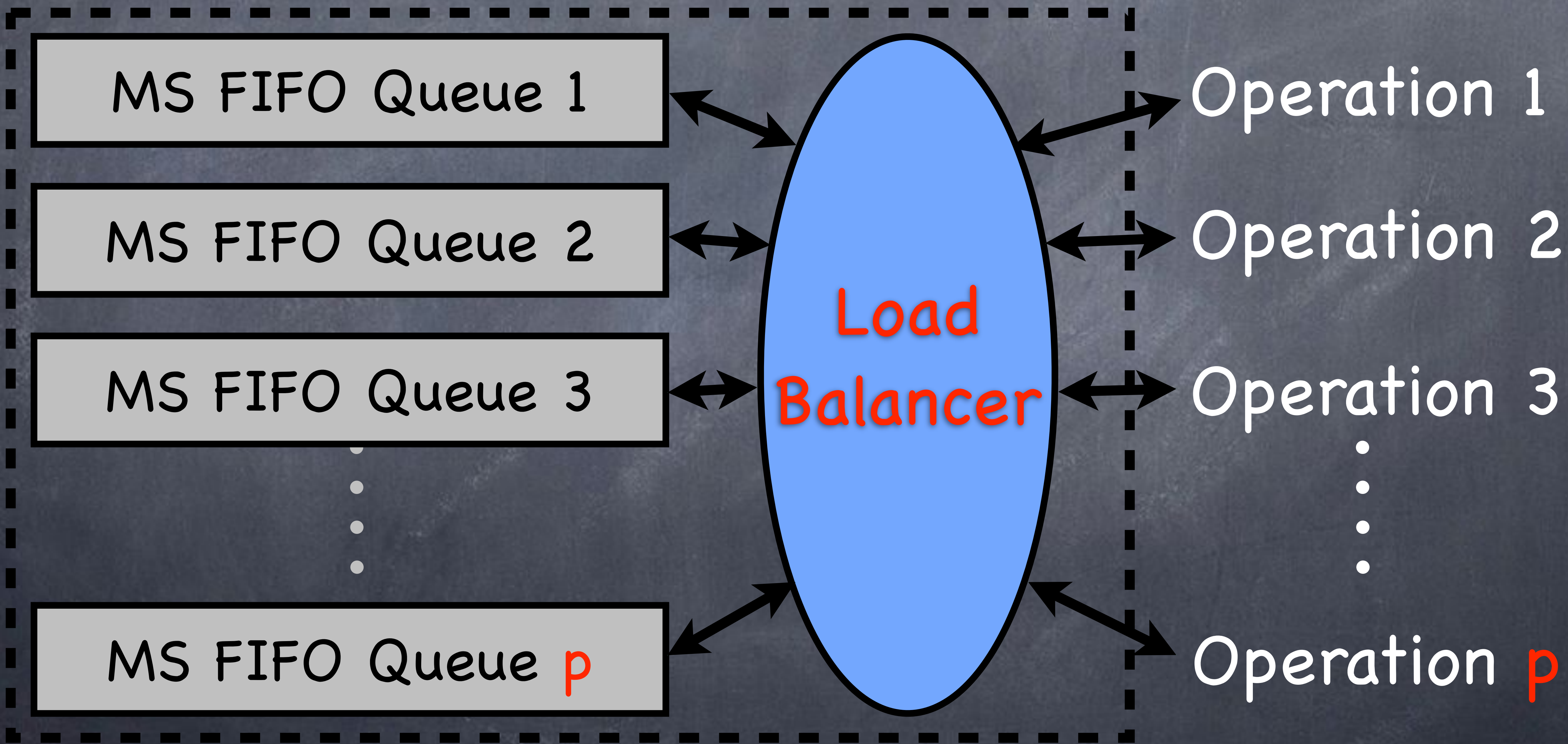
(a) High contention producer-consumer microbenchmark ($c = 250$)

Distributed Queues

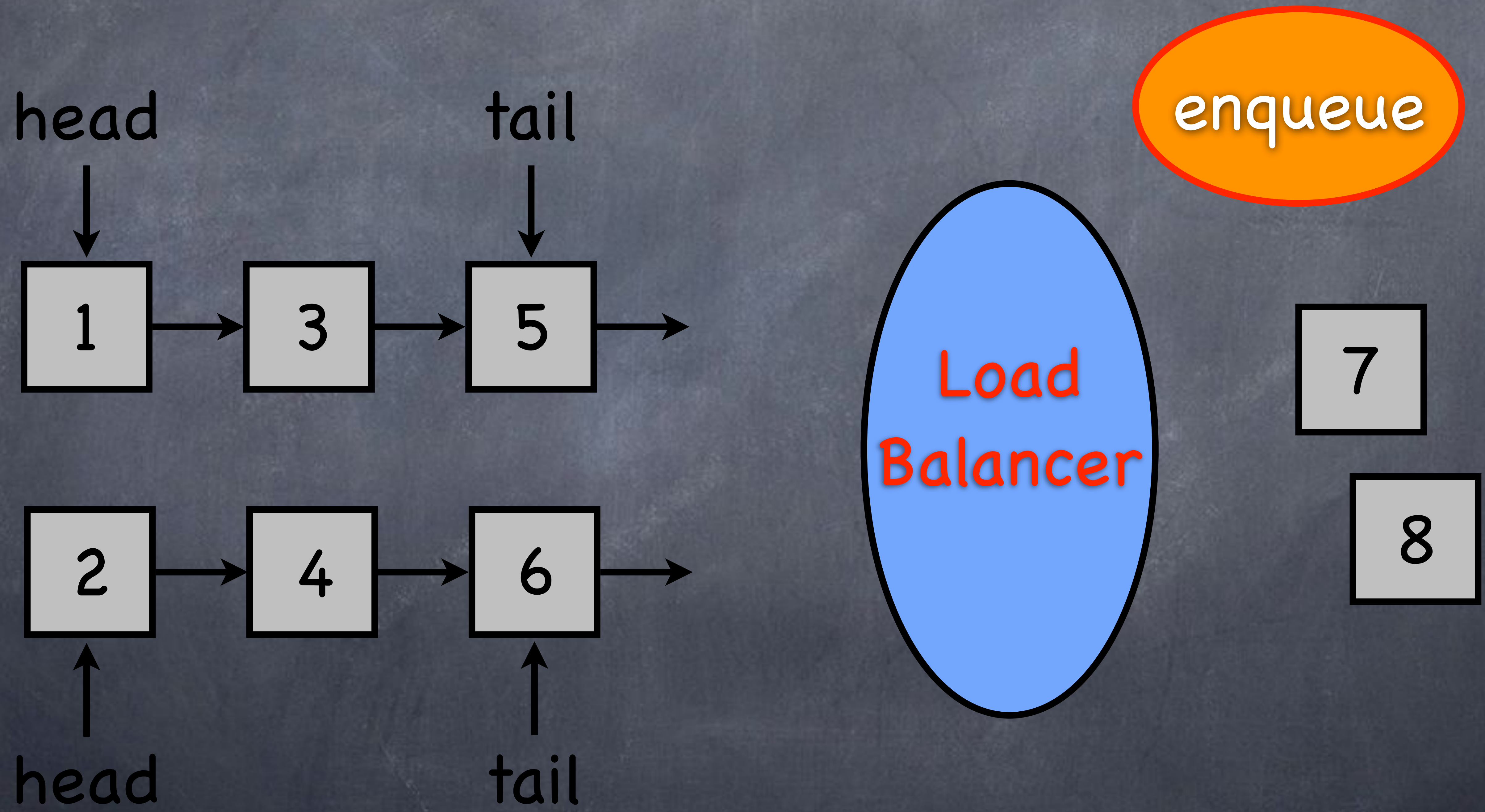
[PODC BA 2011, ICA3PP 2012, Submitted 2013]



Up to p Parallel Enqueues and p Parallel Dequeues



Parallel Access



Emptiness Check?

->

Not Relaxed!

Listing 1: Lock-free load-balanced distributed queue algorithm

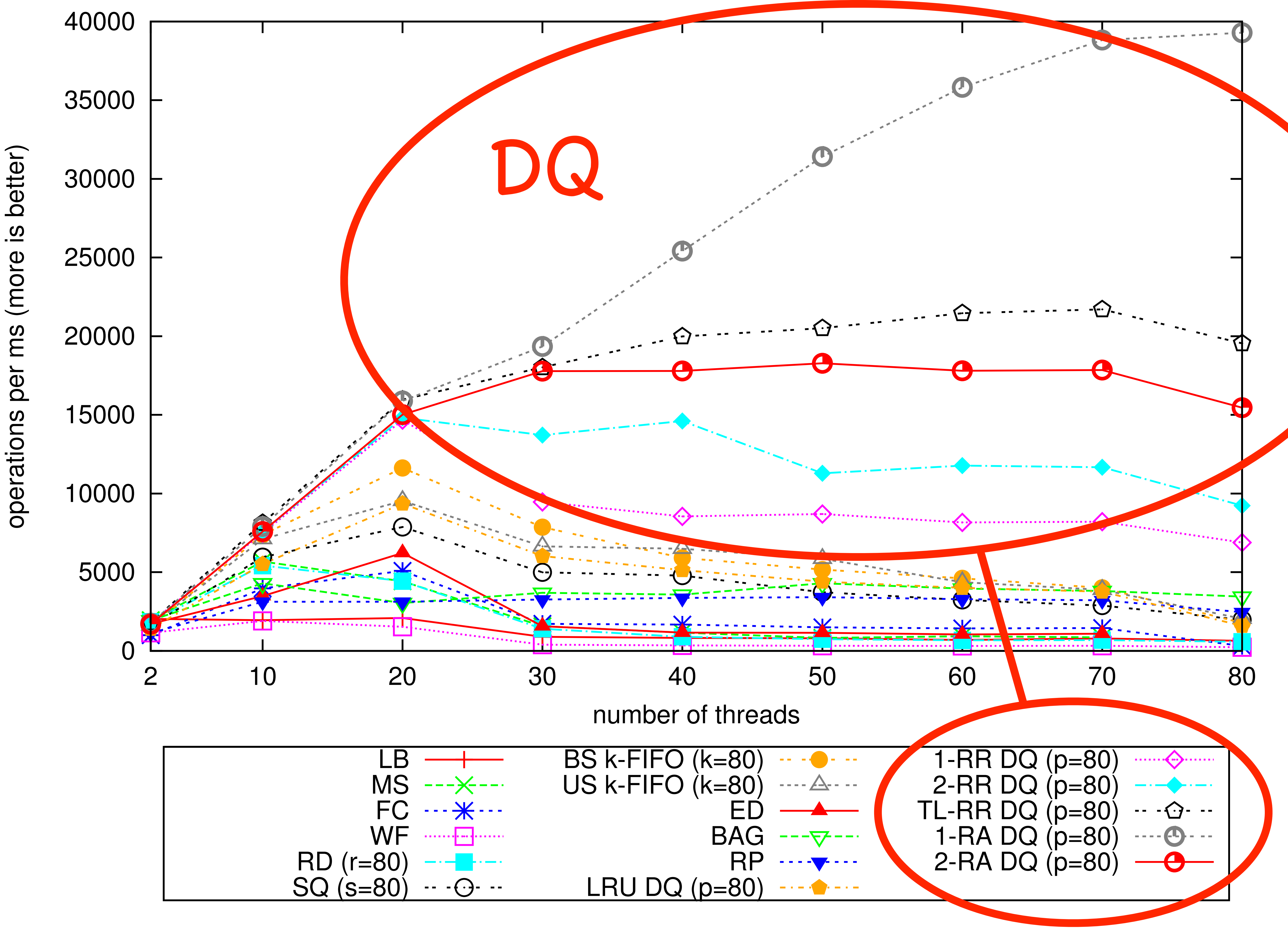
```
enqueue(element):
  index = load_balancer();
  DQ[index].MS_enqueue(element);

dequeue():
  start = load_balancer();
  while true:
    for i in 0 to p-1:
      index = (start + i) % p;
      element, current_tail = DQ[index].MS_dequeue();
      if element != null:
        return element;
      else:
        tail_old[index] = current_tail;
    for i in 0 to p-1:
      if get_tail(DQ[i]) != tail_old[i]:
        start = i;
        break;
      if i == p-1:
        return null;
```

DQ[p]: array of MS queues

tail_old[p]: array of MS tails

1.

2.



(a) High contention producer-consumer microbenchmark ($c = 250$)

Semantics

[Related Work]

Our Stuff

Pools

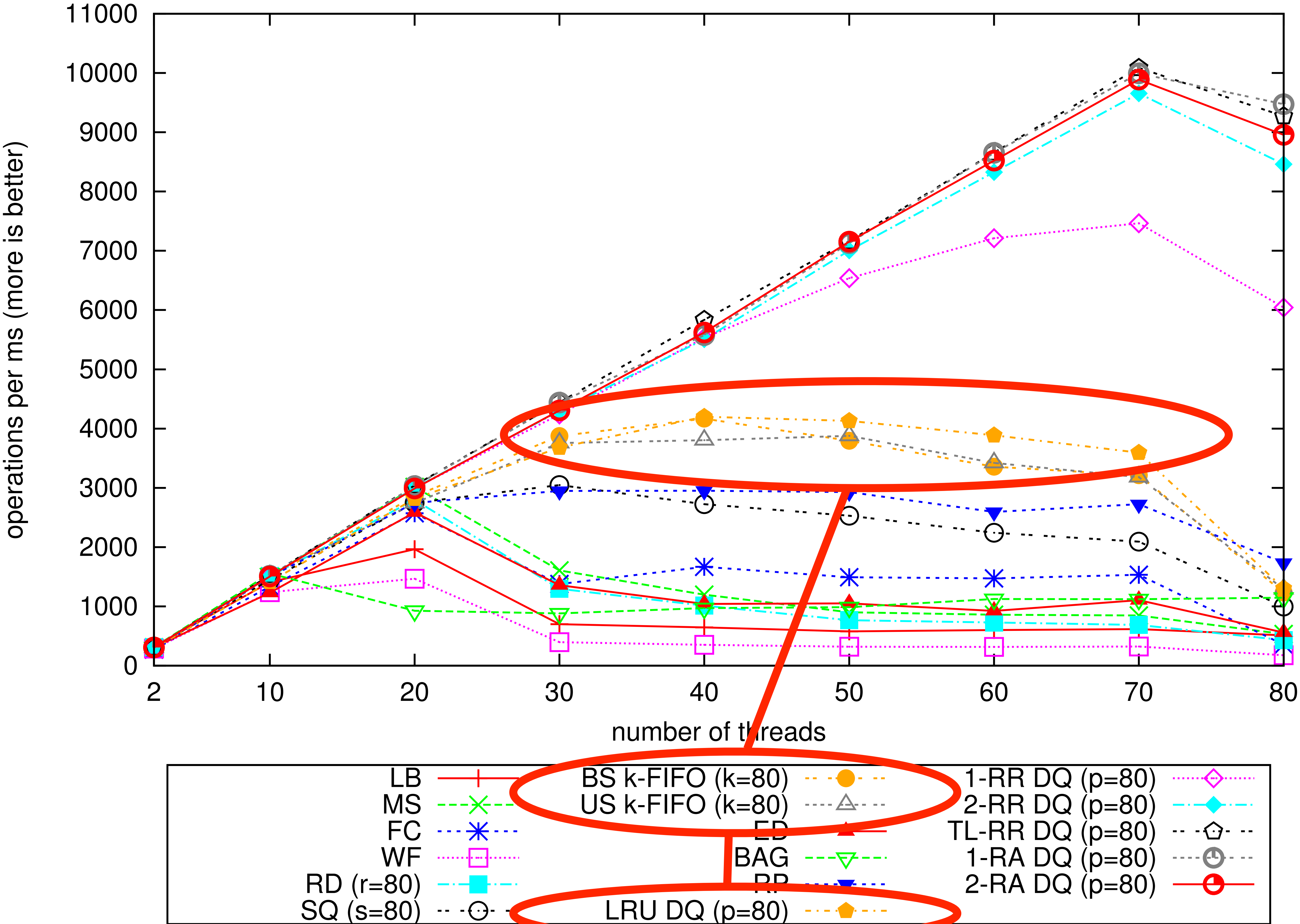
1-RA DQ
2-RA DQ

k-FIFO (k≥0)

TL-RR DQ
2-RR DQ
1-RR DQ

ED
BAG
RP

[Sundell et al.'11]
[Afek et al.'11,'10]



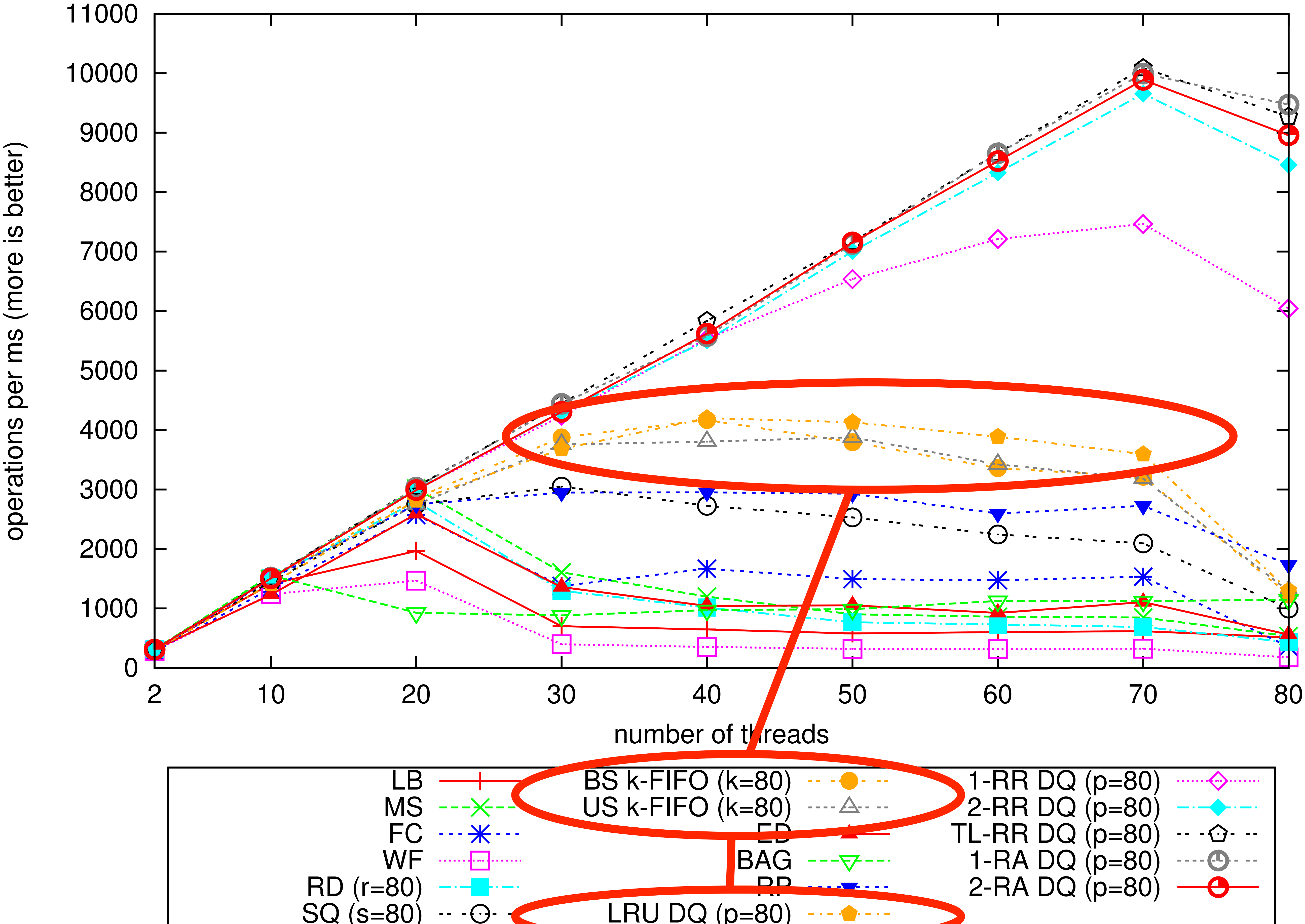
(b) Low contention producer-consumer microbenchmark ($c = 2000$)

Listing 2: Lock-free LRU distributed queue algorithm

```
enqueue(element):
  start = random();
  while true:
    aba_index, aba_count = lowest_aba_tail(start);
    for i in 0 to p-1:
      index = (aba_index + i) % p;
      current_tail = get_tail(DQ[index]);
      if current_tail.aba == aba_count &&
         DQ[index].try_MS_enqueue(element, current_tail):
        return;

dequeue():
  start = random();
  while true:
    aba_index, aba_count = lowest_aba_head(start);
    check_emptiness = true;
    clear(empty_queue);
    for i in 0 to p-1:
      index = (aba_index + i) % p;
      current_head = get_head(DQ[index]);
      if current_head.aba == aba_count:
        element, current_tail =
          DQ[index].try_MS_dequeue(current_head);
        if element == FAILED:
          check_emptiness = false;
        else if element == null:
          tail_old[index] = current_tail;
          empty_queue[index] = true;
        else:
          return element;

    if check_emptiness && there_is_any(empty_queue):
      for i in 0 to p-1:
        if empty_queue[i] &&
           (get_tail(DQ[i]) != tail_old[i]):
          start = i;
          break;
        if i == p-1:
          return null;
```

LRU DQ:

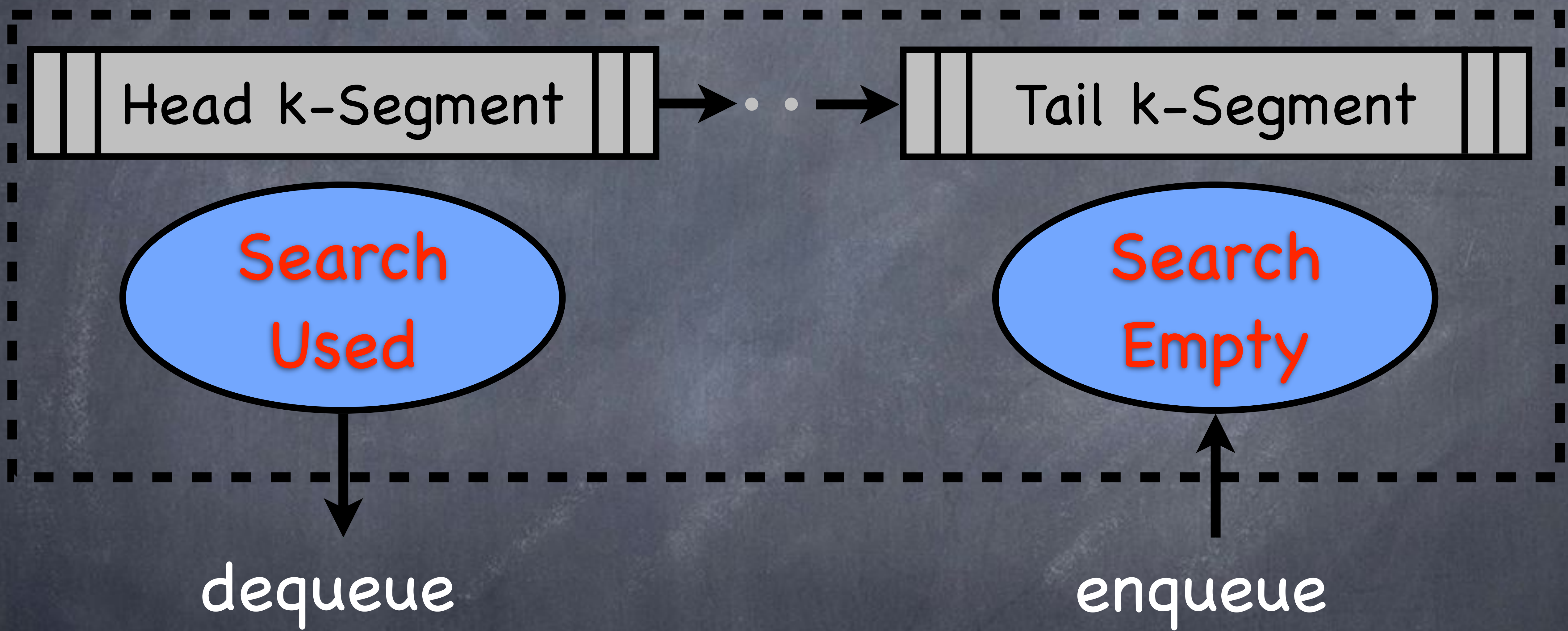
max difference of
tail/head
ABA counters
is one!
->
there are two
partitions of MS queues
with lowest/highest
ABA counters
->
enqueue/dequeue
@one_of_lowest



(b) Low contention producer-consumer microbenchmark ($c = 2000$)

Segmented Queues (SQ)

[Afek,Korland,Yanovsky 2010]



-> BS, US k-FIFO Queues

[_,Lippautz,Payer 2012]

Emptiness Check?

->

Not Relaxed!

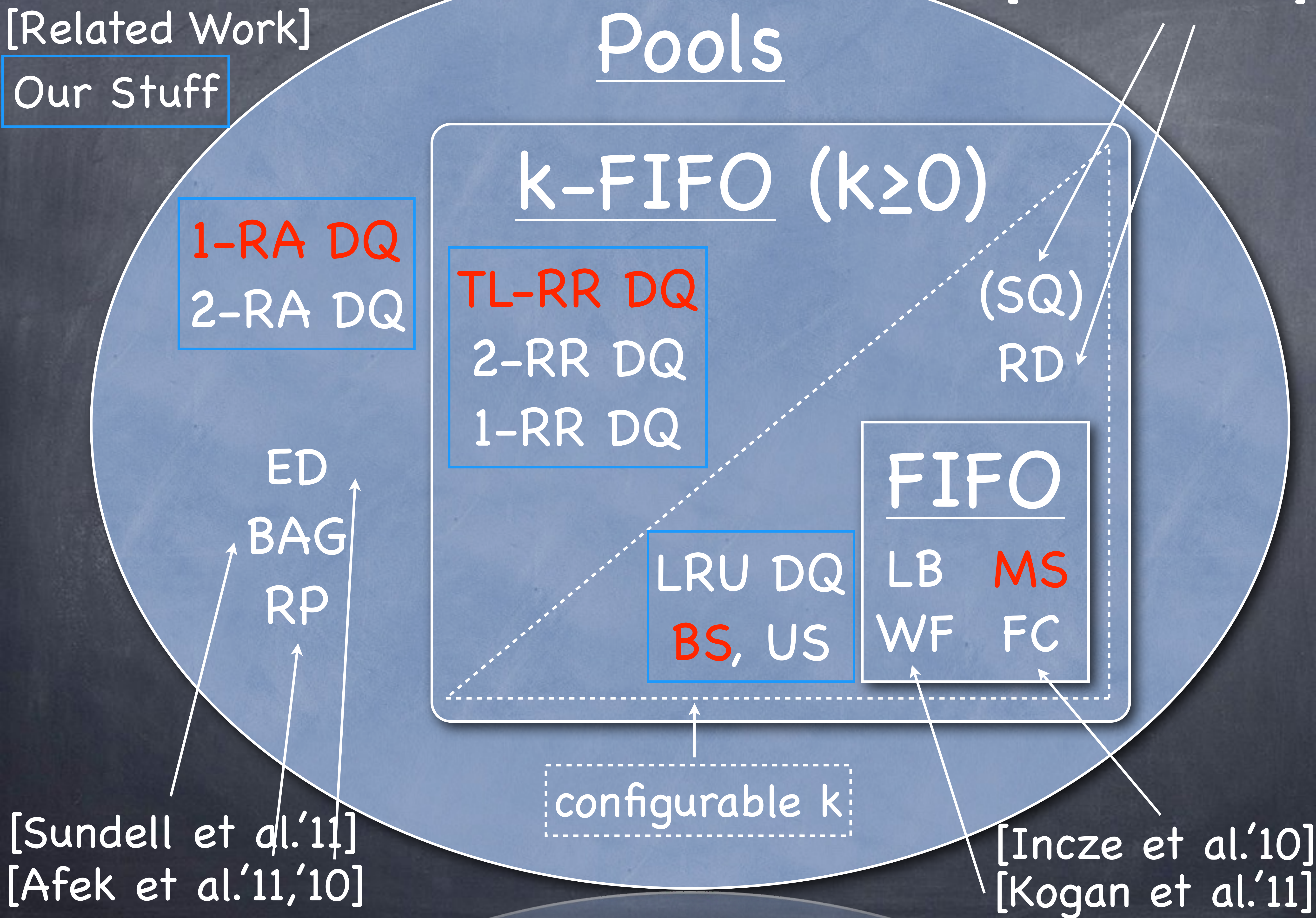
enqueue

```
bool enqueue(item):
  while true:
    tail_old = get_tail();
    head_old = get_head();
    item_old, index = find_empty_slot(tail_old, k, TESTS);
    if tail_old == get_tail():
      if item_old.value == EMPTY:
        item_new = atomic_value(item, item_old.counter + 1);
        if CAS(&tail_old[index], item_old, item_new):
          if committed(tail_old, item_new, index):
            return true;
      else:
        if queue_full(head_old, tail_old):
          if segment_not_empty(head_old, k) && head == get_head():
            return false;
          advance_head(head_old, k);
        advance_tail(tail_old, k);

bool committed(tail_old, item_new, index):
  if tail_old[index] != item_new:
    return true;
  head_current = get_head();
  tail_current = get_tail();
  item_empty = atomic_value(EMPTY, item_new.counter + 1);
  if in_queue_after_head(tail_old, tail_current, head_current):
    return true;
  else if not_in_queue(tail_old, tail_current, head_current):
    if !CAS(&tail_old[index], item_new, item_empty):
      return true;
  else: //in queue at head
    head_new = atomic_value(head_current.value, head_current.counter + 1);
    if CAS(&head, head_current, head_new):
      return true;
    if !CAS(&tail_old[index], item_new, item_empty):
      return true;
  return false;
```

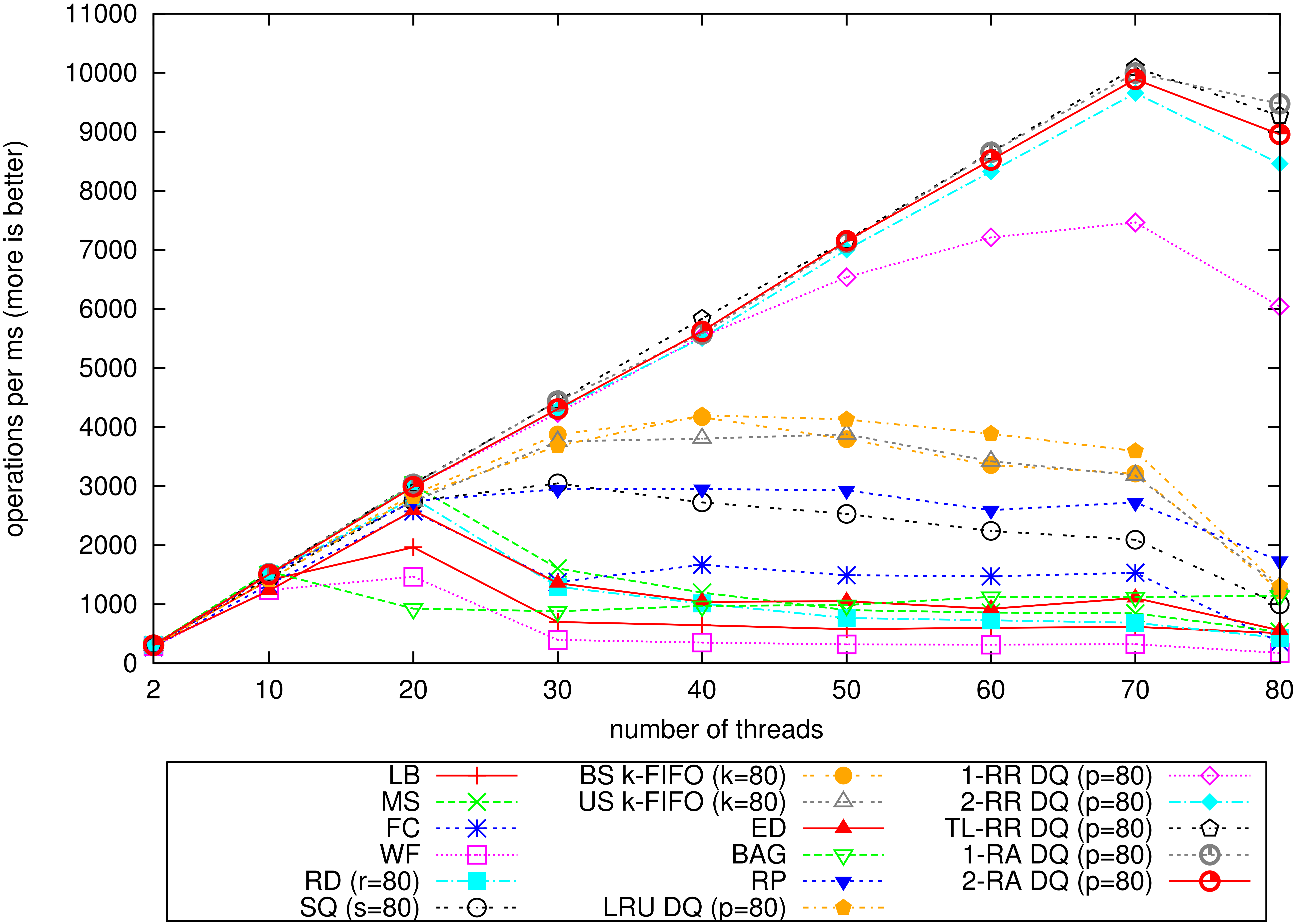
dequeue

```
item dequeue():
  while true:
    tail_old = get_tail();
    head_old = get_head();
    item_old, index = find_item(head_old, k);
    if head_old == head:
      if item_old.value != EMPTY:
        if head_old.value == tail_old.value:
          advance_tail(tail_old, k);
        item_empty = atomic_value(EMPTY, item_old.counter + 1);
        if CAS(&head_old[index], item_old, item_empty):
          return item_old.value;
      else:
        if head_old.value == tail_old.value && tail_old == get_tail():
          return null;
        advance_head(head_old, k);
```

Semantics



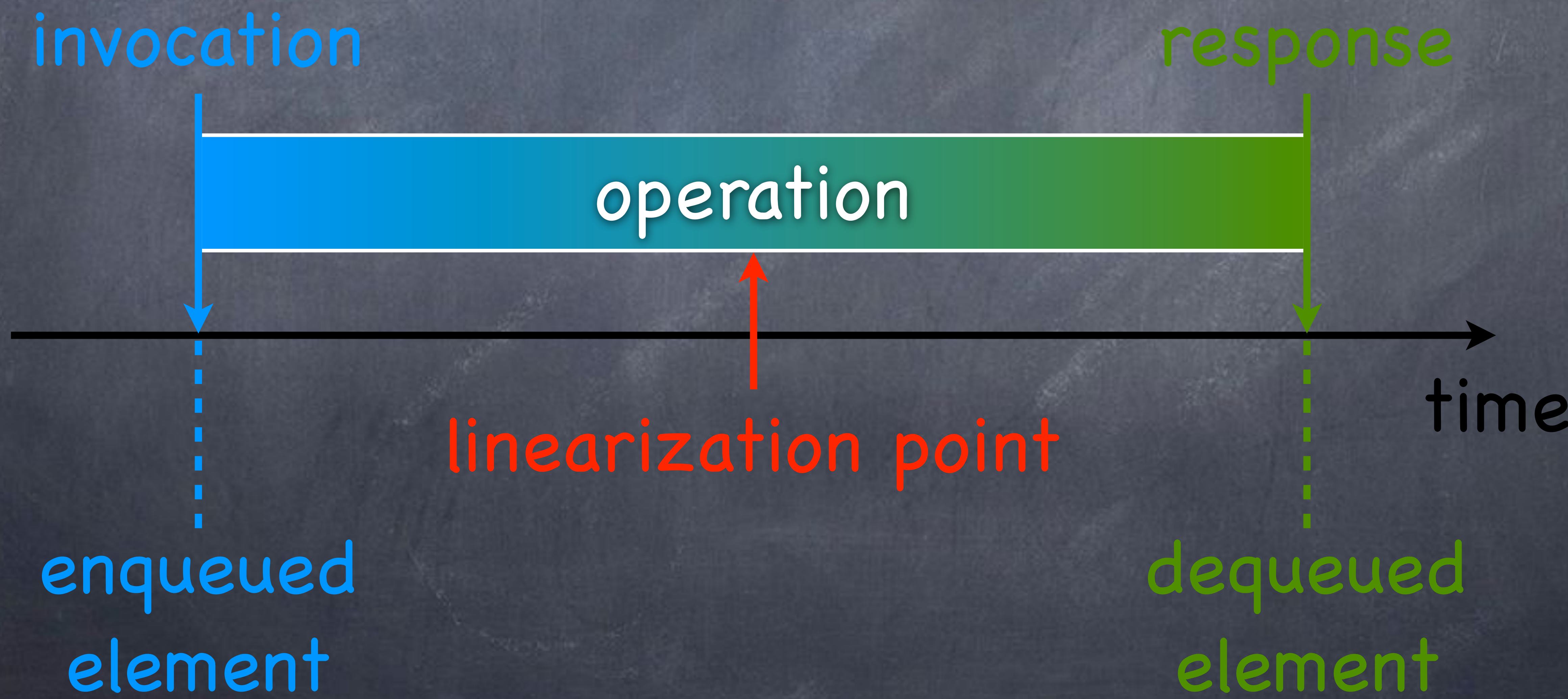
[Related Work]
Our Stuff
[Afek et al.'10]
Pools
k-FIFO (k≥0)
1-RA DQ
2-RA DQ
TL-RR DQ
2-RR DQ
1-RR DQ
(SQ)
RD
ED
BAG
RP
FIFO
LRU DQ
BS, US
LB
MS
WF
FC
configurable k
[Sundell et al.'11]
[Afek et al.'11,'10]
[Incze et al.'10]
[Kogan et al.'11]



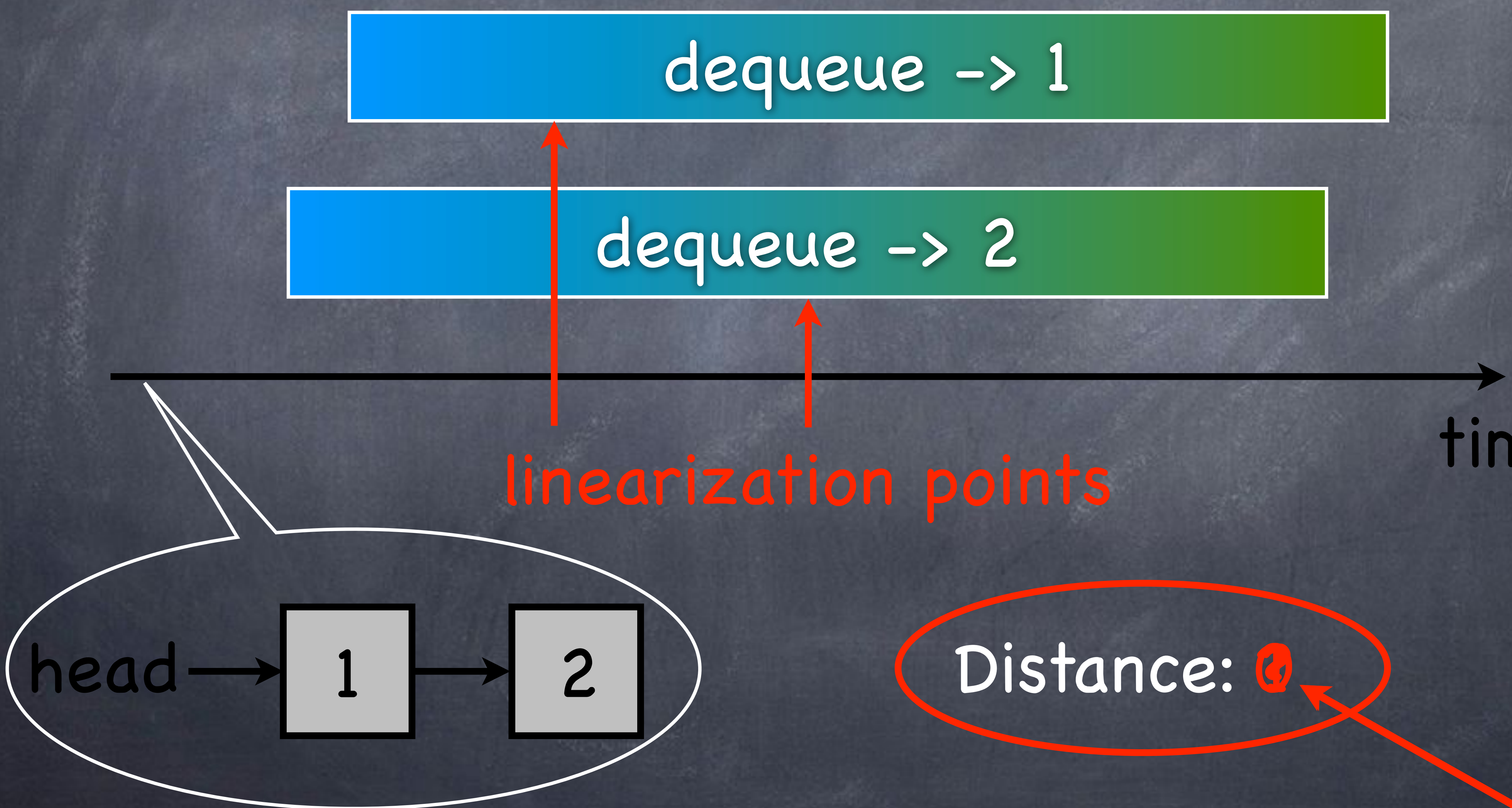
(b) Low contention producer-consumer microbenchmark ($c = 2000$)

(Enhanced) Concurrent History

Sequence of Time-stamped Invocation and Response Events as well as Time-stamped Linearization Points (Approximative)



Measuring "Out-of-Order Distance"



The Actual-Time Linearization

of a concurrent history

is

the sequence of its operations

ordered by

their linearization points

The Actual-Time Distance

of a concurrent history

is

the average

out-of-order distance

of

its actual-time linearization

Actual-Time Distance

measures
re-ordering
due to
semantical relaxation!

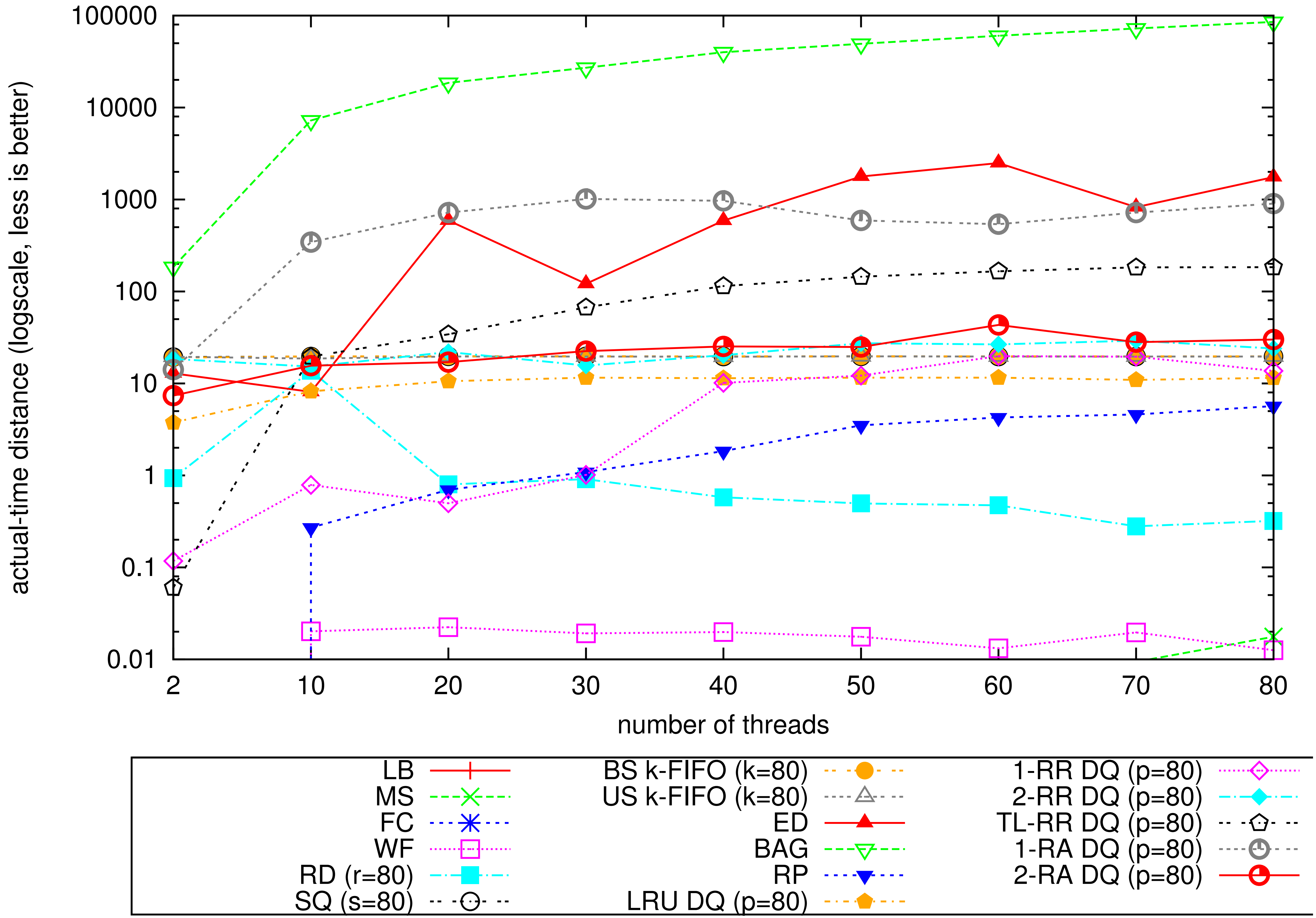
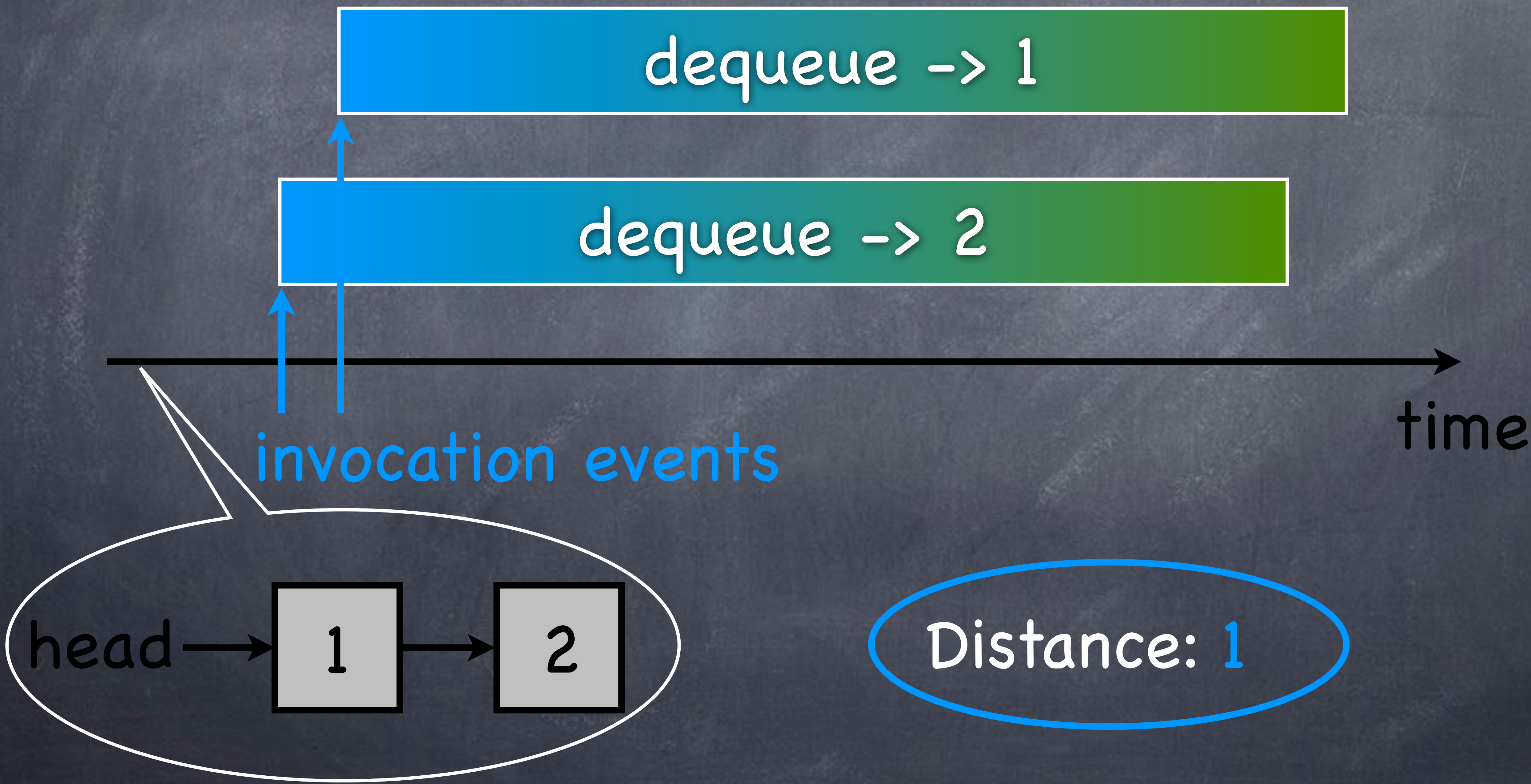


Figure 6: Actual-time distance of the high contention producer-consumer microbenchmark ($c = 250$)

Invocation vs. Linearization



Distance: 1

The Zero-Time Linearization
of a concurrent history
is
the sequence of its operations
ordered by
their invocation events

The Zero-Time Distance

of a concurrent history

is

the average

out-of-order distance

of

its zero-time linearization

Zero-Time Distance

measures

re-ordering

due to

semantical relaxation

and

linearizability!

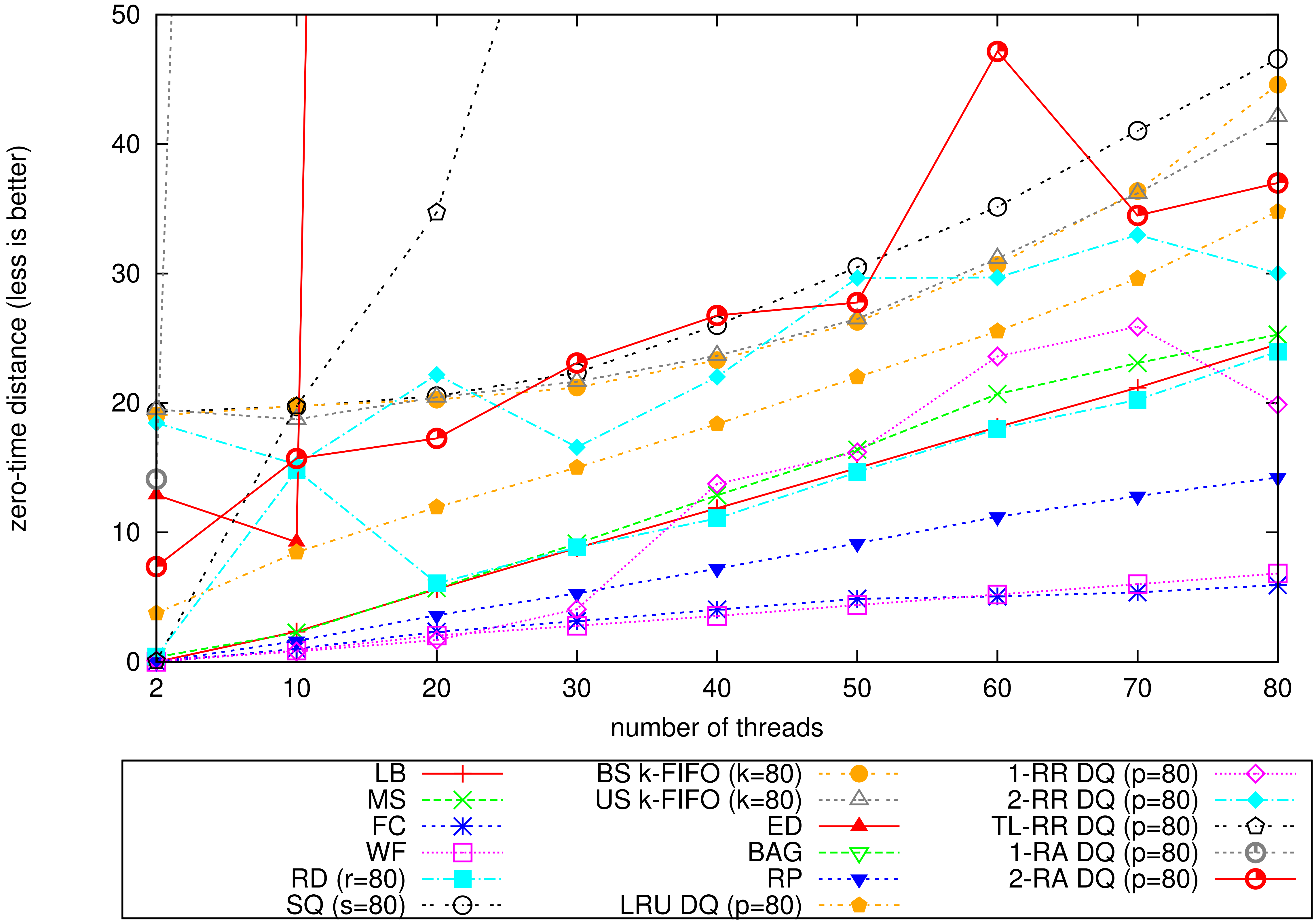


Figure 7: Zero-time distance of the high contention producer-consumer microbenchmark, zoomed-in for better resolution cutting off ED, BAG, TL-RR, and 1-RA ($c = 250$)

Linearization Difference

(difference of zero- and actual-time distance) measures re-ordering due to linearizability!

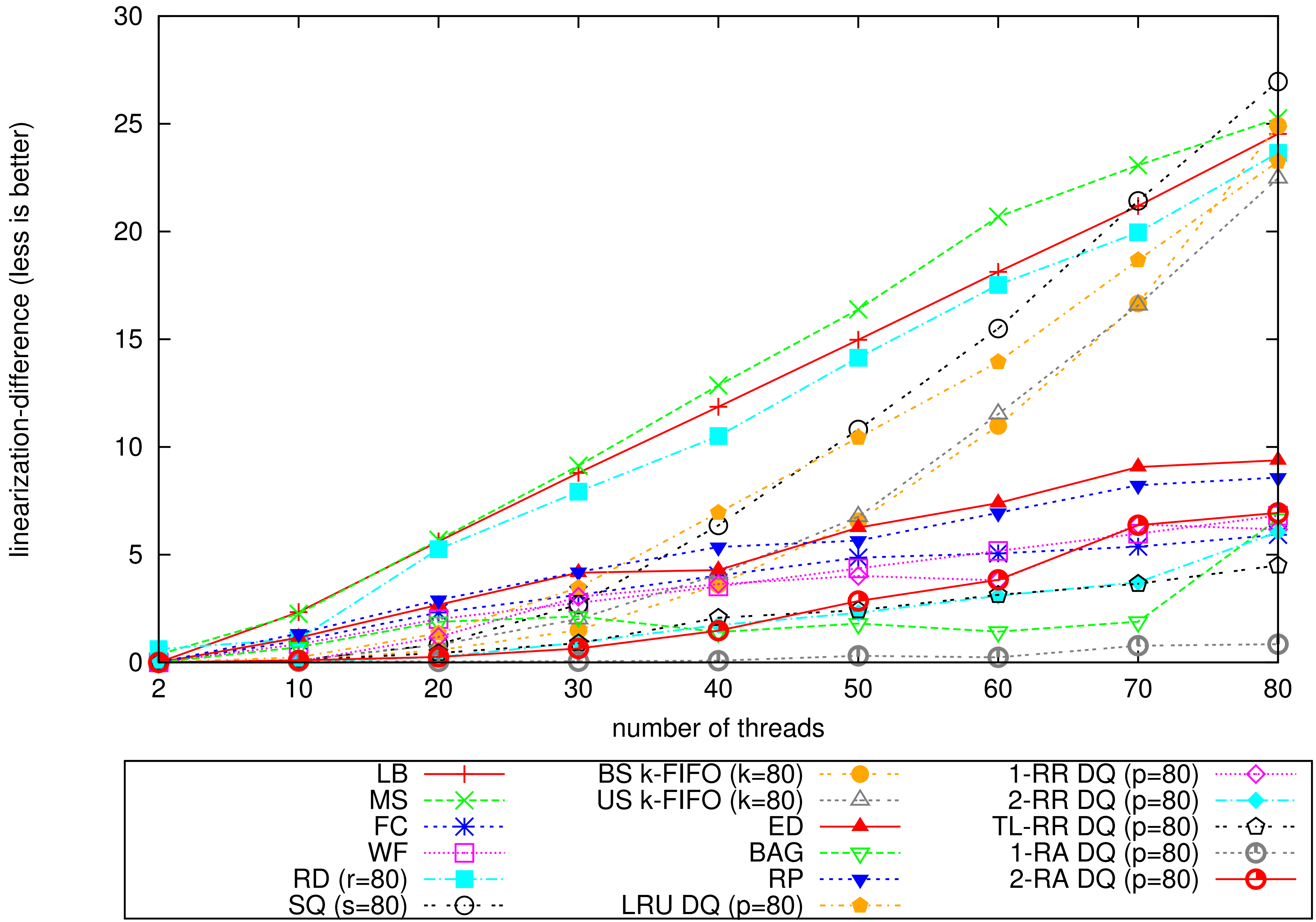
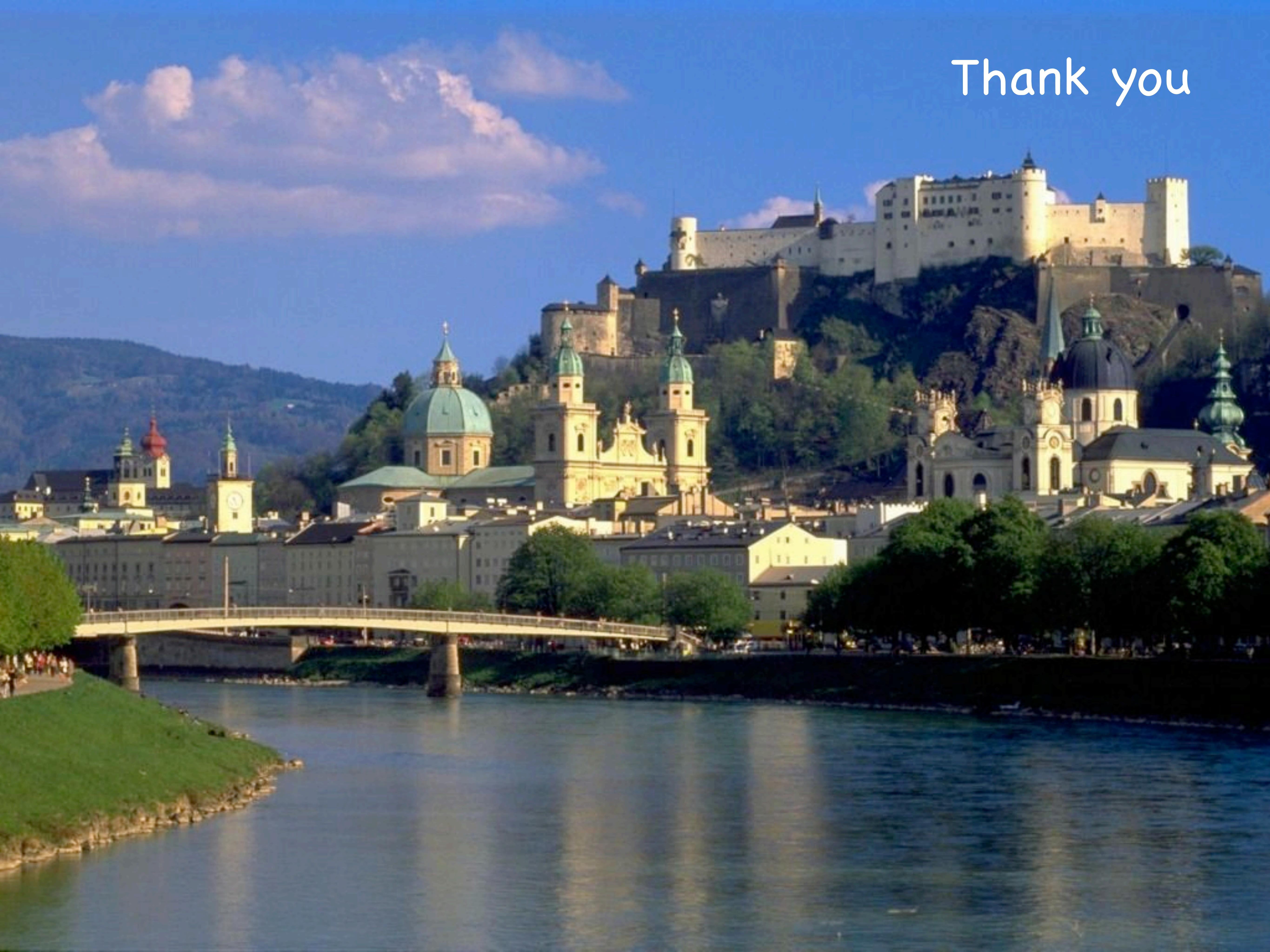


Figure 8: Linearization difference of the high contention producer-consumer microbenchmark ($c = 250$)


Thank you