Multicore Scalability of Concurrent Objects

Hannes Payer

Defensio
University of Salzburg

28/09/2012
Overview

• Quantitatively relaxed concurrent data structures
  
  
  
  
  
  
  
Overview

- **Memory Management**
Multicore Scalability

• Why is multicore scalability an interesting topic?
  • Steadily increasing core count
  • Application performance (throughput, latency) does not necessarily increase with the number of cores
Multicore Scalability

throughput

number of cores

linear scalability
Multicore Scalability

linear scalability

positive scalability

throughput

number of cores
Multicore Scalability

- **Linear scalability**
- **Positive scalability**
- **Negative scalability**

Graph showing throughput vs. number of cores.
Multicore Scalability

- Linear scalability
- Positive scalability
- Negative scalability
- Low performance

throughput vs. number of cores
Multicore Scalability

- Linear scalability
- Positive scalability (high performance)
- Negative scalability
- Positive scalability (low performance)
Research Problem

• Concurrent objects may have scalability bottlenecks under high contention caused by e.g.:
  • Locks
  • Atomic updates of shared state variables
  • Bad caching

• Can we achieve high performance and positive scalability of concurrent objects under high contention?
Motivation

• Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated, POPL’11, Attiya et al.

• Approach: Trade-off performance and scalability versus semantical relaxation of the concurrent object

• We study semantically relaxed versions of concurrent FIFO queues

• Incorrect Systems: It’s not the Problem, It’s the Solution, DAC’12, Kirsch & Payer
k-FIFO Queue

- Elements may be returned out-of-FIFO-order up to k
- Example k=2:

```
head
A B C D E
tail
```
k-FIFO Queue

- Elements may be returned out-of-FIFO-order up to \( k \)
- Example \( k=2 \):

```
A B C D E
```

Head

Tail
k-FIFO Queue

- Elements may be returned out-of-FIFO-order up to k
- Example k=2:
k-FIFO Queue

- Elements may be returned out-of-FIFO-order up to k
- Example k=2:
k-FIFO Queue

- Elements may be returned out-of-FIFO-order up to k
- Example k=2:
k-FIFO Queue

- Elements may be returned out-of-FIFO-order up to k
- Example k=2:
k-FIFO Queue

- Elements may be returned out-of-FIFO-order up to $k$
- Example $k=2$:
k-FIFO Queue

- Elements may be returned out-of-FIFO-order up to $k$
- Example $k=2$:
k-FIFO Queue

- Elements may be returned out-of-FIFO-order up to $k$
- Example $k=2$: 

```
D
```

- head
- tail
k-FIFO Queue

- Elements may be returned out-of-FIFO-order up to $k$
- Example $k=2$: 

![Diagram of a k-FIFO Queue with head and tail markers](image)
k-FIFO Queue

- Elements may be returned out-of-FIFO-order up to $k$
- Example $k=2$:

```
  head
↓
  tail
```
k-FIFO Queue

- Two new metrics:
  - Age: counts the number of elements a given element overtook
  - Lateness: counts the number of elements that overtook a given element
Partitioned k-FIFO Queue
Partitioned k-FIFO Queue

1:   enqueue(element):
2:    while true:
3:      tail_old = get_tail();
4:      item, index = find_empty_slot(tail_old);
5:      if tail_old == get_tail():
6:        if item == EMPTY:
7:          if CAS(tail_old->segment[index], EMPTY, element):
8:            if committed(tail_old, element, index):
9:              return;
10:            else:
11:              advance_tail(tail_old);
Partitioned k-FIFO Queue

1: dequeue():
2:   while true:
3:     head_old = get_head();
4:     item, index = find_item(head_old);
5:     tail_old = get_tail();
6:     if head_old == head_tail():
7:       if item != EMPTY:
8:         if head.old == tail.old:
9:           advance_tail(tail_old);
10:          if CAS(head_old->segment[index], item, EMPTY):
11:             return item;
12:          else:
13:            if head_old == tail_old:
14:              return null;
15:            advance_head(head_old);
Distributed k-FIFO Queue

FIFO Queue

enqueue

dequeue
Distributed k-FIFO Queue

FIFO Queue 1

FIFO Queue 2

...

FIFO Queue p

Load Balancer

enqueue

dequeue
Distributed k-FIFO Queue

enqueue(element):
index = load_balancer();
fifo[index].enqueue(element);
atomic_increment(&fifo[index].size);
Distributed k-FIFO Queue

1: dequeue():
2:   tail_old[p];
3:   start = load_balancer();
4:   while true:
5:     for i in 0 to p-1:
6:       index = (start + i) % p;
7:       element, tail_old[index] = fifo[index].dequeue();
8:       if element != null:
9:         atomic_decrement(&fifo[index].size);
10:        return element;
11:     for i in 0 to p-1:
12:       if fifo[i].tail != tail_old[i]:
13:         start = i;
14:         break;
15:     if i == p-1:
16:       return null;
Distributed k-FIFO Queue Load Balancers

- Global round-robin (RR) $k = \text{number of threads} \times p$
- Thread-local round-robin (TL-RR) $k = \text{number of threads} \times p$
- Random (RA) $k = \text{bounded probabilistically}$
- 2-random (2RA) $k = \text{bounded probabilistically}$
- Hierarchical random (H-RA) $k = \text{bounded probabilistically}$
- Hierarchical 2-random (H-2RA) $k = \text{bounded probabilistically}$
Experiments

Performance high contention producer-consumer benchmark 1:1
Experiments

Age high contention producer-consumer benchmark 1:1
Experiments

Lateness high contention producer-consumer benchmark 1:1
Experiments

Performance low contention producer-consumer benchmark 1:1
Experiments

Age low contention producer-consumer benchmark 1:1
Experiments

Lateness low contention producer-consumer benchmark 1:1
Experiments

Mandelbrot producer-consumer benchmark 1:4
Conclusions

• Relaxed semantics of concurrent objects may be key to provide better performance and scalability on multicore systems

• We already obtained promising results also for other concurrent data structures such as stacks

• Future work: other concurrent objects, transactional memory, ...