

# Scalable Producer-Consumer Pools based on Elimination-Diffraction Trees

Y. Afek   G. Korland   M. Natanzon   N. Shavit

European Conference on Parallel Processing (Euro-Par) 2010

Presented by

Michael Lippautz

`mlippautz@cs.uni-salzburg.at`

Concurrency and Memory Management Seminar

Department of Computer Sciences, University of Salzburg

# Contents

What is this about?

The parts

- Balancer

- Diffraction tree

- Elimination

- Elimination-diffraction (ED) tree

Benchmarks and results

Summary and Discussion

# What is this about?

## Problem

Provide a pool implementation that scales and performs well in low and high contention scenarios.

## Author's solution

Build a lock-free distributed pool structure based on *elimination* and *diffraction* paradigms.

The description is general, but the implementation is based on Java.

## Further structure

1. Balancer
2. Diffracting tree
3. First step towards an elimination-diffraction (ED) tree
4. Elimination
5. ED tree

# Balancer

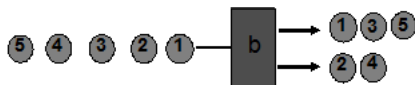


Figure: Balancer [1]

- ▶ Single input wire
- ▶ Two output wires (top, bottom)
- ▶ Items arriving at input wire are sent to top or bottom output wire
- ▶ Fair: Top wire count is the same or at most one more

Implementation: `compareAndSet` (toggle bit)

## Diffracting tree

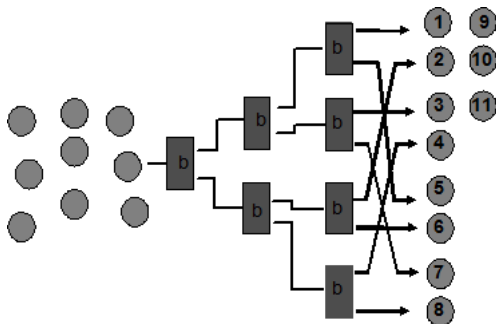


Figure: A diffracting tree  $Tree[3]$  [1]

"The  $Tree[k]$  network of width  $k$  is a binary tree of balancers constructed inductively by taking two  $Tree[k/2]$  networks of balancers and perfectly shuffling their outputs." [2]

# Diffracting tree

## Step property (fairness)

In any quiescent state, for output wires  $y_1 \dots y_n$ :

- ▶ Either  $\forall i, j : \#y_i = \#y_j$
  - ▶ Or  $\exists c : i \leq c < j \wedge \#y_i - \#y_j = 1$
- 
- ▶ (Other words:) Upper wires will always have number of items as the bottom ones, or at most one more.
  - ▶ Diffracting tree may be used to balance access to other data structures.

# Diffracting fairness - how?

- ▶ Diffracting tree is a counting tree
- ▶ Remember: Balancers are fair (CAS toggle)
- ▶ Each level gets a significance value:  $2^{level}$
- ▶ Balancers may account for traversing with the level's significance
- ▶ Items traverse tree recursively

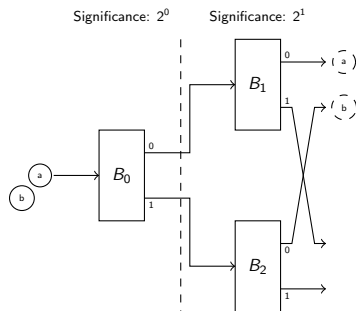


# Diffracting fairness - how?

- ▶ *toggle()* returns toggle bit that was set before toggling

Example: Assume a toggles at  $B_0$  before b

- ▶ Outputwire for a =  $2^0 \cdot \text{toggle}(B_0) + 2^1 \cdot \text{toggle}(B_1) = 0$
- ▶ Outputwire for b =  $2^0 \cdot \text{toggle}(B_0) + 2^1 \cdot \text{toggle}(B_2) = 1$



## Diffraction tree – a first step

- ▶ Add lock-free queues to output wires of the tree

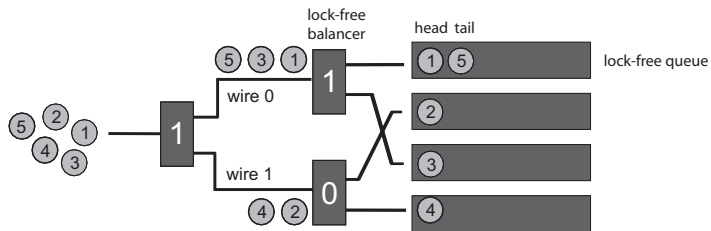


Figure: Diffraction tree with queues as output [2]

# Diffracting tree

## Obvious drawbacks

- ▶ Lots of contention on the root balancer
- ▶ Also lots of contention on balancers further down the tree (static size)

# Elimination

## Observation

An even number of traversals doesn't change a balancers state.

## Elimination array

- ▶ Two meeting pop ops will not touch the balancer
- ▶ Two meeting push ops will not touch the balancer
- ▶ A push and a pop will cancel out

Parameters: Array size, trials, timeout, slot index range

## Putting it all together: ED tree

- ▶ Each balancer now has a producer & a consumer toggle

Example: 1, 2, 3, 4, 5 already in pool (toggles set accordingly)

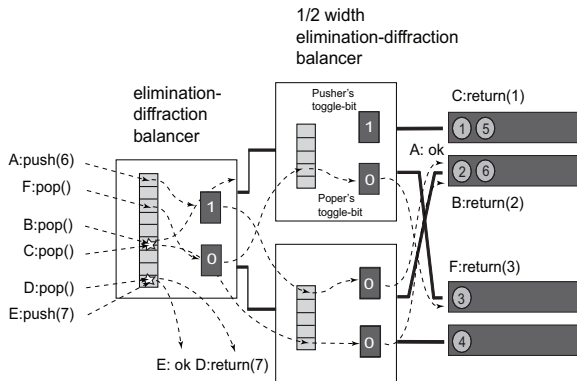


Figure: ED tree [2]

# Benchmarks

- ▶ Producer/consumer benchmark
- ▶ Different queues get plugged into the ED tree
- ▶ Performance and scalability is compared against these queues
- ▶ Queues:
  - ▶ Synchronous
  - ▶ Blocking (Java: `LinkedBlockingQueue`)
  - ▶ Non-blocking (Java: `ConcurrentLinkedQueue`; Michael-Scott queue)

(Different queues provide different pool semantics. → Discussion)

# Results (excerpt)

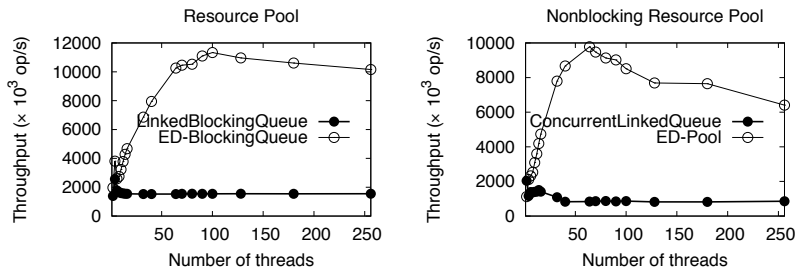


Figure: Throughput over threads for blocking and non-blocking queues in ED trees [2]

# Results (excerpt)

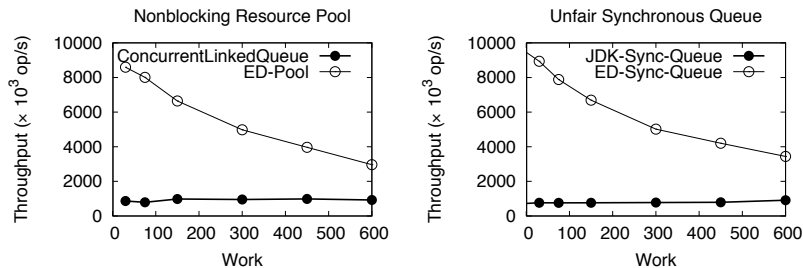


Figure: Throughput over workload; different queues [2]



# Results (excerpt)

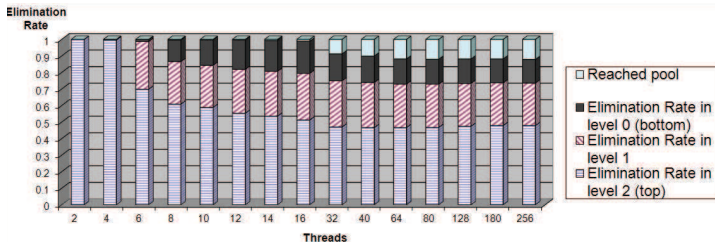


Figure: Elimination rate by levels [2]

# Summary

## ED trees

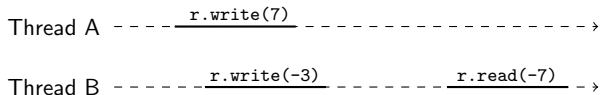
- ▶ ... are structures to distribute access to queues
- ▶ ... are lock-free
- ▶ ... apparently scale well against Java queues

## Interlude: quiescent consistency

### Informally

An implementation of a sequential specification is quiescent consistent if method calls separated by a period of quiescence appear to take effect their in real-time order.

Example:



**Figure:** This implementation is not quiescent consistent, because we would expect either 7 or  $-3$ , not a mixture of both

## Interlude: linearizability

### Informally

An implementation of a sequential specification is linearizable if method calls that overlap in time can take effect in arbitrary order whereas method calls that do not overlap in time have to happen in sequential order.

Example:

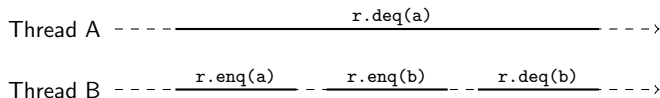


Figure: A linearizable implementation of a FIFO queue

## Interlude: sequential specification pool

### put

$$put(e)(p) = p \cdot e$$

- ▶ A method call *put* on a pool  $p$  with an element  $e$  puts the element into the sequence of elements of the pool.

### get

$$get(e)(p) = \begin{cases} p & \text{if } e = NULL, p = \varepsilon \\ (e_1 \dots e_{i-1} e_{i+1} \dots e_n) & \text{if } e = e_i, q = e_1 \dots e_n \\ error & \text{otherwise} \end{cases}$$

- ▶ A method call *get* on a pool  $p$  with an element  $e$  removes the element  $e$  from the sequence of items in the pool, if the pool is not empty.
- ▶ If the pool is empty, it returns *NULL*. The pool is not changed.

# Properties


## The good

- ▶ Structure allows a dynamic number of threads
- ▶ Apparently performs and scales well (trusting figures)

## The bad

- ▶ Lots of parameters (elimination)
- ▶ Not quiescent consistent (if used with non-blocking queues)<sup>1</sup>
- ▶ Empty check not linearizable<sup>1</sup>
- ▶ Semantic changes depending on plugged-in queue

---

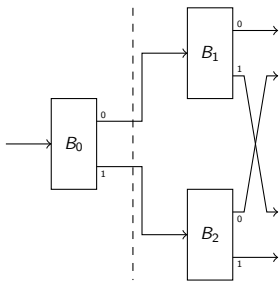
<sup>1</sup>Considering the previously mentioned sequential spec. 

## Semantics – empty check

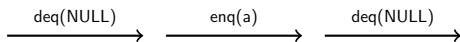
The arguments refer to the previously mentioned sequential specification.

- ▶ Semantics of the empty check depends on queue
- ▶ Blocking queue: no empty check (dequeue returning NULL) at all
- ▶ Non-blocking queue: empty check is not linearizable (not even quiescent consistent)

## Empty check: quiescent consistency, linearizability



The following sequence can happen with non-blocking queues plugged into the ED tree:



**Not linearizable! Not quiescent consistent!** <sup>2</sup>

---

<sup>2</sup>Regarding the previously mentioned sequential specification.



# References



M. Natanzon.

Building scalable producer-consumer pools based on elimination-diffraction trees.

Master's thesis, Tel Aviv University, July 2010.



Y. Afek, G. Korland, M. Natanzon, and N. Shavit.

Scalable producer-consumer pools based on elimination-diffraction trees.

In *Proc. European Conference on Parallel Processing (Euro-Par)*, pages 151–162. Springer, 2010.