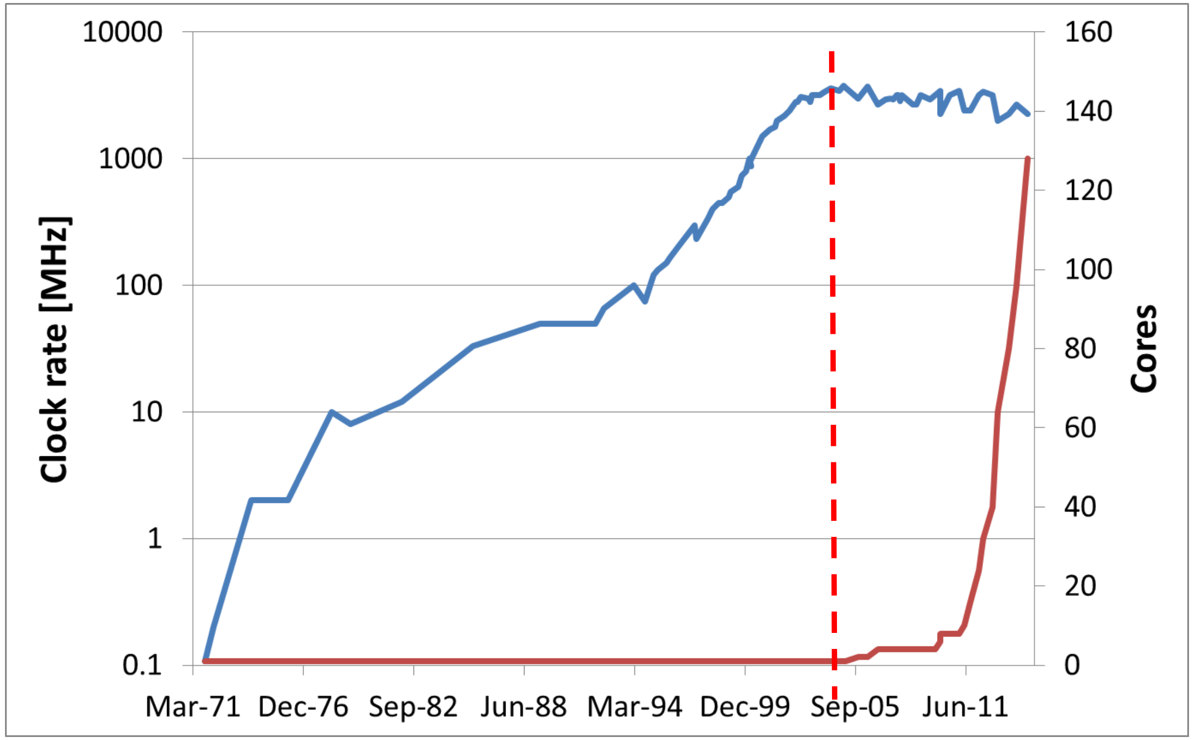
Data Structures of the Future:
Concurrent, Optimistic, and Relaxed

Dan Alistarh

ETH Zurich / IST Austria

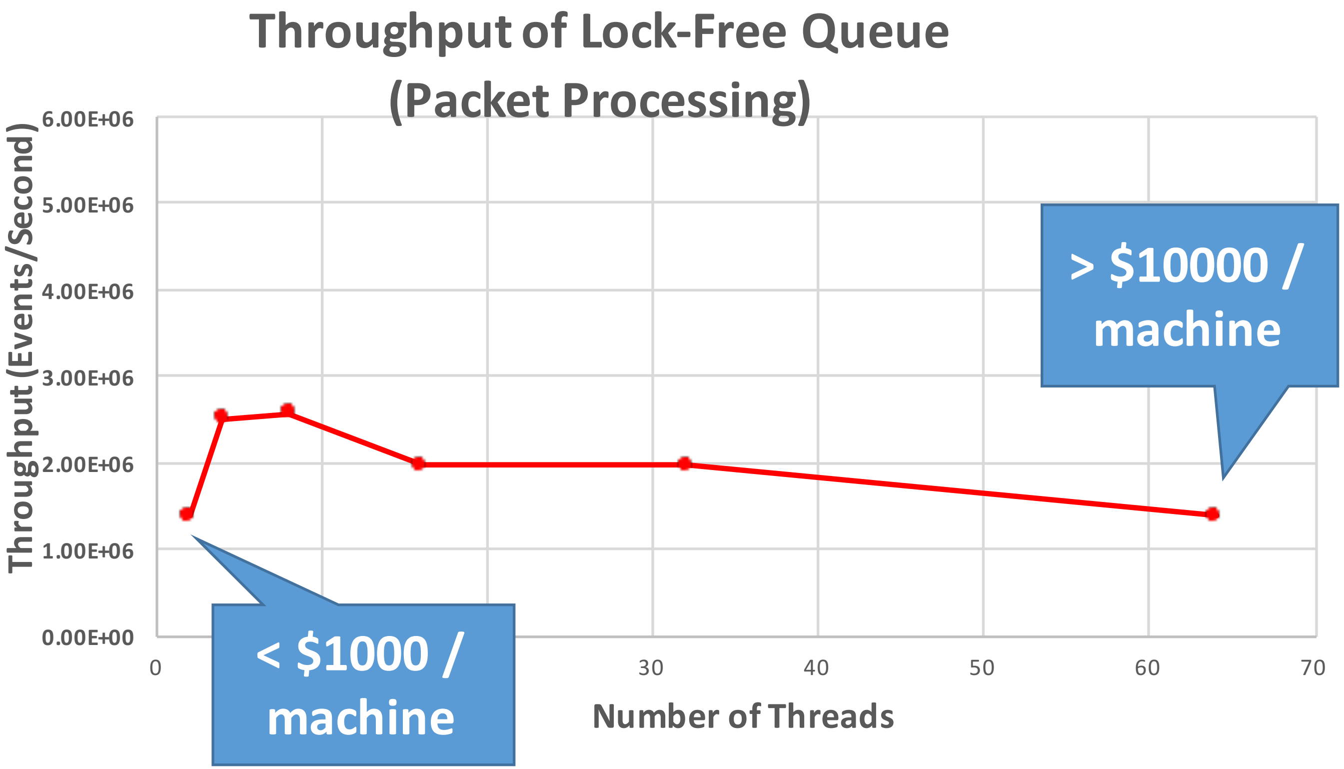
Why *Concurrent*?



Simple: **To get speedup on newer hardware**.

Scaling: more ***threads*** should imply more ***useful work.***

The Problem with Concurrency



Concurrency can be ***very bad value for money.***

Is this problem *inherent*?

Inherent Sequential Bottlenecks

Data structures with **strong ordering semantics**

- Stacks, Queues, Priority Queues, Exact Counters

Theorem: Given **n threads**, **any deterministic**, **strongly ordered** data structure has an execution in which a processor takes ***linear in n time*** to return.
[Ellen, Hendler, Shavit, SICOMP 2013]
[Alistarh, Aspnes, Gilbert, Guerraoui, JACM 2014]

This is **bad news** because of **Amdahl's Law**

To get **performance**, it is **critical** to **speed up shared data structures**.

Today’s Talk

<u>Theorem</u>: Given **n threads**, **any deterministic**, **strongly ordered** data structure has an execution in which a processor takes ***linear in n time*** to return.

[Ellen, Hendler, Shavit, SICOMP 2013]

[Alistarh, Aspnes, Gilbert, Guerraoui, JACM 2014]

How can we scale such data structures?

Theory ↔ Software ↔ Hardware

New **Hardware Instructions**!

New **Data Structure Designs**!

Lock-Free Data Structures 101

- **Optimistic programming patterns**
  - Do not use locks, but atomic instructions (Compare&Swap)
- *Blocking of one thread shouldn't stop the whole system*
- Lots of implementations: HashTables, Lists, Trees, Queues, Stacks, etc.

```
Memory location R;
void fetch-and-increment ( ) {
        int val;
        do {

                val = Read( R );
                new_val = val + 1;

        } while (! Compare&Swap ( &R, val, new_val ));
        return val;
  }
```

Example: Lock-free counter.

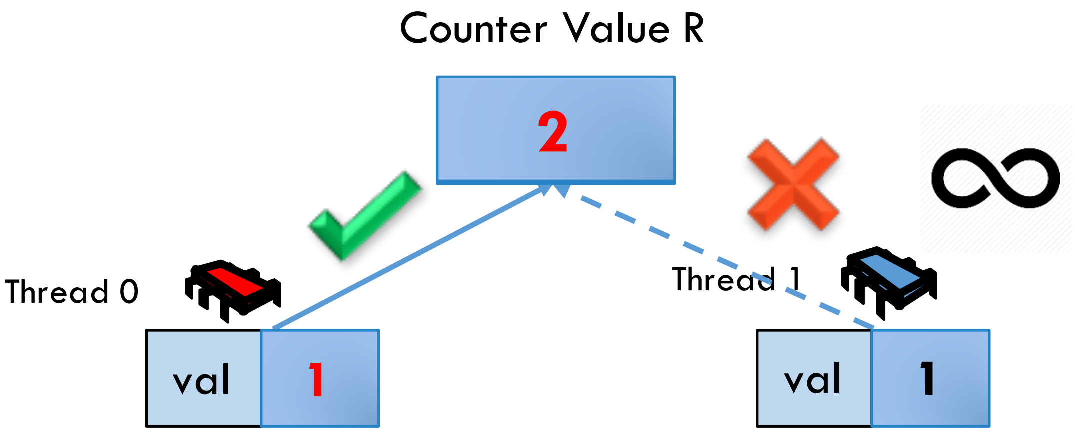
The Lock-Free Paradox

```
Memory location R;
void fetch-and-increment ( ) {
        int val;
        do {

                val = Read( R );
                new_val = val + 1;

        } while (! Compare&Swap ( &R, val, new_val ));
        return val;
  }
```

Example: Lock-free counter.



In theory, threads could ***starve*** in **optimistic lock-free implementations**.

Use **more complex *wait-free*** algorithms.

Practice: this **doesn't always happen**. Threads **rarely starve**.

Why?

Analyzing Lock-Free Patterns

- ***Stochastic Scheduler*** [STOC14, Transact15]*:*
  - *At each scheduling step, the next scheduled thread picked from a distribution* $\mathbf{p} = (\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_n})$ with $\mathbf{p_i} > \mathbf{0}$ for all $\boldsymbol{i}$

Stochastic Scheduler + **Lock-Free Algorithm** = **Stochastic Contention Game**

Theorem 1: Under **any stochastic scheduler**, any **lock-free algorithm** is **wait-free** with probability 1*.* [Alistarh, Censor-Hillel, Shavit, STOC 14 / JACM 16].

Theorem 2: Under **high contention**, roughly **one in** $\mathbf{\Theta}\,(\,1\,/\,\mathbf{norm}_2(\,\boldsymbol{p}\,))$ **ops** succeeds. [Alistarh, Sauerwald, Vojnovic, PODC 15]

The Contention Game

Distribution

$(p_1, p_2, \ldots, p_n)$

Location
R
***Value* = 2**

READ (R)
CAS (R, old, old + 1)
success

READ (R)
CAS (R, old, old + 1)
success

READ (R)
CAS (R, old, old + 1)
success

READ (R)
CAS (R, old, old + 1)
success

Given arbitrary ***p***, what is the *stationary behaviour* of this system?

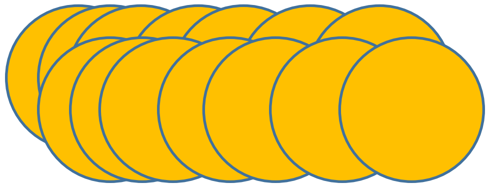
The **Contention Game,** Balls&Bins view

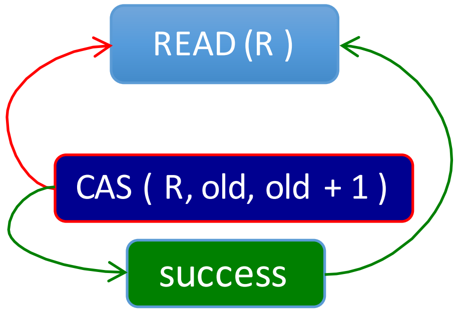
Rules for the Counter

- **Bins = threads**
- **Balls = steps**
- **Placement according to *p***
- **To Complete the Operation**
  - **3 balls before others**
- **Resets all bins with 2 Balls**
- **Winner keeps o**

Distribution

$(p_1, p_2, \ldots, p_n)$





How many balls does *a bin receive on average* between two wins?

Step Complexity

How many *total balls are distributed* between two wins, on average?

System Latency

The Result

Theorem. Given arbitrary distribution p and constant-length lock-free algorithm, the following hold:

- System latency is Θ (1 / norm$_2$(p))
- Individual latency is Θ (norm$_2$ (p) / p_i^2)

Other game types covered, e.g. obstruction-free algorithms.

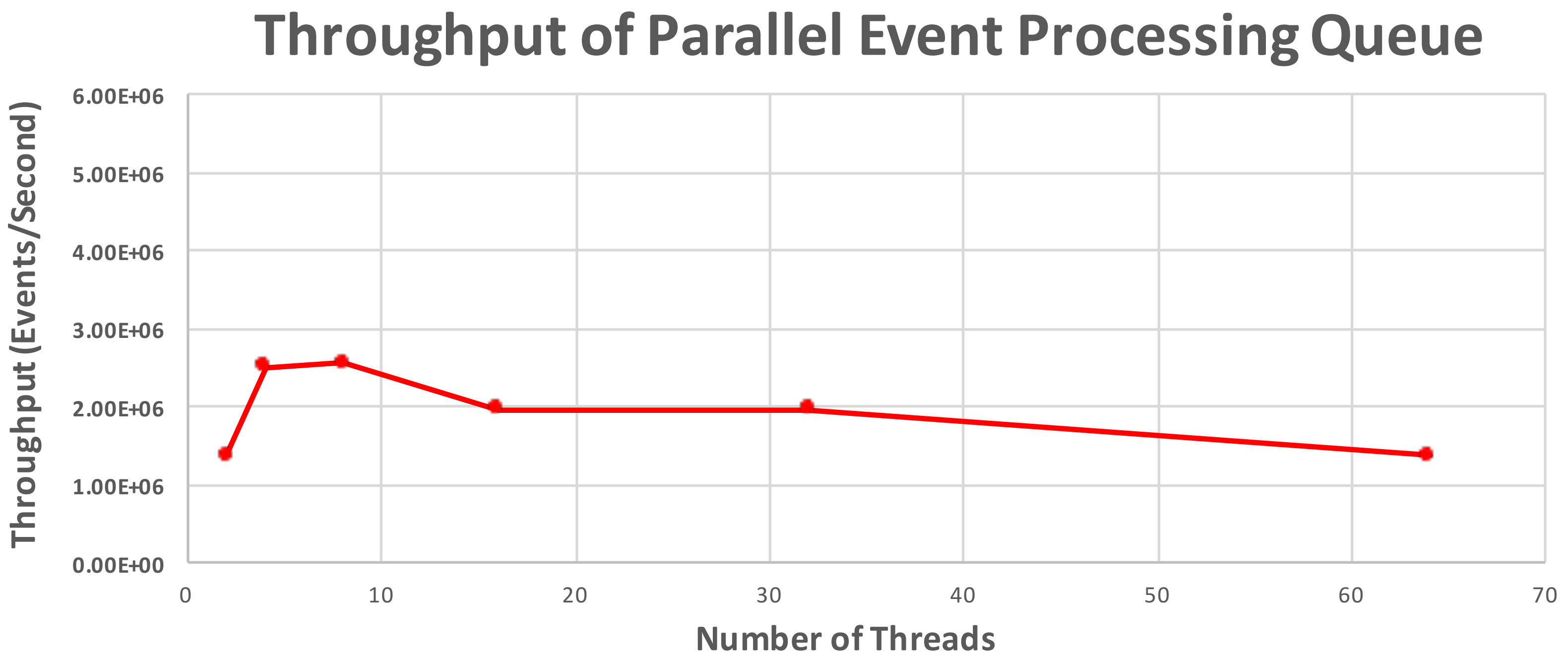
Examples:

1. Uniform p = (1/n, 1/n, …, 1/n):
   - System latency is **Θ (sqrt n)** [ACHS, JACM 16]
   - Individual latency is **Θ (n sqrt n)**
2. Non-uniform p = (↗1, ↘0, …, ↘0)
   - System latency is (close to) **constant**
   - Individual latency is either **constant,** or ↘ **0**

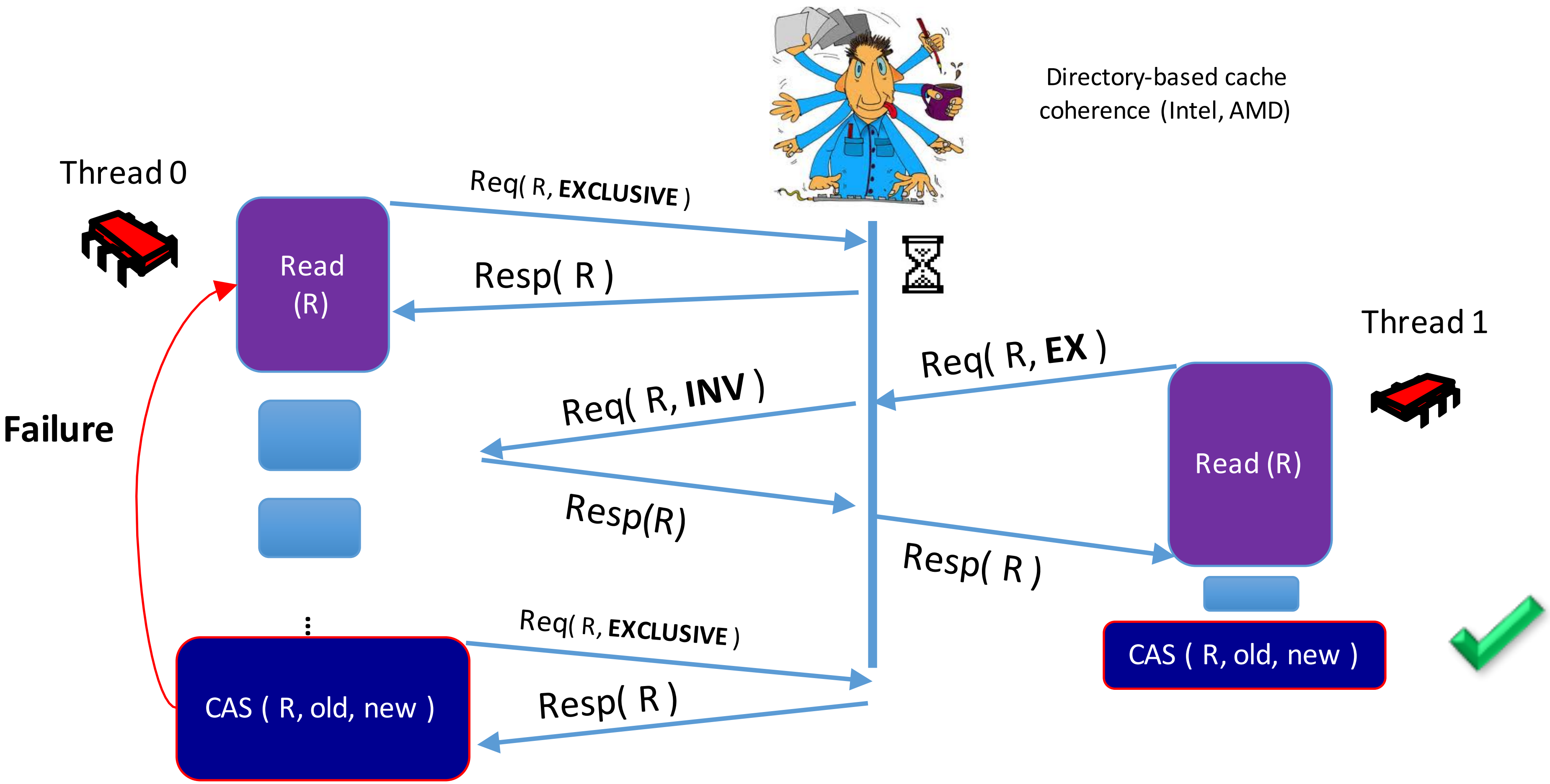
Fairness-Throughput Trade-off

Moral: Under **high contention**, roughly **one in sqrt (n) ops** succeeds.

Why does this graph look so bad?

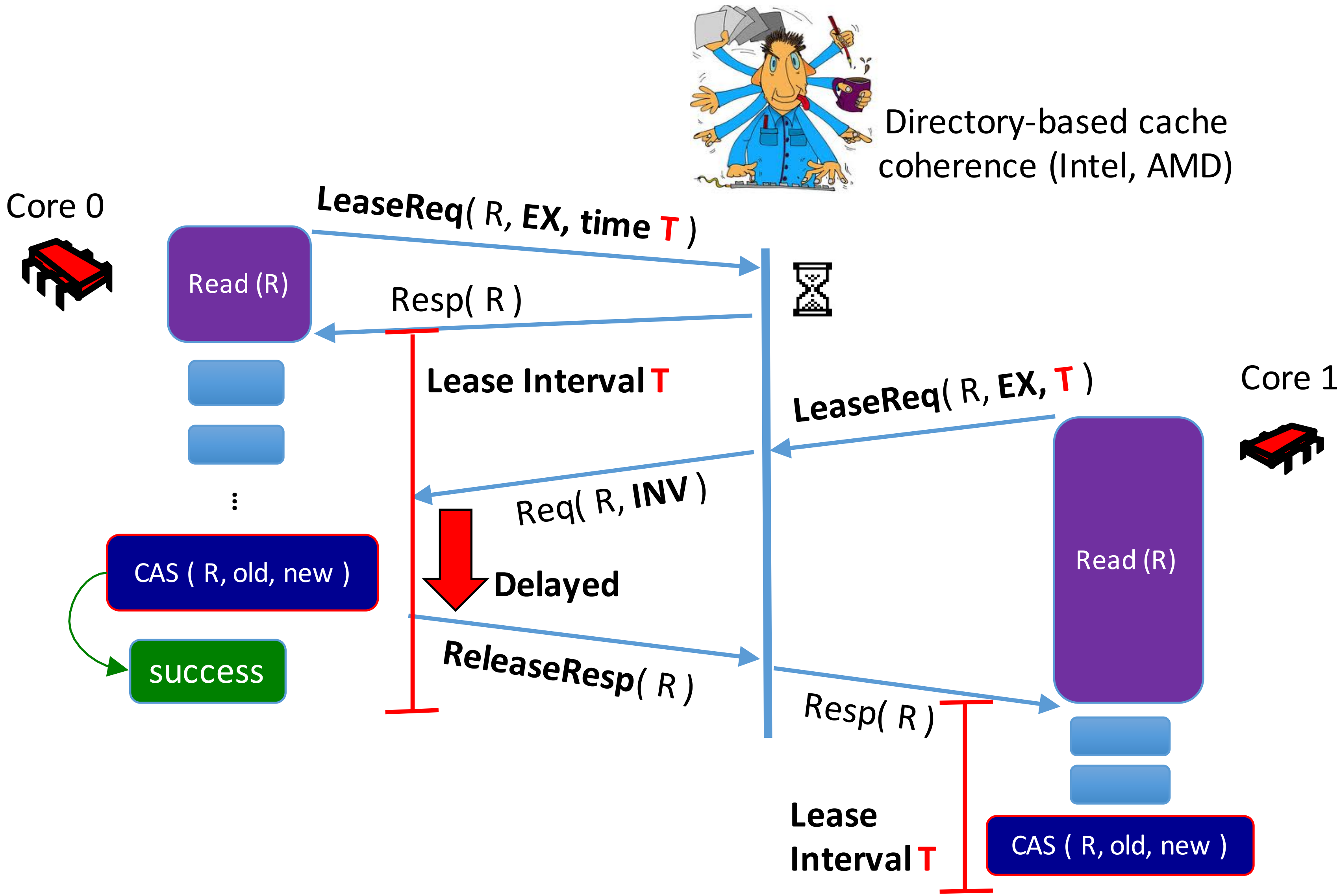


What Happens at the Hardware Level?



We waste time because ownership of R circulates without useful work!
Example: At **64 threads, only one in 8 message exchanges** is useful.

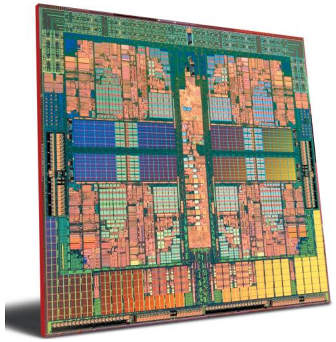
Fixing it: Lease/Release [Alistarh, Haider, Hasenplaugh, PPOPP 2016]



Each transfer results in **at least one useful operation**!

Doubling down on optimism!

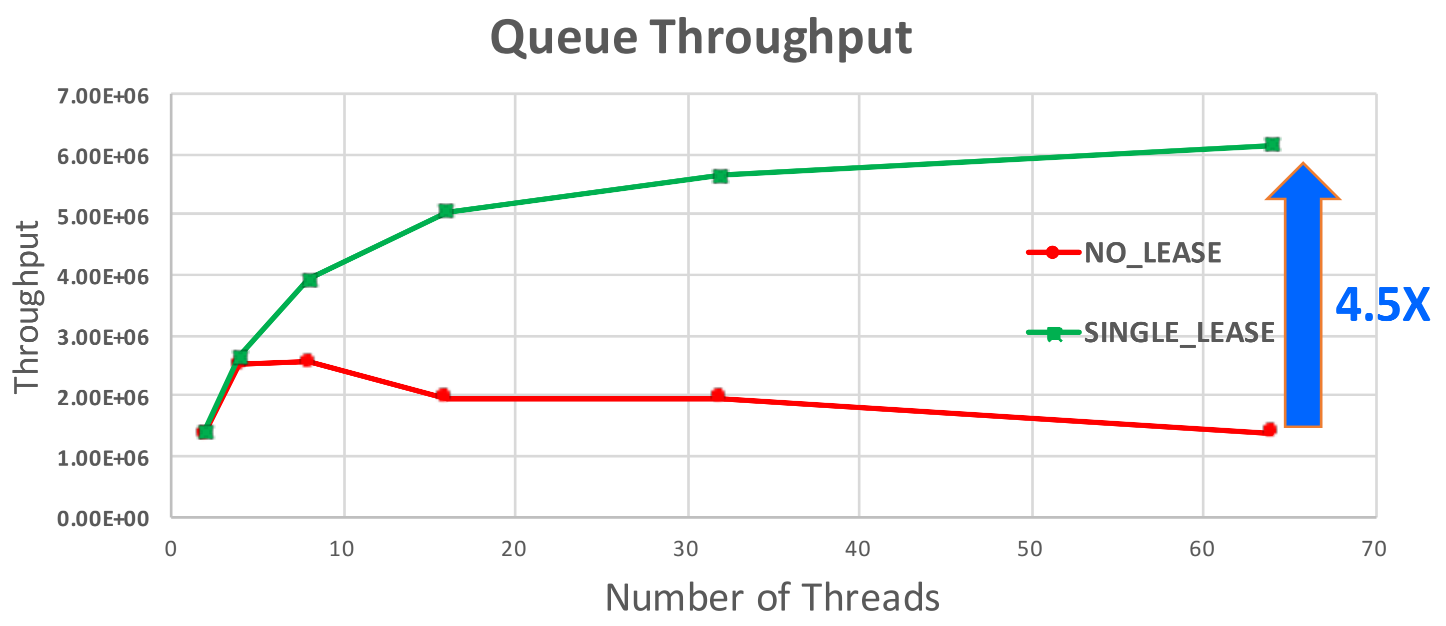
Lease/Release, More Precisely



- Programmer **optimistically** leases variables for **bounded time**
  - void ReqLease(void* address, int data_size, time T);
  - void ReqRelease(void* address, int data_size, time T);
  - Lease time in the order of **1000 cycles**
- **Performance penalty if leases expire before operation completion**
  - Usually occurs < 5% of the time
- Prototype in the **MIT Graphite Processor Simulator**
  - Directory-based MESI Cache Coherence Protocol
  - Protocol remains **provably correct**
  - **Minimal changes** to the architecture

Does it work?

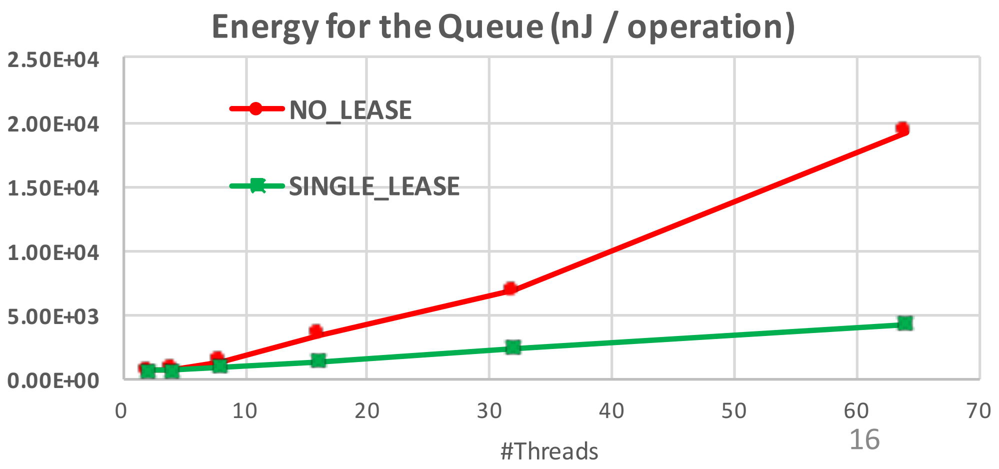
Packet Processing Queue with Lease-Release
(Simulated in Graphite)



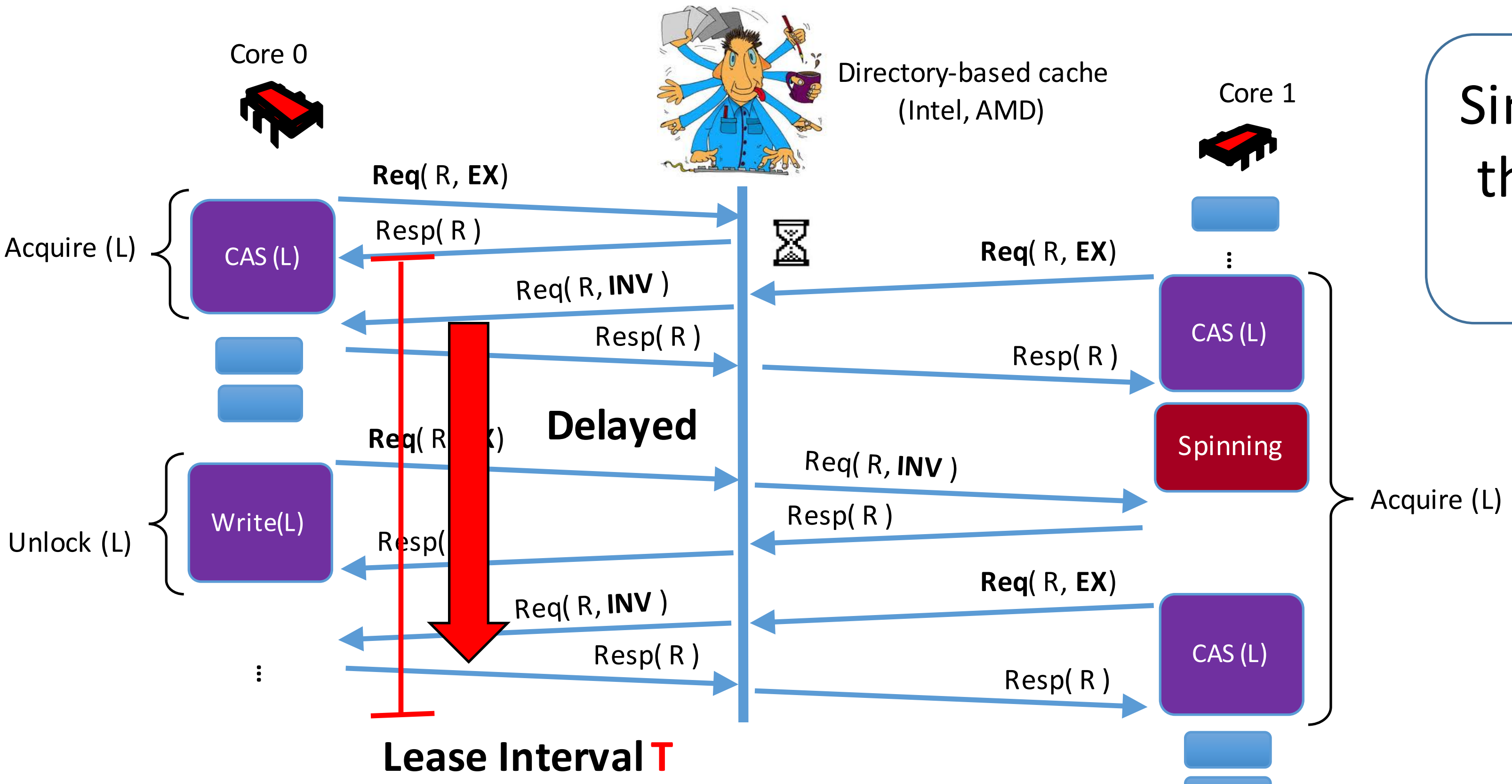
- Dequeue Operation

1. **Top_Node** = **Lease&Read**(**Head**)
2. **Next_Node** = Read(**Top_Node.ptr**)
3. **ATOMIC**

```
{
    if (Read( Head ) == Top_Node ) then
        Write&Release( Head , Next_Node )
    else
        Release and goto 1
}
```



What Else? **Locks**

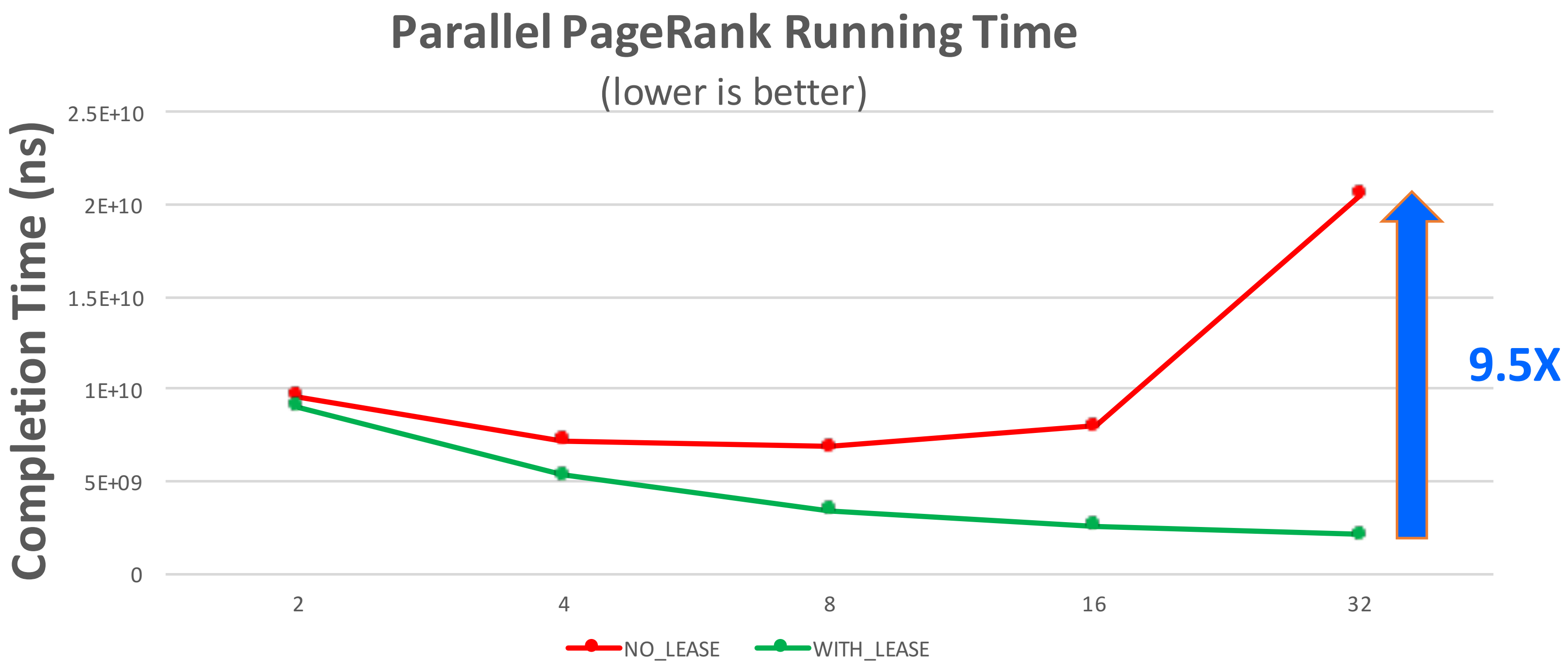


Simply Lease the lock on acquire!

Can we avoid the wasted coherence messages?

PageRank with L/R

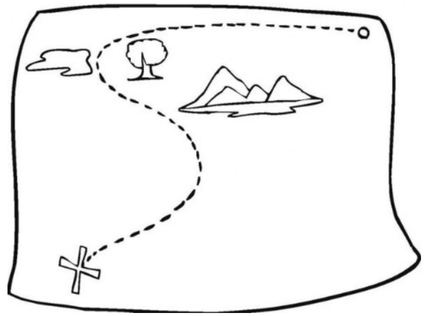
- Works with **lock-based programs** as well
  - Lease the lock before acquiring it
  - Release before giving it up



Lease/Release

- **Hardware Lock Queues [iQOLB: Rajwar, Kaegi, Goodman; HPCA 2000]**
  - **Locks using Load-Linked / Store-Conditional**
  - Load-Linked takes a “lease” on the lock, Store-Conditional “releases”
  - **Applied automatically** by the processor speculation mechanism
- **Transient Blocking Synchronization [Shalev, Shavit; Sun Tech Report 2004]**
  - Propose Load&Lease / Store&Release instructions for **non-coherent DSM machines**
  - Different semantics, never implemented
- **The paper also contains:**
  - Hardware **implementation details** (no directory modifications!)
  - Blueprint for implementing **multiple concurrent leases** (transactions)
  - **Lots of experiments**

The High-Level View

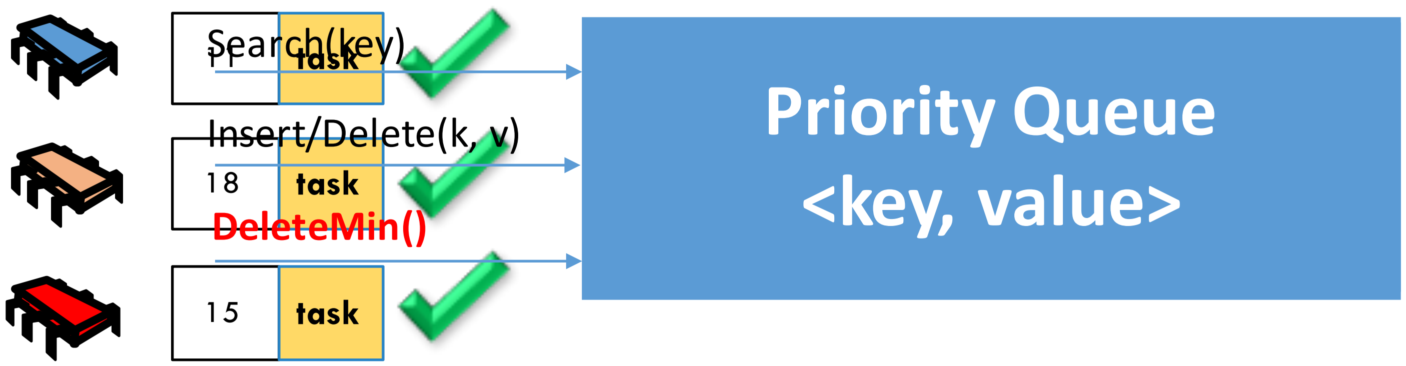


- The Problem with Concurrency
  - **Inherent bottlenecks** lead to **meltdowns**
- Why?
  - **Contention** hurts optimistic patterns, **quantifiably so**
- Lease/Release:
  - We can now **scale bottlenecks, within reason**
  - **Optimism enforced at the hardware level**

Can we scale **beyond bottlenecks**?

Let's Relax!

Concurrent Priority Queues



Methods:

- **Get Top Task**
- **Insert a Task**
- **Search for Task**

Extremely useful:

- Graph Operations (Shortest Paths)
- Operating System Kernel
- Time-Based Simulations

We are looking for a fast ***concurrent*** Priority Queue.

The Problem

Target: ***fast, concurrent*** Priority Queue**.**

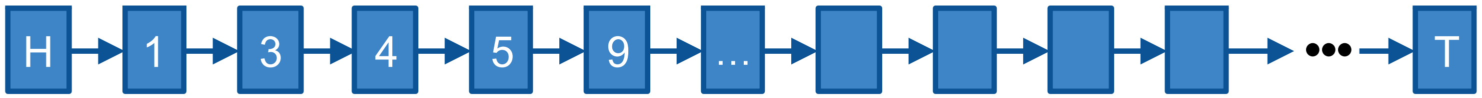
Lots *of work* on the topic:

[Sanders97], [Lotan&Shavit00], [Sundell&Tsigas07],
[Linden&Jonsson13], [Lenhart et al. 14], [Wimmer et al.14]

Current solutions **are hard to scale**:
DeleteMin is ***highly contended***.
Everyone wants the same element!

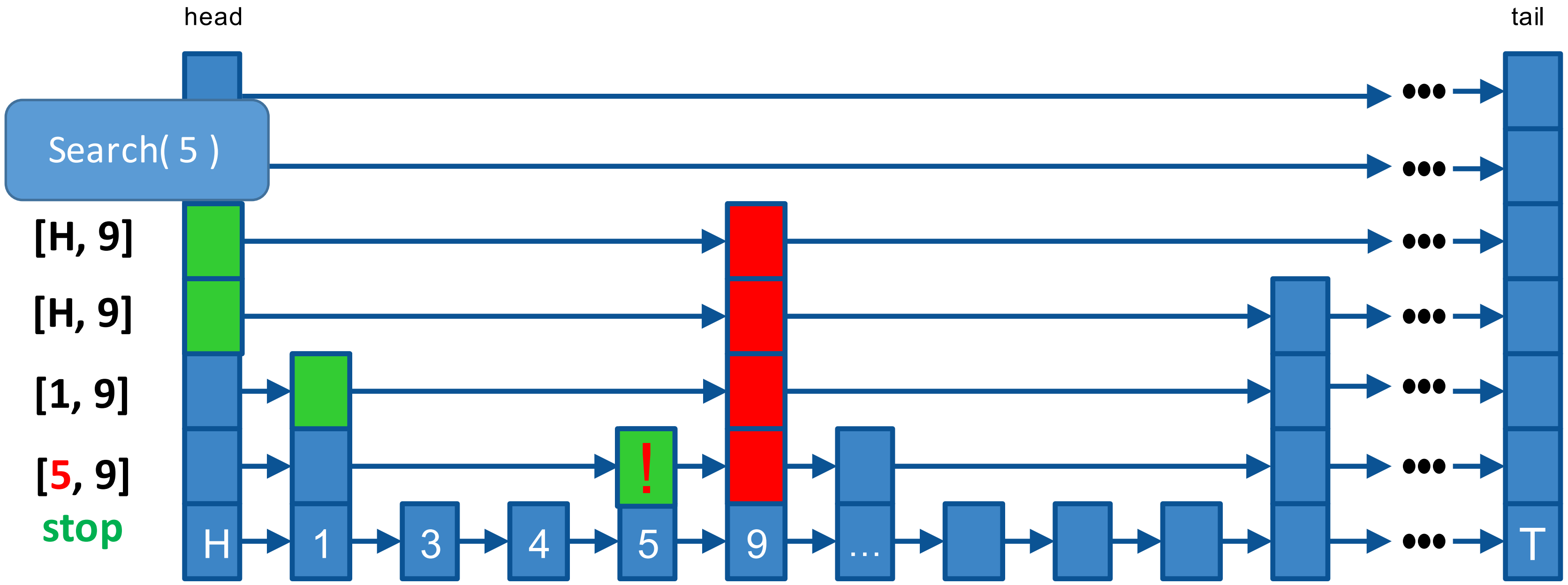
Concurrent Solution

- Linked list, **sorted by priority**
- Each node has **random “height”** (geometrically distributed with parameter ½)
- Elements at the same height form their own lists



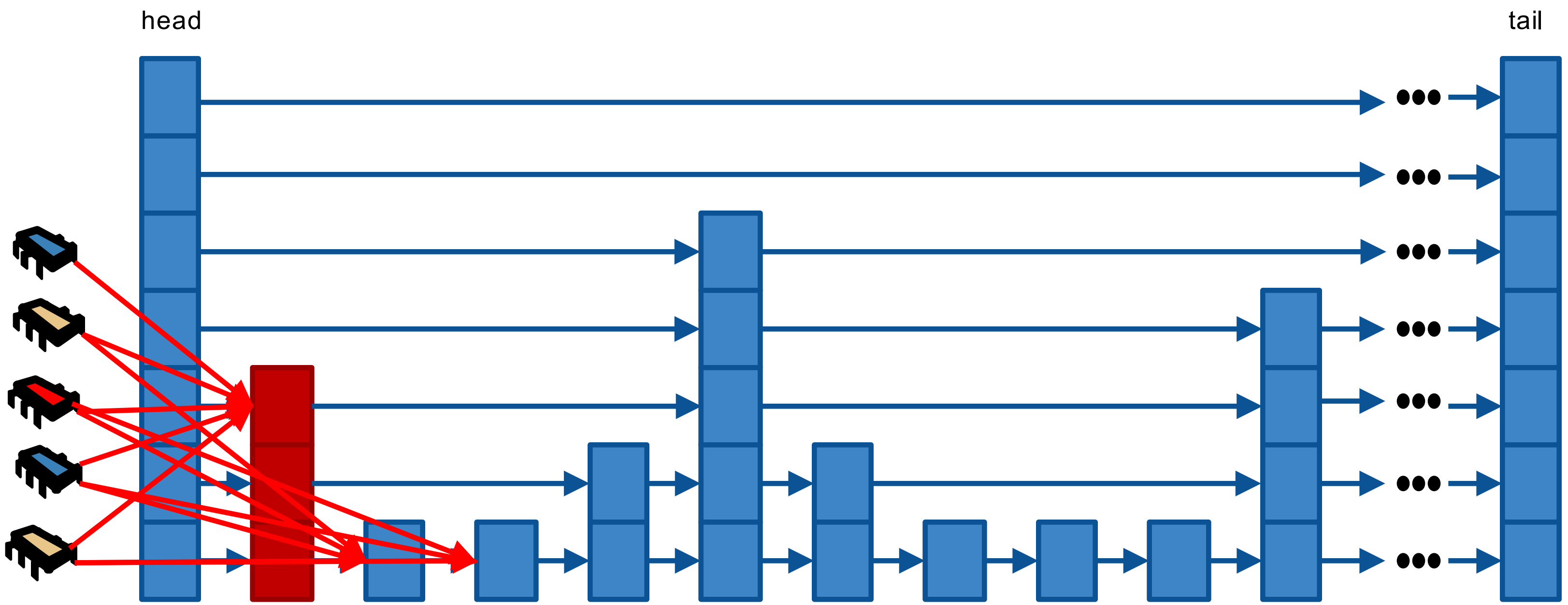
Concurrent Solution: the SkipList [Pugh90]

- Linked list, **sorted by priority**
- Each node has **random “height”** (geometrically distributed with parameter ½)
- Elements at the same height form their own lists
- **Average time** Search, Insert, Delete ***logarithmic***, work ***concurrently*** [Pugh98, Fraser04]



The SkipList as a PQ

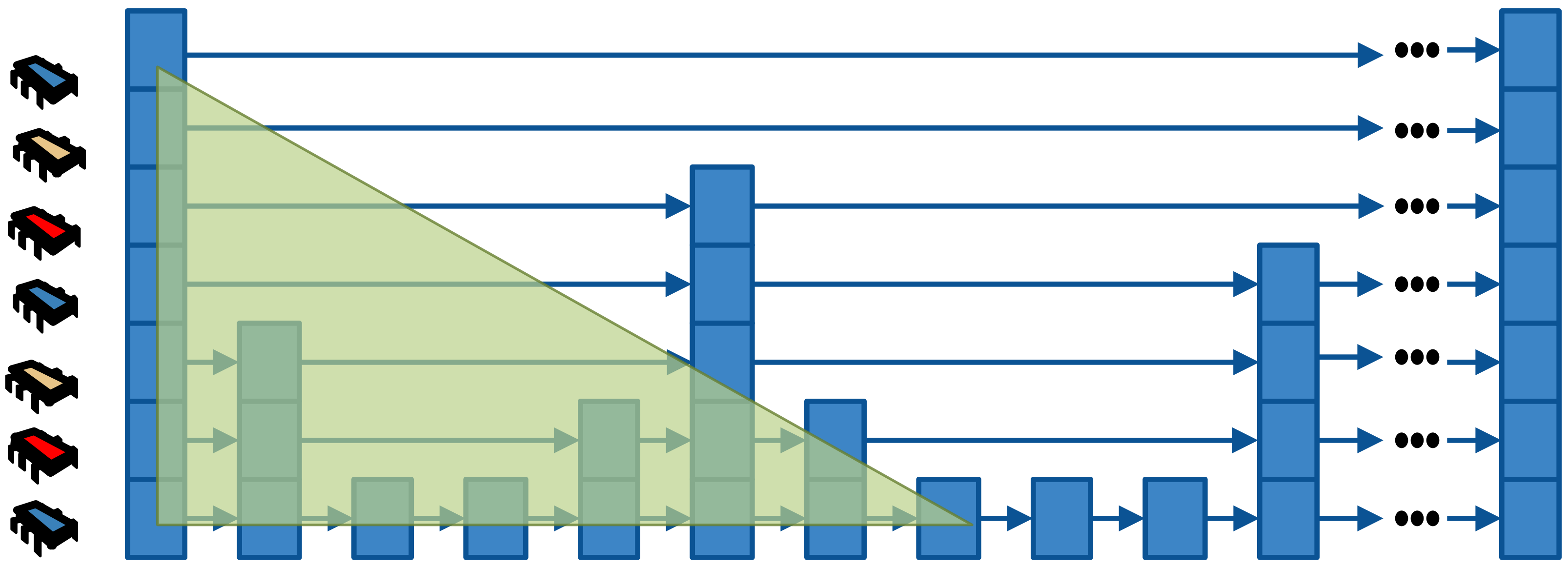
- DeleteMin: simply remove the smallest element from the bottom list
- All processors compete for smallest element
- Does not ***scale***!



I. Lotan and N. Shavit. Skiplist-Based Concurrent Priority Queues. 2000.

The Idea: Relax!

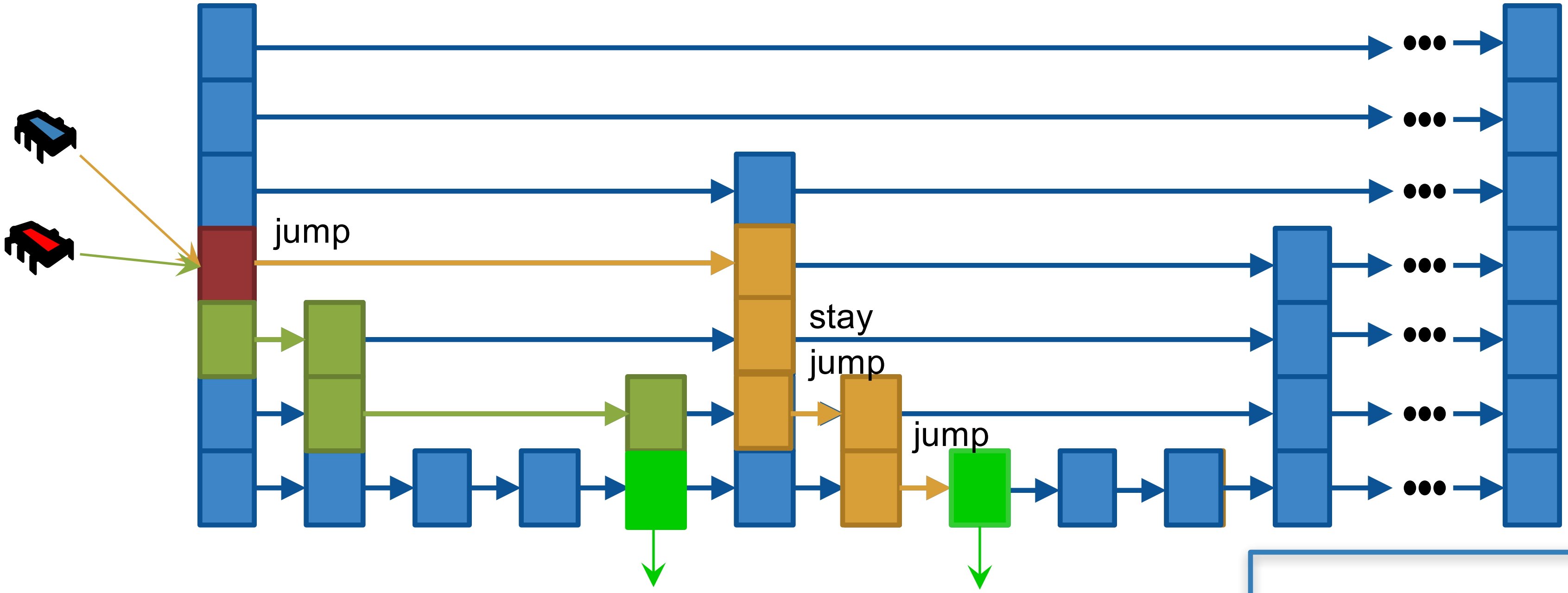
- We want to choose an item at random with ‘good’ guarantees
- Minimize ***loss of exactness*** by only choosing items near the ***front of the list***
- Minimize ***contention*** by keeping ***collision probability low***



DeleteMin: The Spray [Alistarh, Kopinsky, Li, Shavit, PPoPP 2015]

procedure Spray()

- At each skiplist level, **flip coin** to ***stay*** or ***jump forward***
- Repeat for each level from **log n** down to **1** (the bottom)
- ***As if removing a random priority element near the head***



Two examples for starting height 4

Spray and pray?

SprayList Probabilistic Guarantees

✓ Maximum value returned by Spray has rank $O(n \log^3 n)$

- Sprays aren't too wide

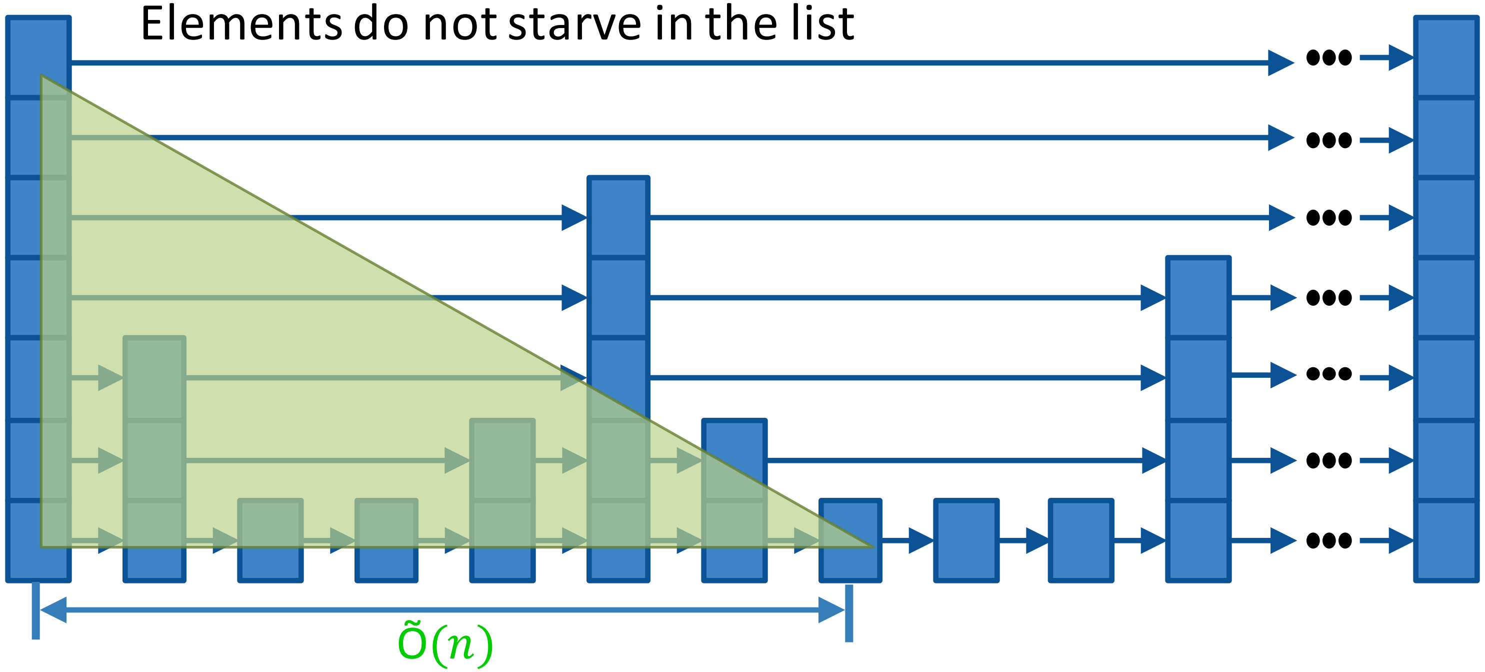
✓ For all x, p(x) = $\tilde{O}(1/n)$

- Sprays don't cluster too much

✓ If x > y is returned by some Spray, then p(y) = $\tilde{\Omega}(1/n)$

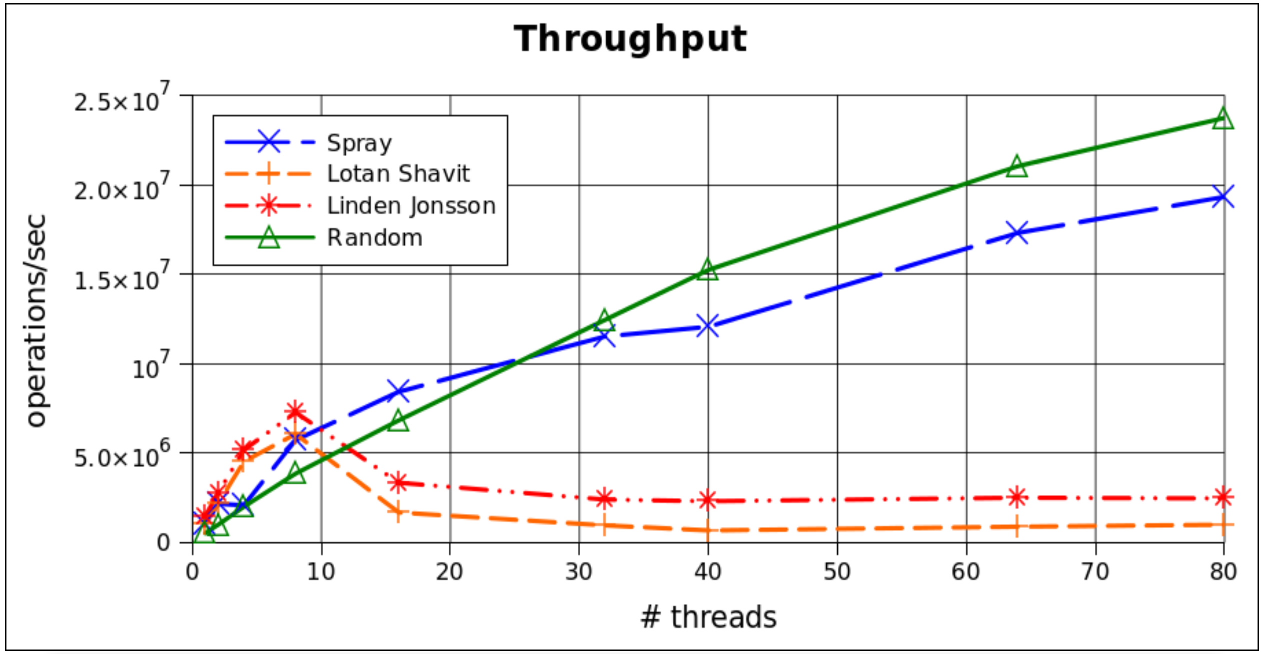
- Elements do not starve in the list

p(x) = probability that a spray returns value at index *x*



One Benchmark

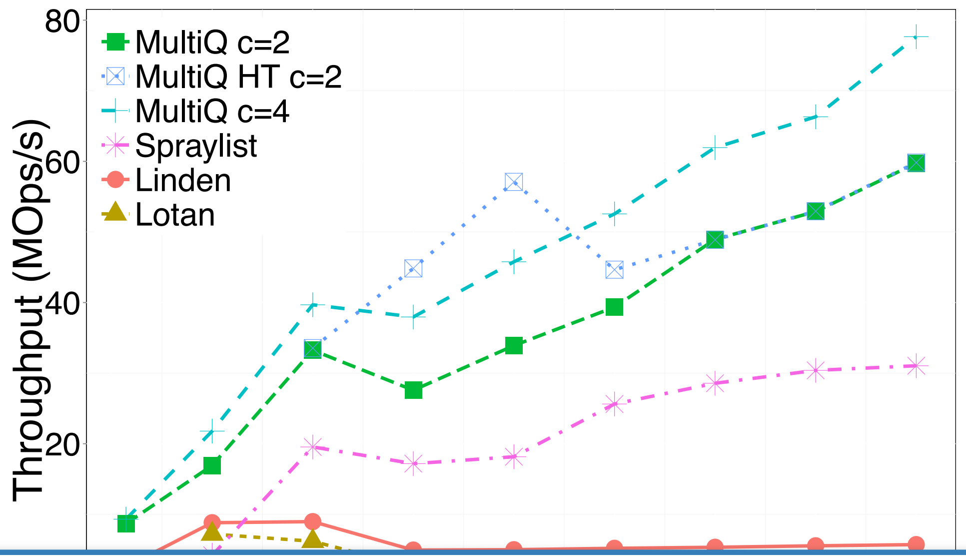
- Discrete Event Simulation
- **Exact algorithms** have **negative scaling** after 8 threads
- **SprayList competitive** with the **random remover** **(no guarantees, incorrect execution)**



In many ***practical*** settings (D.E.S., shortest paths), priority inversions are ***not expensive***.

The MultiQueue [Rihani, Dementiev, Sanders, SPAA 15]

- **n lock-free or lock-based queues**
- **Insert: pick a random queue, lock, and insert into it**
- **Remove: pick two queues at random, lock and remove the better element**



Looks good, but does it actually **guarantee** anything?

The Random Process

WLOG, elements are **consecutive labels.**

1. **Insert Elements u.a.r.**
2. **Remove using two choices**

- **Cost = rank of element removed among remaining elements**

Cost(2) = 2

Cost(4) = 3

Cost(1) = 1

Q1	Q2	Q3	Q4
1	4	2	5
6	7	3	9
10	12	8	11
13	16	15	14

We are interested in the **average rank** removed at each step.

Intuitively, the **distance from optimal**.

The Result

Theorem: Given ***n*** queues, for any ***t > 0***, the cost at ***t*** is **O(n)** in expectation, and **O(n log n)** w.h.p.

- Strategy 1: **reduction to power of two-choices analysis? [Azar et al., SICOMP 99]**
- Would apply if we could **equate queue size** with **top label (round-robin insert)**

1	1	1	1
2	2	2	2
3	3	3	3

The reduction **does not hold** in general, and in fact **experimentally height and top priority appear to be uncorrelated**.

The Result

Theorem: For any $t > 0$, the cost at t is **O(n)** in expectation, and **O(n log n)** w.h.p.

- Strategy 2: **some simple sort of induction**
- The **initial cost distribution** is **nice**; can we prove it **always stays nice**?

1

2

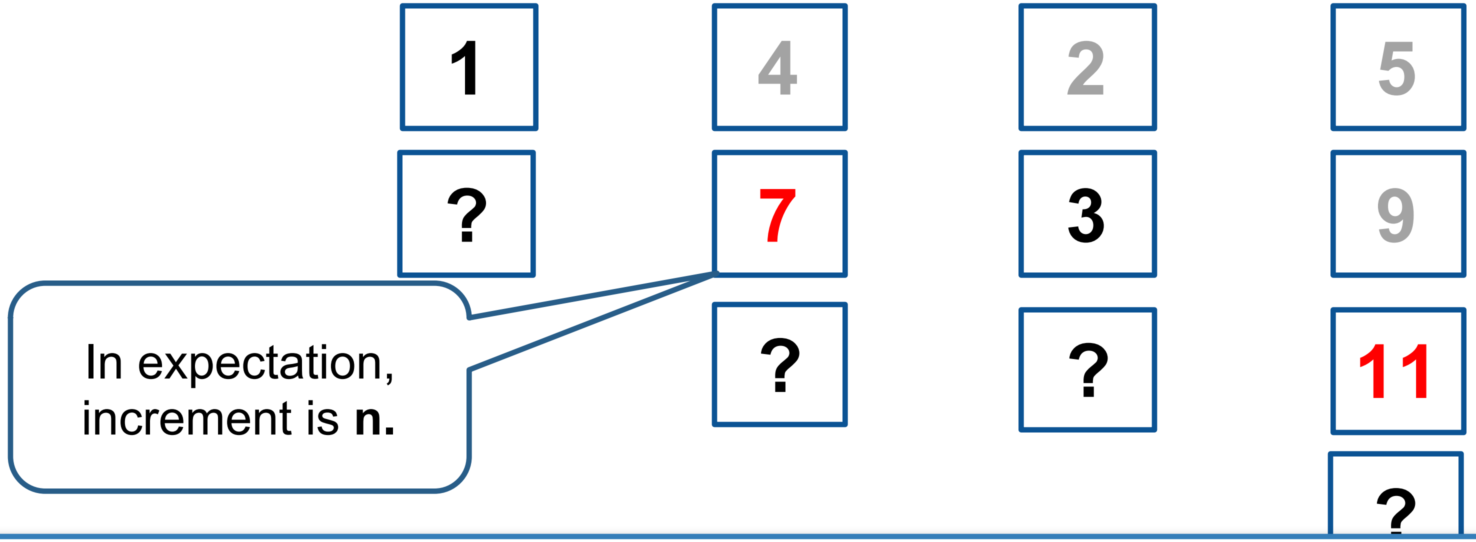
⋮

Hard case: over time, we'll eventually get **arbitrary distributions**. We have to prove that the algorithm gets out of those **reasonably fast**.

The Result

Theorem 1: For any $t > 0$, the cost at t is **O(n)** in expectation, and **O(n log n)** w.h.p.

- Strategy 3: **some ~~simple~~ complicated sort of induction / potential argument**
- **Idea:** characterize what's going on **step-by-step**



Problem: the behavior at a step is **highly correlated** with what happened in **previous steps**.

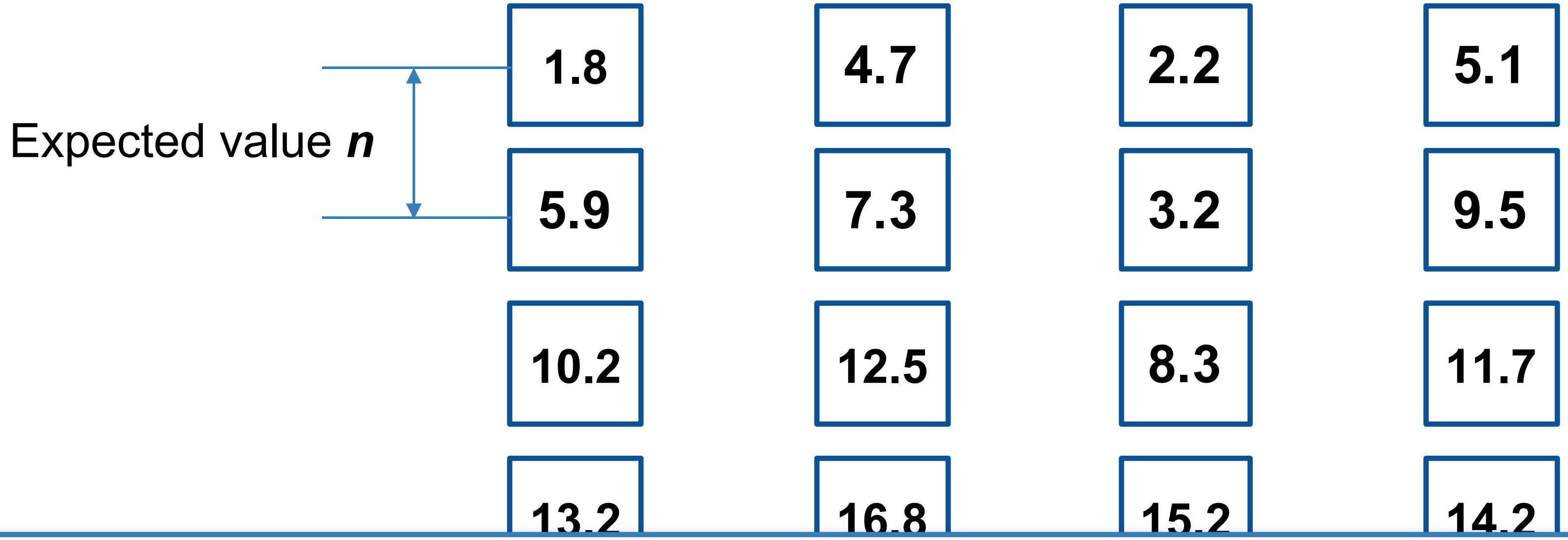
Proof Strategy

Theorem 1: For any $t > 0$, the cost at t is **O(n)** in expectation, and **O(n log n)** w.h.p.

- **Step 1**: reduce to an **uncorrelated exponential process**
  - Prove that the rank distribution is preserved
- **Step 2**: characterize the exponential process
  - Characterize average weight on top of queues via potential argument
- **Step 3:** characterize rank distribution of exponential process
  - Prove that average rank is O(n)

Step 1: The exponential process

- **Insert:** pick a **random queue**
- Insert **exponentially distributed increment with mean n** into it
- **Remove:** pick two queues at random, remove the lower **label**
- **Cost:** the ***rank*** of the element removed **(still)**



Theorem: The **distribution of removed ranks** is **the same** in the **discrete process** and in the **exponential process**.

Pr[rank k is in queue j] = 1 / n.

Holds since **the exponential is memoryless.**

Step 2: Analyzing the exponential process

- Fix a removal step $\boldsymbol{t}$. Let $\boldsymbol{w_i(t)}$ be the label (real value) on top of bin $\boldsymbol{i}$.
- Let $\boldsymbol{x_i(t)} = \frac{\boldsymbol{w_i(t)}}{\boldsymbol{n}}$ (normalized weights), and $\boldsymbol{\mu(t)} = \sum_{i=1}^{n} \boldsymbol{x_i(t)/n}$
- Let $\boldsymbol{\Phi(t)} = \sum_{i=1}^{n} \mathbf{exp}(\boldsymbol{x_i(t)} - \boldsymbol{\mu(t)})$ and $\boldsymbol{\Psi(t)} = \sum_{i=1}^{n} \boldsymbol{exp}(-(\boldsymbol{x_i(t)} - \boldsymbol{\mu(t)}))$.

Theorem: For any t > 0, $\mathbb{E}[\boldsymbol{\Phi(t)}+\boldsymbol{\Psi(t)}] = \boldsymbol{O(n)}$.

Uses tools from [Peres, Talwar, Wieder, R.S.A. 14]

- **No more correlations:** since weight increments are independent of previous steps, we can bound the expected increase in potential at each step.
- **Bad configurations**: $\Phi(t)$ and $\Psi(t)$ cannot both be large at the same time. If their sum breaks the O(n) barrier, then the large potential will decrease very fast.
- $\boldsymbol{\Phi(t)}+\boldsymbol{\Psi(t)}$ is then a super-martingale, which implies the bound.

Step 3: What does all this have to do with ***ranks***?

- Let $\boldsymbol{B_{>s}(t)}$ be the number of bins with weight $> \mu + s$ at time t.
- Let $\boldsymbol{B_{<-s}(t)}$ be the number of bins with weight $< \mu - s$ at time t.

Theorem: For any t > 0, $\mathbb{E}[B_{>s}(t)] = O\left(\frac{n}{\exp\frac{s}{n}}\right)$ and $\mathbb{E}[B_{<-s}(t)] = O\left(\frac{n}{\exp\frac{s}{n}}\right)$.

Weights become "rarefied" at ranks s-higher and s-lower than the mean value.

- But on average, we'll choose something close to the mean value! So, we conclude:

Theorem: For any ***t > 0***, the rank cost at ***t*** is **O(n)** in expectation.

Worst-case bound follows in a similar way.

Applications

We can use this for **approximate queues, stacks, counters, timestamps.**

What if we do two choices only β% of the time?
(one choice otherwise)

Theorem: For any $t > 0$, the cost at t is $\mathbf{O(n / \beta^*)}$ in expectation, and $\mathbf{O(n \log n / \beta^*)}$ w.h.p.

Works well in practice.

What if the input distribution is *biased*?

Still works (within reason).

Concurrent Data Structures

"The data structures of our childhood are changing."

Nir Shavit

Data structures such as the Spraylist and the MultiQueue merge both **relaxed semantics** and **optimistic progress** to **achieve scalability.**

A relaxation renaissance

[KarpZhang93], [DeoP92], [Sanders98], [HenzingerKPSS13], [NguyenLP13], [WimmerCVTT14], [LenhartNP15], [RihaniSD15], [JeffreySYES16]

The Last Slide

<u>Theorem</u>: Strongly ordered data structures **won't scale**.

[Ellen, Hendler, Shavit, SICOMP 2013]

[Alistarh, Aspnes, Gilbert, Guerraoui, JACM 2014]

How can we scale them?

Theory ↔ Software ↔ Hardware

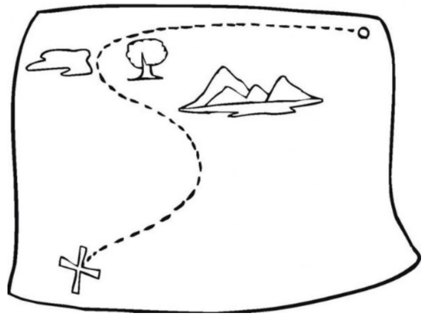
How do we **specify** and **prove** relaxed data structures correct?

How do these data structures interact with **existing applications**?

What **new data structures** are out there?

Can we prove **stronger lower bounds**?

Workshop Announcement



- **Theory & Practice in Concurrent Data Structures**
  - Co-located with DISC 2017 (Vienna)
- **Overall goals**
  - Fostering collaboration between practically-minded (PPoPP, SOSP etc) conferences, and the PODC/DISC community
  - New challenges in concurrent data structure design
- **Precise goals**
  - Better benchmarks for concurrent data structures
  - Real applications and practical issues (e.g. memory management)
  - Usefulness of relaxed designs