Increment-and-Freeze

Every Cache, Everywhere, All of the Time

Michael Bender
Stony Brook University

Daniel DeLayo
Stony Brook University

William Kuszmaul
Massachusetts Institute of Technology

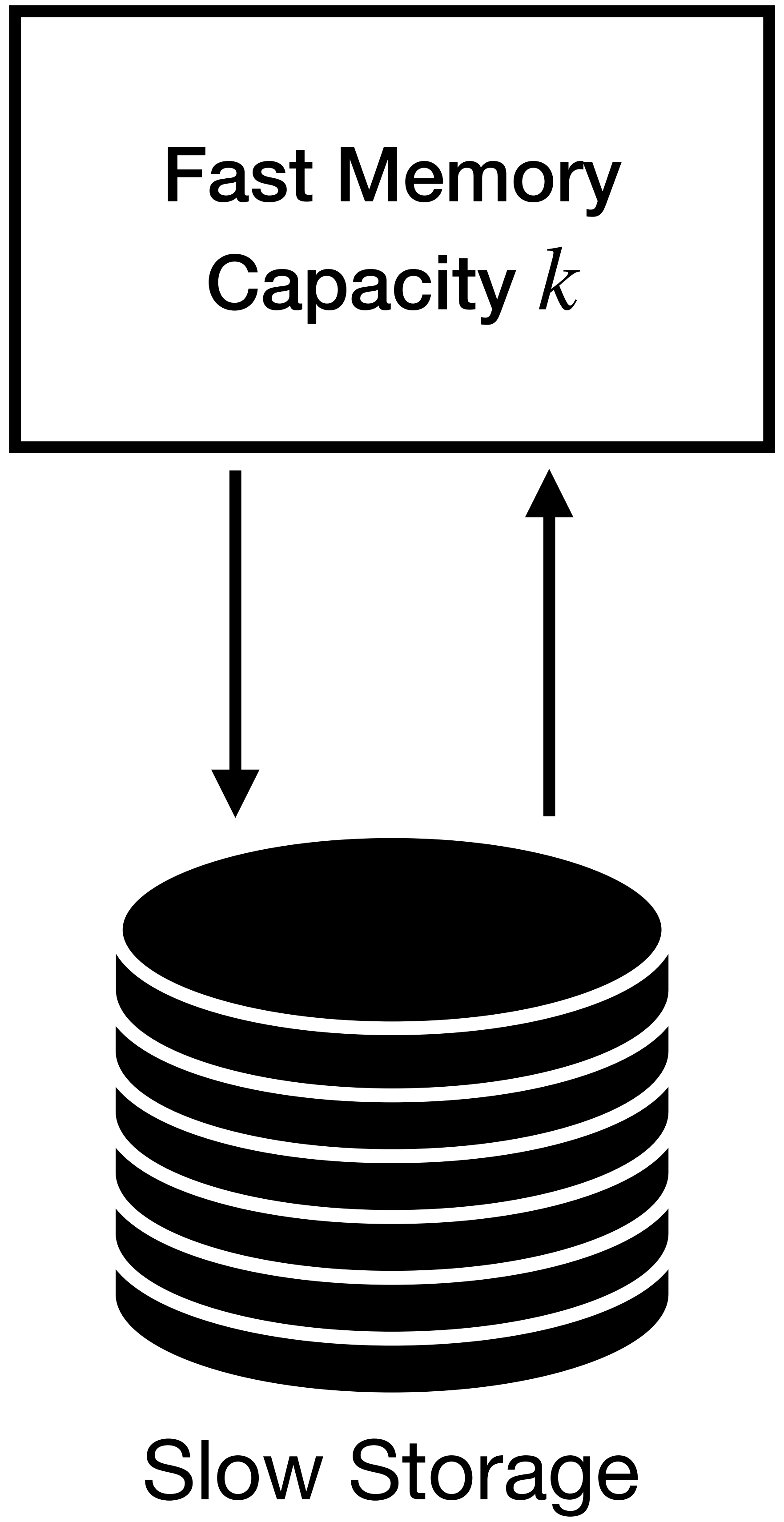
Bradley Kuszmaul

Evan West
Stony Brook University

The Paging Problem

Foundation

- Stream of **page** requests, e.g. ABACB

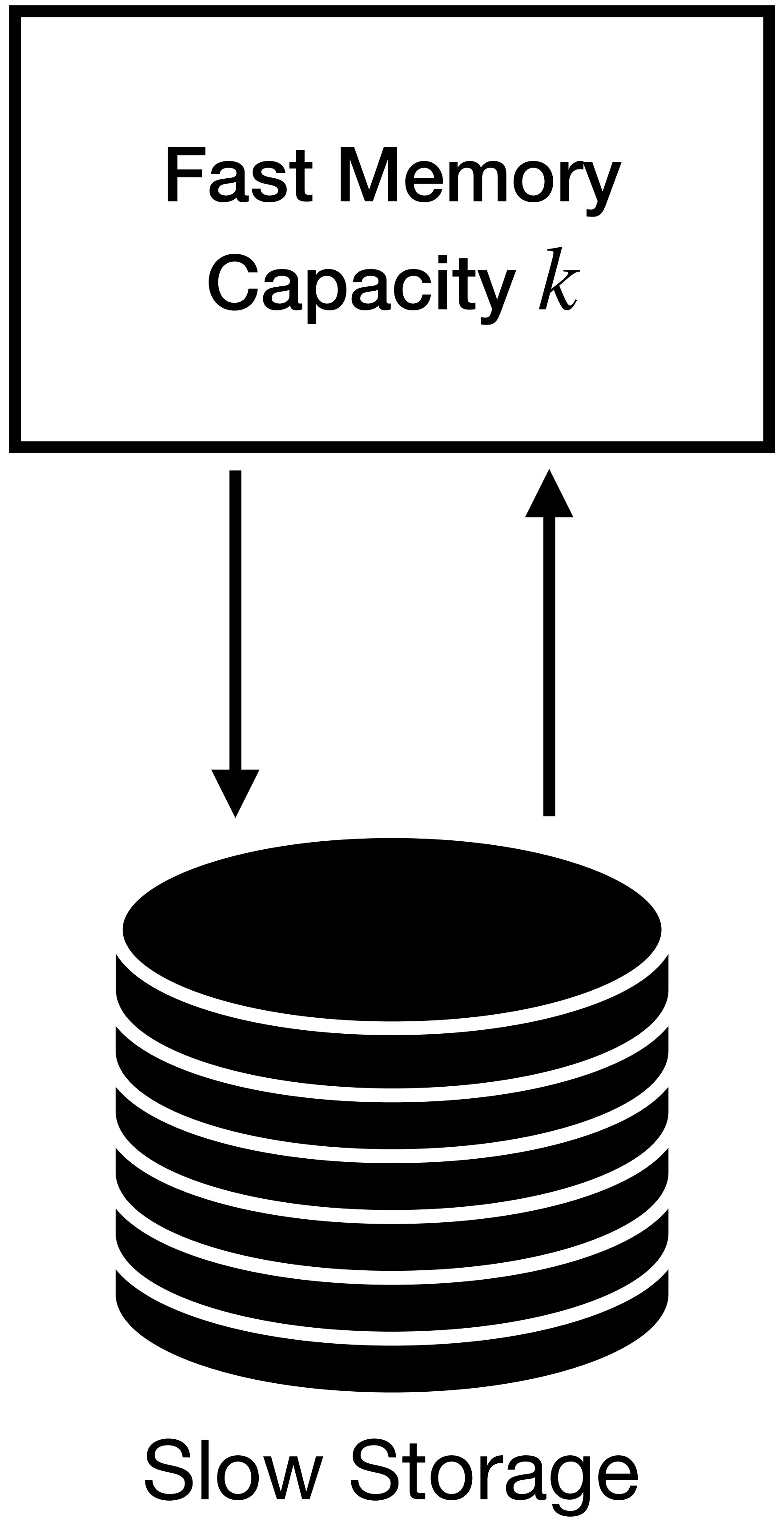


The Paging Problem

Foundation

- Stream of **page** requests, e.g. ABACB

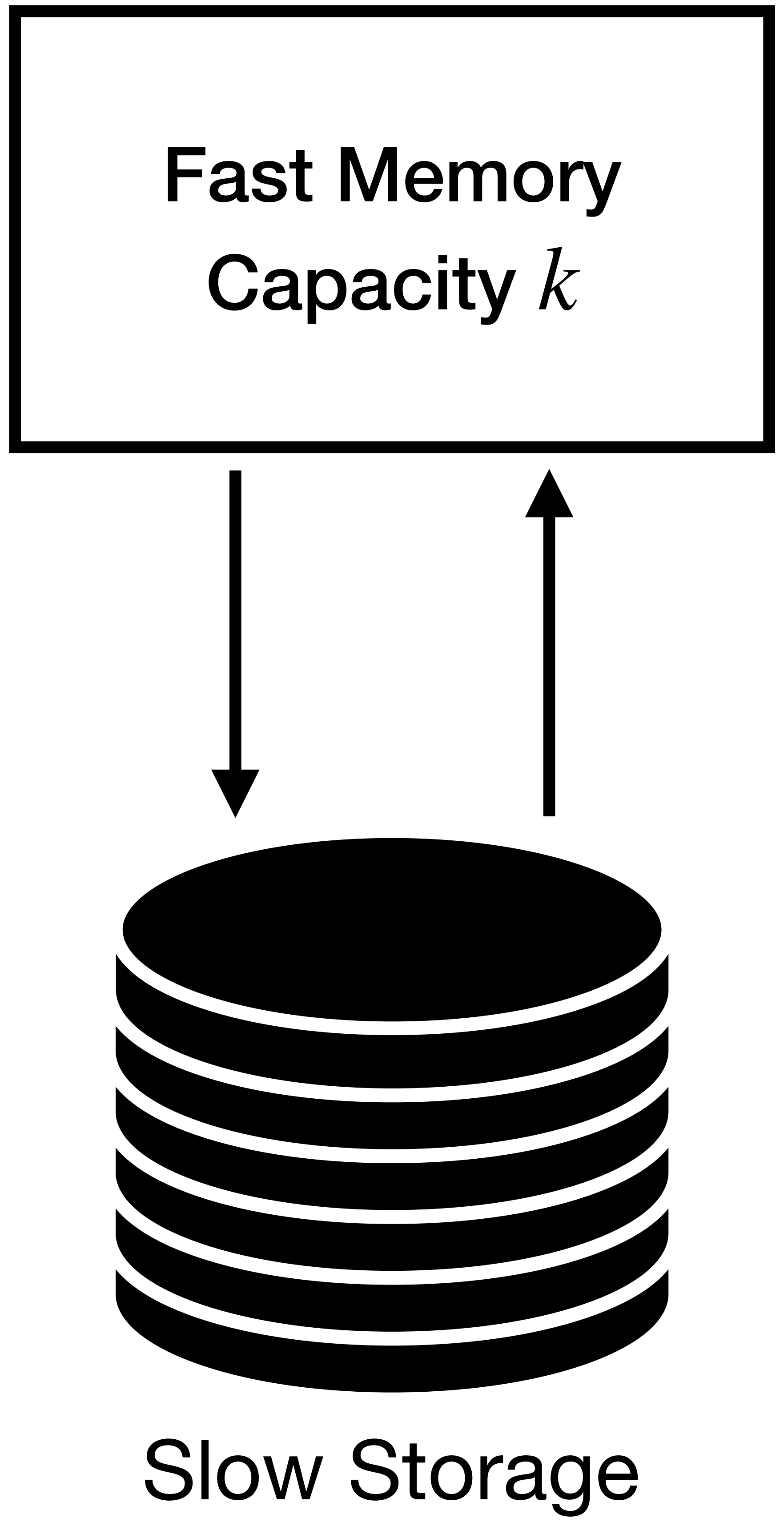
- Pages held within slow storage and must be cached in fast memory to be served
  - Fast **Hit** if page already cached, slow **miss** if not



The Paging Problem

Foundation

- Stream of **page** requests, e.g. ABACB
- Pages held within slow storage and must be cached in fast memory to be served
    - Fast **Hit** if page already cached, slow **miss** if not
- Algorithms for the paging problem make eviction decisions.
  Evicting the least recently used page is known solution



Getting on the Same Page

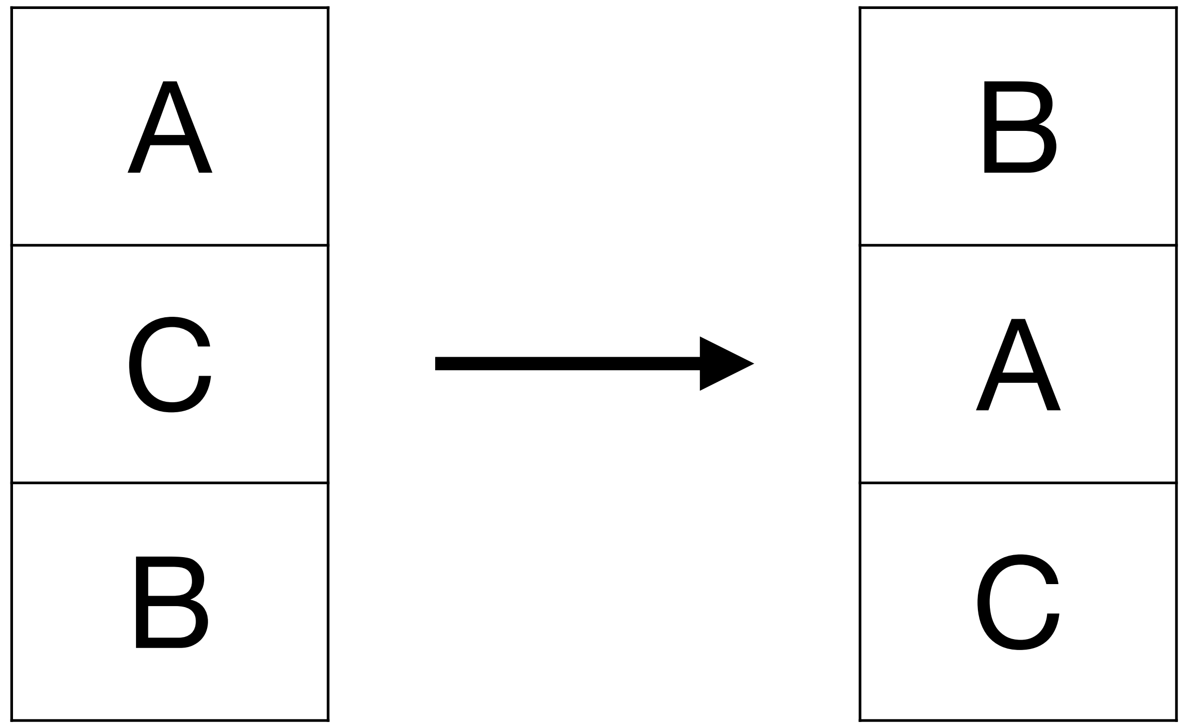
Review of LRU

- LRU orders pages as a stack with the most recently accessed pages on top and least recently accessed on bottom

Getting on the Same Page

Review of LRU

- LRU orders pages as a stack with the most recently accessed pages on top and least recently accessed on bottom



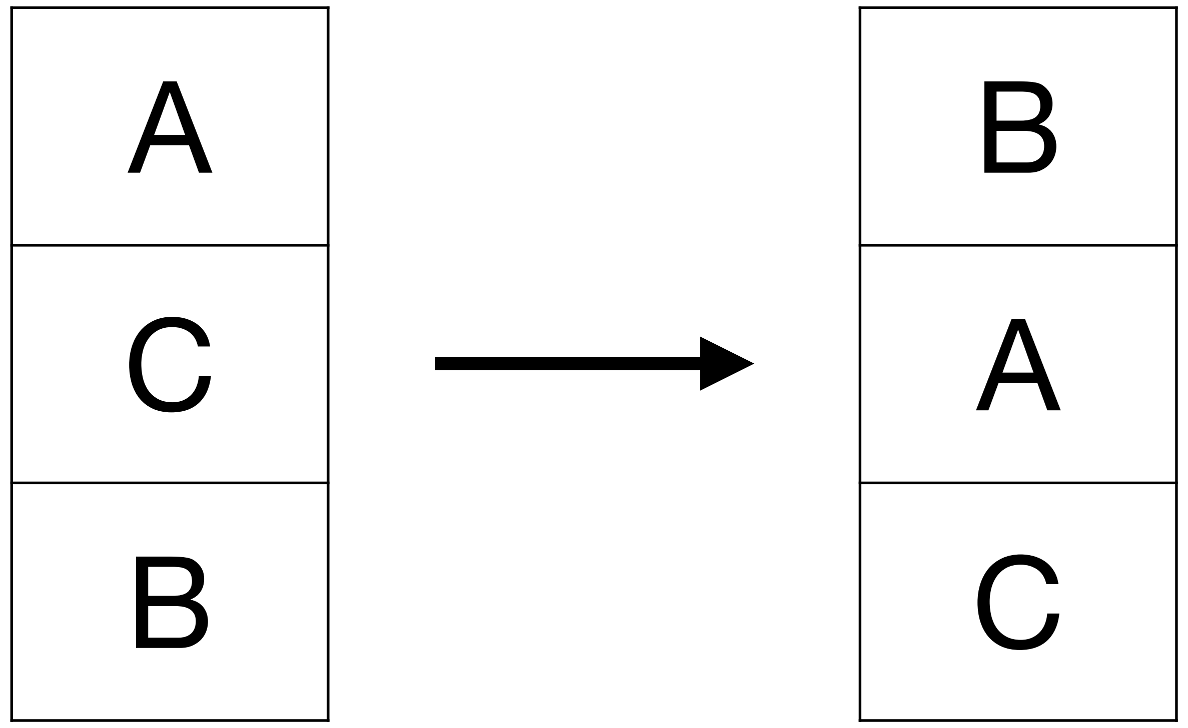
BACAB

Hit!

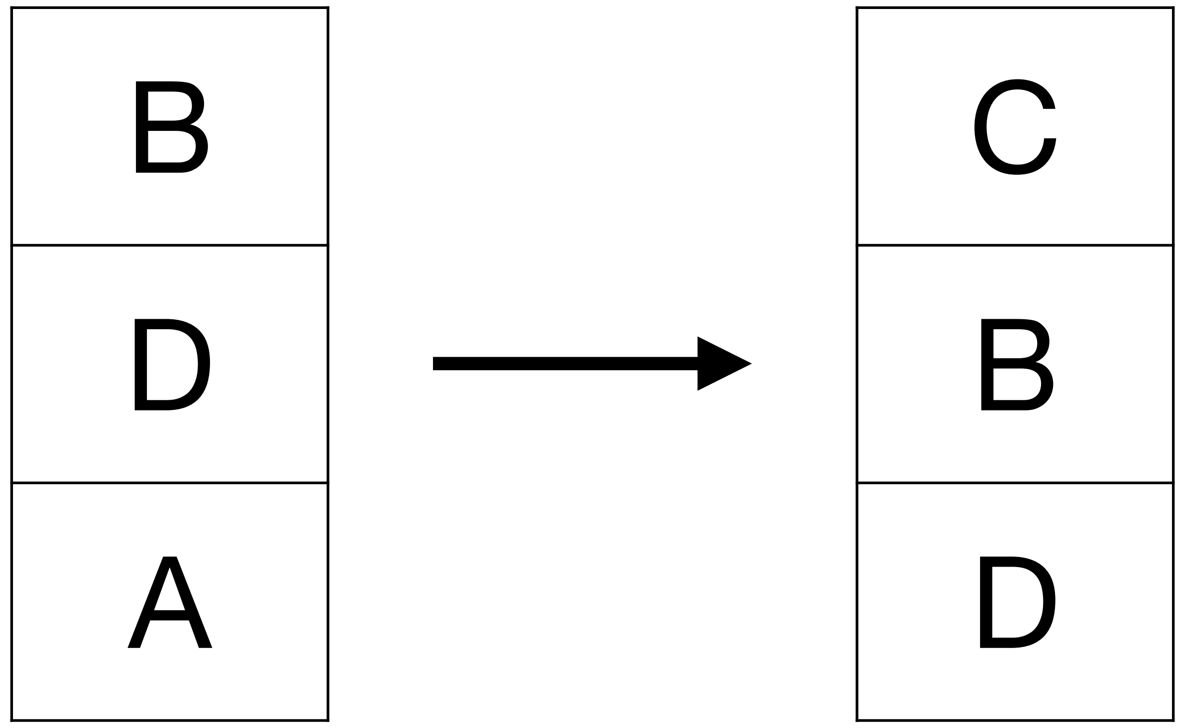
Getting on the Same Page

Review of LRU

- LRU orders pages as a stack with the most recently accessed pages on top and least recently accessed on bottom



BACAB
Hit!



ABDBC
Miss!

LRU Hit-rate Curves

Simulating Caches with LRU-Hit Rate Curves

- LRU hit-rate curves give the hit rate of every cache size for a sequence of page requests

- Sequence of page requests generated by execution of some program

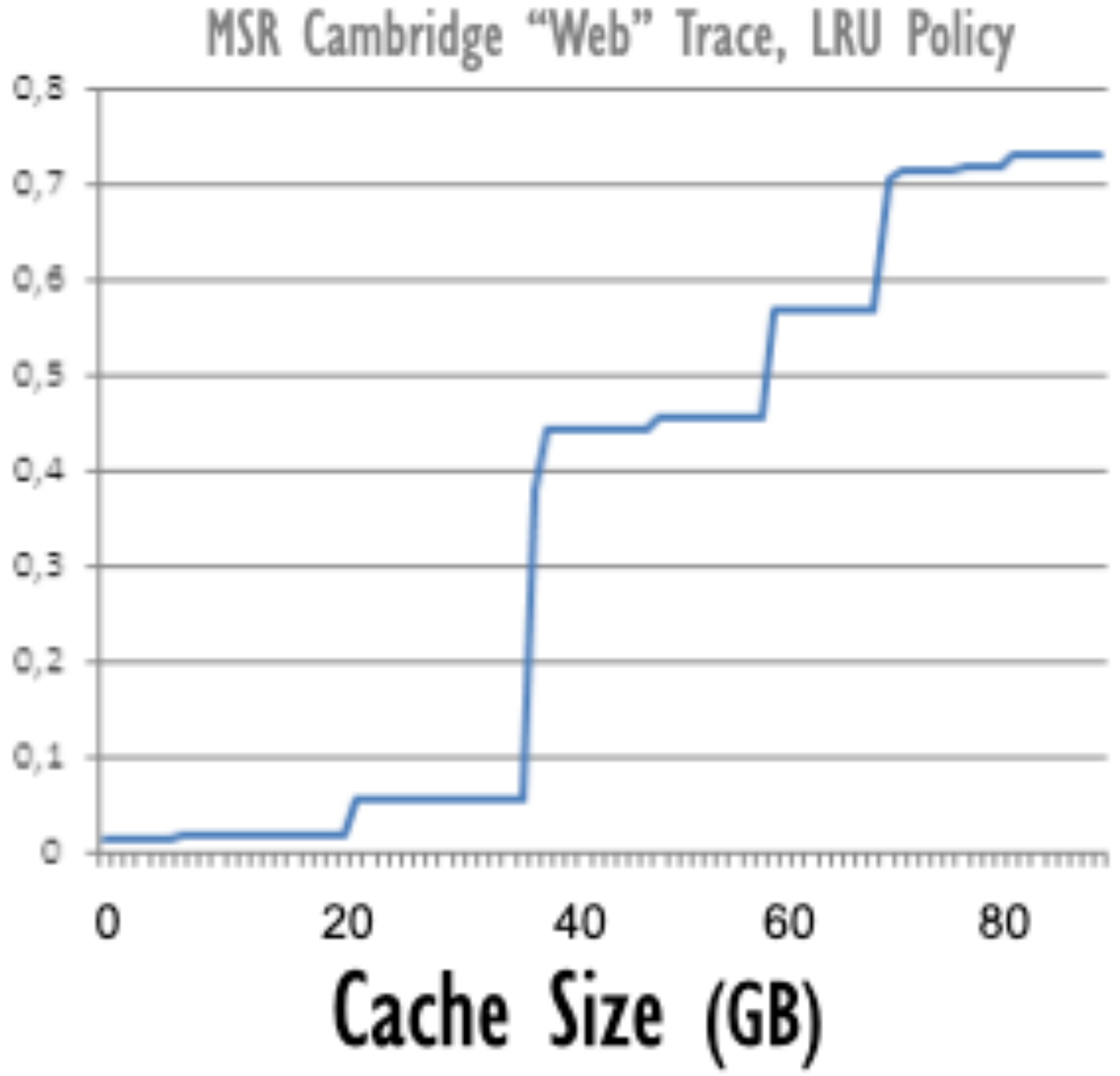


Image: [Druidi et al. 2015]

Got Cache Questions?

LRU hit-rate curves answer them

- The bigger the cache, the more expensive it is
- Misses are also expensive: user latency, server load

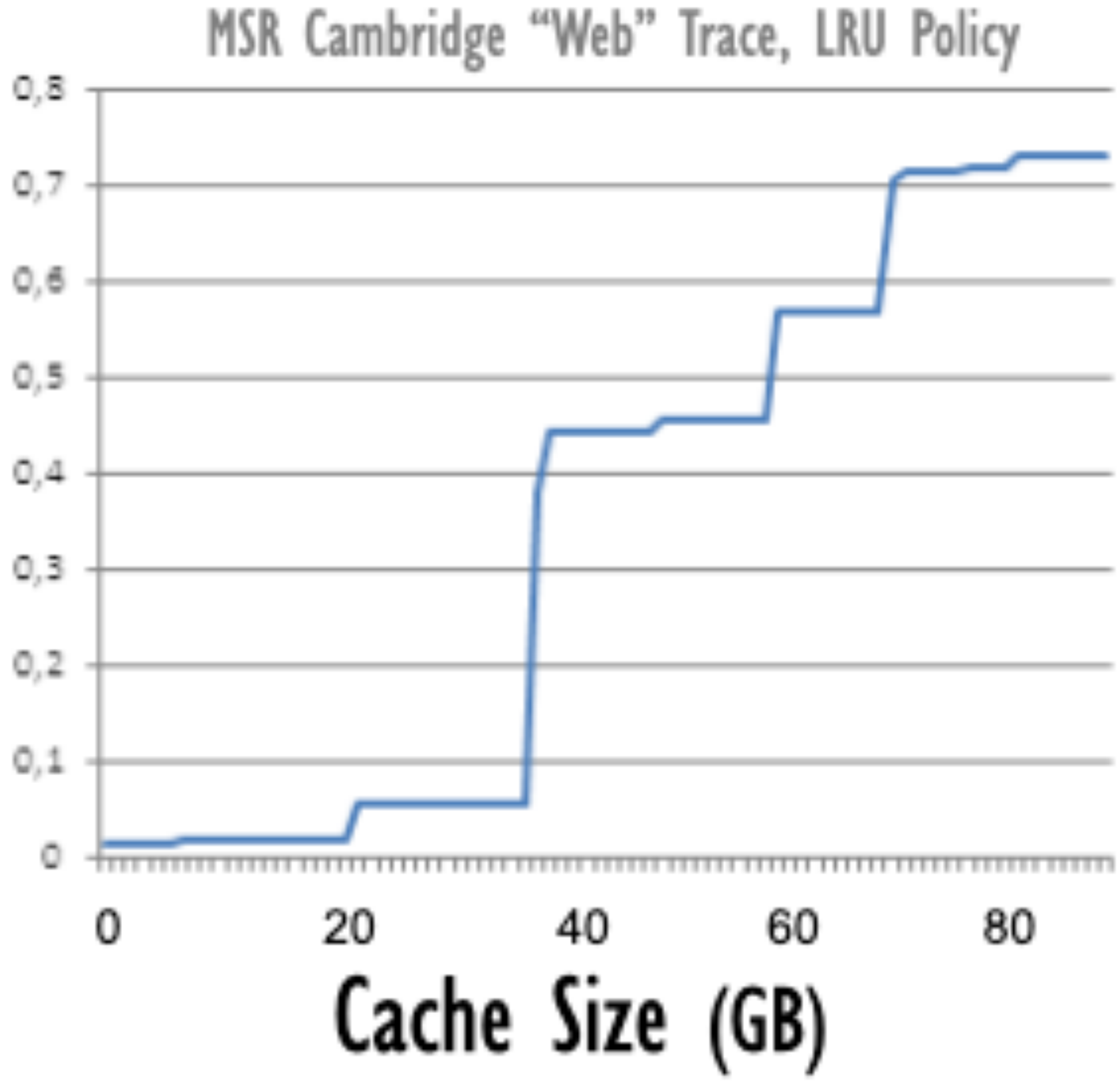


Image: [Druidi et al. 2015]

Got Cache Questions?

LRU hit-rate curves answer them

- The bigger the cache, the more expensive it is
- Misses are also expensive: user latency, server load
- Reduce cost by shrinking cache size?

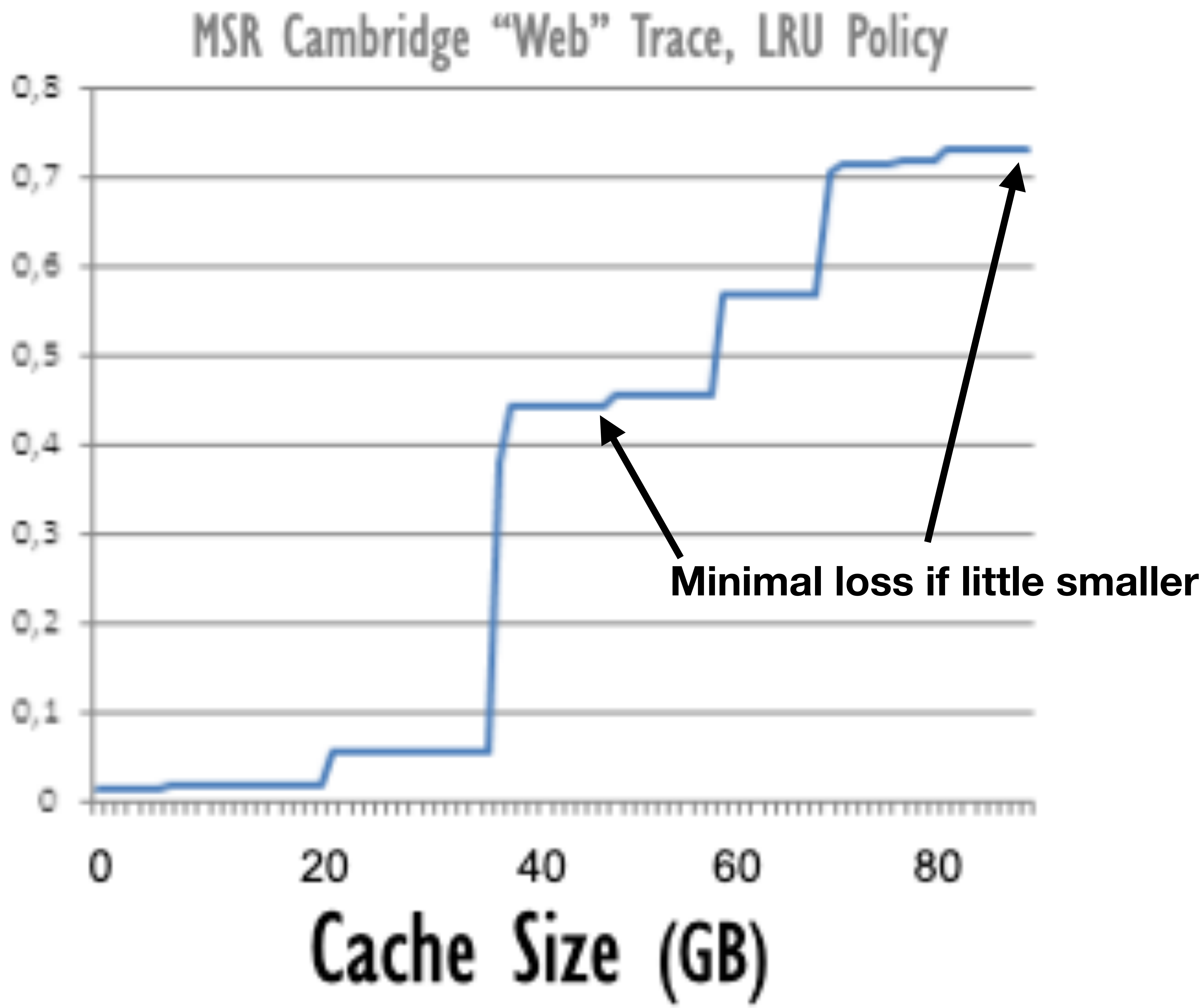


Image: [Druidi et al. 2015]

Got Cache Questions?

LRU hit-rate curves answer them

- The bigger the cache, the more expensive it is
- Misses are also expensive: user latency, server load

- Reduce cost by shrinking cache size?
- Improve hit rate via small increase?

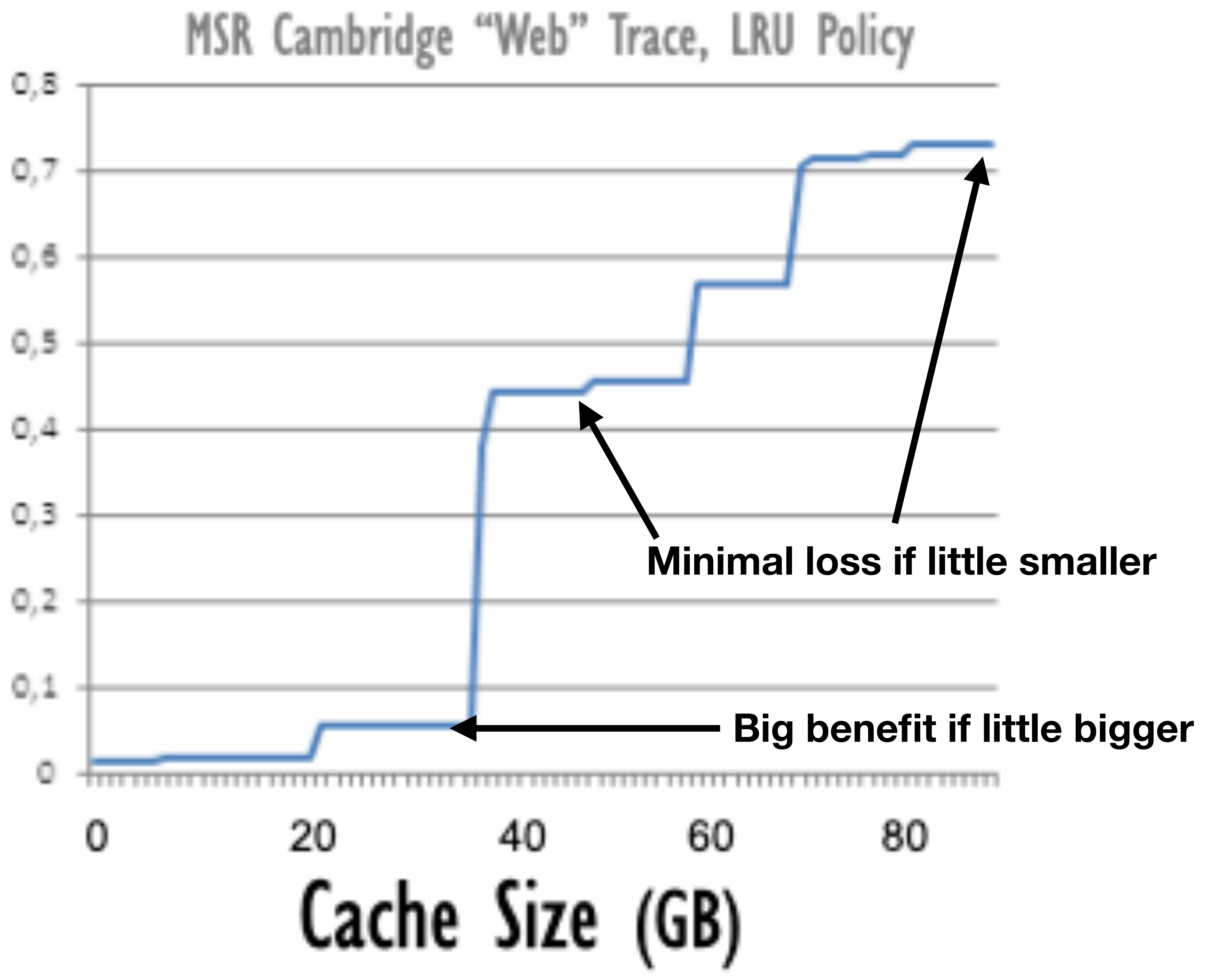


Image: [Druidi et al. 2015]

More Questions

How is my cache heuristic behaving?

- Most caches do not actually use LRU
  - e.g. Clock or ML heuristic approach

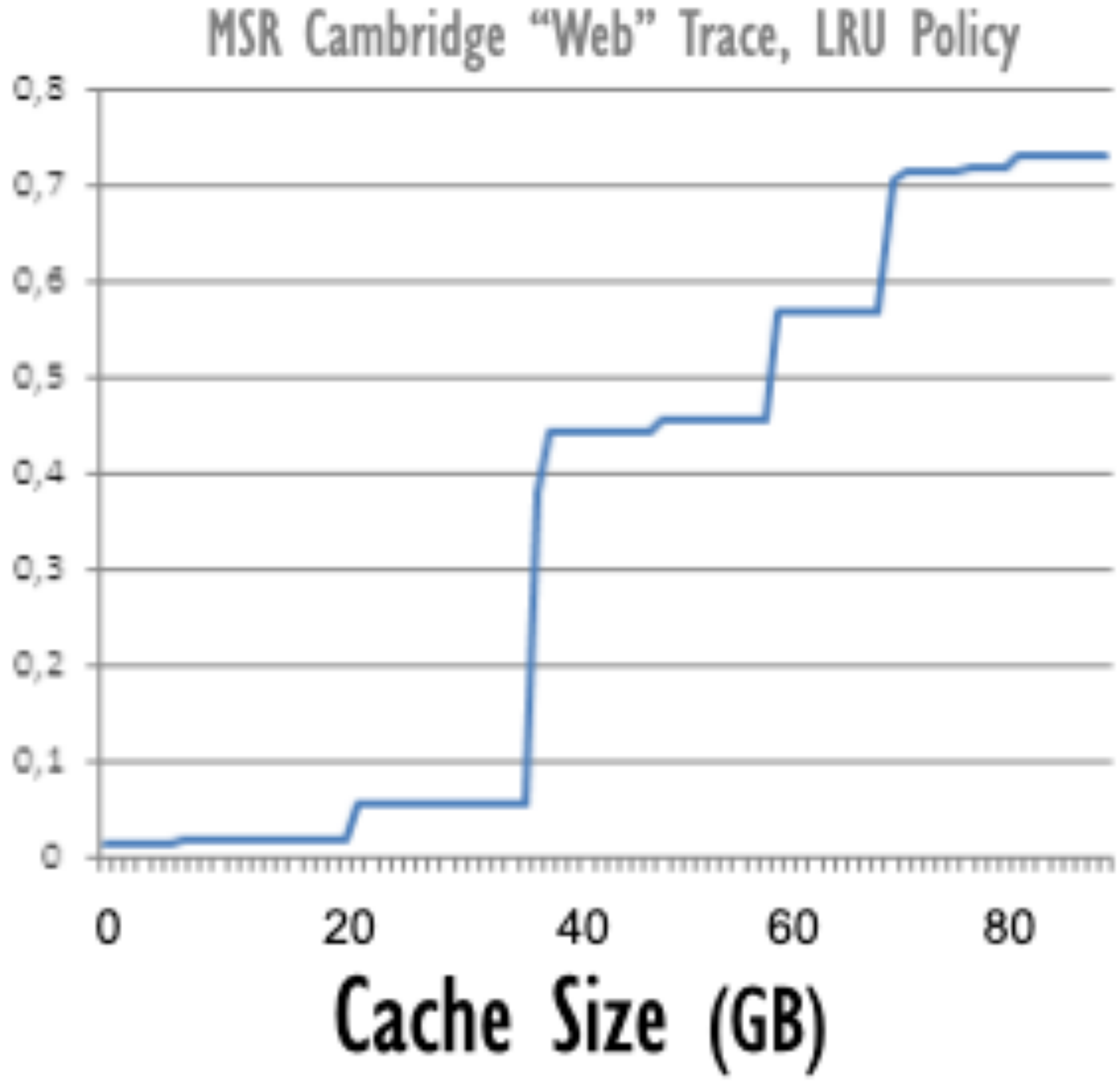


Image: [Druidi et al. 2015]

More Questions

How is my cache heuristic behaving?

- Most caches do not actually use LRU
  - e.g. Clock or ML heuristic approach

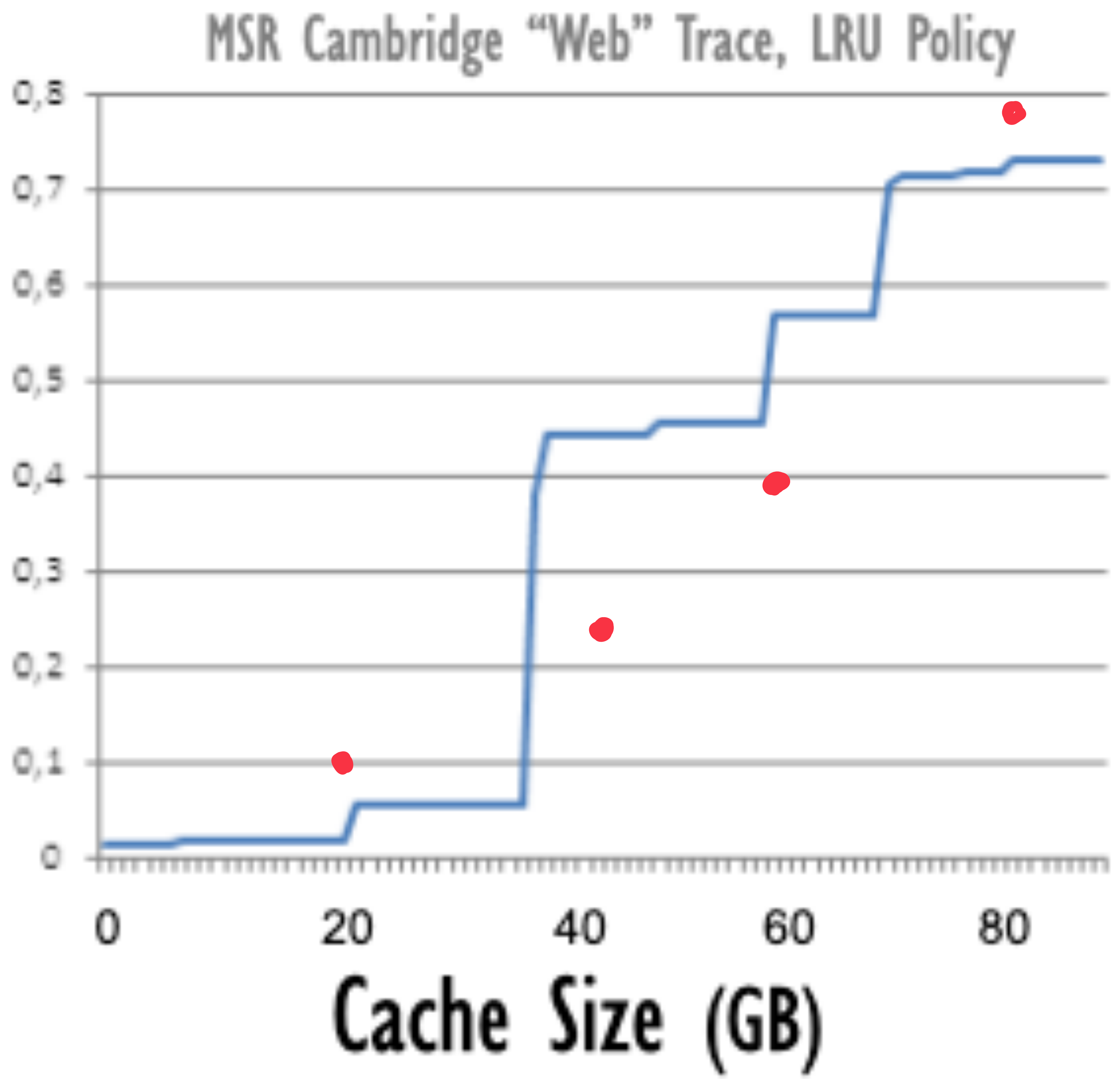


Image: [Druidi et al. 2015]

More Questions

How is my cache heuristic behaving?

- Most caches do not actually use LRU
  - e.g. Clock or ML heuristic approach
- To what extent is our eviction heuristic helping as compared to LRU?

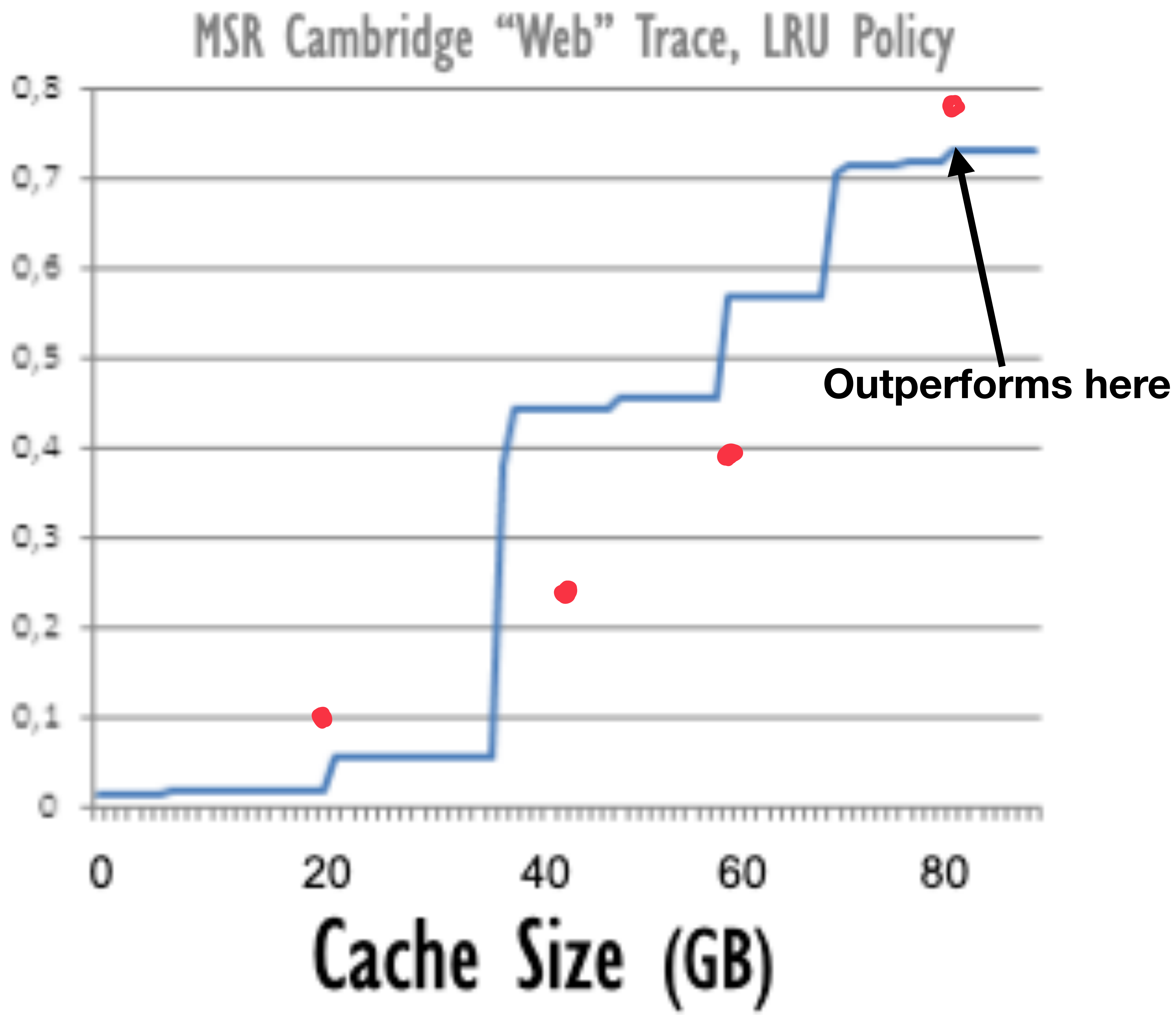


Image: [Druidi et al. 2015]

More Questions

How is my cache heuristic behaving?

- Most caches do not actually use LRU
  - e.g. Clock or ML heuristic approach
- To what extent is our eviction heuristic helping as compared to LRU?
  - Or is it hurting?

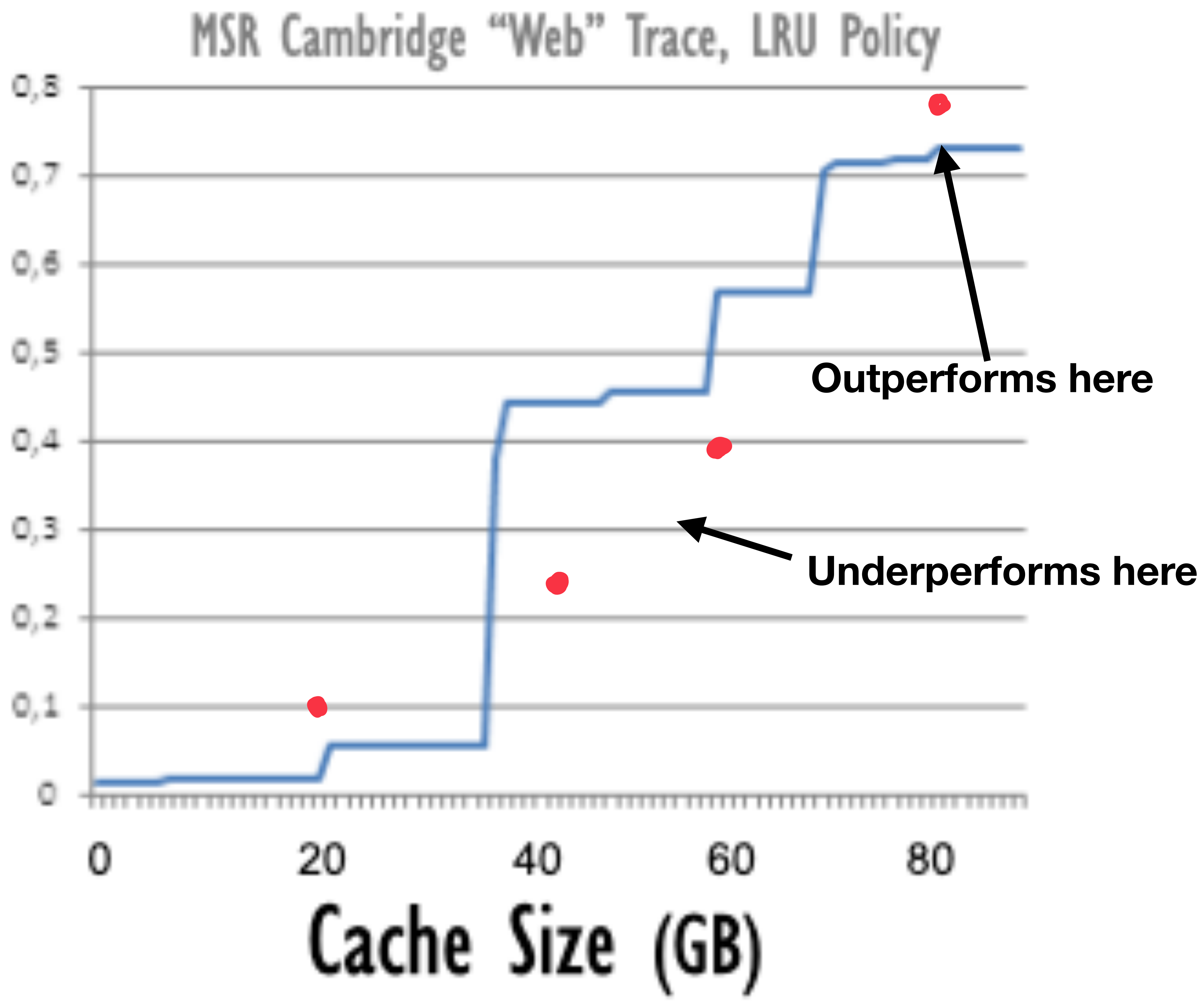


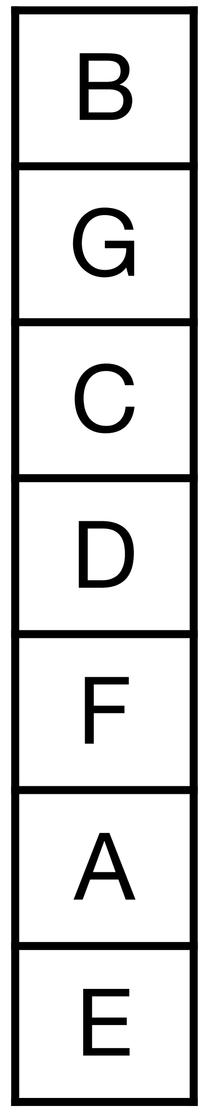
Image: [Druidi et al. 2015]

Augmented Tree Algorithms

State of the Art

- 1970 Mattson et al. compute LRU Hit-rate Curve from the stack
  - $O(n^2)$ time algorithm

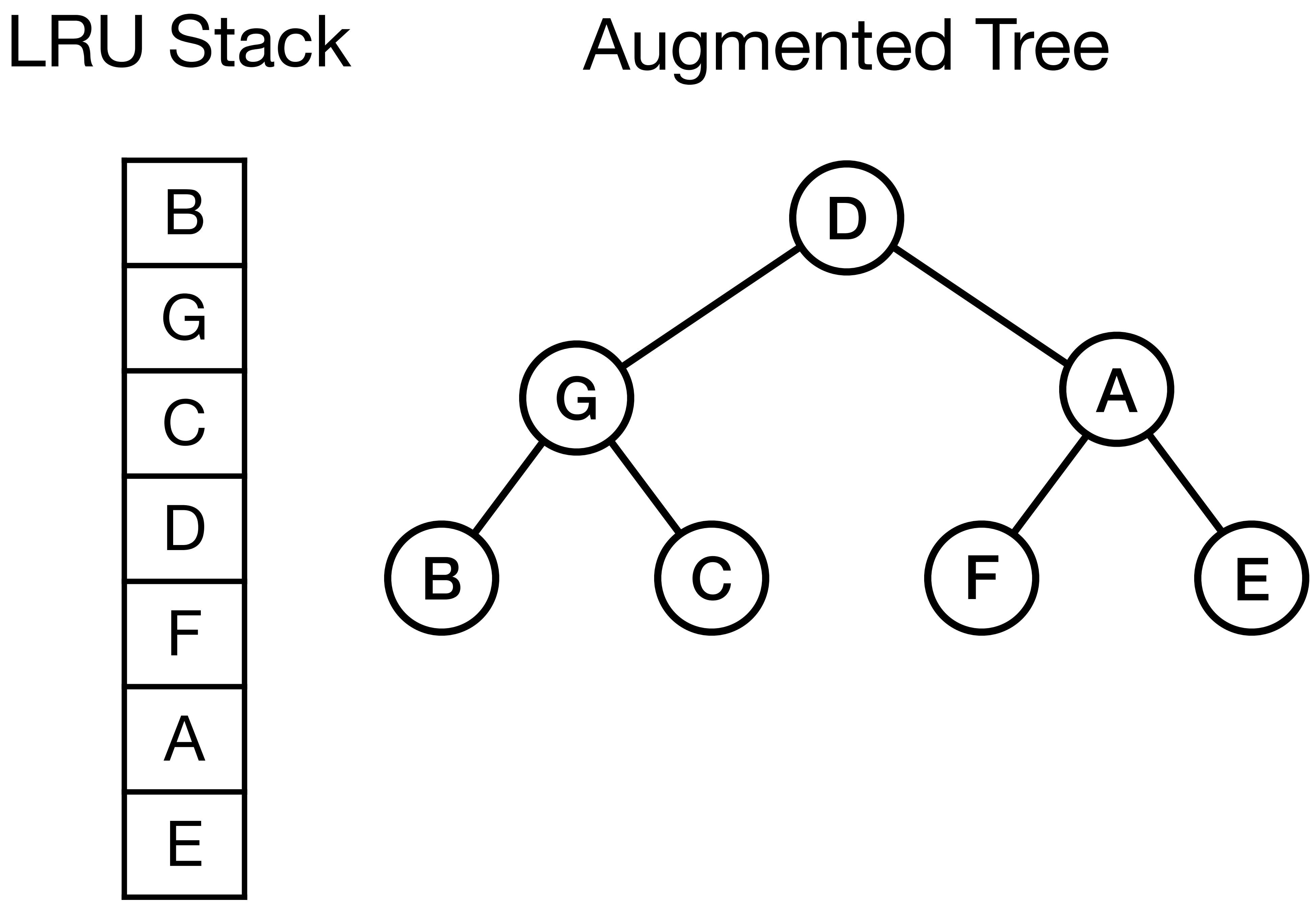
LRU Stack



Augmented Tree Algorithms

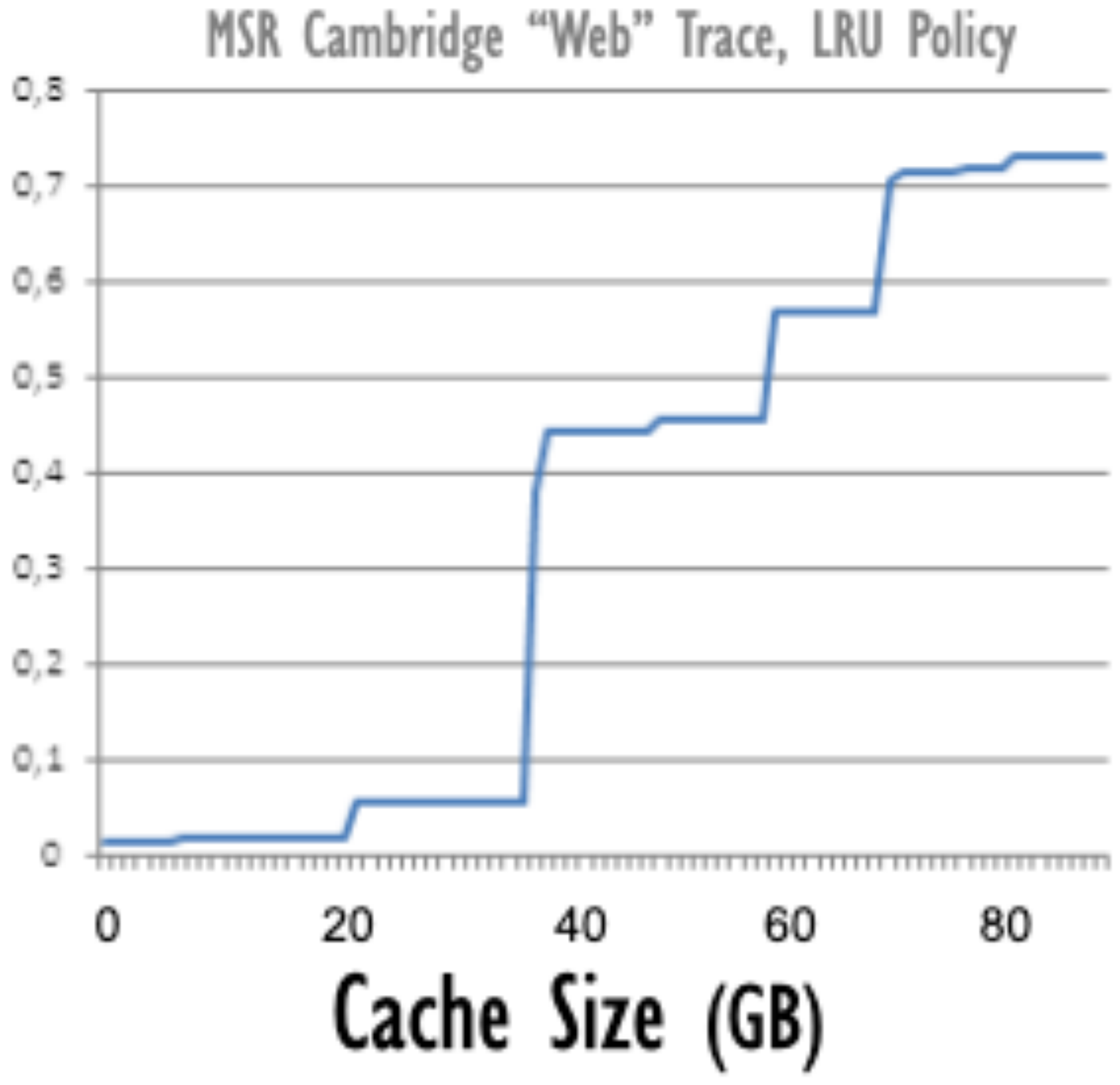
State of the Art

- 1970 Mattson et al. compute LRU Hit-rate Curve from the stack
  - $O(n^2)$ time algorithm
- 1975, Bennett and Kruskal store the stack as an augmented binary tree with order statistics
  - $O(n \log n)$ time algorithm
  - Best known RAM model complexity



This talk, Hit-rate Curve Computation In:

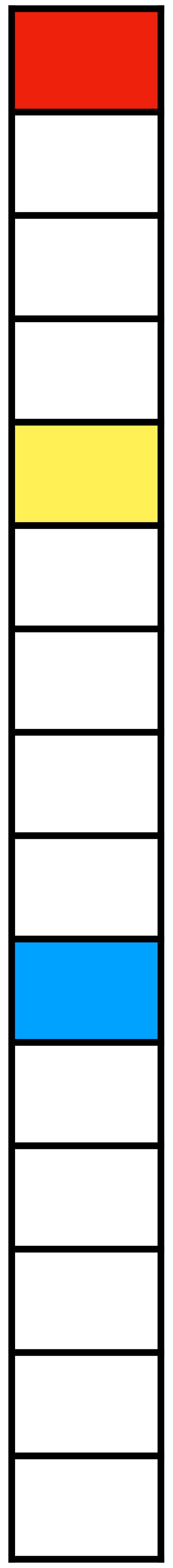
- The external-memory model
  - $\text{sort}(n) = O\left(\frac{n}{B} \log_{M/B} \frac{n}{B}\right)$ I/Os
- Parallelism
  - $O(\log^2 n)$ span
  - $O(n \log n)$ work



Lack of Locality

A Fundamental Challenge

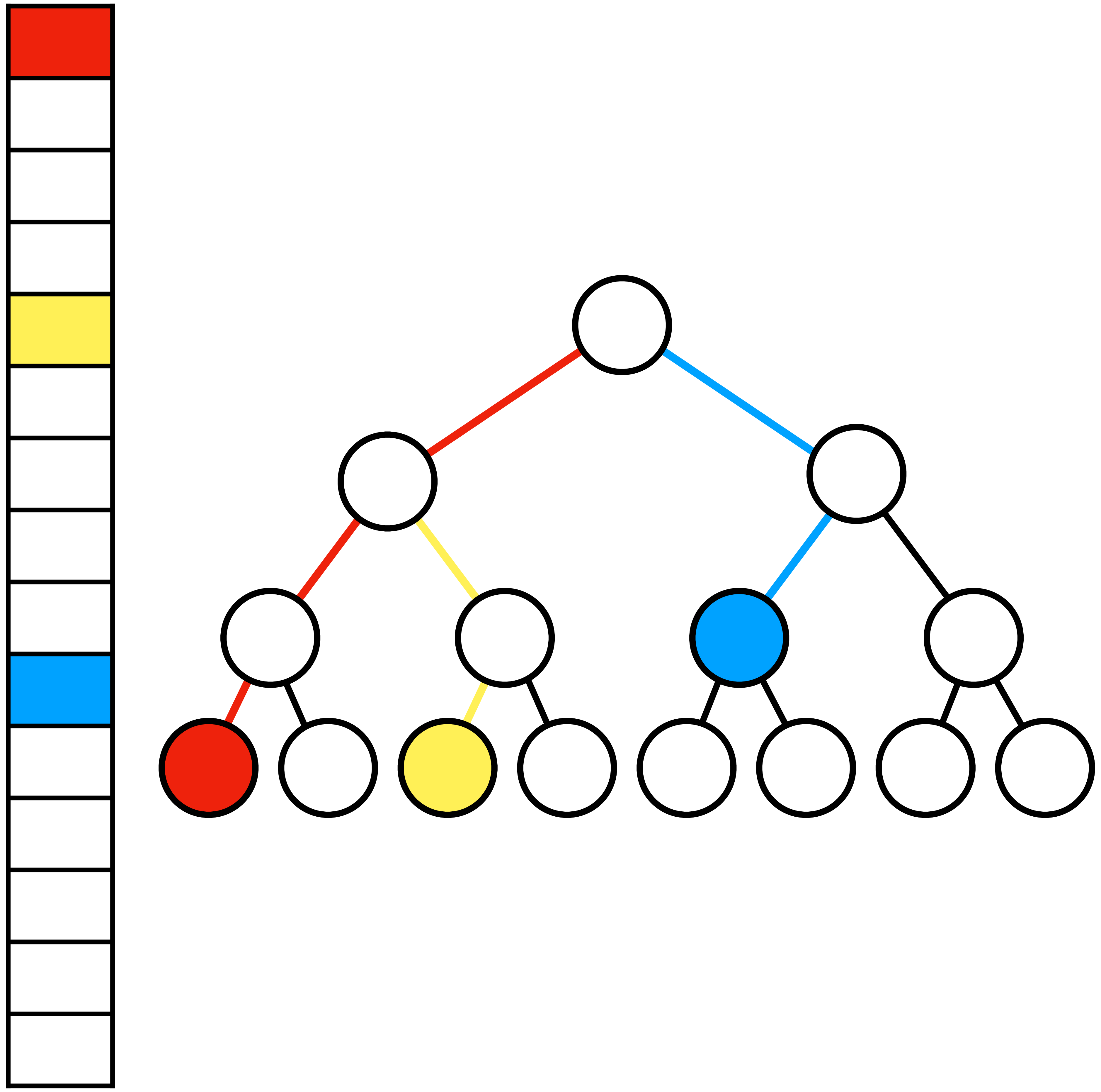
- Accesses to the LRU stack may be random



Lack of Locality

A Fundamental Challenge

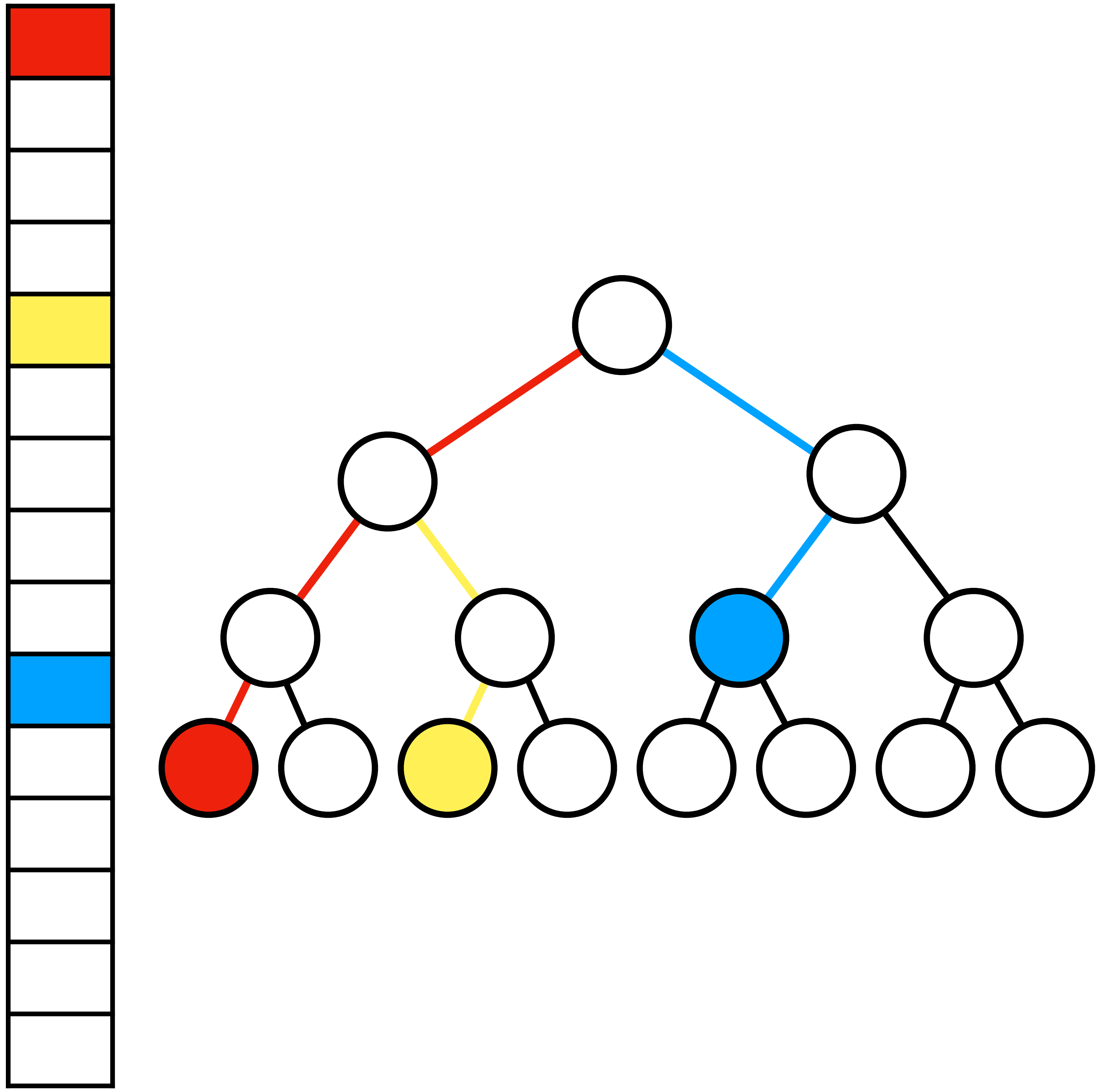
- Accesses to the LRU stack may be random
- Augmented tree: $O(\log n)$ cache misses per request
    - $O(n \log n)$ I/Os in total in EM model



Lack of Locality

A Fundamental Challenge

- Accesses to the LRU stack may be random
- Augmented tree: $O(\log n)$ cache misses per request
  - $O(n \log n)$ I/Os in total in EM model
- Time to compute hit-rate curve is **100x greater** than running time of program



Parallelism

Necessary for practical performance

- We want to keep pace with a cache that may be receiving requests from multiple processes or users

Parallelism

Necessary for practical performance

- We want to keep pace with a cache that may be receiving requests from multiple processes or users

- Existing work “PARDA”: Achieves parallelism at cost of additional memory
  - Chunk up requests sequence and use multiple trees

PARDA: [Niu et al. 2012]

Parallelism

Necessary for practical performance

- We want to keep pace with a cache that may be receiving requests from multiple processes or users

- Existing work "PARDA": Achieves parallelism at cost of additional memory
  - Chunk up requests sequence and use multiple trees

- Perhaps not surprising, we need both parallelism and data locality

PARDA: [Niu et al. 2012]

Increment-and-Freeze

The Increment-and-Freeze Algorithm

LRU hit-rate curves with locality and parallelism

- Can surprisingly solve Hit-rate Curve without representing a LRU-stack
  - Accesses to the stack are fundamentally random

The Increment-and-Freeze Algorithm

LRU hit-rate curves with locality and parallelism

- Can surprisingly solve Hit-rate Curve without representing a LRU-stack
  - Accesses to the stack are fundamentally random

- Increment-and-Freeze uses a divide-and-conquer strategy to compute the stack depth of every request

Finding Stack Distances

- Initialize an Array $A[n]$ to all zeros. Indexed by 1
  - When the algorithm concludes, A holds the stack distance of all n requests

Finding Stack Distances

- Initialize an Array $A[n]$ to all zeros. Indexed by 1
  - When the algorithm concludes, A holds the stack distance of all n requests

- **Stack distance**: the number of unique requests between an occurrence of a page and its next occurrence.
  - ABBBA: stack distance of first A is 2
  - ABCDA: stack distance of first A is 4

Operations

- Increment-and-Freeze consists of two operations

Operations

Surprising Stuff

- Increment-and-Freeze consists of two operations
  - Increment(i, j, r): Increment array values $[i, j)$ by r
  - Freeze(i): Freeze array value $A[i]$, prevent it from being incremented more

Operations

- Increment-and-Freeze consists of two operations
  - Increment(i, j, r): Increment array values $[i, j)$ by r
  - Freeze(i): Freeze array value $A[i]$, prevent it from being incremented more
- Goal: After processing all operations, A contains the stack distance of each request
  - Trivial to construct hit-rate curve from stack distances

Building Operations

- Each request j becomes $I(\text{prev}(j), j, 1)$ and $F(\text{prev}(j))$

- Example: ABEBA

  0 0 0 0 0 Initialize

Building Operations

- Each request j becomes $I(\text{prev}(j), j, 1)$ and $F(\text{prev}(j))$

- Example: ABEBA

  0 0 0 0 0 Initialize

  0 0 0 0 0 A: I(0,1,1) F(0)

Building Operations

- Each request j becomes $I(\text{prev}(j), j, 1)$ and $F(\text{prev}(j))$

- Example: ABEBA

  0 0 0 0 0 Initialize

  0 0 0 0 0 A: $I(0,1,1)$ $F(0)$

  1 0 0 0 0 B: $I(0,2,1)$ $F(0)$

Building Operations

- Each request j becomes $I(\text{prev}(j), j, 1)$ and $F(\text{prev}(j))$

- Example: ABEBA

  0 0 0 0 0 Initialize

  0 0 0 0 0 A: I(0,1,1) F(0)

  1 0 0 0 0 B: I(0,2,1) F(0)

  2 1 0 0 0 E: I(0,3,1) F(0)

Building Operations

- Each request j becomes $I(\text{prev}(j), j, 1)$ and $F(\text{prev}(j))$

- Example: ABEBA

  0 0 0 0 0 Initialize

  0 0 0 0 0 A: I(0,1,1) F(0)

  1 0 0 0 0 B: I(0,2,1) F(0)

  2 1 0 0 0 E: I(0,3,1) F(0)

  2 2 1 0 0 B: I(2,4,1) F(2)

Building Operations

- Each request j becomes $I(\text{prev}(j), j, 1)$ and $F(\text{prev}(j))$

- Example: ABEBA

0 0 0 0 0 Initialize

0 0 0 0 0 A: I(0,1,1) F(0)

1 0 0 0 0 B: I(0,2,1) F(0)

2 1 0 0 0 E: I(0,3,1) F(0)

2 2 1 0 0 B: I(2,4,1) F(2)

3 2 2 1 0 A: I(1,5,1) F(1)

Divide and Conquer Structure

- $O(n^2)$ time because increments are expensive
  - Need to merge increment operations
  - Can merge neighboring increments that affect the same range

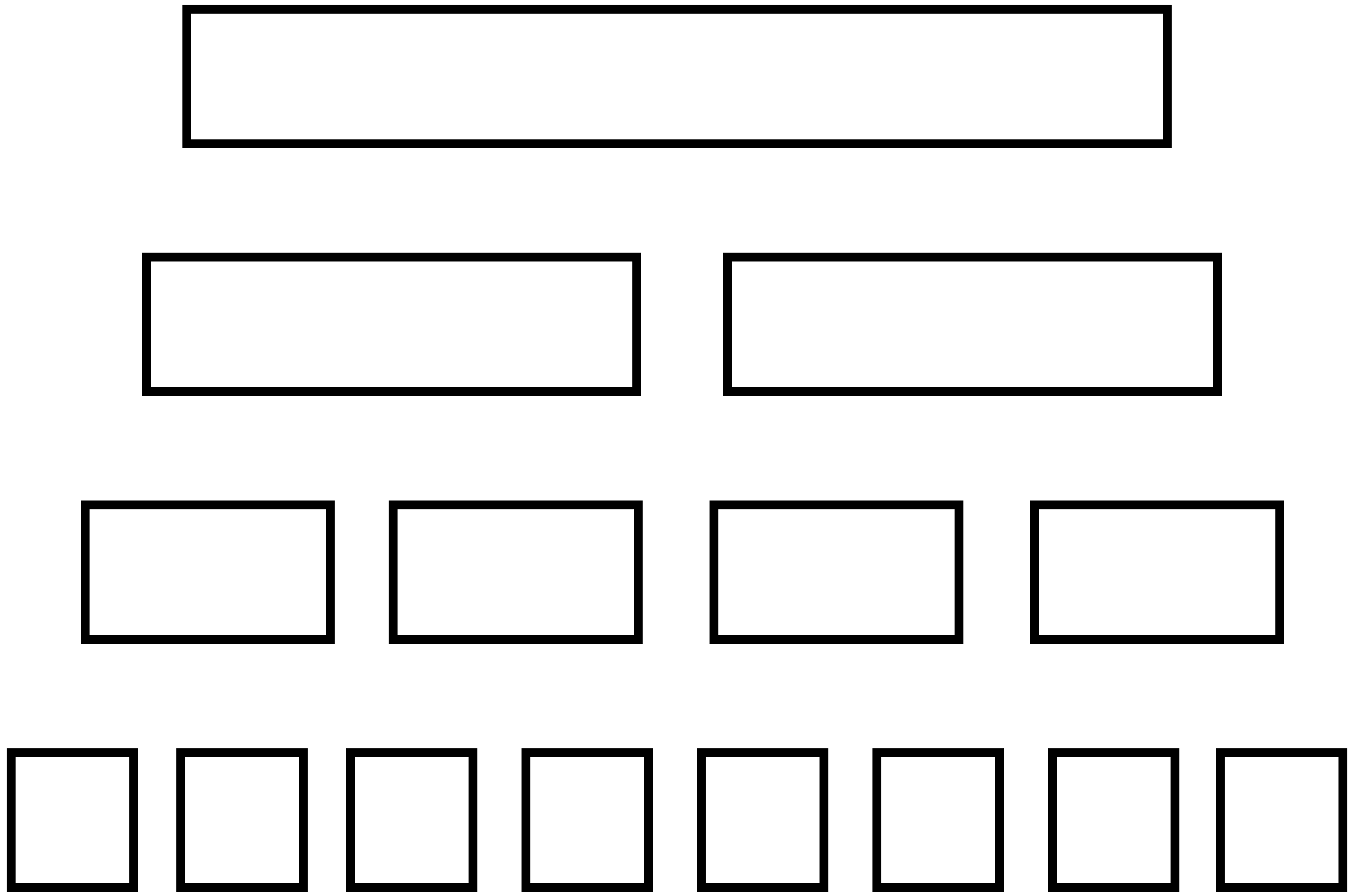
Divide and Conquer Structure

- $O(n^2)$ time because increments are expensive
  - Need to merge increment operations
  - Can merge neighboring increments that affect the same range

- Partition procedure divides a range of request indices in half
  - Operations are restricted to only affect their respective side of the partition
  - One Increment may become two

Divide and Conquer Structure

- Divide-and-conquer performed via repeated partitions
- Even though Increments may split
    - $O(n)$ operations per level



Increment-and-Freeze Complexity

The base algorithm

- RAM model: $O(n)$ operations per level, $O(\log n)$ levels
  - $O(n \log n)$ time total

Increment-and-Freeze Complexity

The base algorithm

- RAM model: $O(n)$ operations per level, $O(\log n)$ levels
  - $O(n \log n)$ time total

- External memory model: Scan at each level, $O(\frac{n}{B} \log n)$ I/Os

Increment-and-Freeze Complexity

The base algorithm

- RAM model: $O(n)$ operations per level, $O(\log n)$ levels
  - $O(n \log n)$ time total
- External memory model: Scan at each level, $O(\frac{n}{B} \log n)$ I/Os
- PRAM model: single-threaded partition, subproblems in other threads, thus $O(n)$ span and $O(n \log n)$ work

Lightning Round

Theoretical Extensions

See the paper :)

- External Memory: $\text{sort}(n) = O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ I/Os

Theoretical Extensions

See the paper :)

- External Memory: sort$(n) = O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ I/Os

- PRAM: Span $O(\log^2 n)$, work $O(n \log n)$
  - Cluster sum: cool application of parallel prefix sums

Implementation

See the paper x2 :)

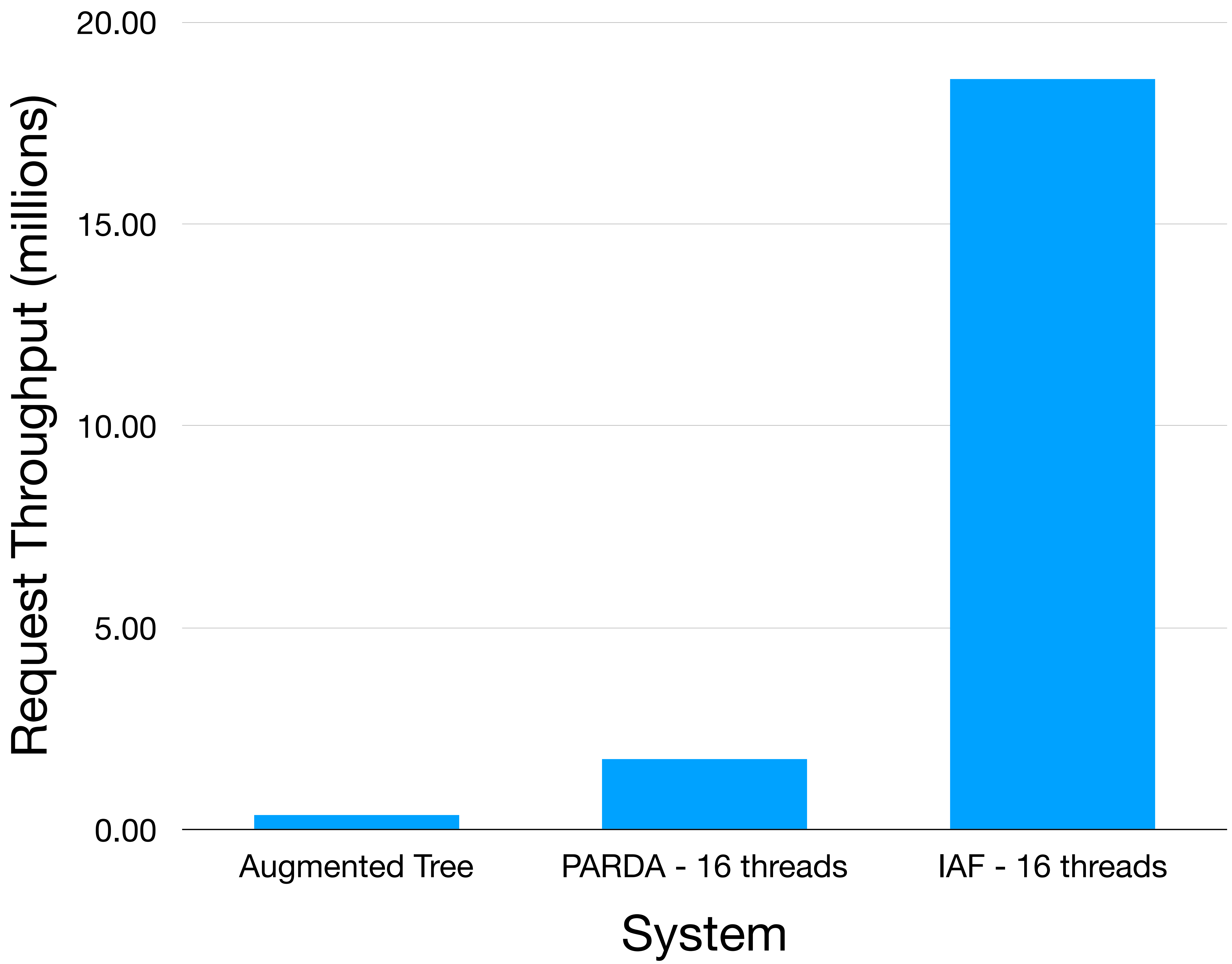
- We implemented the base Increment-and-Freeze algorithm

- Highly optimized via a number of cool tricks
  - Faster! Uses less memory!

Results

See the paper x3 :)

- Single-threaded
  - 9x faster than augmented tree
  - 8x faster than splay tree
- Cuts a 13 hour computation down to only 12 minutes



Conclusion

- Increment-and-Freeze
    - Computing LRU hit-rate curves with data locality and parallelism
- Everyone operating a cache should have real-time telemetry
    - This work has the potential to enable real-time cache analysis

More Slides

Operations

Example

- Request sequence: ABA
  - A -> $I(0,1,1)$, $F(0)$
  - B -> $I(0,2,1)$, $F(0)$
  - A -> $I(1,3,1)$, $F(1)$

- Full op sequence: $I(0,1,1)$, $F(0)$, $I(0,2,1)$, $F(0)$, $I(1,3,1)$, $F(1)$

Sampling

- Efficient approaches for computing LRU hit-rate curves down sample the key space. No quality guarantees for curve

- If we are trying to understand why our paging heuristic is underperforming, sampling may hide the answer.

- Increment-and-Freeze composes with sampling, further improving performance

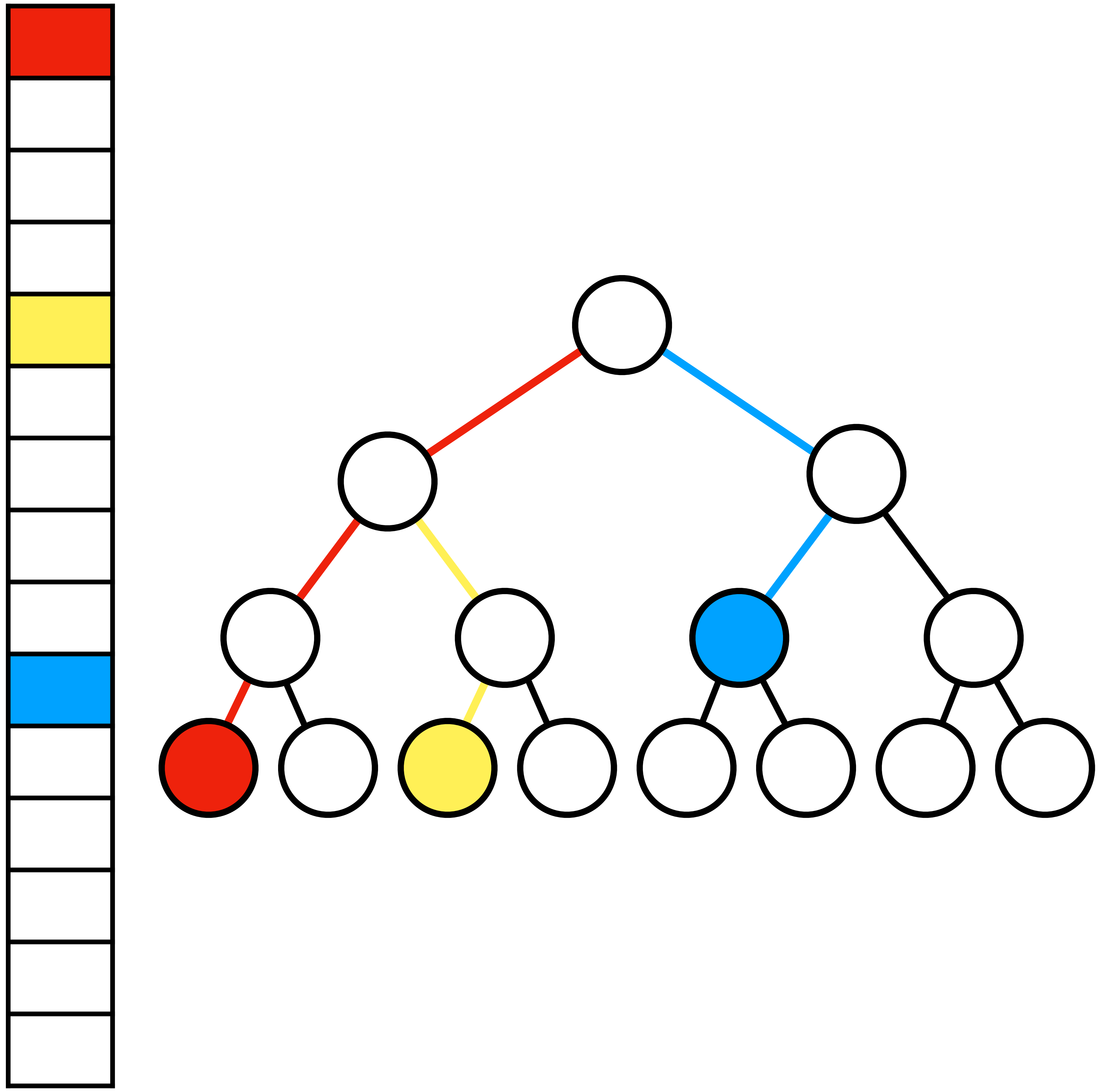
Lack of Locality

Why Hit-rate Curve Computation is 100x Slower

- Example: Building a hit rate curve for L3 cache

- At most 1 cache miss per access when running executable

- Versus $O(\log n)$ cache misses per access when producing the hit-rate curve!



Operations

Creating operations from requests

- prev(j): The index of the previous request that references the same page as j
  - For example: ABCAC, prev$(4) = 1$

Comparison with PARDA

Comparable speedup without memory cost