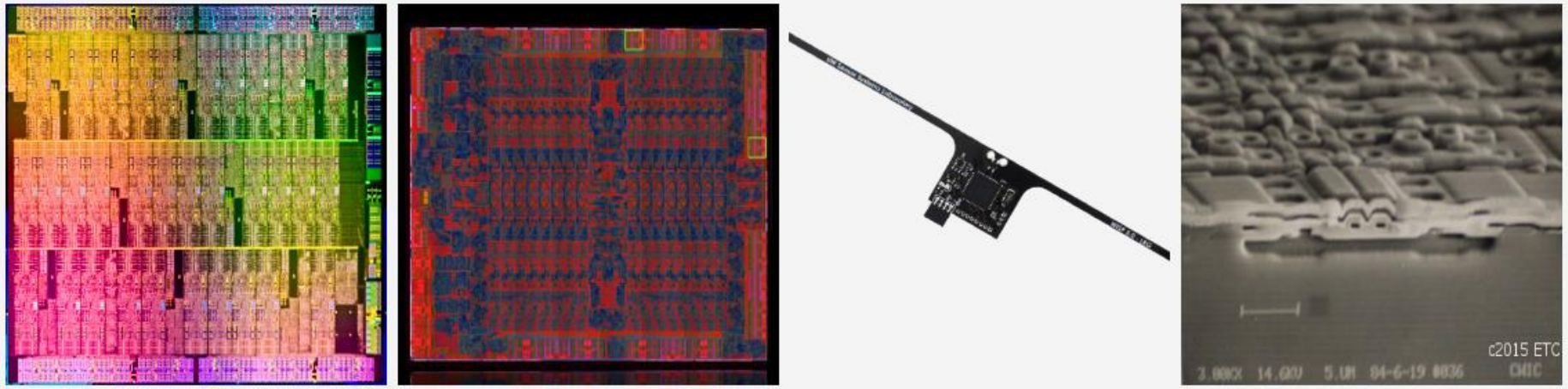
18-344: Computer Systems and the Hardware-Software Interface

Fall 2023



Course Description

Lecture 16: Sparsity, *cont*

This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

Credit: Brandon Lucia

Today: Sparse Problems

- Hardware and software strategies for optimizing sparse problems

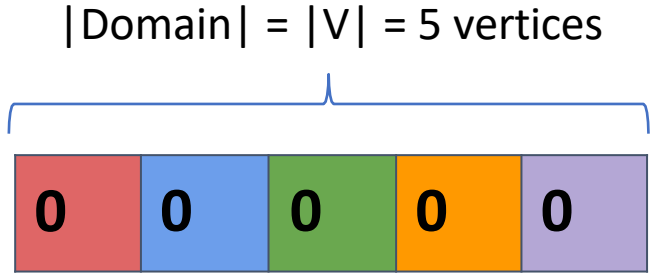
(with acknowledgements to Vignesh Balaji, CMU ECE PhD 2021, now at Nvidia for contributions to this material)

Propagation Blocking: Optimizing Sparse Irregular Writes to Improve Cache Locality

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache

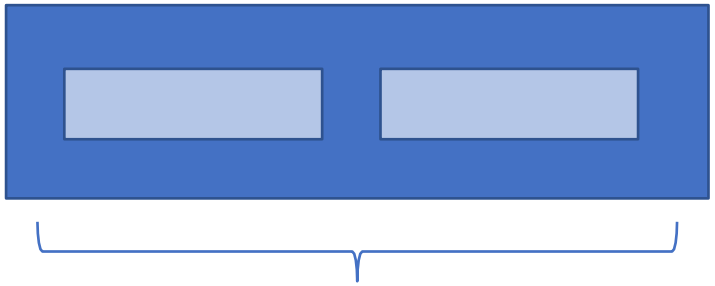
0	1
2	0
1	0
0	2
2	3
0	4
0	3

COO
(EdgeList)



Recall: irregular accesses into vertex data array based on e.dst *which are essentially random*

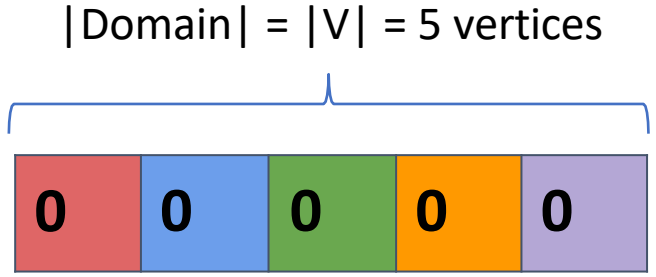
Bad for the cache: the size of the *domain* of vertex data array entries is |V|, but the cache holds only |C| << |V| entries



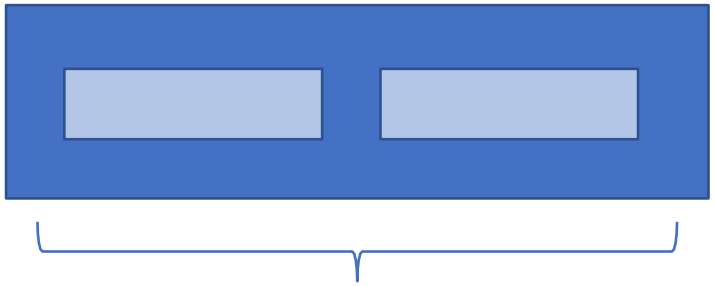
Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache

0	1
2	0
1	0
0	2
2	3
0	4
0	3

COO
(EdgeList)



Bad for the cache: the size of the *domain* of vertex data array entries is |V|, but the cache holds only |C| << |V| entries

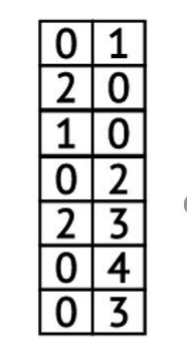


Recall: irregular accesses into vertex data array based on e.dst *which are essentially random*

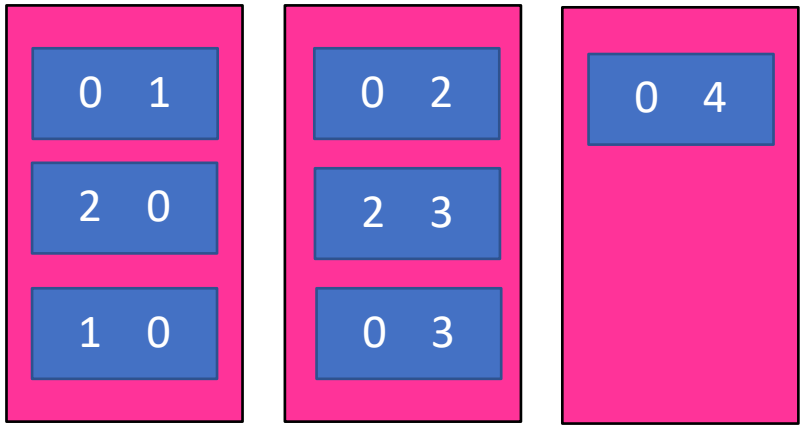
Key idea in propagation blocking: Limit the domain of updates to a *sub-space* of vertices, **V***, so that |V*| <= |C| and do multiple sub-spaces of V*s, so that all V*s together = V

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache

Create "Bins" that hold input elements (edges from the edge list)



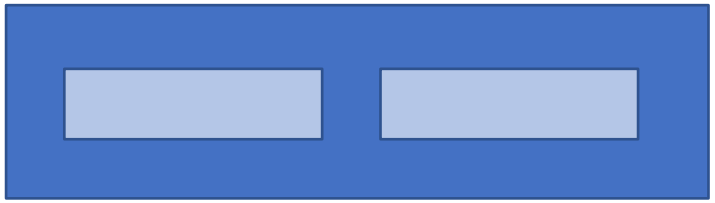
COO
(EdgeList)



Bin 0:
dst 0-1

Bin 1:
dst 2-3

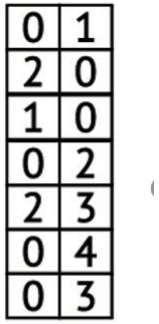
Bin 2:
dst 4-5



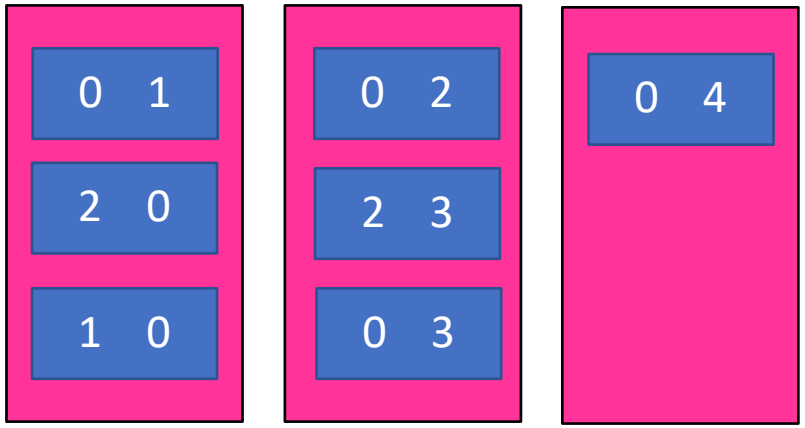
dstData

Remember: dstData[e.dst] ++
and e.dst is random, from edge list

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache



COO
(EdgeList)

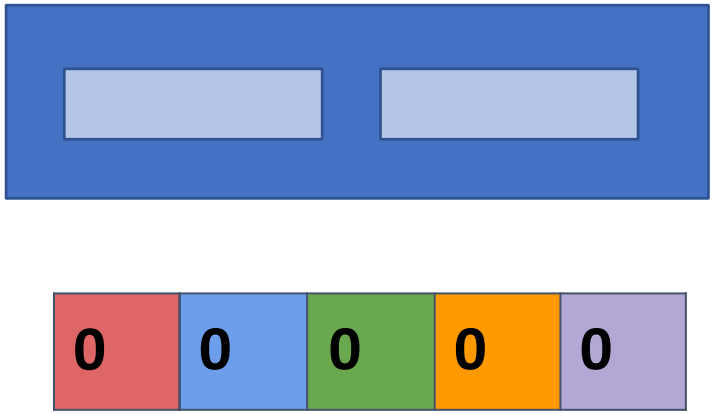


Bin 0:
dst 0-1

Bin 1:
dst 2-3

Bin 2:
dst 4-5

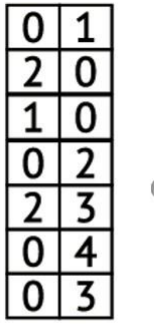
Execute the kernel for one bin at a time



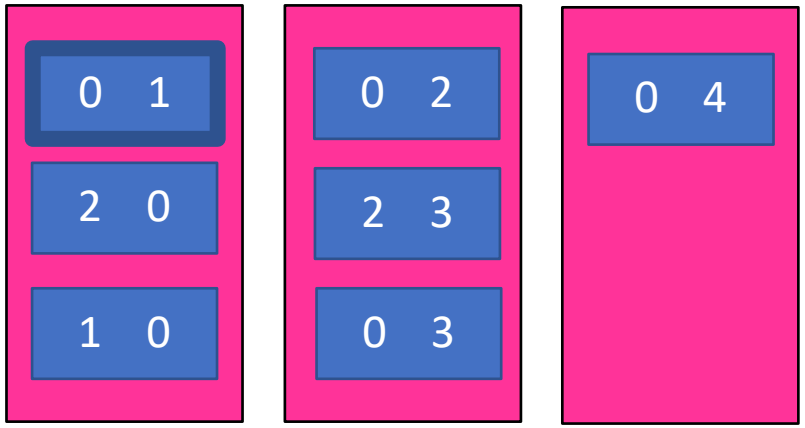
dstData

Remember: dstData[e.dst] ++
and e.dst is random, from edge list

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache



COO
(EdgeList)

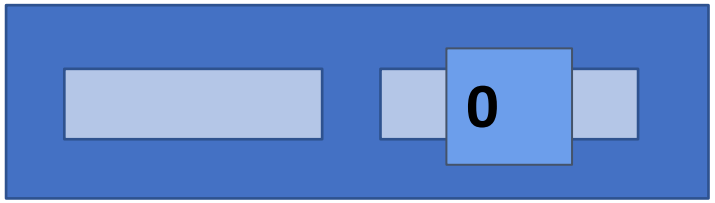


Bin 0:
dst 0-1

Bin 1:
dst 2-3

Bin 2:
dst 4-5

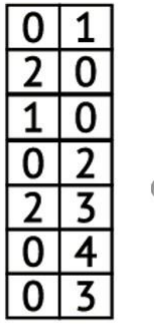
Execute the kernel for one bin at a time



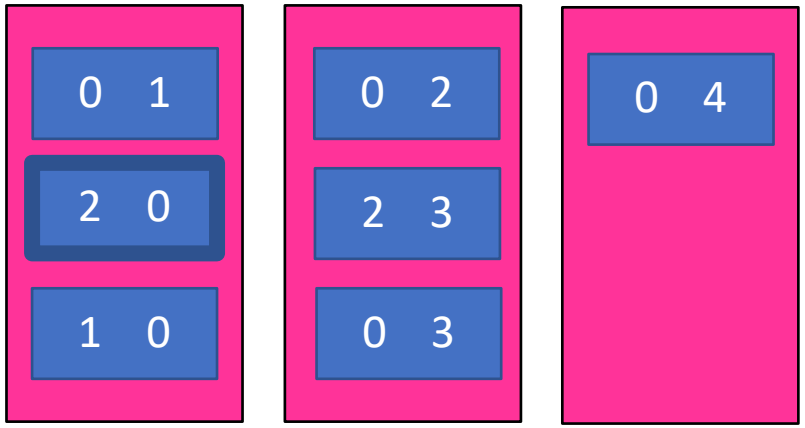
dstData

Remember: dstData[e.dst] ++
and e.dst is random, from edge list

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache



COO
(EdgeList)

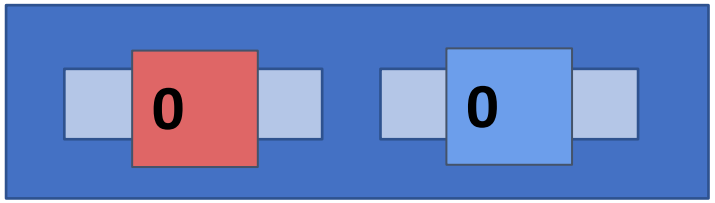


Bin 0:
dst 0-1

Bin 1:
dst 2-3

Bin 2:
dst 4-5

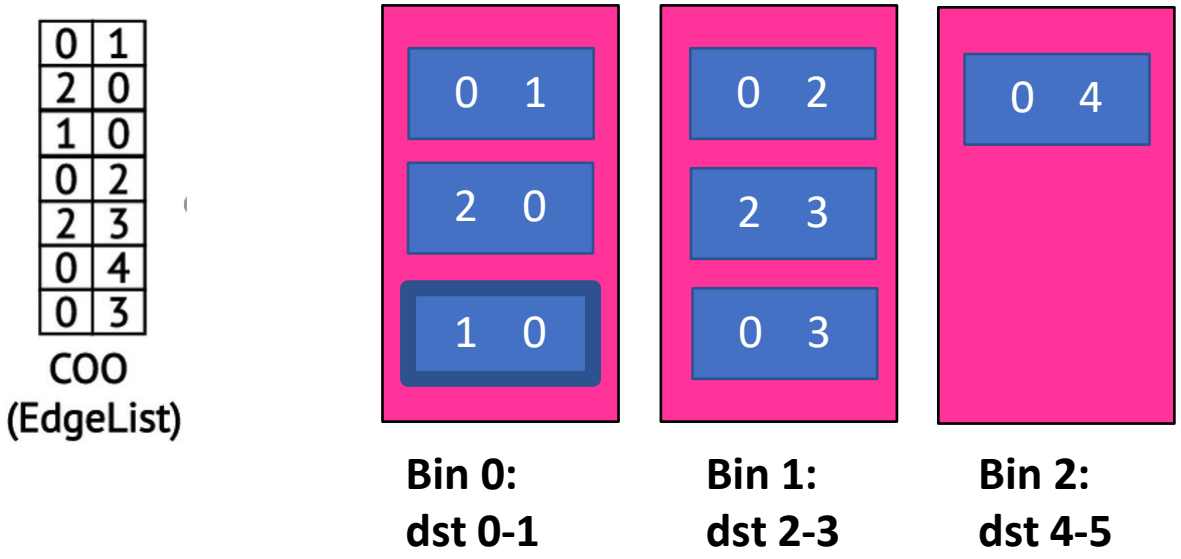
Execute the kernel for one bin at a time



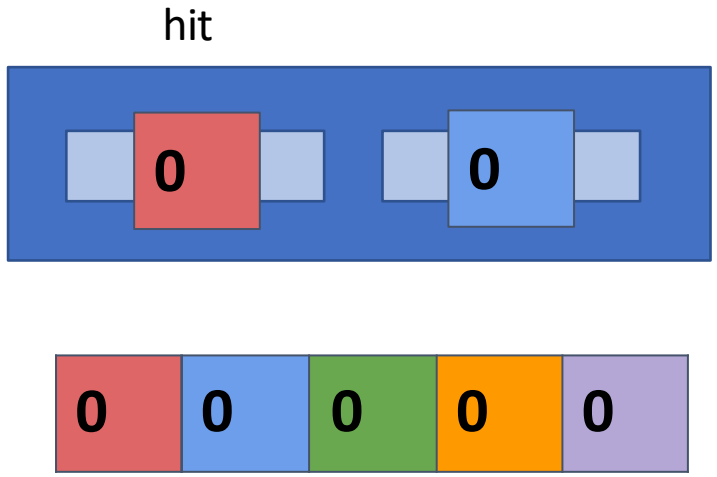
dstData

Remember: dstData[e.dst] ++
and e.dst is random, from edge list

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache



Execute the kernel for one bin at a time

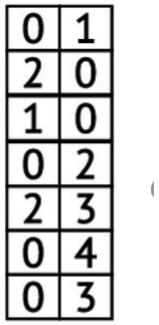


dstData

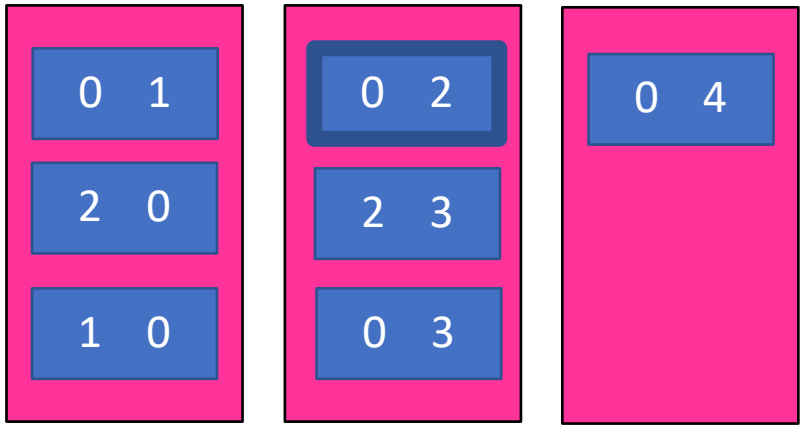
Remember: dstData[e.dst] ++

and e.dst is random, from edge list

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache



COO
(EdgeList)

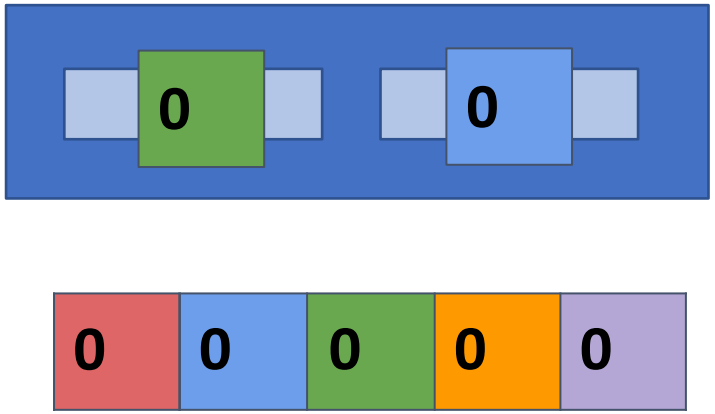


Bin 0:
dst 0-1

Bin 1:
dst 2-3

Bin 2:
dst 4-5

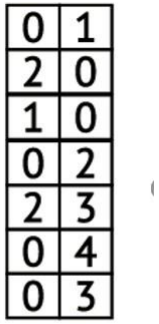
Execute the kernel for one bin at a time



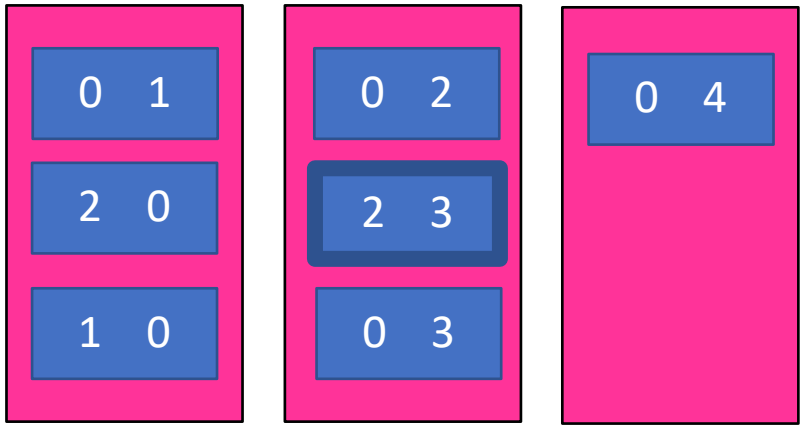
dstData

Remember: dstData[e.dst] ++
and e.dst is random, from edge list

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache



COO
(EdgeList)

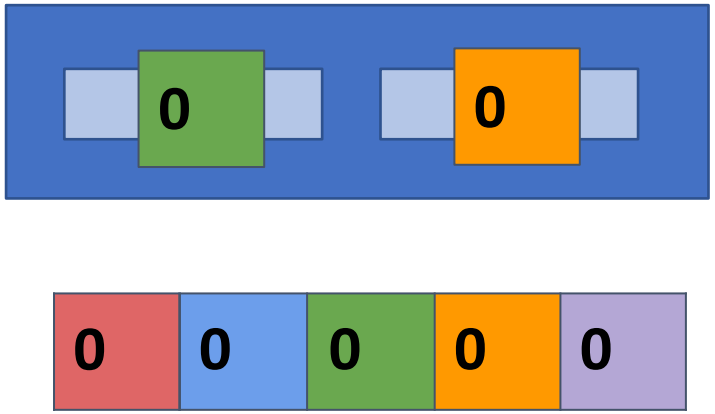


Bin 0:
dst 0-1

Bin 1:
dst 2-3

Bin 2:
dst 4-5

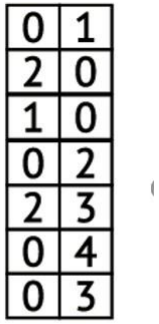
Execute the kernel for one bin at a time



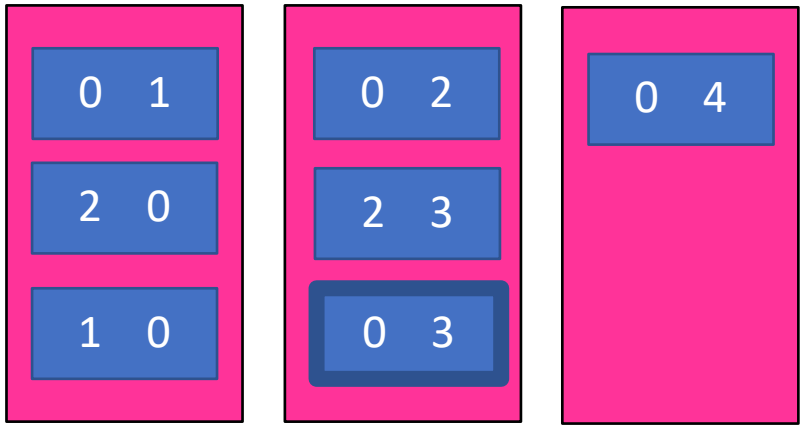
dstData

Remember: dstData[e.dst] ++
and e.dst is random, from edge list

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache



COO
(EdgeList)

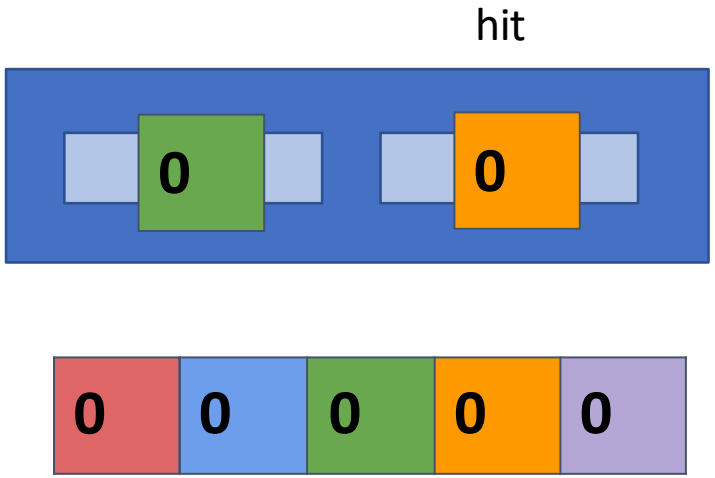


Bin 0:
dst 0-1

Bin 1:
dst 2-3

Bin 2:
dst 4-5

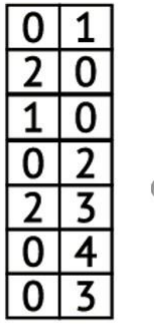
Execute the kernel for one bin at a time



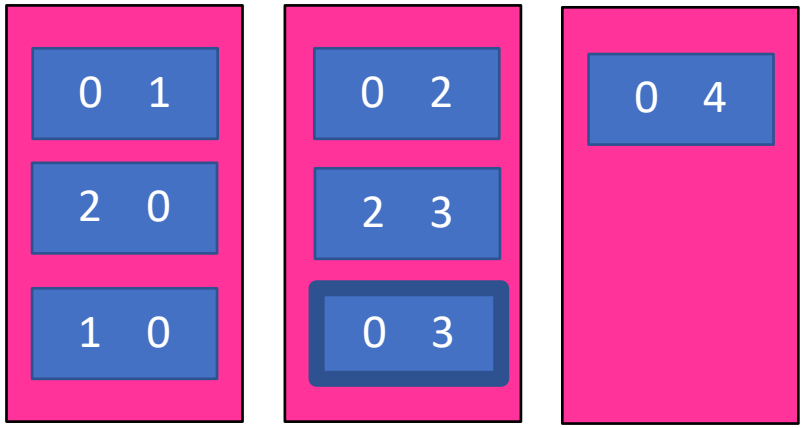
dstData

Remember: dstData[e.dst] ++
and e.dst is random, from edge list

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache



COO
(EdgeList)

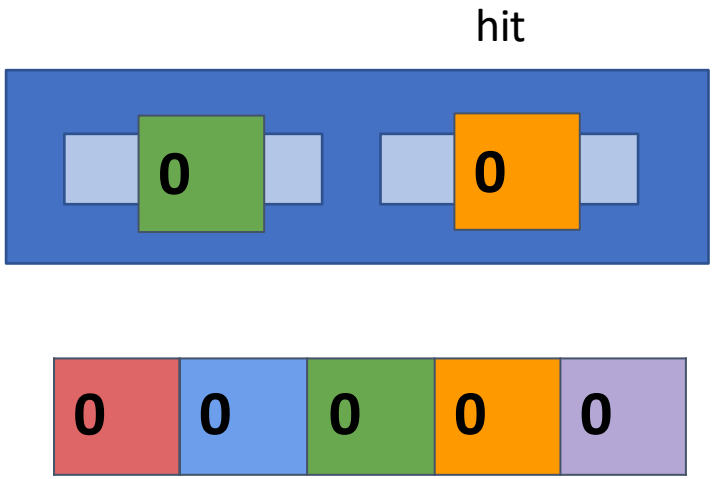


Bin 0:
dst 0-1

Bin 1:
dst 2-3

Bin 2:
dst 4-5

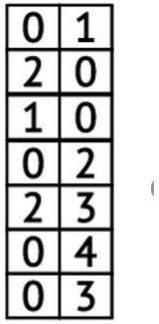
How to decide how many vertices go in each of your Propagation Blocker's bins?



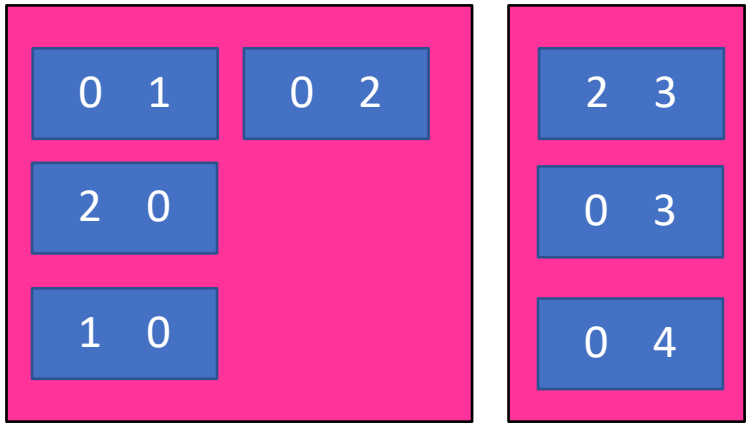
dstData

Remember: dstData[e.dst] ++
and e.dst is random, from edge list

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache



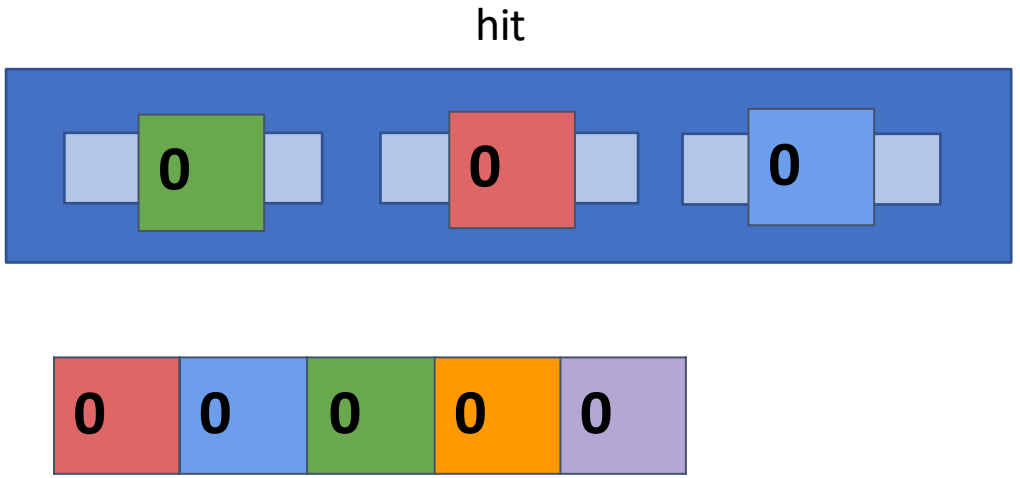
COO
(EdgeList)



Bin 0:
dst 0-2

Bin 1:
dst 3-5

Match destinations per bin to number of vertices worth of dstData that can fit in cache at one time

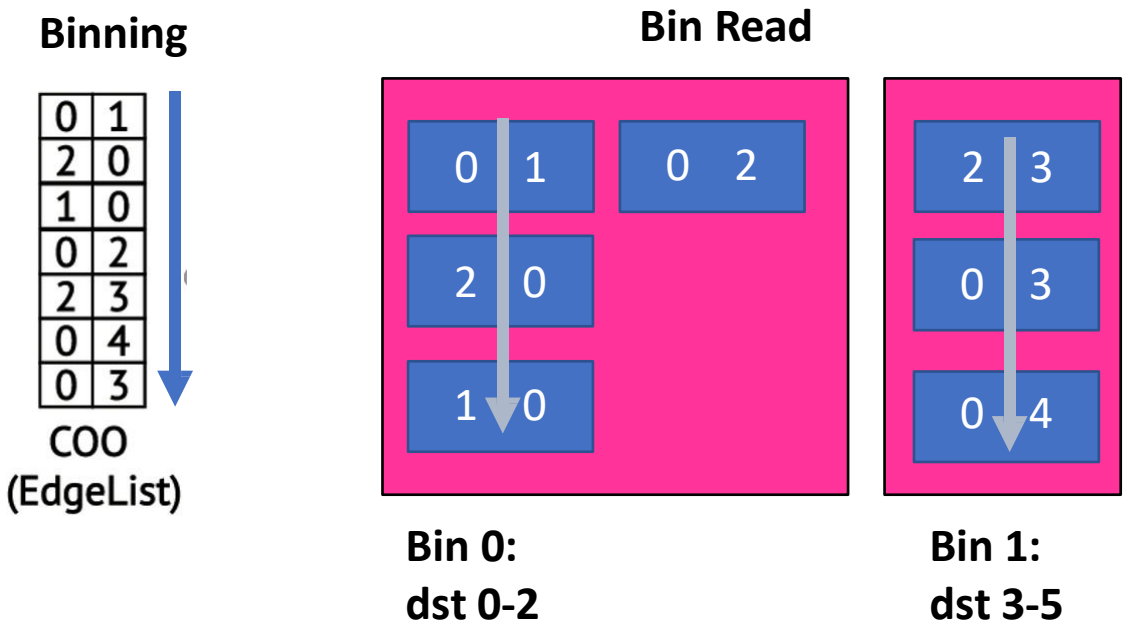


dstData

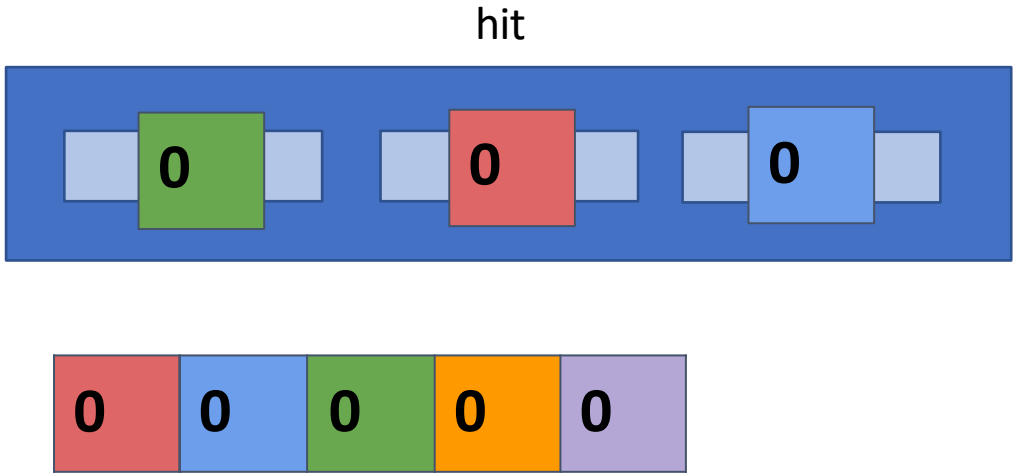
Remember: dstData[e.dst] ++
and e.dst is random, from edge list

Propagation Blocking: Performance Analysis

Traverse the edge list twice instead of once



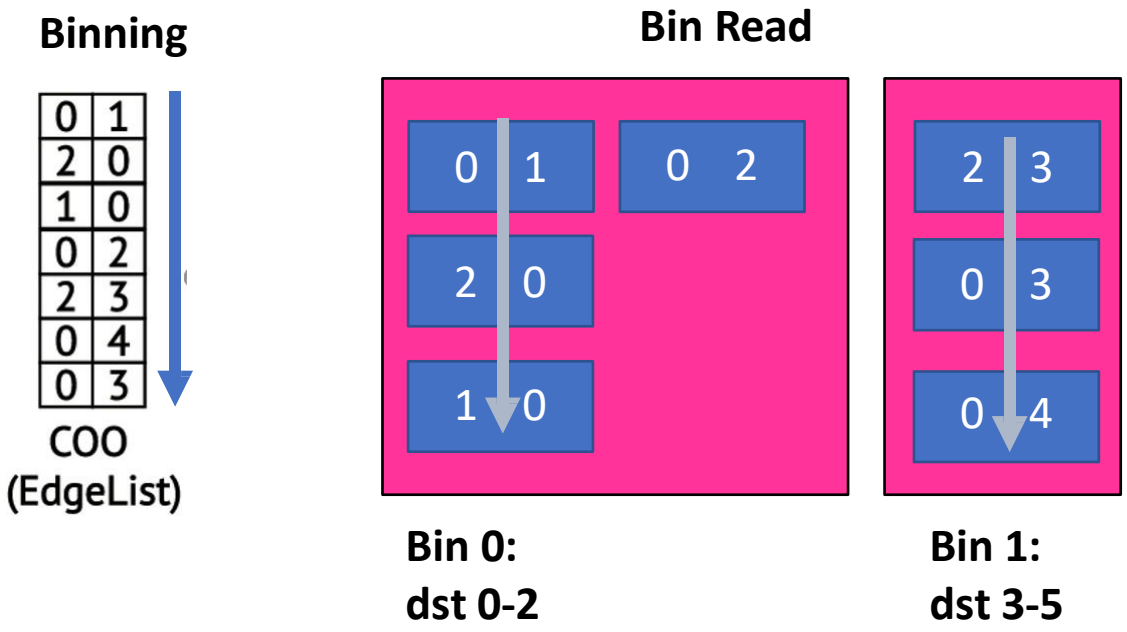
All locations written fit in cache! Compulsory misses on dstData[] only: all the rest hit.



Remember: dstData[e.dst] ++
and e.dst is random, from edge list

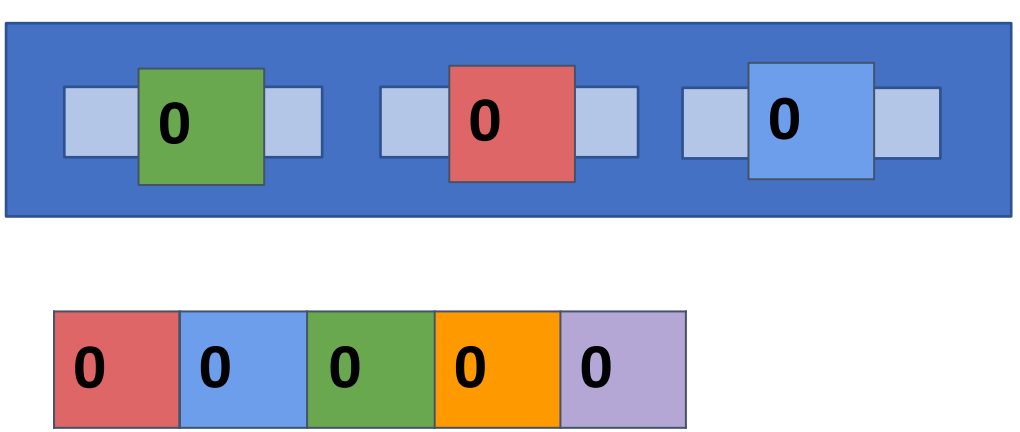
Propagation Blocking: Performance Analysis

Traverse the edge list twice instead of once



What about the performance of reading the edge list during binning?

All locations written fit in cache! Compulsory misses on dstData[] only: all the rest hit.

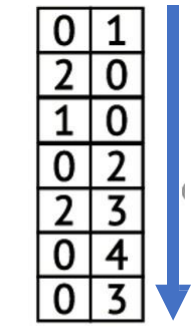


Remember: dstData[e.dst] ++
and e.dst is random, from edge list

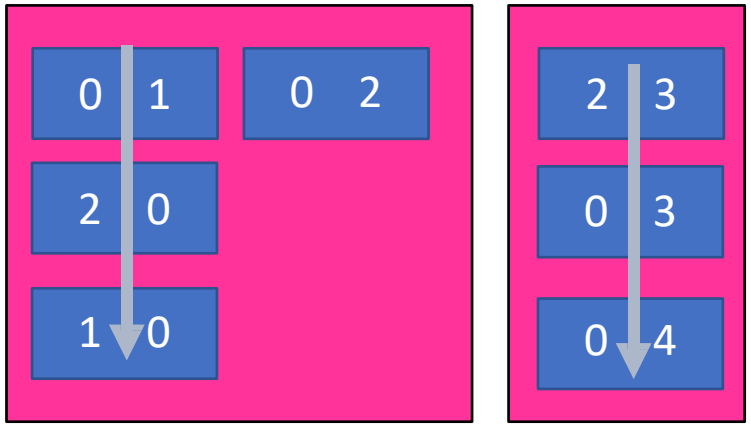
Propagation Blocking: Performance Analysis

Traverse the edge list twice instead of once

Binning



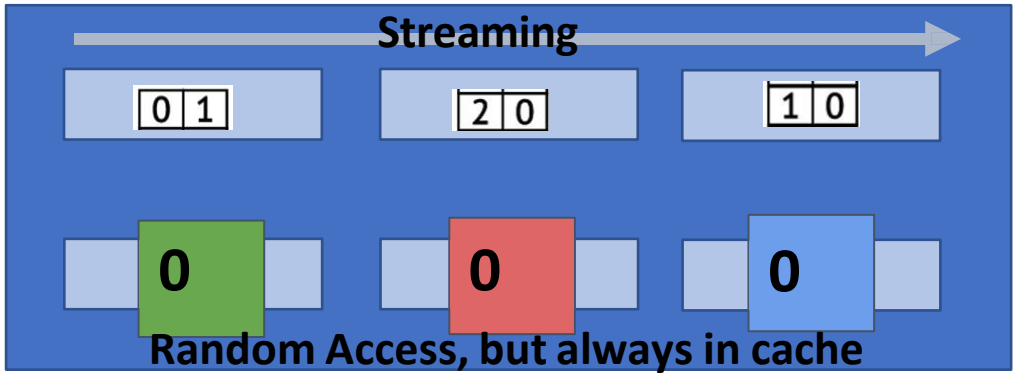
Bin Read



Bin 0:
dst 0-2

Bin 1:
dst 3-5

Usually save a little space in cache for *streaming edge list* data. Easy to cache.



dstData

Remember: dstData[e.dst] ++
and e.dst is random, from edge list

What about propagation blocking for irregular reads?

Propagation Blocking

```
PropagationBlocking_EdgeCount(EdgeList E){

  Bins B[];
  for edge in E{
    add_to_bin( find_bin(edge) )
  }

  for bin in B{
    for e in bin{
      dstData[e.dst]++
    }
  }

}
```

Reducing Pagerank Communication via Propagation Blocking

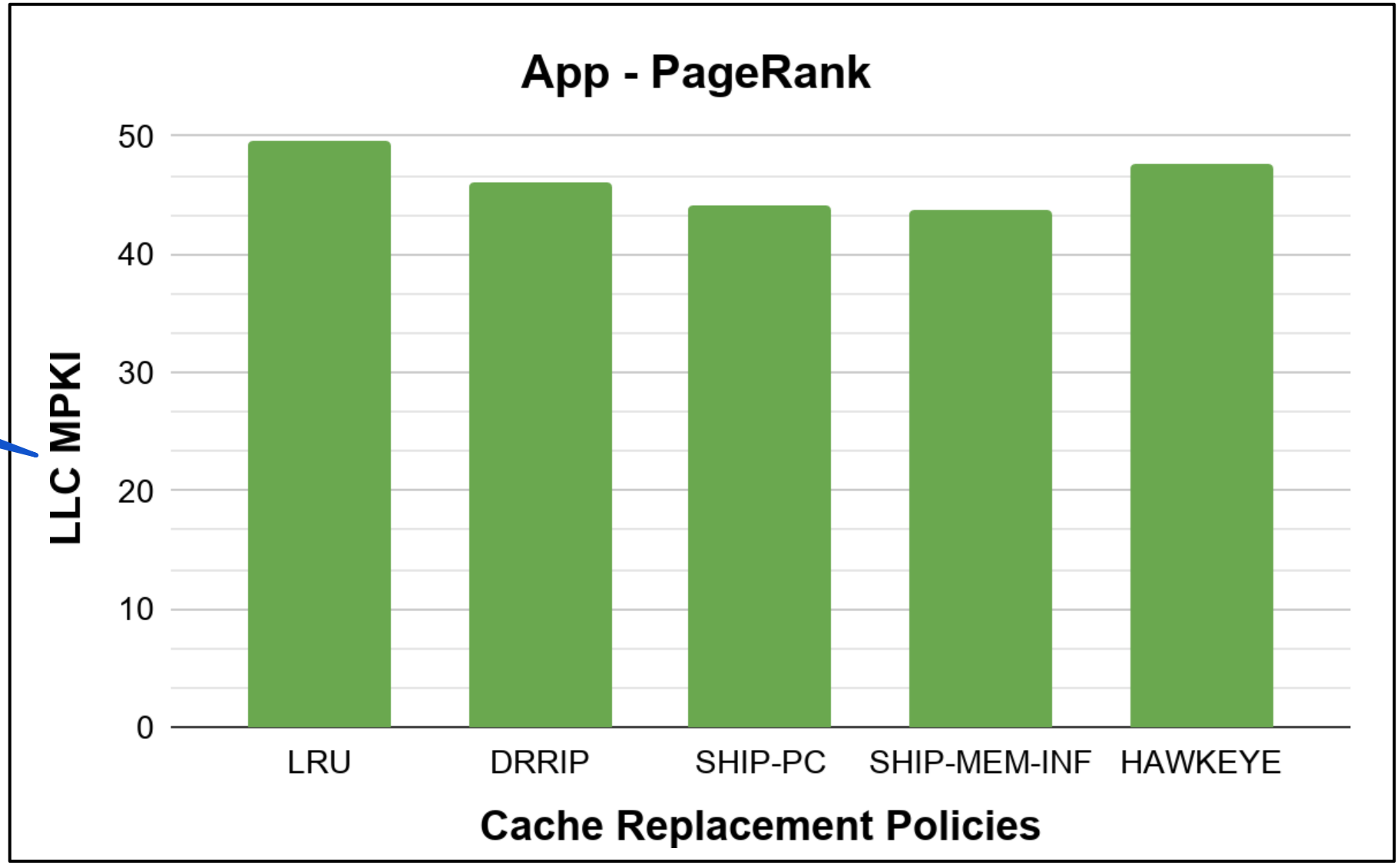
Scott Beamer*
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California
sbeamer@lbl.gov

Krste Asanović David Patterson
Electrical Engineering & Computer Sciences Department
University of California
Berkeley, California
{krste,pattrsn}@eecs.berkeley.edu

Application of Propagation Blocking for Graph Applications (Page Rank only, at first) discovered in 2017
(Prior work on “radix partitioning” applied the idea to other domains, but not graphs)

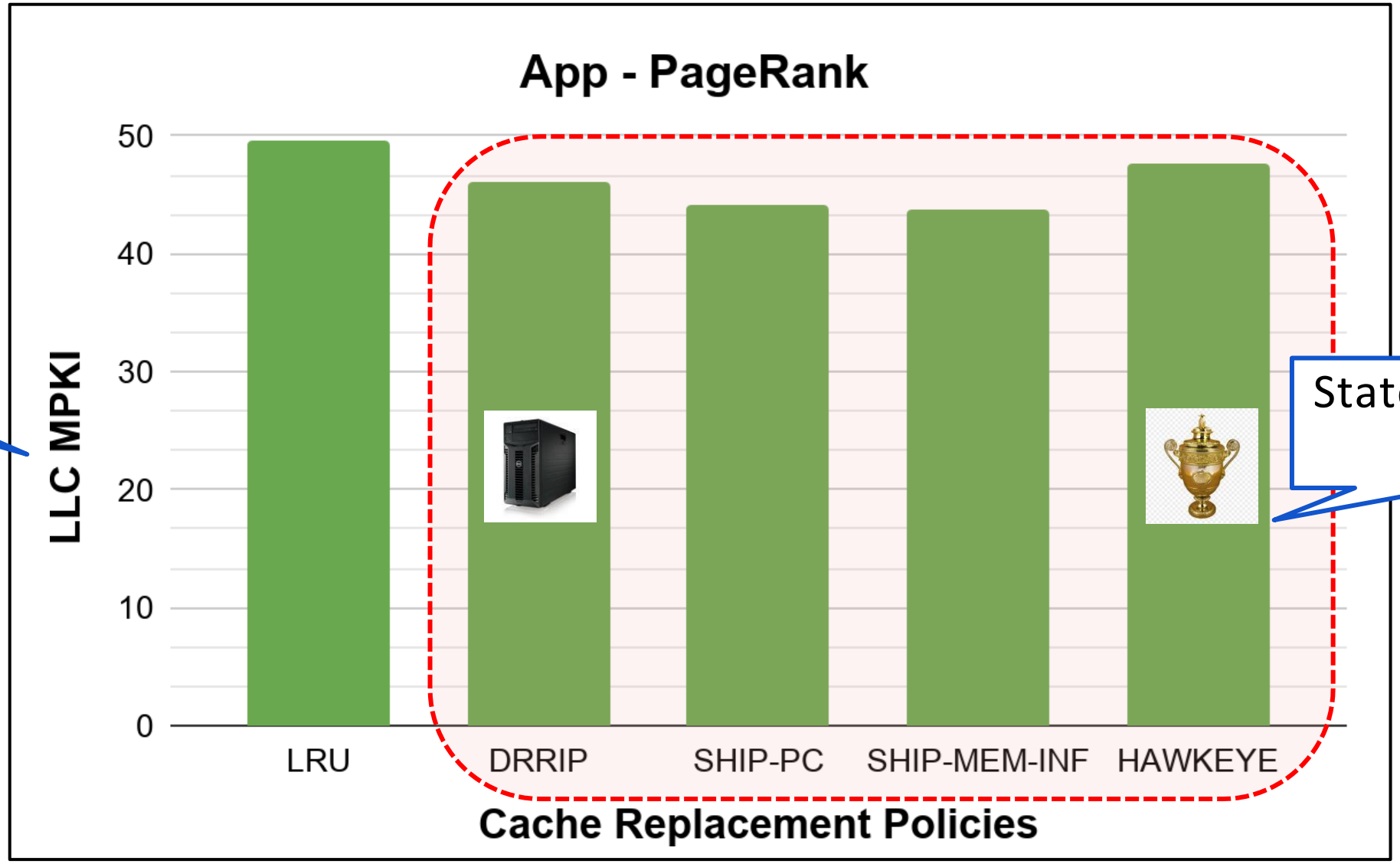
Cache Locality determines Overall Performance
What about better replacement policies?

Existing Replacement Policies Are Insufficient

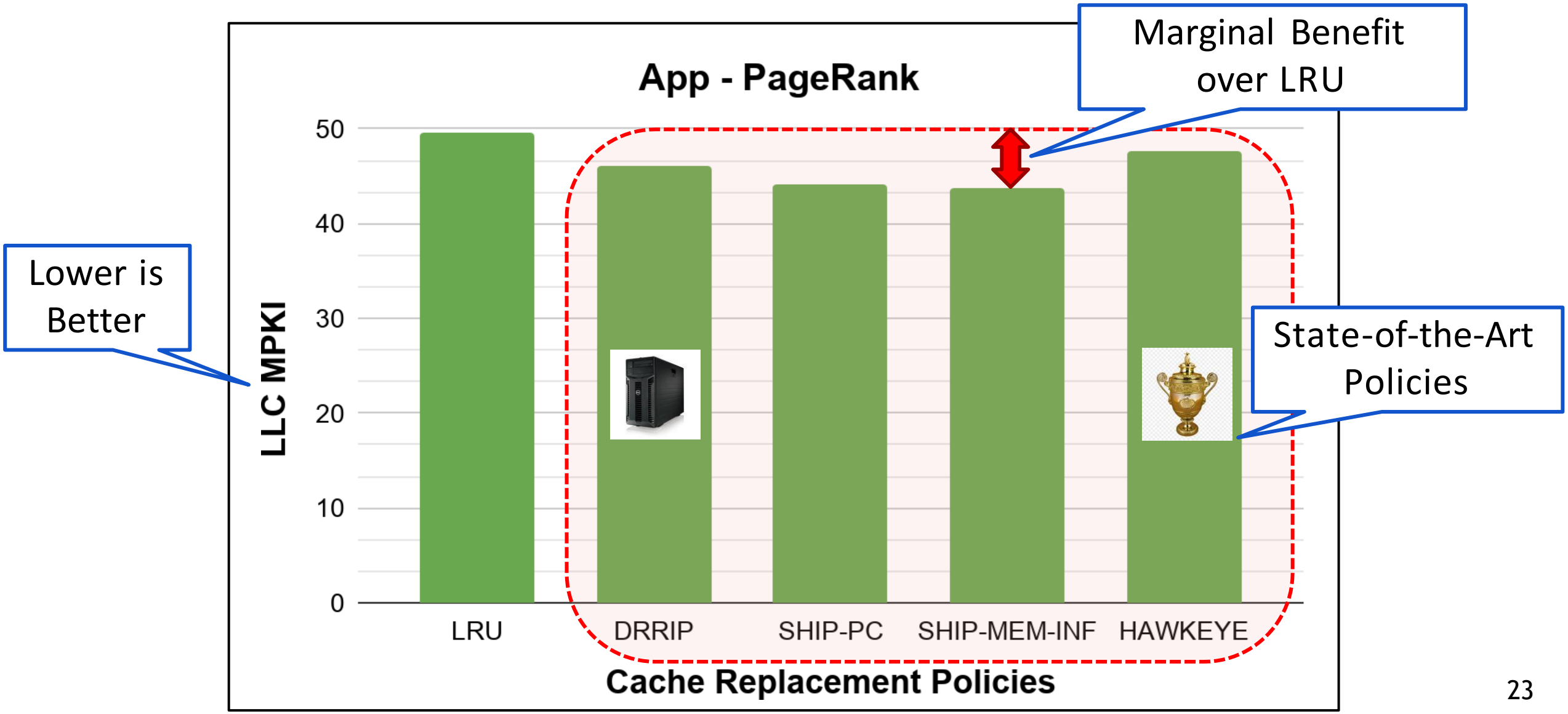


Lower is Better

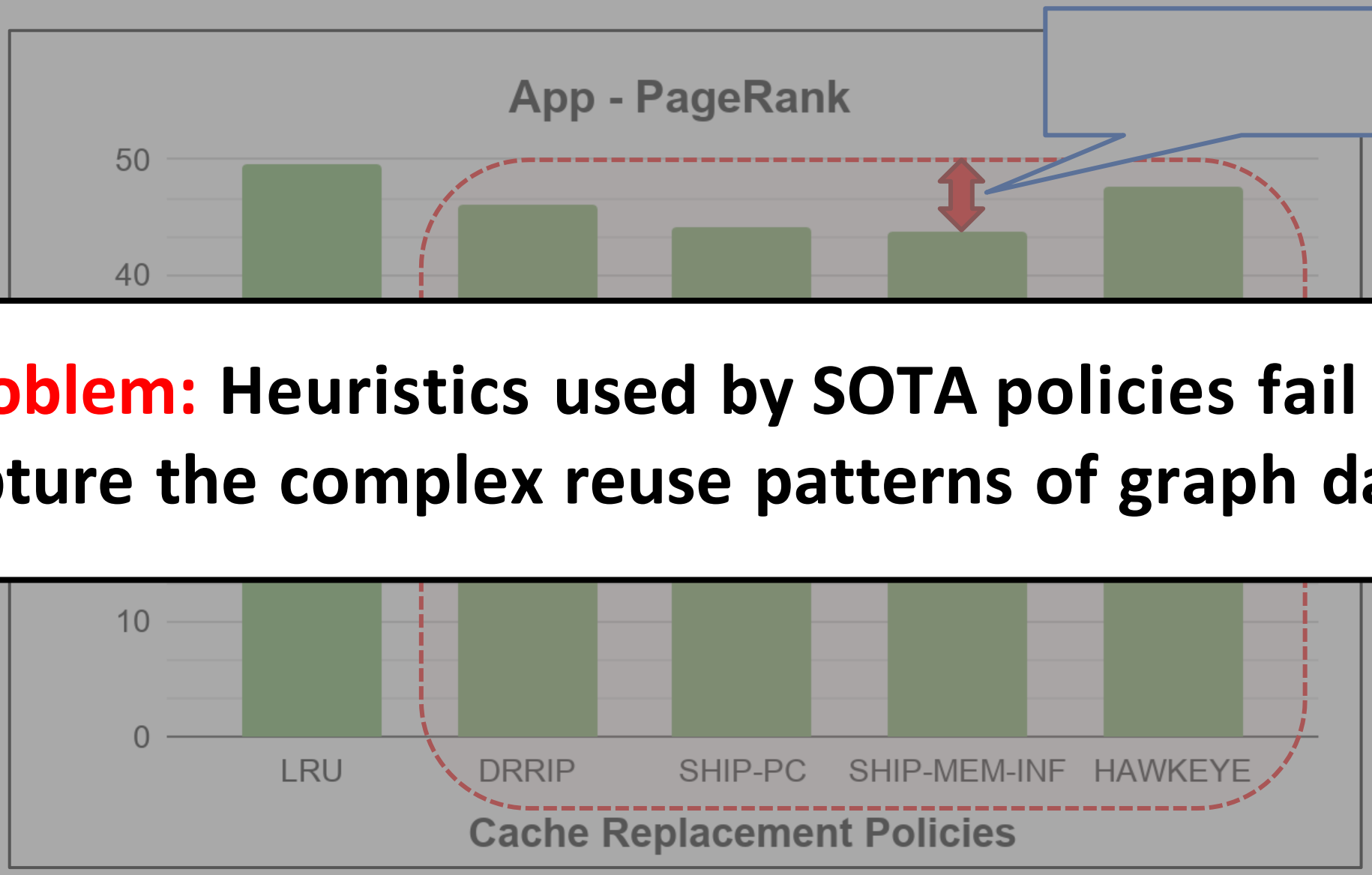
Existing Replacement Policies Are Insufficient



Existing Replacement Policies Are Insufficient

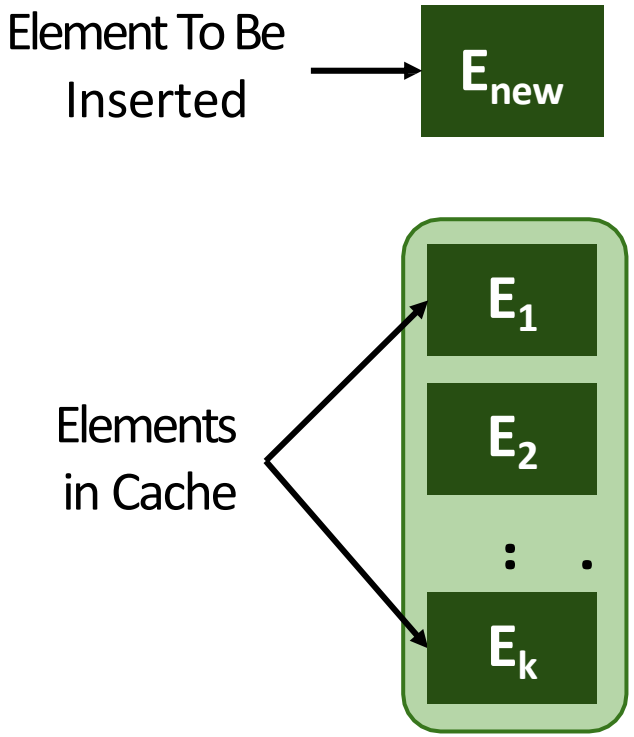


Existing Replacement Policies Are Insufficient



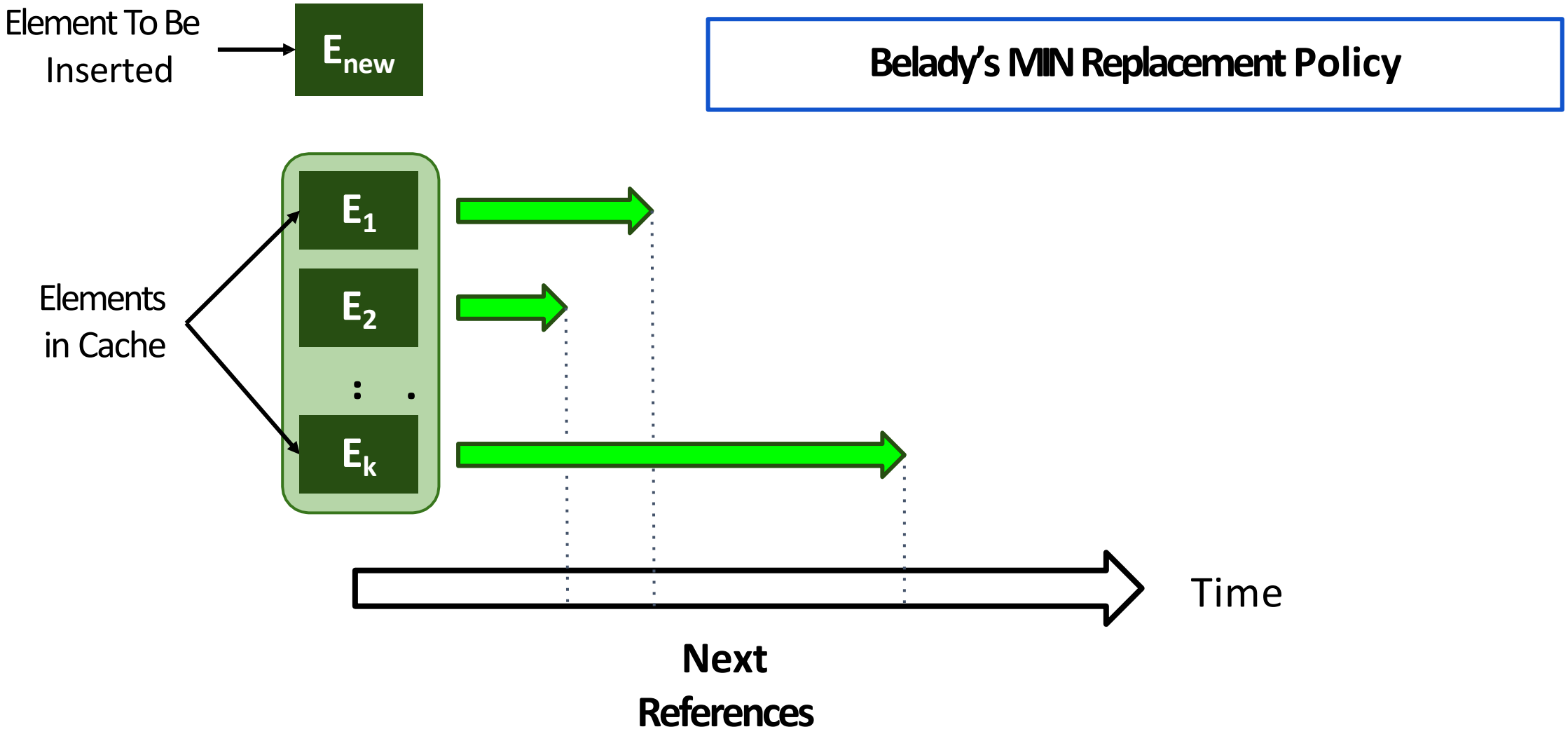
Problem: Heuristics used by SOTA policies fail to capture the complex reuse patterns of graph data

Is It Possible To Do Better Cache Replacement?

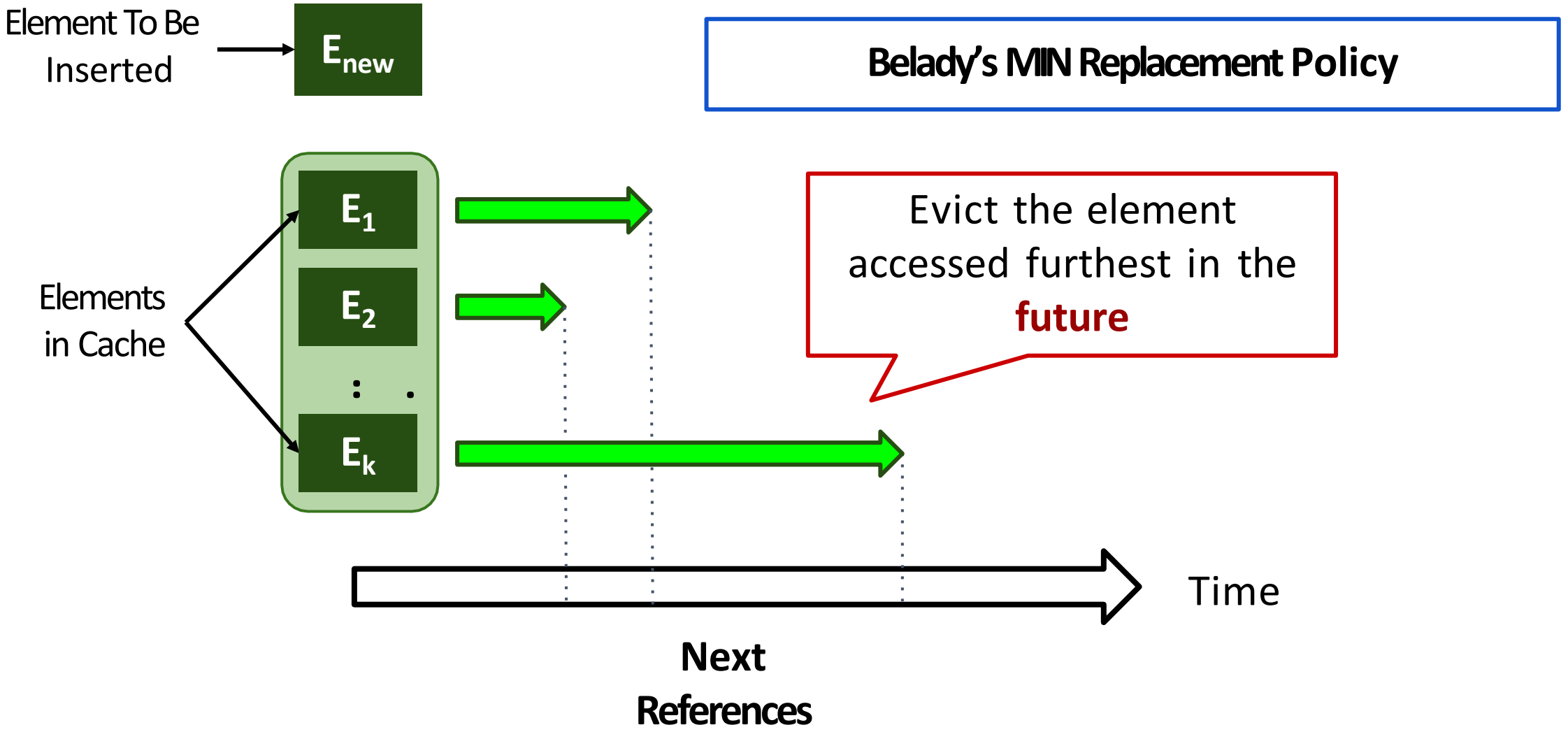


Belady's MIN Replacement Policy

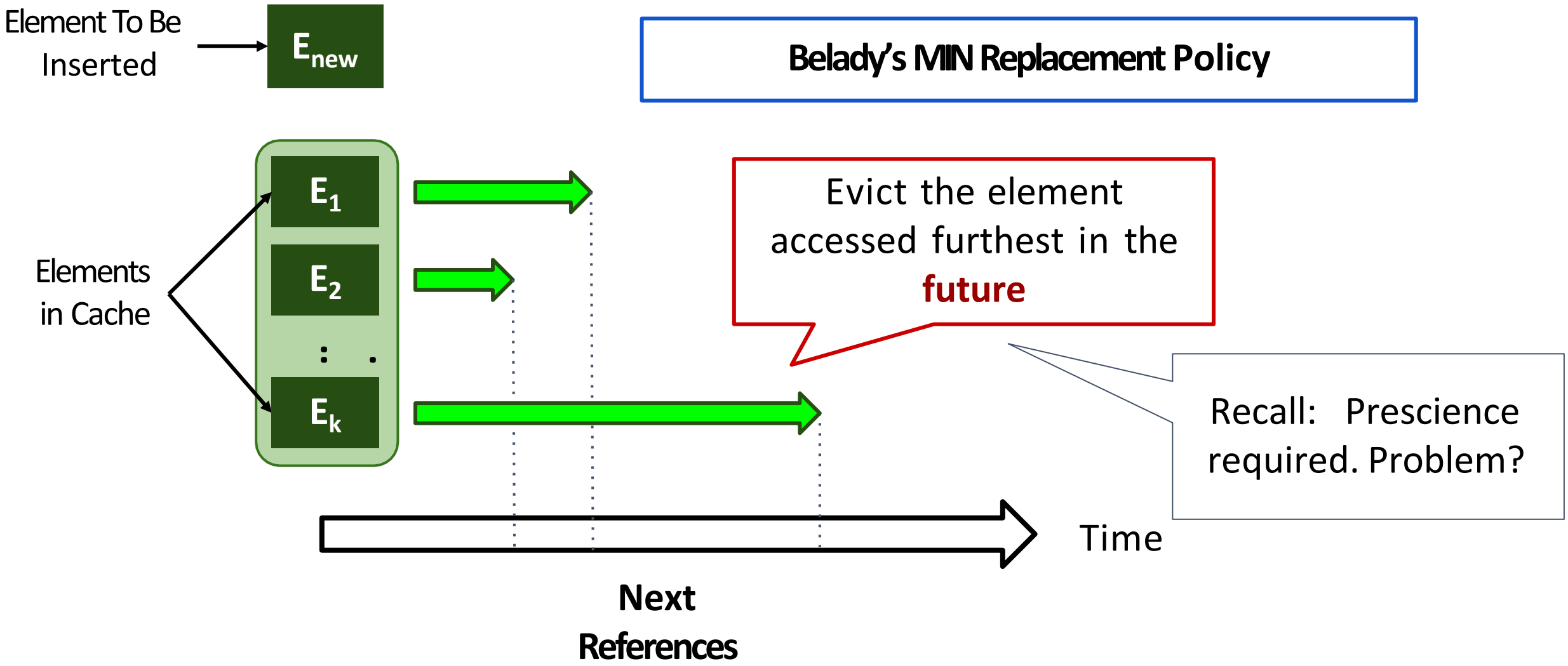
Is It Possible To Do Better Cache Replacement?



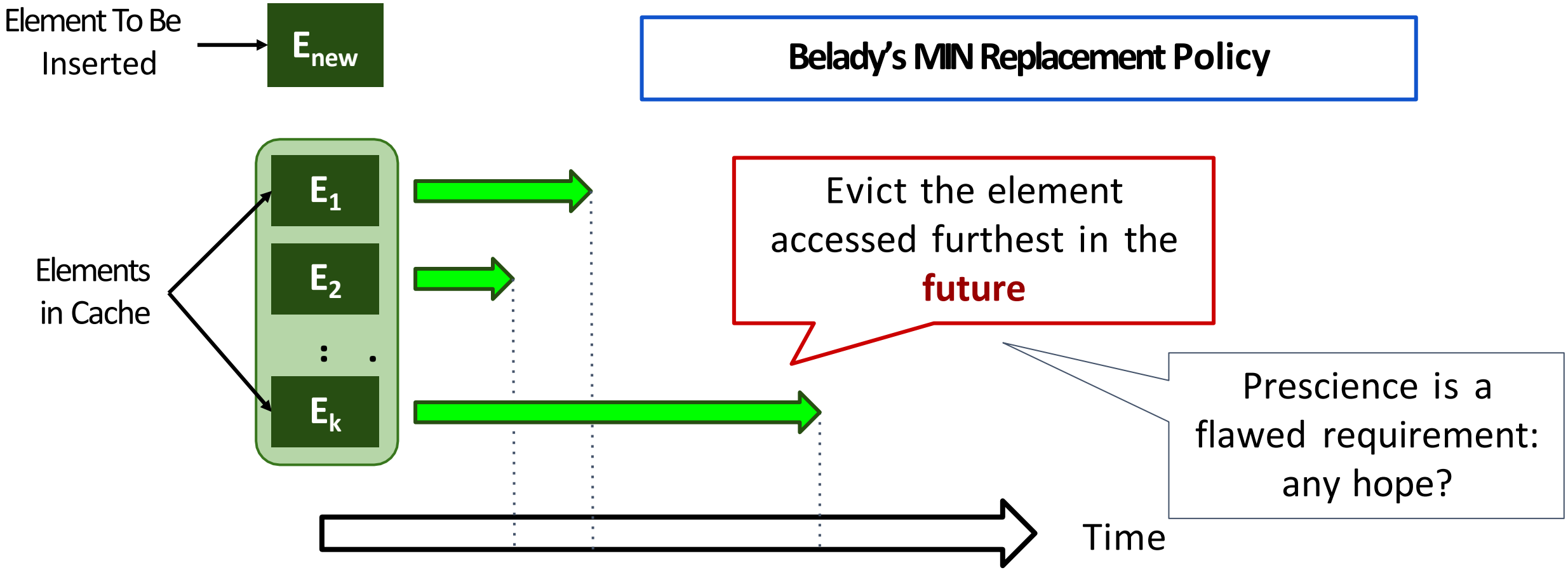
Is It Possible To Do Better Cache Replacement?



Is It Possible To Do Better Cache Replacement?



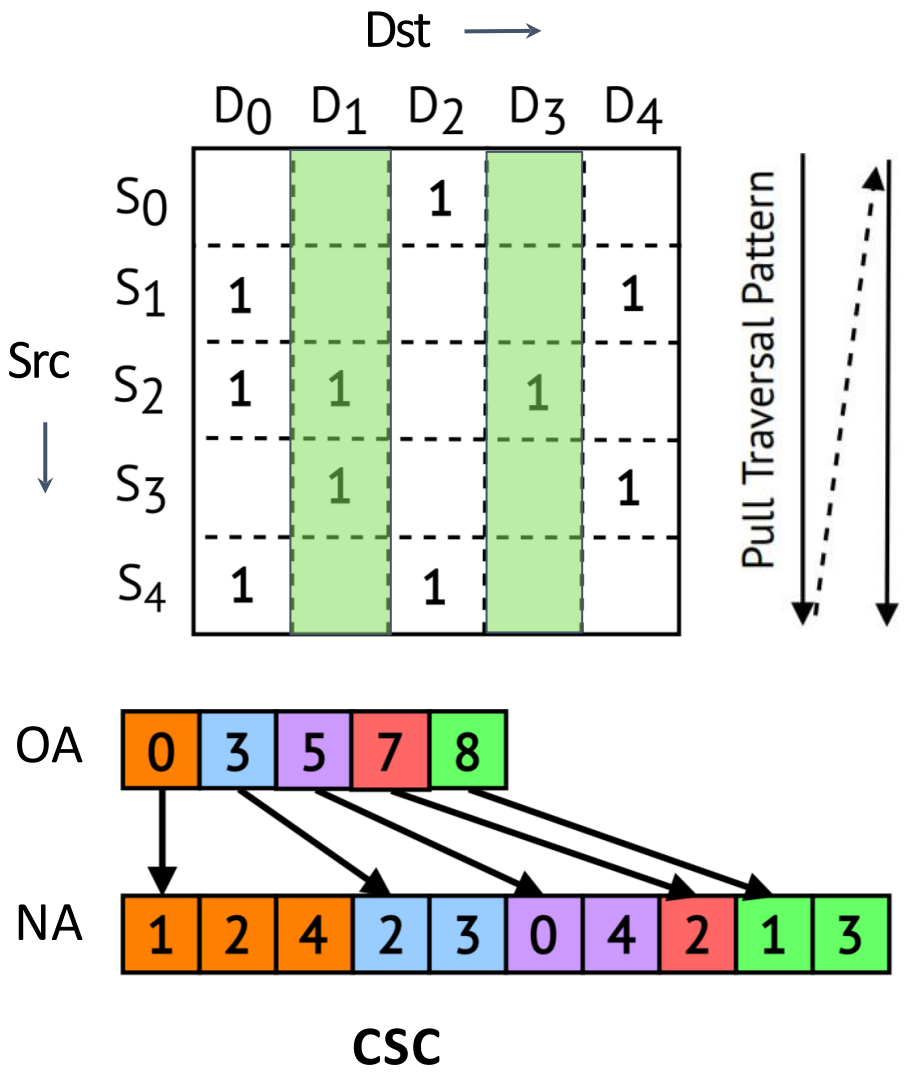
Is It Possible To Do Better Cache Replacement? YES!



Key Observation: The Graph's Transpose Efficiently Encodes Future Accesses

Key Graph Application Property That Enables Belady's OPT

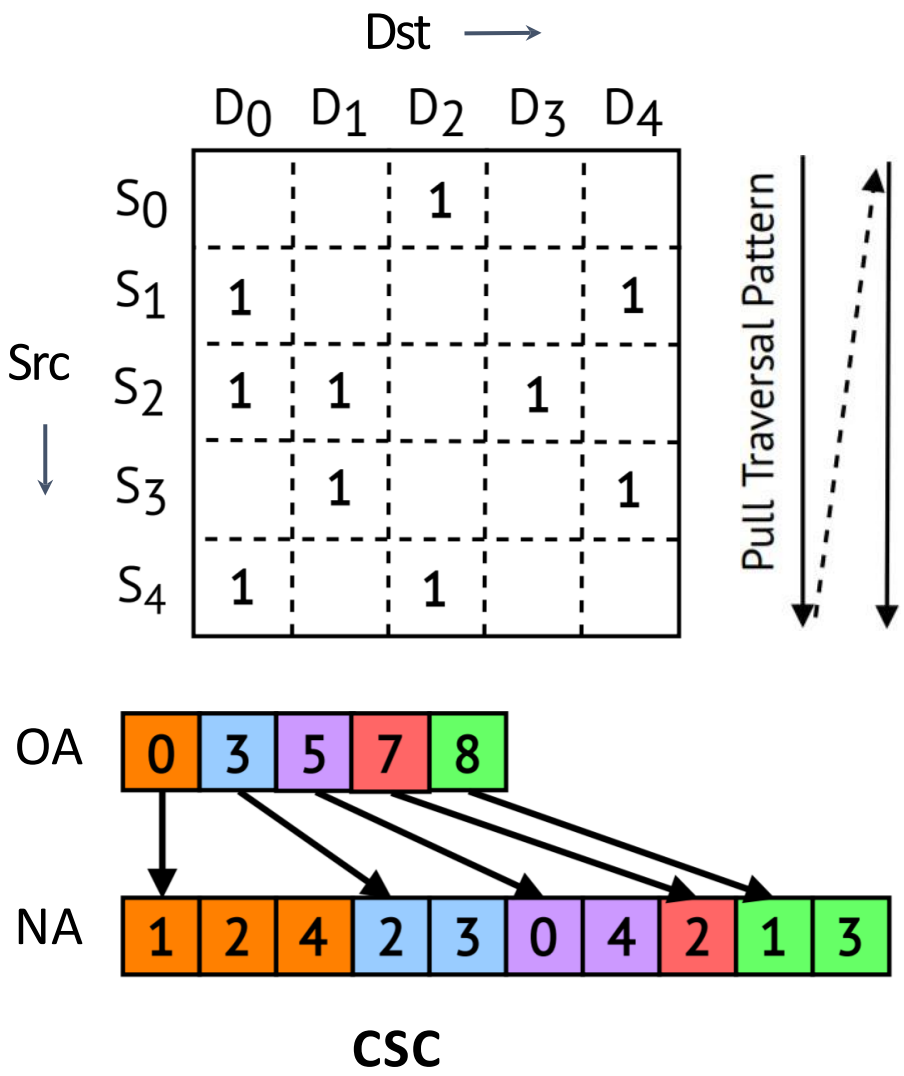
Key Graph Application Property That Enables Belady's OPT



Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

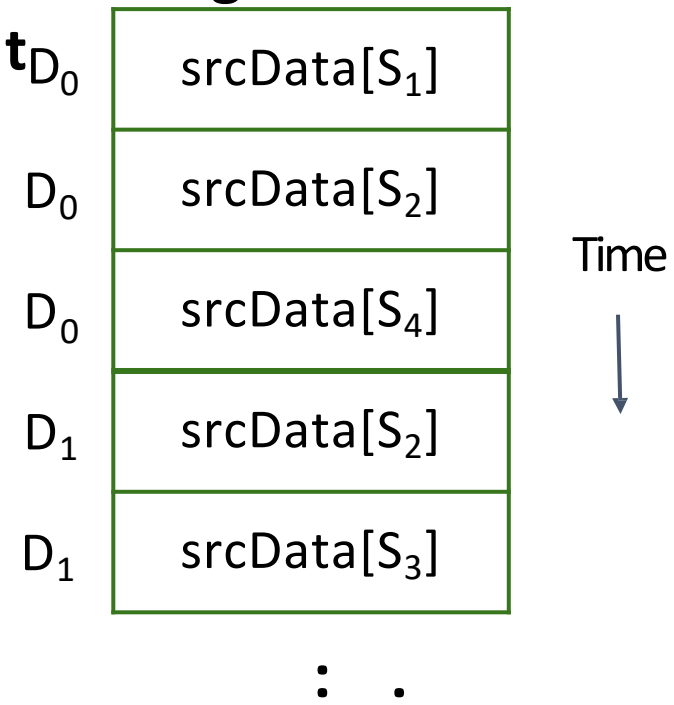
Key Graph Application Property That Enables Belady's OPT



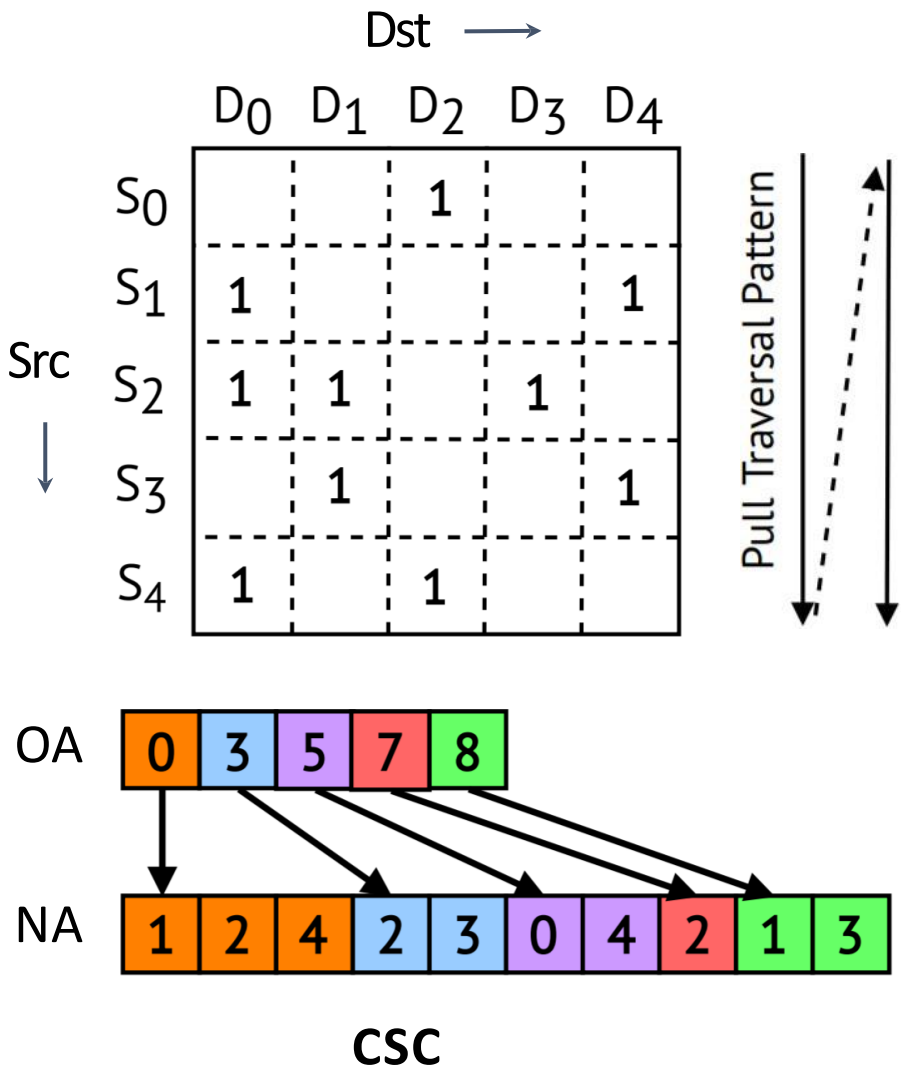
Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

CurrDs Irregular Data Stream



Key Graph Application Property That Enables Belady's OPT

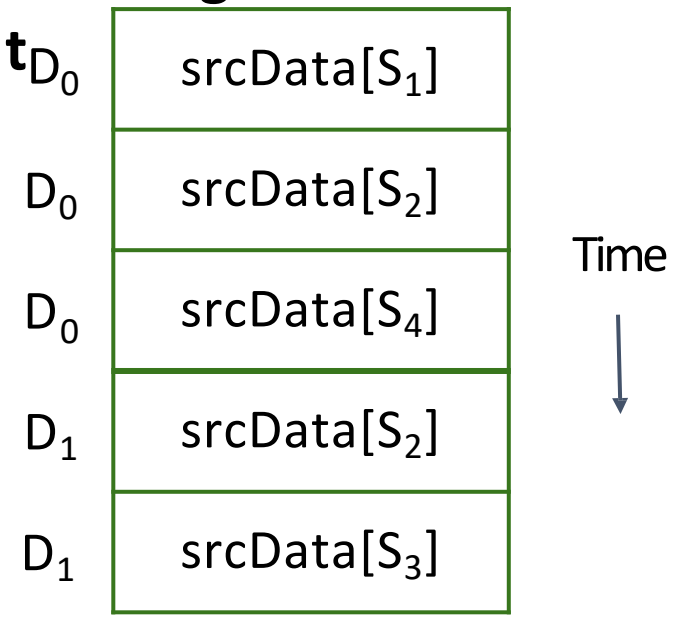


Pull Execution (*CSC Traversal*)

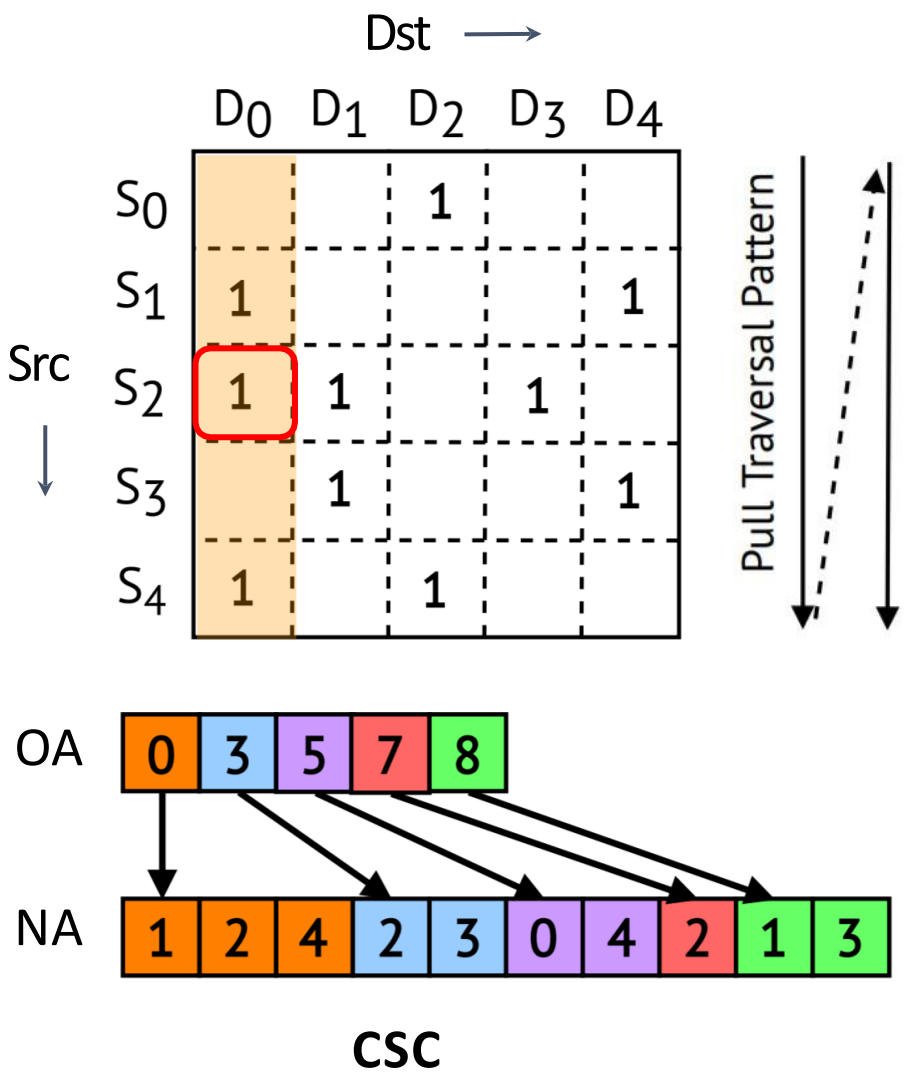
```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

Key Property: Dst-IDs are like timestamps for irregular accesses

CurrDs Irregular Data Stream



Key Graph Application Property That Enables Belady's OPT

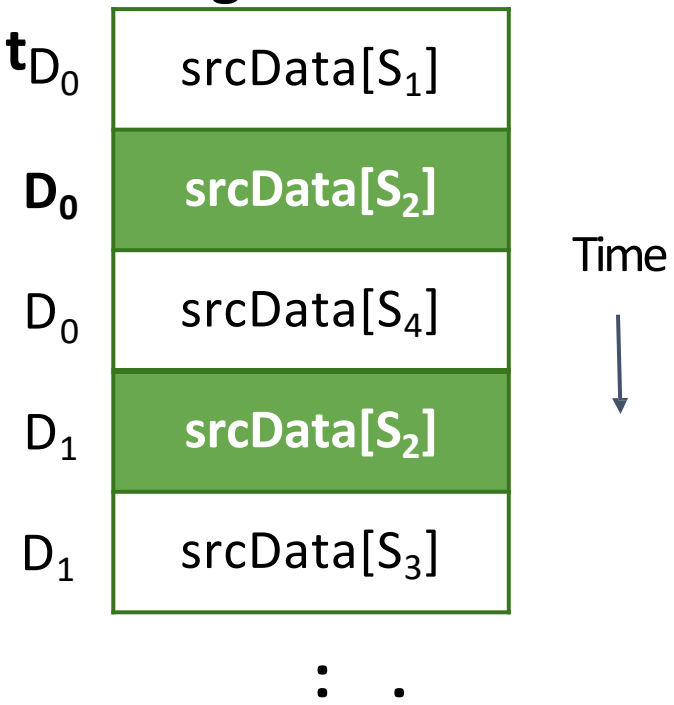


Pull Execution (*CSC Traversal*)

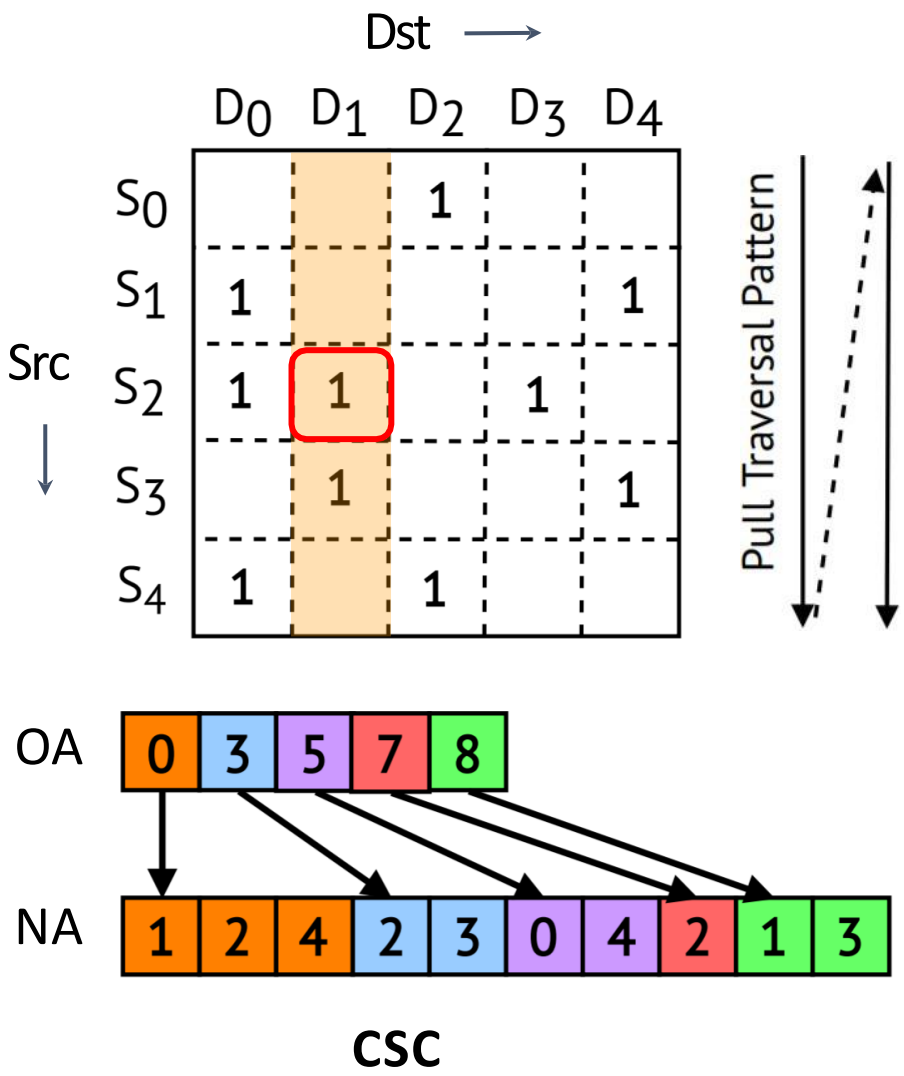
```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

Key Property: Dst-IDs are like timestamps for irregular accesses

CurrDs Irregular Data Stream



Key Graph Application Property That Enables Belady's OPT

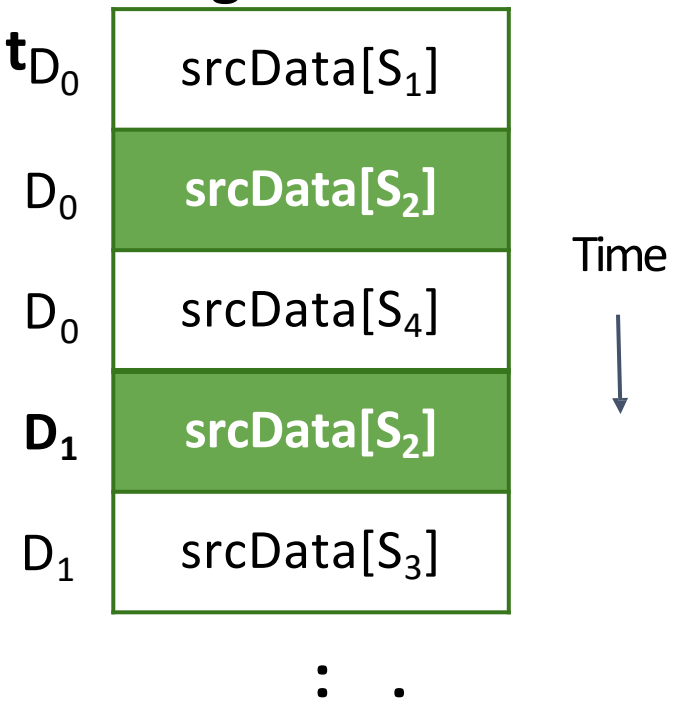


Pull Execution (*CSC Traversal*)

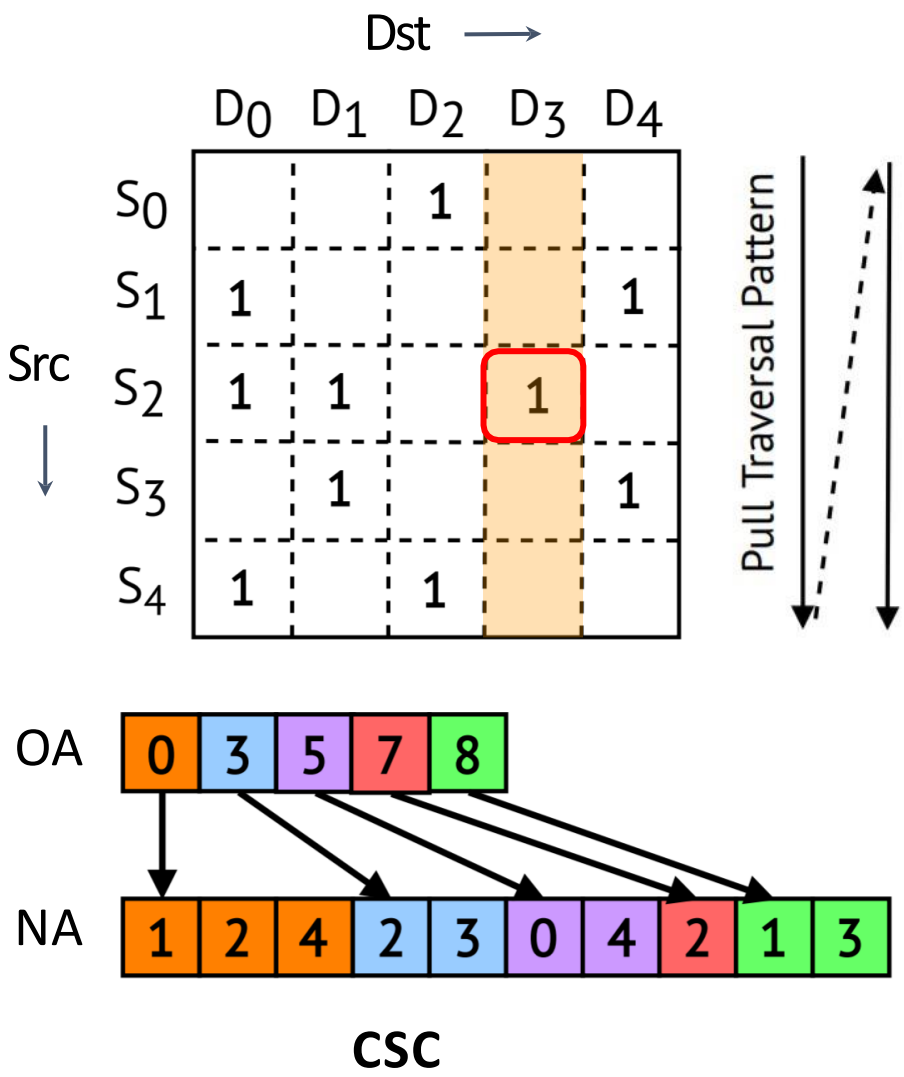
```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

Key Property: Dst-IDs are like timestamps for irregular accesses

CurrDs Irregular Data Stream



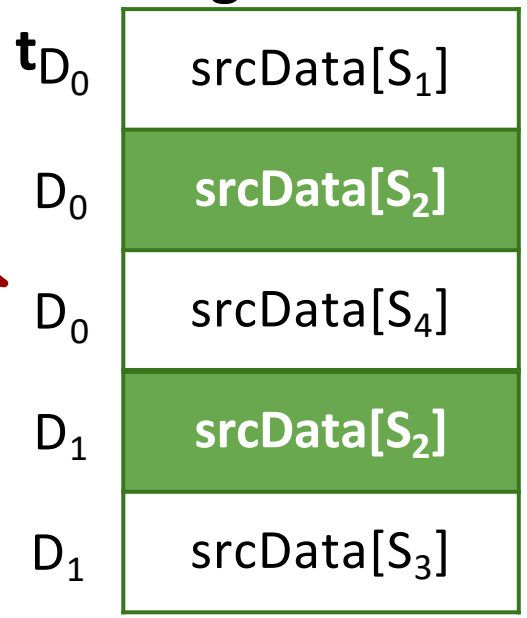
Key Graph Application Property That Enables Belady's OPT



Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

CurrDs Irregular Data Stream

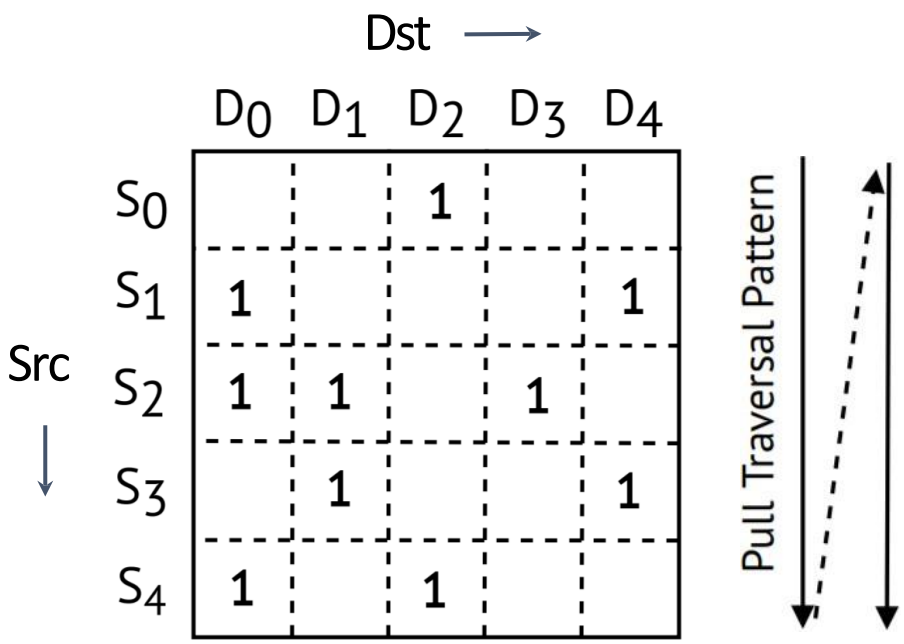


Key Property: Dst-IDs are like timestamps for irregular accesses

srcData[S_2] is accessed at $D_0 \Rightarrow D_1 \Rightarrow D_3$

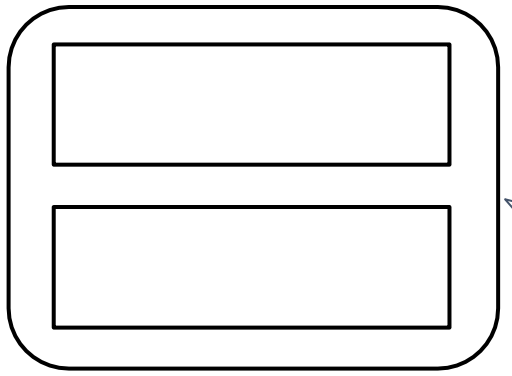
Using The Graph's Transpose For Optimal Replacement

Using The Graph's Transpose For Optimal Replacement



Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```



2-way Set-Associative

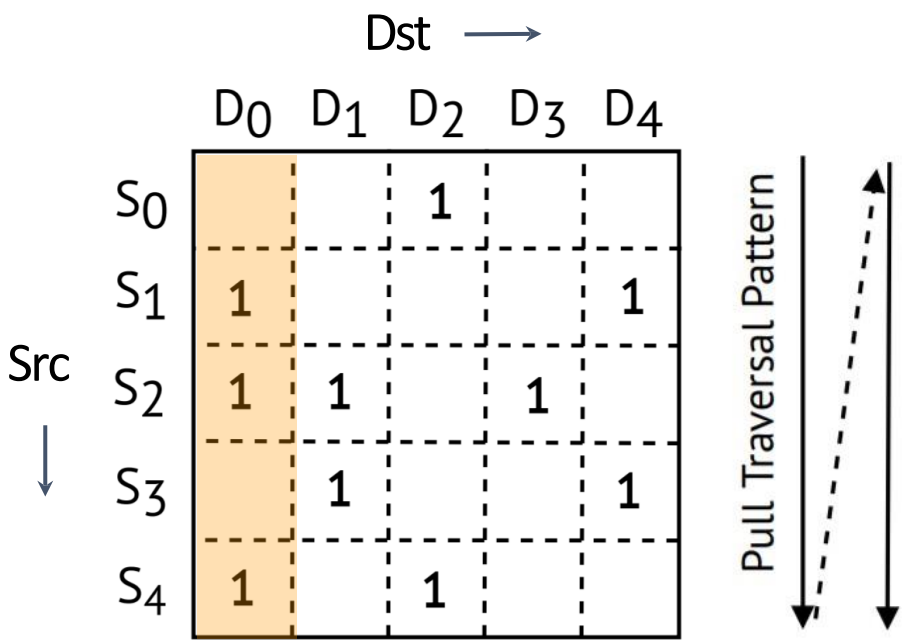
Assumptions:

1. One srcData elem per line
2. Only irregular data enters the cache

CurrDs	Irregular Data Stream
t_{D_0}	srcData[S_1]
D_0	srcData[S_2]
D_0	srcData[S_4]
D_1	srcData[S_2]
D_1	srcData[S_3]

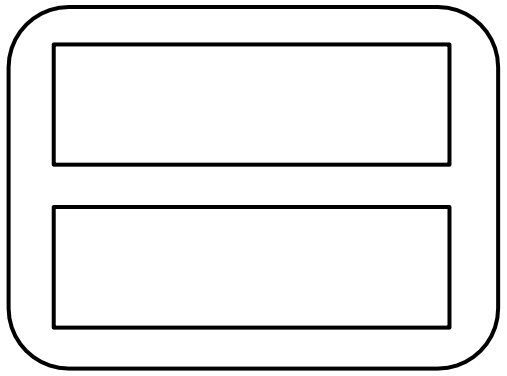
Time

Using The Graph's Transpose For Optimal Replacement



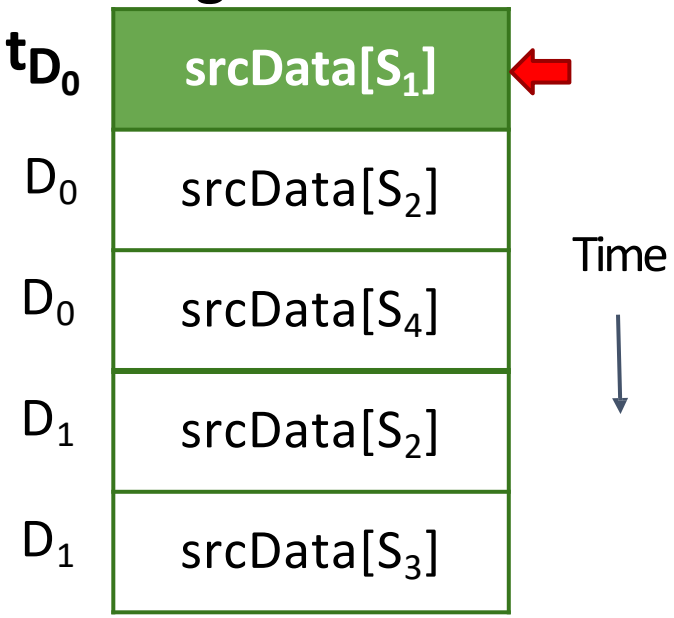
Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

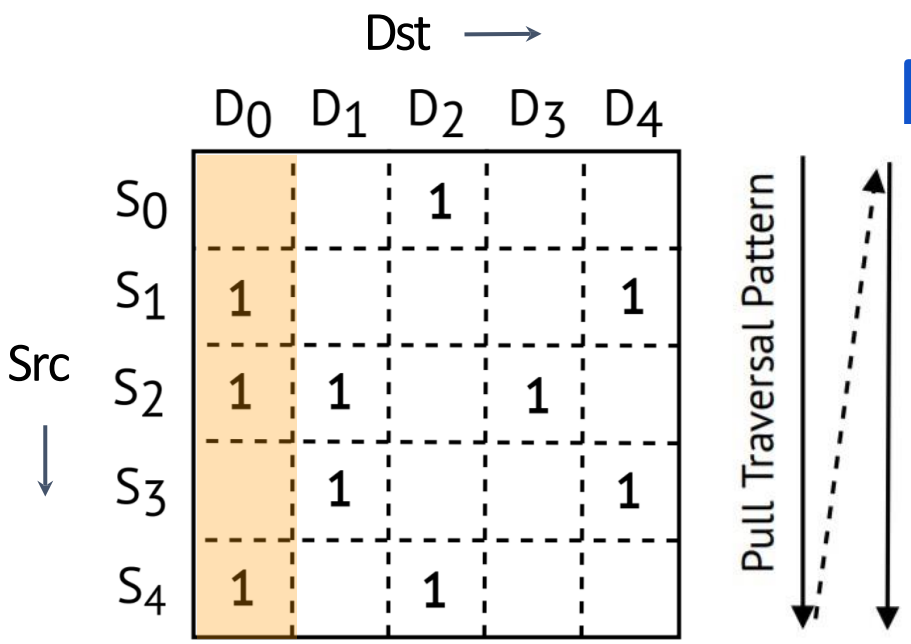


2-way Set-Associative

CurrDs **Irregular Data Stream**

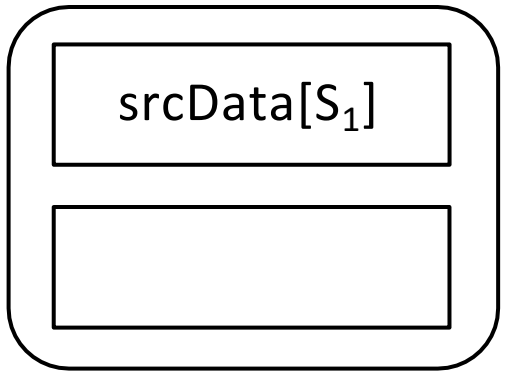


Using The Graph's Transpose For Optimal Replacement



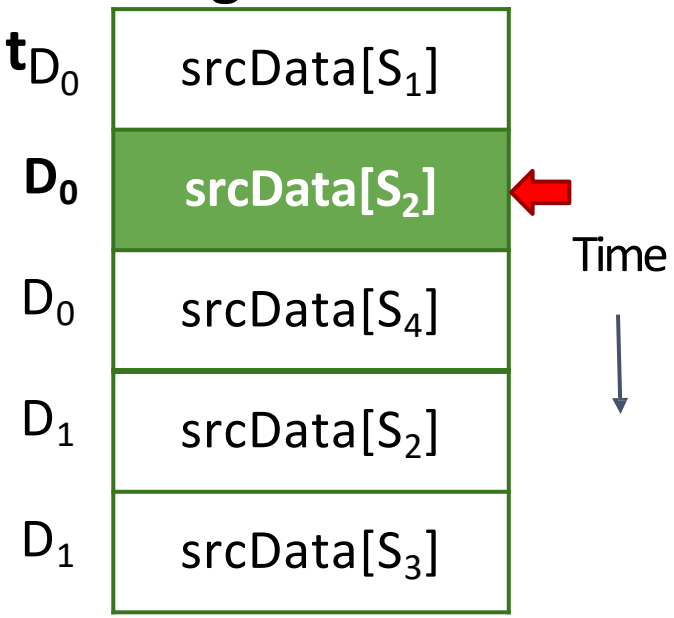
Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

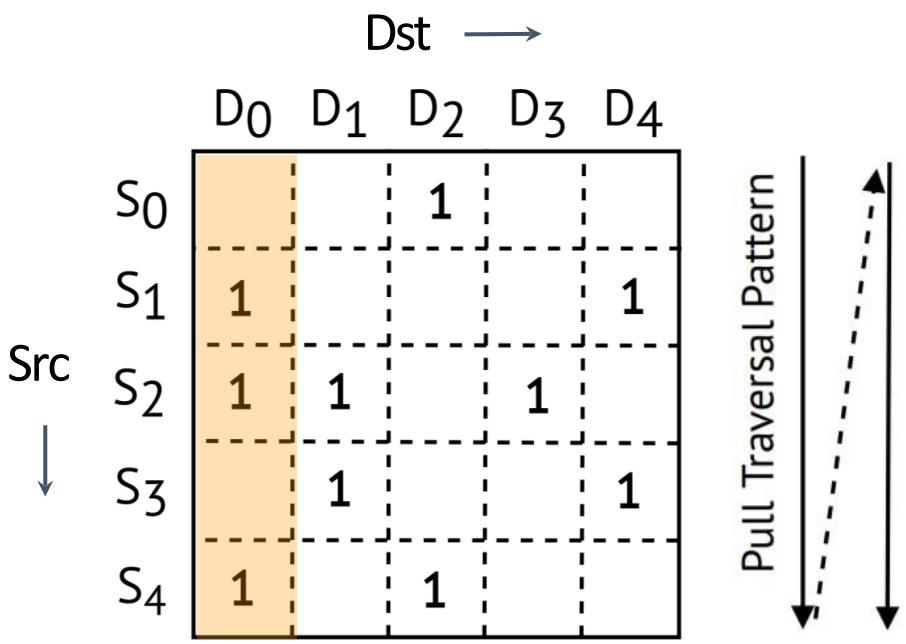


2-way Set-Associative

CurrDs Irregular Data Stream



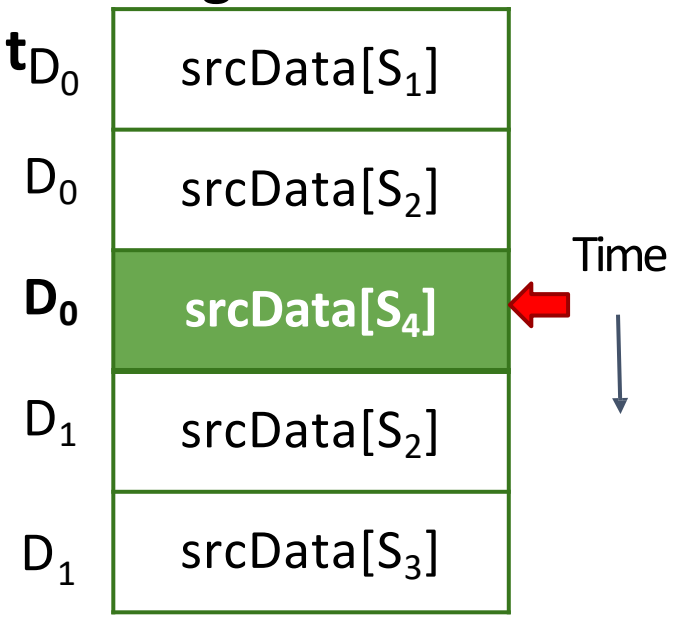
Using The Graph's Transpose For Optimal Replacement



Pull Execution (*CSC Traversal*)

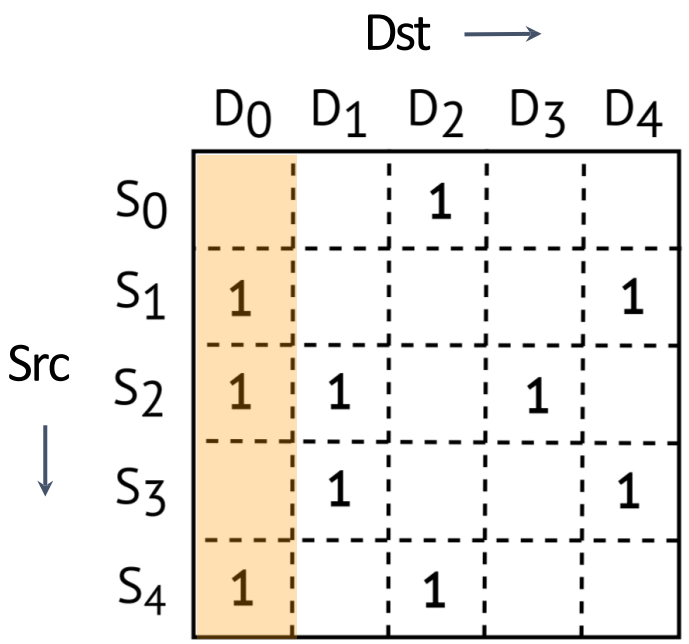
```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

CurrDs Irregular Data Stream



2-way Set-Associative

Using The Graph's Transpose For Optimal Replacement



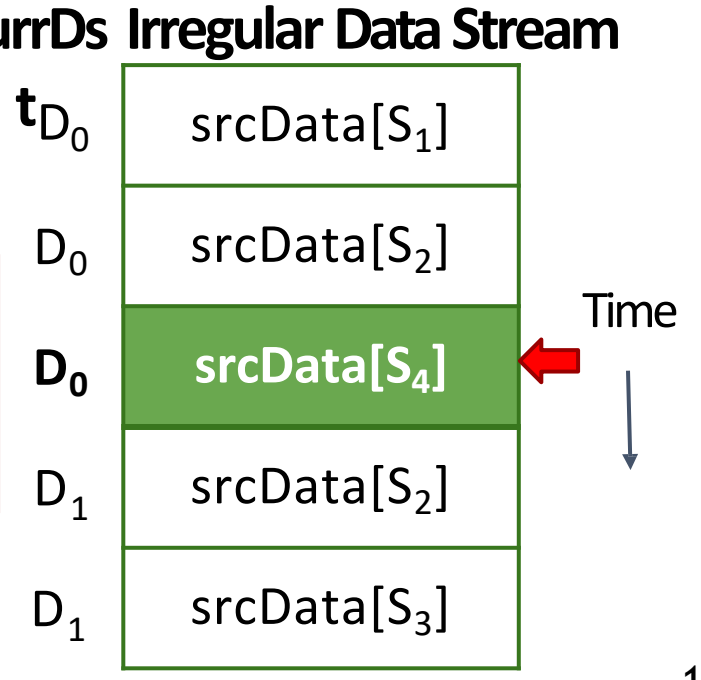
Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

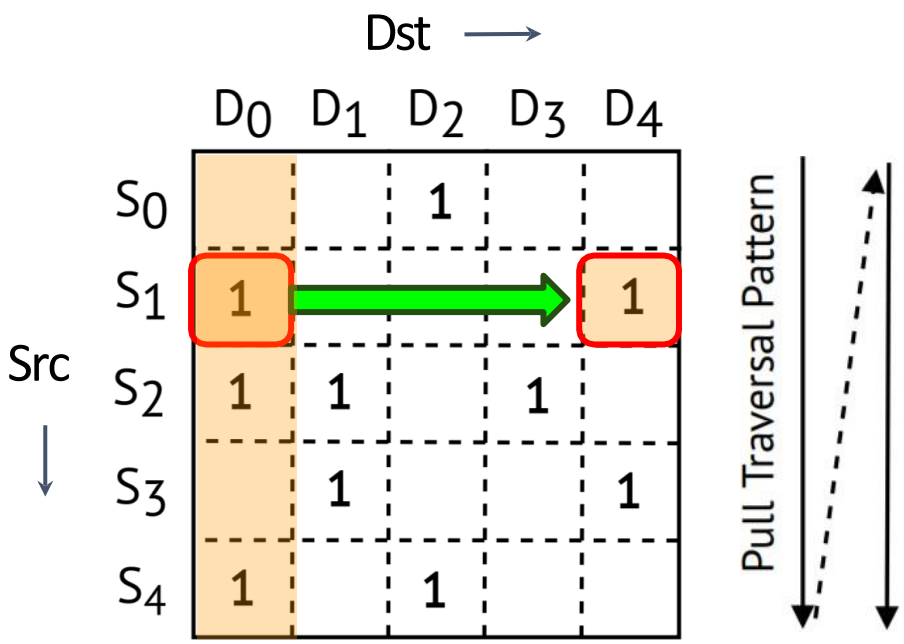
Which line should we evict?:

- srcData[S_1]
- srcData[S_2]

2-way Set-Associative



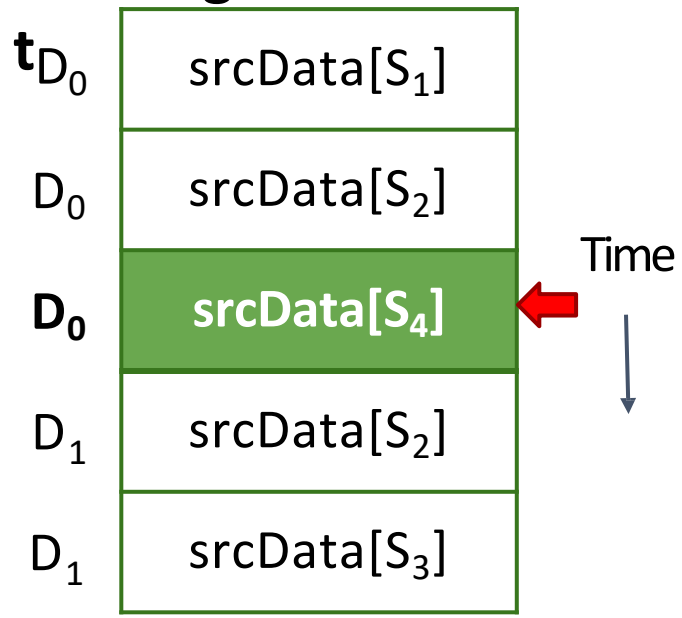
Using The Graph's Transpose For Optimal Replacement



Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

CurrDs **Irregular Data Stream**



Which line should we evict?:

- srcData[S_1] **(nextRef @ D_4)**
- srcData[S_2]

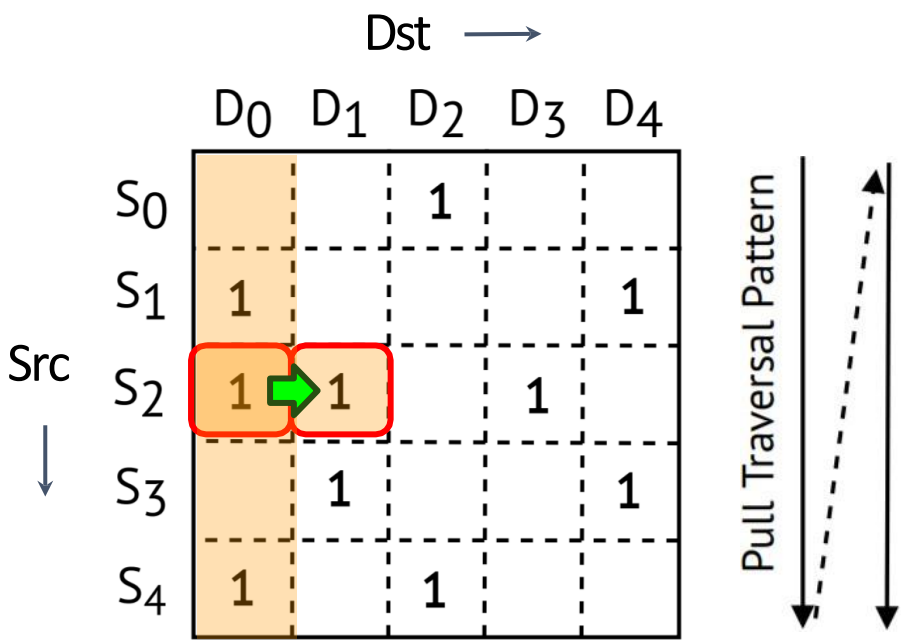
2-way Set-Associative

Out-Neigh(S_1): **D_0** D_4

CS R

: .

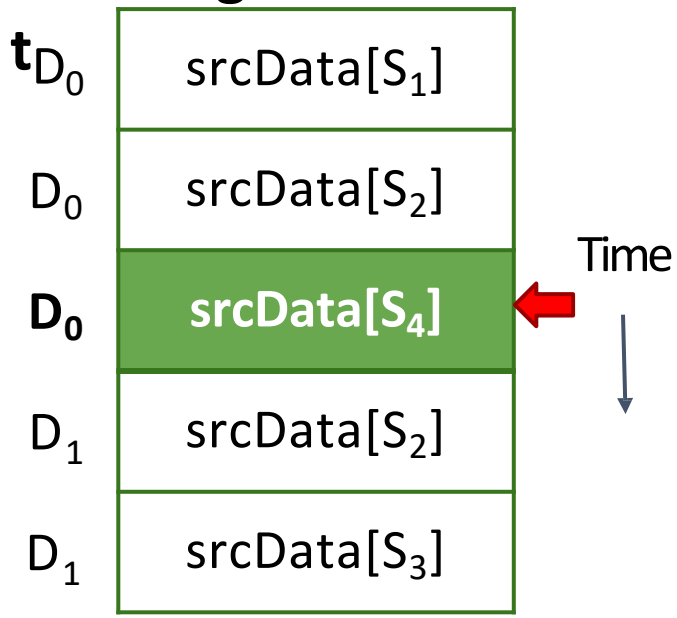
Using The Graph's Transpose For Optimal Replacement



Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

CurrDs Irregular Data Stream

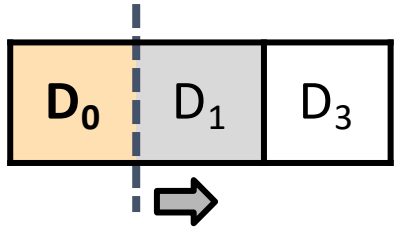


Which line should we evict?:

- srcData[S_1] **(nextRef @ D_4)**
- srcData[S_2] **(nextRef @ D_1)**

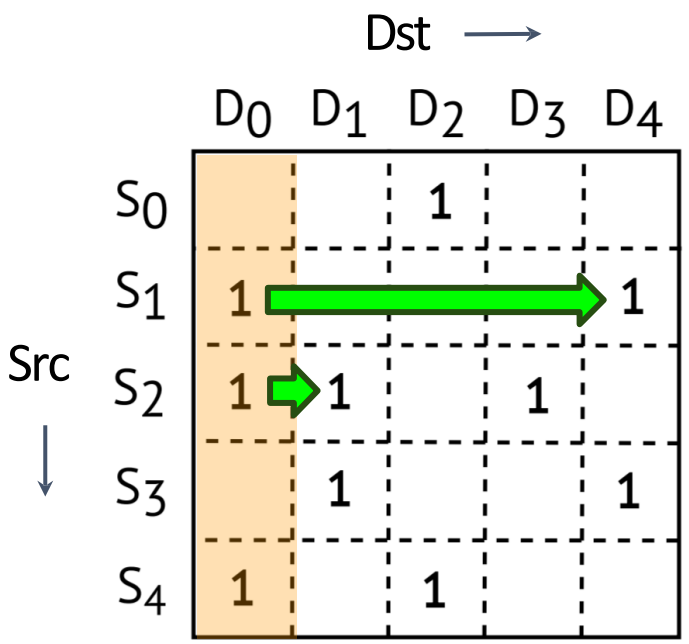
2-way Set-Associative

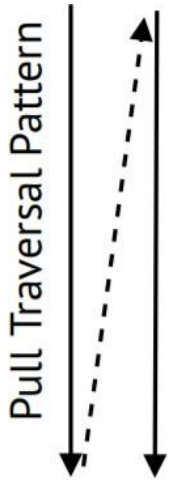
Out-Neigh(S_2):



CS R : .

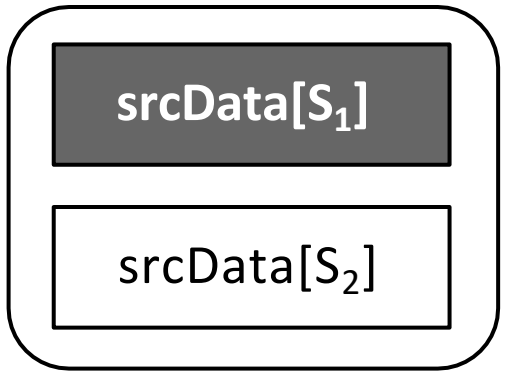
Using The Graph's Transpose For Optimal Replacement





Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

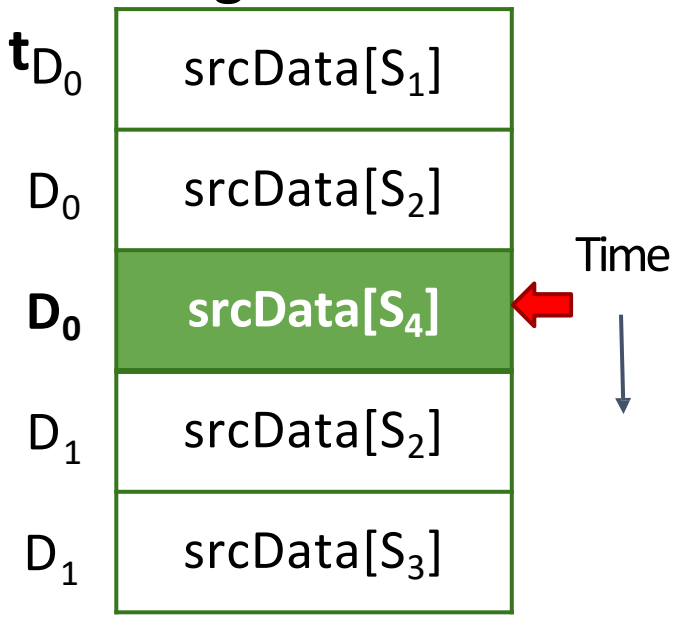


2-way Set-Associative

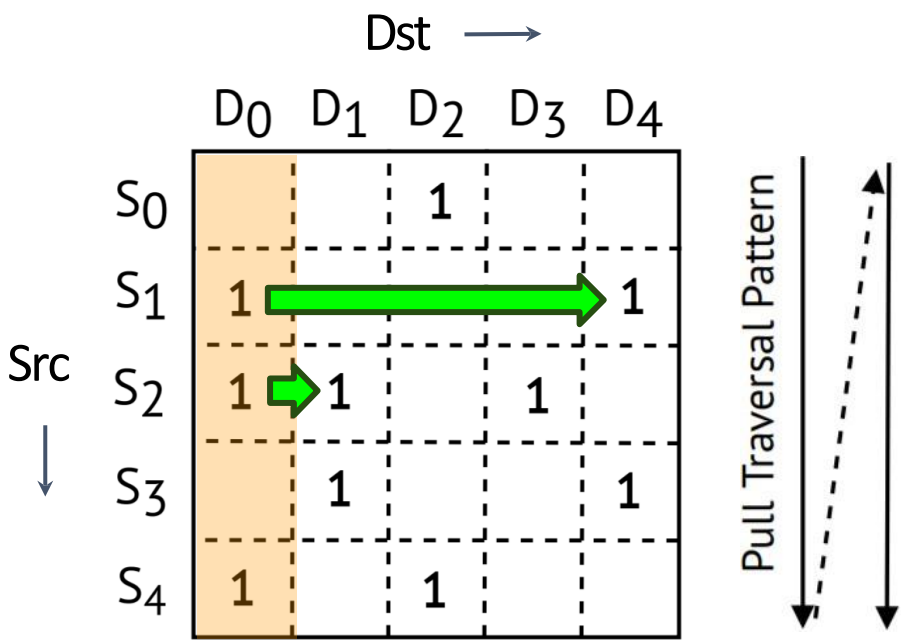
Which line should we evict?:

- srcData[S_1] **(nextRef @ D_4)** ✓
- srcData[S_2] **(nextRef @ D_1)**

CurrDs Irregular Data Stream

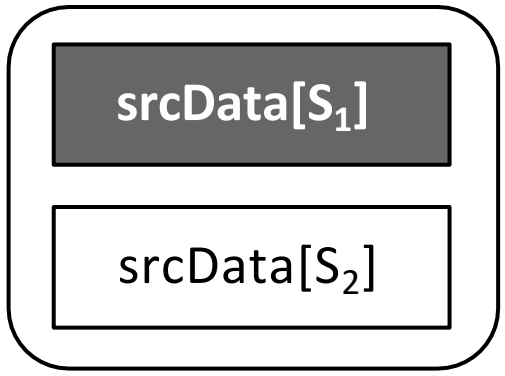


Using The Graph's Transpose For Optimal Replacement



Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```



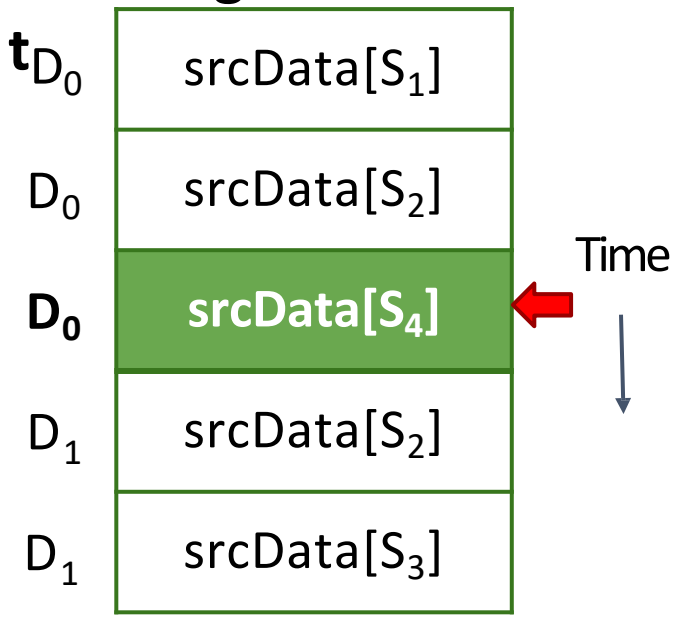
2-way Set-Associative

Which line should we evict?:

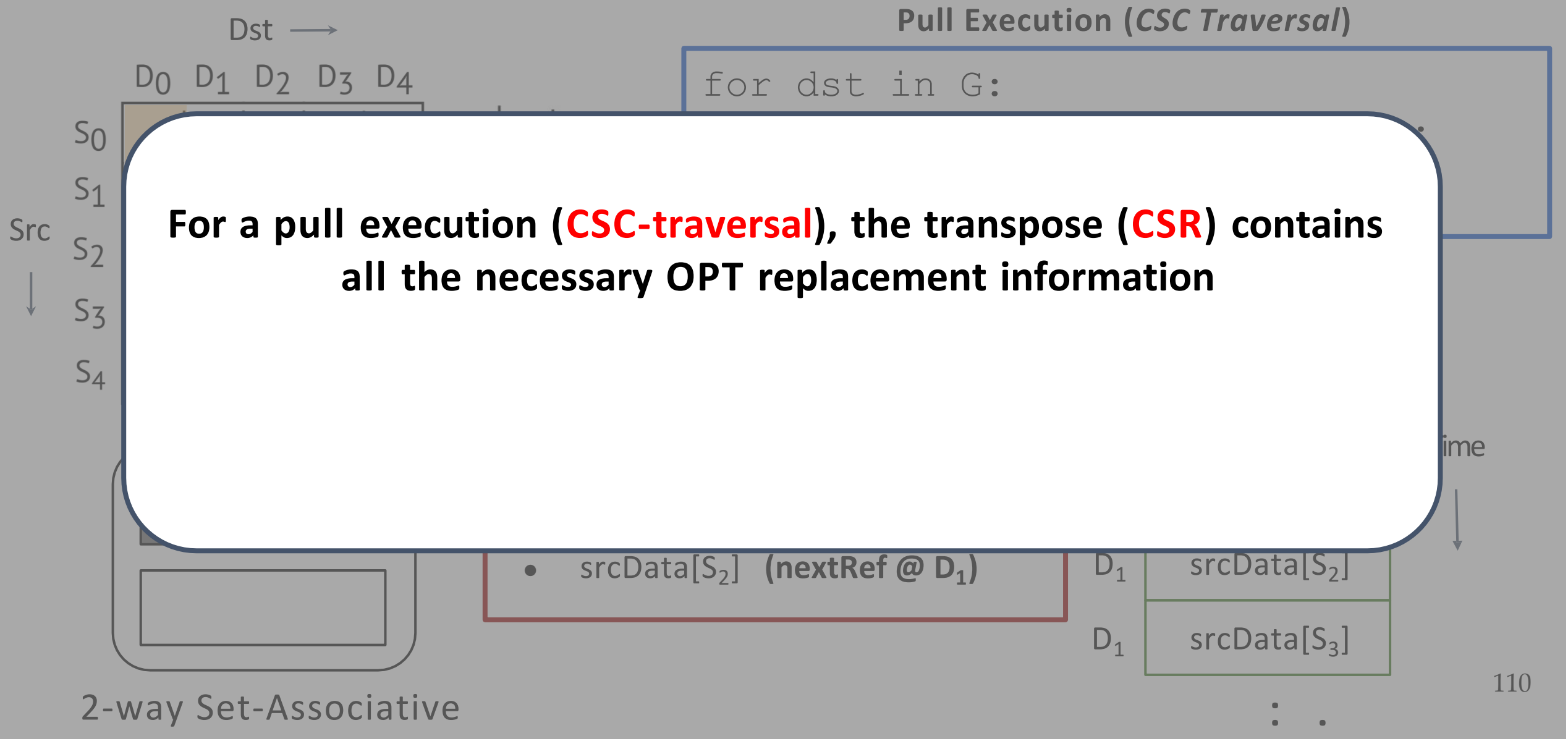
- srcData[S_1] **(nextRef @ D_4)** ✓
- srcData[S_2] **(nextRef @ D_1)**

Key Question: how to query next reference while running the program?

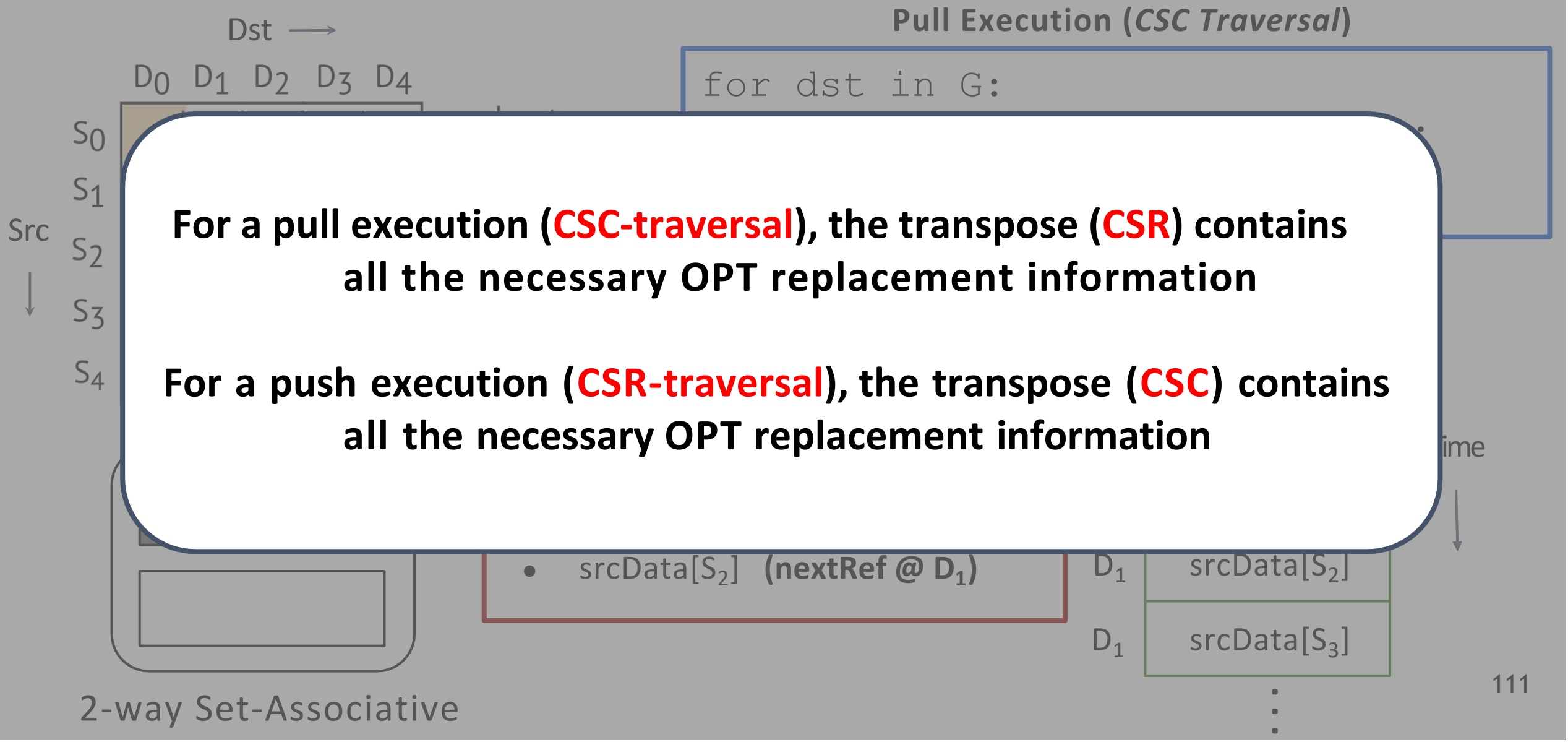
CurrDs Irregular Data Stream



Using The Graph's Transpose For Optimal Replacement

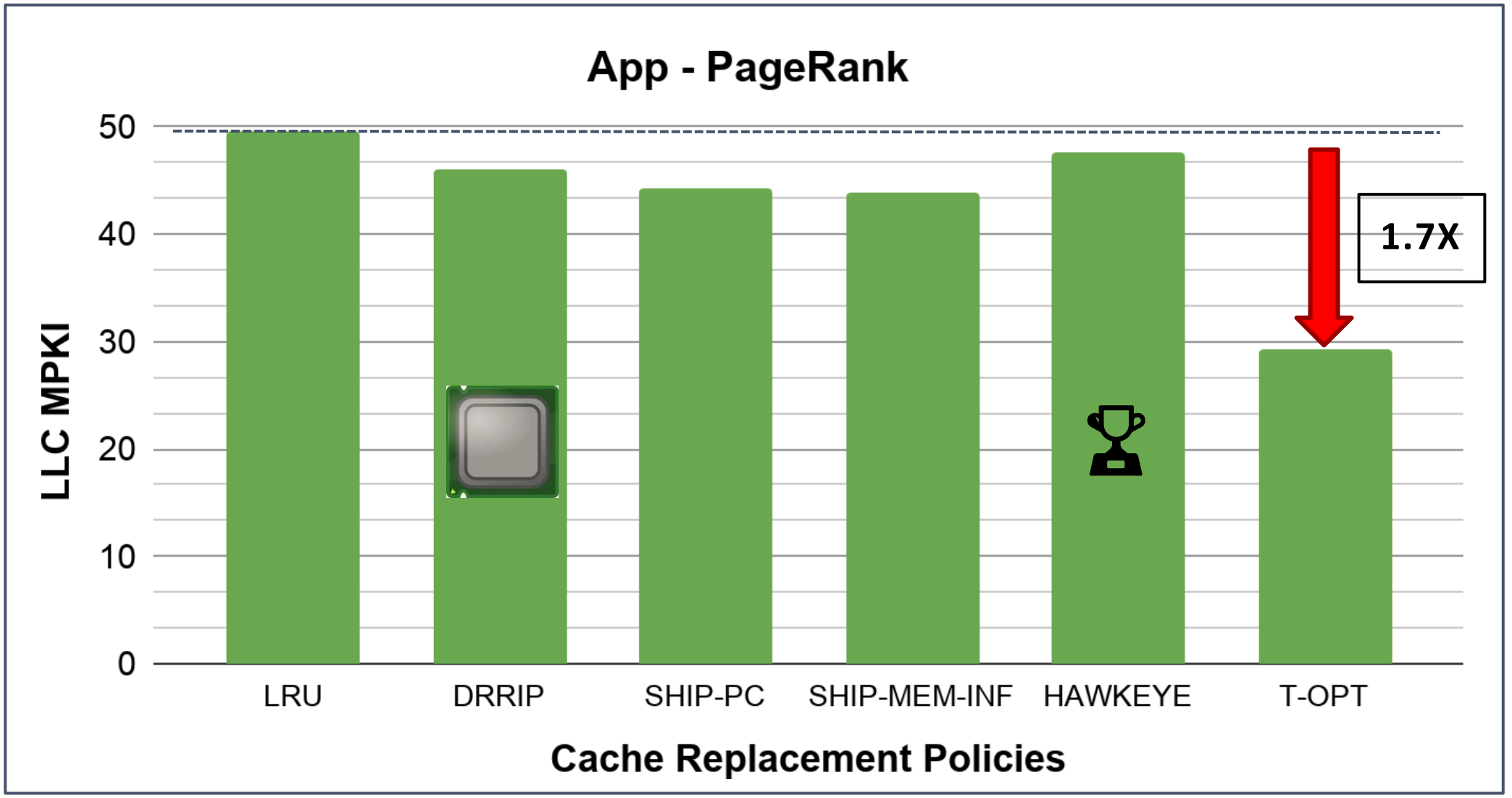


Using The Graph's Transpose For Optimal Replacement

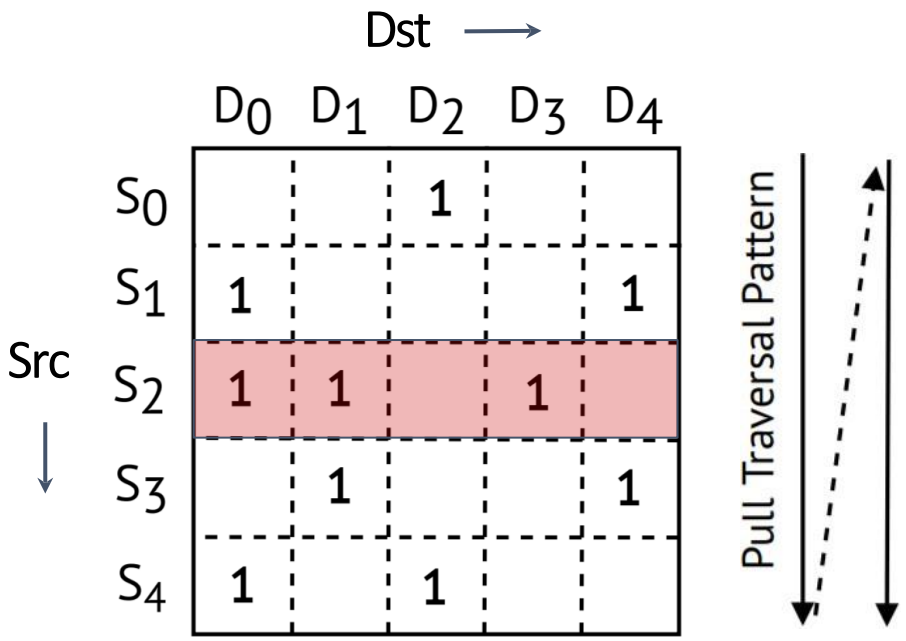


Transpose-based OPT (T-OPT) Provides Large Gains

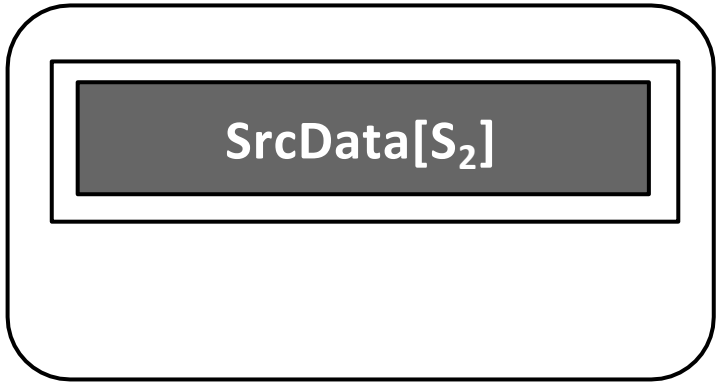
Transpose-based OPT (T-OPT) Provides Large Gains



Transpose-based OPT (T-OPT) Incurs Overheads

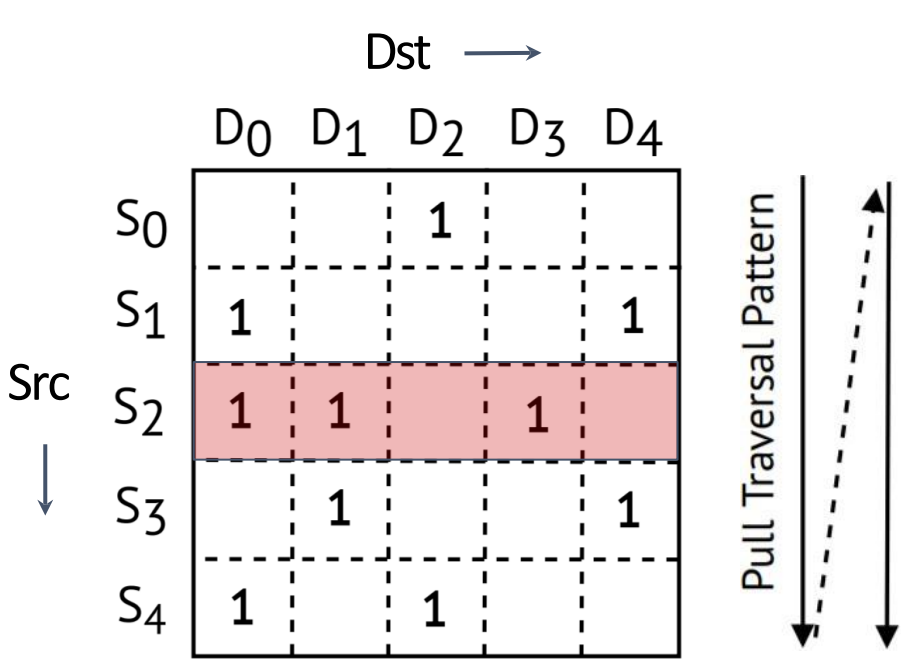


Finding Next References Using The Transpose



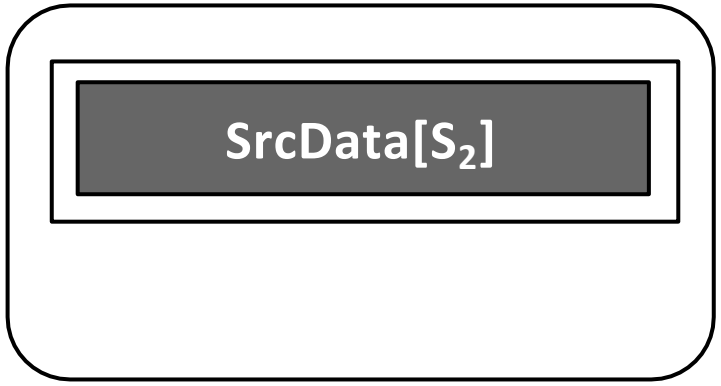
Set-Associative Cache

Transpose-based OPT (T-OPT) Incurs Overheads

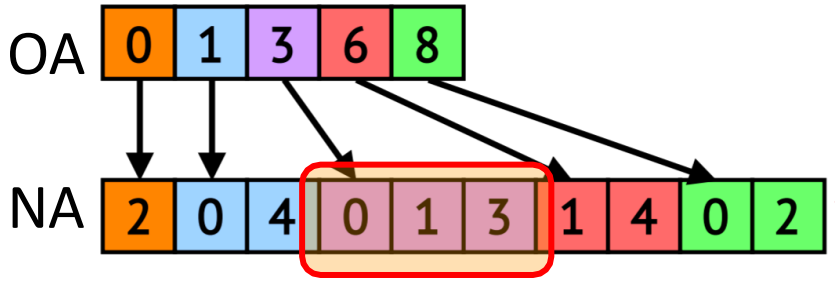


Finding Next References Using The Transpose

DRAM Access overhead



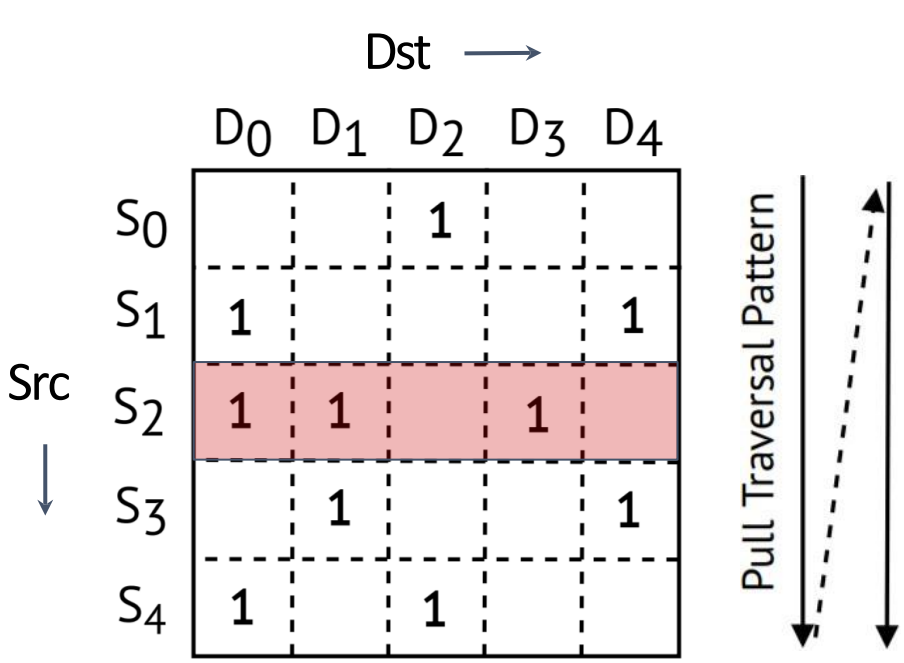
Set-Associative



CSR *(Transpose)*

Transpose accesses are irregular

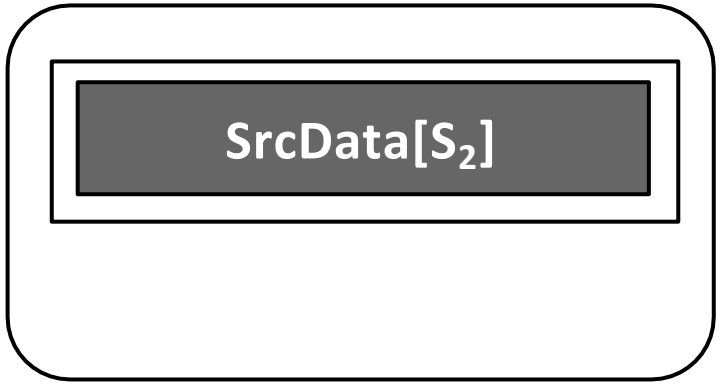
Transpose-based OPT (T-OPT) Incurs Overheads



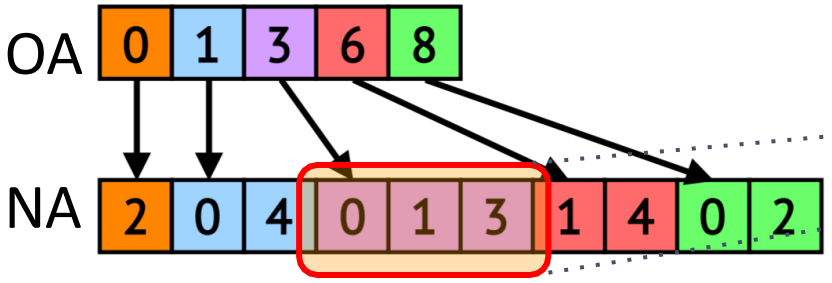
Finding Next References Using The Transpose

DRAM Access overhead

Runtime Traversal overhead



Set-Associative Cache

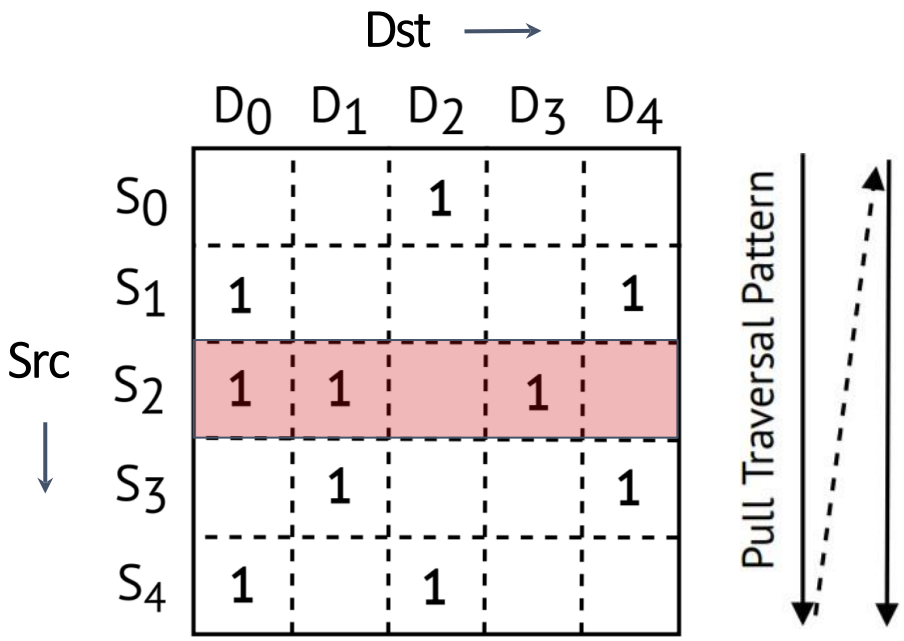


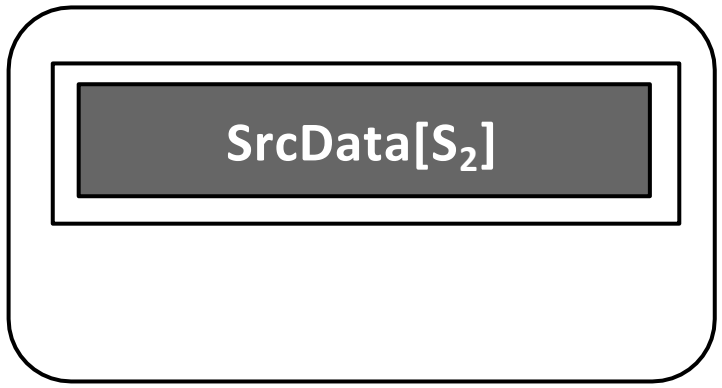
CSR *(Transpose)*

Out-Neigh(S_2): D_0 D_1 D_3

Need to scan neighbors

Transpose-based OPT (T-OPT) Incurs Overheads





Set-Associative Cache

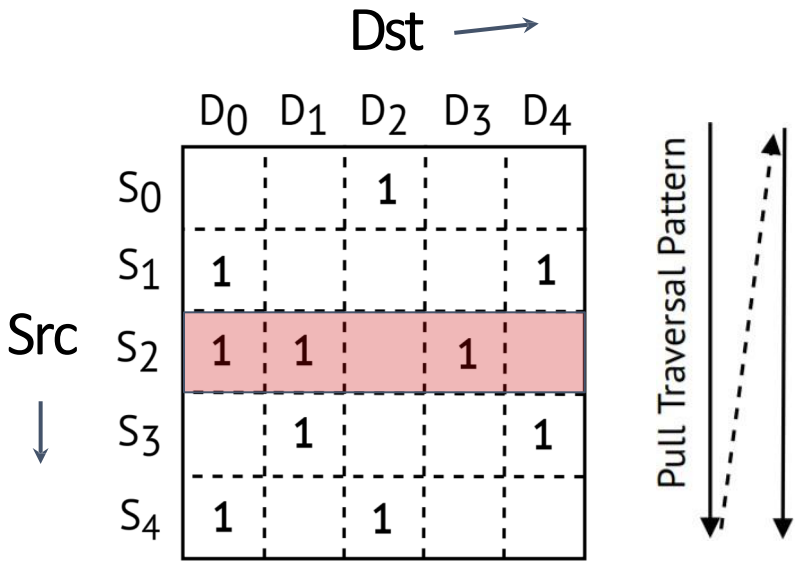
Finding Next References Using The Transpose

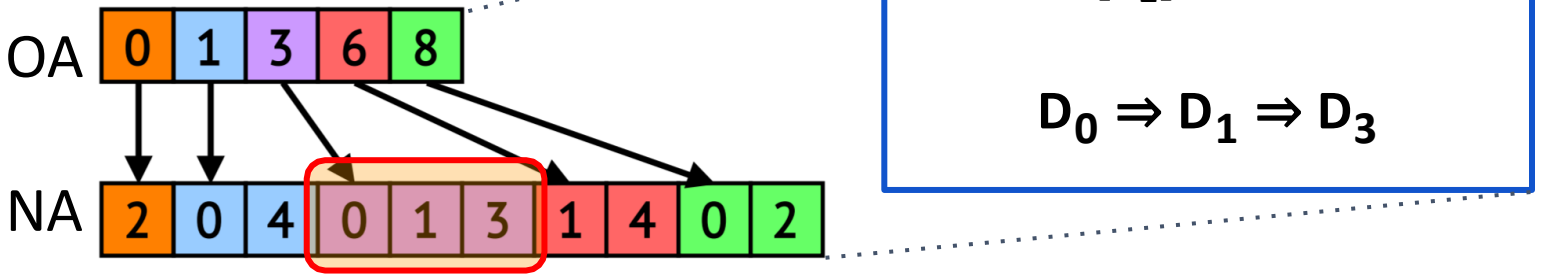
DRAM Access overhead

Runtime Traversal overhead

Question: How do we retrieve the next reference information from the graph's transpose **without all the cost of traversing the graph?**

Main Technique: Use Quantization To Compress The Transpose



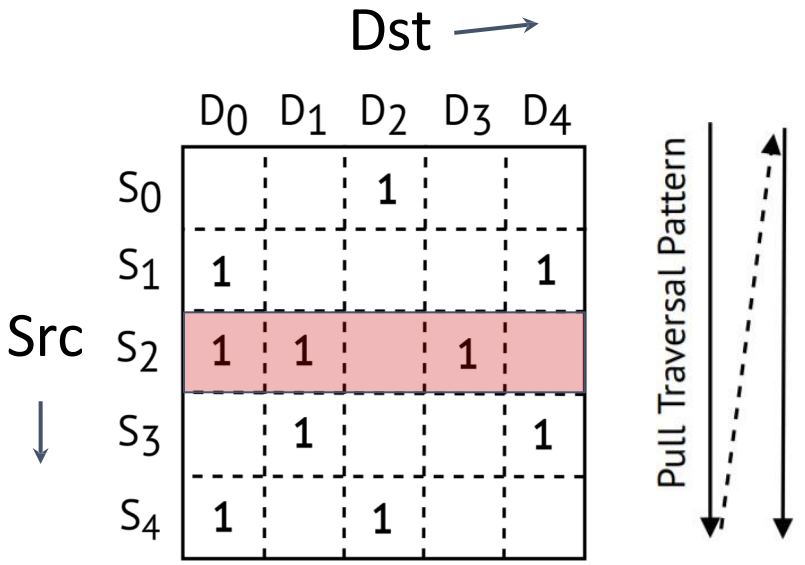


CSR
(Transpose)

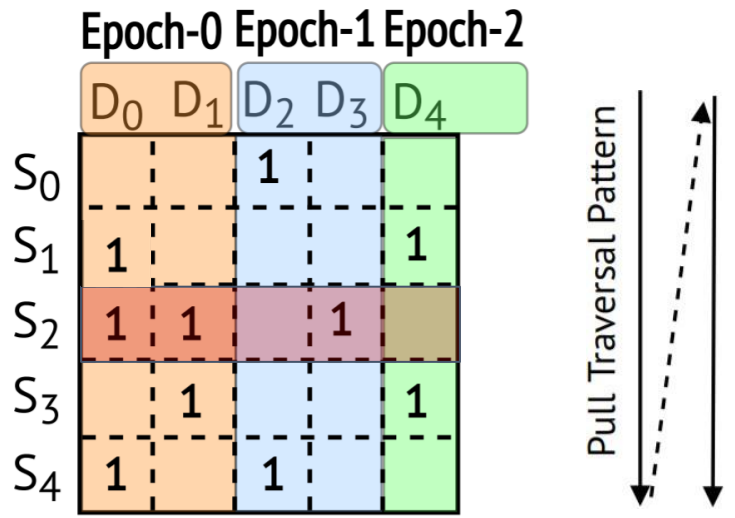
srcData[S_2] Accessed at:

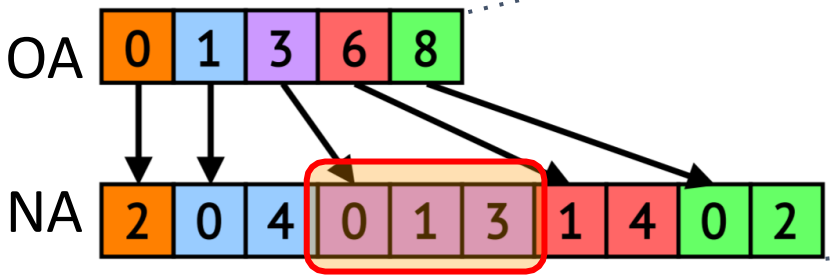
$D_0 \Rightarrow D_1 \Rightarrow D_3$

Main Technique: Use Quantization To Compress The Transpose



Divide execution into coarse-grained epochs



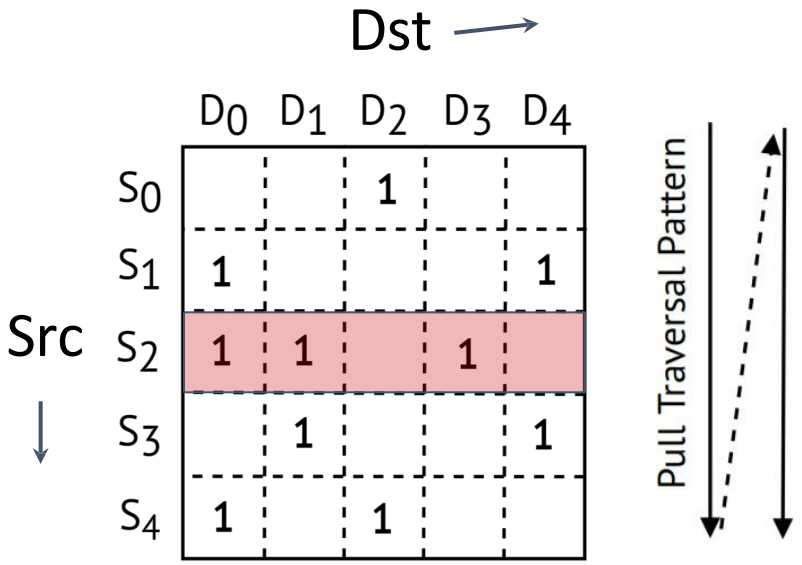


CSR
(Transpose)

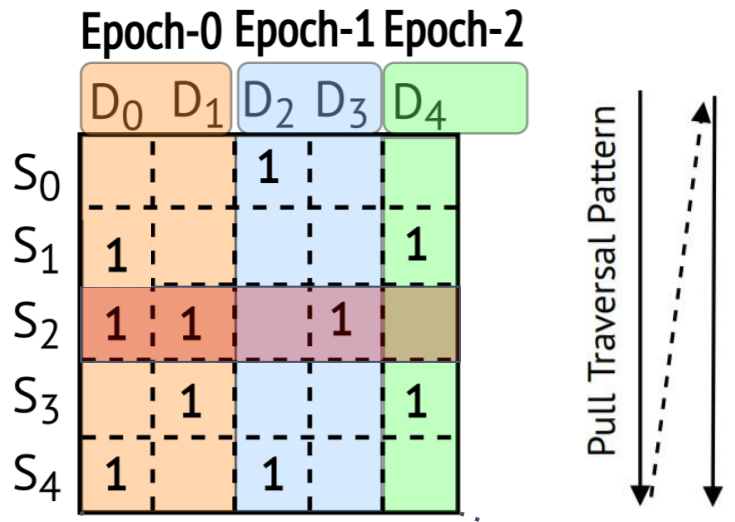
srcData[S_2] Accessed at:

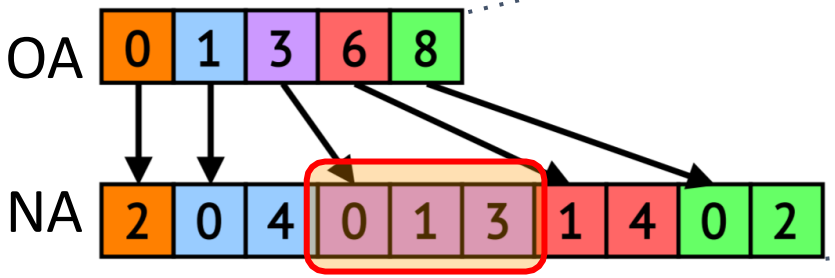
$D_0 \Rightarrow D_1 \Rightarrow D_3$

Main Technique: Use Quantization To Compress The Transpose



Divide execution into coarse-grained epochs





CSR
(Transpose)

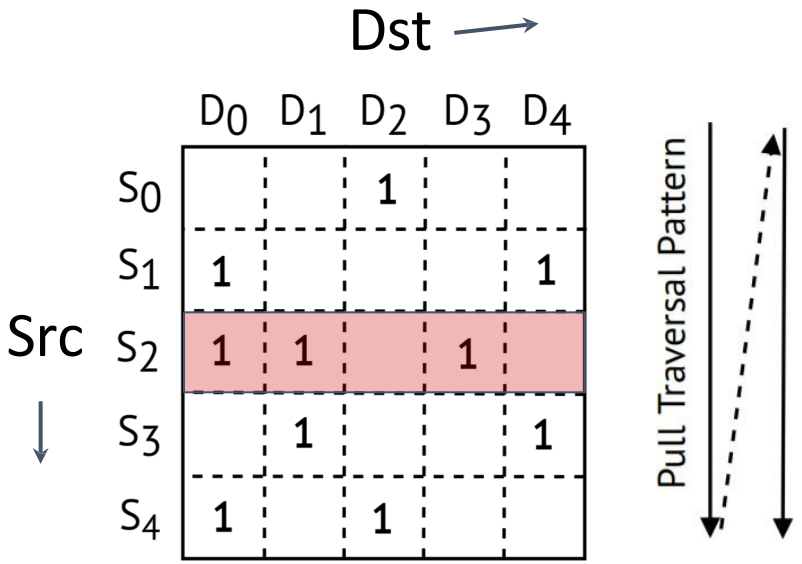
srcData[S_2] Accessed at:

$D_0 \Rightarrow D_1 \Rightarrow D_3$

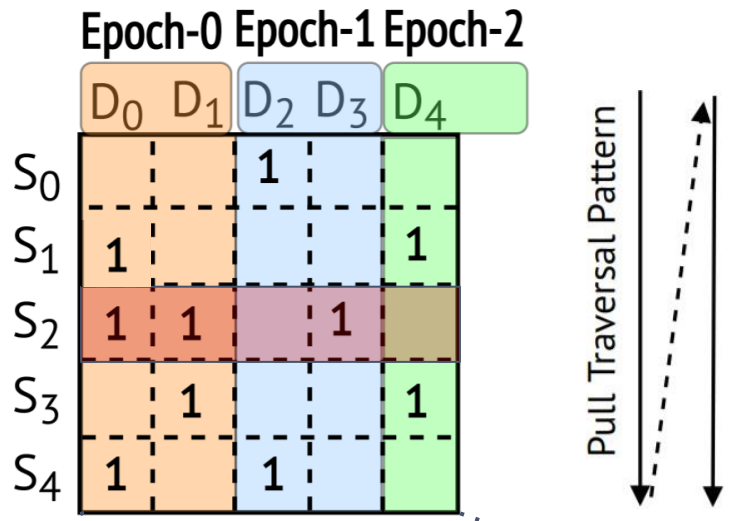
srcData[S_2] Accessed at:

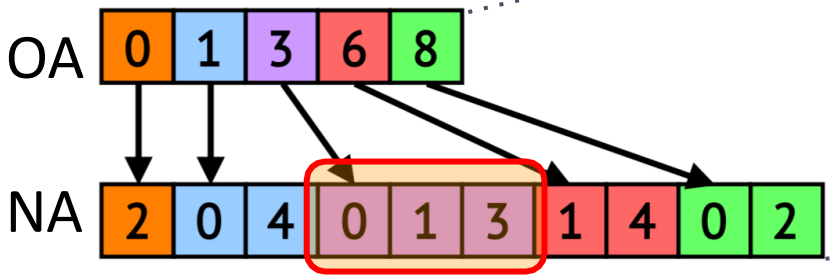
$E_0 \Rightarrow E_1$

Main Technique: Use Quantization To Compress The Transpose



Divide execution into coarse-grained epochs





CSR
(Transpose)

srcData[S_2] Accessed at:

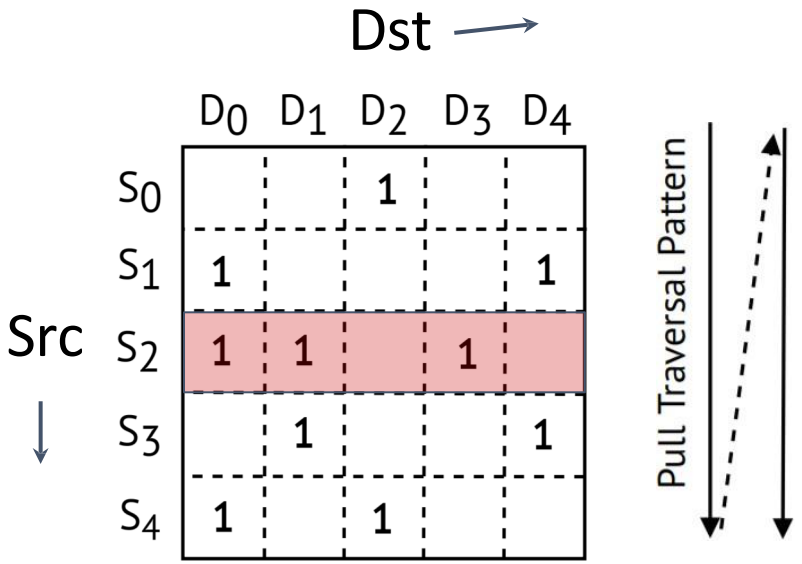
$D_0 \Rightarrow D_1 \Rightarrow D_3$

srcData[S_2] Accessed at:

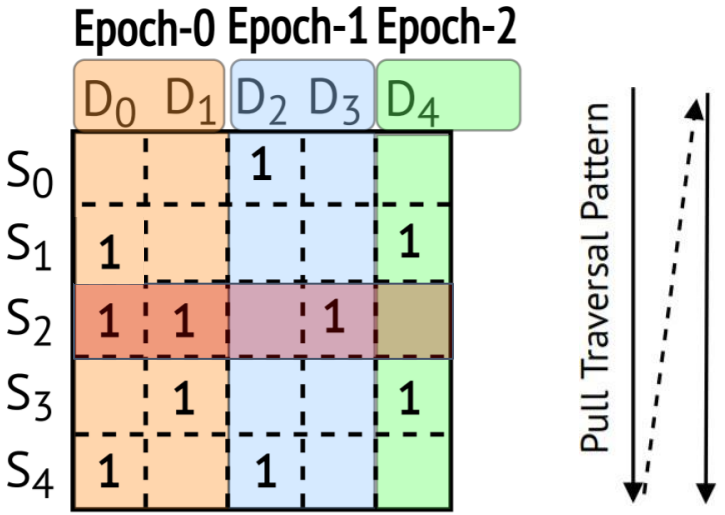
$E_0 \Rightarrow E_1$

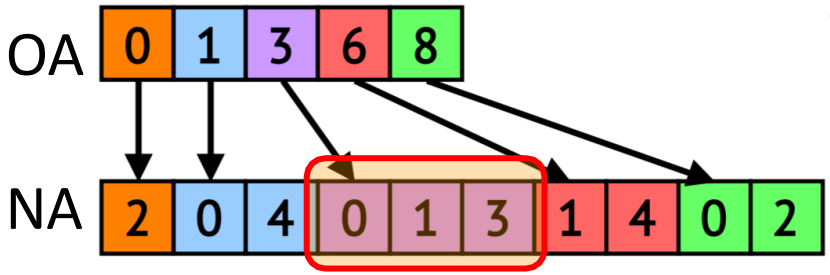
No. of epochs ⬇ ⇒ Next Ref Quantization ⬆
(accuracy vs metadata size trade-off)

Main Technique: Use Quantization To Compress The Transpose



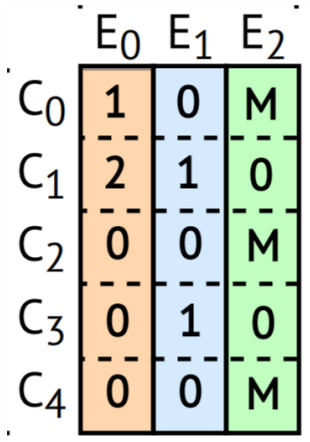
Divide execution into coarse-grained epochs





CSR
(Transpose)

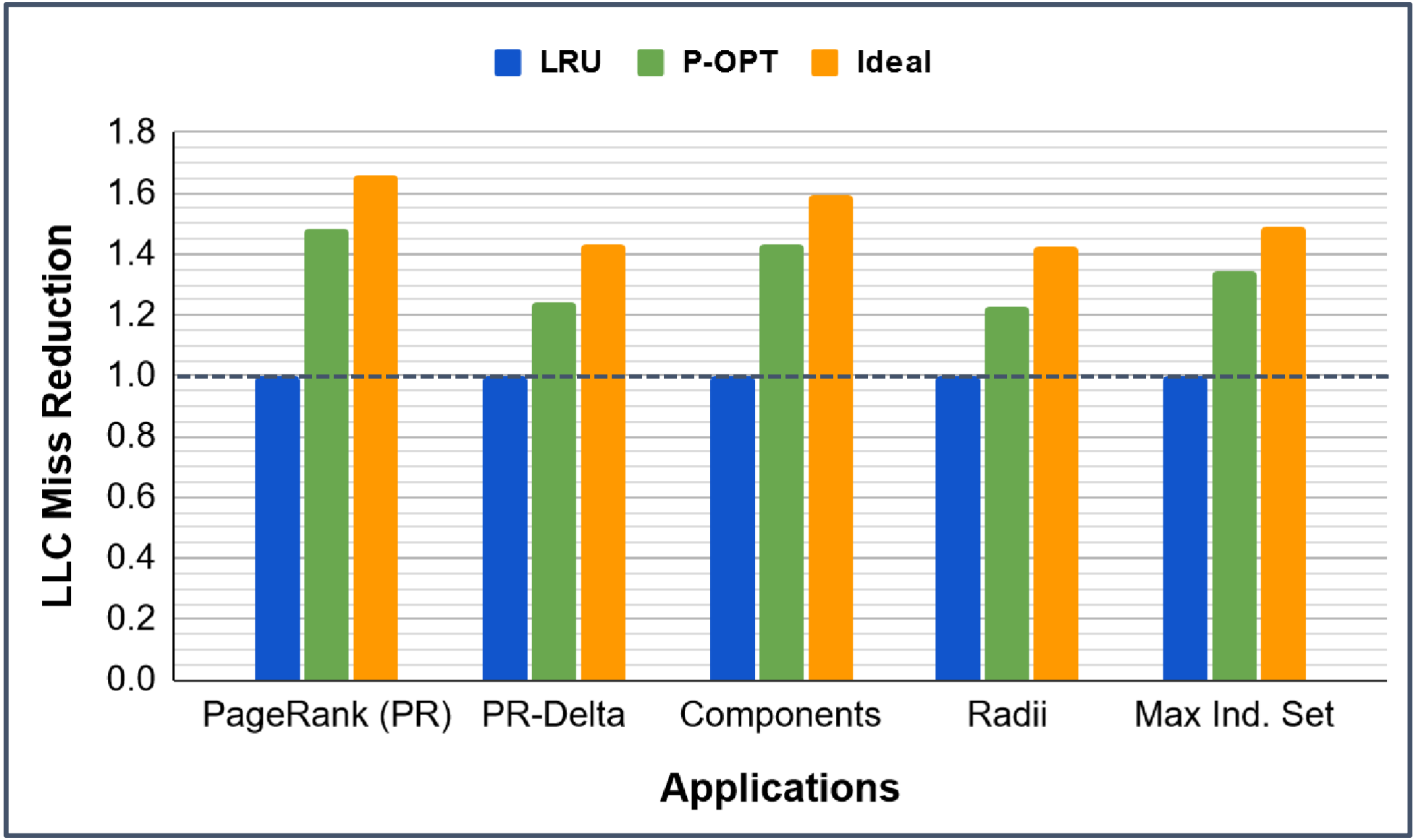
Quantization enables compression of transpose data



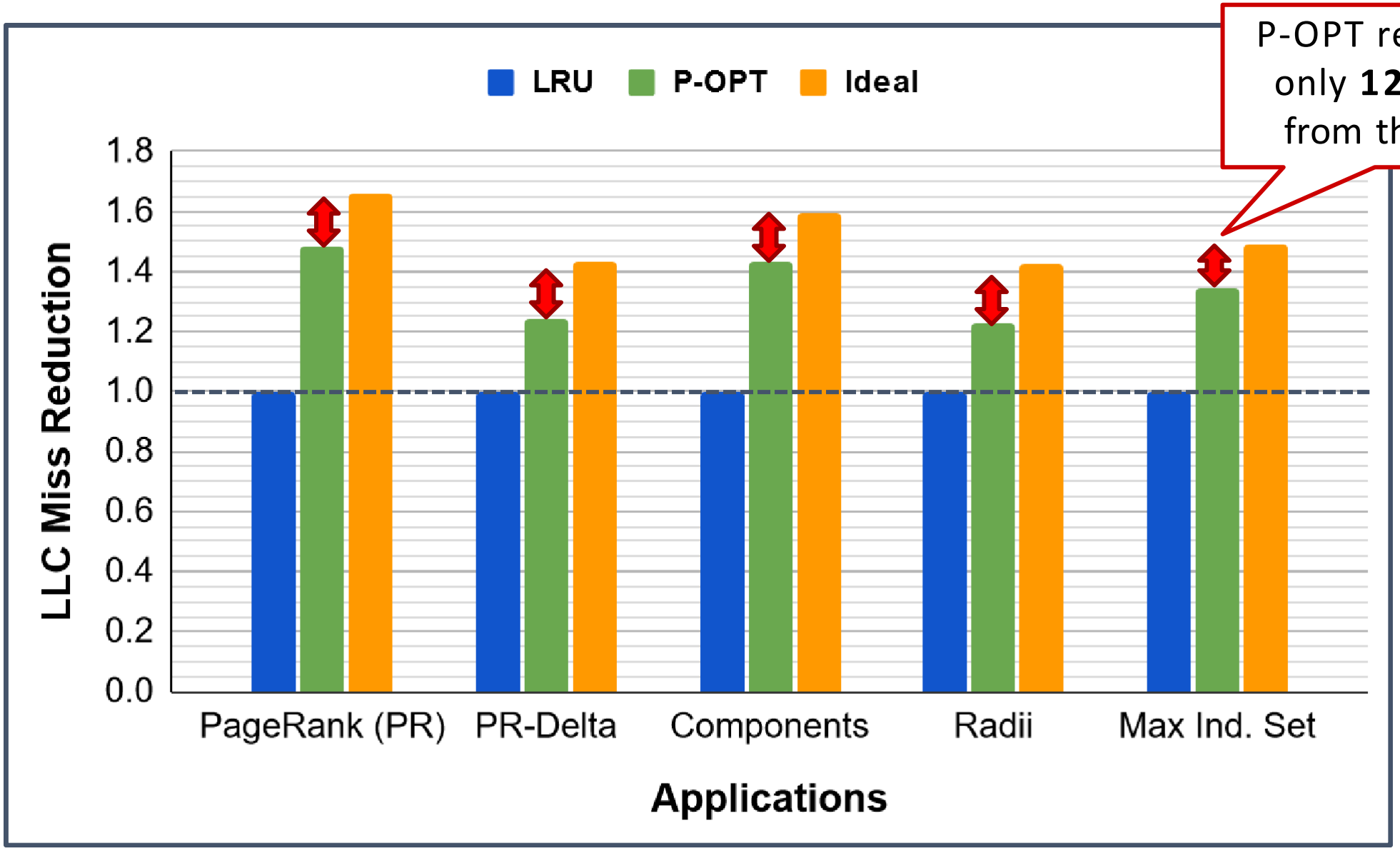
Size the reref matrix to fit in cache!

Rereference Matrix
(Quantized Transpose)

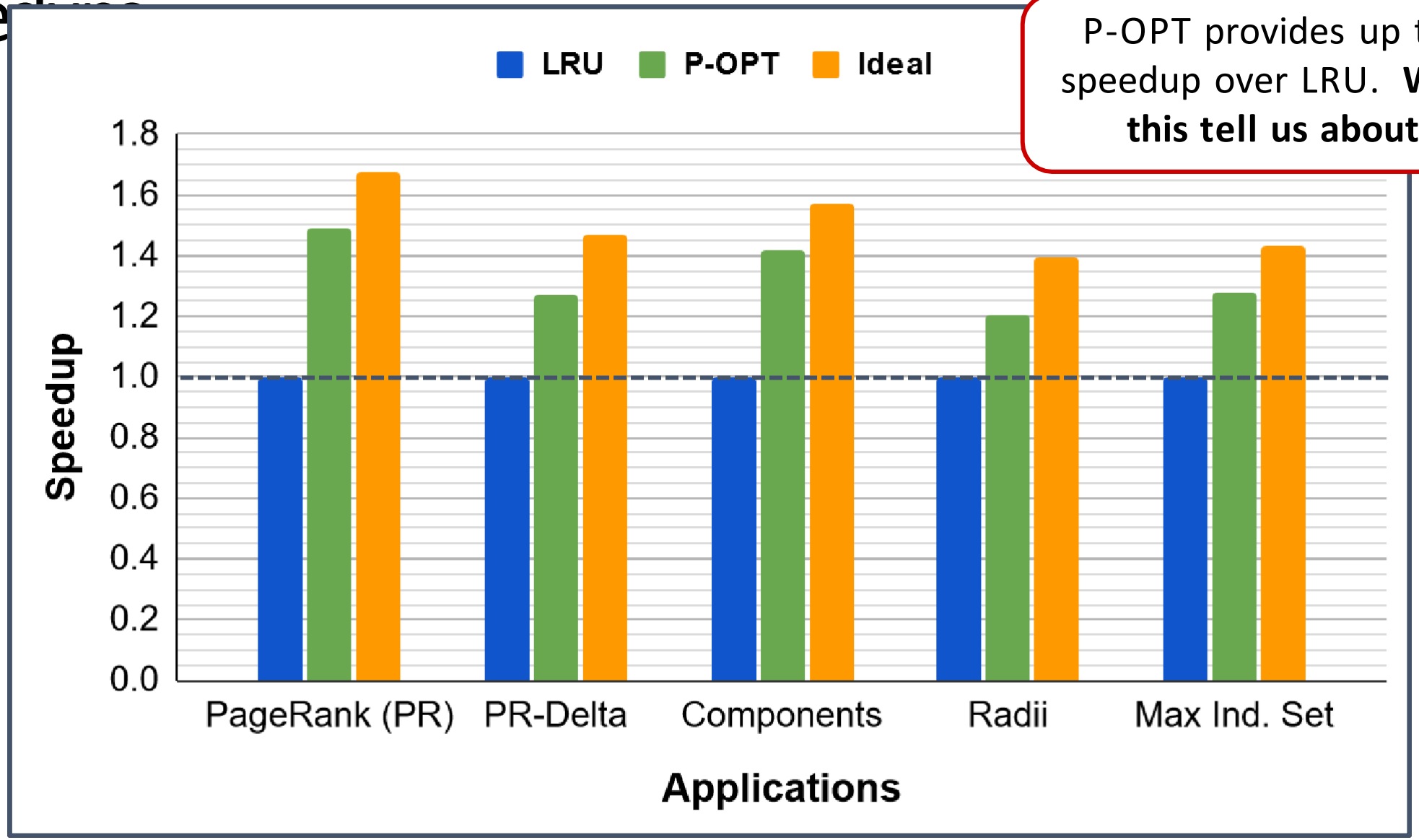
P-OPT Improves Cache Locality



P-OPT Improves Cache Locality



P-OPT's LLC Miss Reductions Directly Translate To Speedups



P-OPT provides up to **1.56x** speedup over LRU. **What does this tell us about LRU?**

What did we just learn?

- Sparse problems are ones that manipulate large, mostly-zero matrices
- Sparsity makes caching a useful part of the matrix hard
- Roofline model shows how close to peak perf. an app is
- Propagation blocking bins updates making irregular data fit in cache
- P-OPT is a *practical* implementation of Belady's OPT for graphs

(with acknowledgements to Vignesh Balaji, CMU ECE PhD 2021, now at Nvidia for contributions to this

Takeaways

- P-OPT achieves close to ideal performance (*quantization can be an effective tool in making a design practical*)
- Creative thinking can provide solutions when hardware can be customized, e.g. lossy compression (epochs) and other-than-LRU prediction (transpose)