P-OPT: Practical Optimal Cache Replacement for Graph Analytics

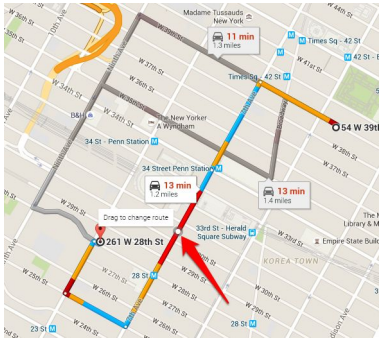
Vignesh Balaji
CMU

Neal Crago
NVIDIA

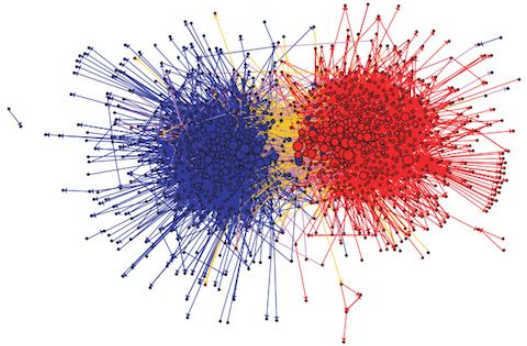
Aamer Jaleel
NVIDIA

Brandon Lucia
CMU

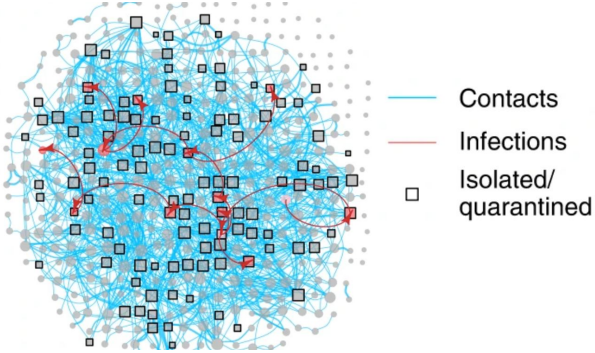
Graph Analytics Has Many Important Applications



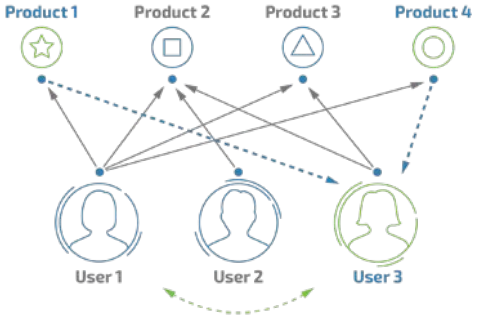
Path Planning



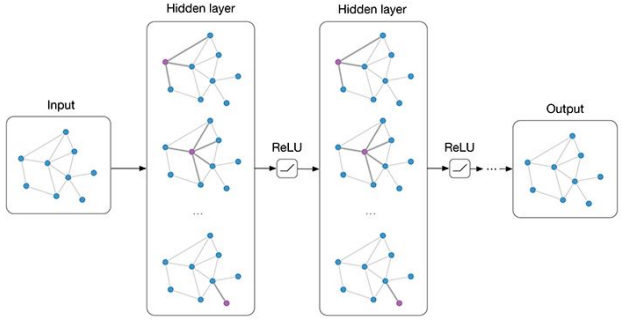
Social network analysis



Epidemiology



Recommender systems



Graph Neural Networks

Large Graphs Can Be Processed In A Single Node

Large Graphs Can Be Processed In A Single Node

Input Graph	Vertices	Edges	Memory
Twitter-2010	41M	1.4B	5.5GB
SK-WebCrawl	50.6M	1.95B	7.6GB
UK-WebCrawl	106M	6.6B	50GB
Yahoo Search	1.41B	6.6B	65GB
Hyperlink Graph	1.72B	64.4B	499GB
Facebook-2015	2B	400B	2.9TB

Compressed Representation + Vertex Data

Large Graphs Can Be Processed In A Single Node

Input Graph	Vertices	Edges	Memory
Twitter-2010	41M	1.4B	5.5GB
SK-WebCrawl	50.6M	1.95B	7.6GB
UK-WebCrawl	106M	6.6B	50GB
Yahoo Search	1.41B	6.6B	65GB
Hyperlink Graph	1.72B	64.4B	499GB
Facebook-2015	2B	400B	2.9TB

Large Graphs Can Be Processed In A Single Node

Input Graph	Vertices	Edges	Memory
Twitter-2010	41M	1.4B	5.5GB
SK-WebCrawl	50.6M	1.95B	7.6GB
UK-WebCrawl	106M	6.6B	50GB
Yahoo Search	1.41B	6.6B	65GB
Hyperlink Graph	1.72B	64.4B	499GB
Facebook-2015	2B	400B	2.9TB

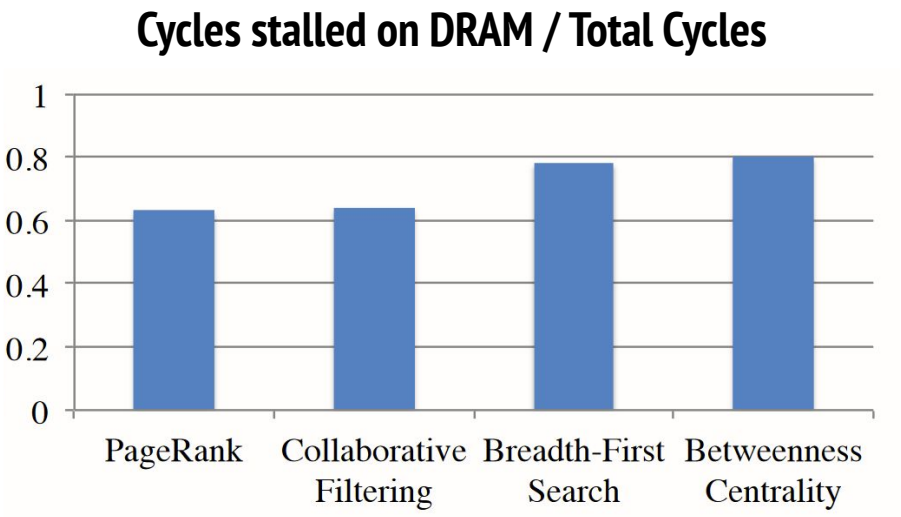
Programmability ↑

Performance ↑

[Programmability] *"WTF: The Who To Follow Service at Twitter" WWW 13;* [Performance] *"Scalability! But at what COST?" Usenix HOTOS'15*

Single Node Graph Processing Performance is Sub-Optimal

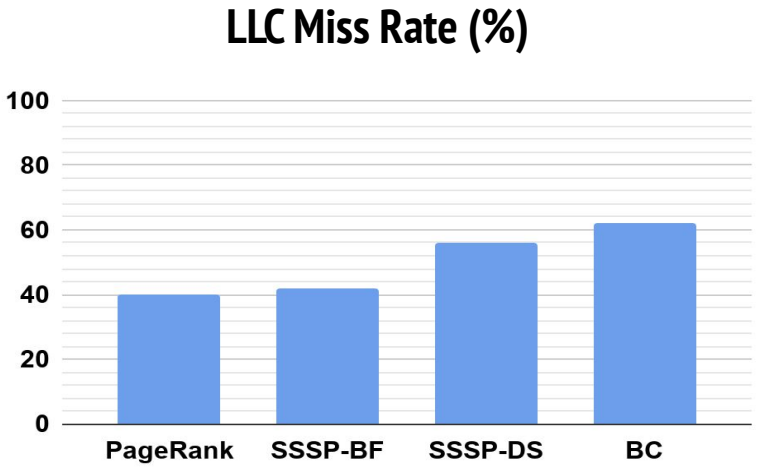
Single Node Graph Processing Performance is Sub-Optimal

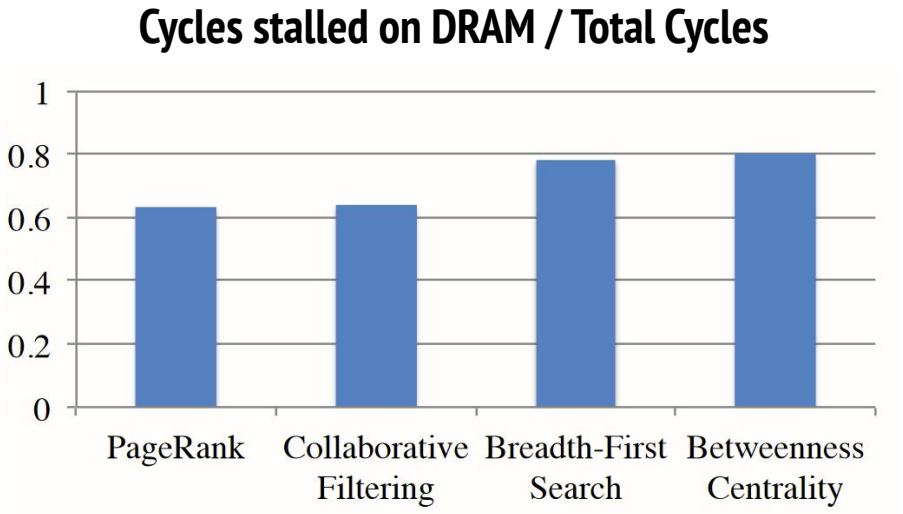


Graph Application Performance is DRAM-latency bound

Right Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

Single Node Graph Processing Performance is Sub-Optimal





High LLC Miss Rate leads to many long-latency DRAM accesses

Graph Application Performance is DRAM-latency bound

Right Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

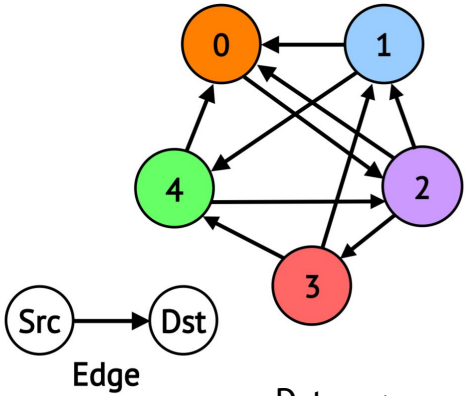
Outline

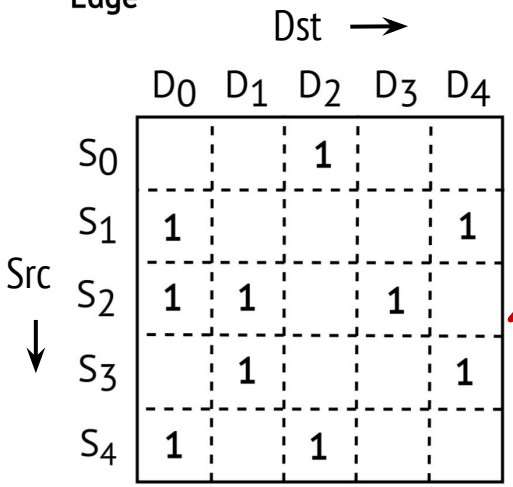
- **Primary Bottleneck of Graph Analytics ⇒ Poor Cache Locality ✓**
- Reasons for Poor Cache Locality
- Belady's OPT Replacement Policy Is Viable for Graph Processing
- P-OPT: A Practical Optimal Cache Replacement Policy

Outline

- Primary Bottleneck of Graph Analytics ⇒ Poor Cache Locality ✓
- **Reasons for Poor Cache Locality ⇦**
  - **Irregular Memory Accesses**
  - **Existing Replacement Policies Are Insufficient**
- Belady's OPT Replacement Policy Is Viable for Graph Processing
- P-OPT: A Practical Optimal Cache Replacement Policy

Graph Processing Overview



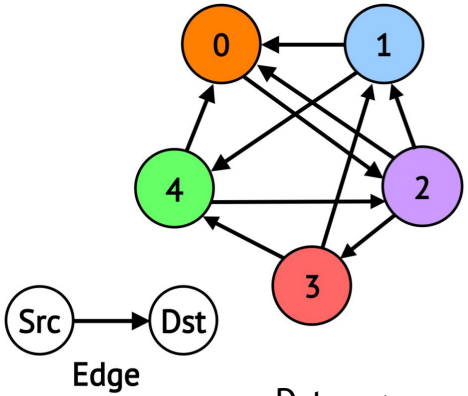


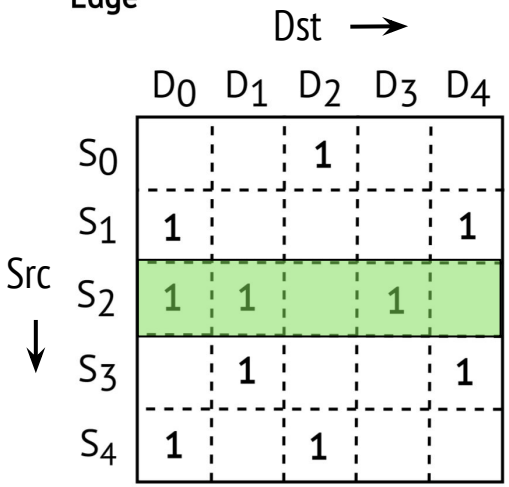
Typical graphs are extremely sparse

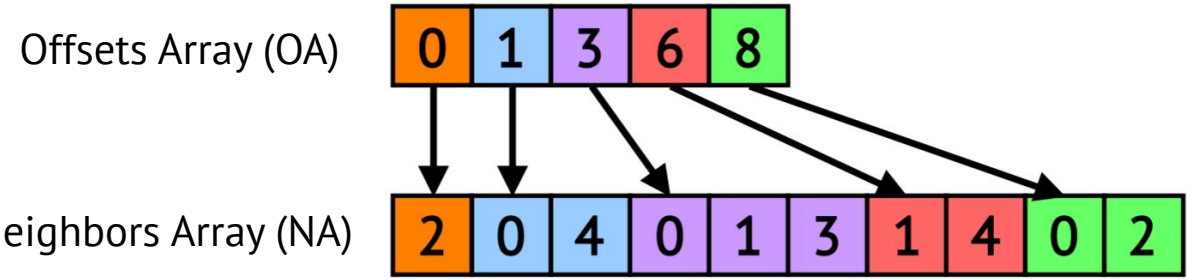
⬇️

% of Non-Zero Entries ~ 10^{-5}

Graph Processing Overview



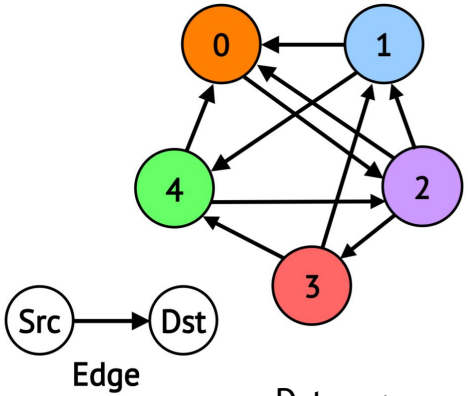


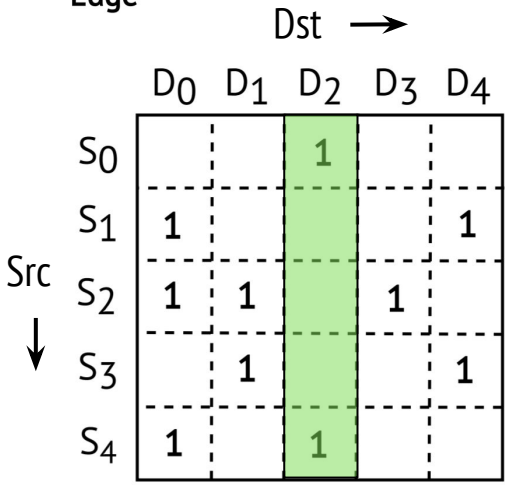


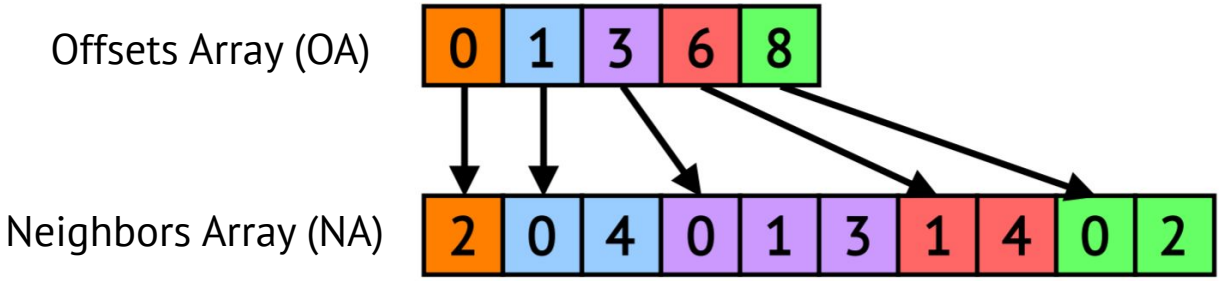
Non-Zero Coordinates sorted by SrcIDs

Compressed Sparse Row (CSR)
Outgoing Neighbors

Graph Processing Overview

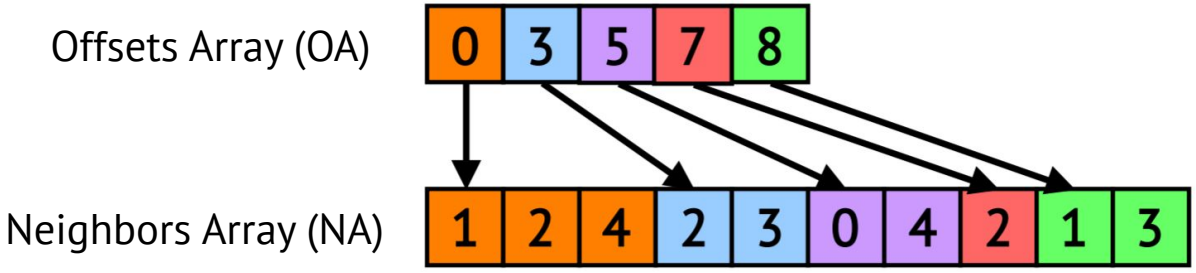






Non-Zero Coordinates sorted by SrcIDs

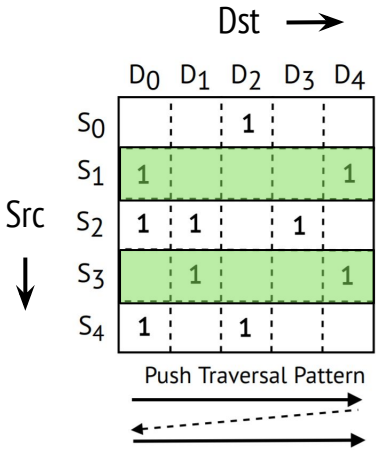
Compressed Sparse Row (CSR)
Outgoing Neighbors



Non-Zero Coordinates sorted by DstIDs

Compressed Sparse Column (CSC)
Incoming Neighbors

Graph Processing Overview

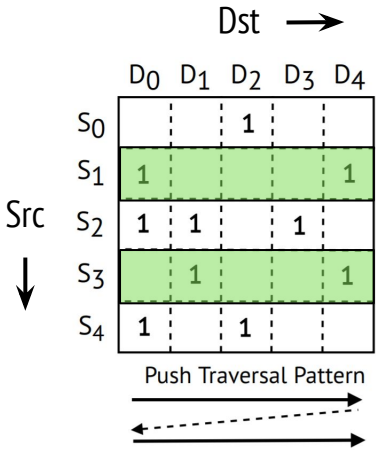


Push Execution

```
for src in Frontier:
    for dst in out_neighs(src):
        dstData[dst] += srcData[src]
```

CSR Traversal

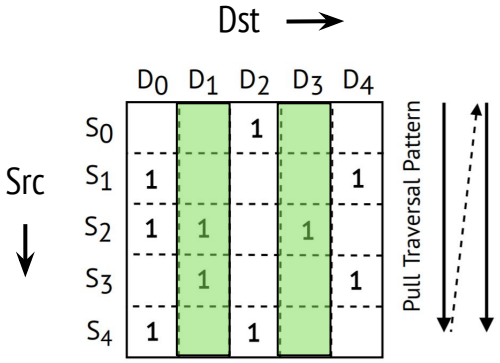
Graph Processing Overview



Push Execution

```
for src in Frontier:
    for dst in out_neighs(src):
        dstData[dst] += srcData[src]
```

CSR Traversal

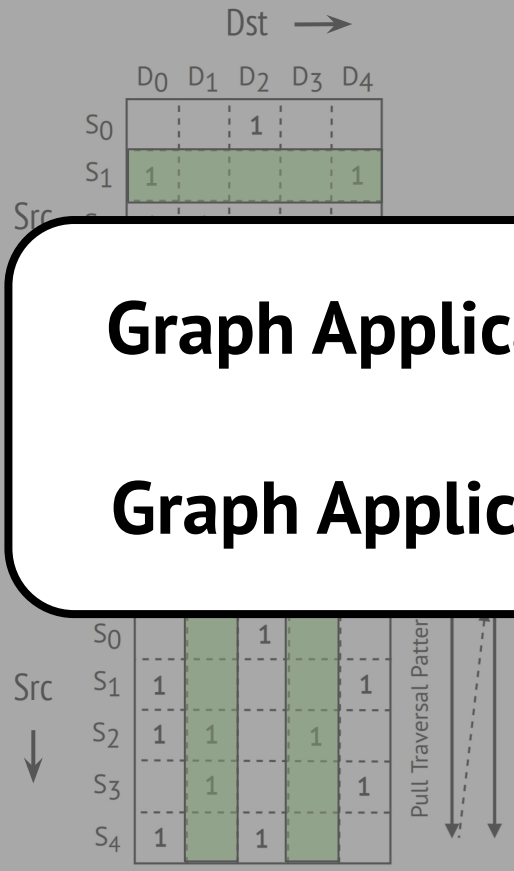


Pull Execution

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

CSC Traversal

Graph Processing Overview



Push Execution

```
for src in Frontier:
    for dst in out_neighs(src):
```

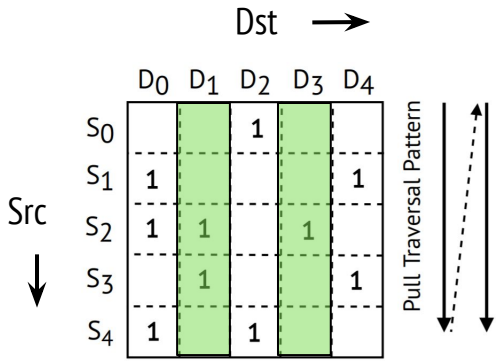
Graph Applications switch between Push and Pull

Graph Applications require both the CSR *and* CSC

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

CSC Traversal

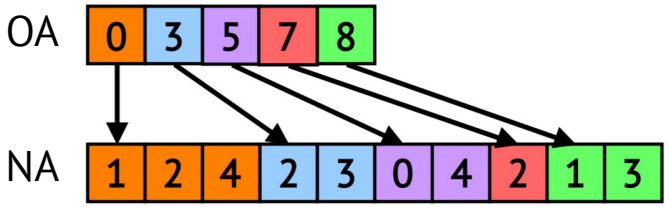
Source of Poor Locality ⇒ Irregular Memory Accesses



Pull Execution

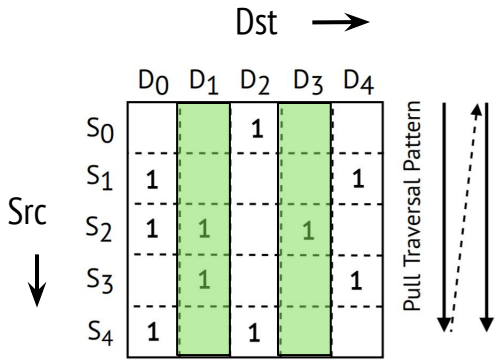
```
for dst in G:
    for src in in_neighs(dst):
        if src in Frontier:
            dstData[dst] += srcData[src]
```

CSC Traversal



CSC

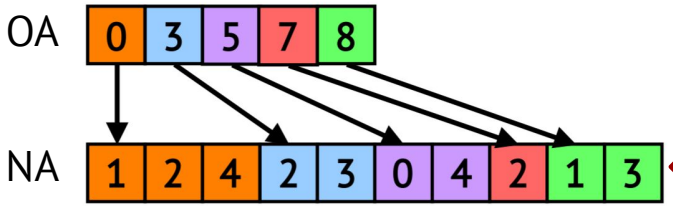
Source of Poor Locality ⇒ Irregular Memory Accesses



Pull Execution

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

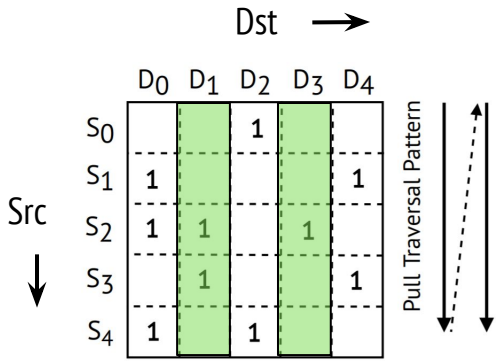
CSC Traversal



CSC

CSC contents can be arbitrarily ordered

Source of Poor Locality ⇒ Irregular Memory Accesses

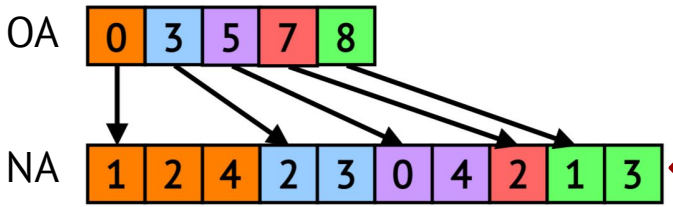


Pull Execution

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

CSC Traversal

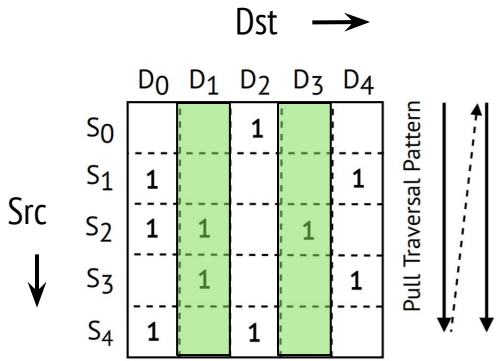
Irregular Memory Accesses



CSC

CSC contents can be arbitrarily ordered

Source of Poor Locality ⇒ Irregular Memory Accesses

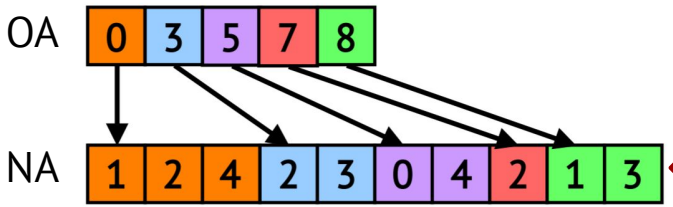


Pull Execution

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

CSC Traversal

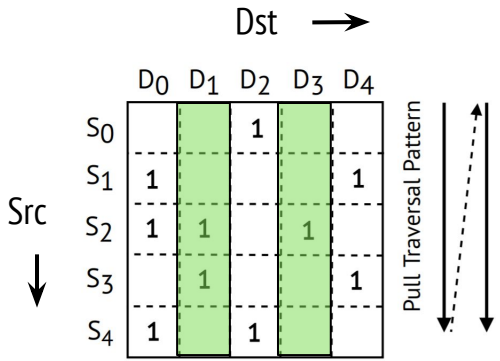
Irregular Memory Accesses



CSC

Irregular Data Footprint >> LLC Size

Source of Poor Locality ⇒ Irregular Memory Accesses

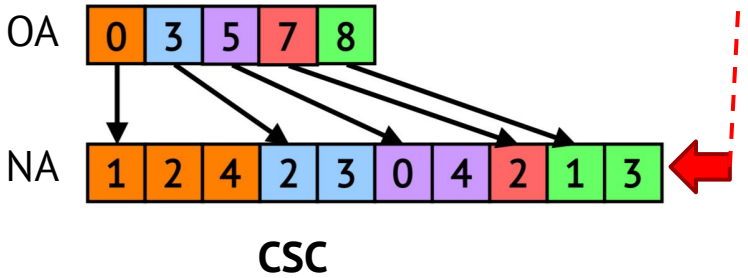


Pull Execution

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

CSC Traversal

Irregular Memory Accesses



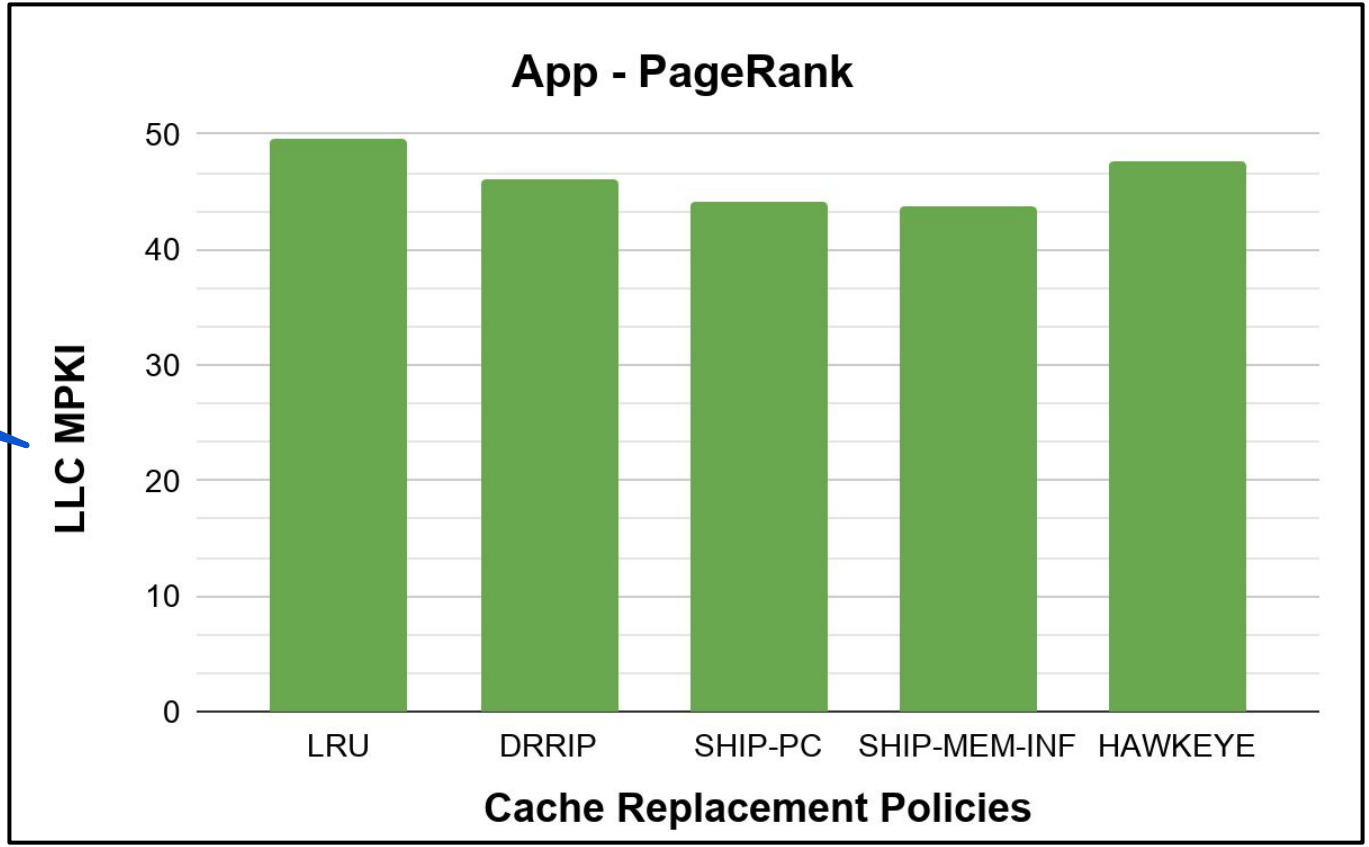
Irregular Data Footprint >> LLC Size

Size of srcData ~ **128MB** (32M * 4B)

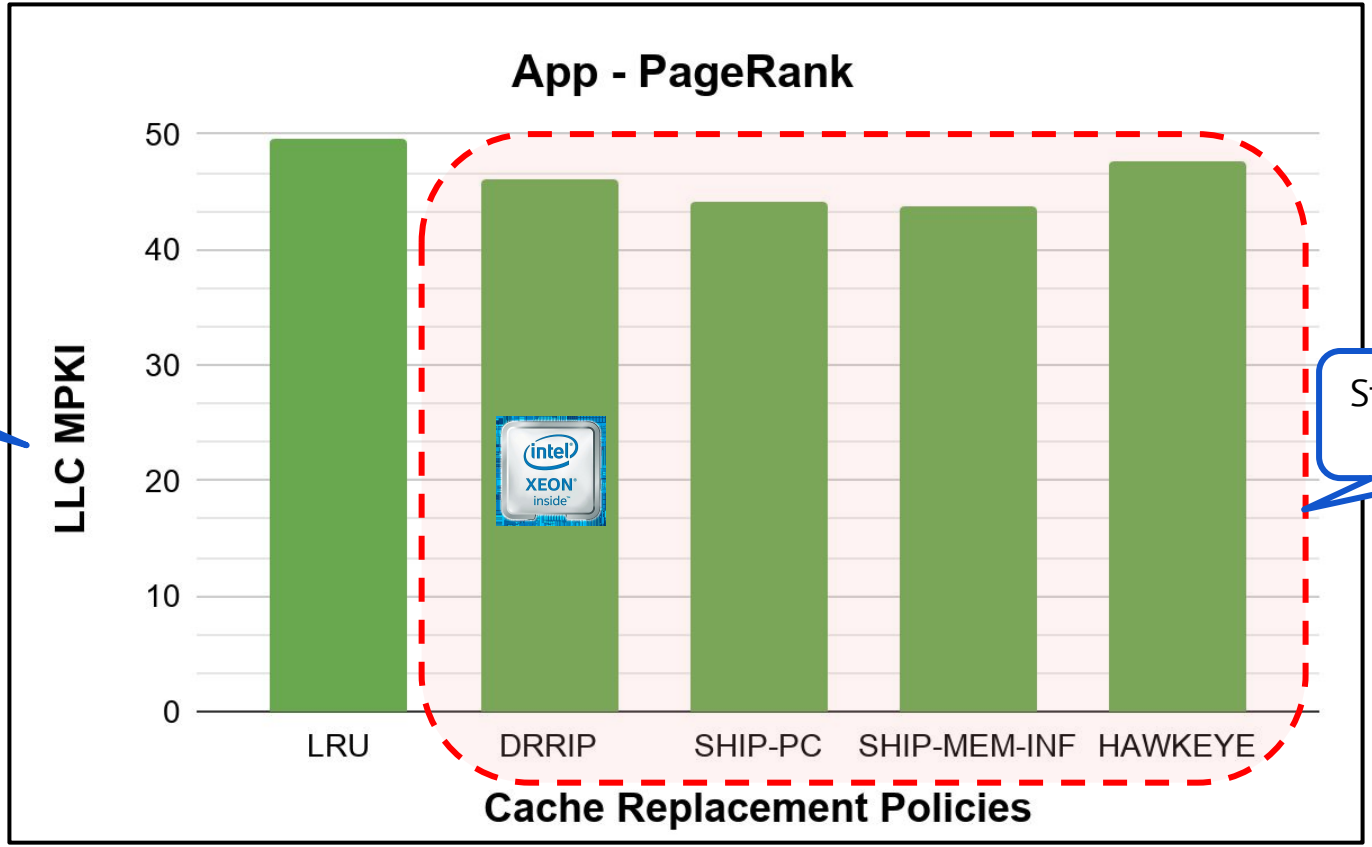
Existing Replacement Policies Are Insufficient

Existing Replacement Policies Are Insufficient

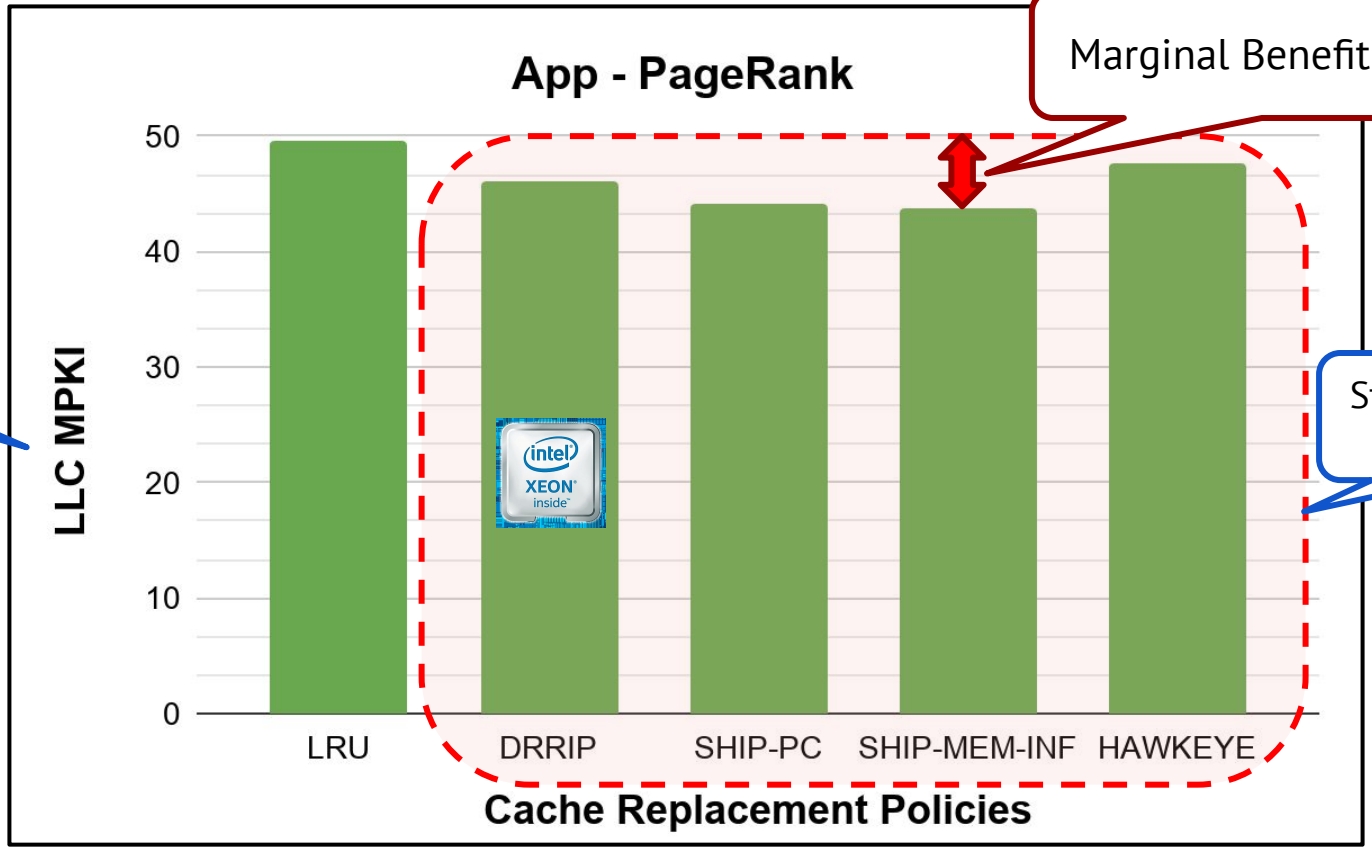
Lower is Better



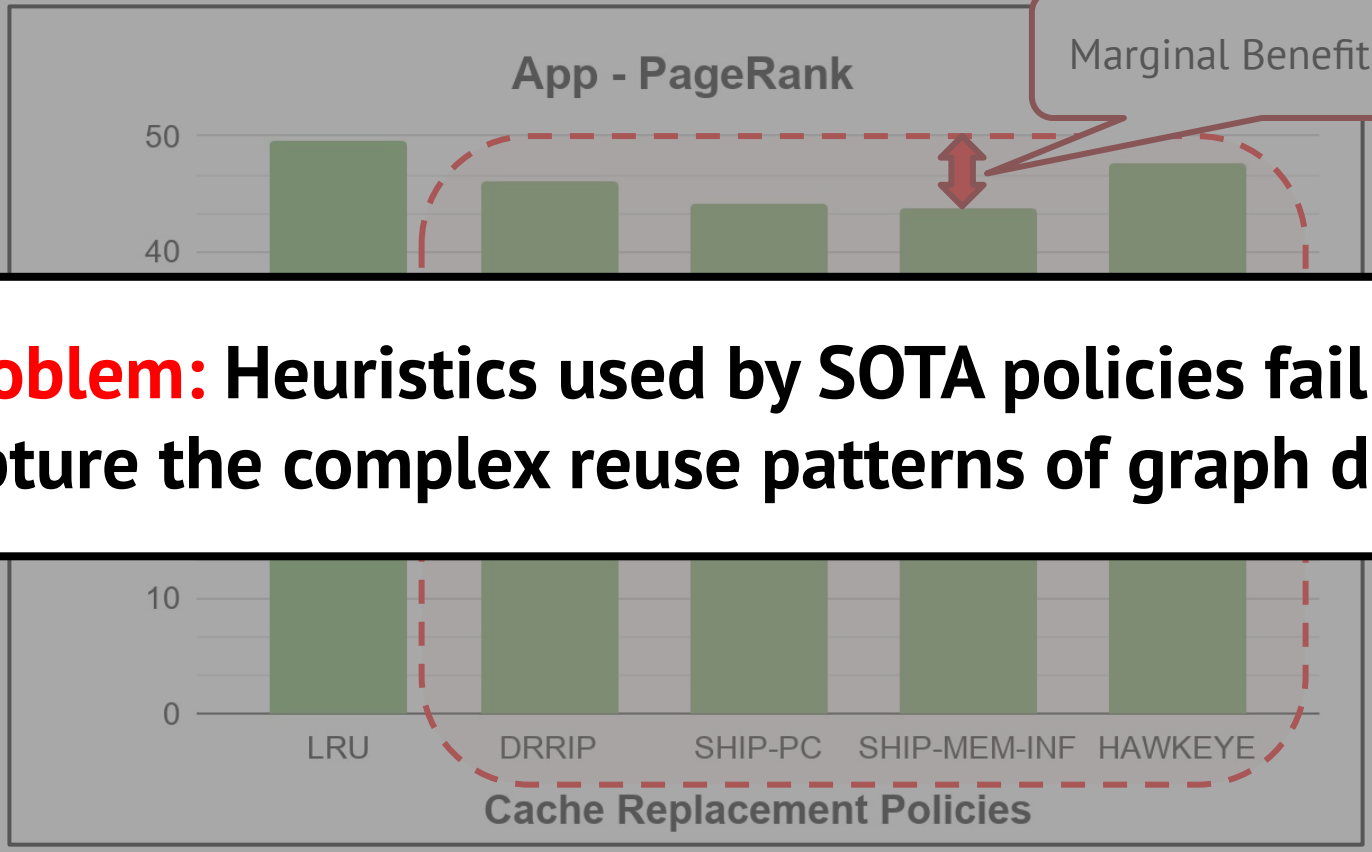
Existing Replacement Policies Are Insufficient



Existing Replacement Policies Are Insufficient

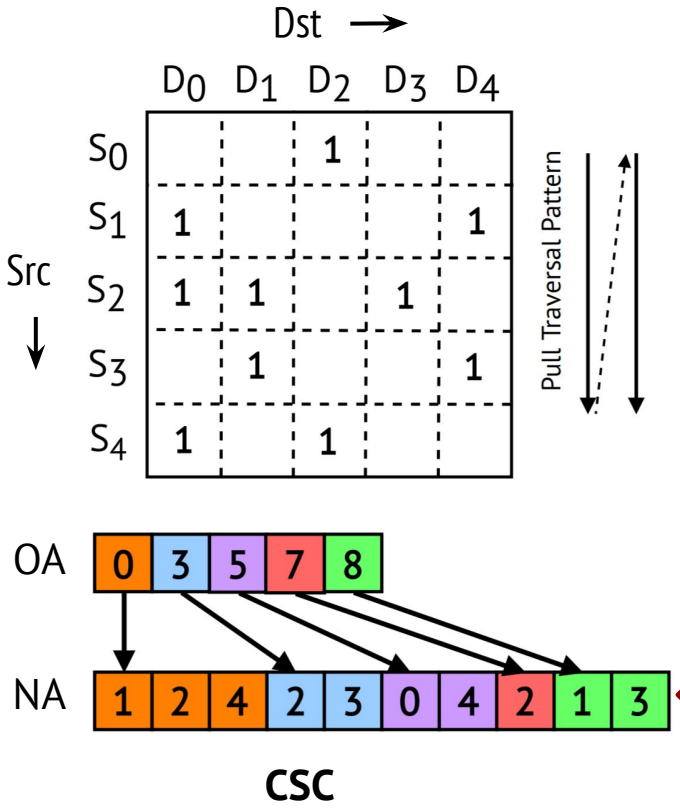


Existing Replacement Policies Are Insufficient



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

Capturing (Irregular) Graph Data Reuse Is Challenging



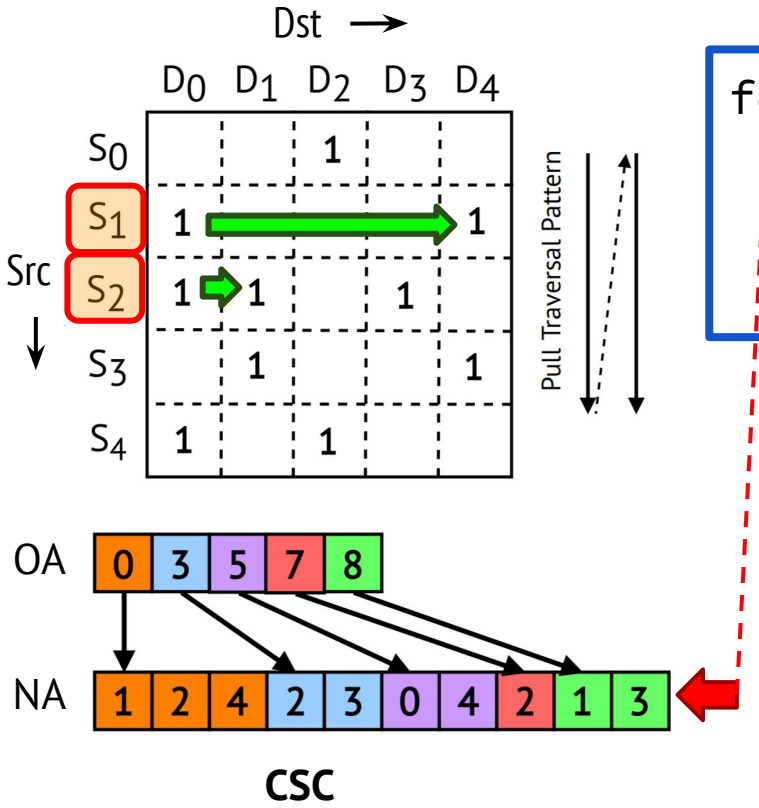
Pull Execution

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

CSC Traversal

Irregular Memory Accesses

Capturing (Irregular) Graph Data Reuse Is Challenging



Pull Execution

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

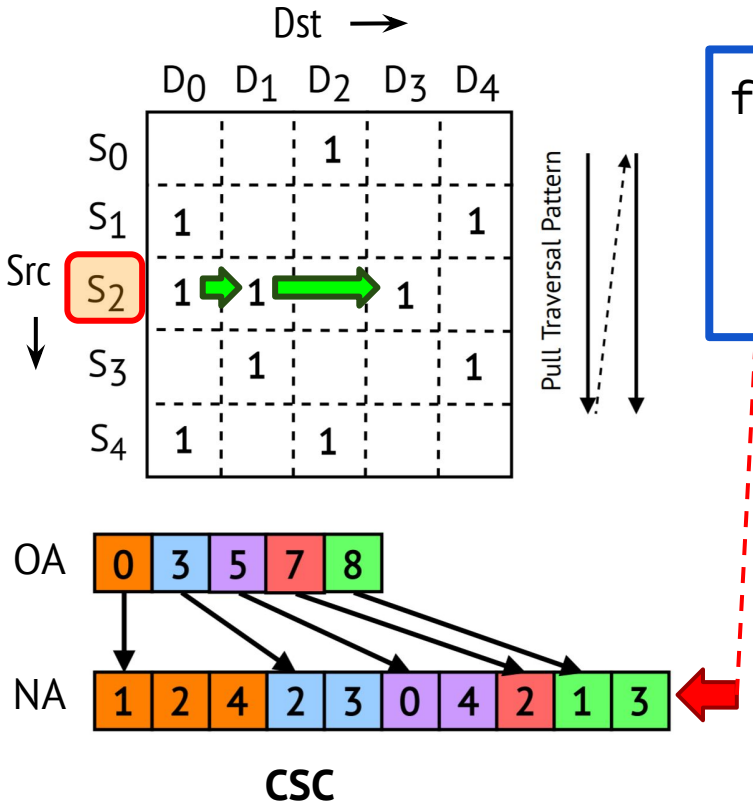
CSC Traversal

Irregular Memory Accesses

Irregular Data (srcData) Reuse is:

1. **Unique for each vertex**

Capturing (Irregular) Graph Data Reuse Is Challenging



Pull Execution

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

CSC Traversal

Irregular Memory Accesses

Irregular Data (srcData) Reuse is:

1. Unique for each vertex
2. **Dynamically Varying**

Outline

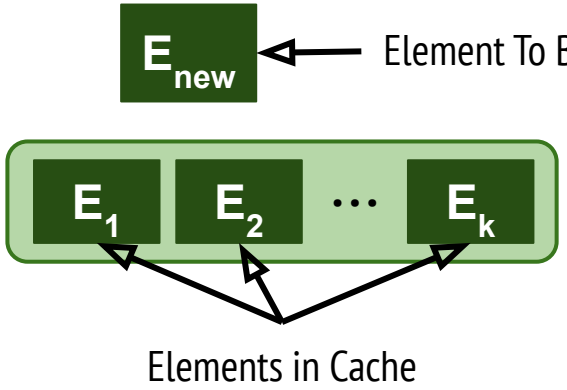
- Primary Bottleneck of Graph Analytics ⇒ Poor Cache Locality ✓
- **Reasons for Poor Cache Locality ✓**
  - **Irregular Memory Accesses**
  - **Existing Replacement Policies Are Insufficient**
- Belady's OPT Replacement Policy Is Viable for Graph Processing
- P-OPT: A Practical Optimal Cache Replacement Policy

Outline

- Primary Bottleneck of Graph Analytics ⇒ Poor Cache Locality ✓
- Reasons for Poor Cache Locality ✓
  - Irregular Memory Accesses
  - Existing Replacement Policies Are Insufficient
- **Belady's OPT Replacement Policy Is Viable for Graph Processing ⇦**
  - **Graph Structure allows Optimal Cache Replacement**
  - **Larger Gains than SOTA Policies**
- P-OPT: A Practical Optimal Cache Replacement Policy

Main Insight: Belady's OPT Is Viable For Graph Processing

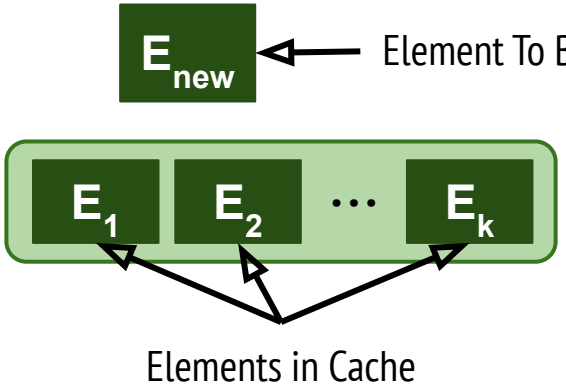
Main Insight: Belady's OPT Is Viable For Graph Processing



Belady's Optimal Replacement Policy: Evict the element which will be accessed furthest in the **future**

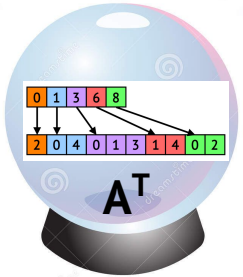
Belady's Replacement Policy is a theoretical upper-bound

Main Insight: Belady's OPT Is Viable For Graph Processing



Belady's Optimal Replacement Policy: Evict the element which will be accessed furthest in the **future**

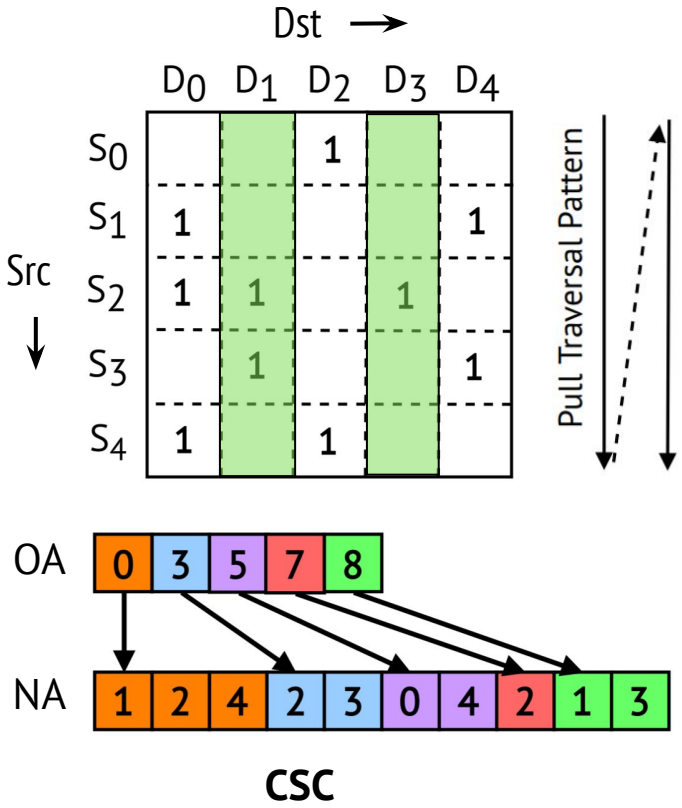
Belady's Replacement Policy is a theoretical upper-bound



Key Observation: The Graph's Transpose Allows Optimal Cache Replacement

Key Graph Application Property That Enables Belady's OPT

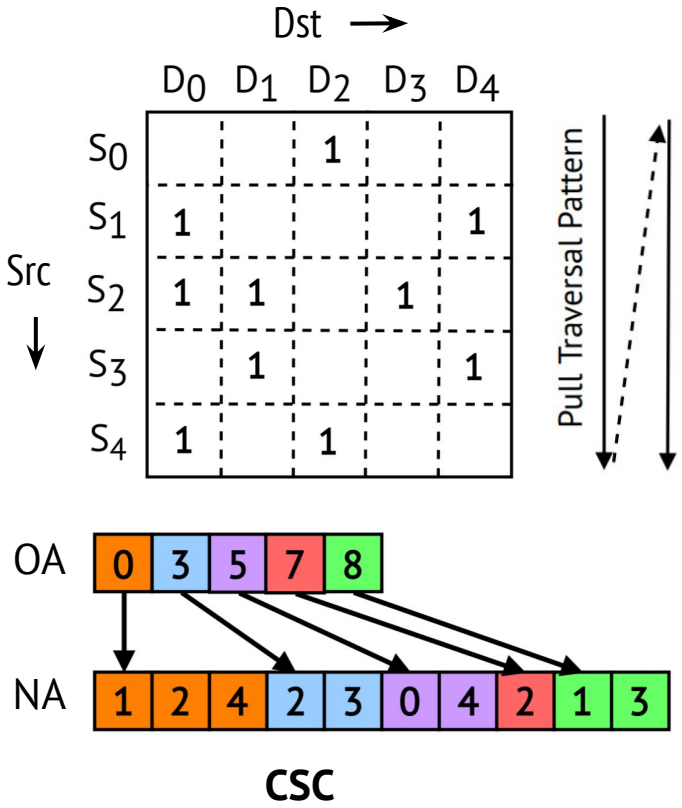
Key Graph Application Property That Enables Belady's OPT



Pull Execution (CSC Traversal)

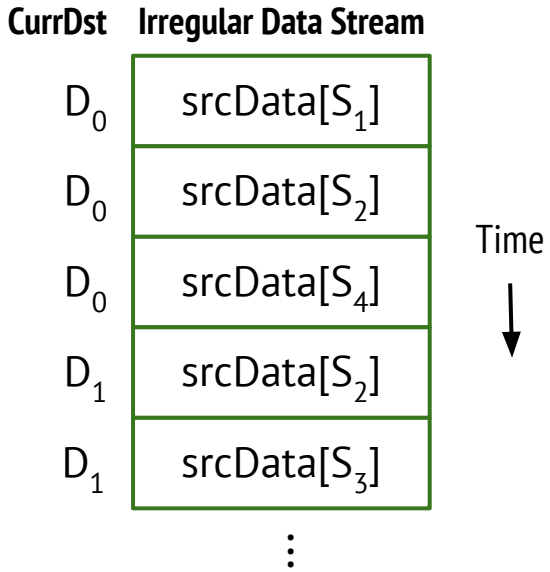
```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

Key Graph Application Property That Enables Belady's OPT

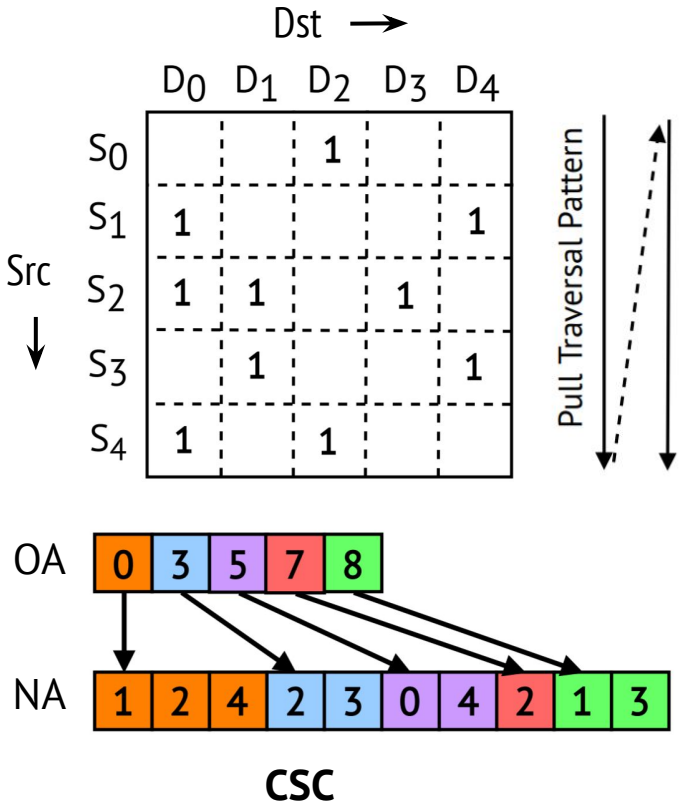


Pull Execution (CSC Traversal)

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



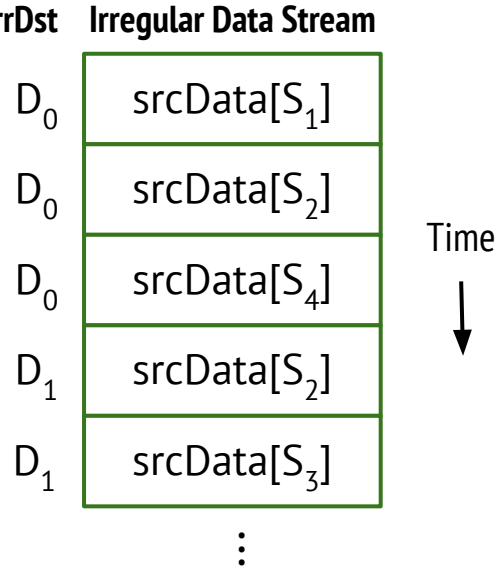
Key Graph Application Property That Enables Belady's OPT



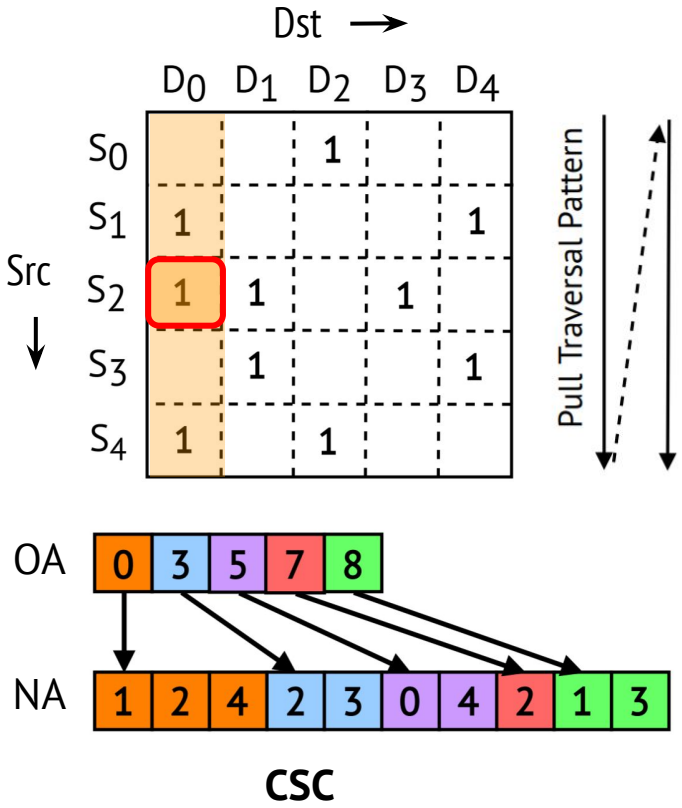
Pull Execution (CSC Traversal)

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

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



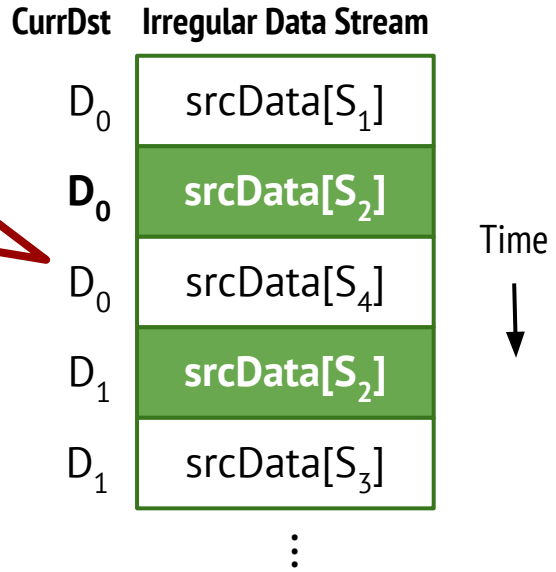
Key Graph Application Property That Enables Belady's OPT



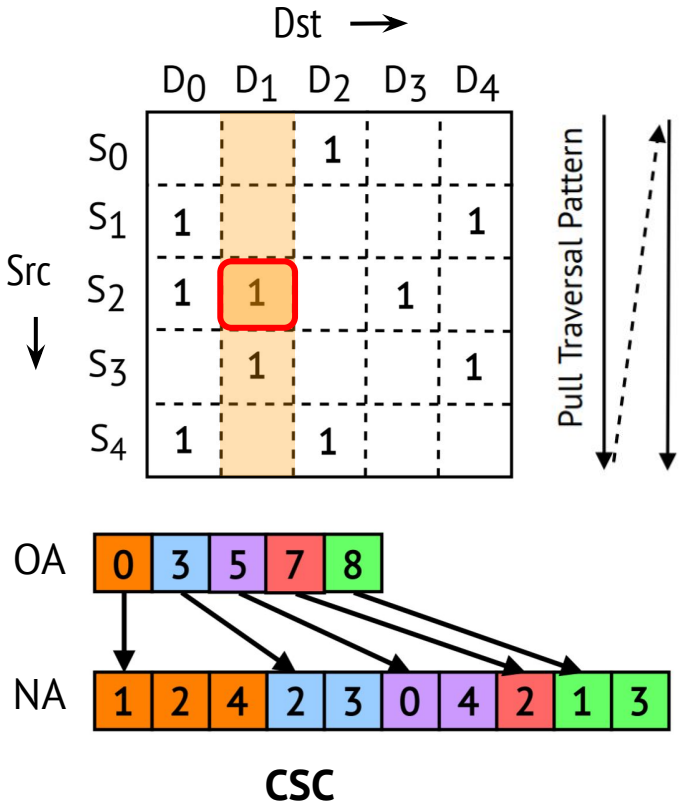
Pull Execution (CSC Traversal)

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

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



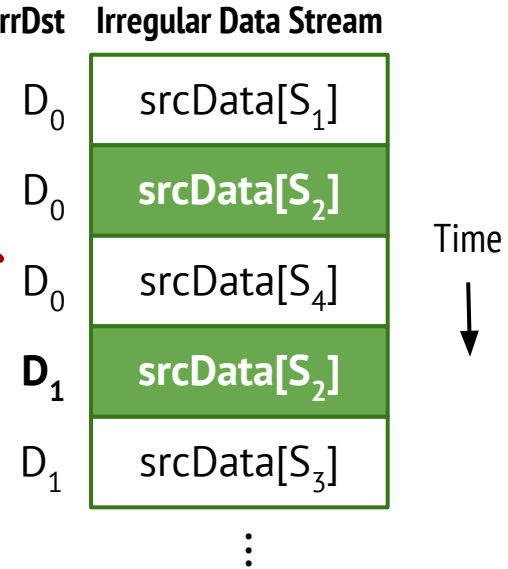
Key Graph Application Property That Enables Belady's OPT



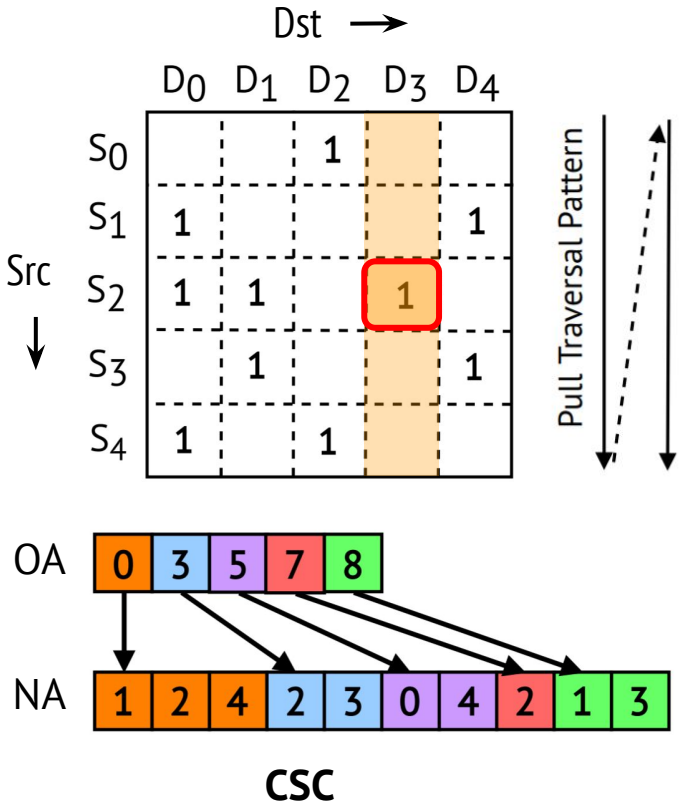
Pull Execution (CSC Traversal)

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

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



Key Graph Application Property That Enables Belady's OPT

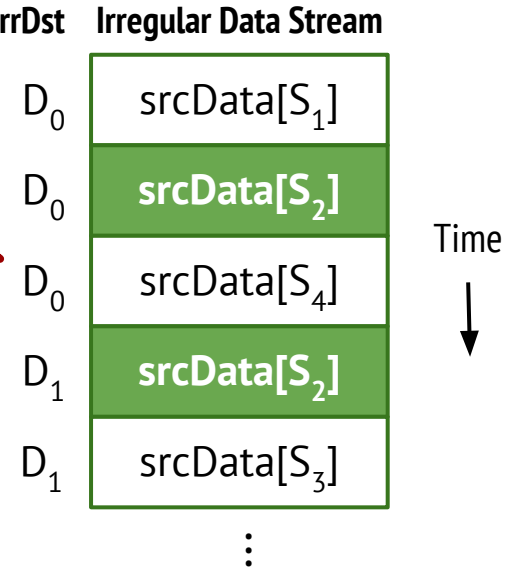


Pull Execution (CSC Traversal)

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

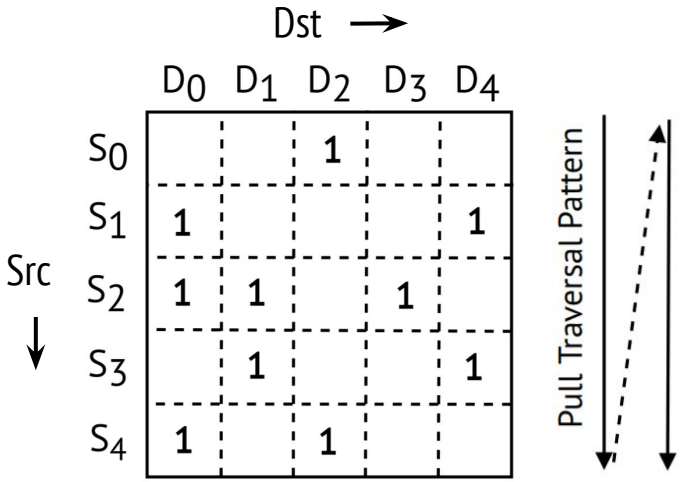
Key Property: Dst-IDs are like timestamps for irregular accesses

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



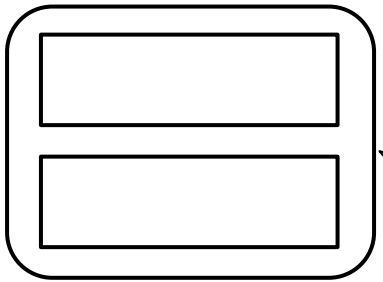
Using The Graph's Transpose For Optimal Replacement

Using The Graph's Transpose For Optimal Replacement



Pull Execution (CSC Traversal)

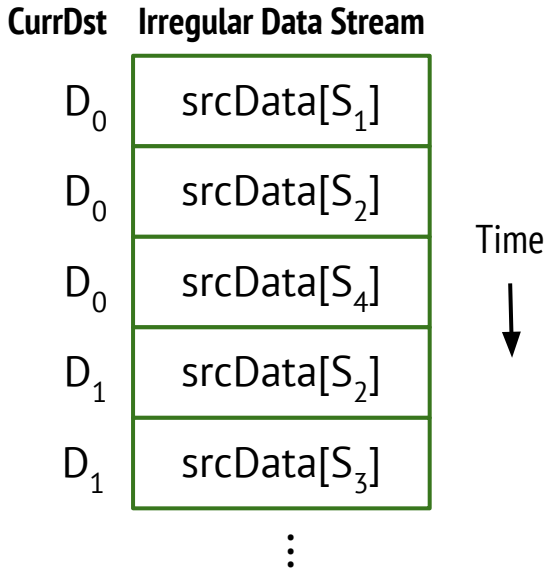
```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```



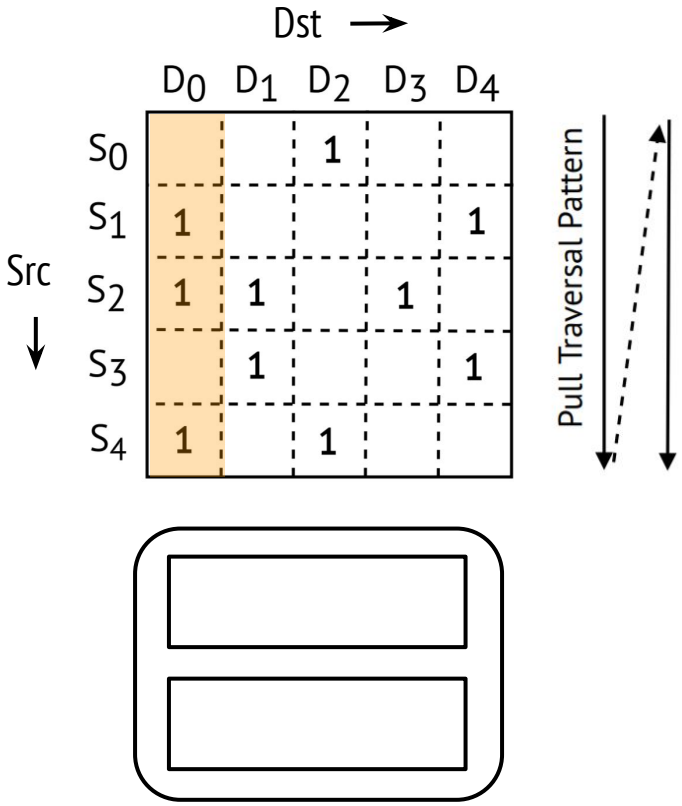
2-way Set-Associative Cache

Assumptions:

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

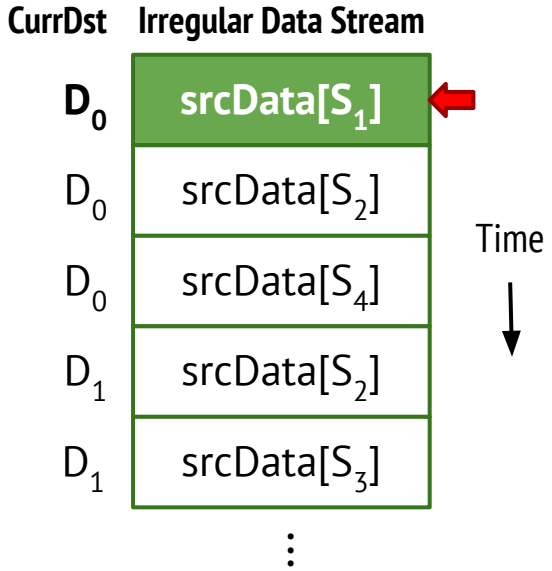


Using The Graph's Transpose For Optimal Replacement

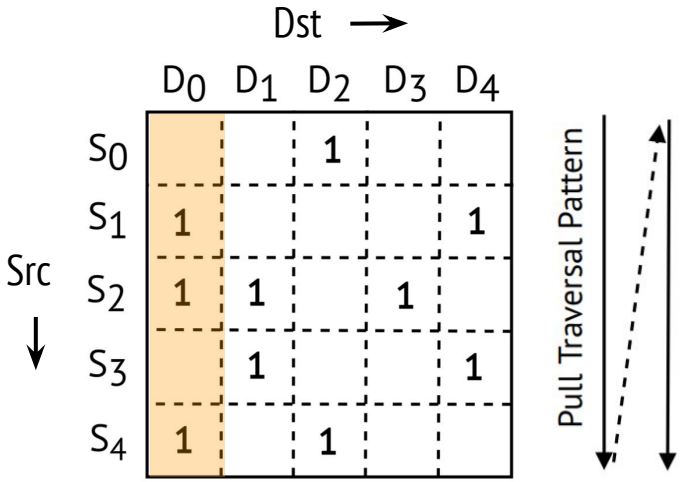


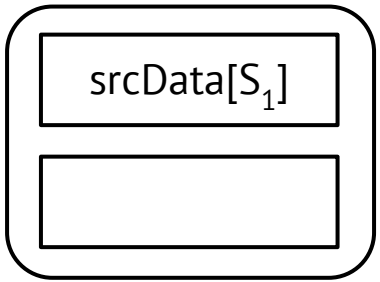
Pull Execution (CSC Traversal)

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



Using The Graph's Transpose For Optimal Replacement

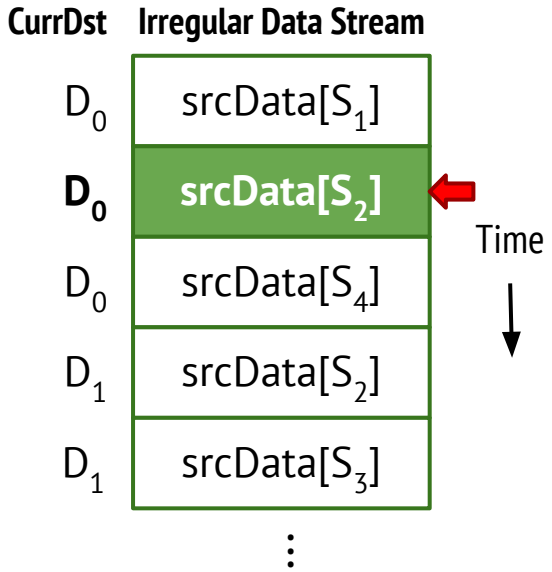




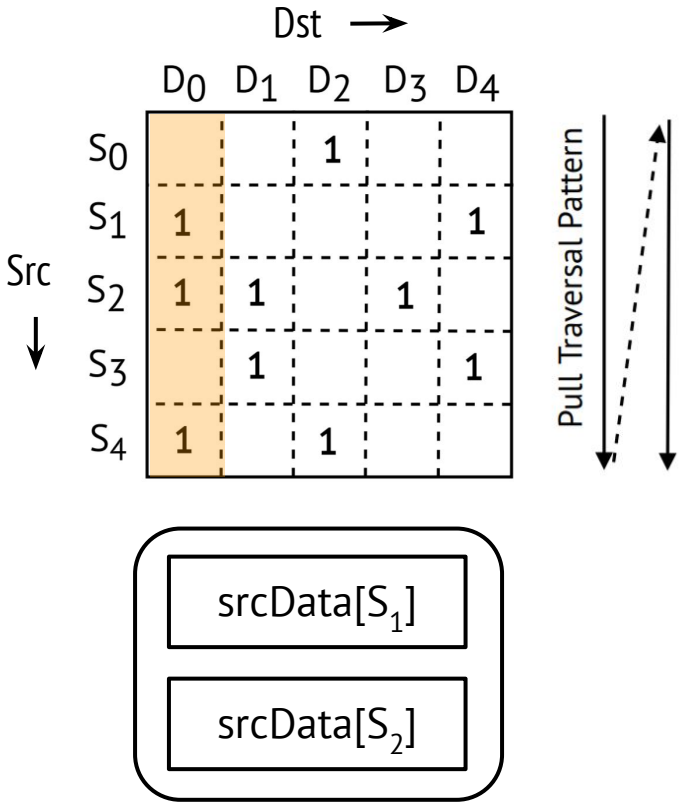
2-way Set-Associative Cache

Pull Execution (CSC Traversal)

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

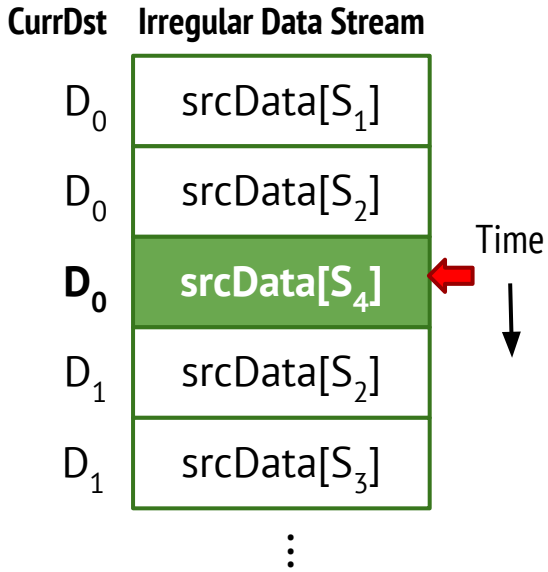


Using The Graph's Transpose For Optimal Replacement

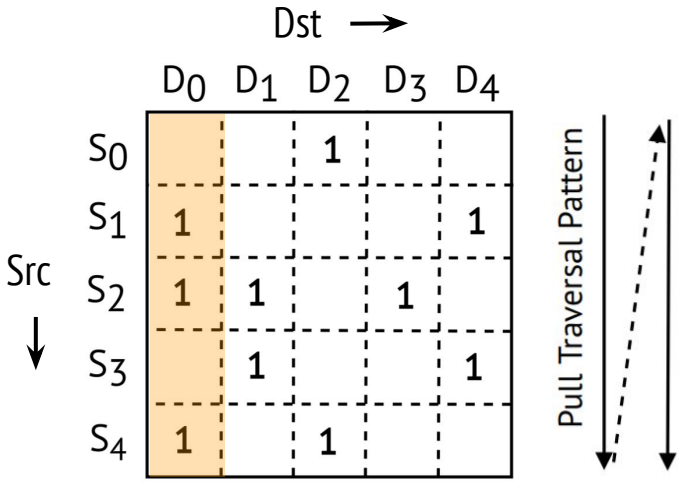


Pull Execution (CSC Traversal)

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

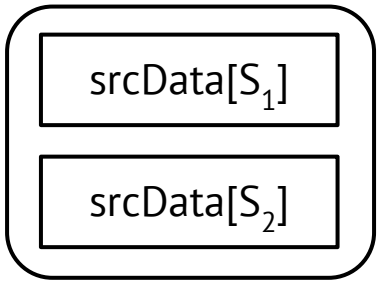


Using The Graph's Transpose For Optimal Replacement



Pull Execution (CSC Traversal)

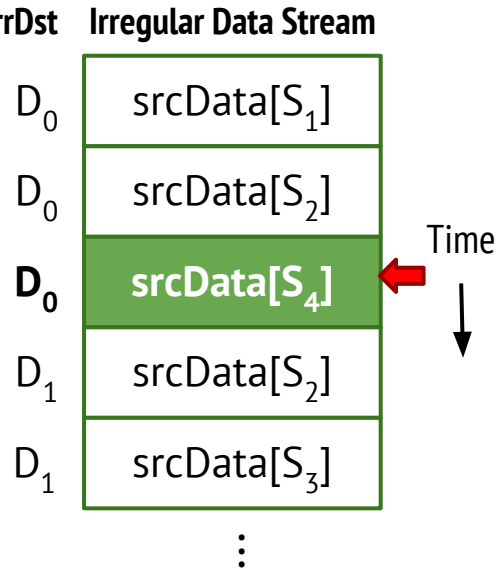
```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```



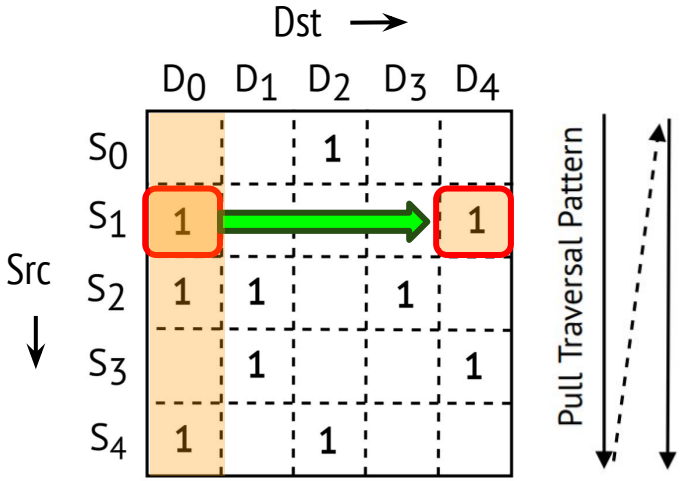
2-way Set-Associative Cache

Which line should we evict?:

- srcData[S_1]
- srcData[S_2]

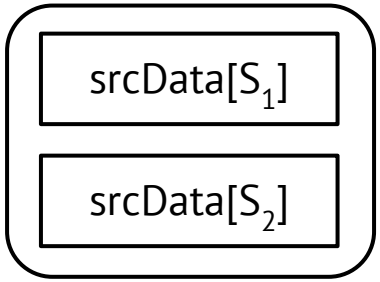


Using The Graph's Transpose For Optimal Replacement



Pull Execution (CSC Traversal)

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

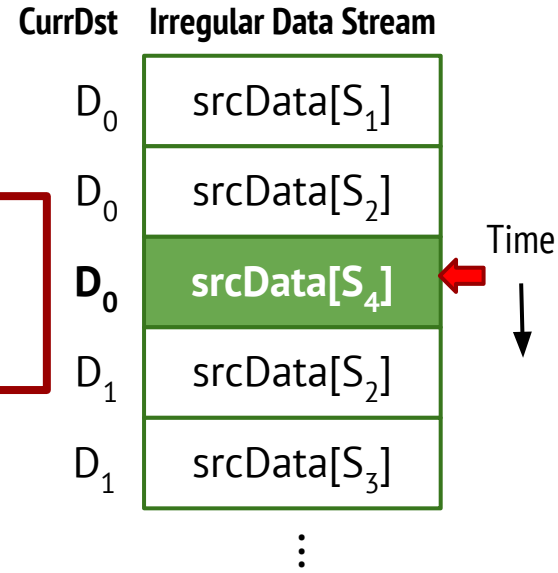


2-way Set-Associative Cache

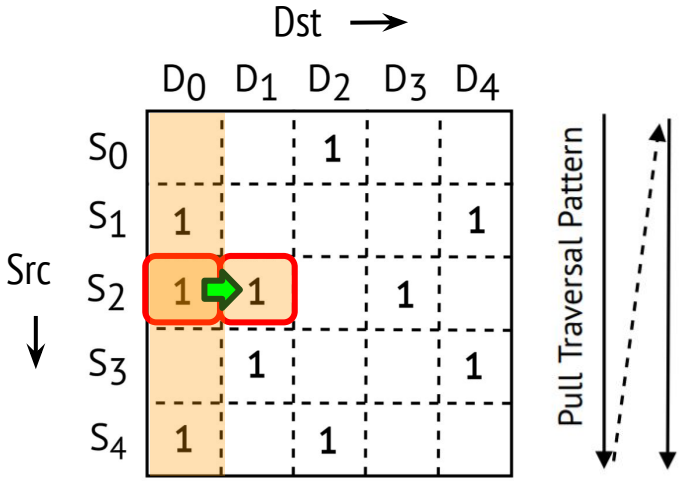
Which line should we evict?:

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

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

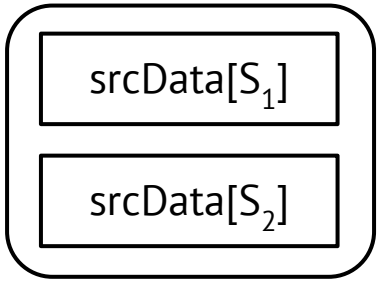


Using The Graph's Transpose For Optimal Replacement



Pull Execution (CSC Traversal)

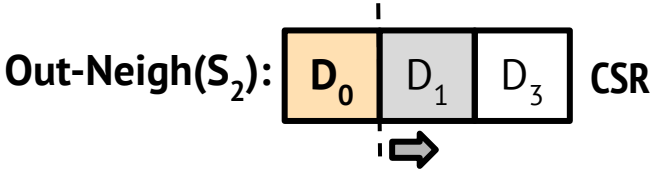
```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

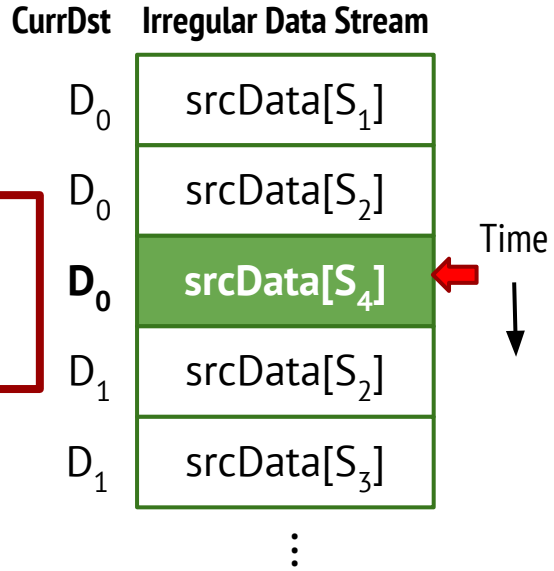


2-way Set-Associative Cache

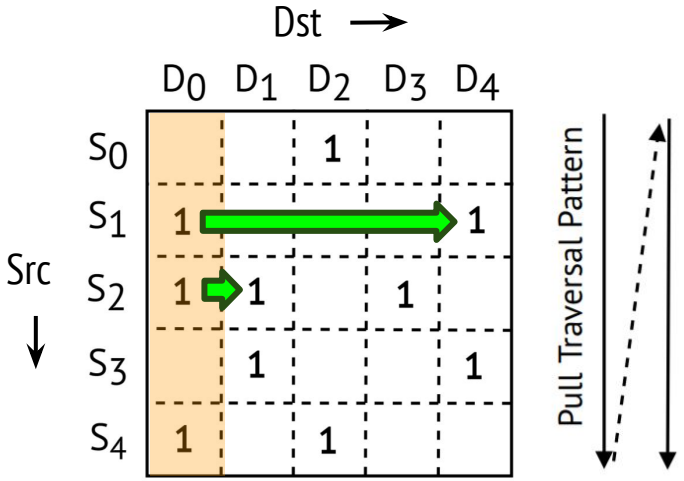
Which line should we evict?:

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



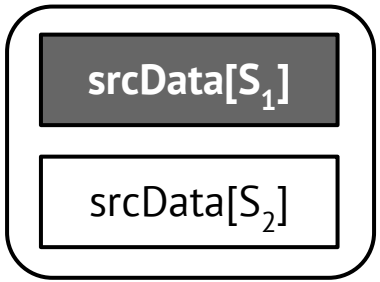


Using The Graph's Transpose For Optimal Replacement



Pull Execution (CSC Traversal)

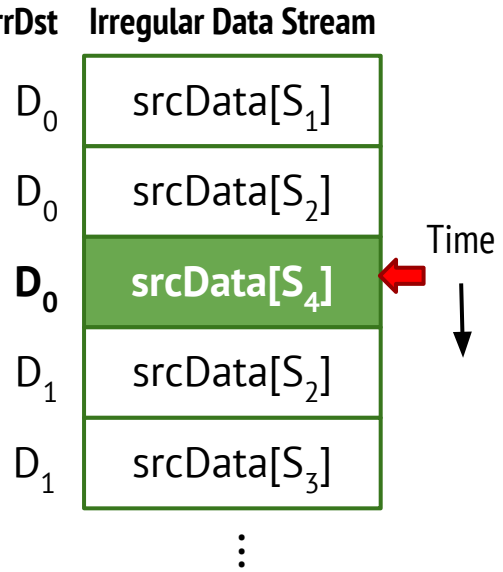
```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```



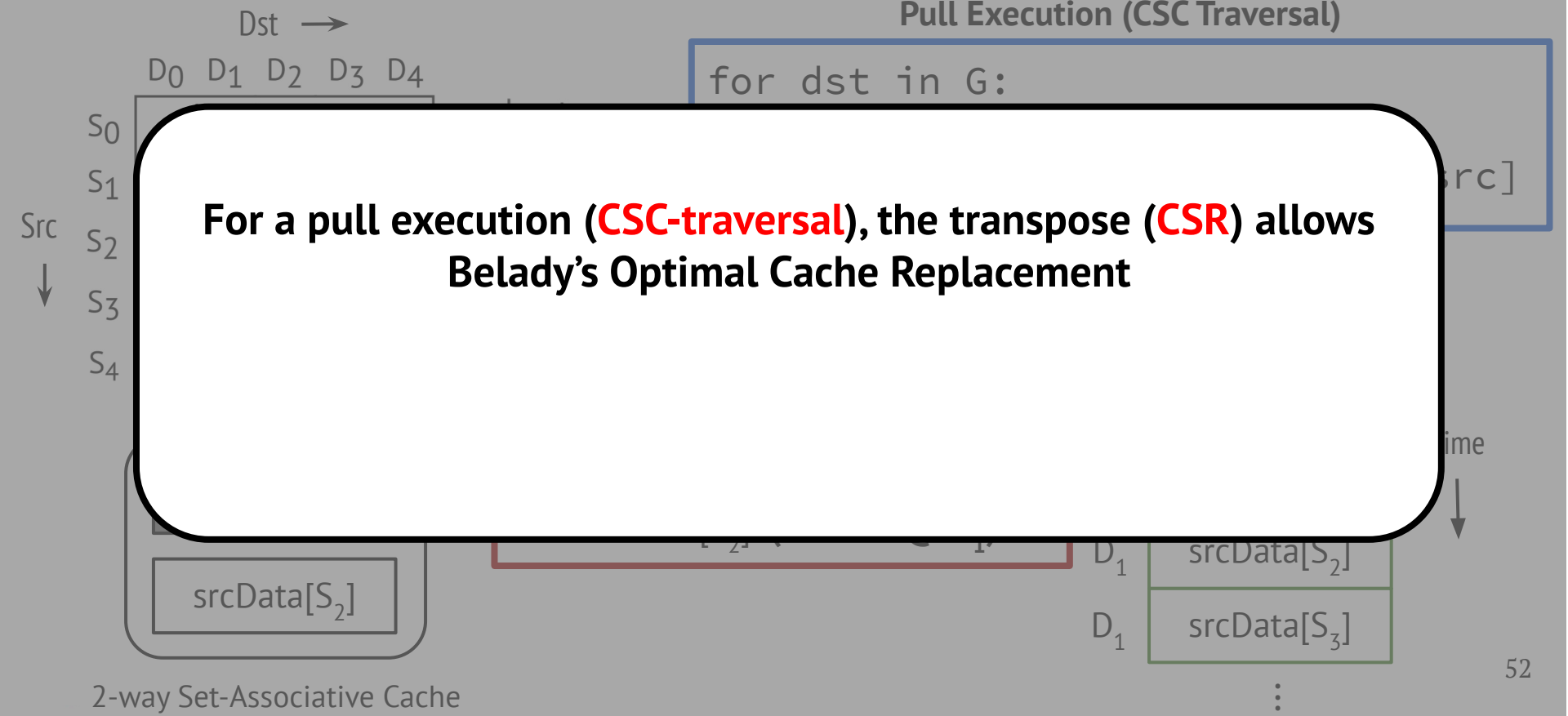
2-way Set-Associative Cache

Which line should we evict?:

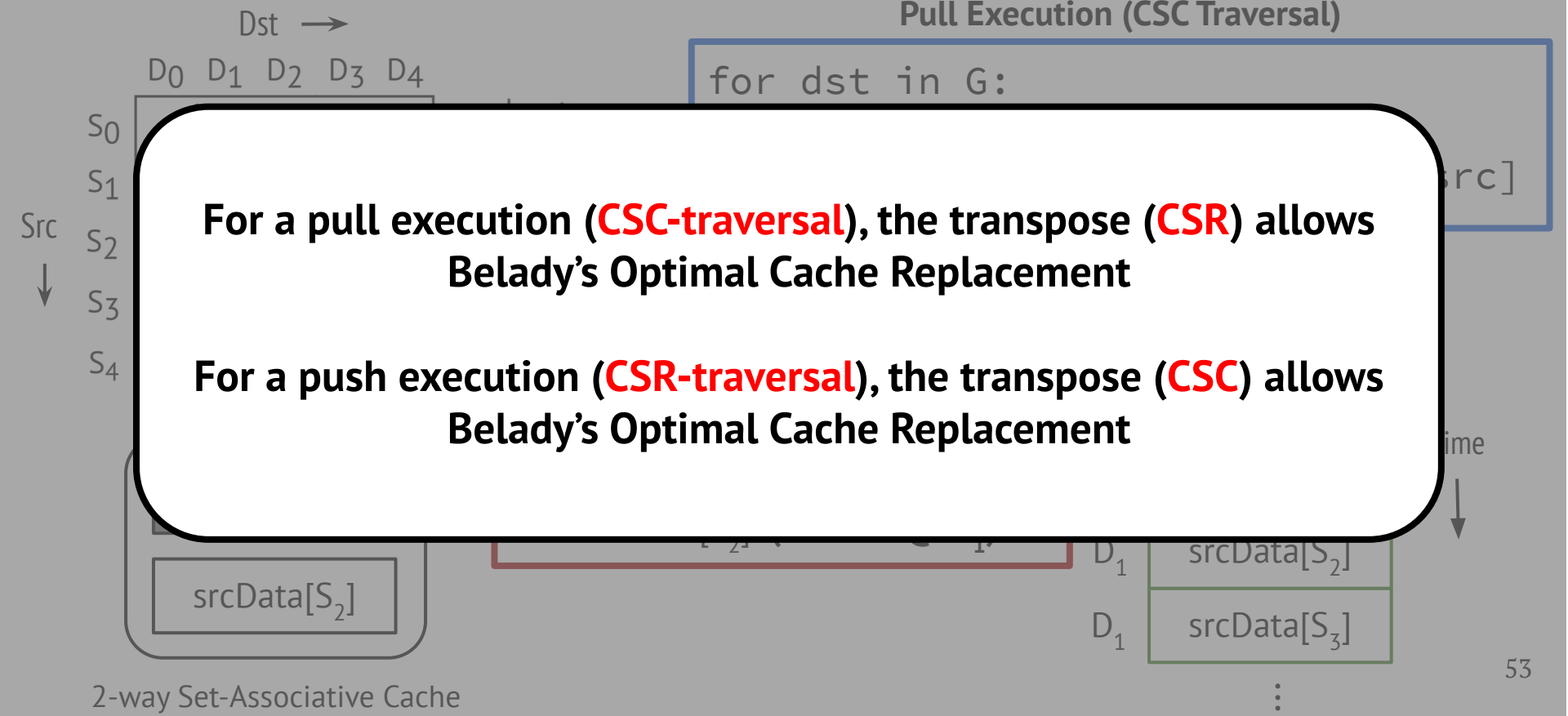
- srcData[S_1] **(nextRef @ D_4)** ✔
- srcData[S_2] **(nextRef @ D_1)**



Using The Graph's Transpose For Optimal Replacement



Using The Graph's Transpose For Optimal Replacement

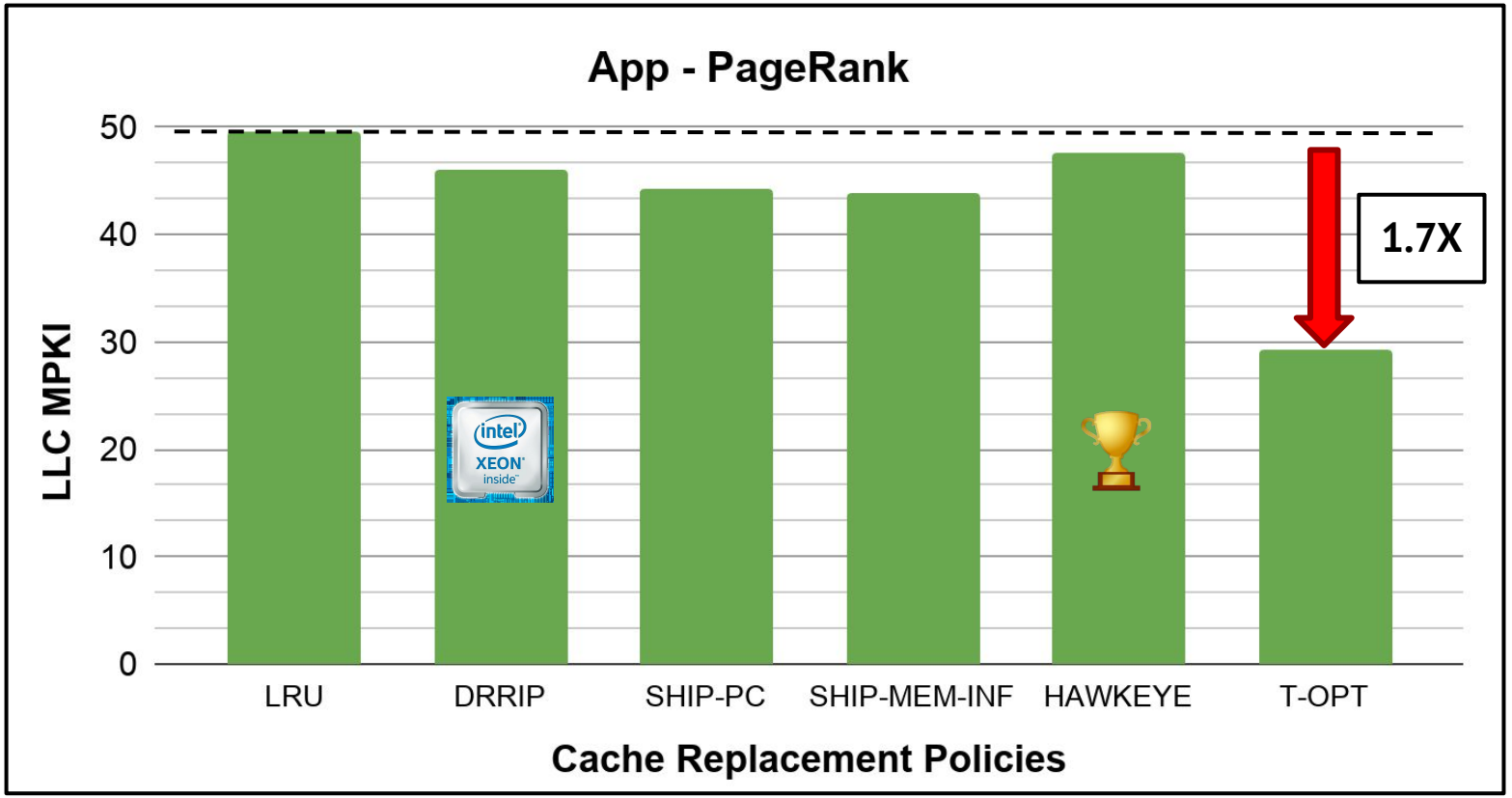


For a pull execution (CSC-traversal), the transpose (CSR) allows Belady's Optimal Cache Replacement

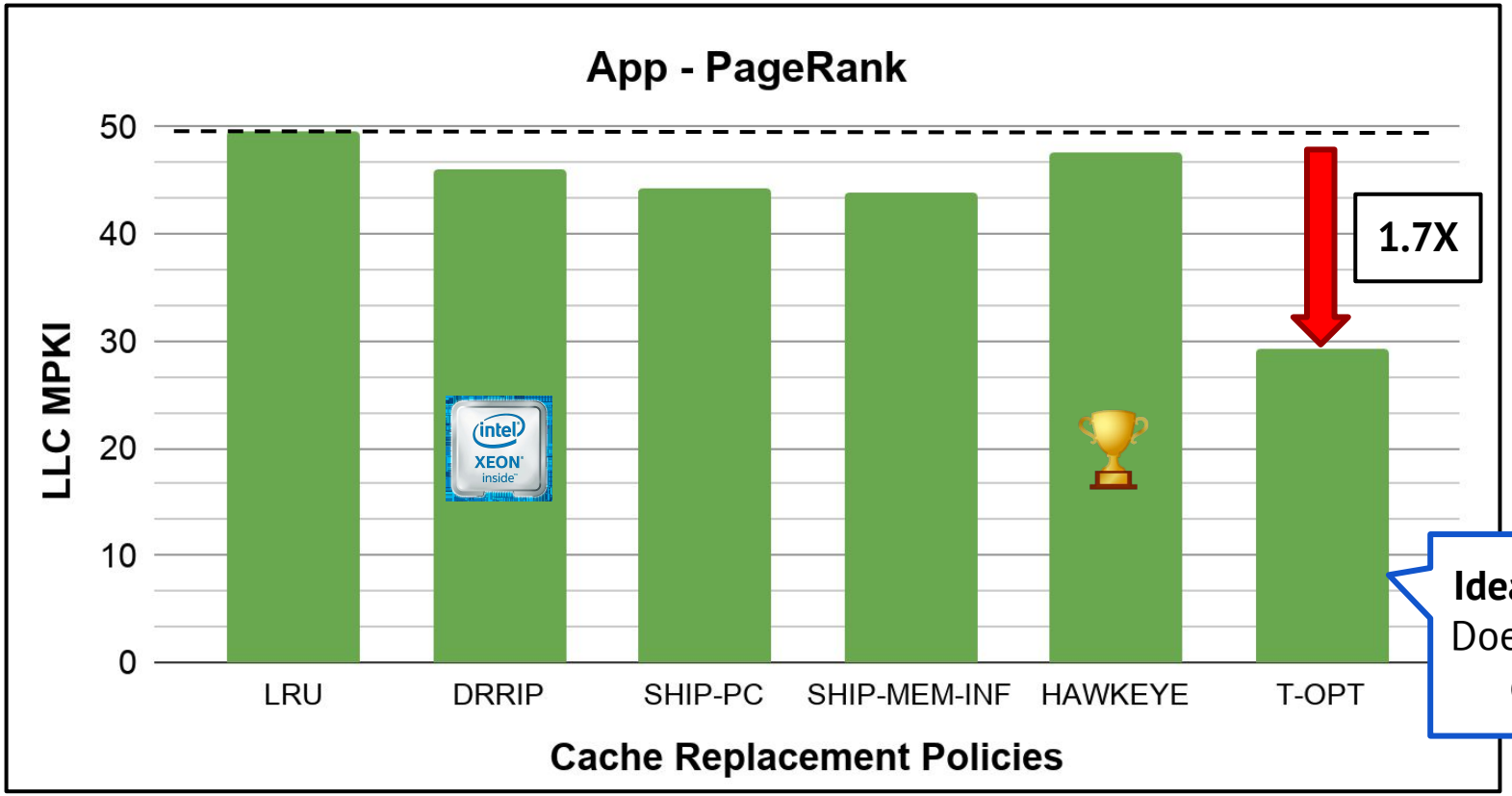
For a push execution (CSR-traversal), the transpose (CSC) allows Belady's Optimal Cache Replacement

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

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

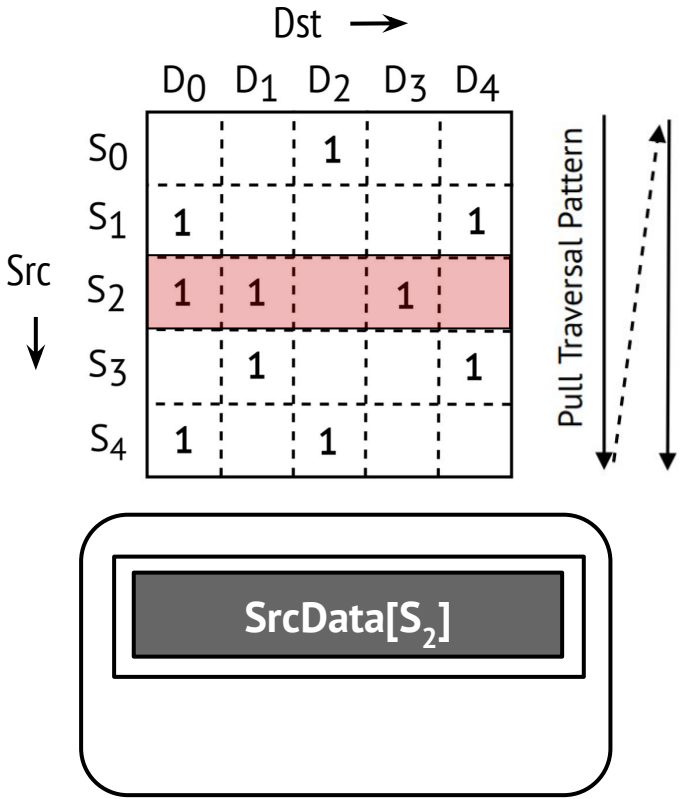


Transpose-based OPT (T-OPT) Provides Large Gains



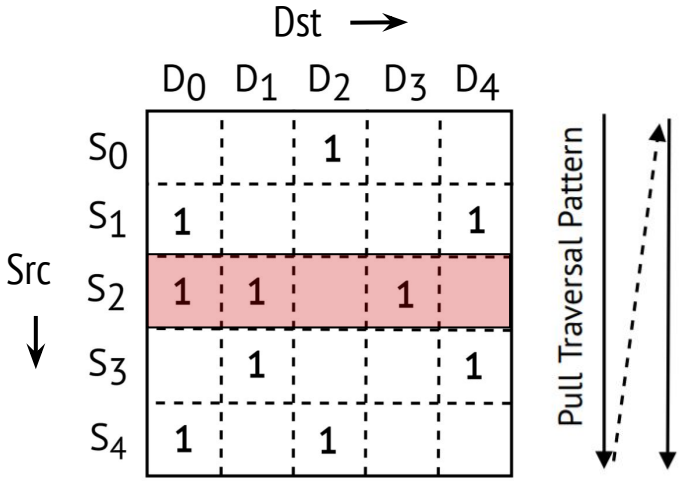
Idealized Result: Does not include overheads

Transpose-based OPT Replacement Incurs Overheads



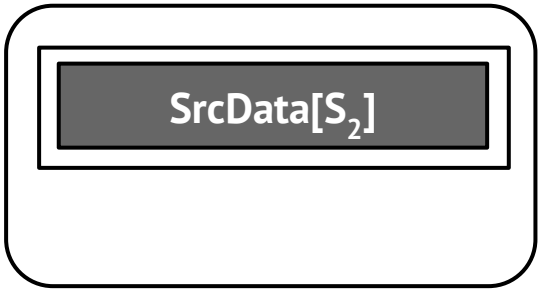
Finding Next References Using The Transpose

Transpose-based OPT Replacement Incurs Overheads

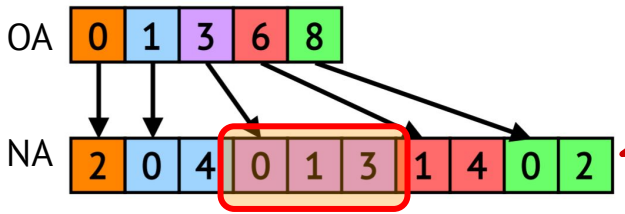


Finding Next References Using The Transpose

DRAM Access overhead



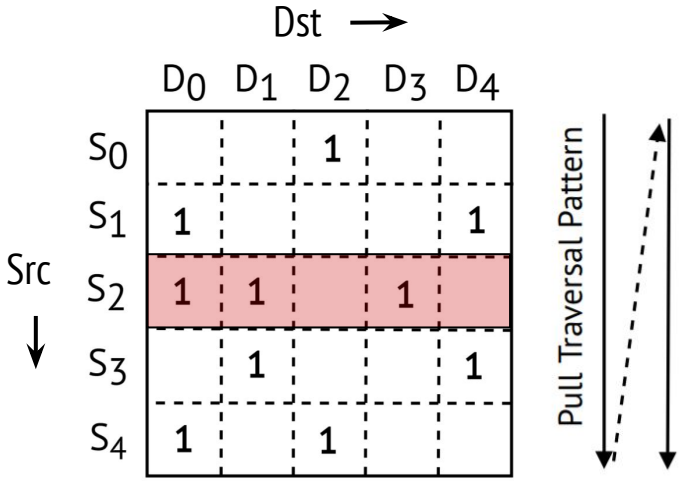
Set-Associative Cache



CSR *(Transpose)*

Transpose accesses are irregular

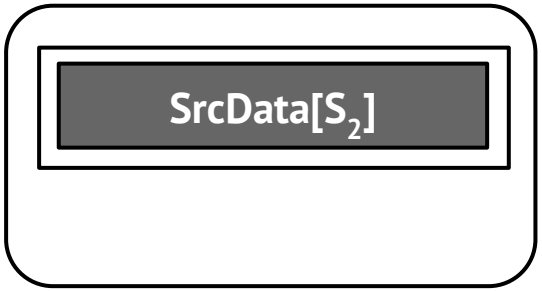
Transpose-based OPT Replacement Incurs Overheads



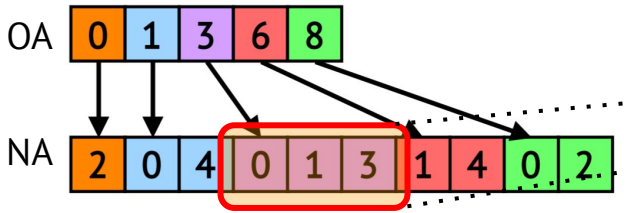
Finding Next References Using The Transpose

DRAM Access overhead

Runtime Traversal overhead



Set-Associative Cache

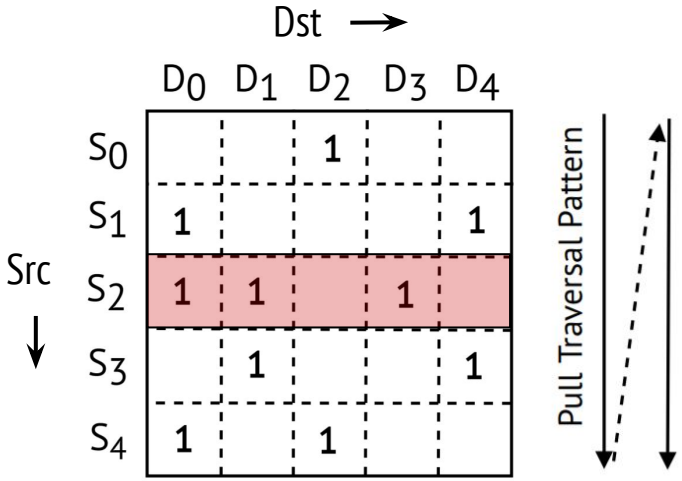


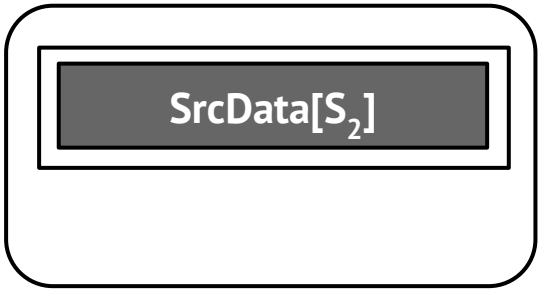
CSR *(Transpose)*

Out-Neigh(S_2): D_0 D_1 D_3

Need to scan neighbors

Transpose-based OPT Replacement Incurs Overheads





Set-Associative Cache

Finding Next References Using The Transpose

DRAM Access overhead

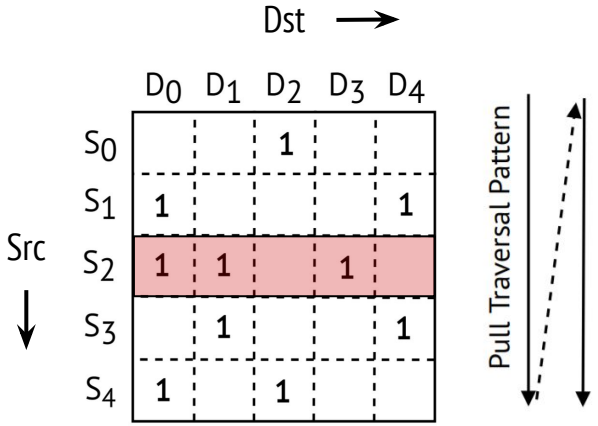
Runtime Traversal overhead

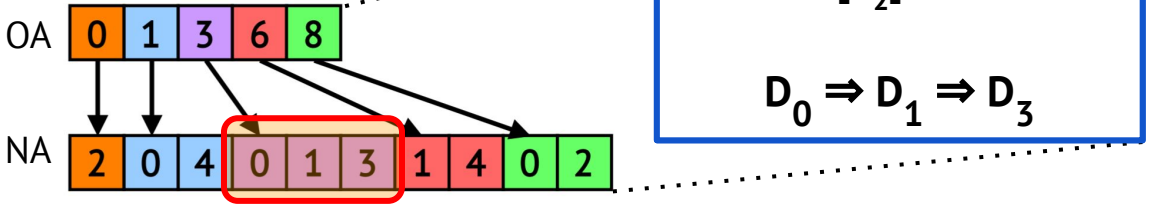
Question: How do we retrieve the next reference information from the graph's transpose **without these overheads?**

Outline

- Primary Bottleneck of Graph Analytics ⇒ Poor Cache Locality ✓
- Reasons for Poor Cache Locality ✓
  - Irregular Memory Accesses
  - Existing Replacement Policies Are Insufficient
- Belady's OPT Replacement Policy Is Viable for Graph Processing ✓
  - Graph Structure allows Optimal Cache Replacement
  - Larger Gains than SOTA Policies
- **P-OPT: A Practical Optimal Cache Replacement Policy ⇦**
  - **Reducing Overheads using Quantization**
  - **P-OPT achieves close to ideal performance**

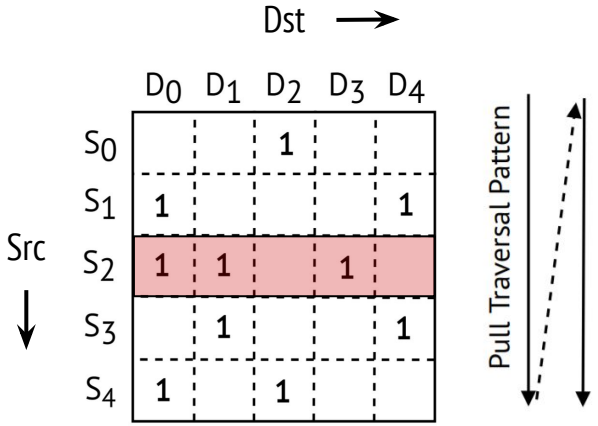
Main Technique: Use Quantization To Compress The Transpose



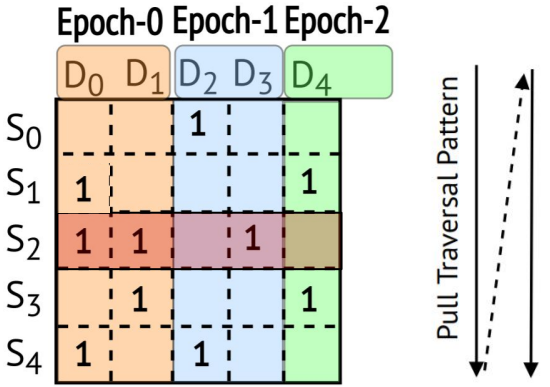


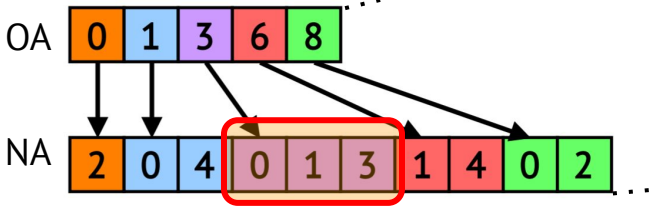
CSR
(Transpose)

Main Technique: Use Quantization To Compress The Transpose



Divide execution into coarse-grained epochs



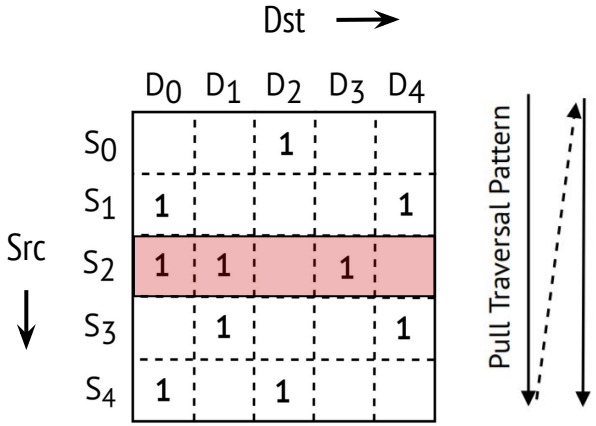


CSR
(Transpose)

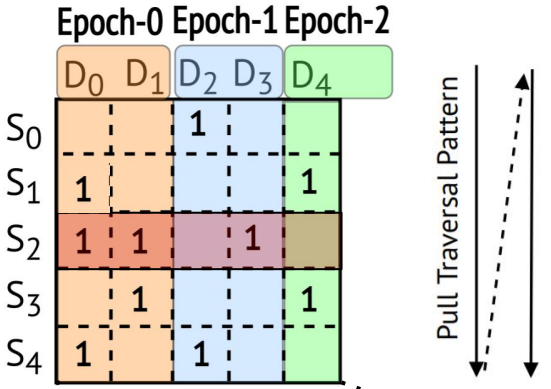
srcData[S_2] Next Refs:

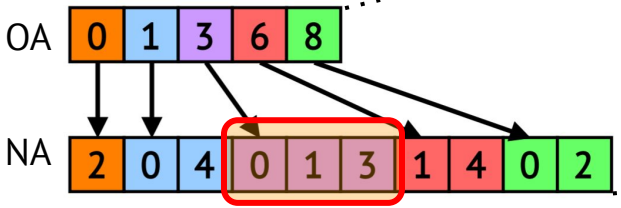
$D_0 \Rightarrow D_1 \Rightarrow D_3$

Main Technique: Use Quantization To Compress The Transpose



Divide execution into coarse-grained epochs





CSR
(Transpose)

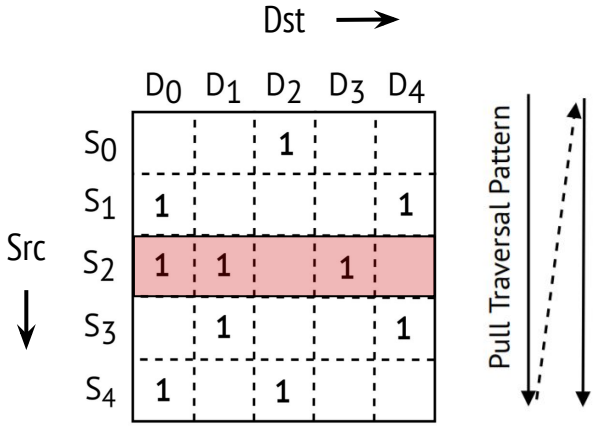
srcData[S_2] Next Refs:

$D_0 \Rightarrow D_1 \Rightarrow D_3$

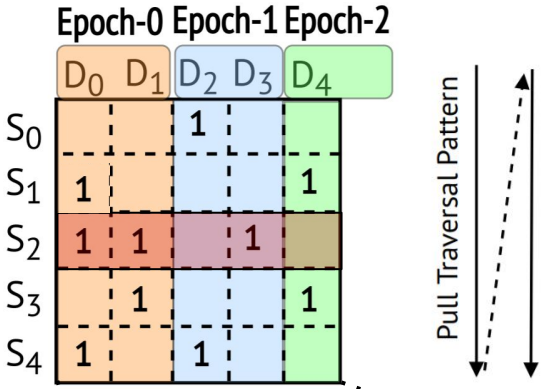
srcData[S_2] Next Refs:

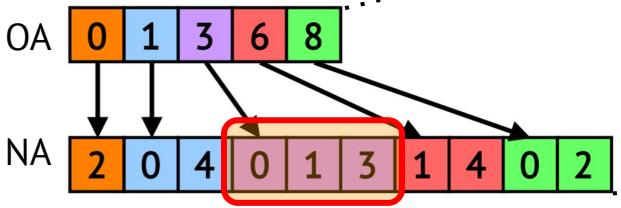
$E_0 \Rightarrow E_1$

Main Technique: Use Quantization To Compress The Transpose



Divide execution into coarse-grained epochs





CSR
(Transpose)

srcData[S_2] Next Refs:

$D_0 \Rightarrow D_1 \Rightarrow D_3$

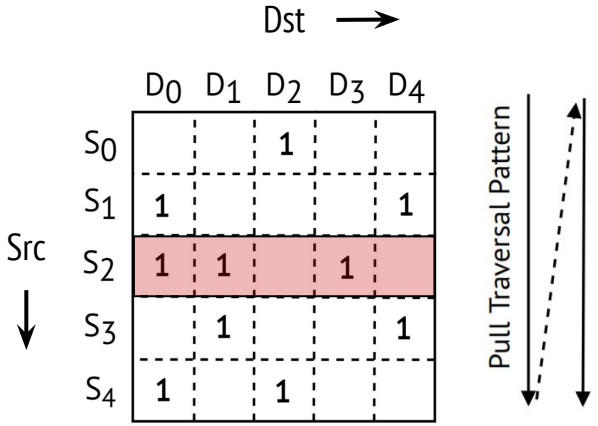
srcData[S_2] Next Refs:

$E_0 \Rightarrow E_1$

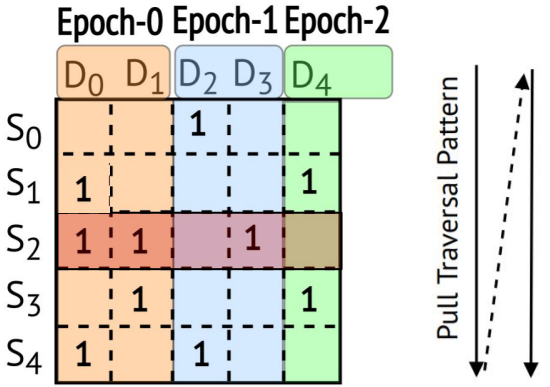
No. of epochs ⬇️ ⇒ Next Ref Quantization ⬆️

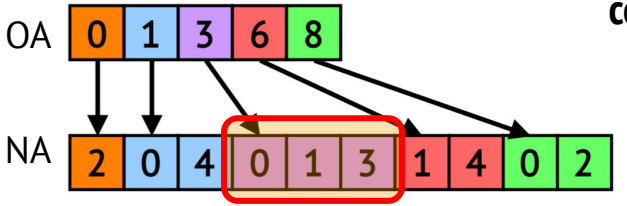
(accuracy vs metadata size trade-off)

Main Technique: Use Quantization To Compress The Transpose



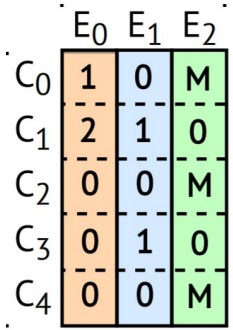
Divide execution into coarse-grained epochs





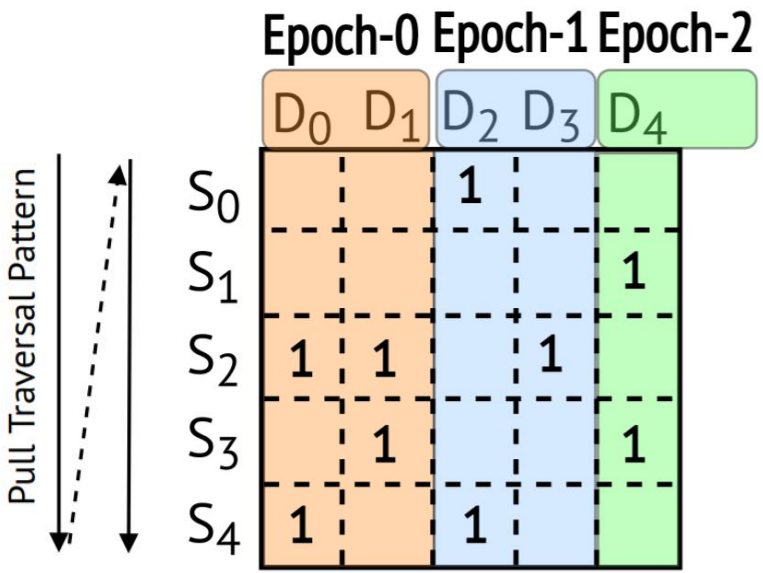
CSR
(Transpose)

Quantization enables compression of transpose data

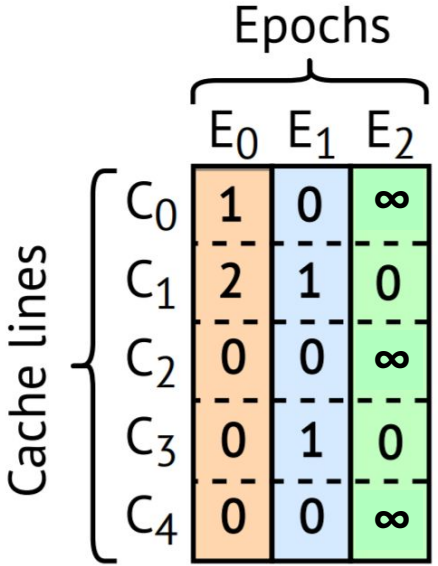


Rereference Matrix
(Quantized Transpose)

Rereference Matrix Eliminates T-OPT's overheads



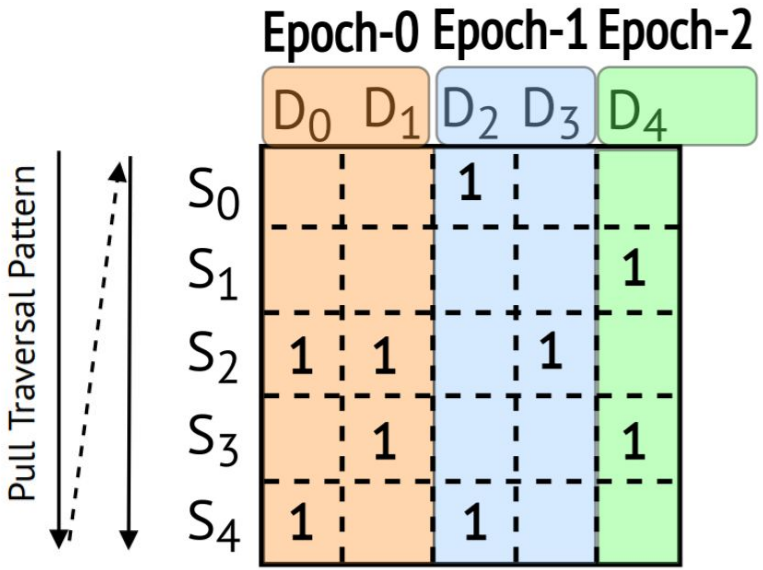
Epoch Execution Model



Rereference Matrix

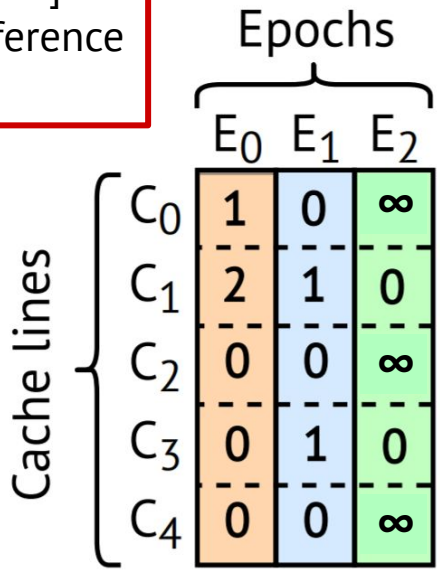
Constructed using the graph's transpose

Rereference Matrix Eliminates T-OPT's overheads



Epoch Execution Model

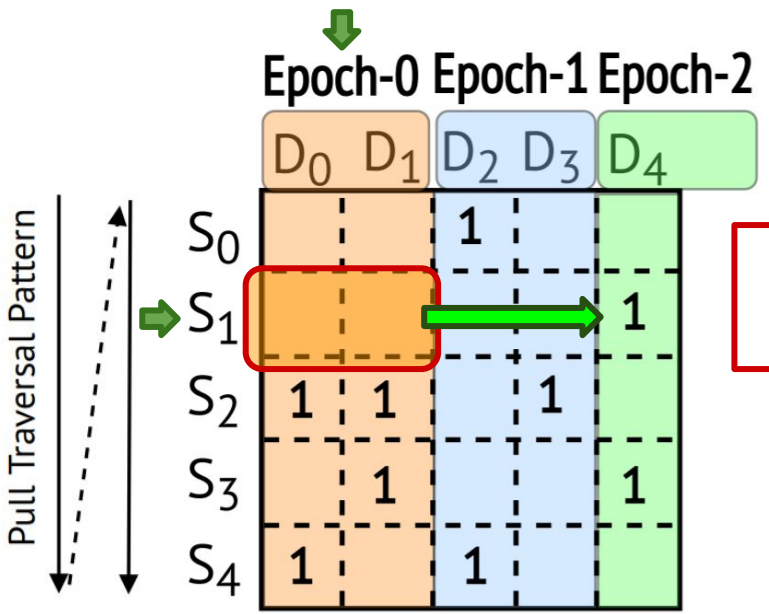
RM[Cacheline][Epoch] = Distance to Next Reference (in epochs)



Rereference Matrix

Constructed using the graph's transpose

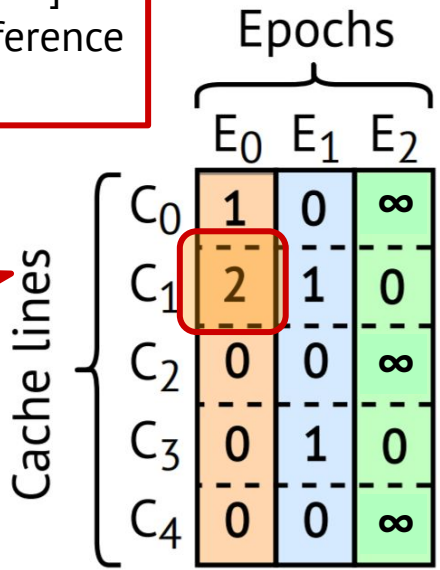
Rereference Matrix Eliminates T-OPT's overheads



Epoch Execution Model

RM[Cacheline][Epoch] = Distance to Next Reference (in epochs)

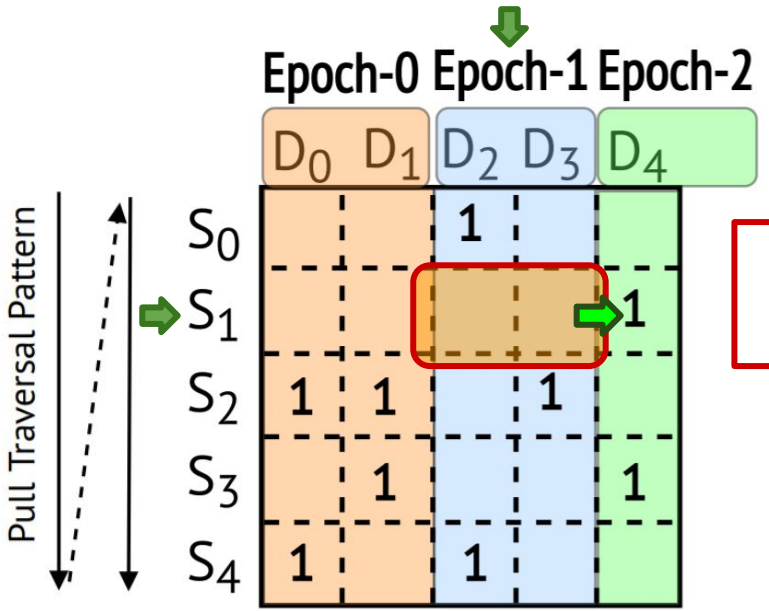
RM[C_1][E_0] = E_2 - E_0



Rereference Matrix

Constructed using the graph's transpose

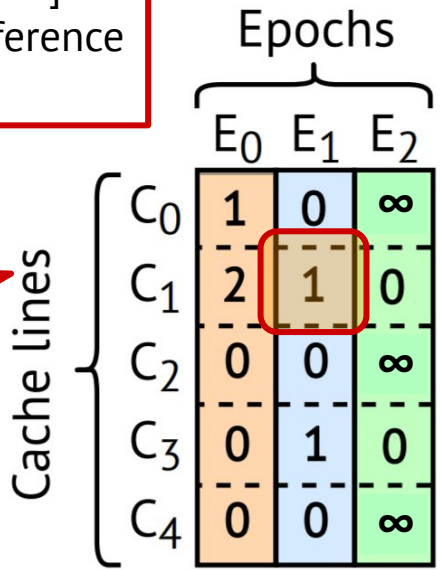
Rereference Matrix Eliminates T-OPT's overheads



Epoch Execution Model

RM[Cacheline][Epoch] = Distance to Next Reference (in epochs)

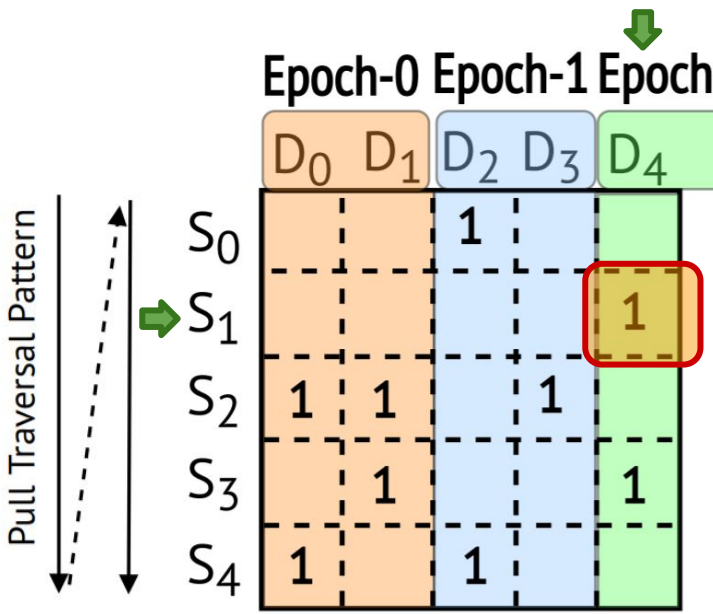
$RM[C_1][E_1] = E_2 - E_1$



Rereference Matrix

Constructed using the graph's transpose

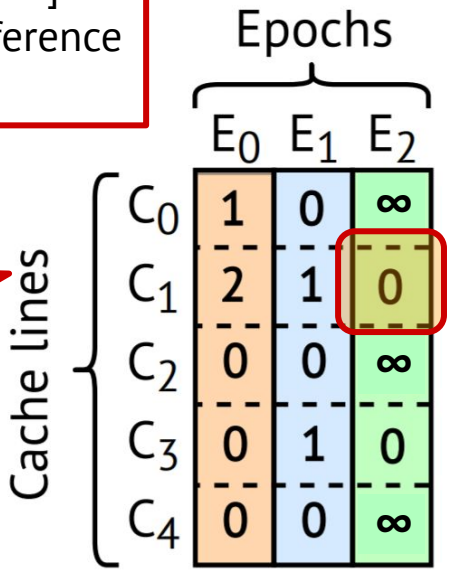
Rereference Matrix Eliminates T-OPT's overheads



Epoch Execution Model

RM[Cacheline][Epoch] = Distance to Next Reference (in epochs)

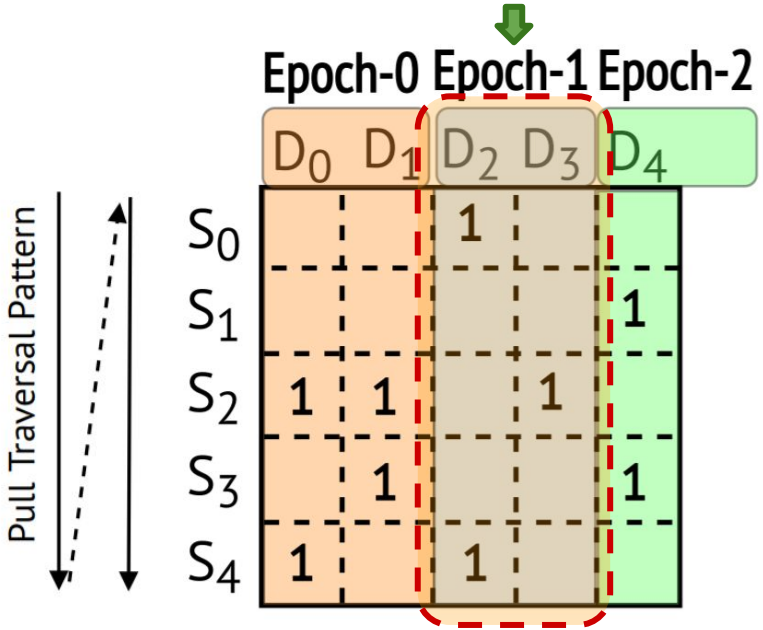
$RM[C_1][E_2] = E_2 - E_2$



Rereference Matrix

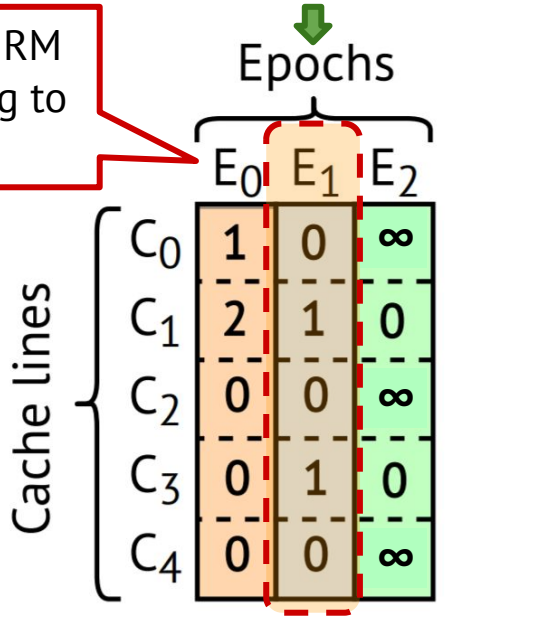
Constructed using the graph's transpose

Rereference Matrix Eliminates T-OPT's overheads



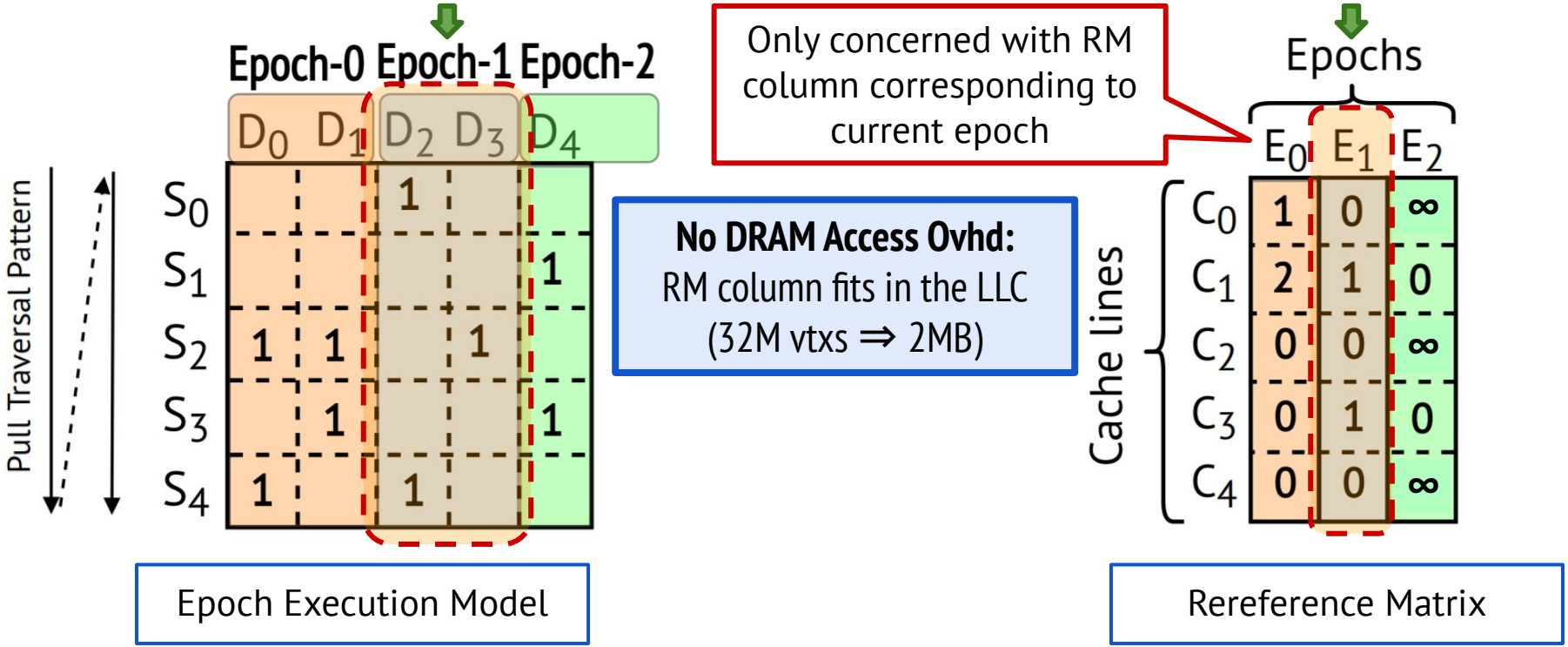
Epoch Execution Model

Only concerned with RM column corresponding to current epoch

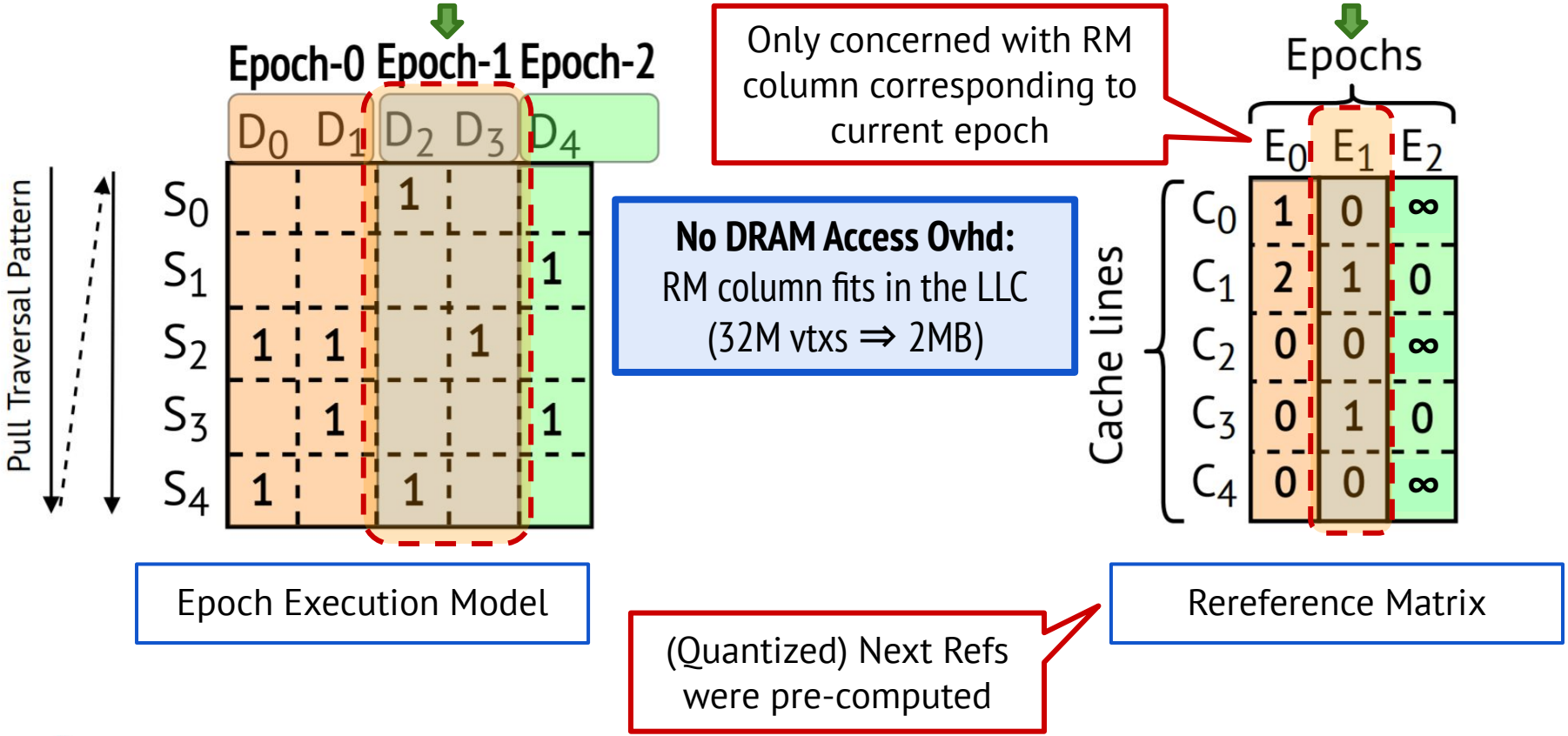


Rereference Matrix

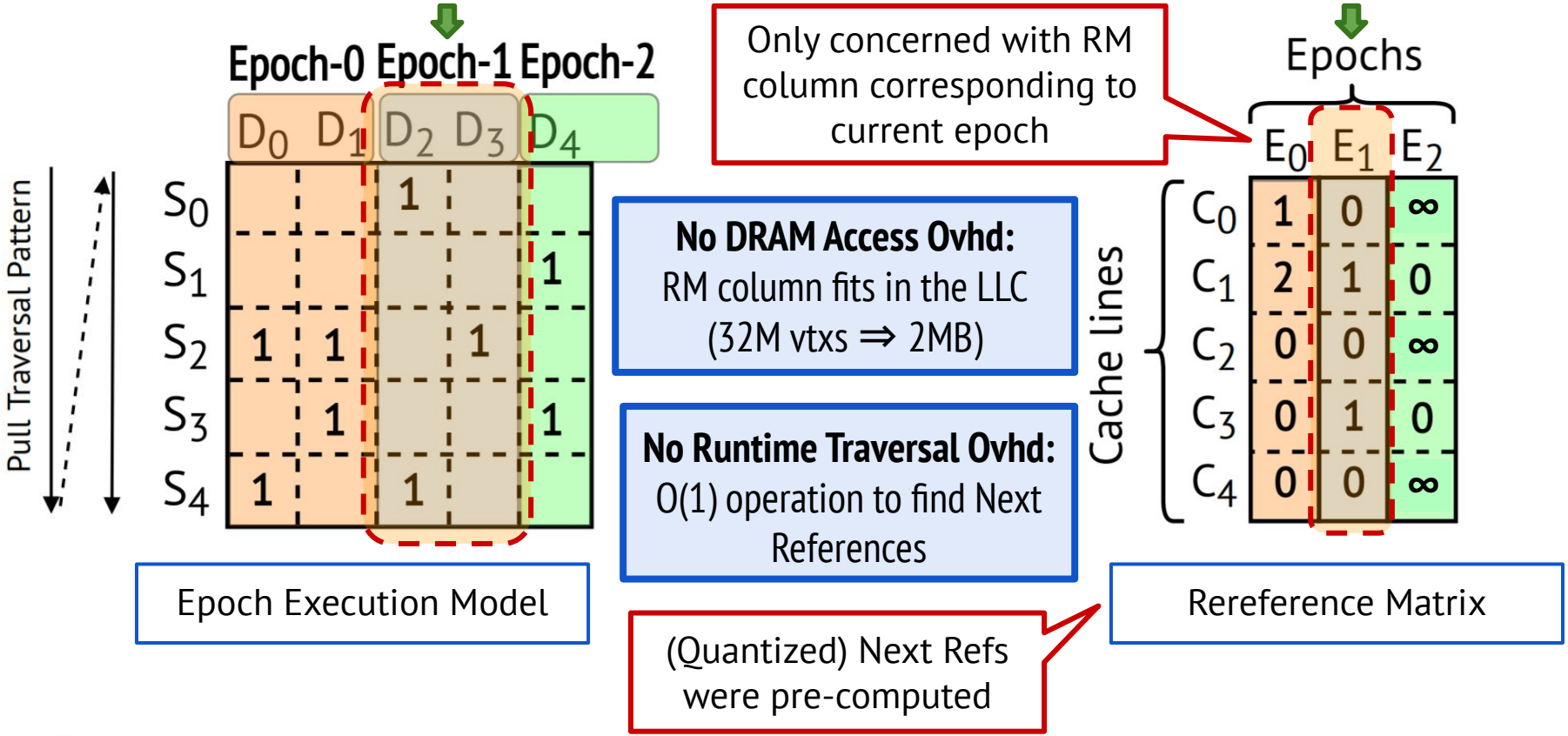
Rereference Matrix Eliminates T-OPT's overheads



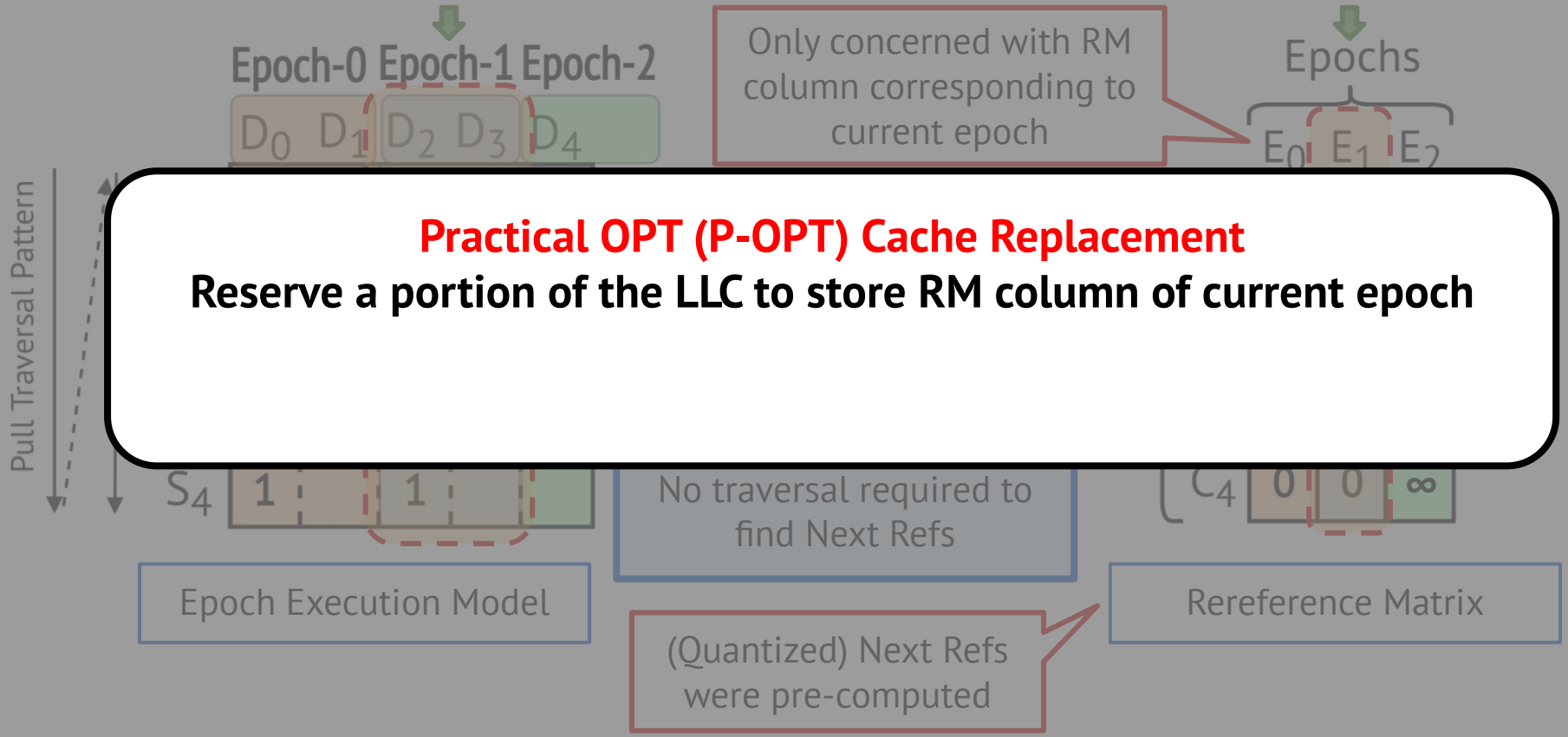
Rereference Matrix Eliminates T-OPT's overheads



Rereference Matrix Eliminates T-OPT's overheads



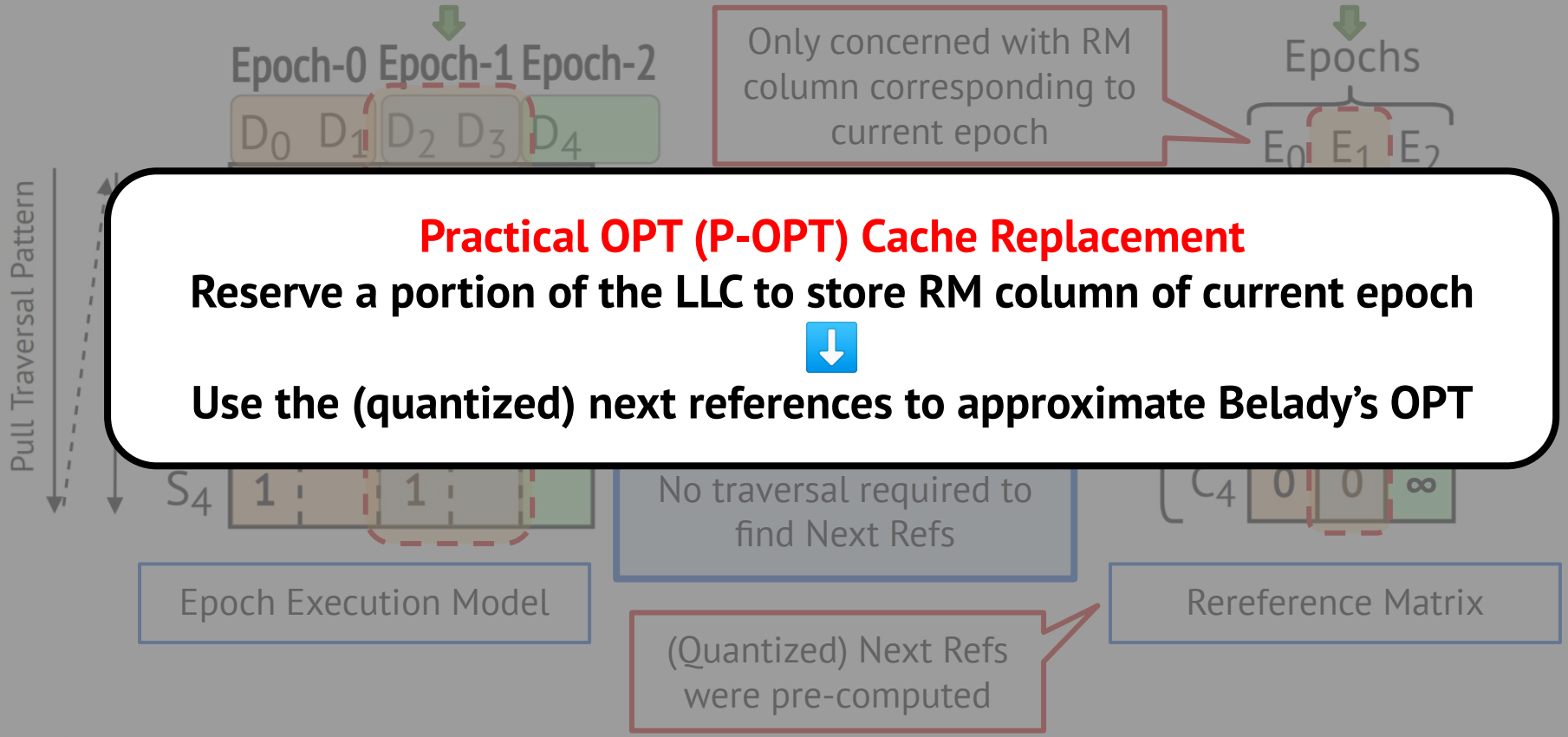
Rereference Matrix Eliminates T-OPT's overheads



Practical OPT (P-OPT) Cache Replacement

Reserve a portion of the LLC to store RM column of current epoch

Rereference Matrix Eliminates T-OPT's overheads



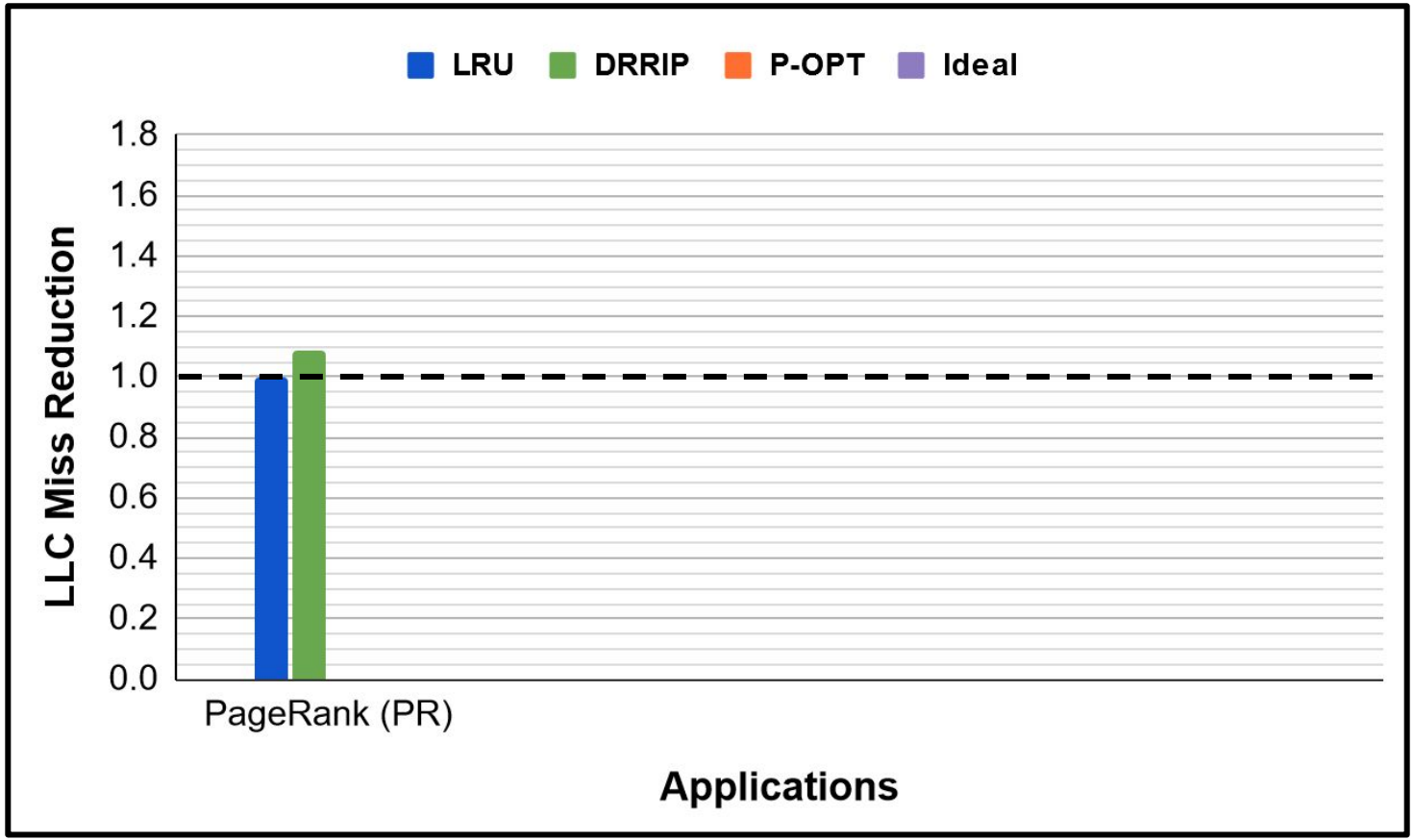
Practical OPT (P-OPT) Cache Replacement

Reserve a portion of the LLC to store RM column of current epoch

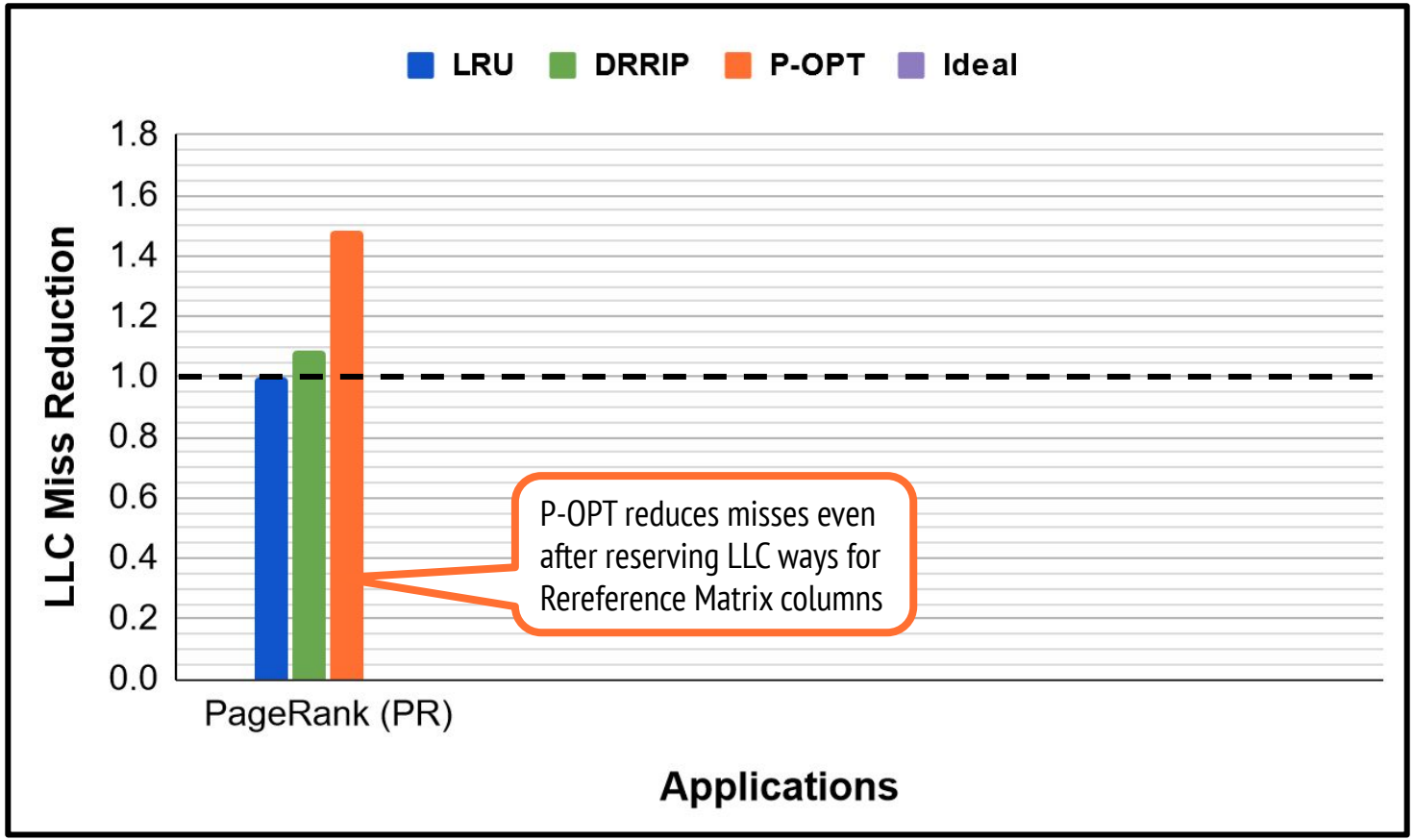
⬇️

Use the (quantized) next references to approximate Belady's OPT

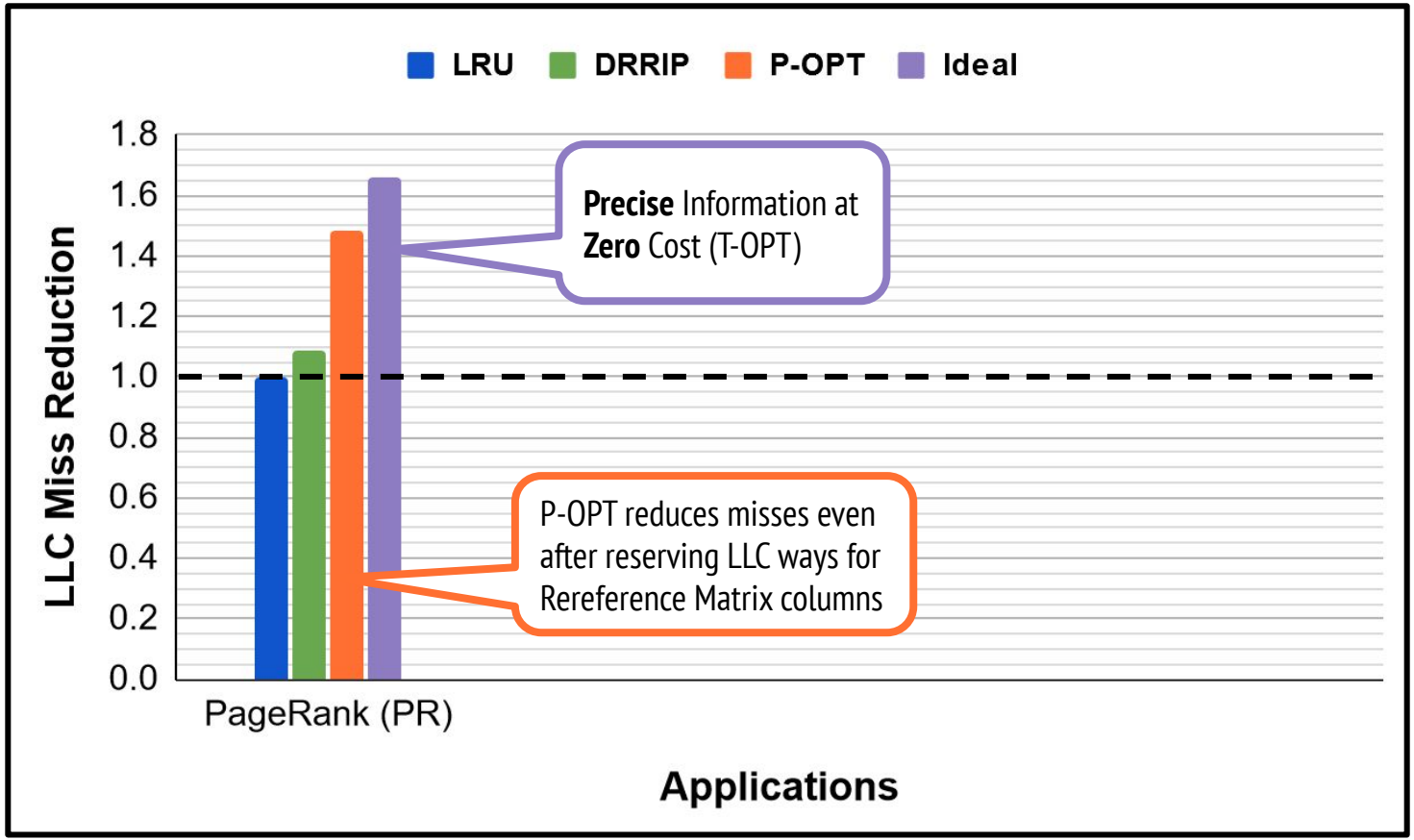
P-OPT Improves Cache Locality



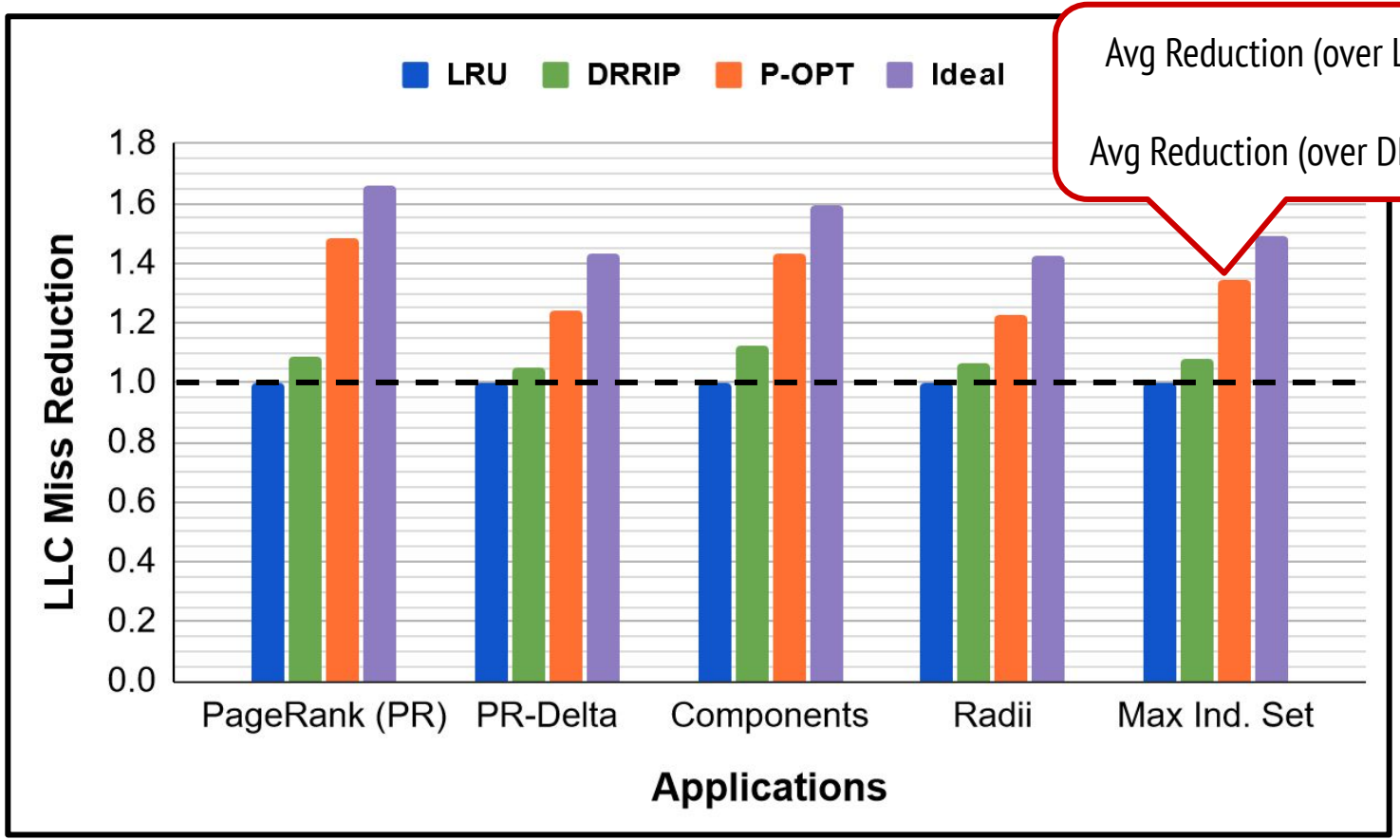
P-OPT Improves Cache Locality



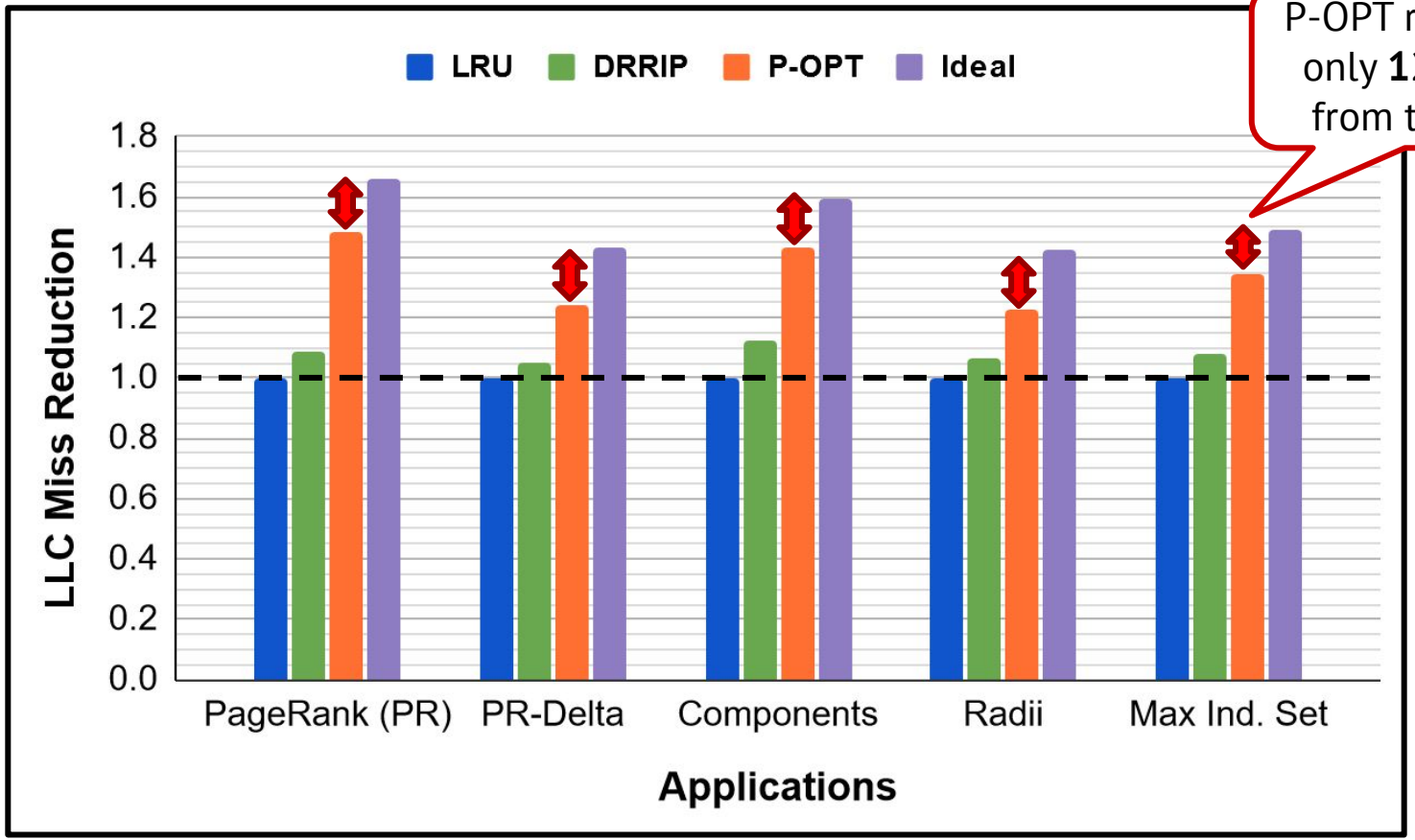
P-OPT Improves Cache Locality



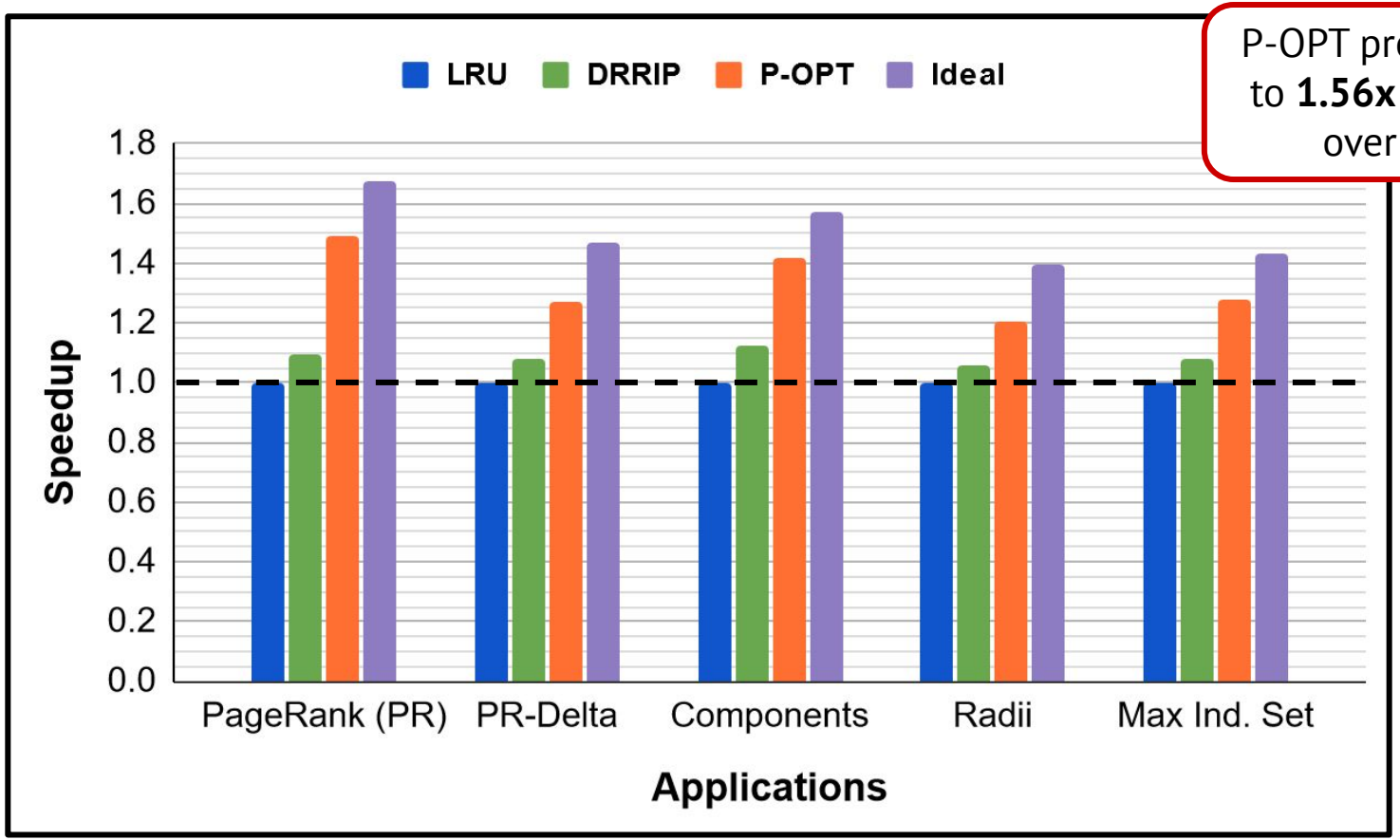
P-OPT Improves Cache Locality



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

More Details In The Paper

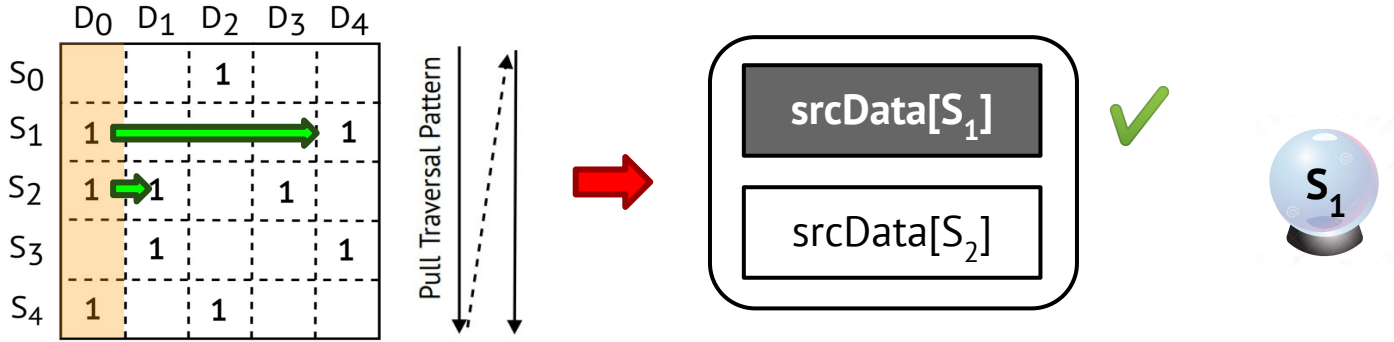
- Modified Rereference Matrix Designs
  - Offsets quality loss from quantization
- Generalized P-OPT Design
  - Support for NUCA caches, handling multiple irregular datatypes, …
- P-OPT is complementary to tiling optimizations:
  - 1-D Tiling (Cagra), Propagation Blocking (PHI)
- P-OPT benefits are agnostic to graph structure and vertex ordering:
  - Unlike prior graph optimizations (HATS, GRASP)
- Cache Simulators available at:

https://github.com/CMUAbstract/POPT-CacheSim-HPCA21

Summary

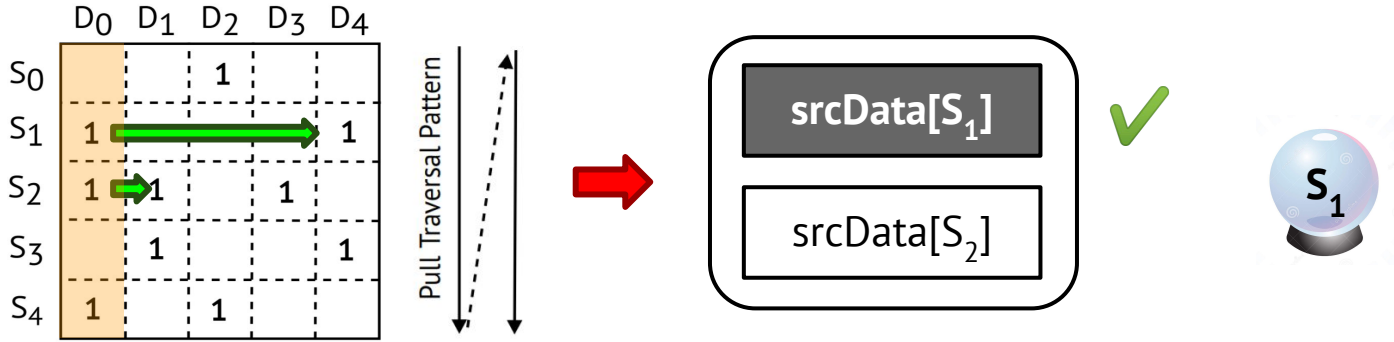
Summary

- **Idea #1:** Graph's transpose can be used for Belady's Optimal Cache Replacement

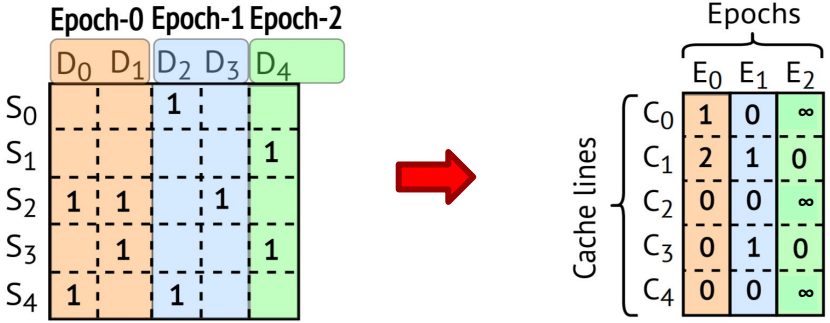


Summary

- **Idea #1:** Graph's transpose can be used for Belady's Optimal Cache Replacement



- **Idea #2:** Quantizing transpose data enables P-OPT (Practical OPTimal replacement)



P-OPT: Practical Optimal Cache Replacement for Graph Analytics

Vignesh Balaji
CMU

Neal Crago
NVIDIA

Aamer Jaleel
NVIDIA

Brandon Lucia
CMU

https://github.com/CMUAbstract/POPT-CacheSim-HPCA21

Electrical & Computer ENGINEERING

Carnegie Mellon

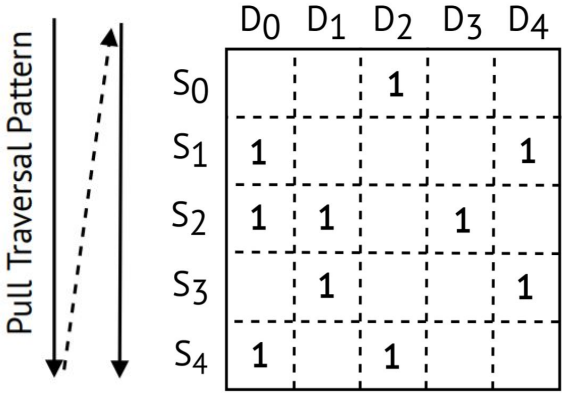
NVIDIA

Backup Slides

P-OPT: Practical OPT Policy Using The Rereference Matrix

Pull Execution

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



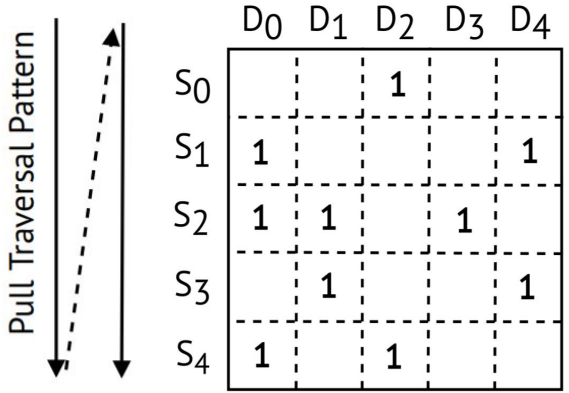
P-OPT: Practical OPT Policy Using The Rereference Matrix

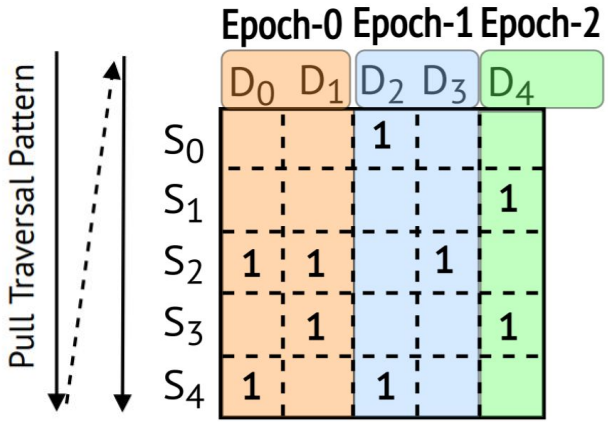
Pull Execution

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

P-OPT Modified Pull Execution

```
for epoch in numEpochs:
  stream_RM_column(epoch)
  for dst in epoch:
    for src in in_neighs(dst):
      dstData[dst] += srcData[src]
```



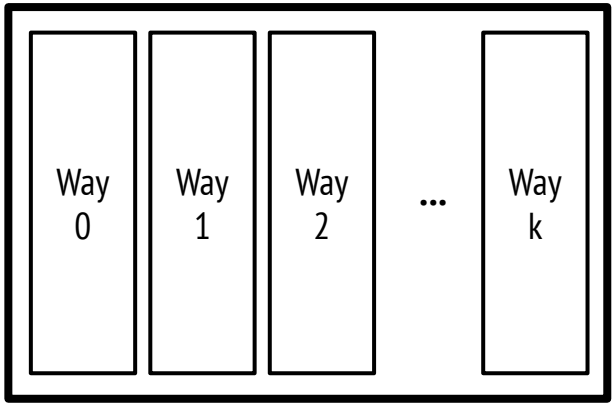


P-OPT: Practical OPT Policy Using The Rereference Matrix

P-OPT Modified Pull Execution

```
for epoch in numEpochs:
  stream_RM_column(epoch)
  for dst in epoch:
    for src in in_neighs(dst):
      dstData[dst] += srcData[src]
```

	E_0	E_1	E_2
C_0	1	0	M
C_1	2	1	0
C_2	0	0	M
C_3	0	1	0
C_4	0	0	M

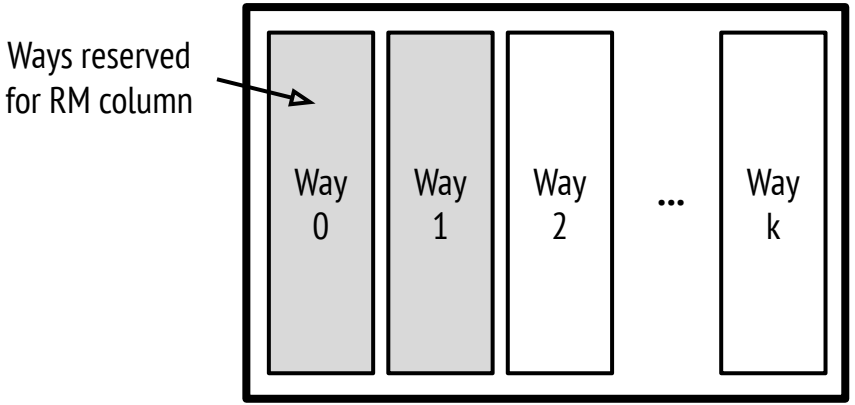


Set-Associative LLC

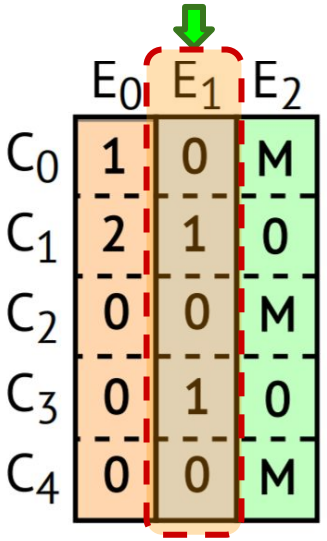
P-OPT: Practical OPT Policy Using The Rereference Matrix

P-OPT Modified Pull Execution

```
for epoch in numEpochs:
  stream_RM_column(epoch)
  for dst in epoch:
    for src in in_neighs(dst):
      dstData[dst] += srcData[src]
```



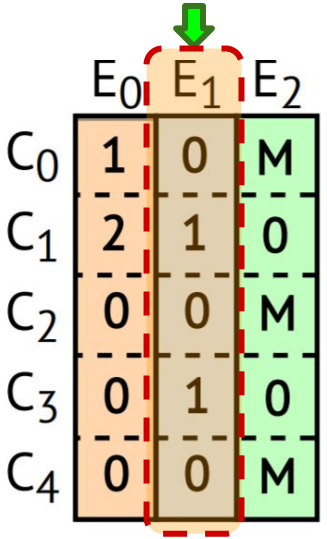
Set-Associative LLC

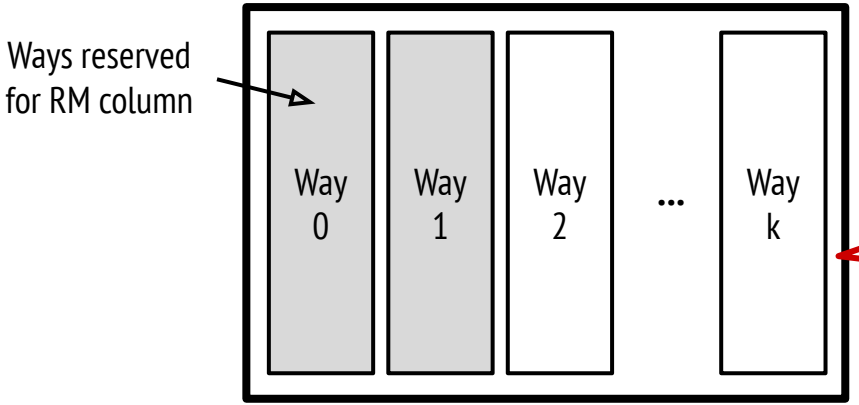


P-OPT: Practical OPT Policy Using The Rereference Matrix

P-OPT Modified Pull Execution

```
for epoch in numEpochs:
  stream_RM_column(epoch)
  for dst in epoch:
    for src in in_neighs(dst):
      dstData[dst] += srcData[src]
```

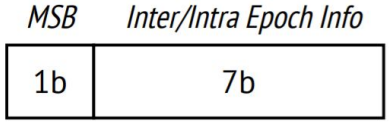




Set-Associative LLC

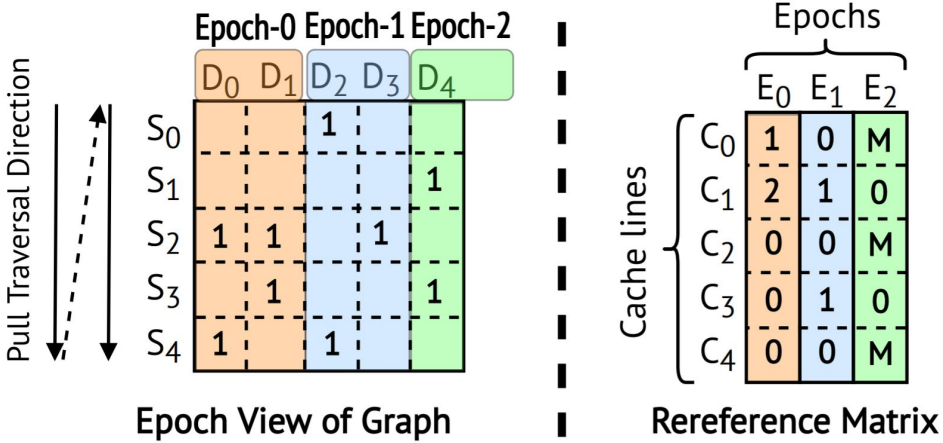
Quantized Next Refs in the reserved LLC ways enable (approximate) Belady's replacement in the remaining ways

Modified Rereference Matrix To Handle Quantization Loss



Rereference Matrix Entry

MSB == 0	*Cacheline Referred in this epoch (7 bits encode last Reference within Epoch)*
MSB == 1	*No reference this epoch (7 bits encode distance to next Epoch)*



Algorithm 2 Finding the next reference via Rereference Matrix

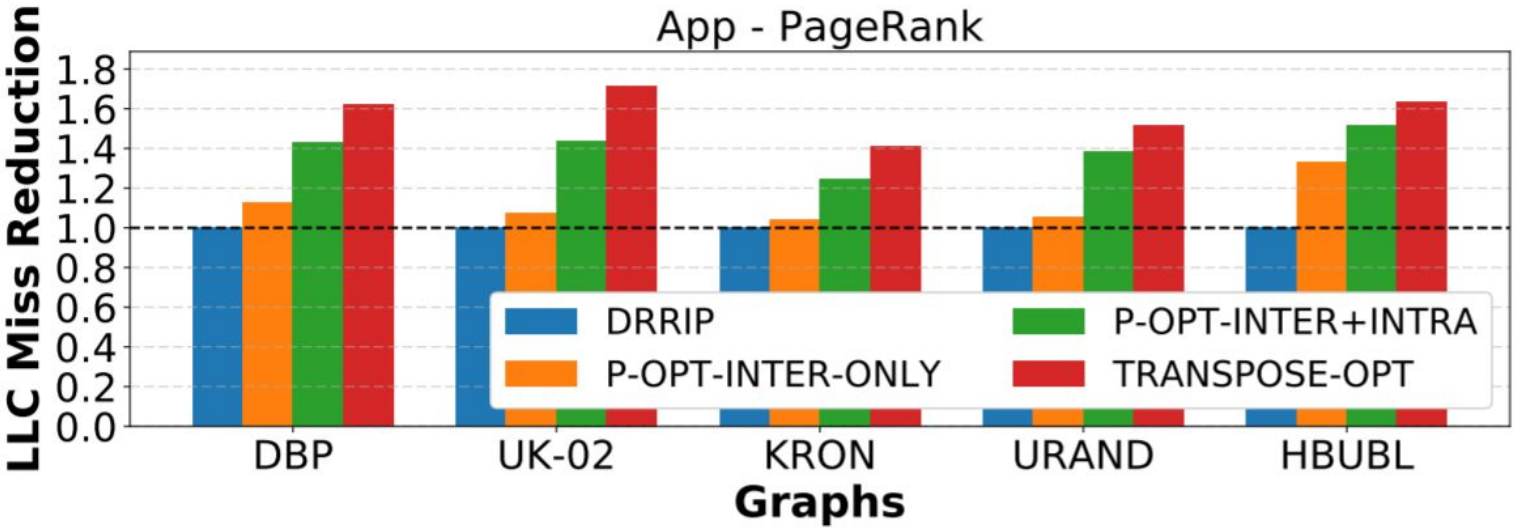
```
1:  procedure FINDNEXTREF(clineID, currDstID)
2:      epochID ← currDstID/epochSize
3:      currEntry ← RerefMatrix[clineID][epochID]
4:      nextEntry ← RerefMatrix[clineID][epochID + 1]
5:      if currEntry[7] == 1 then
6:          return currEntry[6 : 0]
7:      else
8:          lastSubEpoch ← currEntry[6 : 0]
9:          epochStart ← epochID * epochSize
10:         epochOffset ← currDstID − epochStart
11:         currSubEpoch ← epochOffset/subEpochSize
12:         if currSubEpoch ≤ lastSubEpoch then
13:             return 0
14:         else
15:             if nextEntry[7] == 1 then
16:                 return 1 + nextEntry[6 : 0]
17:             else
18:                 return 1
```

Modified Rereference Matrix vs Basic Rereference Matrix

MSB	*Inter/Intra Epoch Info*
1b	7b

Rereference Matrix Entry

MSB == 0	*Cacheline Referred in this epoch (7 bits encode last Reference within Epoch)*
MSB == 1	*No reference this epoch (7 bits encode distance to next Epoch)*



Rereference Matrix Organization Within The LLC

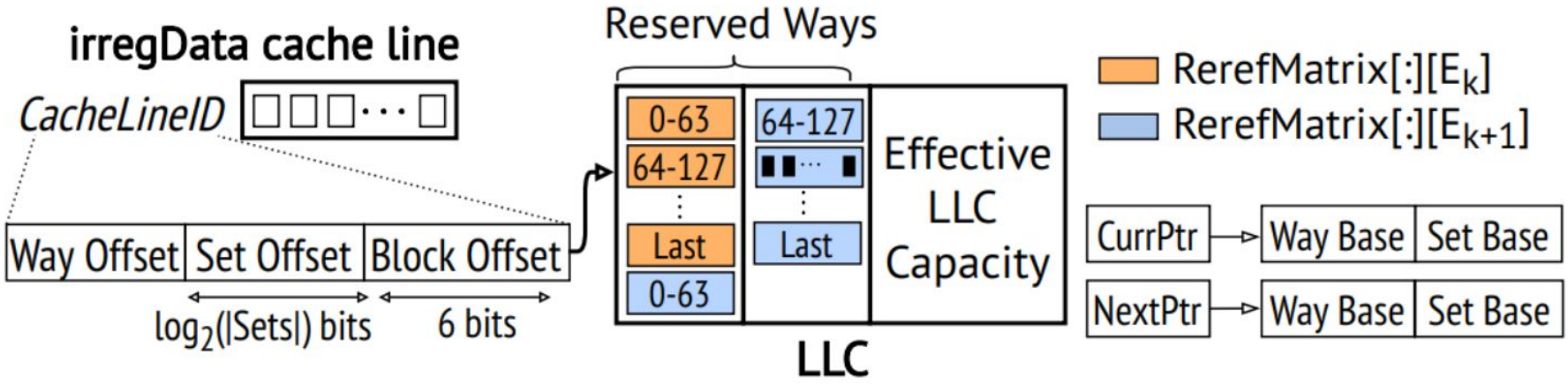


Fig. 8: **Organization of Rereference Matrix columns in the LLC:** *P-OPT pins Rereference Matrix columns in the LLC.*

P-OPT Architecture Modifications

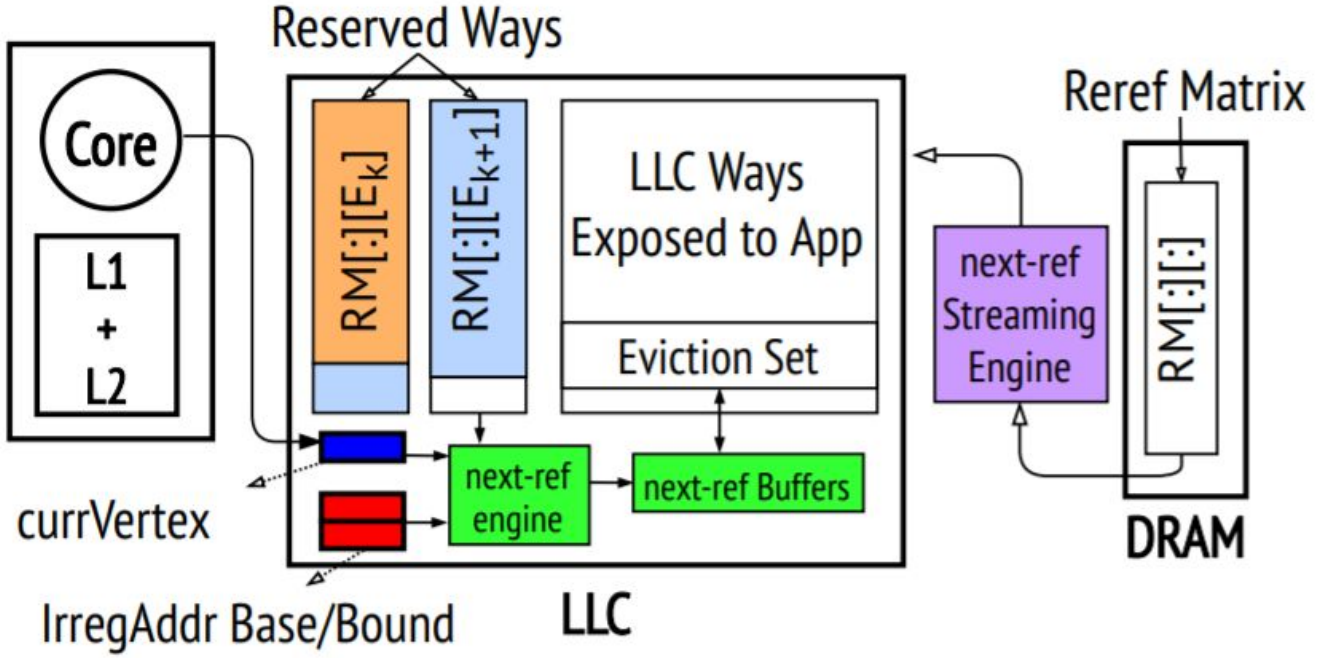


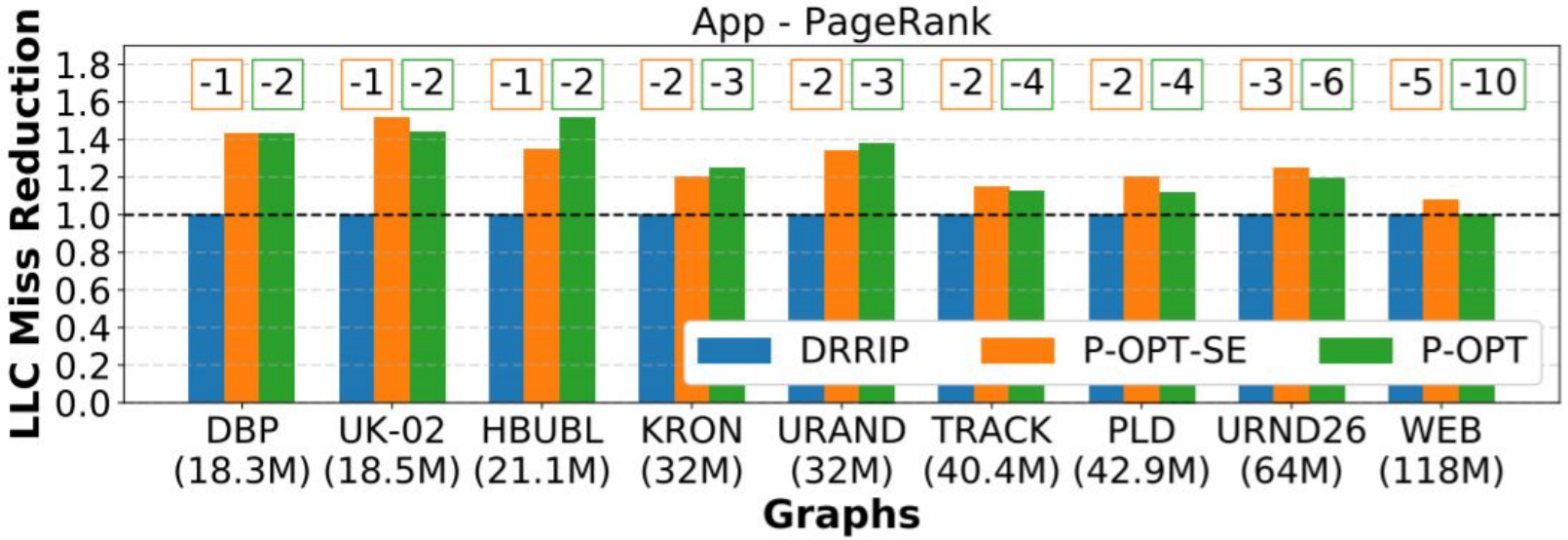
Fig. 9: **Architecture extensions required for P-OPT:** *Components added to a baseline architecture are shown in color.*

P-OPT Scalability With Graph Size

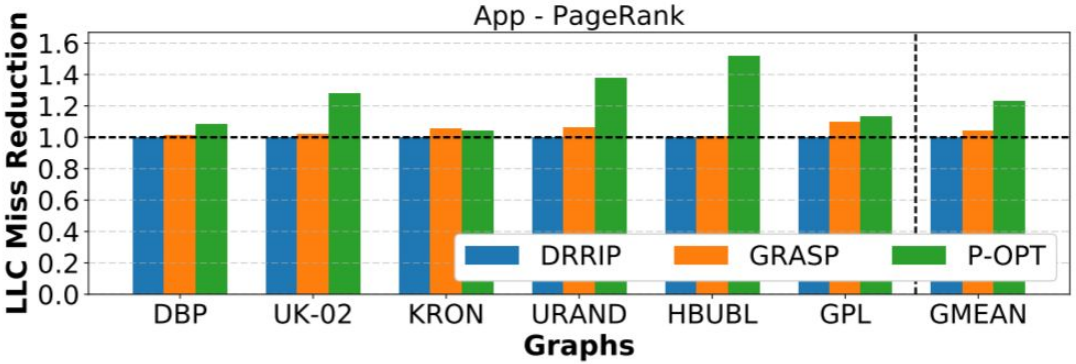
MSB	*Inter/Intra Epoch Info*
1b	7b

Rereference Matrix Entry

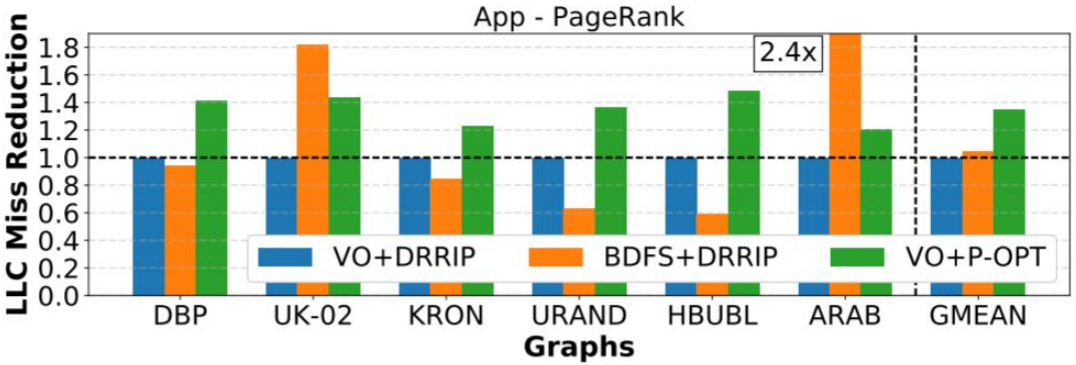
MSB == 0	*Cacheline Referred in this epoch (7 bits encode last Reference within Epoch)*
MSB == 1	*No reference this epoch (7 bits encode distance to next Epoch)*



P-OPT Offers Graph Agnostic Speedups

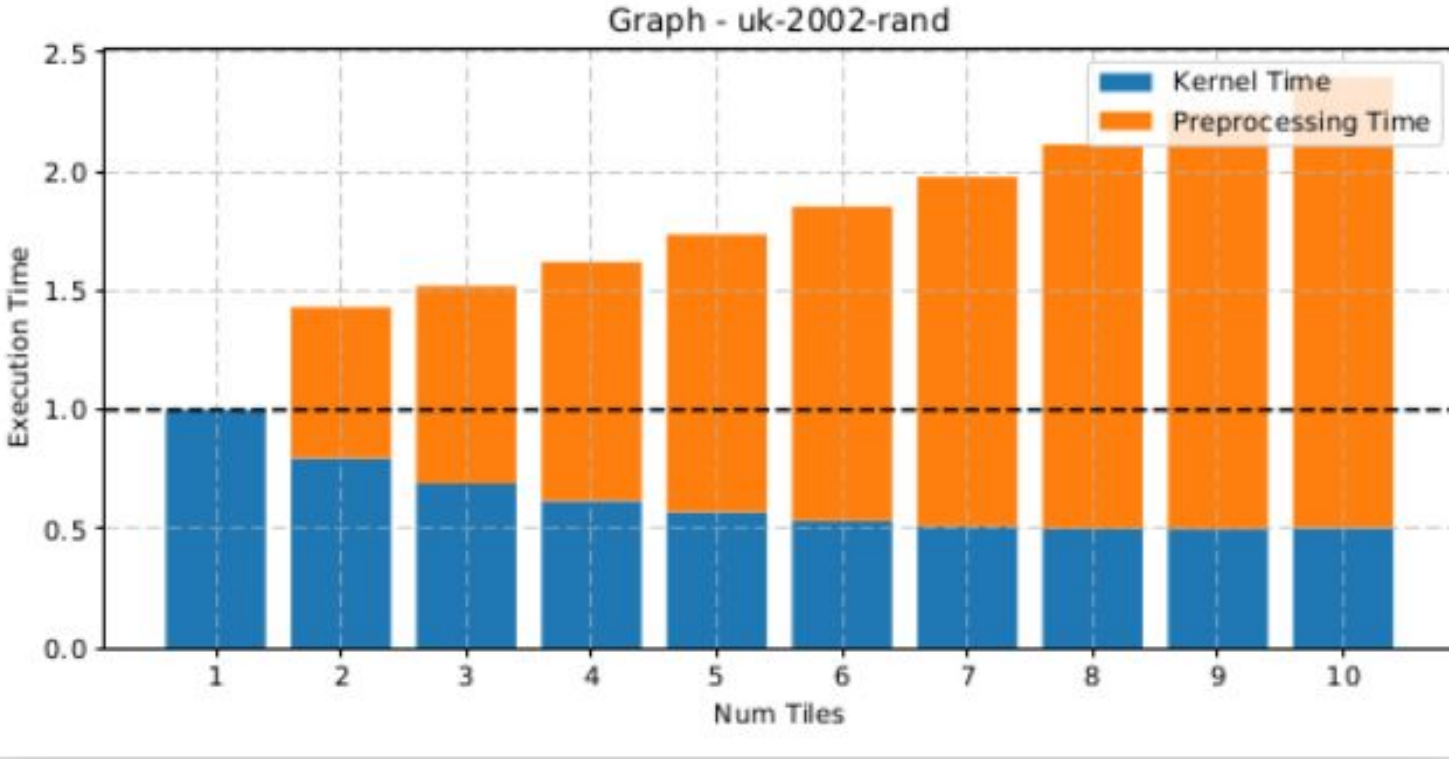


(a) P-OPT compared to GRASP



(b) P-OPT compared to HATS-BDFS

SW 1-D Tiling Incurs High Preprocessing Ovhds



P-OPT & Tiling Are Synergistic Optimizations

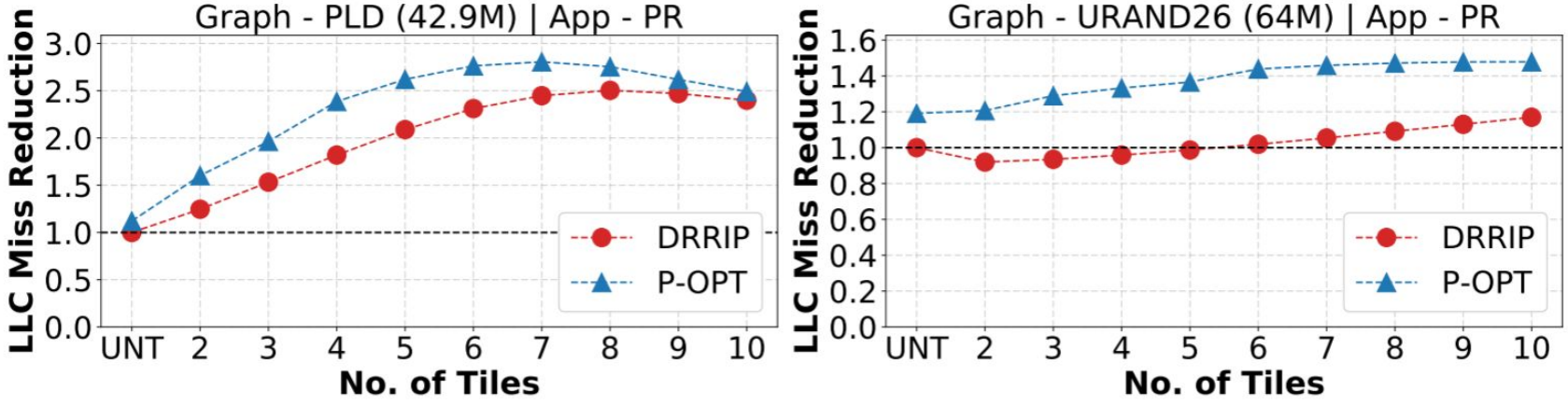
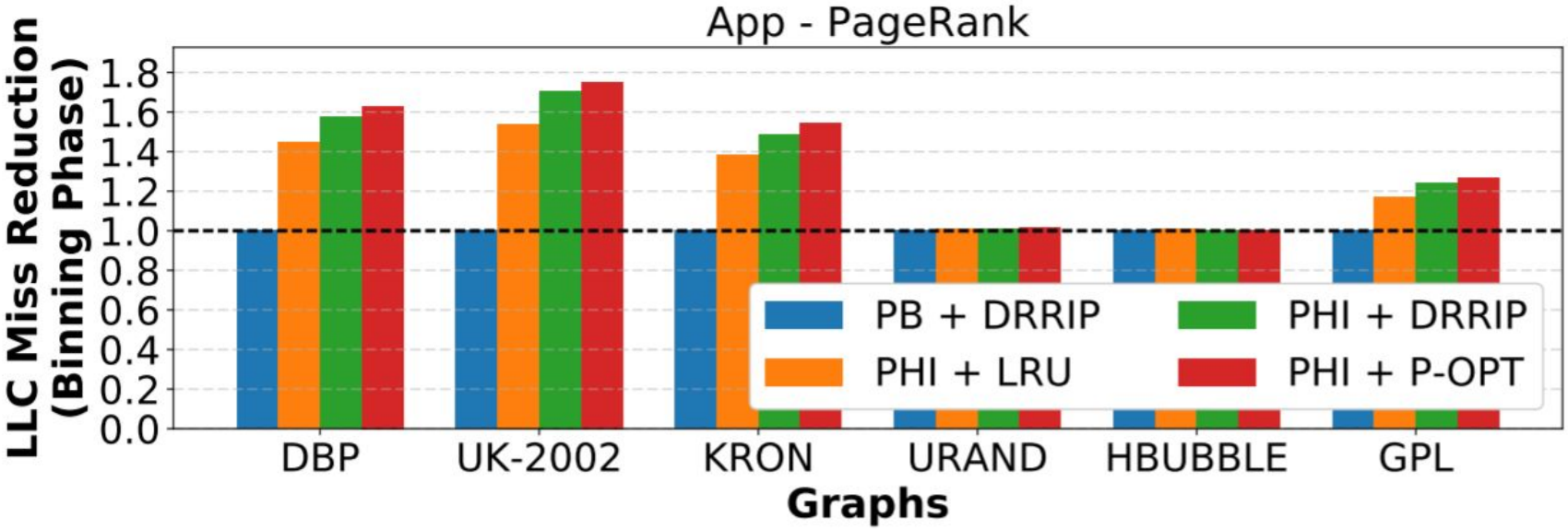


Fig. 13: **P-OPT and Tiling are mutually-enabling optimizations:** *Tiling allows P-OPT to reserve fewer LLC ways while P-OPT can reduce the preprocessing cost of tiling.*

P-OPT Is Complementary to PHI



8-bit Quantization Is Sufficient For P-OPT

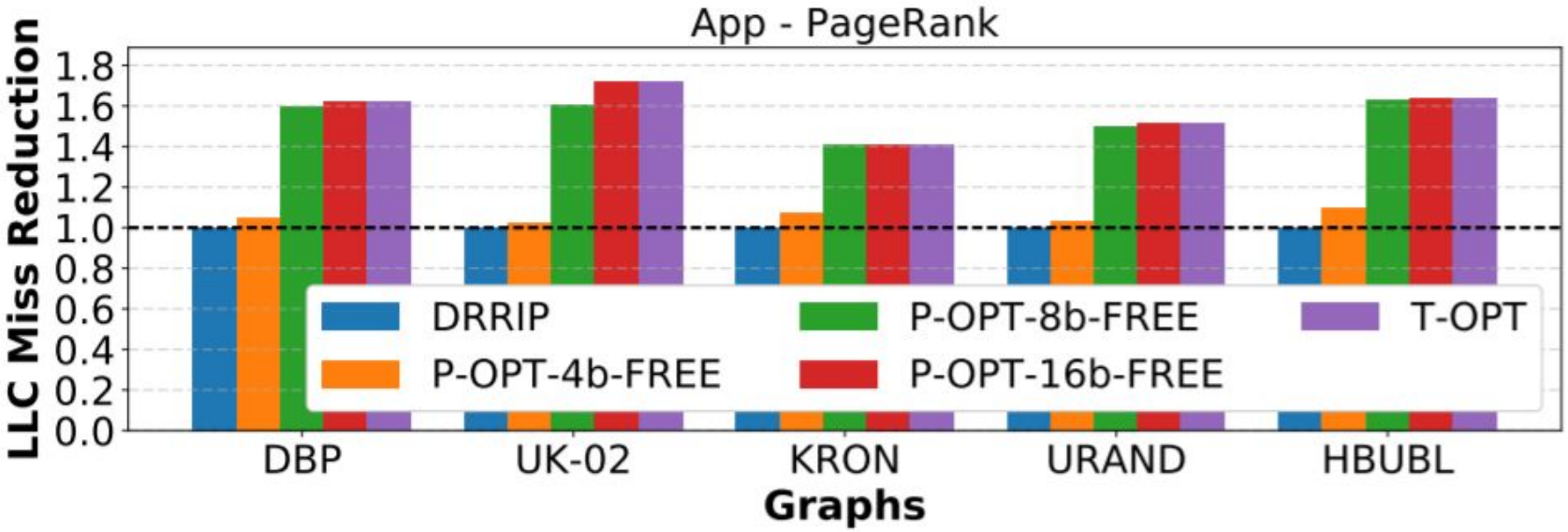
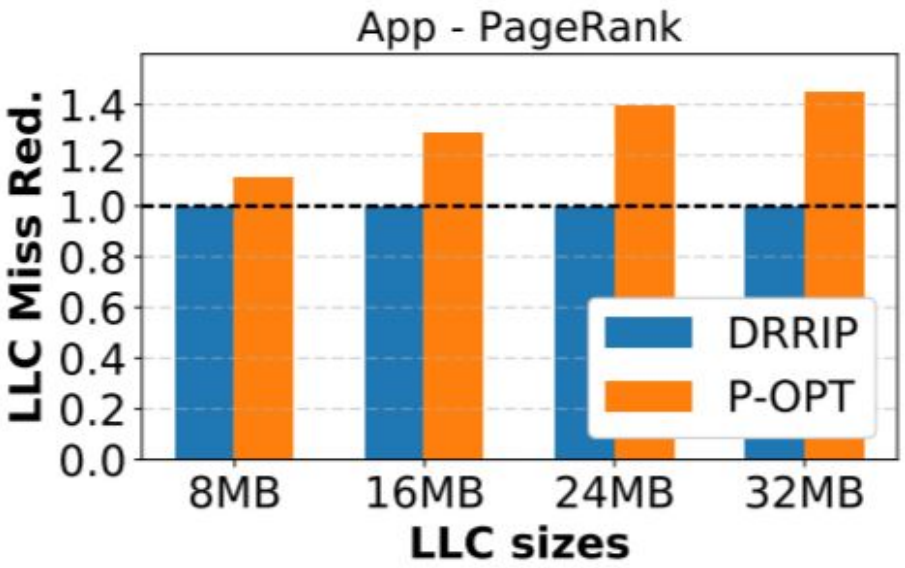
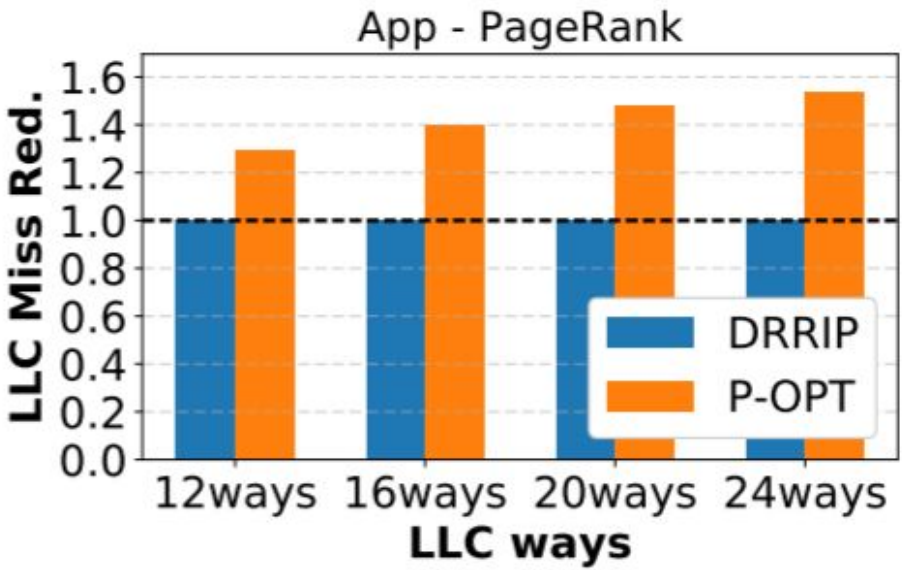


Fig. 15: **P-OPT at different levels of quantization:** *With 8-bit quantization, P-OPT is able to provide a close approximation of the ideal (T-OPT).*

P-OPT's Benefits Increase For Larger, More Associative LLCs





Constructing The Rereference Matrix Is Not Expensive

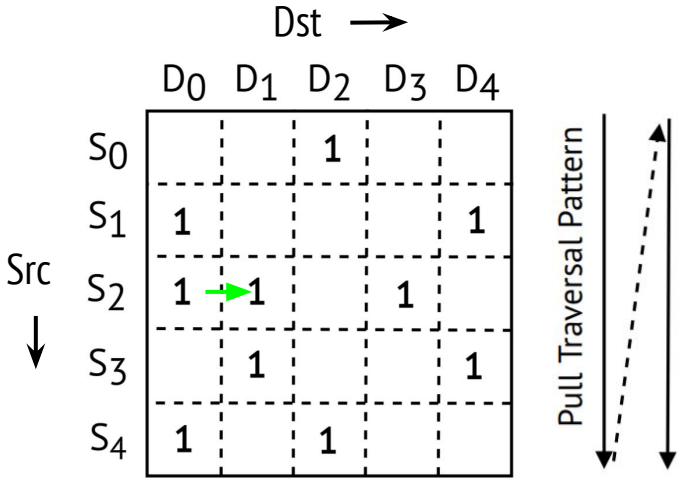
	DBP	UK-02	KRON	URND	HBUBL
POPT Preprocessing Time	0.99s	1.25s	1.59s	1.77s	0.92s
PageRank Execution Time	8.83s	24.64s	4.84s	11.06s	0.89s

TABLE IV: **Relative preprocessing cost for P-OPT**

Average Rereference Matrix Construction Overhead = 19.8%

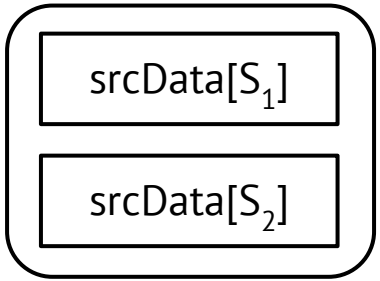
Average Speedup with P-OPT (ignoring preprocessing cost) = 36%

Complexity Improvements with T-OPT



Naive OPT simulation:
Cost of finding vertex's next reference = **O(|E|)**

Transpose-based OPT:
Cost of finding vertex's next reference = **O(|Out_Degree|)**



2-way Set-Associative Cache

Next-Ref of S_2 @ D_0 = D_1

Out-Neighs(S_2) : 0 1 3