Pre-Select Static Caching and Neighborhood Ordering for BFS-like Algorithms on Disk-based Graph Engines

Jiwon Seo in collaboration with

Eunjae Lee, Junghyun Kim, Keunhak Lim, Sam H. Noh

Hanyang University[1], UNIST[2]

Presented in USENIX ATC 2019

Cache Optimizations for Disk-based Graph Engines for BFS-like Algorithms

Jiwon Seo in collaboration with

Eunjae Lee, Junghyun Kim, Keunhak Lim, Sam H. Noh

Hanyang University[1], UNIST[2]

Presented in USENIX ATC 2019

- **Background**
  - Pregel programming model
  - BFS-like algorithms
  - Disk-based graph engine
- **Motivation**
  - Ineffectiveness of page cache for BFS-like Algorithms
- **Our Optimizations**
  - BFS-Aware Static Cache (***BASC***)
  - Neighborhood Ordering (***Norder***)
- **Evaluation**
- **Conclusion**

Pregel Programming Model

- Vertex-Centric Model
- Bulk-Synchronous Parallel Model

→ Graph algorithm as message passing between vertices

→ Messages are delivered in bulk

Pregel Programming Model

- Vertex-Centric Model
- Bulk-Synchronous Parallel Model

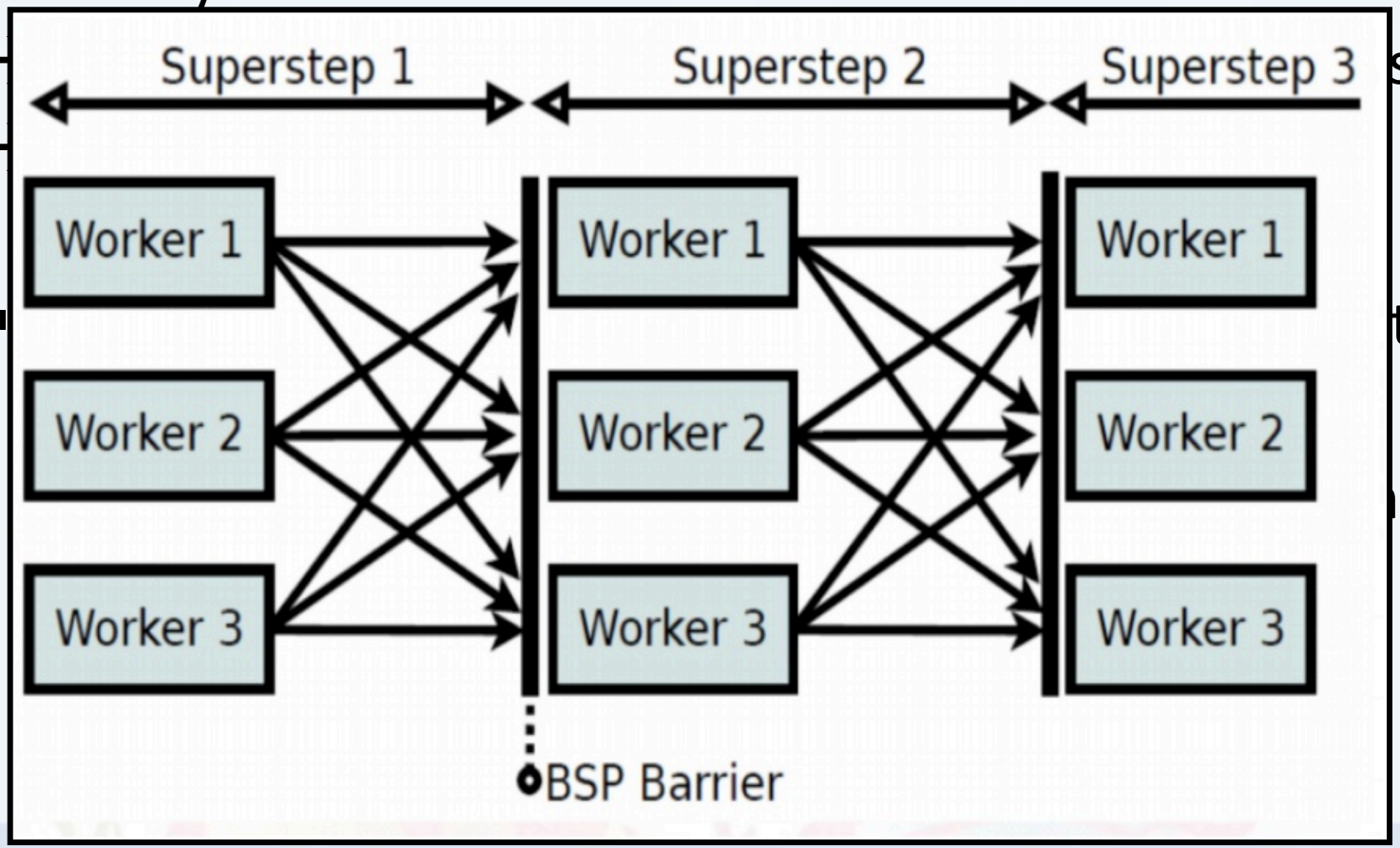
→ Graph algorithm as message passing between vertices

→ Messages are delivered in bulk

- Pregel programs has series of iterations (called super-step)
  - Vertex-function runs on each (active) vertex
  - After an iteration, messages are delivered synchronously

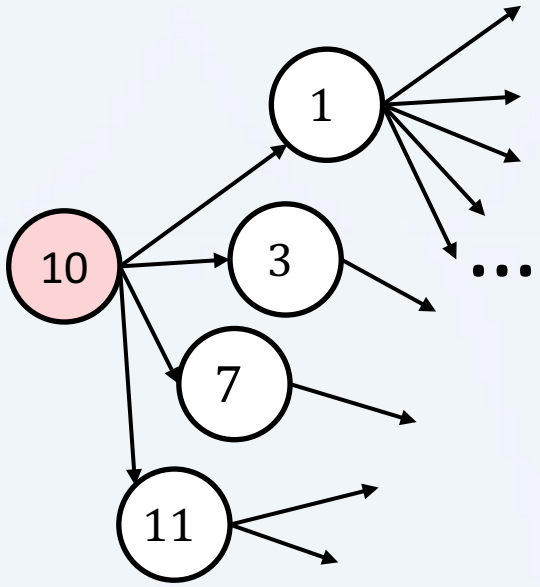
Pregel Programming Model

- Vertex-Centric Model
- Bulk-Synchronous Parallel Model



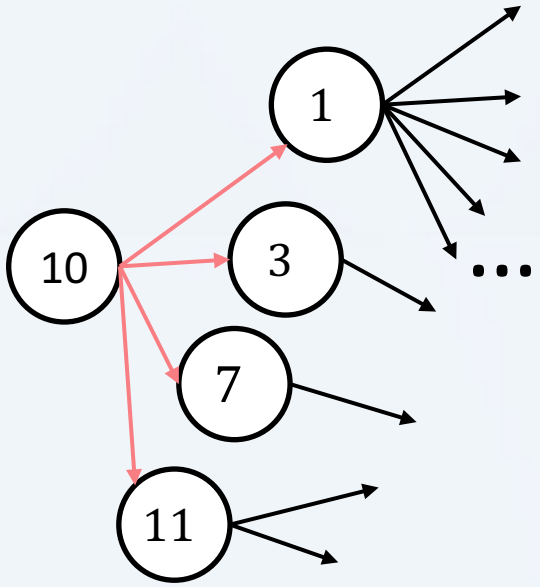
BFS-like Algorithms

- Starts from a subset of vertices
- Traverse the graph recursively
- E.g.
  - *Breath-first search (**BFS**)*
  - *Diameter estimation (**DIAM**)*
  - *Betweenness centrality (**BC**)*
  - *Weakly connected component (**WCC**)*
  - *Shortest path (**SP**)*
  - *All-pair shortest path (**APSP**)*
  - *…*



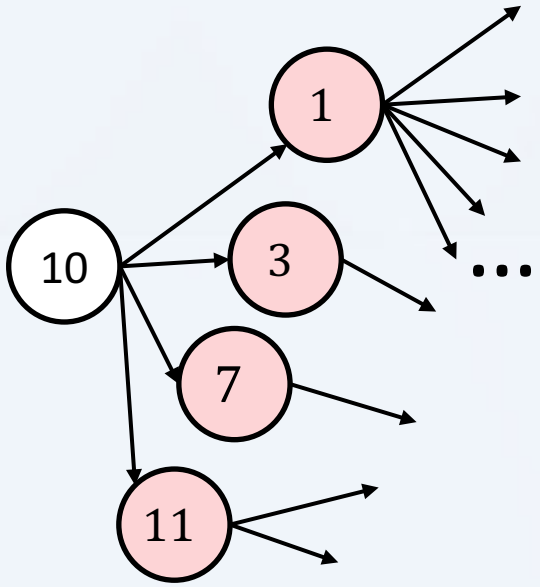
BFS-like Algorithms

- Starts from a subset of vertices
- Traverse the graph recursively
- E.g.
  - *Breath-first search (**BFS**)*
  - *Diameter estimation (**DIAM**)*
  - *Betweenness centrality (**BC**)*
  - *Weakly connected component (**WCC**)*
  - *Shortest path (**SP**)*
  - *All-pair shortest path (**APSP**)*
  - *…*



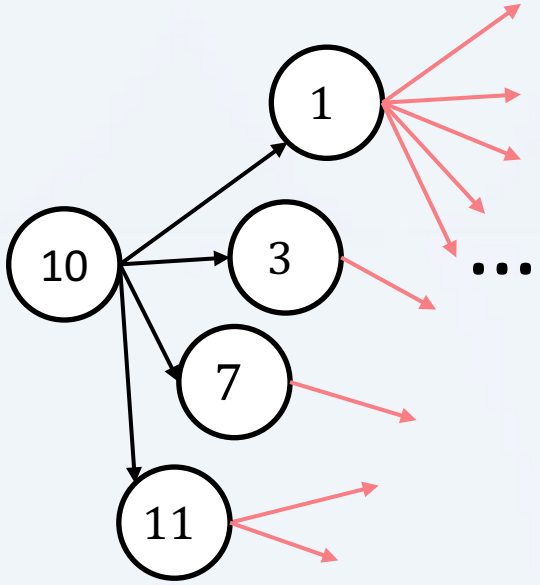
BFS-like Algorithms

- Starts from a subset of vertices
- Traverse the graph recursively
- E.g.
  - *Breath-first search (**BFS**)*
  - *Diameter estimation (**DIAM**)*
  - *Betweenness centrality (**BC**)*
  - *Weakly connected component (**WCC**)*
  - *Shortest path (**SP**)*
  - *All-pair shortest path (**APSP**)*
  - *…*



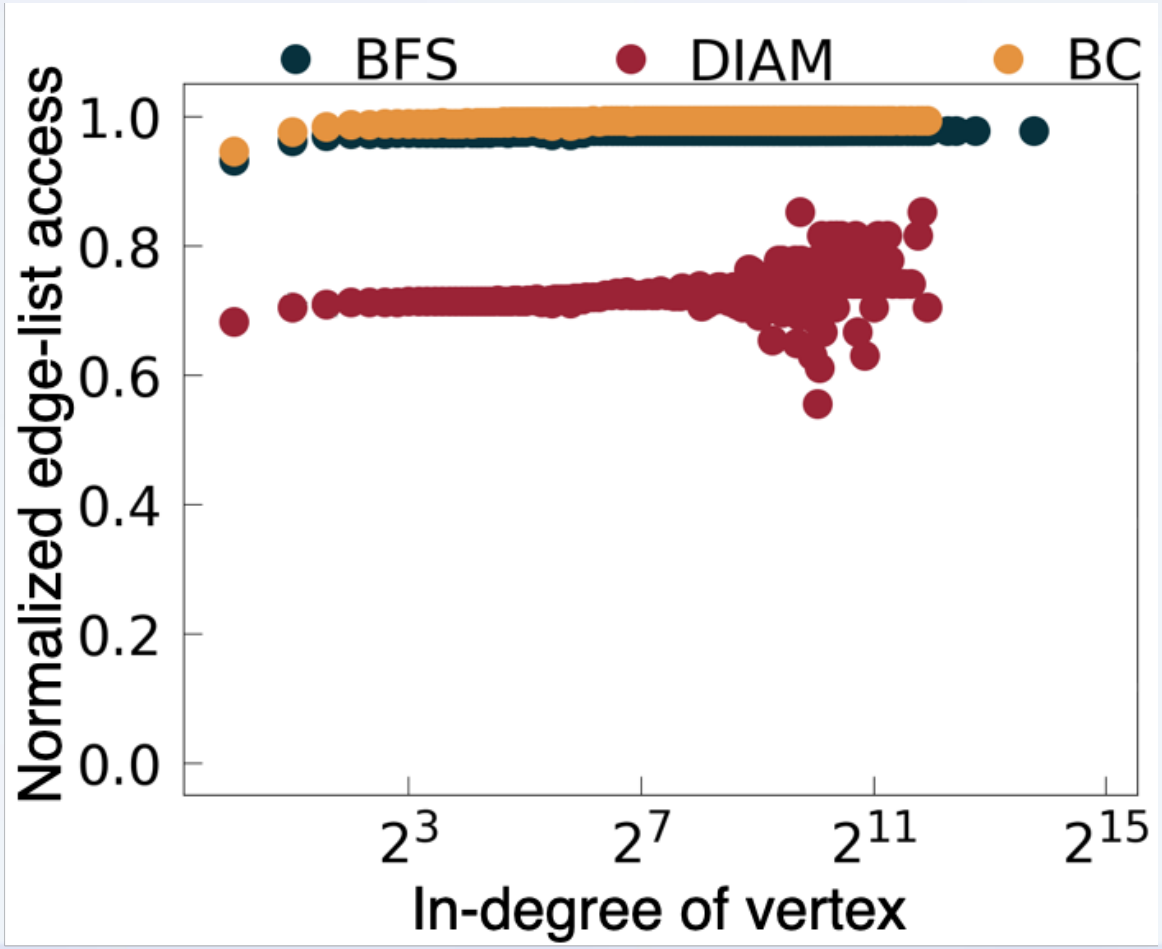
BFS-like Algorithms

- Starts from a subset of vertices
- Traverse the graph recursively
- E.g.
  - *Breath-first search (**BFS**)*
  - *Diameter estimation (**DIAM**)*
  - *Betweenness centrality (**BC**)*
  - *Weakly connected component (**WCC**)*
  - *Shortest path (**SP**)*
  - *All-pair shortest path (**APSP**)*
  - *…*



BFS-like Algorithms

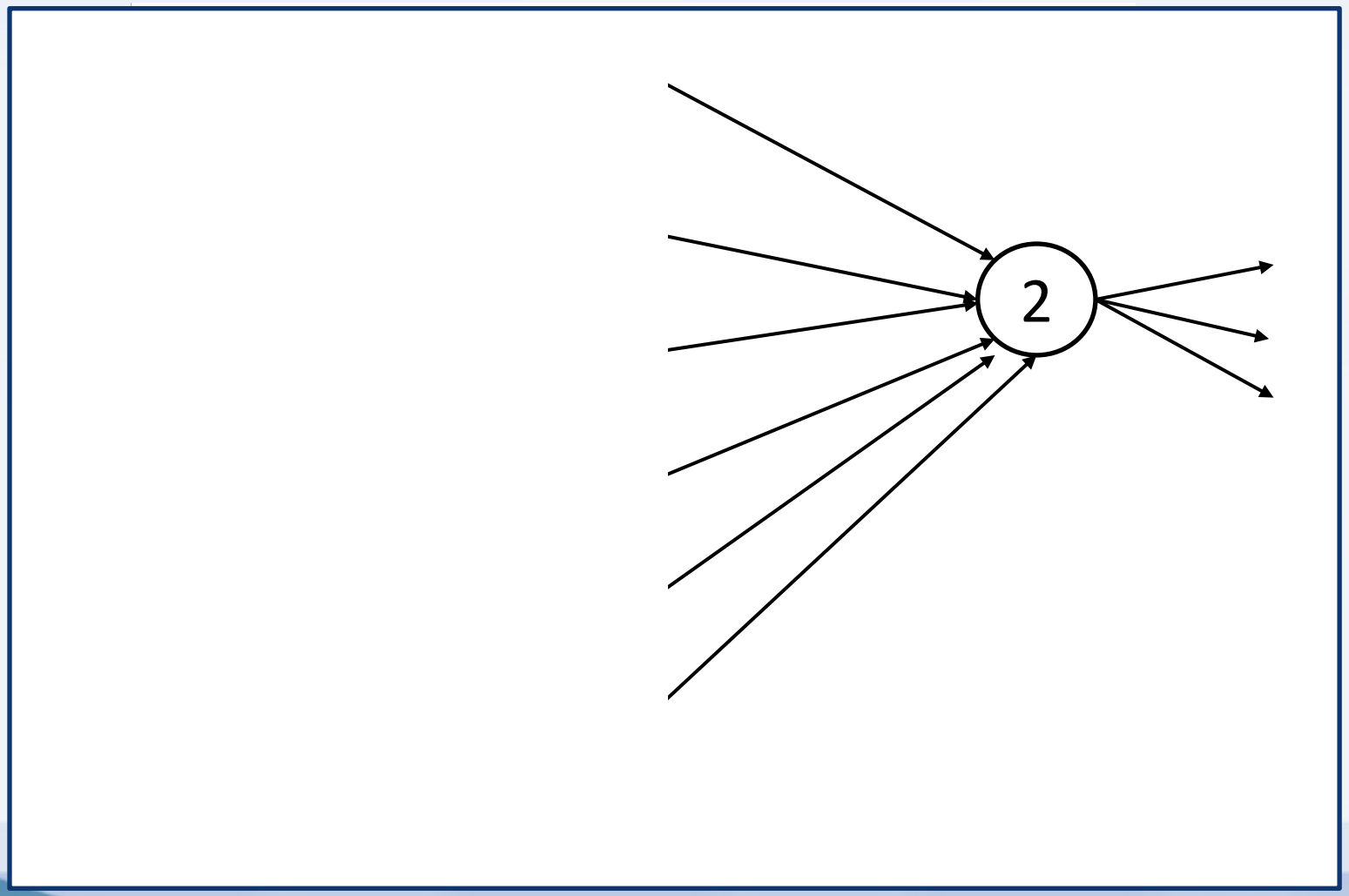
- **Uniform edge-list access**



LiveJournal Network

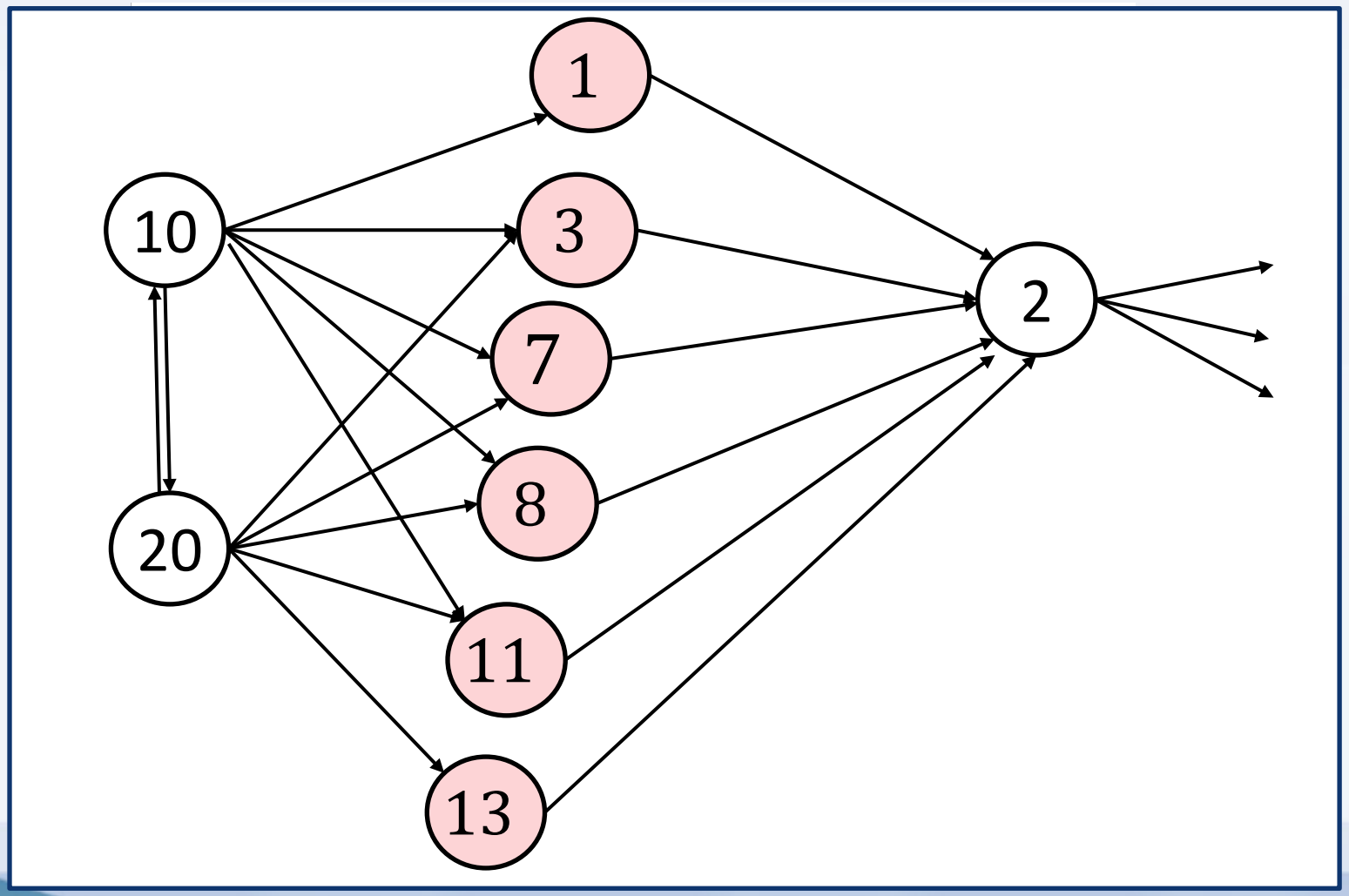
BFS-like Algorithms

- Uniform edge-list access



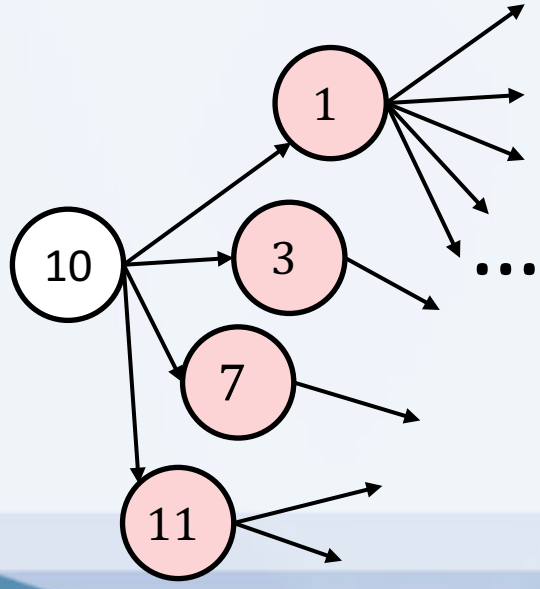
BFS-like Algorithms

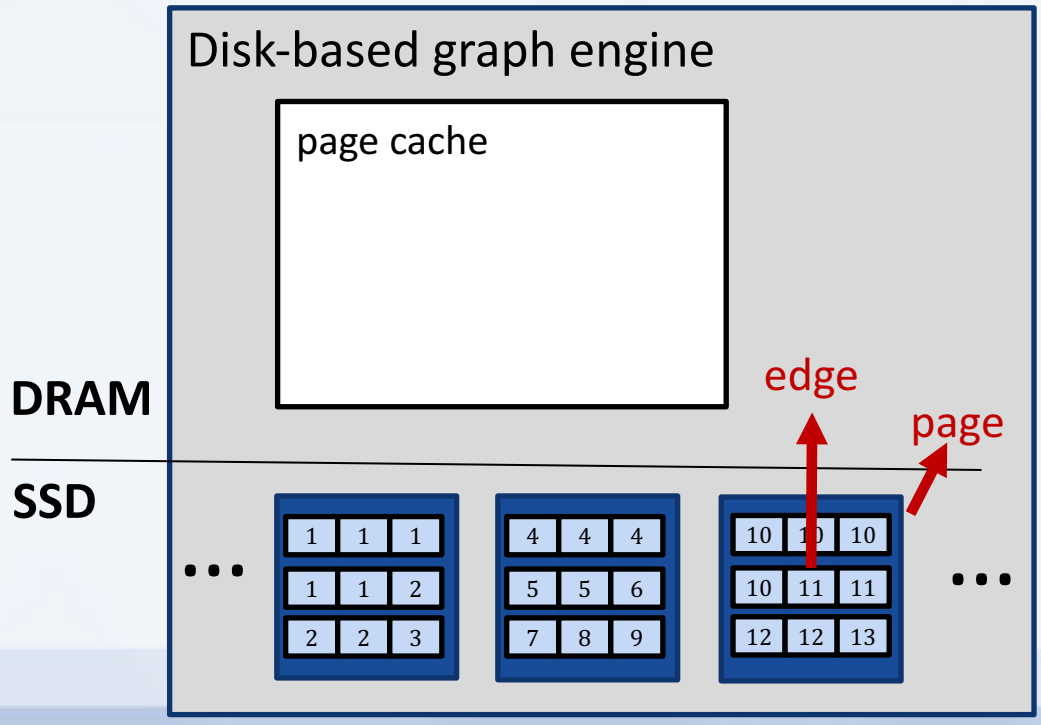
- **Uniform edge-list access**



Disk-based Graph Engine

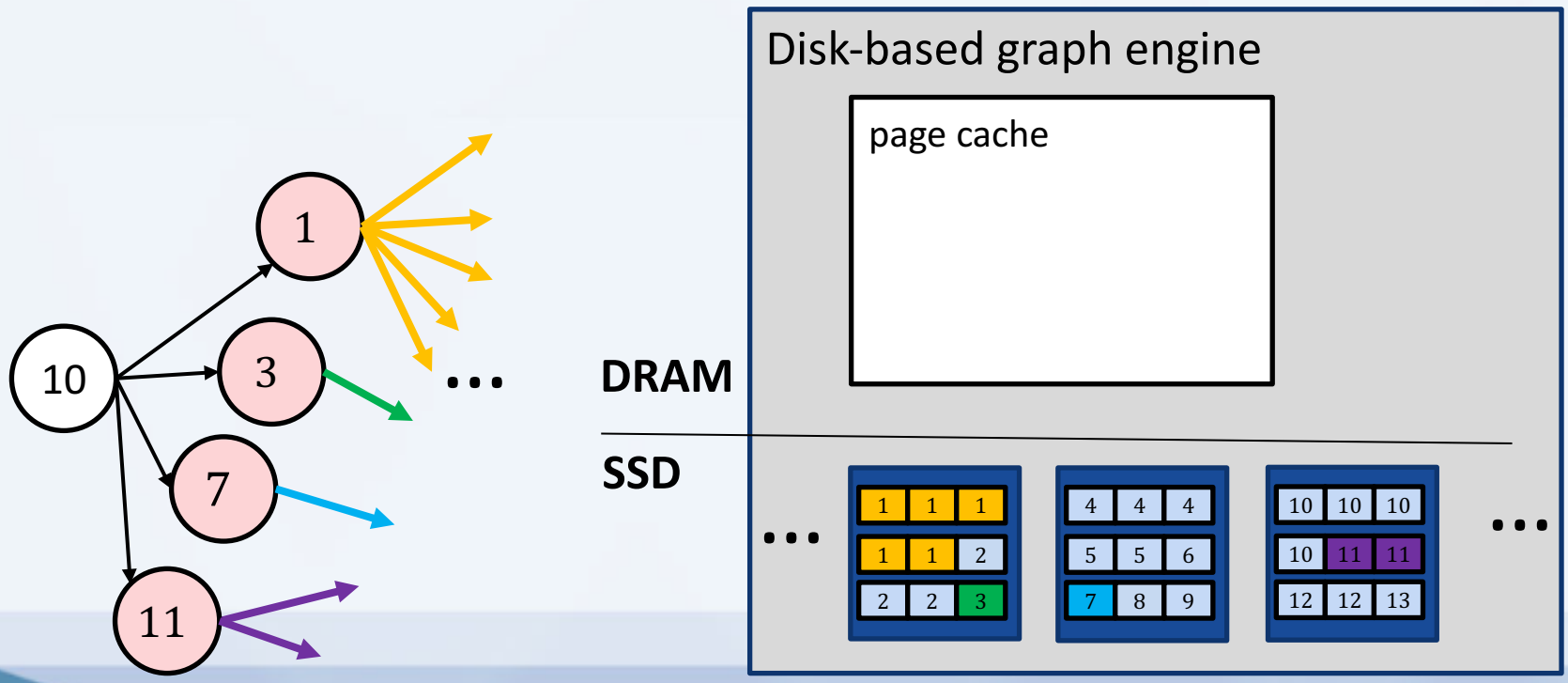
- **Stores edge lists on disk**
  - Sorted by ID of source vertex
  - When vertices are visited, their edge lists are loaded to ***page cache***





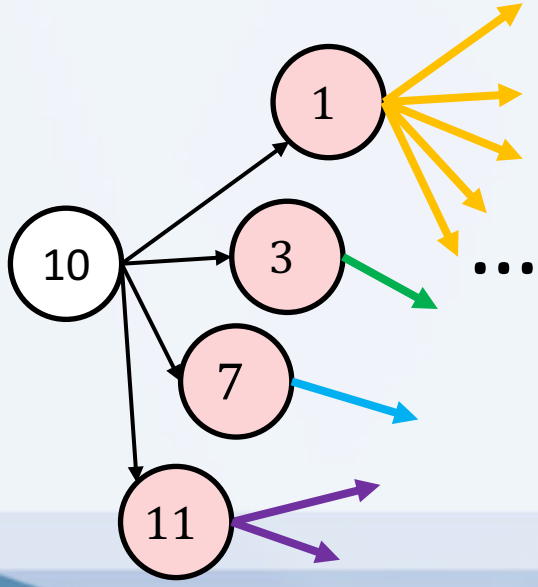
Disk-based Graph Engine

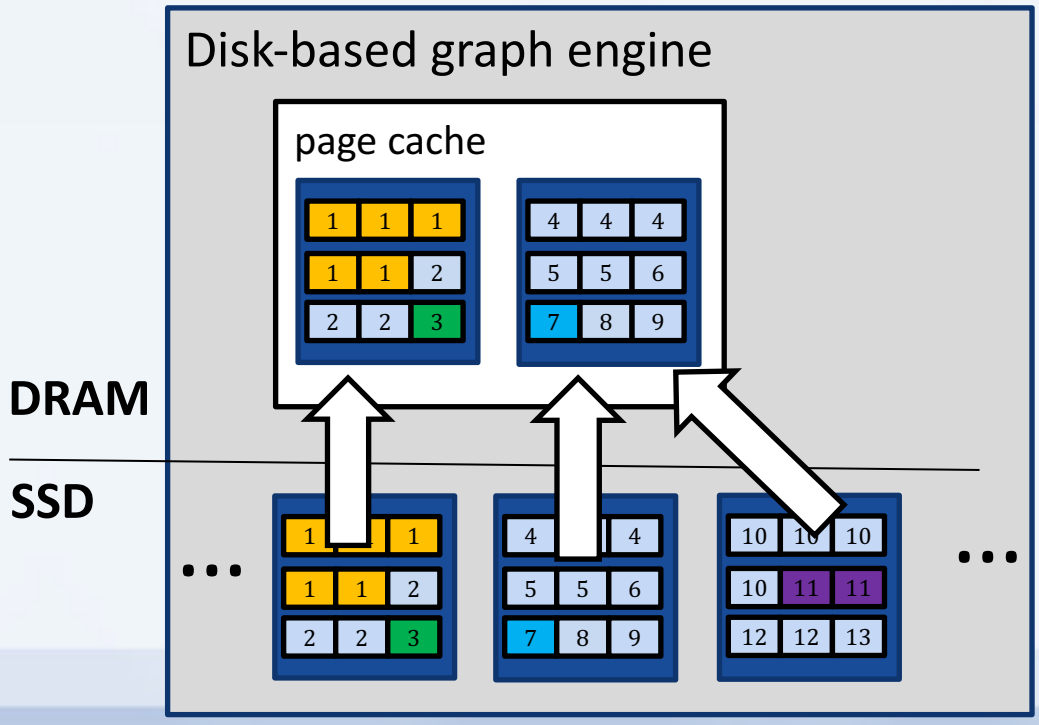
- **Stores edge lists on disk**
  - Sorted by ID of source vertex
  - When vertices are visited, their edge lists are loaded to ***page cache***



Disk-based Graph Engine

- **Stores edge lists on disk**
  - Sorted by ID of source vertex
  - When vertices are visited, their edge lists are loaded to ***page cache***





Disk-based Graph Engine

- **Typical execution steps**
  1. Vertex computation (w/ received messages)
  2. Edge lists retrieval from disk
  3. Sending message to neighbors
  4. 1~3 is repeated

➔ Step 2 is performance bottleneck especially for BFS-like algorithm

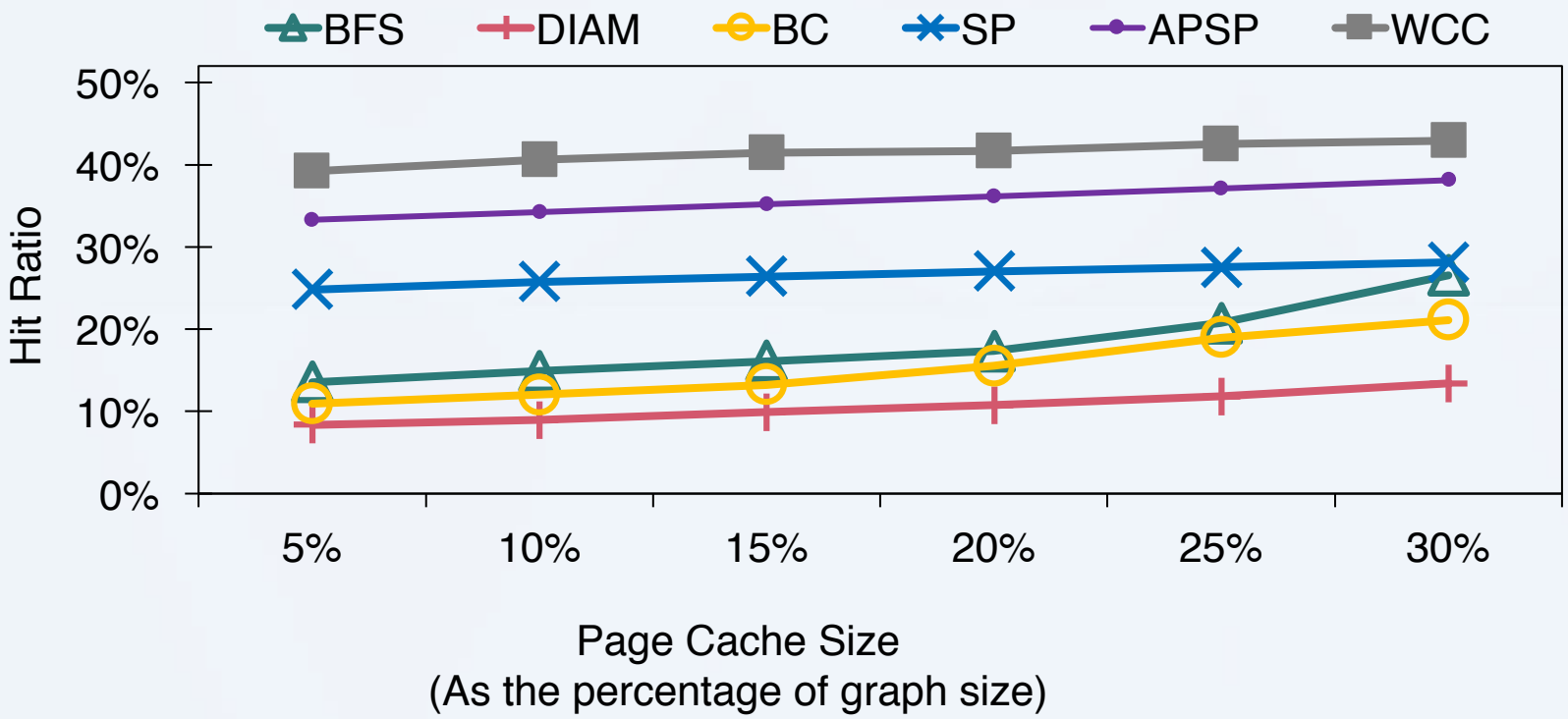
Ineffectiveness of Page Cache for BFS-like Algorithms

We investigated:

1. How page cache size affects hit ratio
2. How page cache size affects execution time
3. Memory utilization of page cache

Ineffectiveness of Page Cache for BFS-like Algorithms

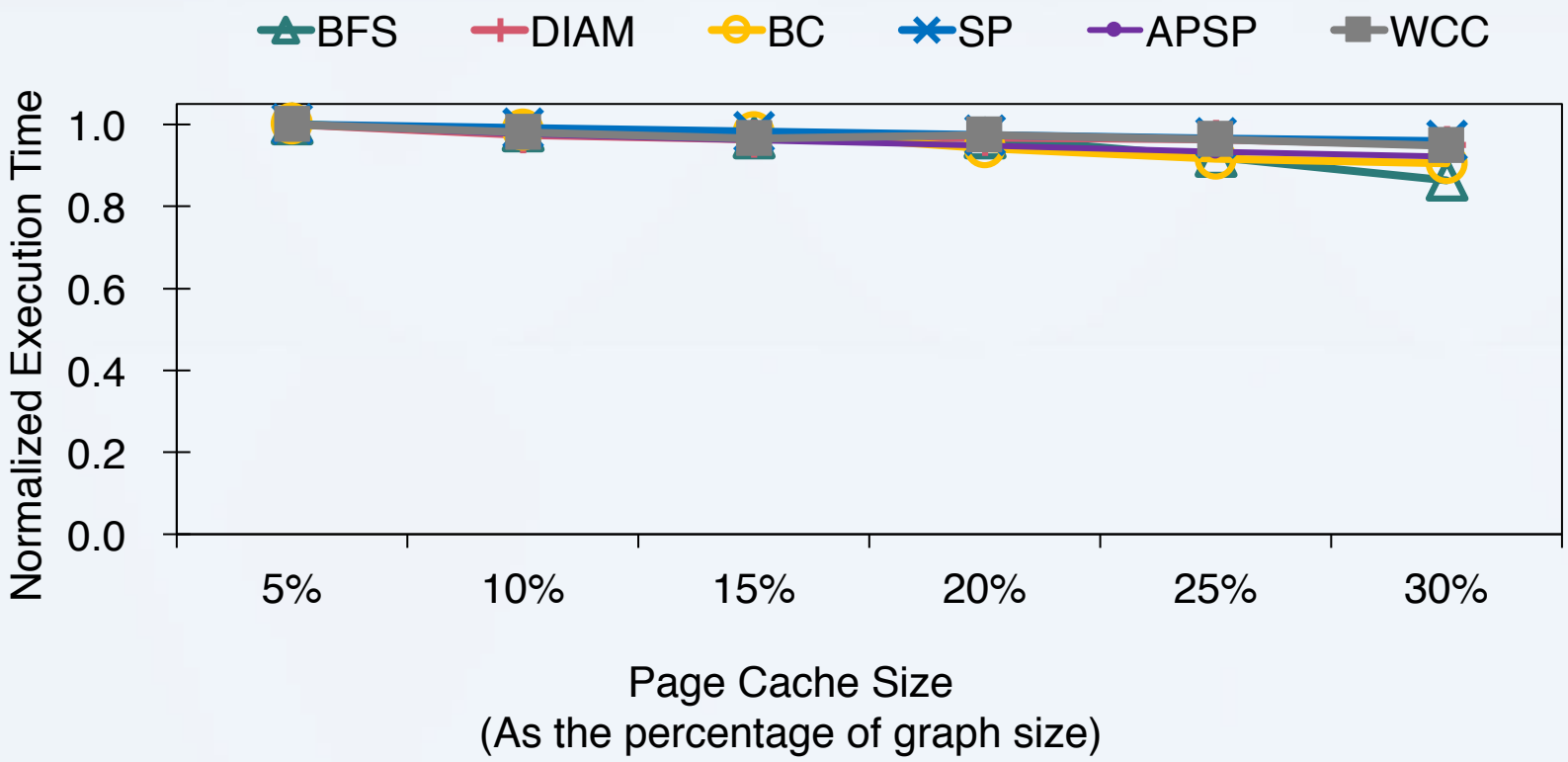
1. How page cache size affects hit ratio



*LiveJournal on FlashGraph

Ineffectiveness of Page Cache for BFS-like Algorithms

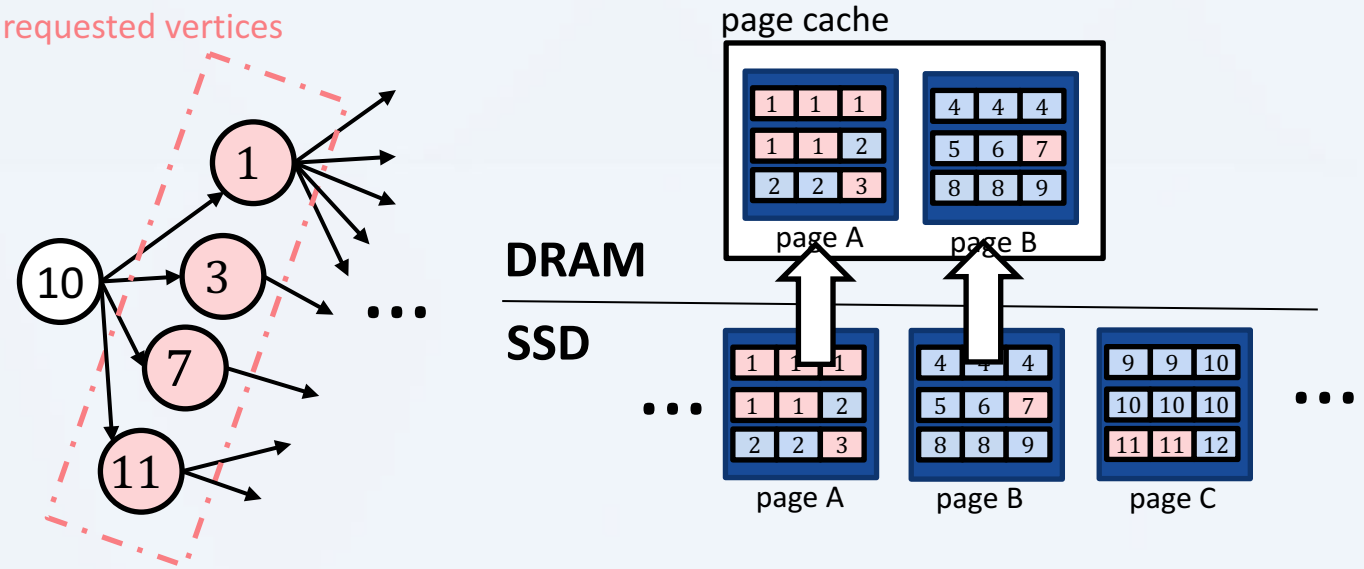
2. How page cache size affects execution time



*LiveJournal on FlashGraph

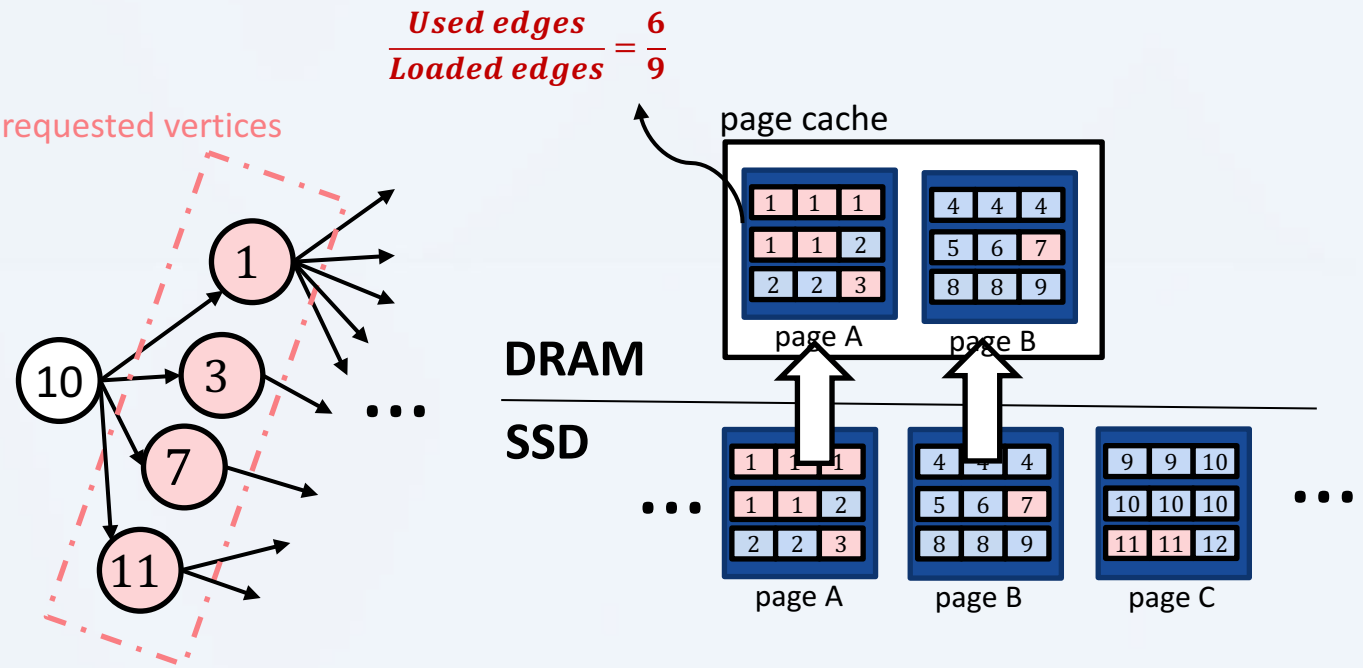
Ineffectiveness of Page Cache for BFS-like Algorithms

3. Memory utilization of page cache
   - Page cache shows <u>poor</u> memory utilization



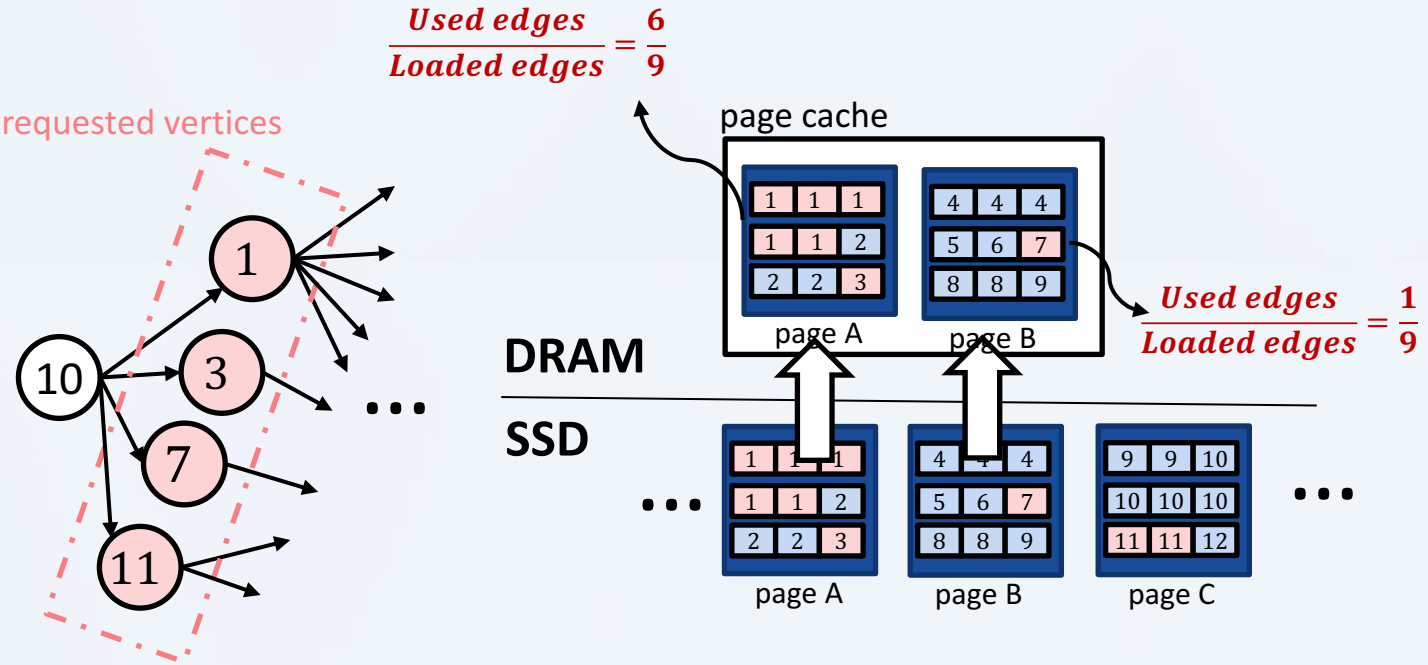
Ineffectiveness of Page Cache for BFS-like Algorithms

3. Memory utilization of page cache
   - Page cache shows <u>poor</u> memory utilization



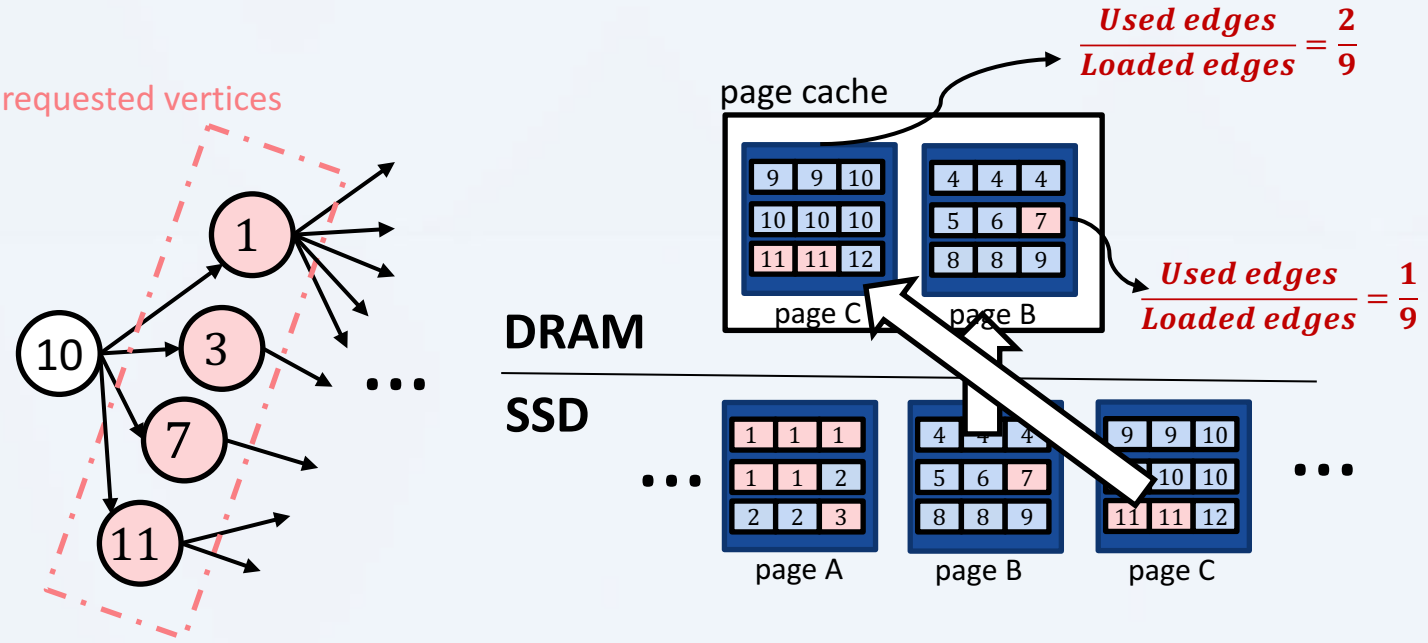
Ineffectiveness of Page Cache for BFS-like Algorithms

3. Memory utilization of page cache
   - Page cache shows <u>poor</u> memory utilization



Ineffectiveness of Page Cache for BFS-like Algorithms

3. Memory utilization of page cache
   - Page cache shows <u>poor</u> memory utilization



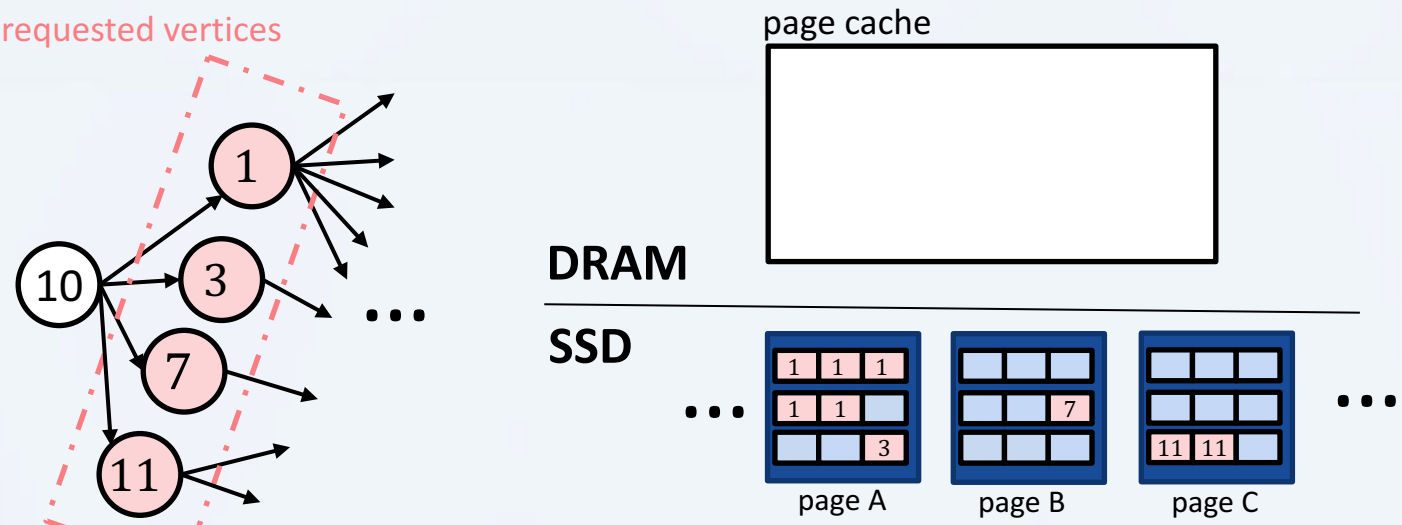
Our Optimization

1. **BFS-Aware Static Cache (*BASC*)**
   - Statically stores selected edge lists

2. **Neighborhood Ordering (*Norder*)**
   - Re-assigning vertex IDs to improve the locality of access

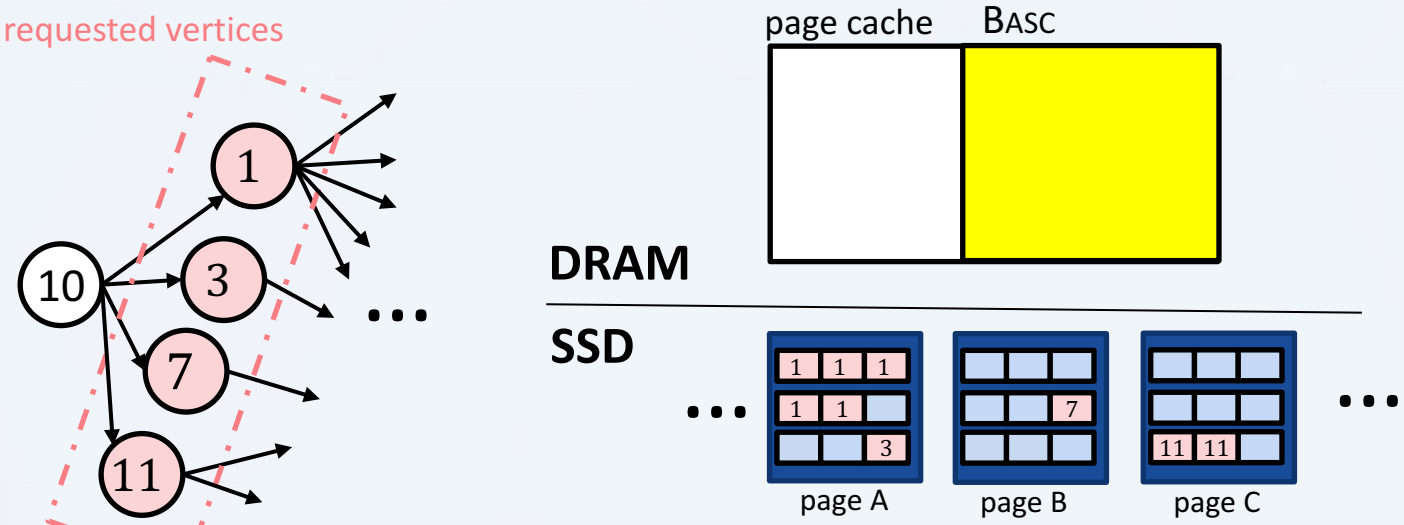
BASC: BFS-Aware Static Cache

- Keep separate cache for selected edge lists
- **Pre-loaded**: edge lists pre-selected through pre-analysis
- **Static**: contents of cache do not change



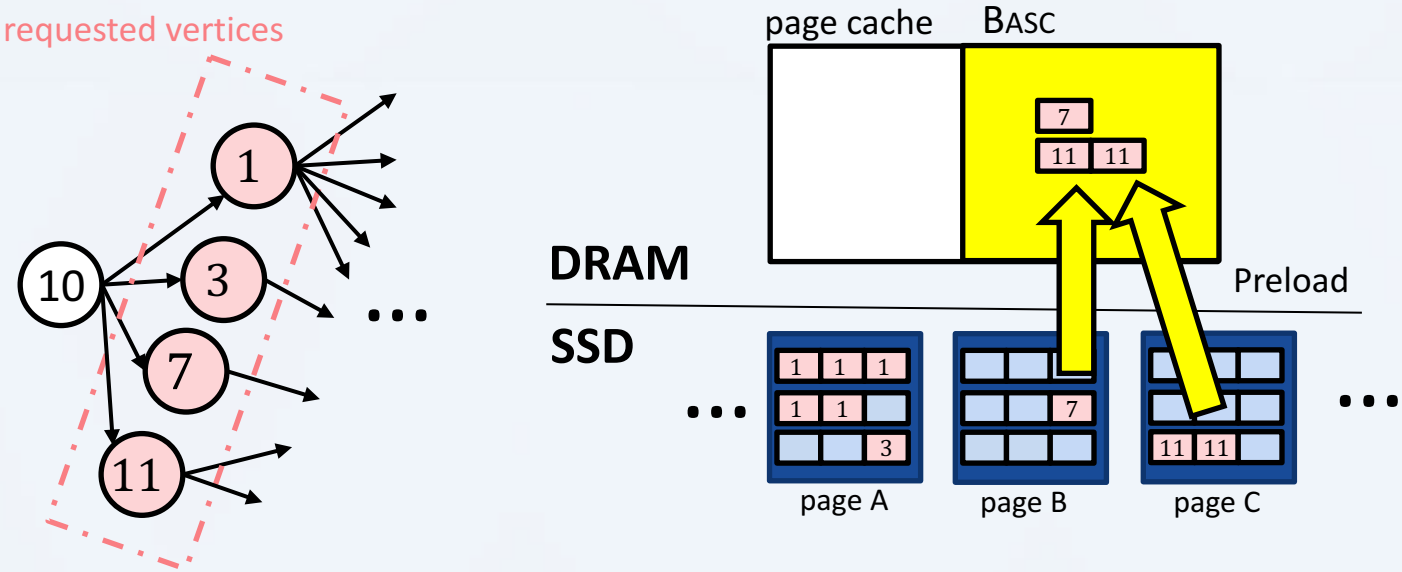
BASC: BFS-Aware Static Cache

- Keep separate cache for selected edge lists
- **Pre-loaded**: edge lists pre-selected through pre-analysis
- **Static**: contents of cache do not change



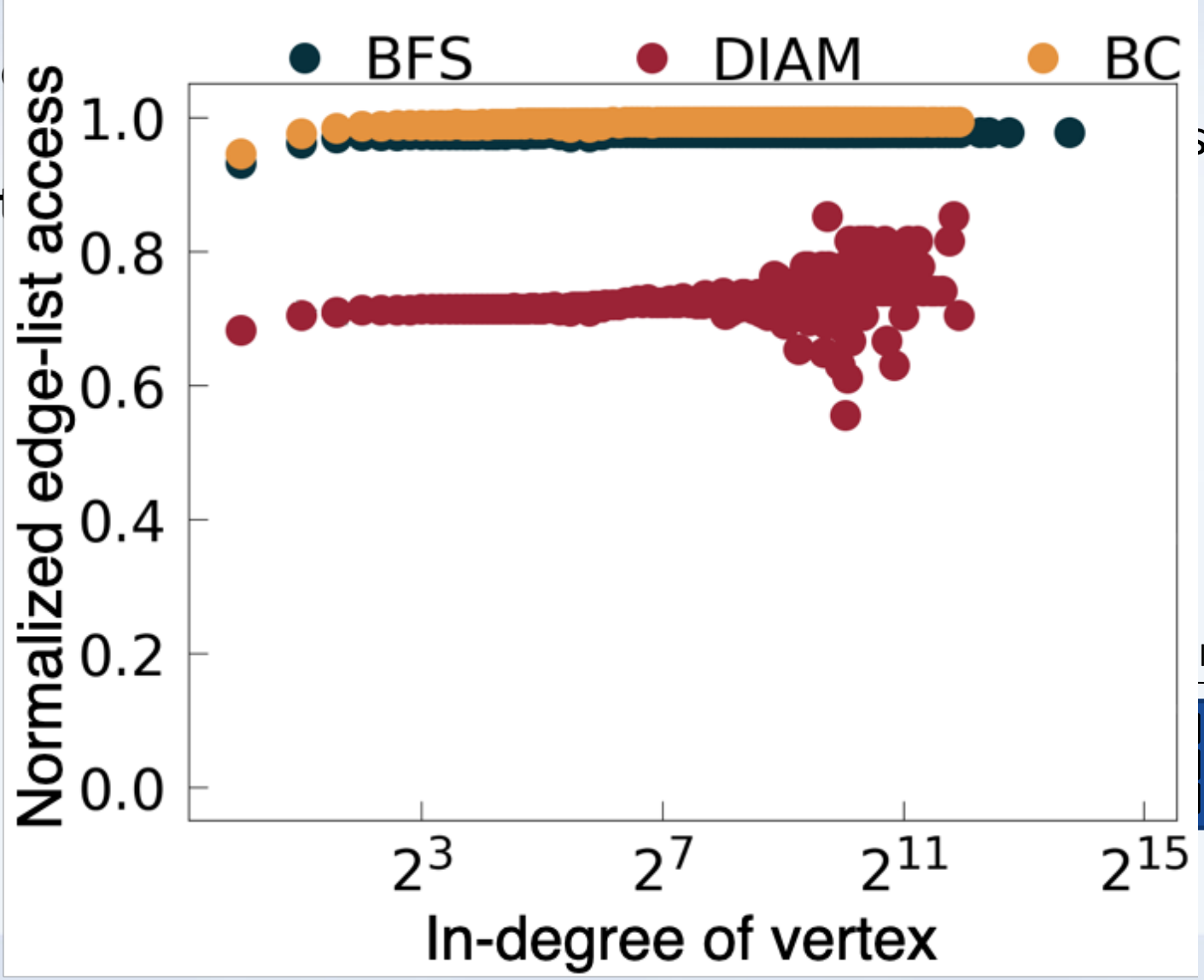
BASC: BFS-Aware Static Cache

- Keep separate cache for selected edge lists
- **Pre-loaded**: edge lists pre-selected through pre-analysis
- **Static**: contents of cache do not change



BASC: BFS-Aware Static Cache

- K
- P
- S



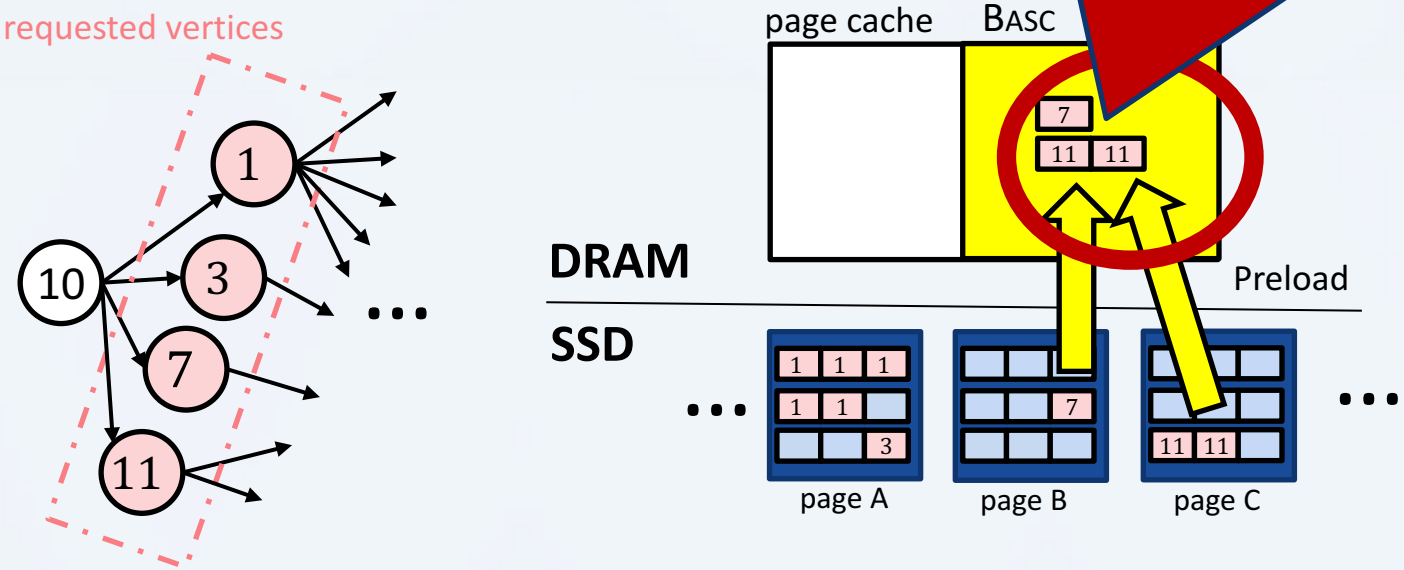
load

...

BASC: BFS-Aware Static Cache

- Keep separate cache for selected edge lists
- **Pre-loaded**: edge lists pre-selected through pre-analysis
- **Static**: contents of cache d

Which vertices to store in BASC?
Consider memory utilization
instead of frequency



Vertex Pre-Selection Problem

- **Find vertices to maximize memory utilization**
  = minimize utilization penalty

Find $v \in C$ **such that**

$$minimize\ F(C) = \sum_{\substack{\text{all edges(u,}v\text{)} \\ v \notin C}} \frac{\text{Prob(u} \rightarrow \text{v is traversed)}}{\text{Utilization}\big(\text{Page(v)},\ \text{neighbors(u)}\big)}$$

$$subject\ to \sum_{v \in C} \deg(v) \leq M\,, \sum_{v \in C} \deg(v) \geq M - \epsilon$$

Vertex Pre-Selection Problem

- **Find vertices to maximize memory utilization**

  = minimize utilization penalty

Find $v \in C$ **such that**

$$minimize\ F(C) = \sum_{\substack{\text{all edges(u,}v\text{)} \\ v \notin C}} \frac{\text{Prob(u} \rightarrow \text{v is traversed)}}{\text{Utilization}\big(\text{Page(v)},\ \text{neighbors(u)}\big)}$$

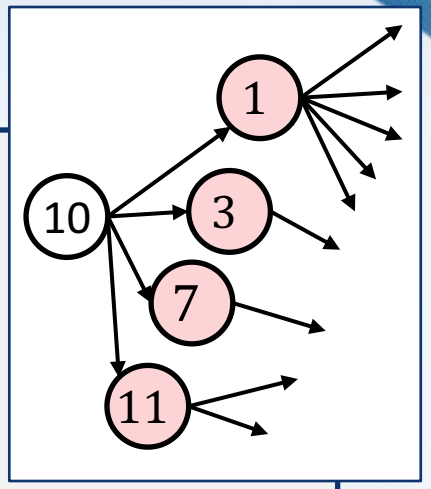
Assumption 1 The neighbor vertices of each vertex are accessed simultaneously. Thus, their edge lists are retrieved at the same time.

Assumption 2 The number of edge list requests for each vertex is equivalent among all the vertices.

Assumption 3 Each edge (u, v) probabilistically issues a request to access the edge list of target vertex v. Due to Assumption 2, the probability of issuing the request is inversely proportional to v's in-degree.

Vertex Pre-Selection Problem

- **Find vertices to maximize memory utilization**
  = minimize utilization penalty



Find $v \in C$ **such that**

$$minimize\ F(C) = \sum_{\substack{\text{all edges(u,}v\text{)} \\ v \notin C}} \frac{\text{Prob(u} \rightarrow \text{v is traversed)}}{\text{Utilization}\big(\text{Page(v)},\ \text{neighbors(u)}\big)}$$

Assumption 1 The neighbor vertices of each vertex are accessed simultaneously. Thus, their edge lists are retrieved at the same time.

Assumption 2 The number of edge list requests for each vertex is equivalent among all the vertices.

Assumption 3 Each edge (u,v) probabilistically issues a request to access the edge list of target vertex v. Due to Assumption 2, the probability of issuing the request is inversely proportional to v's in-degree.

Vertex Pre-Selection Problem

- **Find vertices to maximize memory utilization**
  = minimize utilization penalty

Find $v \in C$ **such that**

$$minimize\ F(C) = \sum_{\substack{\text{all edges(u,}v\text{)} \\ v \notin C}} \frac{\text{Prob(u} \rightarrow \text{v is traversed)}}{\text{Utilization}\big(\text{Page(v)},\ \text{neighbors(u)}\big)}$$

$$= \sum_{u \in V} \sum_{\substack{(u,v) \in E \\ v \notin C}} \left(\frac{1}{r(v) d_i(v)} \frac{1}{\sum_{\substack{(u,n) \in E \\ n \in P(v) \\ n \notin C}} d_o(n)} \right)$$

$$subject\ to \sum_{v \in C} \deg(v) \leq M, \sum_{v \in C} \deg(v) \geq M - \epsilon$$

$r(v)$: The number of vertices stored in the page(v)
$di(v)$: in-degree of v
$do(v)$: out-degree of v
$P(v)$: Set of vertices stored in page(v)

Vertex Pre-Selection Problem

- **Find vertices to maximize memory utilization**

  = minimize utilization penalty

Find $v \in C$ **such that**

$$minimize\ F(C) = \sum_{\substack{\text{all edges(u,}v\text{)} \\ v \notin C}} \frac{\text{Prob(u} \rightarrow \text{v is traversed)}}{\text{Utilization}\big(\text{Page(v)},\ \text{neighbors(u)}\big)}$$

$$= \sum_{u \in V} \sum_{\substack{(u,v) \in E \\ v \notin C}} \left(\frac{1}{r(v) d_i(v)} \frac{1}{\sum_{\substack{(u,n) \in E \\ n \in P(v) \\ n \notin C}} d_o(n)} \right)$$

$$subject\ to \sum_{v \in C} \deg(v) \leq M, \sum_{v \in C} \deg(v) \geq M - \epsilon$$

$r(v)$: The number of vertices stored in the page(v)
$di(v)$: in-degree of v
$do(v)$: out-degree of v
$P(v)$: Set of vertices stored in page(v)

➔ **NP-hard**

GVS: Greedy Vertex Selection

$$minimize\ F(C) = \sum_{\substack{\text{all edges(u,v)} \\ v \notin C}} \frac{\mathbf{Prob(u \rightarrow v\ is\ traversed)}}{\mathbf{Utilization\big(Page(v),\ \ neighbors(u)\big)}}$$

1. Ascribe the inner term *(Prob (…)/Util (…))* to target vertex v
2. Sort vertices by their *penalty(v)/cost(v)*
3. Greedily select top vertices that amount to $1/K$ of BASC
4. Repeat K times

* Refer to our paper for the full description of GVS

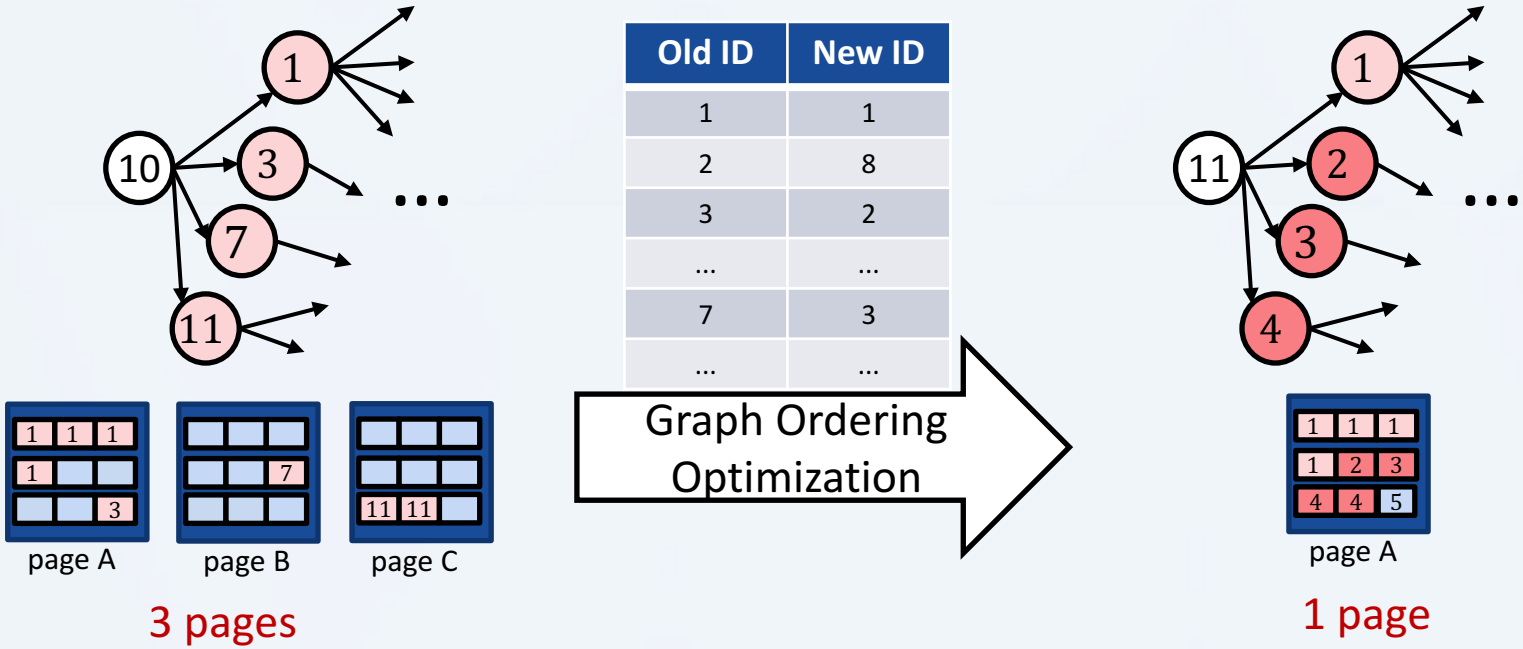
Our Optimization

1. **BFS-Aware Static Cache (*BASC*)**
   - Statically stores selected edge lists

2. **Neighborhood Ordering (*Norder*)**
   - Re-assigning vertex IDs to improve the locality of access

Graph Ordering

- **Preprocessing to re-assign vertex IDs to reduce I/O cost**



I/O Cost Model for BFS-like algorithms

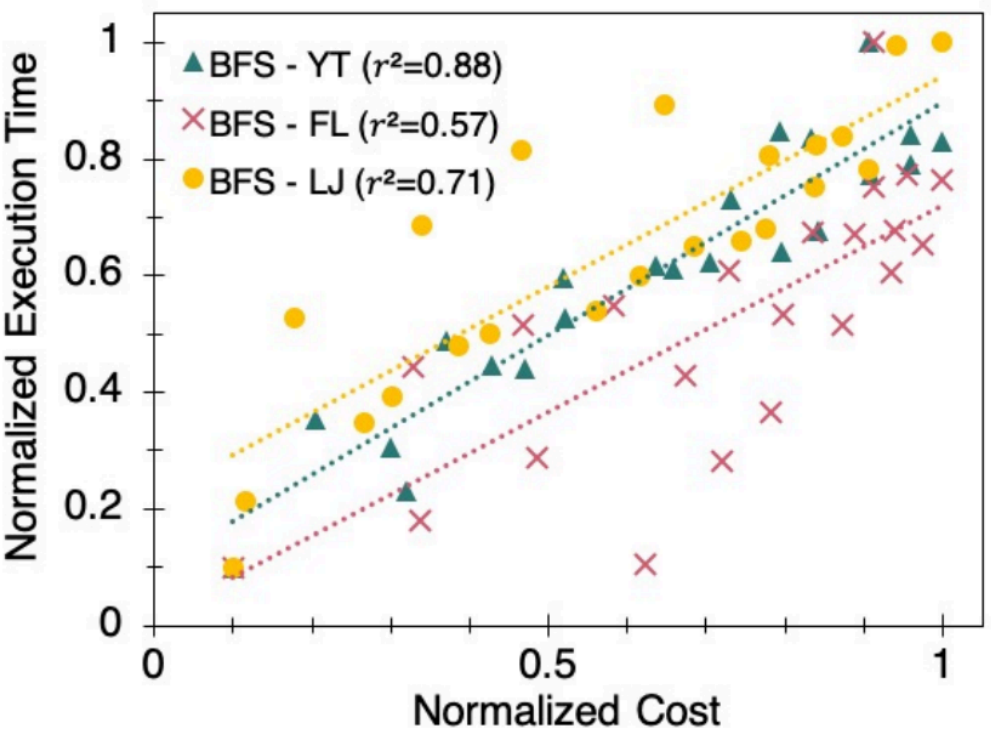
- **Heuristically devised the following I/O cost model**

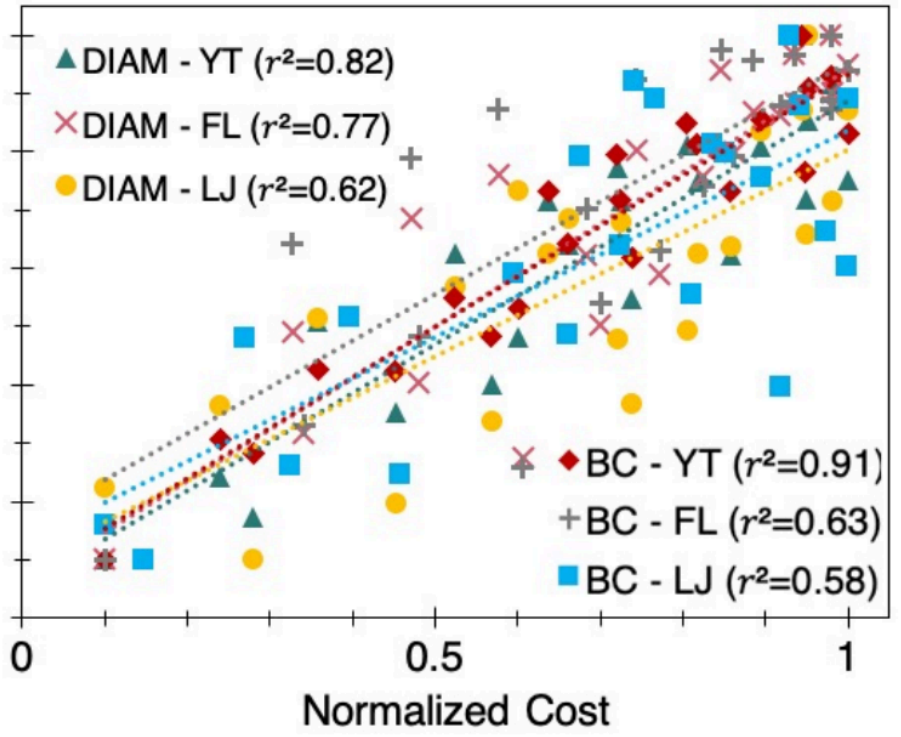
$$Cost = \sum_{v \in V} deg(v) \cdot \sigma^2(nbr(v))$$

- **Reasonably good at estimating the performance**

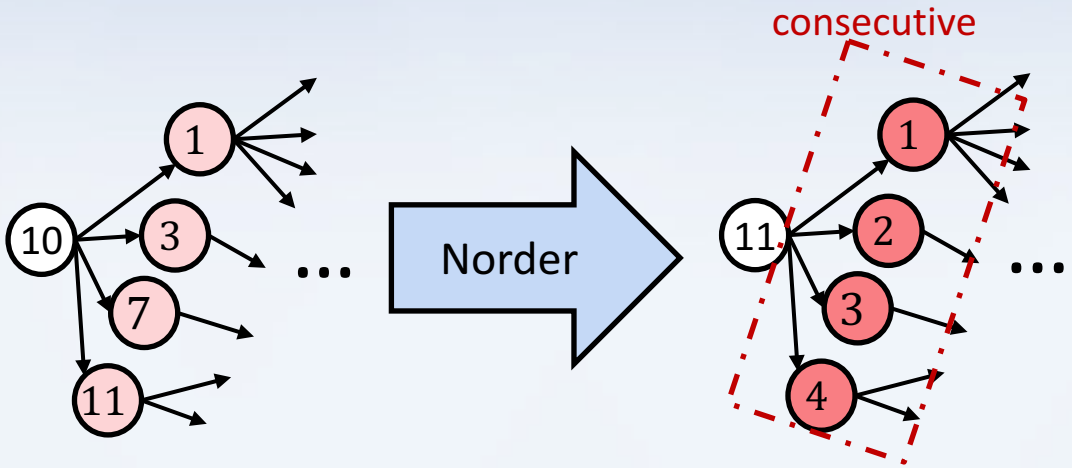
I/O Cost Model for BFS-like algorithms

- **Regression of Execution time on Cost model**





Norder: Neighborhood Ordering



- Assign consecutive IDs to neighbor vertices

Norder Algorithm

1. Sort vertices by their degrees
2. Run bounded BFS from highest-degree vertex and assign IDs
3. Repeat 2 for non-ID-assigned vertices

Evaluation Settings

- **Our optimizations implemented in**
  - FlashGraph [Da et. al, FAST15]
  - Graphene [Liu et. al, ATC17]

- **H/W spec.**
  - Intel Xeon E5-2683 v4 (10GB DRAM)
  - Intel 400GB SATA 3.0 SSD

- **Graph engine configuration**
  - 8 processing threads + 1 I/O thread
  - Page size: 8 KB by default
  - Cache size: 25% of input graph by default
    - Baseline: page cache only
    - BASC: 5% to page cache, 20% to BASC

Evaluation Settings

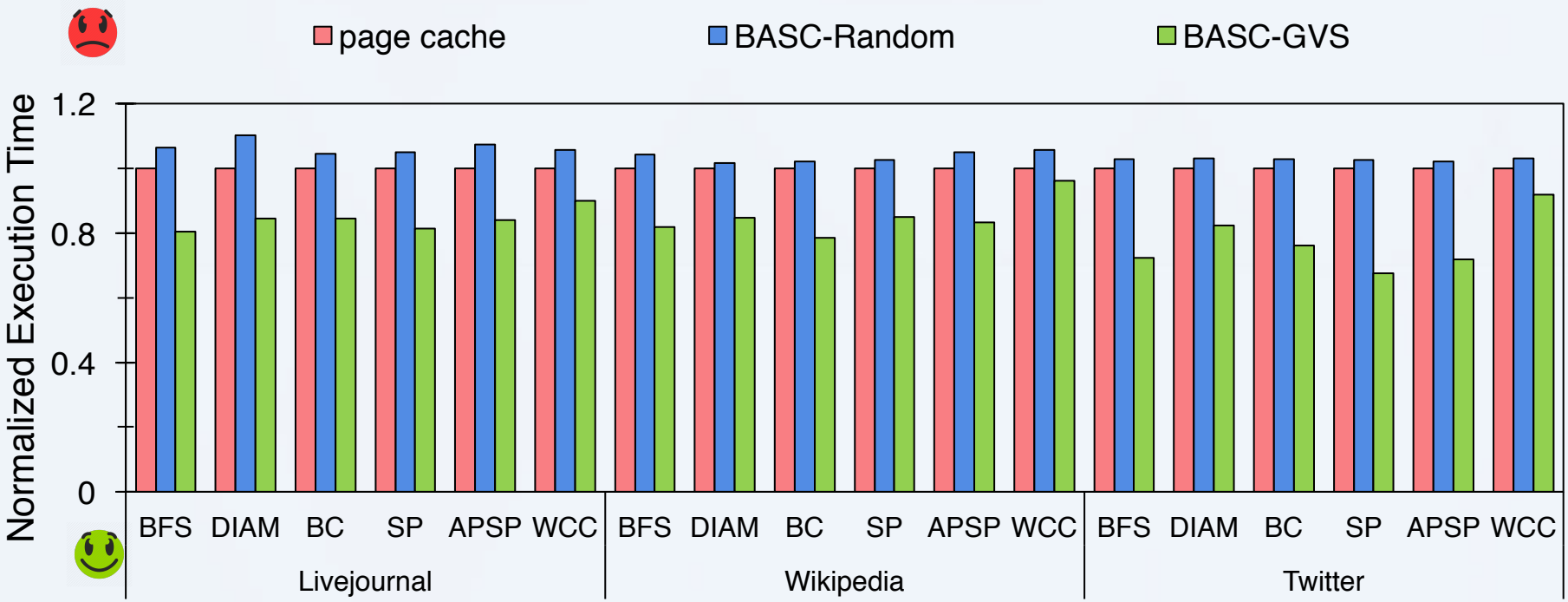
- **Graph algorithm**
  - BFS: Bread-first search
  - DIAM: Diameter estimation
  - BC: Betweenness centrality
  - SP: Shortest path
  - APSP: All pair shortest path
  - WCC: Weakly connected components

- **Data set**

Data set	Vertex	Edge
Youtube	3.2 million	9.4 million
Flickr	2.3 million	33 million
Livejournal	4.8 million	68 million
Wikipedia	18 million	172 million
Twitter	53 million	1.9 billion

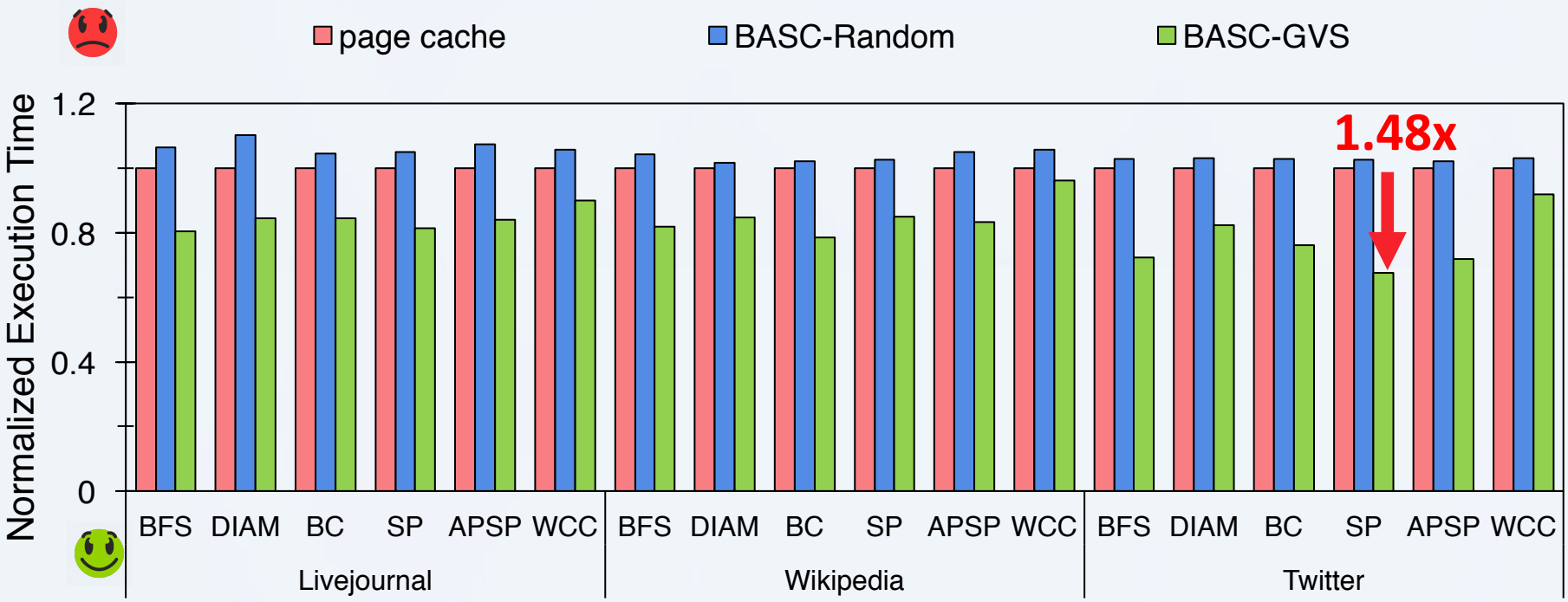
Evaluation of BASC

- Avg. 1.22x, Max. 1.48x speed-up compared to page cache
- Avg. 1.27x, Max. 1.52x speed-up compared to BASC-Random



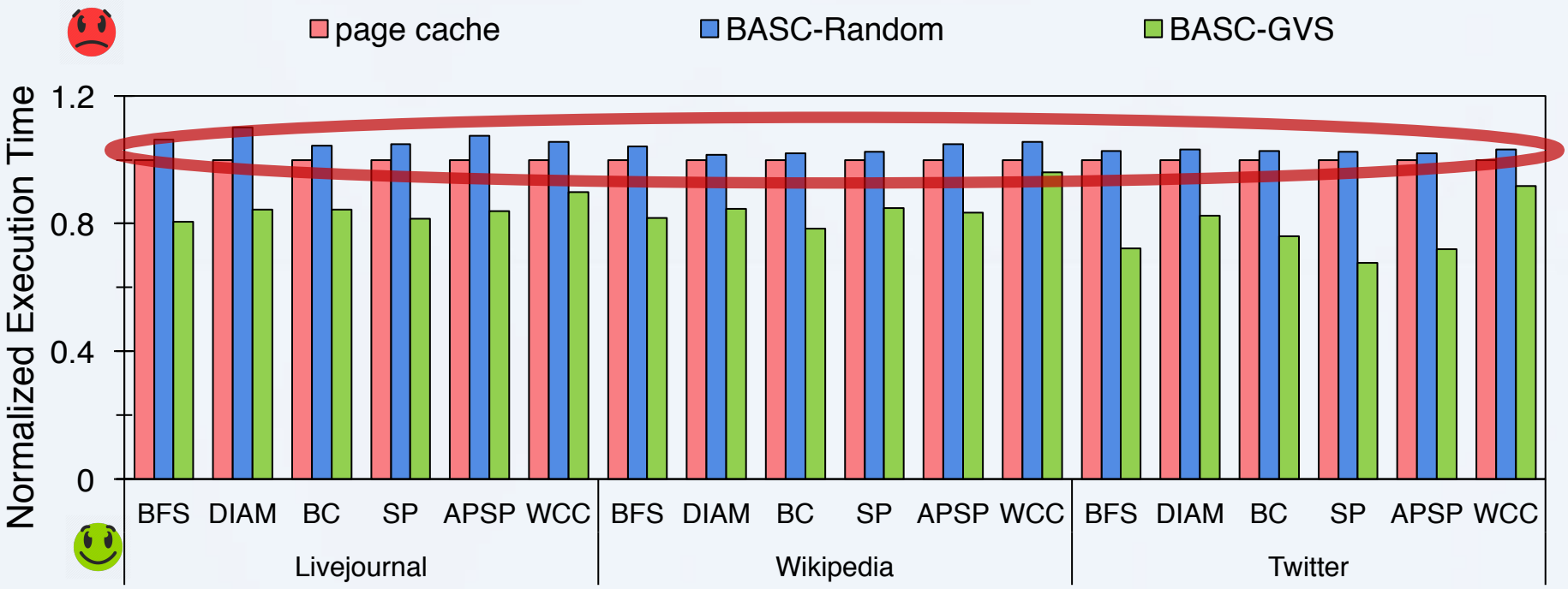
Evaluation of BASC

- Avg. 1.22x, Max. 1.48x speed-up compared to page cache
- Avg. 1.27x, Max. 1.52x speed-up compared to BASC-Random



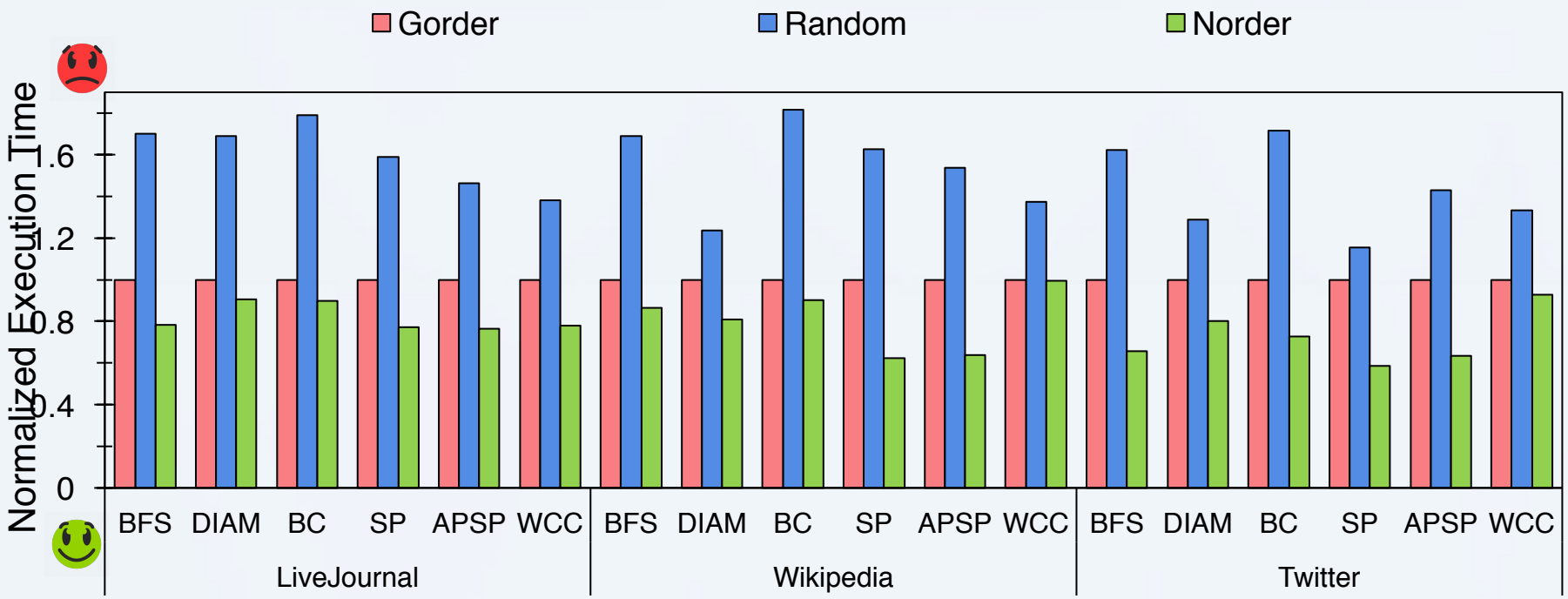
Evaluation of BASC

- Avg. 1.22x, Max. 1.48x speed-up compared to page cache
- Avg. 1.27x, Max. 1.52x speed-up compared to BASC-Random



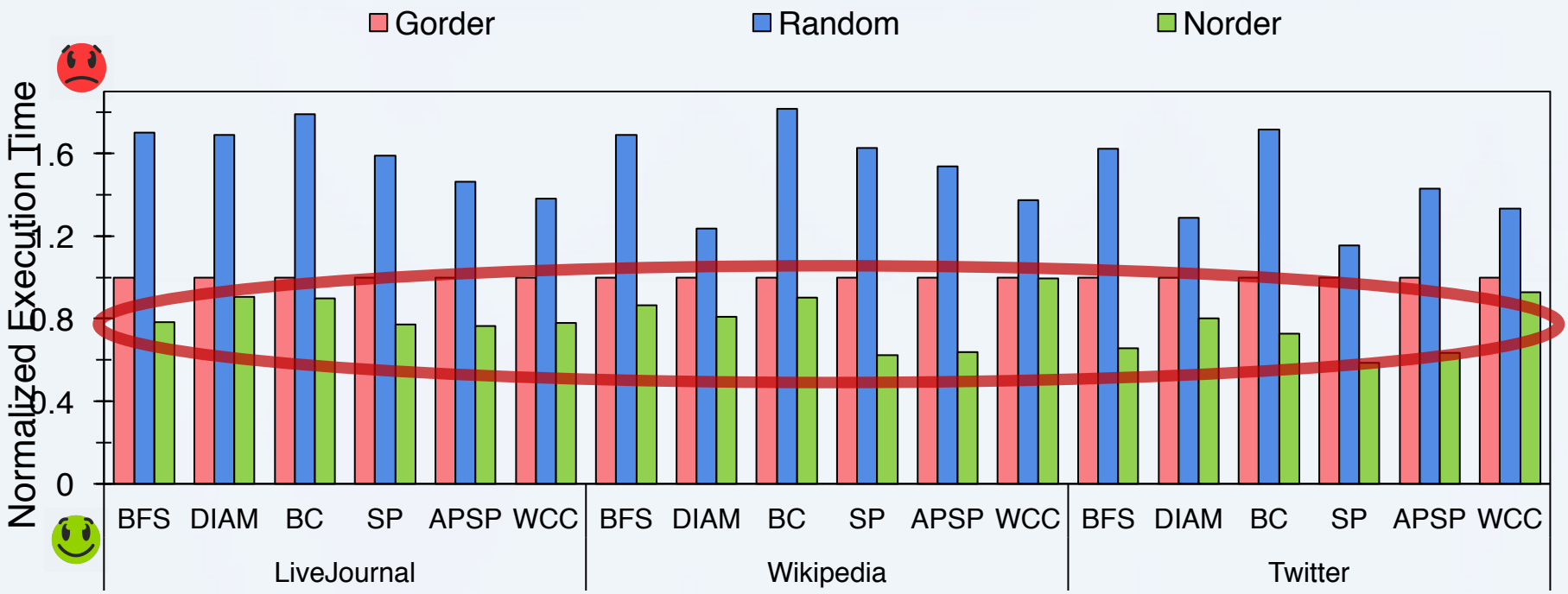
Evaluation of Norder

- Avg. 1.31x, Max. 1.71x speed-up compared to Gorder
- Avg. 1.92x, Max. 2.60x speed-up compared to Random



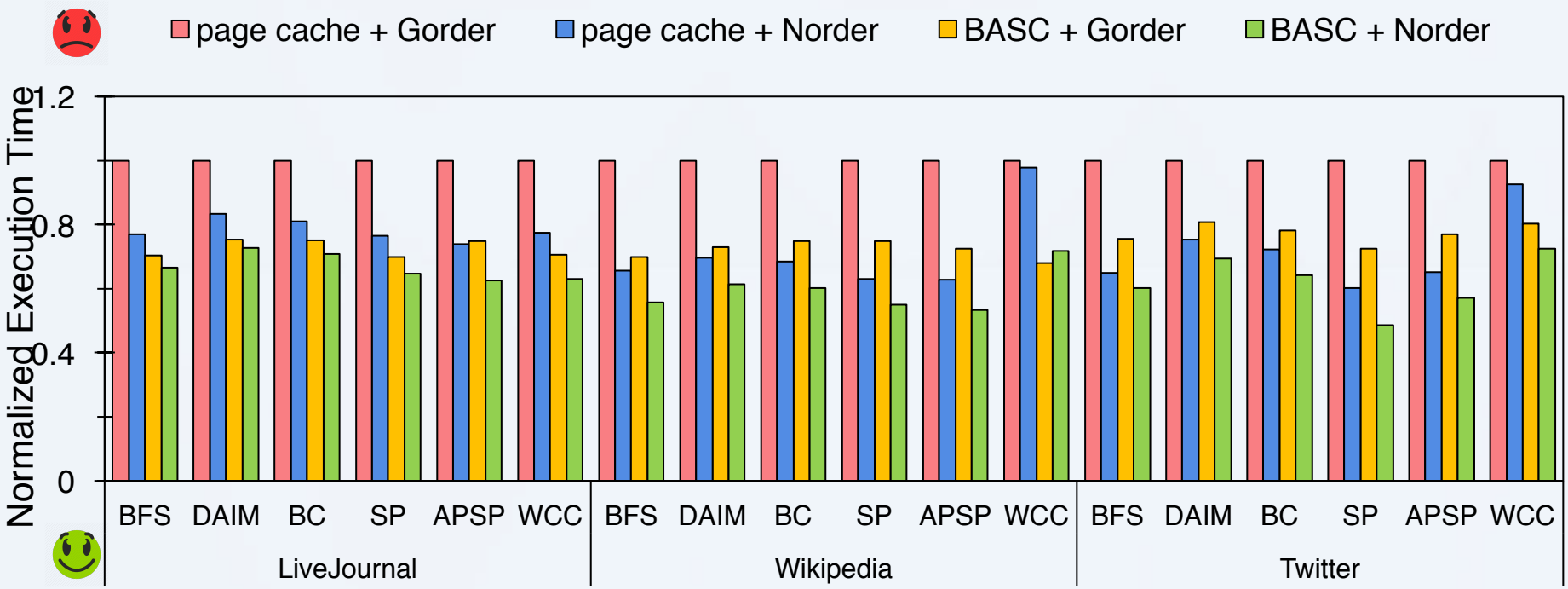
Evaluation of Norder

- Avg. 1.31x, Max. 1.71x speed-up compared to Gorder
- Avg. 1.92x, Max. 2.60x speed-up compared to Random



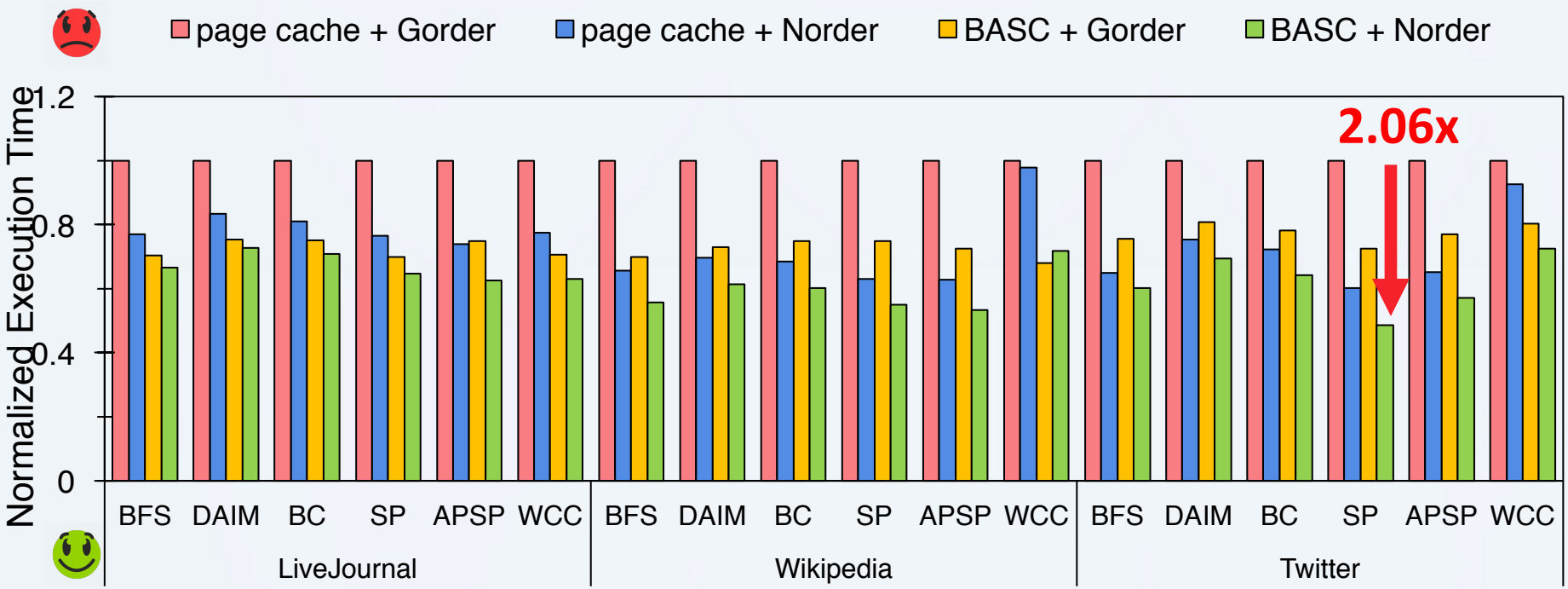
Evaluation of BASC + Norder

- Avg. 1.54x, Max. 2.06x speed-up compared to page cache + Gorder
- Avg. 1.17x, Max. 1.36x speed-up compared to page cache + Norder
- Avg. 1.18x, Max. 1.49x speed-up compared to BASC + Gorder

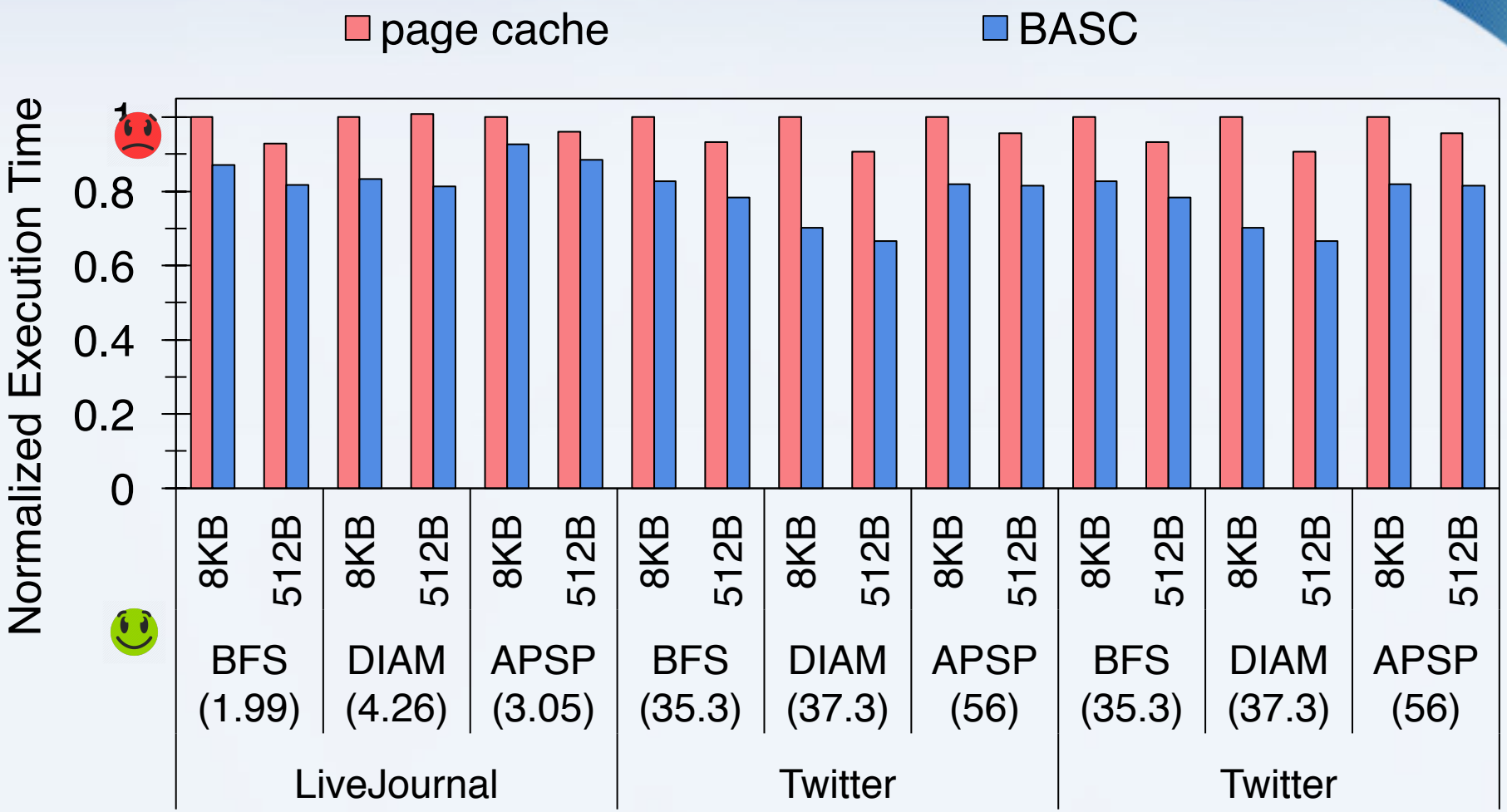


Evaluation of BASC + Norder

- Avg. 1.54x, Max. 2.06x speed-up compared to page cache + Gorder
- Avg. 1.17x, Max. 1.36x speed-up compared to page cache + Norder
- Avg. 1.18x, Max. 1.49x speed-up compared to BASC + Gorder



Efficiency of BASC w/ Graphene's small I/O requests



Preprocessing Overhead of Norder and GVS

- **Preprocessing time of Norder (in seconds)**

	YT	FL	LJ	WK	TW
Gorder	12.5	39.6	45.6	169.3	11687.1
Norder	2.0	2.7	7.2	16.9	243.5

Preprocessing Overhead of Norder and GVS

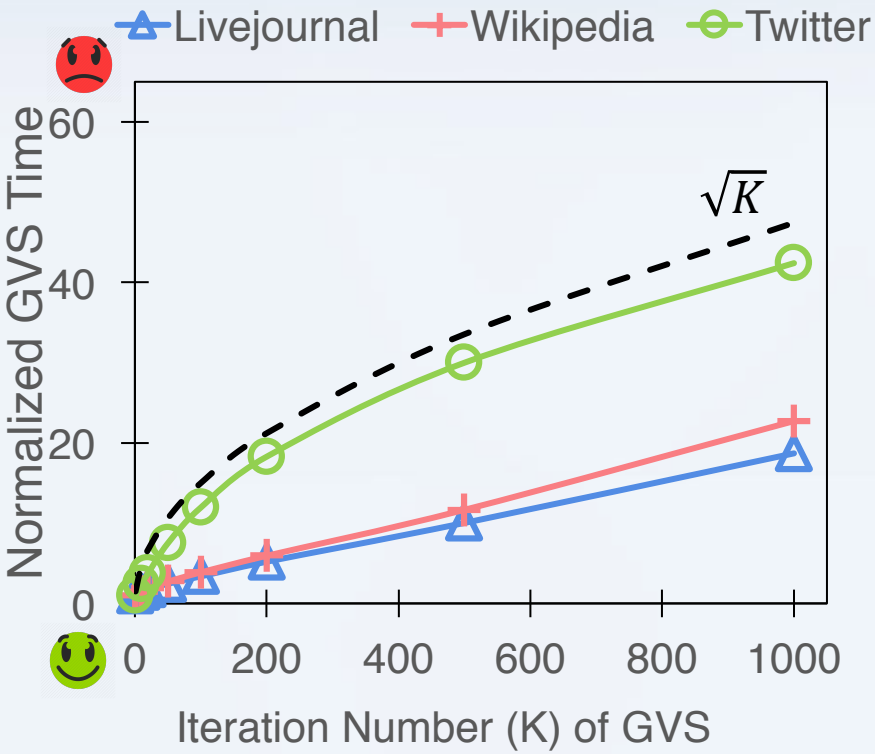
- **Preprocessing time of Norder (in seconds)**

	YT	FL	LJ	WK	TW
Gorder	12.5	39.6	45.6	169.3	11687.1
Norder	2.0	2.7	7.2	16.9	243.5

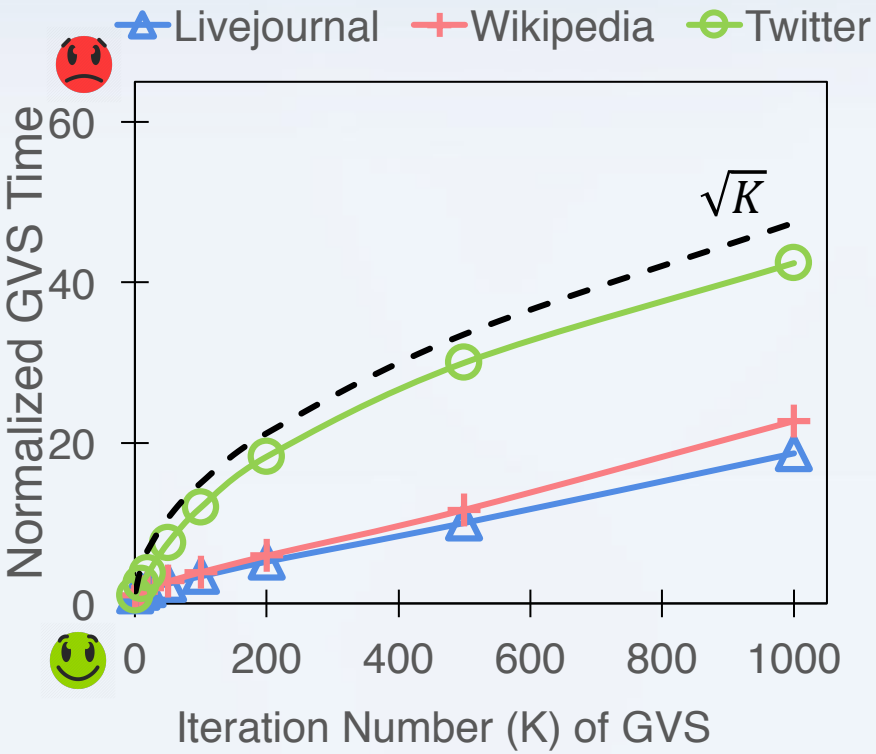
- **Preprocessing time of GVS (in seconds)**

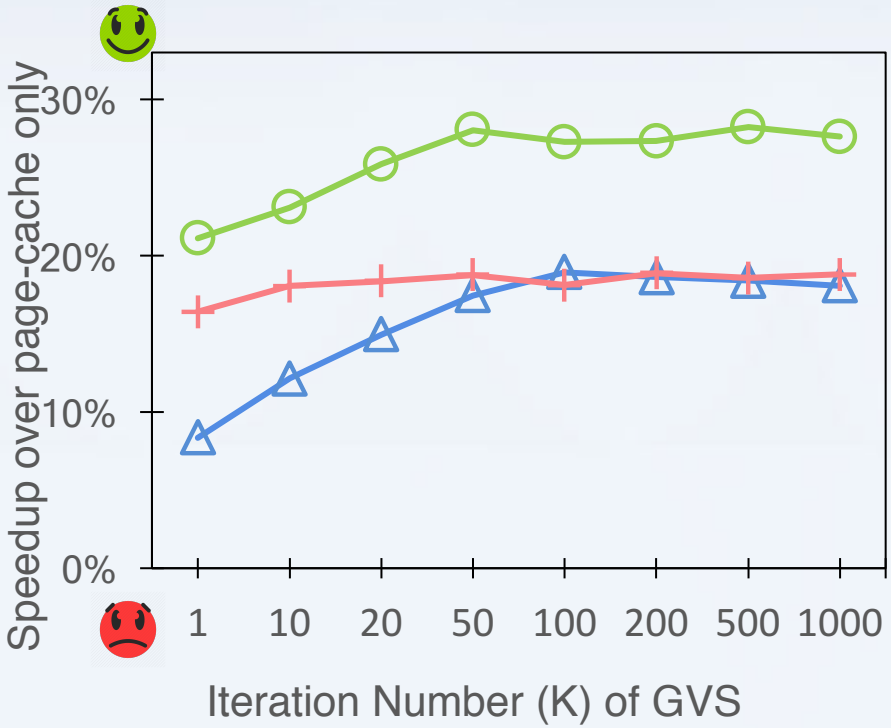
K	YT	FL	LJ	WK	TW
1	4.6	4.9	7.8	19.8	132
10	5.9	7.6	11.3	29.2	321
100	**16.7**	**24.7**	**26.3**	**76.4**	**1581**
1000	94.8	103	146	449	5612

GVS Execution Time w/ increasing K



Effectiveness of GVS w/ increasing K





When K=100, LJ=26.3 sec WK=76.4 sec, TW=26.4 min

More evaluation results in the paper

Conclusion

- **BFS-like algorithms on disk-based graph engine**
  - Uniform edge-list request
  - Page cache not effective
- ***BASC with GVS***
- ***Norder***
- **Evaluation**
  - Implementation on two graph engines
  - Experiments with six BFS-like algorithms
    five real-world graphs
  - 54% faster than existing schemes (up to 2.06 times faster)

Q&A

- seojiwon@gmail.com
- **Thank you**