

# LEA: A Lazy Eviction Algorithm for SSD Cache in Cloud Block Storage

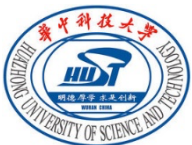
Ke Zhout<sup>†</sup>, Yu Zhang<sup>†</sup>, Ping Huang<sup>§</sup>, Hua Wang<sup>†\*</sup>, Yongguang Ji<sup>‡</sup>,  
Bin Cheng<sup>‡</sup>, Ying Liu<sup>‡</sup>

<sup>†</sup>Huazhong University of Science and Technology,  
<sup>†</sup>Key Laboratory of Information Storage System, Intelligent Cloud Storage  
Joint Research center of HUST and Tencent  
<sup>§</sup>Temple University, <sup>‡</sup>Tencent Technology (Shenzhen) Co., Ltd.

**Reporter : Yu Zhang**

**Intelligent Cloud Storage Joint Research center of HUST and Tencent**

**Tencent 腾讯**



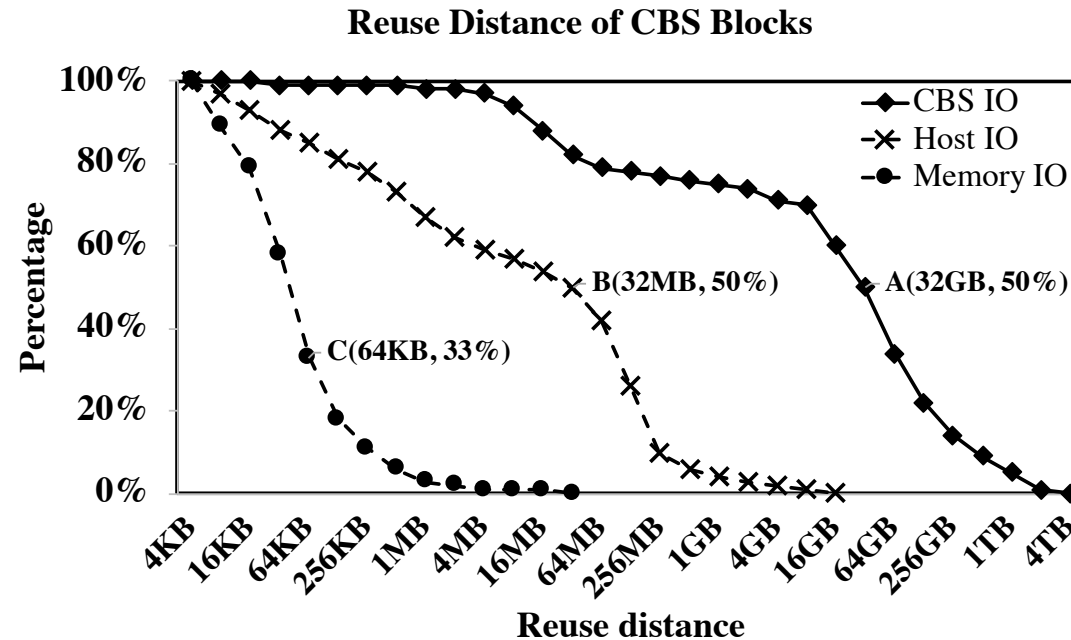
- **Background & Motivation**
- Design & Implementation
- Evaluation
- Conclusion



- Great Random IO Performance
- Nonvolatile Feature
- Shock Resistance
- Energy Conservation
- .....

- Solid State Drives have been widely used as a caching layer in cloud storages nowadays.

- **Existing advanced cache algorithms make replacement on each miss when the cache space is full.**
  - LRU, MRU, LFU, CLOCK and FIFO
  - LRU-K, EELRU, 2Q, MQ, ARC, LIRS and FRD
  - .....



- There is a great percentage of blocks with **large reuse distances** in a typical Cloud Block Storage (**CBS**).
- We also found similar phenomena in other scenarios.

- So, we propose a novel cache algorithm, **Lazy Eviction Algorithm (LEA)**, for CBS and other similar scenarios.
  - LEA does not make replacement by default.
  - Extending the residence time of blocks in cache largely.
  - Results show that LEA can **improve the hit ratio** and **reduce the number of write to SSD** as well.

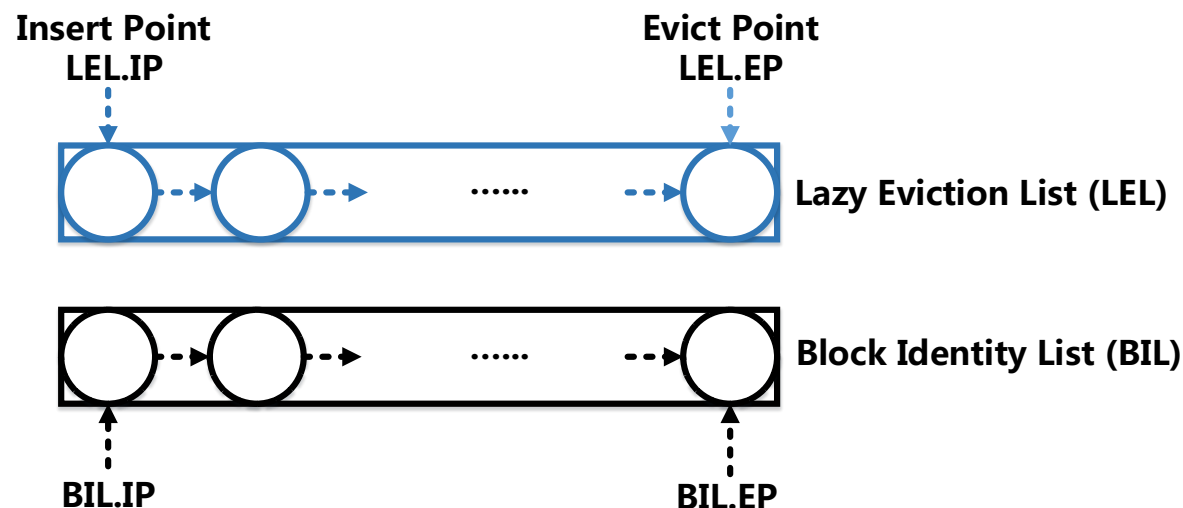
- Background & Motivation
- **Design & Implementation**
- Evaluation
- Conclusion

- **Summary Description of LEA**

- **Assuming that the currently accessed block is X**, LEA does not make replacement by default when there is a cache miss except when the lazy conditions are not met.
- The lazy conditions are defined by two aspects, considering both X and the candidate evicted-block. There are two lazy conditions used
  - **Lazy Condition 1 → The First Access of X**
    - Considering the candidate evicted-block in the cache.
  - **Lazy Condition 2 → The Second or More Times Accesses of X**
    - Much more difficult to satisfy than the previous one.

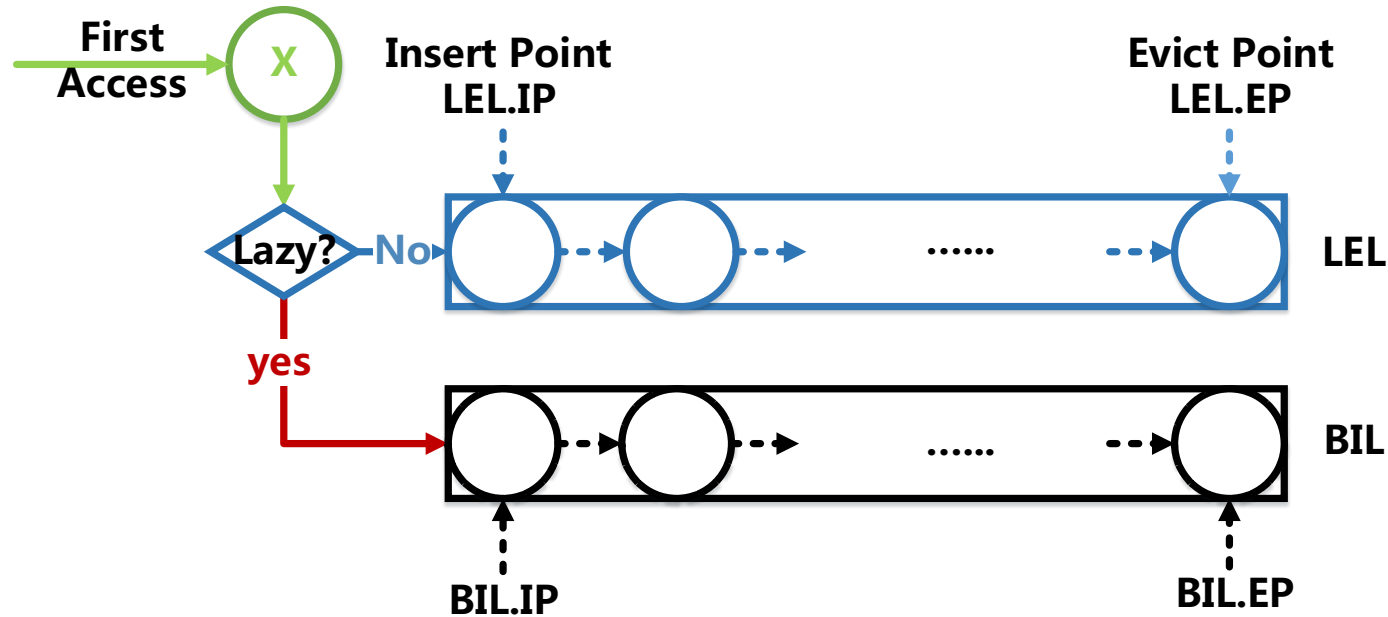


- Implementation of LEA





- Lazy Eviction List (**LEL**) and Block Identity List (**BIL**)
  - LEL is the cache list used to store **block entries**.
  - BIL is a ghost list used to store **identities of blocks**.
- Each list has two end points, i.e., Insertion Point (**IP**) and Eviction Point (**EP**)

- First Access of X

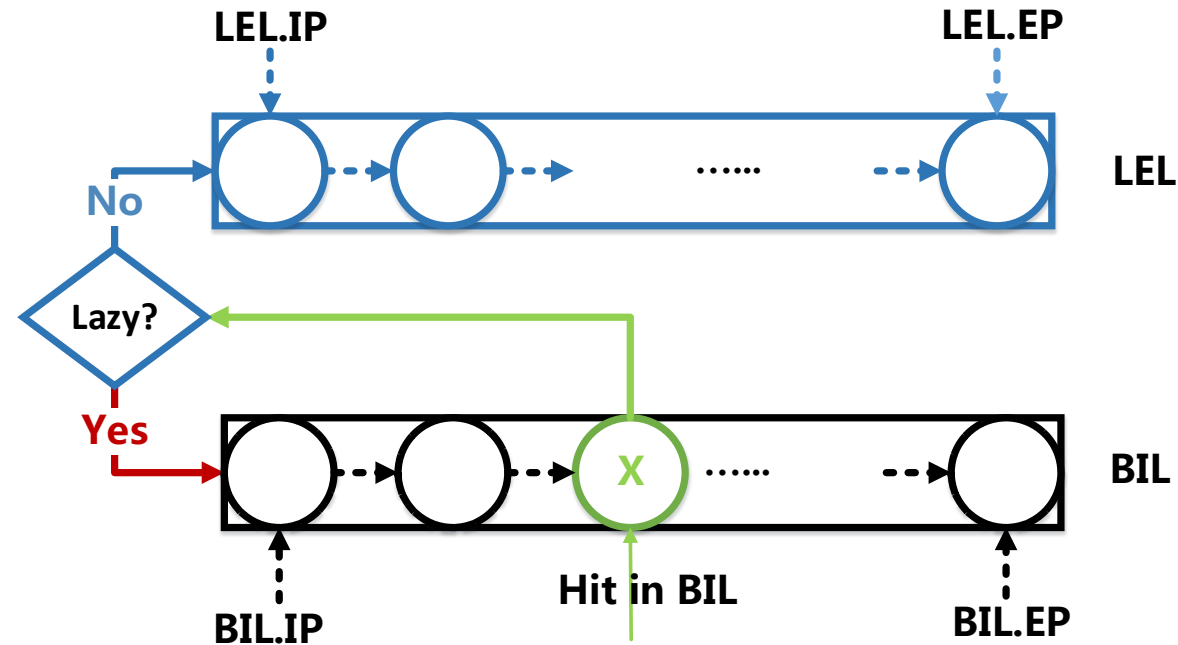


- When block X is accessed for the first time, it will insert the identity of X to BIL **if the first lazy condition is met**, otherwise it will insert the entry of X to LEL.

- Lazy Condition 1

- LEA defines a **flag** to denote the importance and usefulness of a candidate evicted-block for the cache.
  - If **flag** > 0, lazy condition is met! 
  - If **flag** == 0, lazy condition is not met! 
- The **flag** of a block is increased by 1 when the block is hit in the cache or reduced to the half when the block is a candidate evicted-block and the lazy conditions are met.

- Hit in BIL

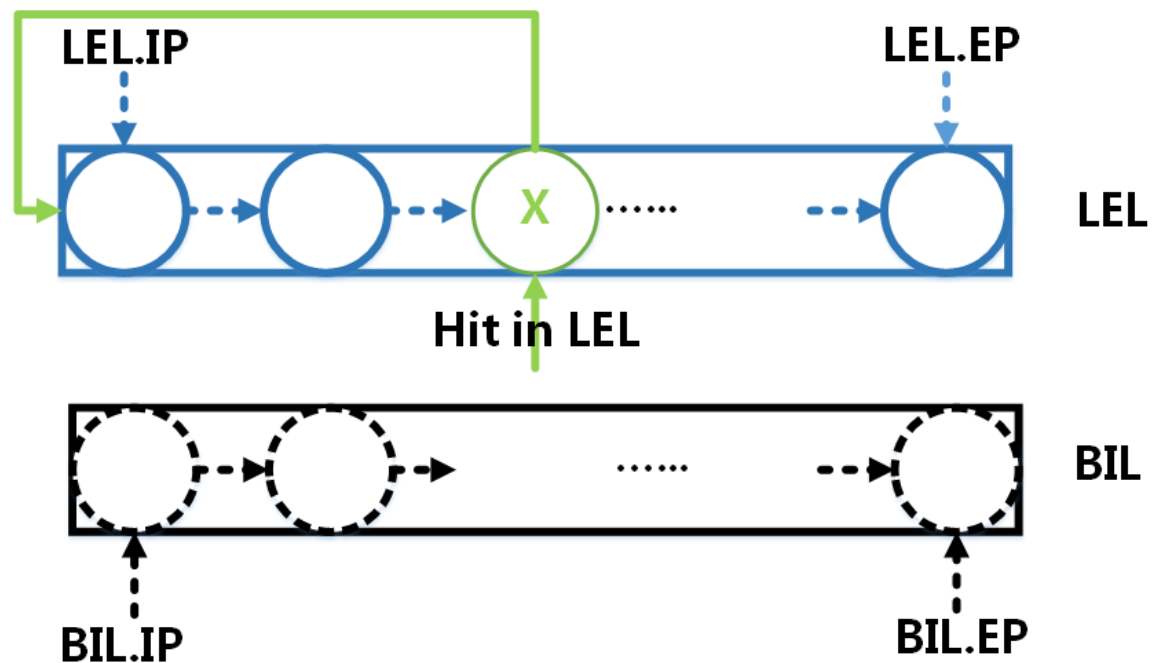


- When the re-reference to block X happens and x is hit in BIL, the processing is similar.

- Lazy Condition 2

- This lazy condition is much more difficult to satisfy than the previous one.
- To satisfy this lazy condition, the **flag** of a candidate evicted-block should be greater than 0, **more over** its age in the cache should be smaller than a threshold.
  - The threshold is **K** times of the block's average reuse distance.
  - From the experiments, we found that for most workloads, the best value of **K** is 1.

- Hit in LEL



- Move X to the Insertion Point of LEL list.

- Background & Motivation
- Design & Implementation
- **Evaluation**
- Conclusion

- **Experimental Method**

- We have evaluated the performance of LEA algorithm using a home-made trace-driven simulator [BlockCacheSim](#).
- We have compared LEA with other four algorithms, including LRU, ARC, LARC as well as OPT .
- Cache sizes are set as several reasonable values to ensure the hit ratio varies from relatively small to large.
- All the results are averages of multiple tests with different cache sizes.

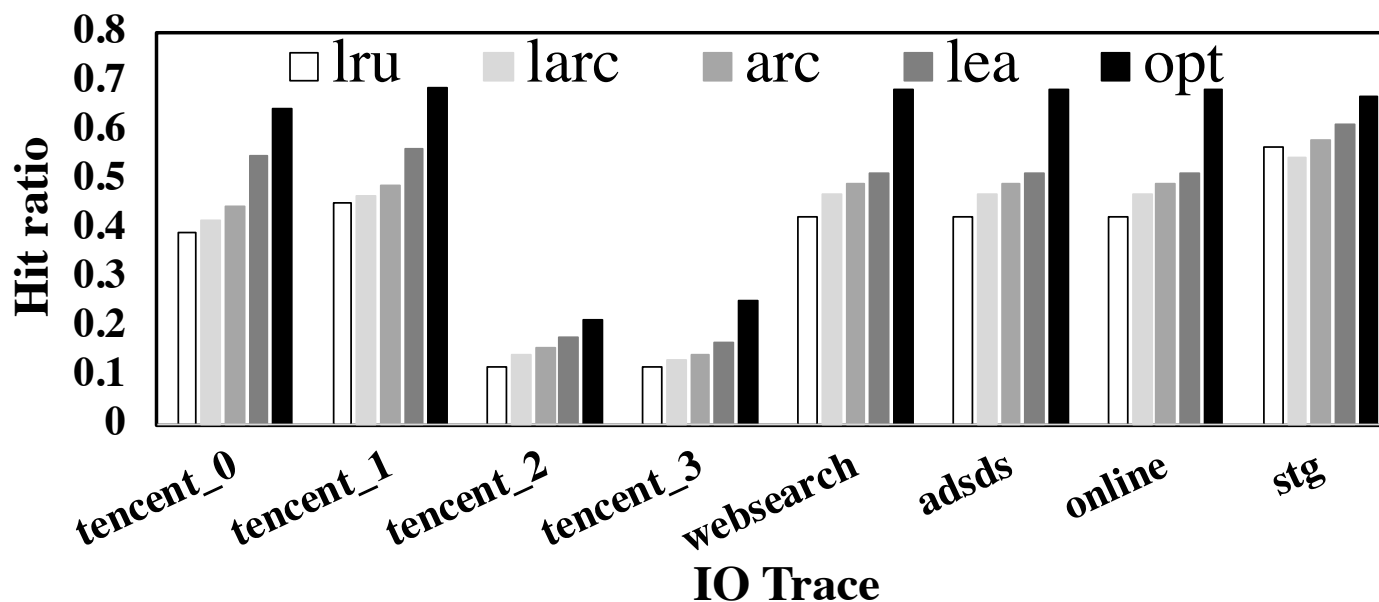


## • Trace Collection

- **IO traces collected from Tencent CBS ( *tencent\_1 to tencent\_4* ).**
  - IO traces from Tencent CBS have been sampled because the amount of original data is specifically large, reaching the scale of petabyte magnitude.
- **Several public IO traces ( *websearch, adsds, online and stg* ).**

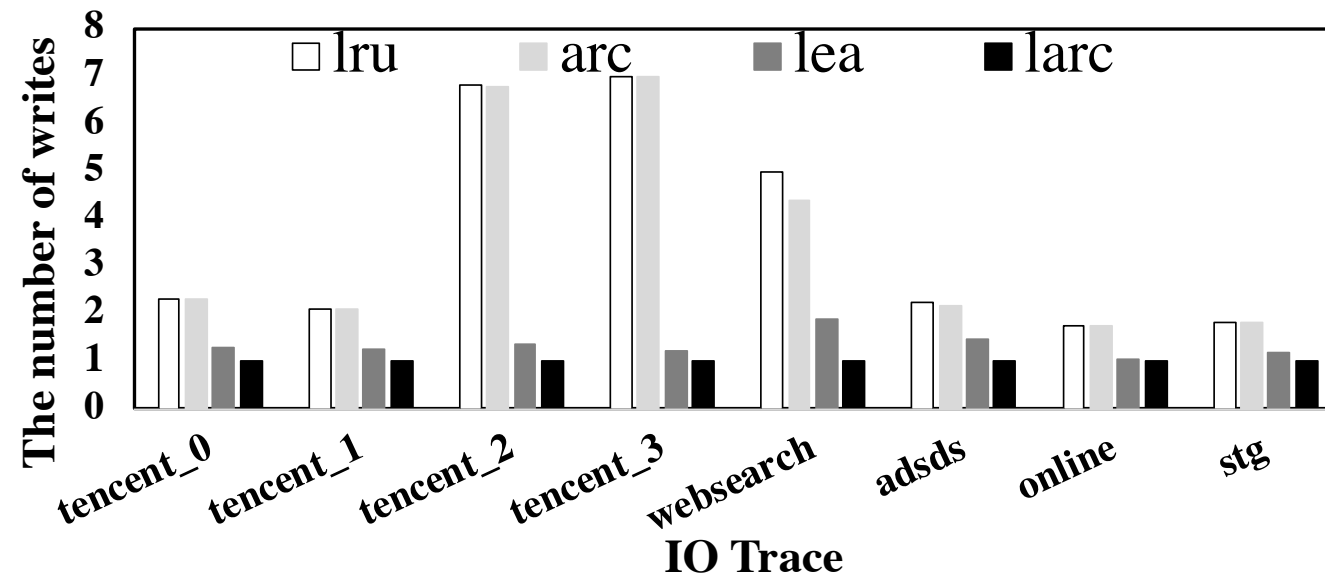
TRACE NAME	Total Traffic (GB)	Read Ratio	Description
tencent_0	149.91	13.95%	write-intensive
tencent_1	156.67	8.05%	write-intensive
tencent_2	108.48	68.44%	read-intensive
tencent_3	82.2	42.49%	read-write balanced
tencent_4	9972.14	19.72%	1 week trace from CBS
websearch	65.82	99.99%	web search
adsds	48.47	96.84%	display ads platform data server
online	54.53	21.80%	course management system of FIU
stg	23.26	31.76%	web staging

- Hit ratio



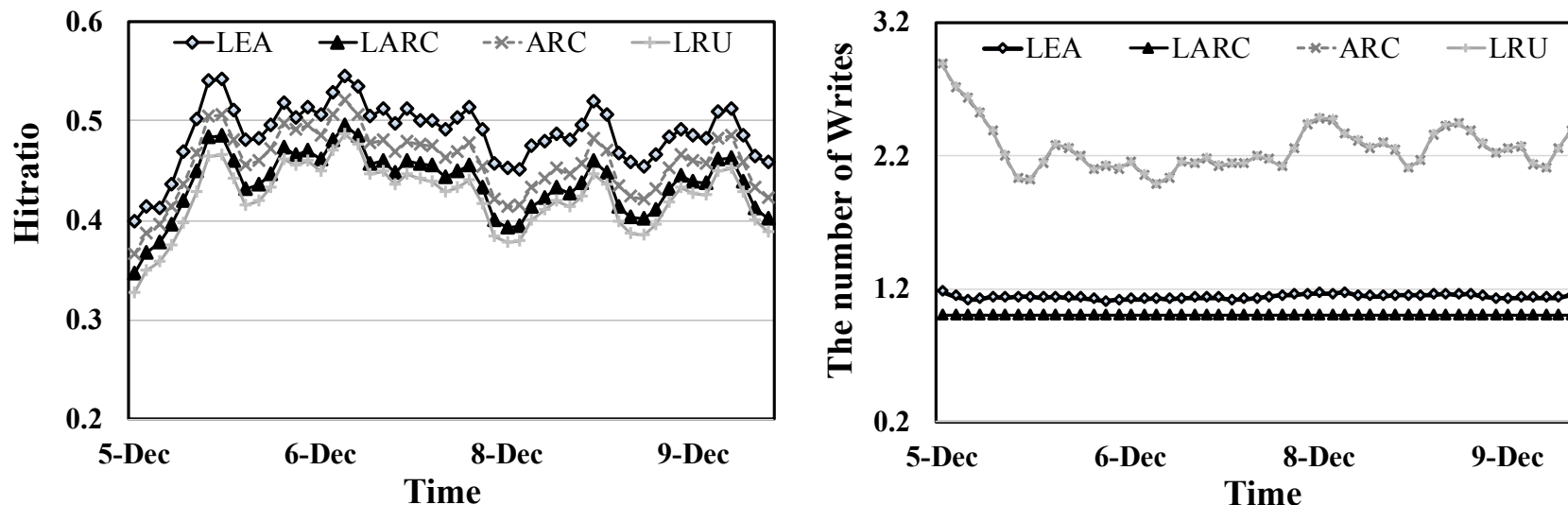
- On average, the hit ratio of LEA (45.13%) exceeds LRU (36.45%) by **23.80%**, LARC (39.00%) by **15.70%** and ARC (41.15%) by **9.66%**.

- The number of writes to SSD



- Numbers in this figure are normalized.
- LEA is of **62.85%** lower than LRU and of **61.85%** lower than ARC.
- LARC has the lowest write traffics, which is of **34.65%** lower than LEA. However this is achieved at the expense of hit ratio.

- The Consistence and Flexibility of LEA



- We use *tencent\_4* to test the consistence and Flexibility of LEA.
- On average, the hit ratio of LEA (47.63%) exceeds LRU (40.90%) by **16.45%**, LARC (42.39%) by **12.32%** and ARC (44.48%) by **7.08%**.
- The number of writes to SSD when using LEA is of **50.64%** lower than LRU and **50.21%** lower than ARC.

- Background & Motivation
- Design & Implementation
- Evaluation
- **Conclusion**

- We propose a novel cache algorithm LEA which is suitable for cloud block storages and other similar scenarios.
- LEA has very low overhead of complexity of  $O(1)$ .
- Experimental results show that LEA not only outperforms the state-of-the-art cache algorithms in hit ratio, but also reduces the number of writes to the SSD cache significantly.

**Thank You & Questions?**