Coupling Decentralized Key-Value Stores with Erasure Coding

Liangfeng Cheng[1], Yuchong Hu[1], **Patrick P. C. Lee[2]**

[1]Huazhong University of Science and Technology
[2]The Chinese University of Hong Kong

SoCC 2019

Introduction

- Decentralized key-value (KV) stores are widely deployed
  - Map each KV object deterministically to a node that stores the object via **hashing** in a decentralized manner (i.e., no centralized lookups)
  - e.g., Dynamo, Cassandra, Memcached

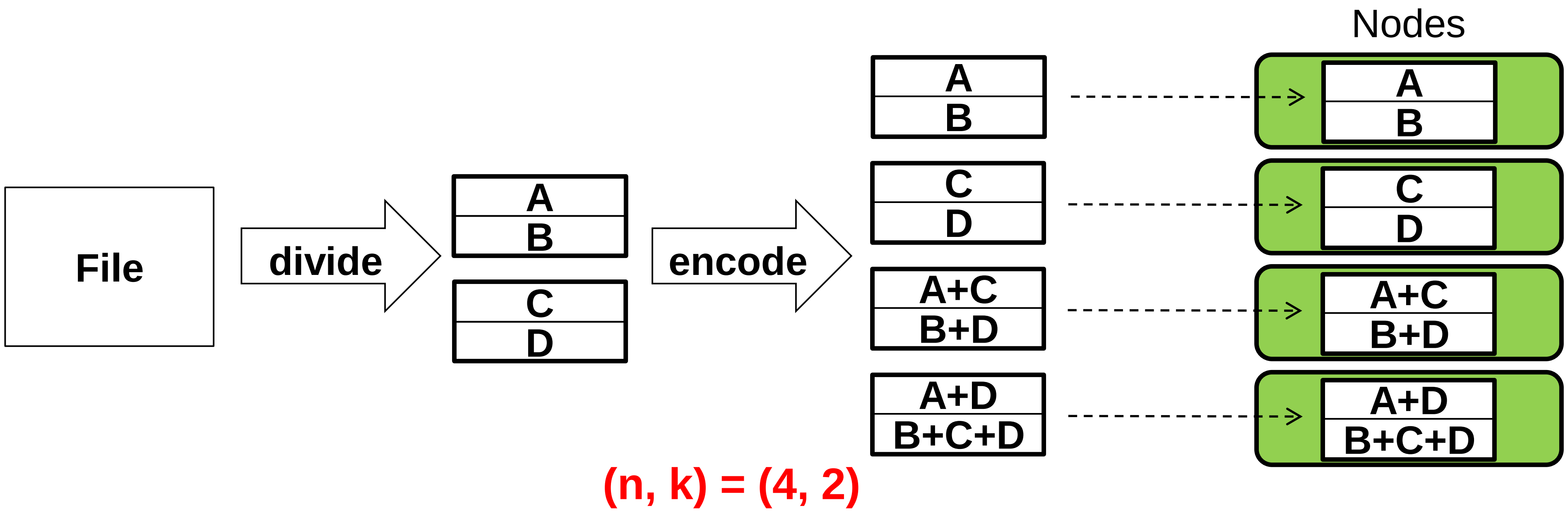
- Requirements
  - **Availability**: data remains accessible under failures
  - **Scalability**: nodes can be added or removed dynamically

Erasure Coding

- Replication is traditionally adopted for availability
  - e.g., Dynamo, Cassandra
  - Drawback: high redundancy overhead
- **Erasure coding** is a promising low-cost redundancy technique
  - Minimum data redundancy via “data encoding”
  - Higher reliability with same storage redundancy than replication
  - e.g., Azure reduces redundancy from 3x (replication) to 1.33x (erasure coding) → PBs saving
- How to apply erasure coding in decentralized KV stores?

Erasure Coding

- Divide file data to **k** equal-size data chunks
- Encode k data chunks to **n-k** equal-size parity chunks
- Distribute the n erasure-coded chunks (stripe) to n nodes
- **Fault-tolerance**: any k out of n chunks can recover file data



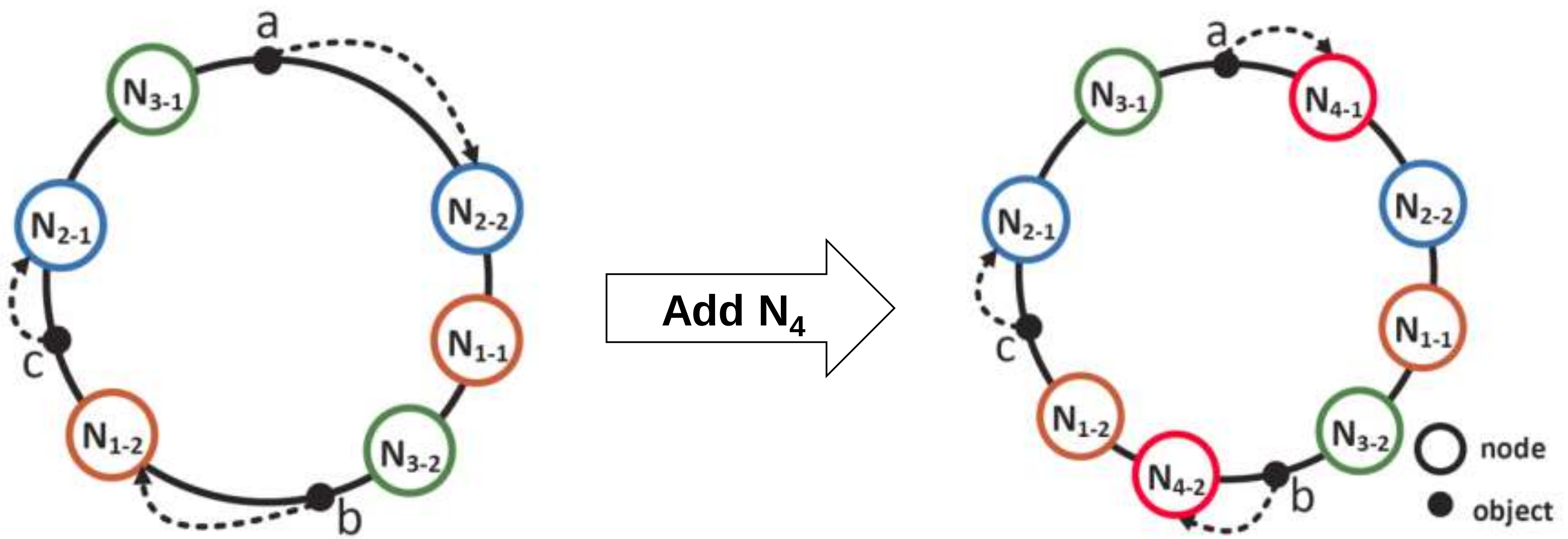
Erasure Coding

- Two coding approaches
  - **Self-coding**: divides an object into data chunks
  - **Cross-coding**: combines multiple objects into a data chunk

- Cross-coding is more appropriate for decentralized KV stores
  - Suitable for small objects
    - e.g., small objects dominate in practical KV workloads [Sigmetrics'12]
  - Direct access to objects

Scalability

- Scaling is a frequent operation for storage elasticity
  - **Scale-out** (add nodes) and **scale-in** (remove nodes)
- **Consistent hashing**
  - Efficient, deterministic object-to-node mapping scheme
  - A node is mapped to multiple **virtual nodes** on a **hash ring** for load balancing

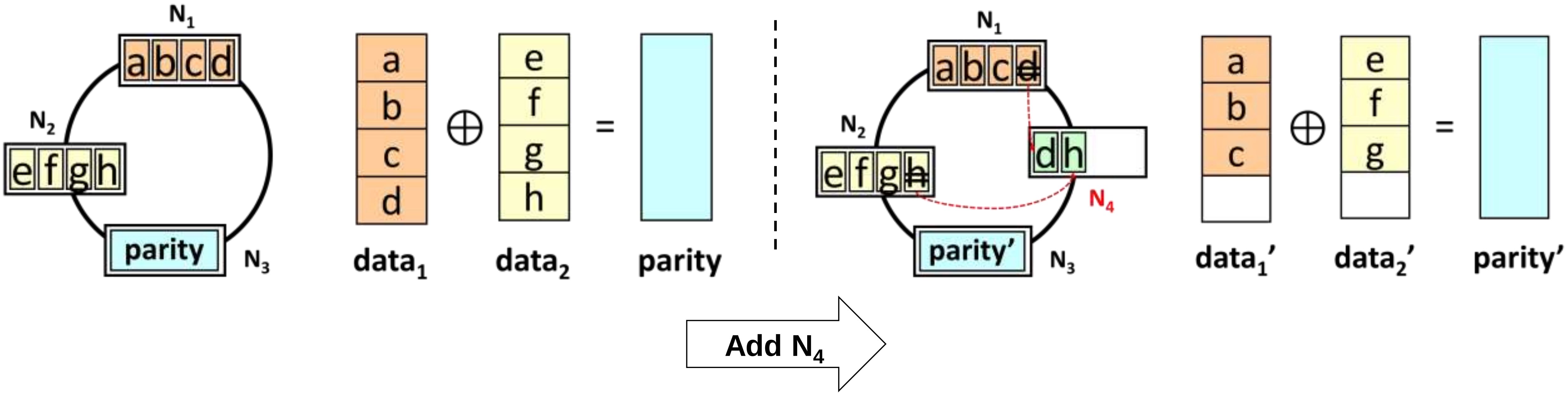


Scalability Challenges

- Replication / self-coding for consistent hashing
  - Replicas / coded chunks are stored after first node in clockwise direction

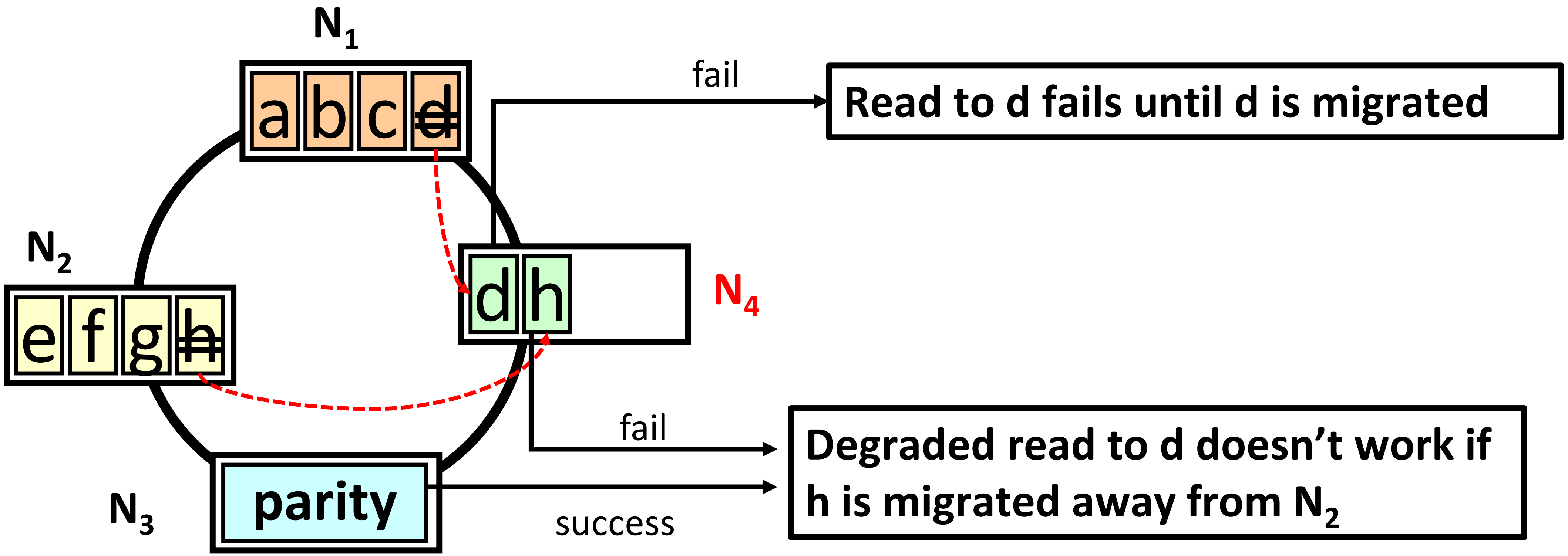
- Cross-coding + consistent hashing?
  - **Parity updates**
  - **Impaired degraded reads**

Challenge 1



- Data chunk updates → parity chunk update
- Frequent scaling → huge amount of data transfers (**scaling traffic**)

Challenge 2



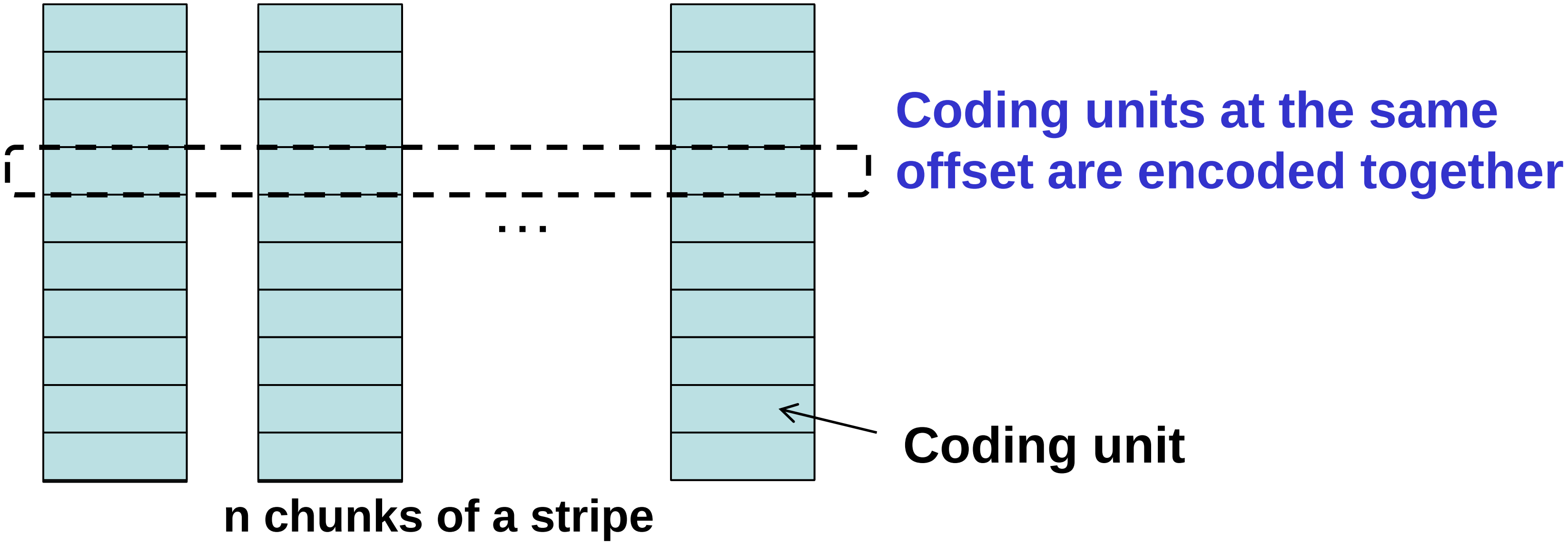
- ➢ Coordinating object migration and parity updates is challenging due to changes of multiple chunks
- ➢ Degraded reads are impaired if objects are in middle of migration

Contributions

- New erasure coding model: **FragEC**
  - Fragmented chunks → no parity updates
- Consistent hashing on **multiple hash rings**
  - Efficient degraded reads
- **Fragmented node-repair** for fast recovery
- **ECHash** prototype built on memcached
  - Scaling throughput: 8.3x (local) and 5.2x (AWS)
  - Degraded read latency reduction: 81.1% (local) and 89.0% (AWS)

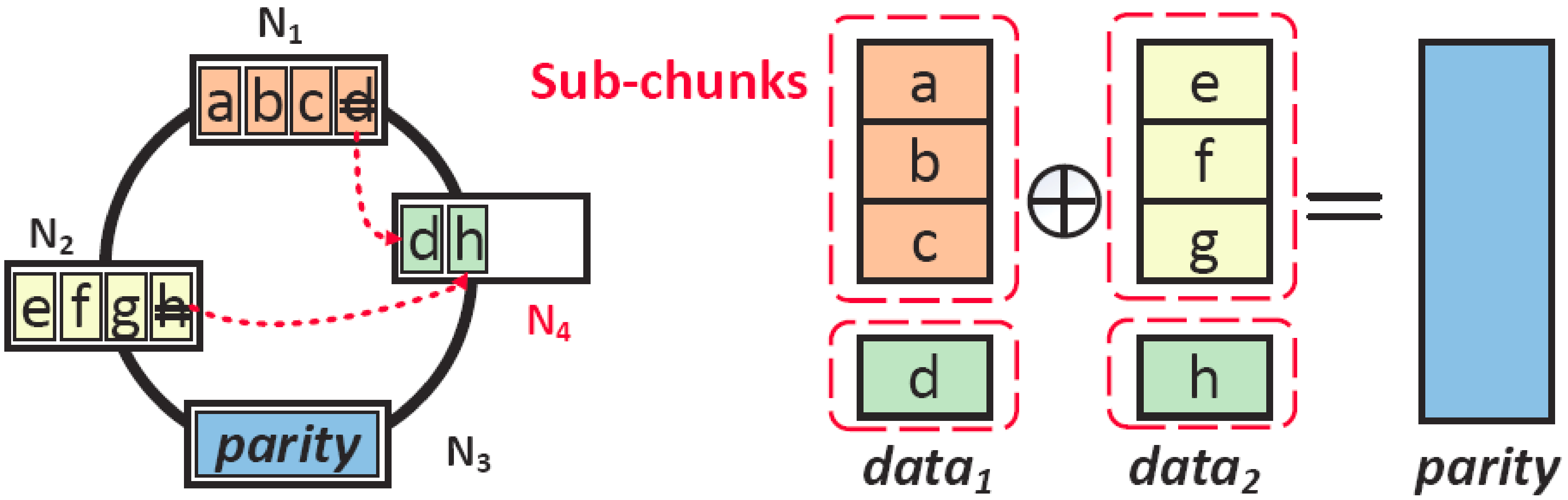
Insight

➢ A coding unit is much smaller than a chunk
- e.g., coding unit size ~ 1 byte; chunk size ~ 4 KiB
- Coding units at the same offset are encoded together in erasure coding



“Repair pipelining for erasure-coded storage”, USENIX ATC 2017

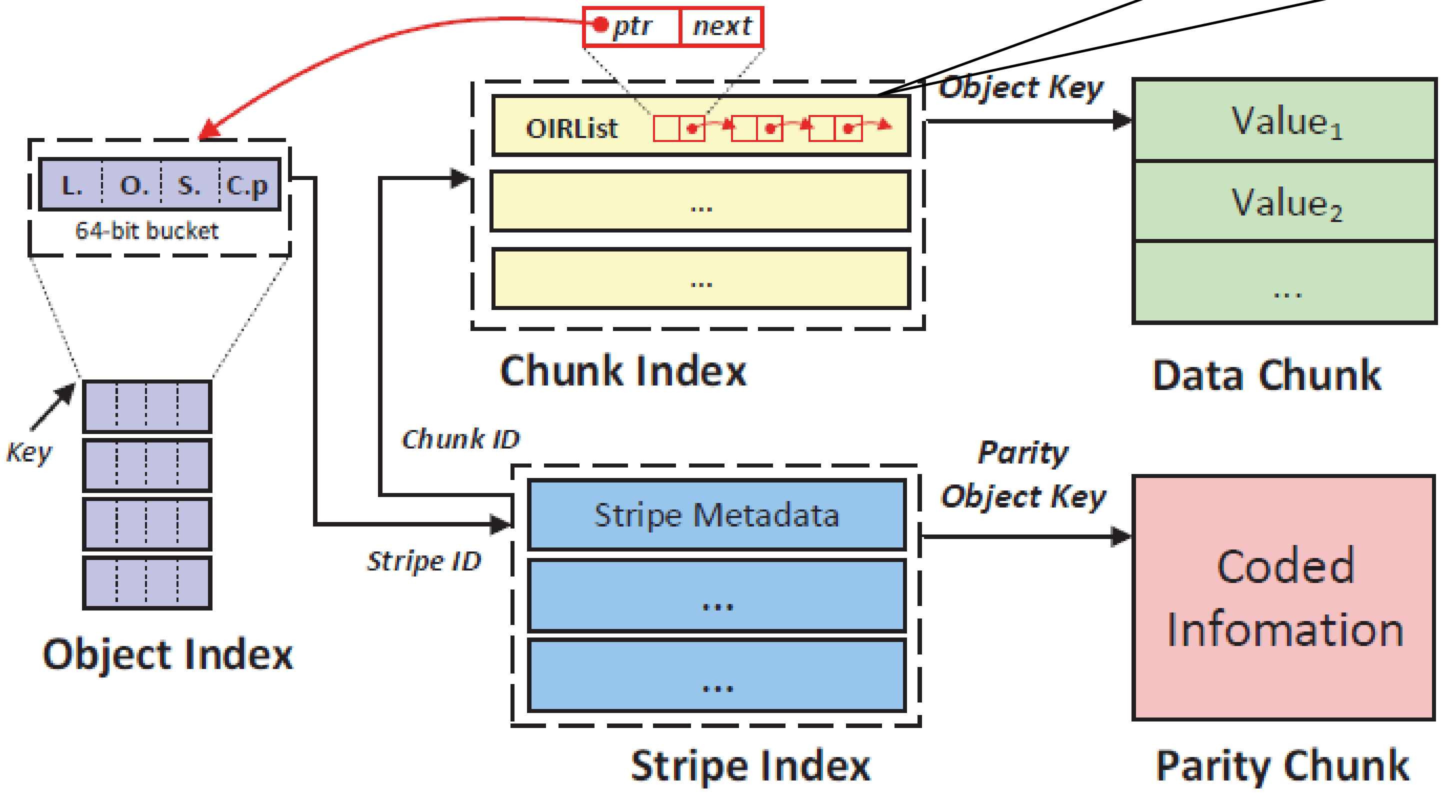
FragEC



- Allow mapping a data chunk to multiple nodes
  - Each data chunk is fragmented to sub-chunks
- Decoupling tight chunk-to-node mappings → no parity updates

FragEC

OIRList lists all object references and offsets in each data chunk

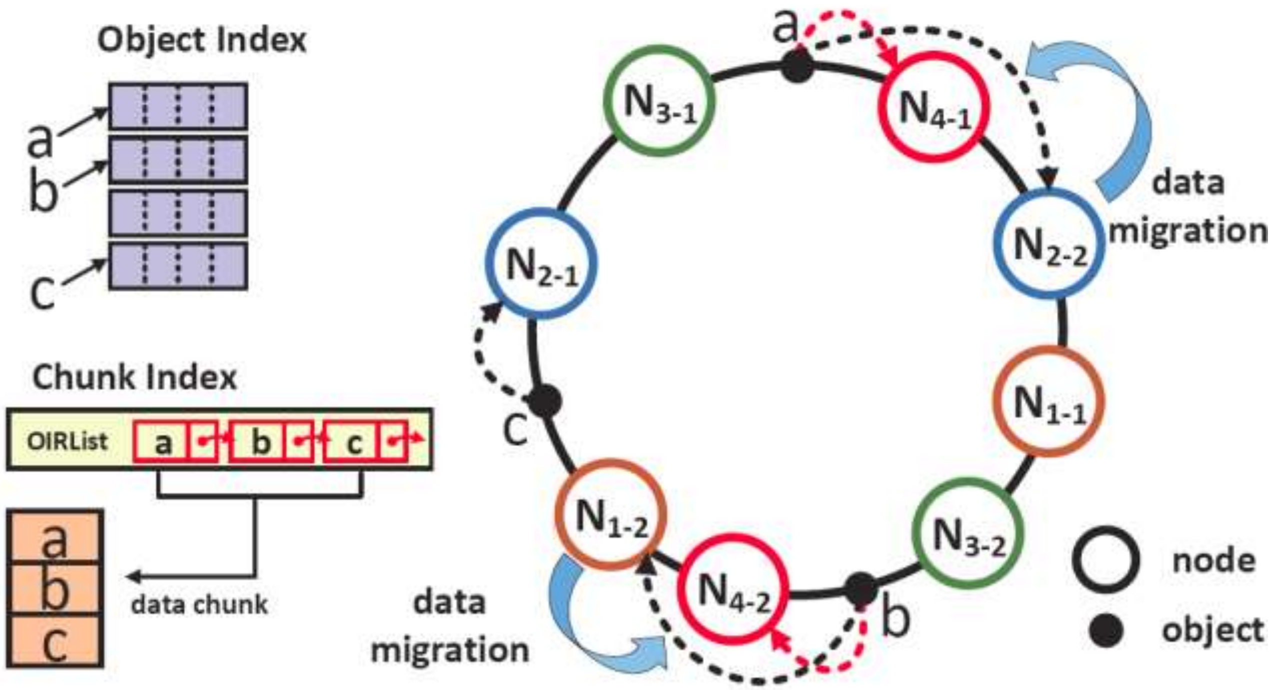


- OIRList records how each data chunk is formed by objects, which can reside in different nodes

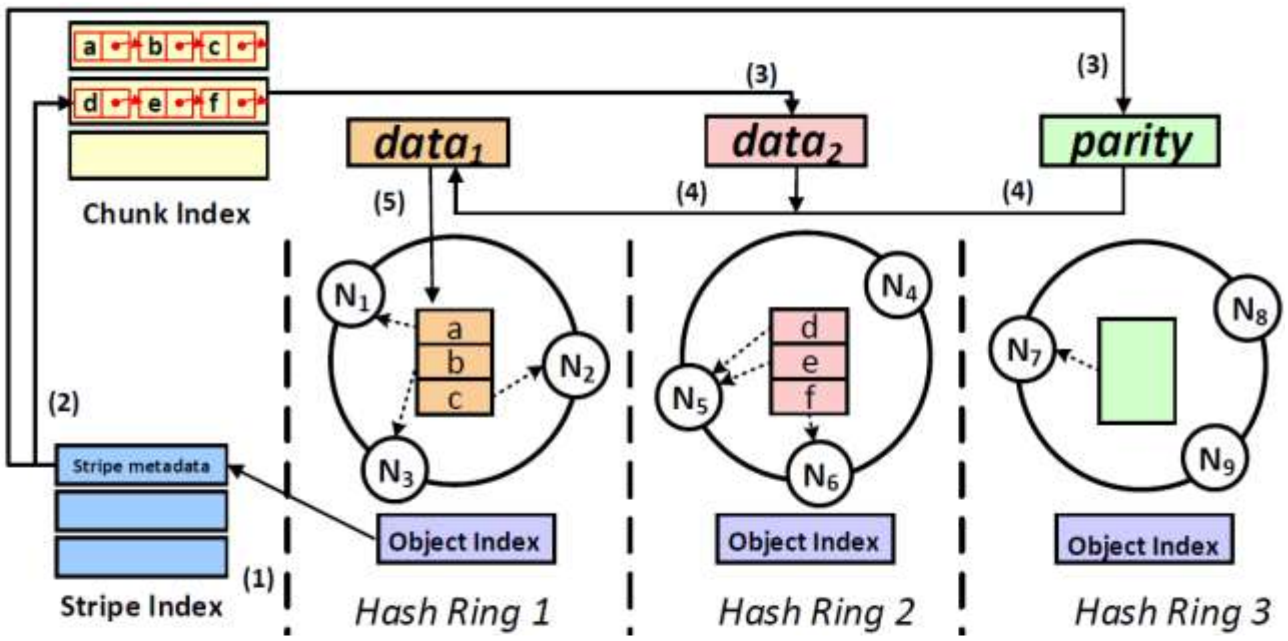
Scaling

- ➢ Traverse Object Index to identify the objects to be migrated
- ➢ Keep OIRList unchanged (i.e., object organization in each data chunk unchanged)

  → No parity updates



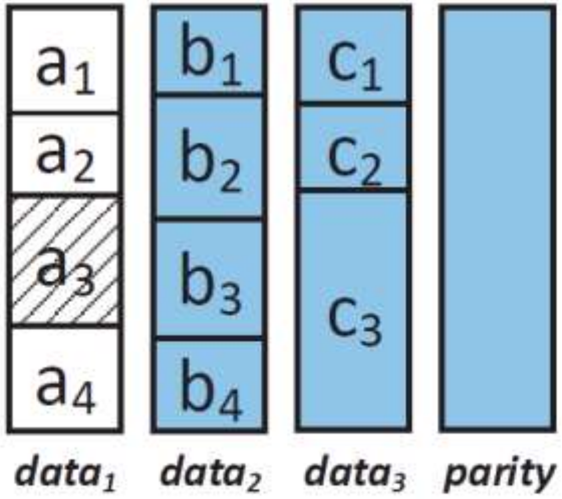
Multiple Hash Rings



- ➢ Distribute a stripe across n hash rings
  - Preserve consistent hashing design in each hash ring
- ➢ Stage node additions/removals to at most n-k chunk updates
  → object availability via degraded reads

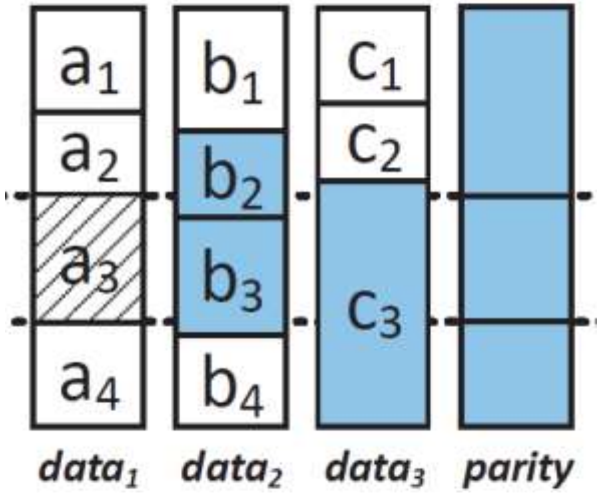
Node Repair

- Issue: How to repair a failed node with only sub-chunks?
  - Decoding whole chunks is inefficient
- **Fragment-repair**: perform repair at a sub-chunk level



Downloads:
$data_2$: b_1, b_2, b_3, b_4
$data_3$: c_1, c_2, c_3
parity

Chunk-repair



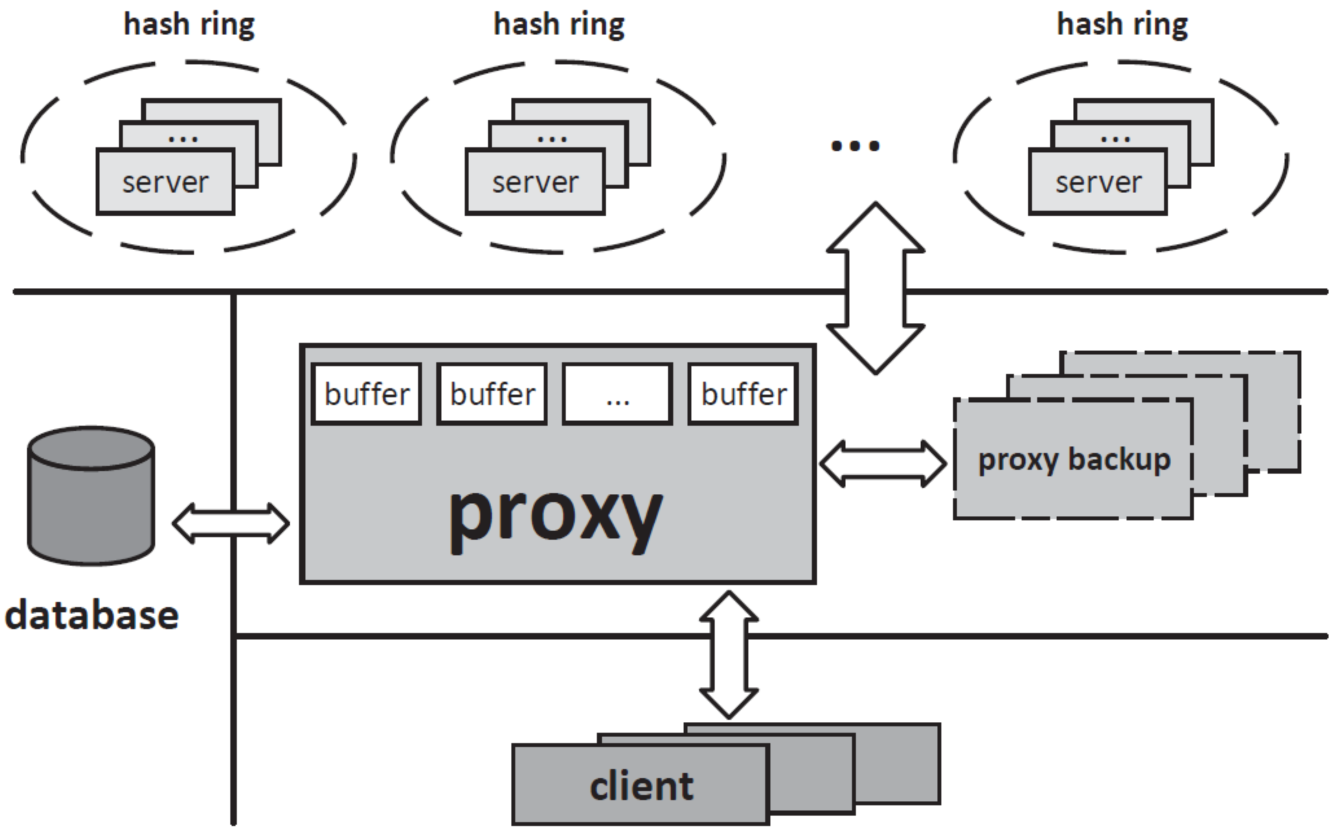
Fragment-repair

Downloads:
$data_2$: b_2, b_3
$data_3$: c_3
parity

Reduce repair traffic

ECHash

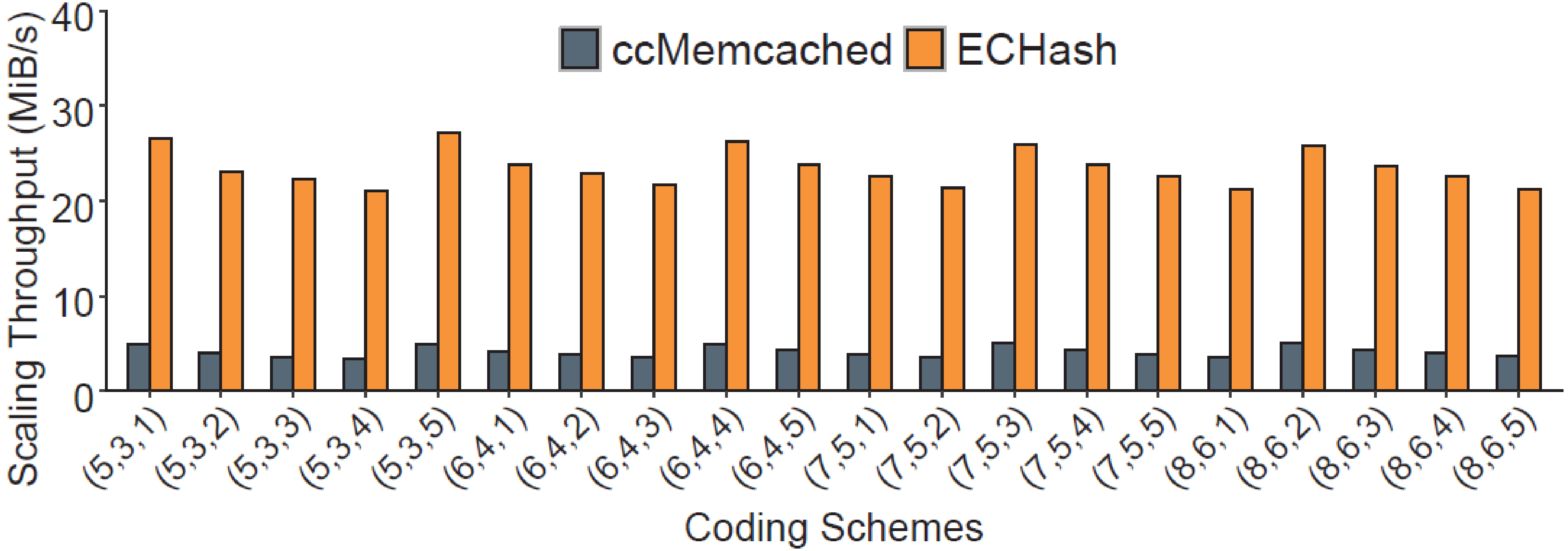
- Built on memcached
  - In-memory KV storage
  - 3,600 SLoC in C/C++
- Intel ISA-L for coding
- Limitations:
  - Consistency
  - Degraded writes
  - Metadata management in proxy



Evaluation

- Testbeds
  - **Local**: Multiple 8-core machines over 10 GbE
  - **Cloud**: 45 Memcached instances for nodes + Amazon EC2 instances for proxy and persistent database
- Workloads
  - Modified YCSB workloads with different object sizes and read-write ratios
- Comparisons:
  - **ccMemcached**: existing cross-coding design (e.g., Cocytus [FAST'16])
  - Preserve I/O performance compared to vanilla Memcached (no coding)
    - See results in paper

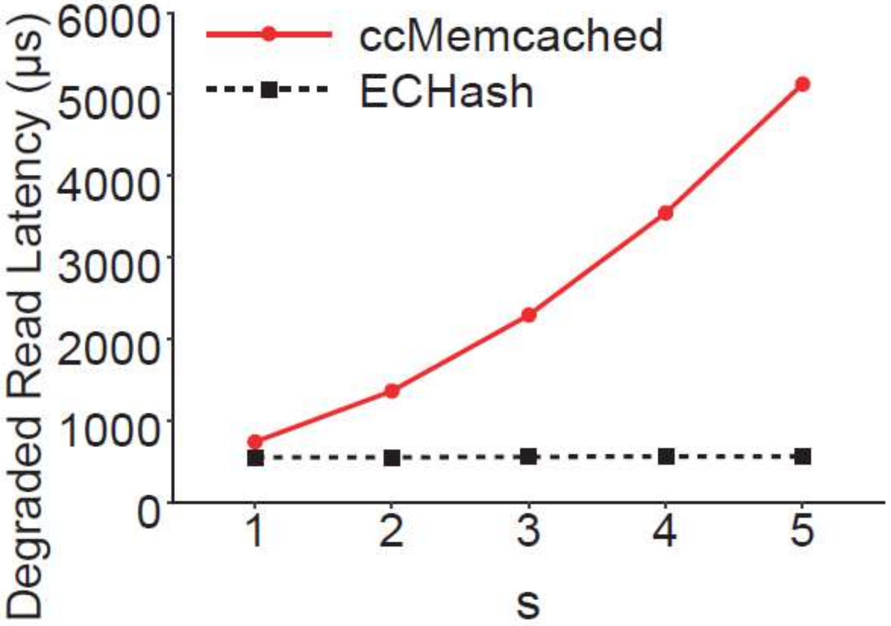
Scaling Throughput in AWS



Scale-out: (n, k, s), where n – k = 2 and s = number of nodes added

➢ ECHash increases scale-out throughput by 5.2x

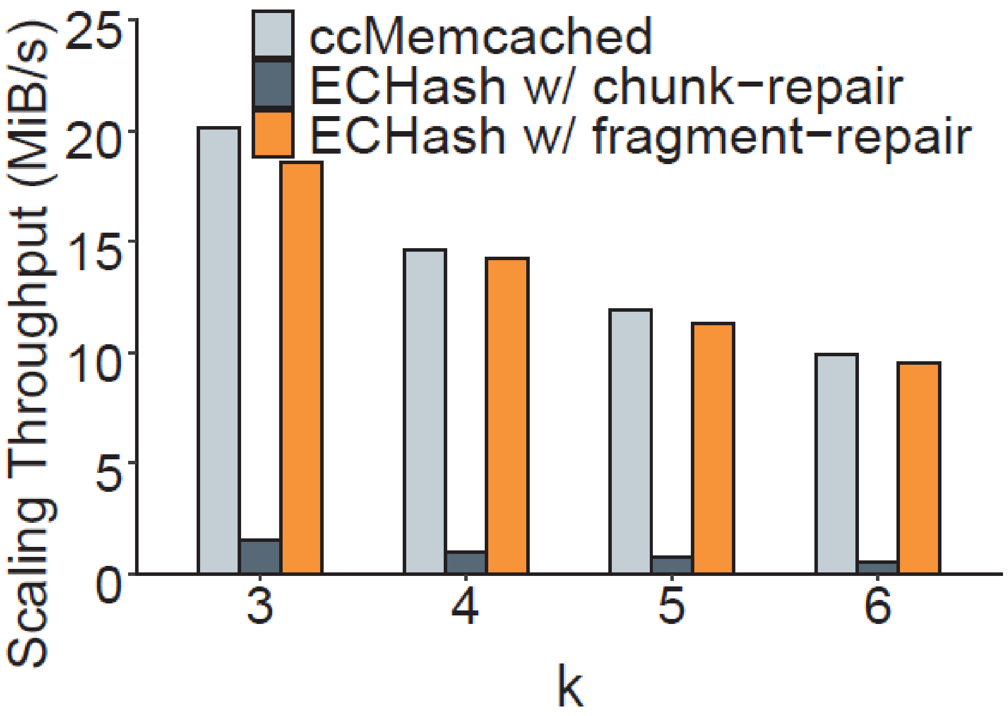
Degraded Reads in AWS



Scale-out: (n, k) = (5, 3) and varying s

- ECHash reduces degraded read latency by up to 89% (s = 5)
  - ccMemcached needs to query the persistent database for unavailable objects

Node Repair in AWS



Scale-out: (n, k) = (5, 3) and varying s

- ➢ Fragment-repair significantly increases scaling throughput over chunk-repair, with slight throughput drop than ccMemcached

Conclusions

- How to deploy erasure coding in decentralized KV stores for small-size objects
- Contributions:
  - FragEC, a new erasure coding model
  - ECHash, a FragEC-based in-memory KV stores
  - Extensive experiments on both local and AWS testbeds
- Prototype:
  - **https://github.com/yuchonghu/echash**