Cache Management for In-Memory Big Data Analytics

Jun ZHANG

Oct 15, 2018

Outline

1. Introduction
2. LRC: Dependency-aware caching
3. OpuS: Fair cache sharing in multi-tenant cloud
4. SP-Cache: Load balancing for cluster caching
5. Summary

My Research Interests

- **Wireless Communications**
  - Machine learning for wireless communications
  - Cellular-enabled UAV communications
- **Edge Intelligence**
  - Mobile edge computing
  - Edge learning
- **Distributed Computing Systems**
  - Big data analytics systems
  - Distributed learning
- Recruiting (@PolyU)
  - 2~3 PhD, 1 Postdoc

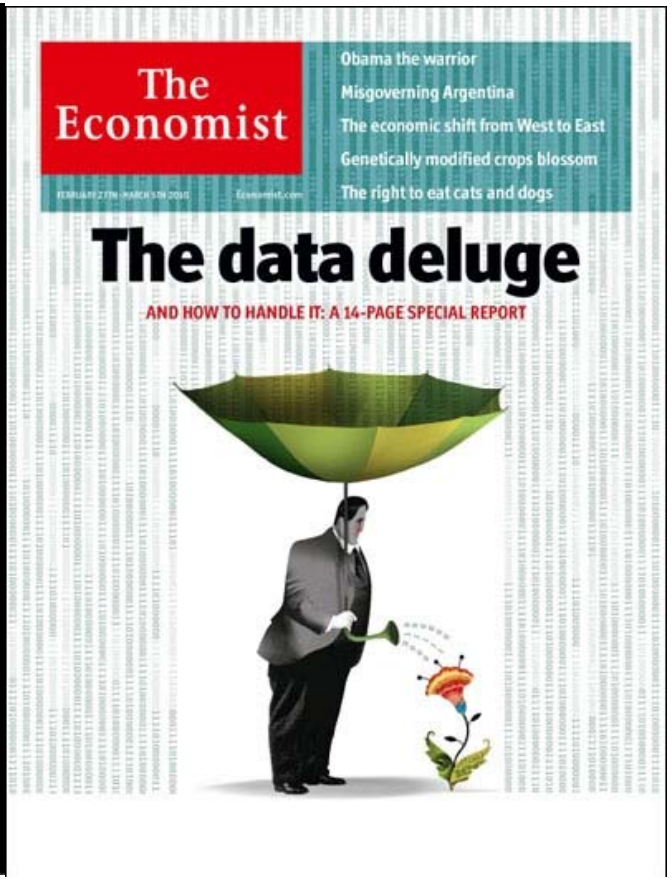
Big Data


BIG
DATA
VOLUME
STORAGE
PROCESSING
INFORMATION
VARIETY
VELOCITY
STREAM
DECISION
SHARING
COLLECTION
TOOLS
TRANSFER
ANALYSIS
DATA
CAPTURE
DEPENDENCIES
ENABLE
TRIGGER
LOGS
TYPES
HIGH
TRENDS
INSIGHT
ENHANCED
AREAS
OUT
DENSITY
BUSINESS
SOFTWARE
ENTERPRISE
SCIENTISTS
INTERNET
SPEED
OPTIMIZATION
PROCESS
REPORT
CAPTURE
CURATE
BIG
TAKE
SENSOR
DOMAIN
GROWS
STATISTICS
METEOROLOGY
RELATIONAL
PETABYTES
AMOUNT
TERABYTES
REQUIRE
IN
RANGE
CAMERAS
MANAGE
SOURCES
SYSTEMS
SIZE
GOVERNMENT
LANEY
DOUG
NEW
ENVIRONMENTAL
TOLERABLE
CONNECTOMICS
INFORMATICS
AND
THAT
MAKING
RESEARCH
SIMULATIONS
NETWORKS
HUNDREDS
PHYSICS
WIRELESS
DISCOVERY
CAPABILITIES
THINGS
HANDLE
BLANKET
BECOMES
ANY
TERM
INCLUDE
CHALLENGES
USING
MANAGEMENT
PROCESSING
DIFFICULT
EXPLORE
THAT
PROCESS
DATABASE
APPLICATIONS
SIMULATION
BYTES
PROCESS
PROBLEMS
IS A
BUSINESS INTELLIGENCE
SEARCH
ON-HAND
3V
GARTNER
COMPLEX
ASSETS
FOR
FORMS
FLOW
VISUALIZATION
LARGE
DURATION
AND
PACKAGES
GENOMICS
FINANCE
MICROPHONES

Big Data


nature
THE BITER BIT
Viral infections for viruses
TROPICAL CYCLONES
The strong get stronger
BLACK HOLE PHYSICS
A new window on the Galactic Centre
BIG DATA
NATUREJOBS
Minnesota musings
SCIENCE IN THE PETABYTE ERA

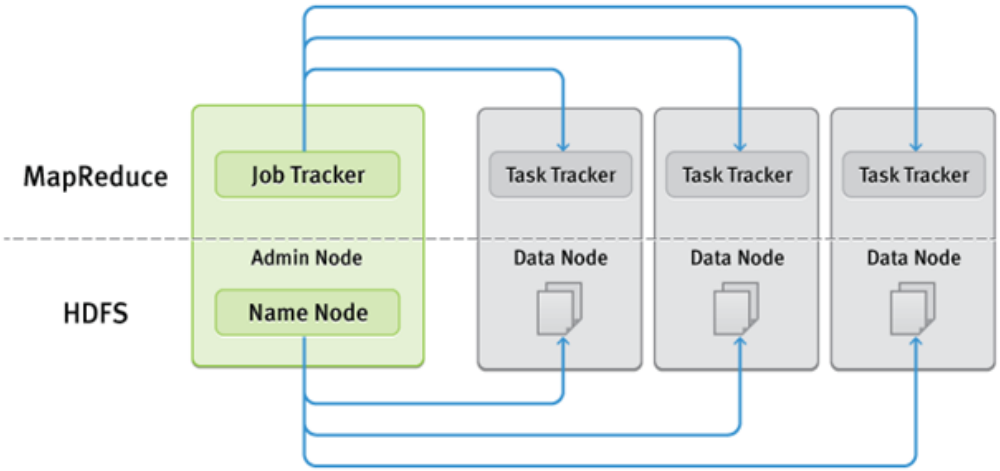

Science
scientific
information
climate
research
analysis
researchers
visualization
new
many
access
human
data
AAAS


The Economist
Obama the warrior
Misgoverning Argentina
The economic shift from West to East
Genetically modified crops blossom
The right to eat cats and dogs
The data deluge
AND HOW TO HANDLE IT: A 14-PAGE SPECIAL REPORT

BIG DATA & AI LANDSCAPE 2018

Big Data Analytics

- Cluster-Computing Frameworks
  - hadoop (December 2011) [1]



  - Spark (May 2014) [2]

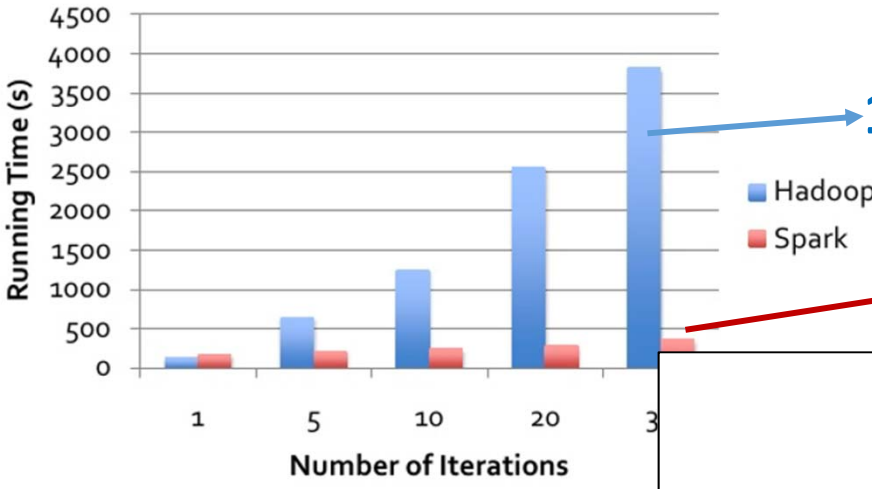
    In-memory cluster computing

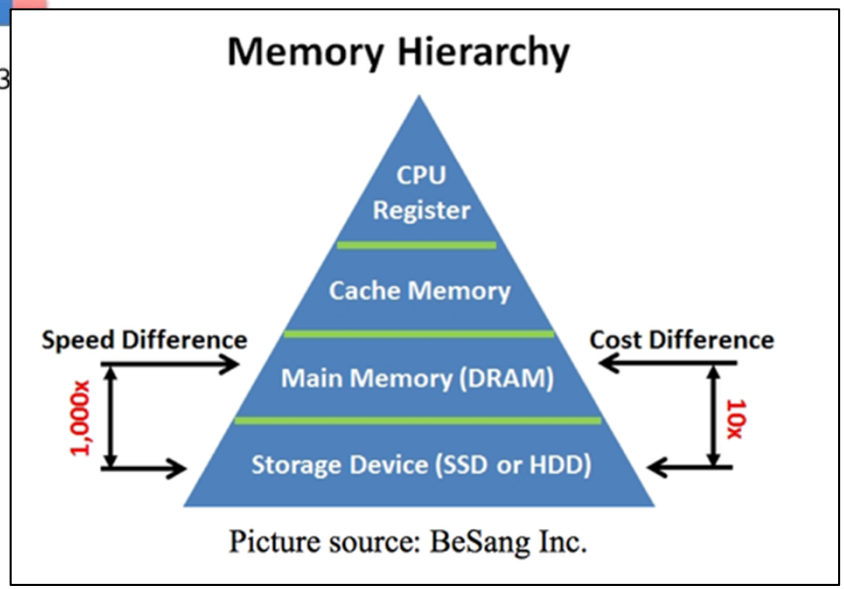
[1] J. Dean and S. Ghemawat. "MapReduce: Simplified data processing on large clusters." In Proc. The 6th Symposium on Operating Systems Design and Implementation (OSDI), pp.137–150, Dec. 2004.

[2] M. Zaharia, M. Chowdhury, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." In NSDI, 2012.

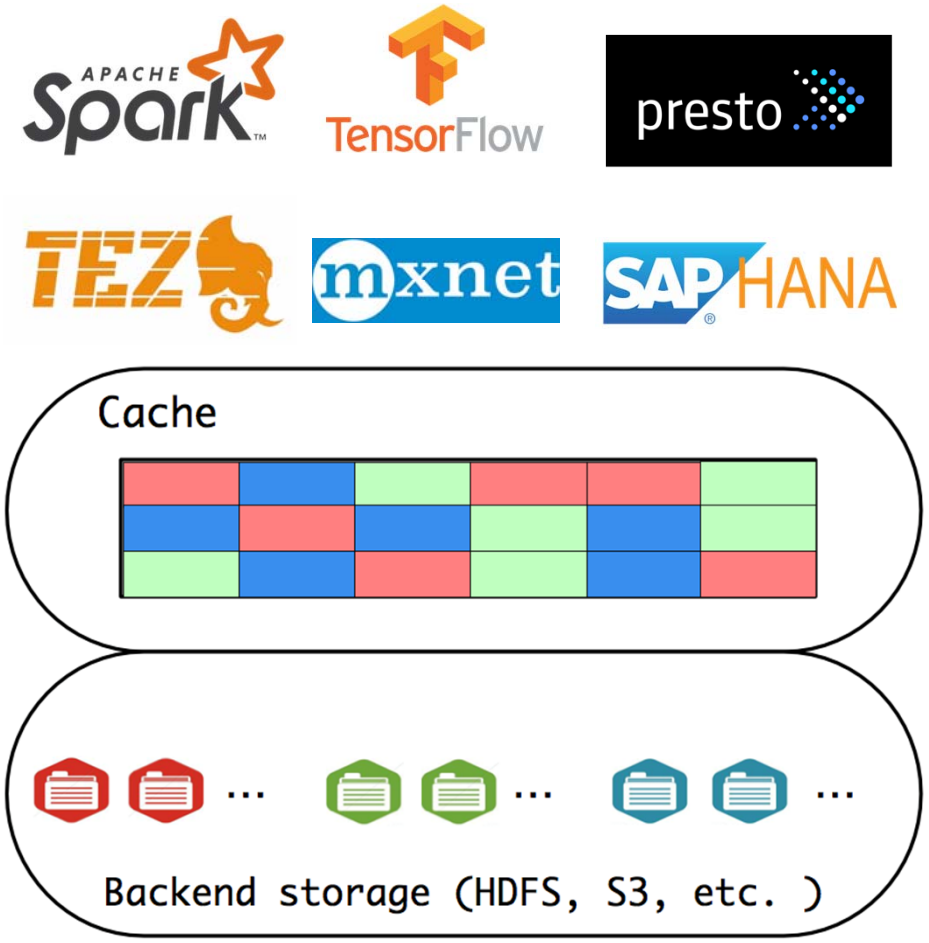
Memory Speeds up Computation

- Caching input data speeds up tasks by orders of magnitude.
  [M. Zaharia, 2012]





In-Memory Data Analytics



Cache Management

- **Cache is limited**
  - What to cache? [INFOCOM 17]
- **Cache is shared resource**
  - How to fairly share cache among users? [ICDCS 18]
- **Distributed caches → Load imbalance**
  - How to balance the load? [SC 18]

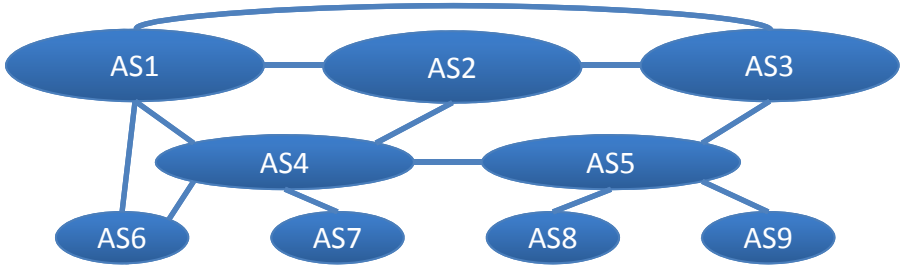
1. LRC

Dependency-aware Caching Management

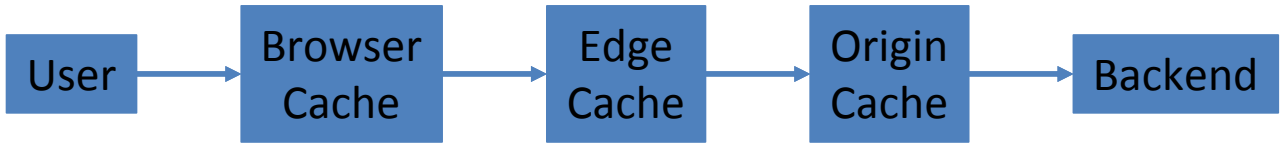
[3] Y. Yu, W. Wang, **J. Zhang**, and K. B. Letaief, "LRC: Dependency-aware cache management in data analytics clusters," in Proc. IEEE INFOCOM 2017, Atlanta, GA, May 2017.

Cache Management

- Crucial for in-memory data analytics systems.
- Well studied in many systems
  - **CDN** (Akamai)



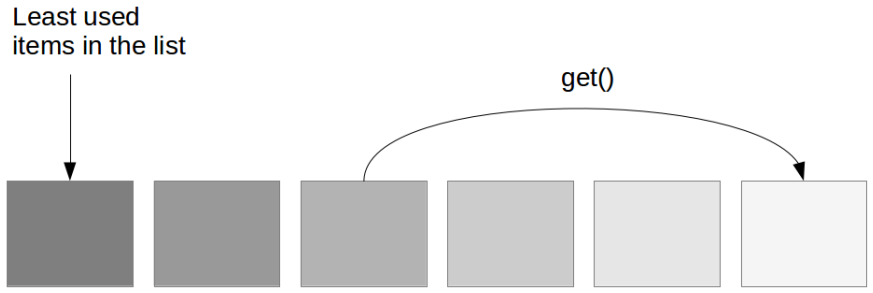
  - **Facebook**



- **Objective**: optimize the ***cache hit ratio***
  - Maximize the chance of in-memory data access.

Existing Solutions

- Least Recently Used (LRU) policy [R. L. Mattson, 1970]
  - Evicts the data block that has not been used for the longest period.
  - Widely employed in prevalent systems, e.g., Spark, Tez and Alluxio.



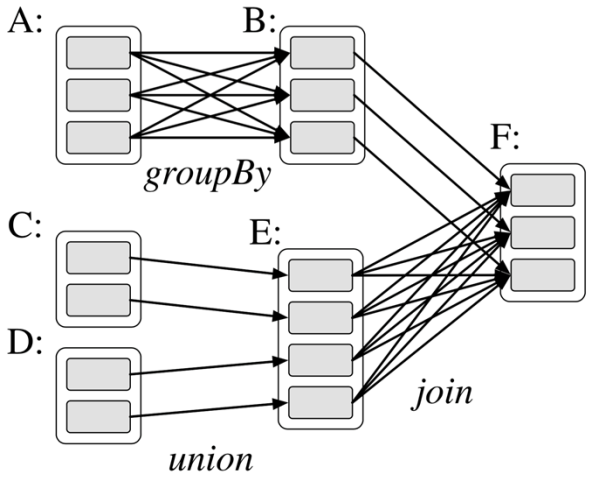
Calling get() for an item, moves it to the top of the cache

- Least Frequently Used (LFU) policy [M. Stonebraker, 1971]
  - Evicts the data block that has been used the least times.

- Summary: "**guessing**" the future data access patterns based on historical information (access recency or frequency).

Data Dependency Reveals Access Patterns

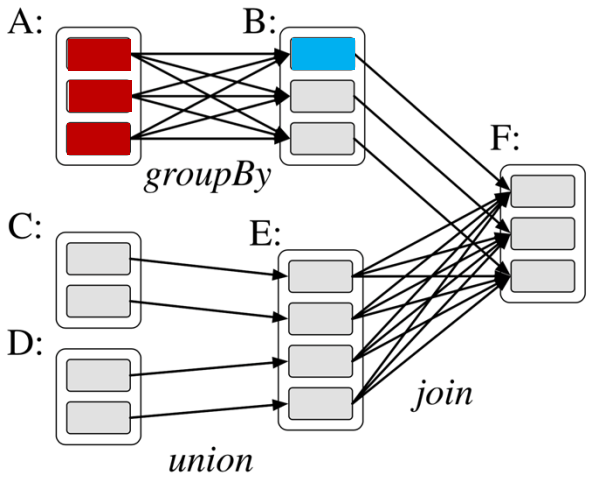
- Application Semantics
  - Data dependency structured as a Directed Acyclic Graph (DAG)



  - Available to the cluster scheduler **before** the job starts
- ***Data access*** follows the dependency DAG.
  - The future is not totally unpredictable.

Data Dependency Reveals All-or-Nothing Property

- ***All-or-Nothing***: a computing task can only be sped up when its dependent data blocks are ***all*** cached in memory.
  - E.g. To compute a block in B, all blocks of A are required. Cache hits of only part of the three blocks makes no difference.



- Cache hit ratio is not appropriate metric for cache performance.

Inefficiency of Existing Cache Polices

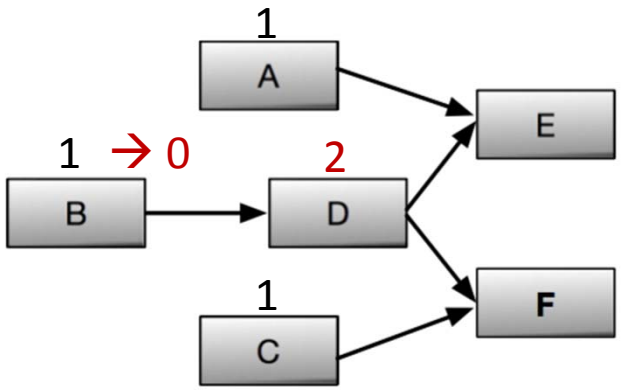
- Oblivious to the data access pattern:
  - Inactive data (no future access) cannot be evicted timely.
  - In our measurement studies, inactive data accounts for **>77%** of the cache space for **>50%** of time.

- Oblivious to the all-or-nothing property:
  - Achieving a high cache hit ratio does not necessarily speed up the computation.

- **Challenge**: How to exploit the data dependency information (DAGs) to clear **the inactive data** efficiently and factor in **the all-or-nothing property**?

LRC: Dependency-Aware Cache Management

- **Reference count**: defined for each data block as the number of downstream tasks depending on it.
  - Dynamically changing over time:



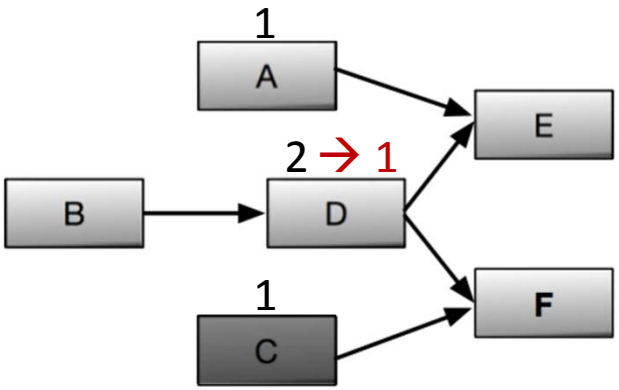
- Least Reference Count (**LRC**) policy: when the cache is full, always evict the data with the least reference count.
  - Inactive data (w/ zero reference count) is evicted first, e.g., block B.

Effective Cache Hit Ratio

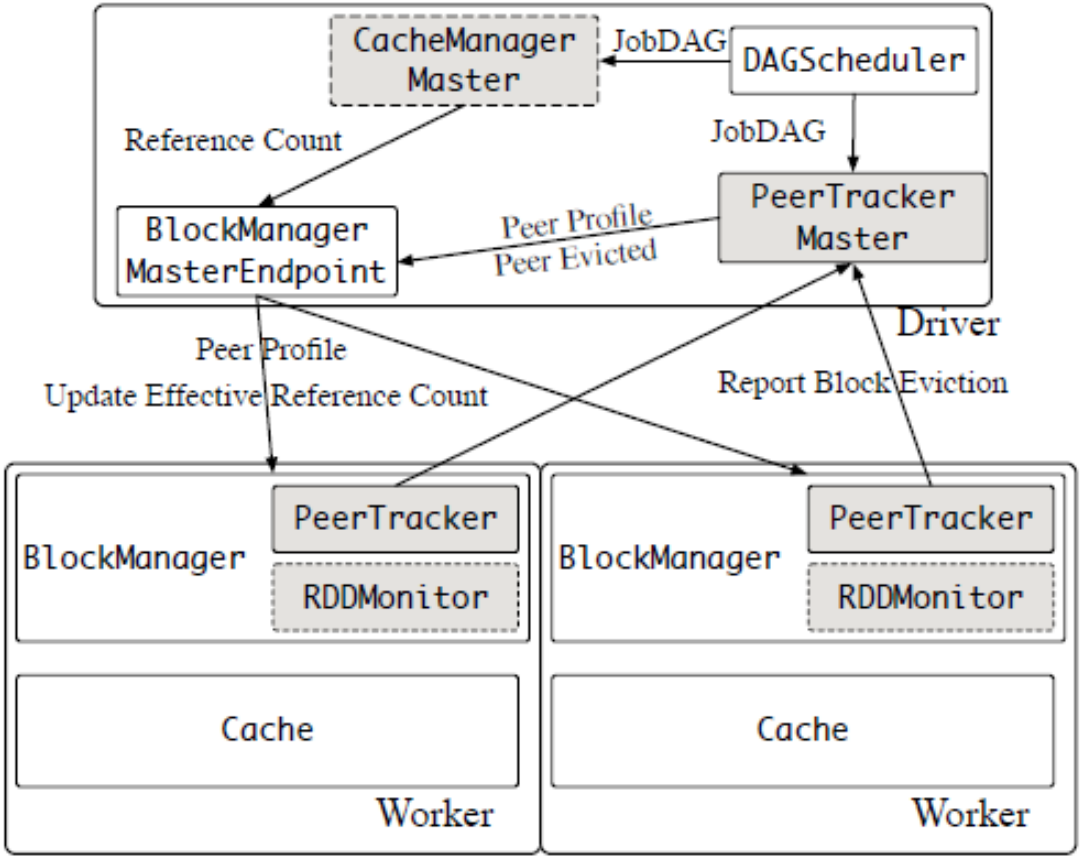
- Factor in the all-or-nothing property?
- *Effective cache hit ratio:* A cache hit is effective when it speeds up the computation, i.e., when all the depended blocks are cached.

Tailor LRC to Optimize Effective Cache Hit Ratio

- A reference to a data block is only "counted" when it effectively speeds up the computation
  - E.g., the reference to block D for computation of block F is not counted if block C is evicted from the cache.



Spark Implementation



Evaluations: Workload Characterization

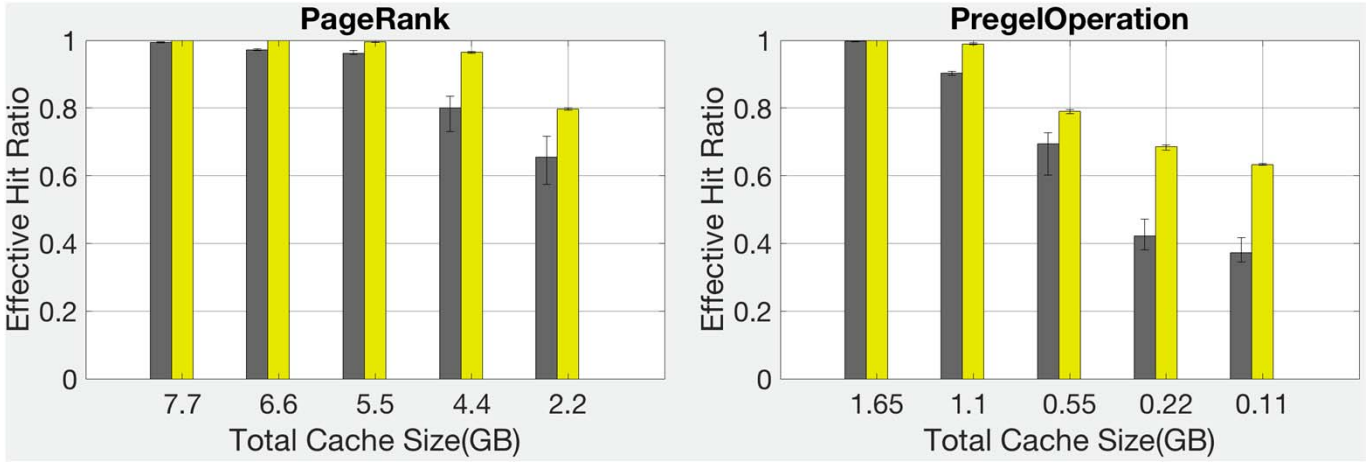
- **Cluster setup**: 20-node Amazon EC2 cluster.
- **Instance type**: m4.large. Dual-core 2.4 GHz Intel Xeon® E5-2676 v3 (Haswell) processors and 8 GB memory.
- **Workloads**: Typical applications from SparkBench [4].

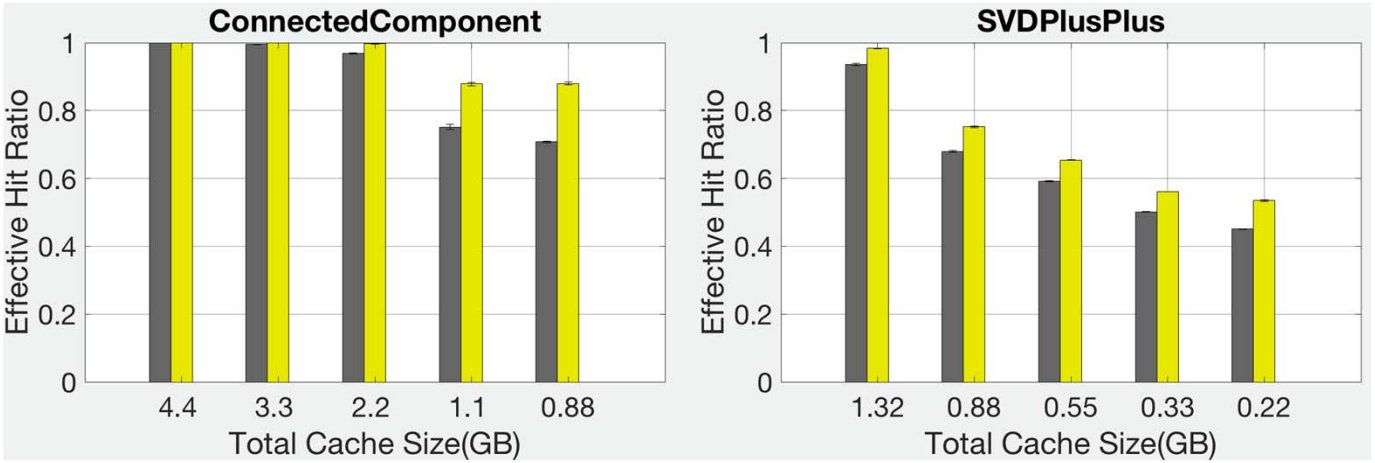
Workload	Cache All	Cache None
Page Rank	56 s	552 s
Connected Component	34 s	72 s
Shortest Paths	36 s	78 s
K-Means	26 s	30 s
Pregel Operation	42 s	156 s
Strongly Connected Component	126 s	216 s
Label Propagation	34 s	37 s
SVD Plus Plus	55 s	120 s
Triangle Count	84 s	99 s
Support Vector Machine (SVM)	72 s	138 s

Not all applications benefit from the improvement of cache management.

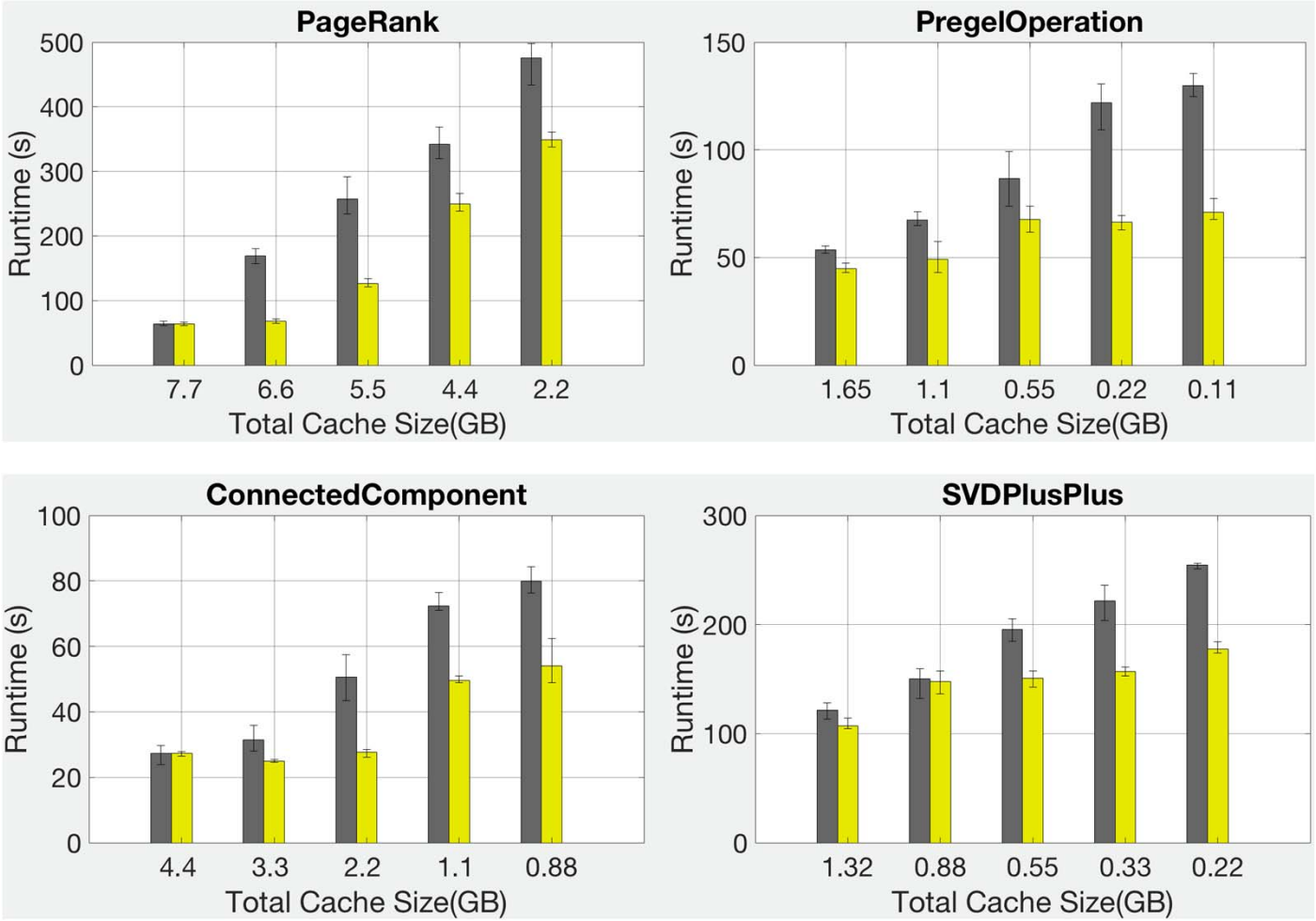
[4] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark," in Proc. 12th ACM International Conf. on Comput. Frontiers, 2015.

Evaluations: Effective Cache Hit Ratio





Evaluations: Application Runtime



Evaluations: Summary

- LRC speeds up typical workloads by up to 60%.

Workload	Cache Size	LRU	LRC	Speedup by LRC
Page Rank	6.6 GB	169.3 s	68.4 s	59.58%
Pregel Operation	0.22 GB	121.9 s	66.3 s	45.64%
Connected Component	2.2 GB	50.6 s	27.6 s	45.47%
SVD Plus Plus	0.88 GB	254.3 s	177.6 s	30.17%

Conclusions

- It is effective to leverage the dependency DAGs to optimize the cache management
- Effective cache hit ratio is a better cache performance metric
  - To account for the all-or-nothing property
- **LRC** – a dependency-aware cache management policy
  - Optimizes the effective cache hit ratio
  - Speeds up typical workloads by up to 60%

2. Fairness

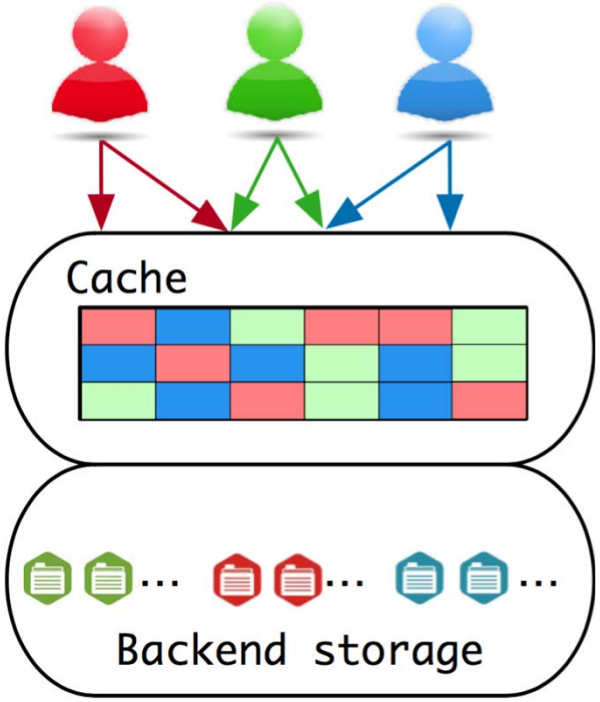
OpuS: Fair and Efficient Cache Sharing

---

[5] Y. Yu, W. Wang, J. Zhang, Q. Weng, and K. B. Letaief, "OpuS: Fair and efficient cache sharing for in-memory data analytics," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, July 2018.

Cache Sharing

- Multiple users/applications and limited cache resources.
- Objective: fairness and efficiency (cache utilization).



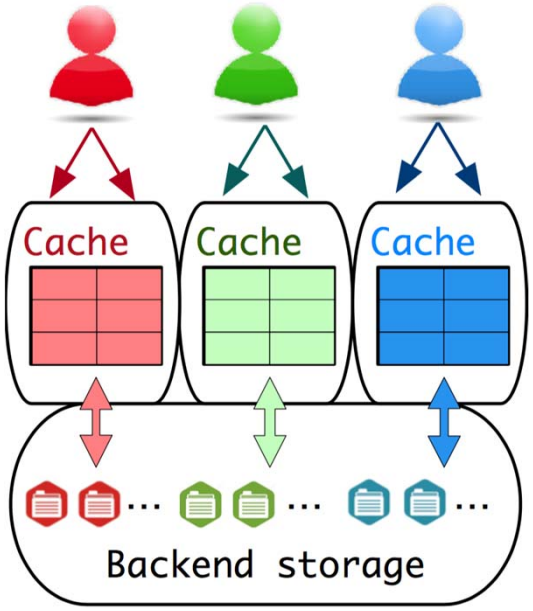
What we want

- Fair:
  - Isolation guarantee: no worse than isolation.
  - Strategy-proofness: cannot cheat to improve.

- Efficient:
  - Pareto efficiency: impossible to improve a user without hurting others.

Isolated Cache?

- Fair:
  - Equal and dedicated.

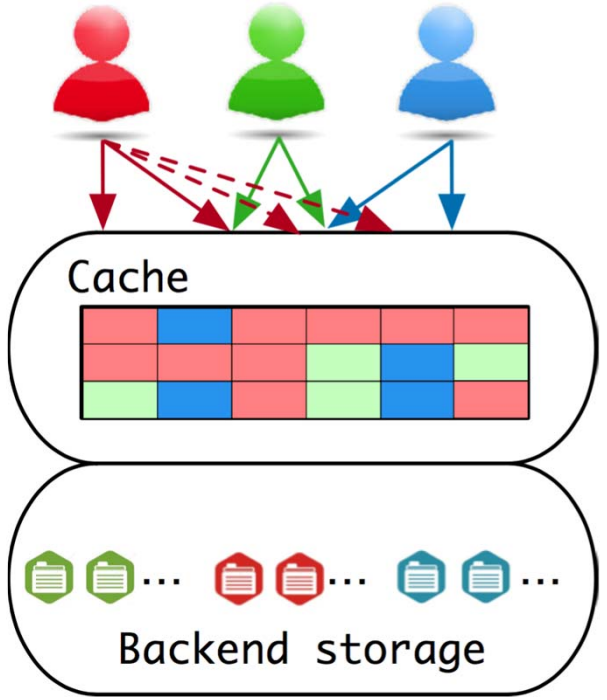


- Inefficient:
  - Duplication.
  - Users might not be able to fully utilize their quota.

Global Sharing?

- Efficient:
  - No duplication; no waste.

E.g., LRU, LFU



- Unfair:
  - Low-demand users get arbitrarily small allocations
  - Prone to strategic behavior.

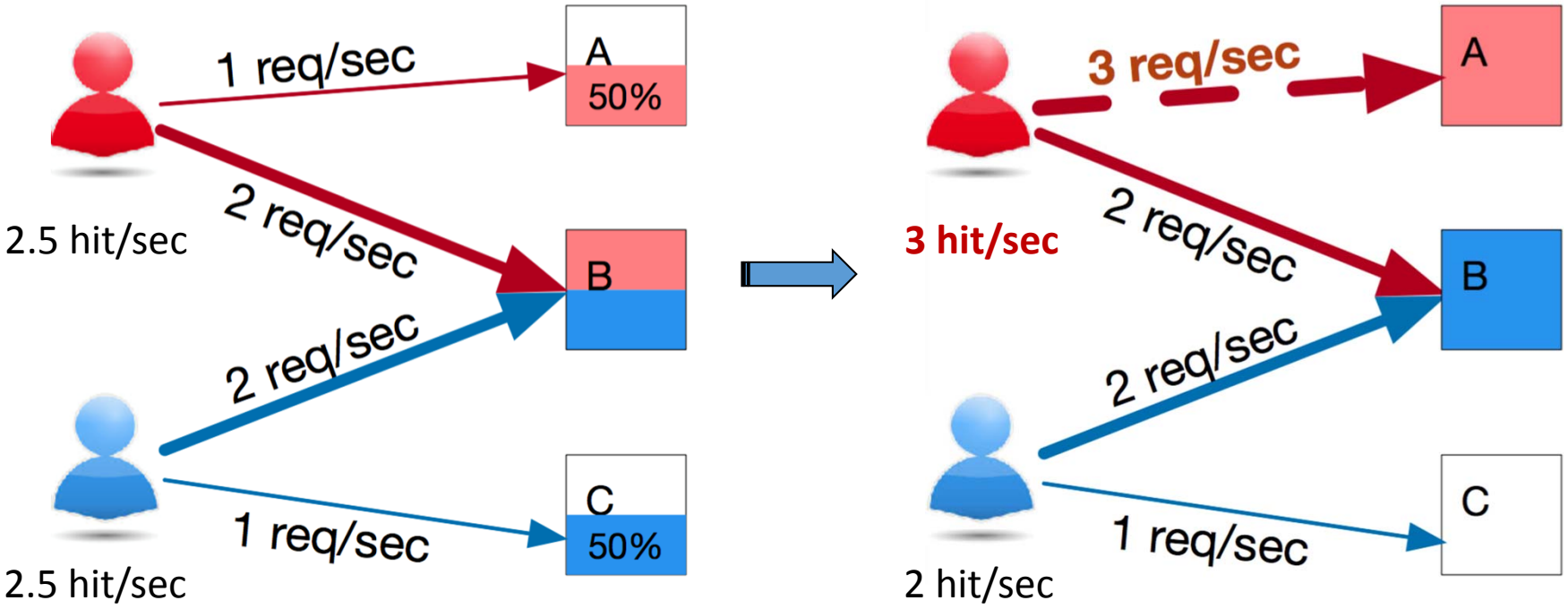
Challenge: Free-riding

- Non-exclusively shared.
  - No user caches files of common interest, in the hope of free-riding.

- Solutions for others resources, e.g., CPU and network, do not work.

Max-min Fair Allocation

- Maximize the minimum cache allocation across all users.
- Suffer from ***free-riding*** strategies:

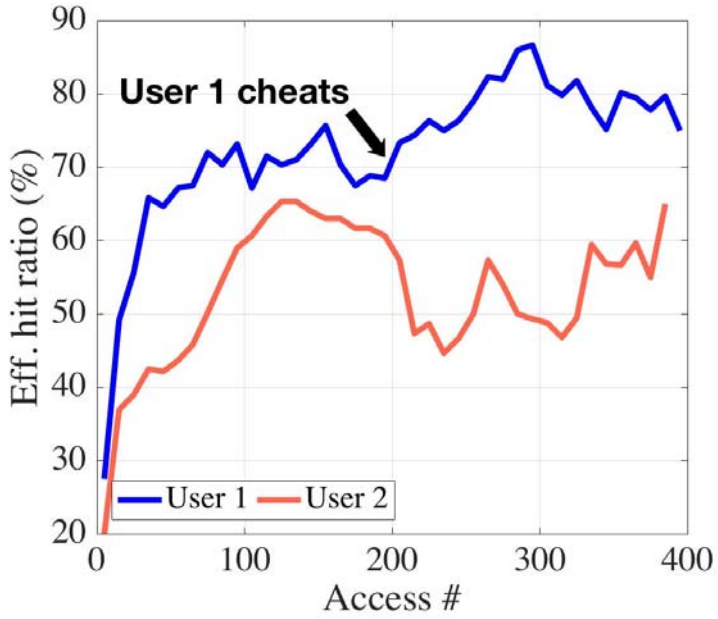


Total budget: 2 unit

FairRide: State-of-the-art [6]

- Built atop max-min fairness; block free-riders with a chosen probability.

- Users can still cheat to improve by other forms of cheating.



FairRide

[6] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica, "FairRide: nearoptimal, fair cache sharing," in *Proc. USENIX NSDI*, 2016.

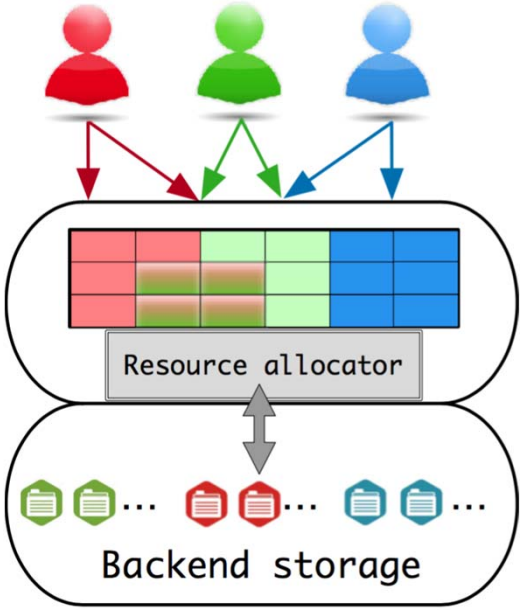
Summary of Existing Solutions

	Isolation guarantee	Strategy-proofness	Pareto-efficiency
Isolation	✓	✓	
Global			✓
Max-min	✓		✓
FairRide	✓		Near max-min

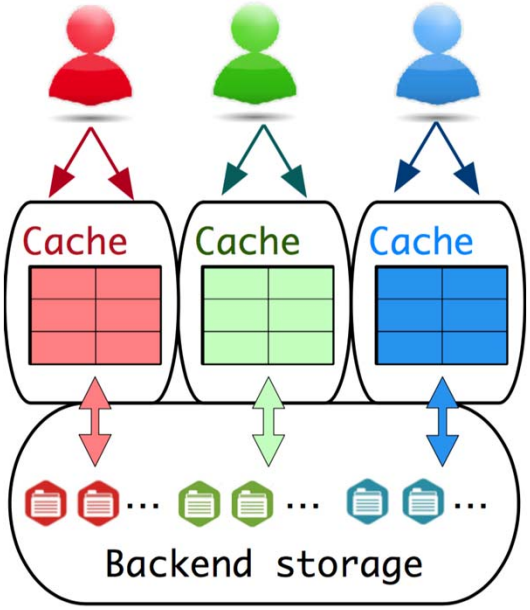
- It was proved in [6] that no allocation can achieve all the three properties due to free-riding.

OpuS

- OpuS: Two-stage **Op**port**u**nistic **S**haring



2. Isolation not guaranteed?



1. Stage I: share the cache.

High efficiency **and high probability of IG.**

Strategy-proof

3. Stage II: isolation.

OpuS

- Stage I: cache sharing
  - High efficiency and high chance of isolation guarantee

→ **Proportional fairness**

$$\text{maximize}_{\mathbf{a}} \sum_i \log H_i(\mathbf{a})$$

Cache hit ratio

OpuS

- Stage I: cache sharing
  - Strategy-proof → **VCG mechanism**
    - Charge each user a "tax": the loss of others due to its existence.

      Free-riding is not "free"!

$$\mathrm{Tax}_i = \sum_{j \neq i} \log H_j(\mathbf{a}') - \sum_{j \neq i} \log H_j(\mathbf{a})$$

Others' utility without user i's existence

    - Discourage any kind of cheating: the utility increment is always less than the extra tax paid.

OpuS

- Stage I: cache sharing
  - The VCG tax is charged via blocking:

$$U_i^{net} = \log H_i(\mathbf{a}) - \mathrm{Tax}_i = \log f_i \cdot H_i(\mathbf{a})$$

Non-blocking prob.

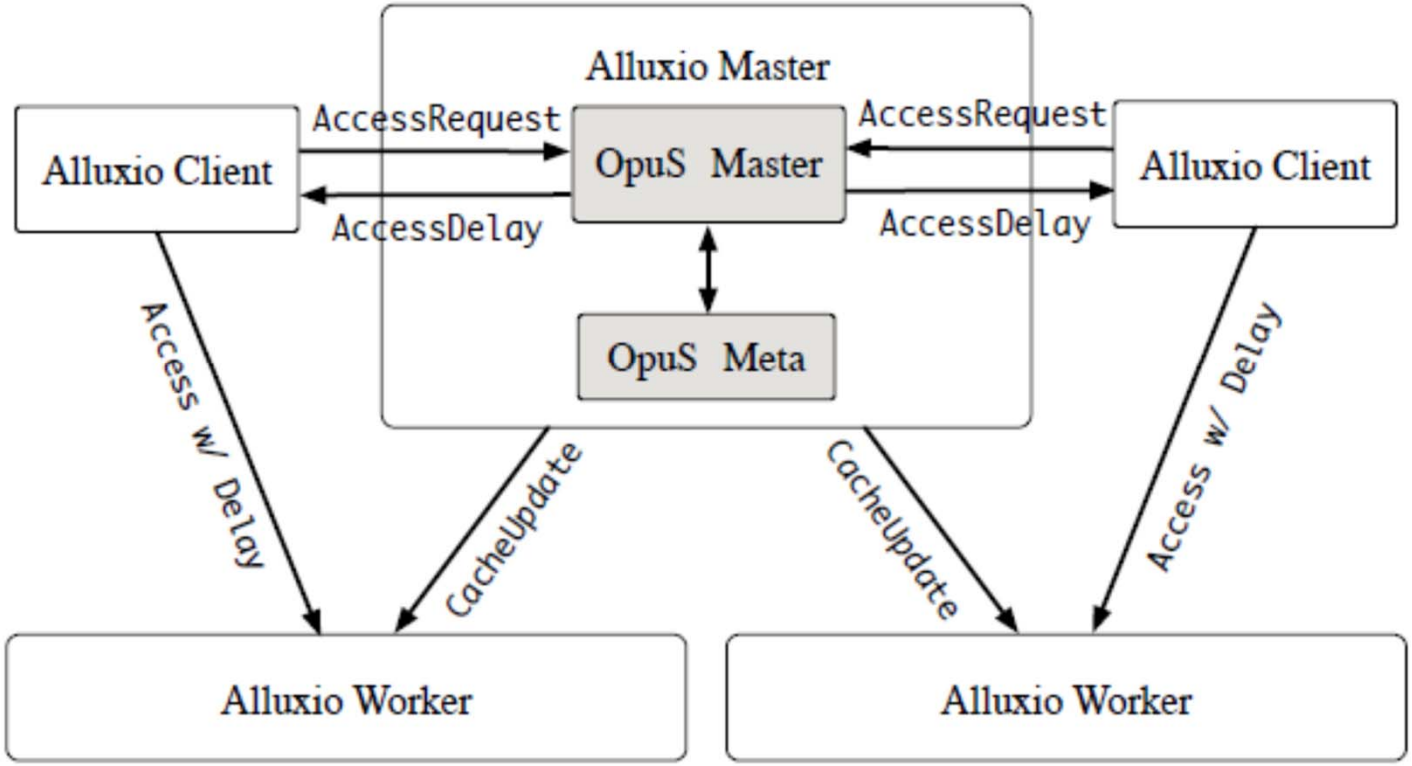
- Stage II: Isolation
  - Reduce to isolation if IG is violated.

Our Contribution

	Isolation guarantee	Strategy-proofness	Pareto-efficiency
Isolation	✓	✓	
Global			✓
Max-min	✓		✓
FairRide	✓		Near max-min
OpuS	✓	✓	Near optimal

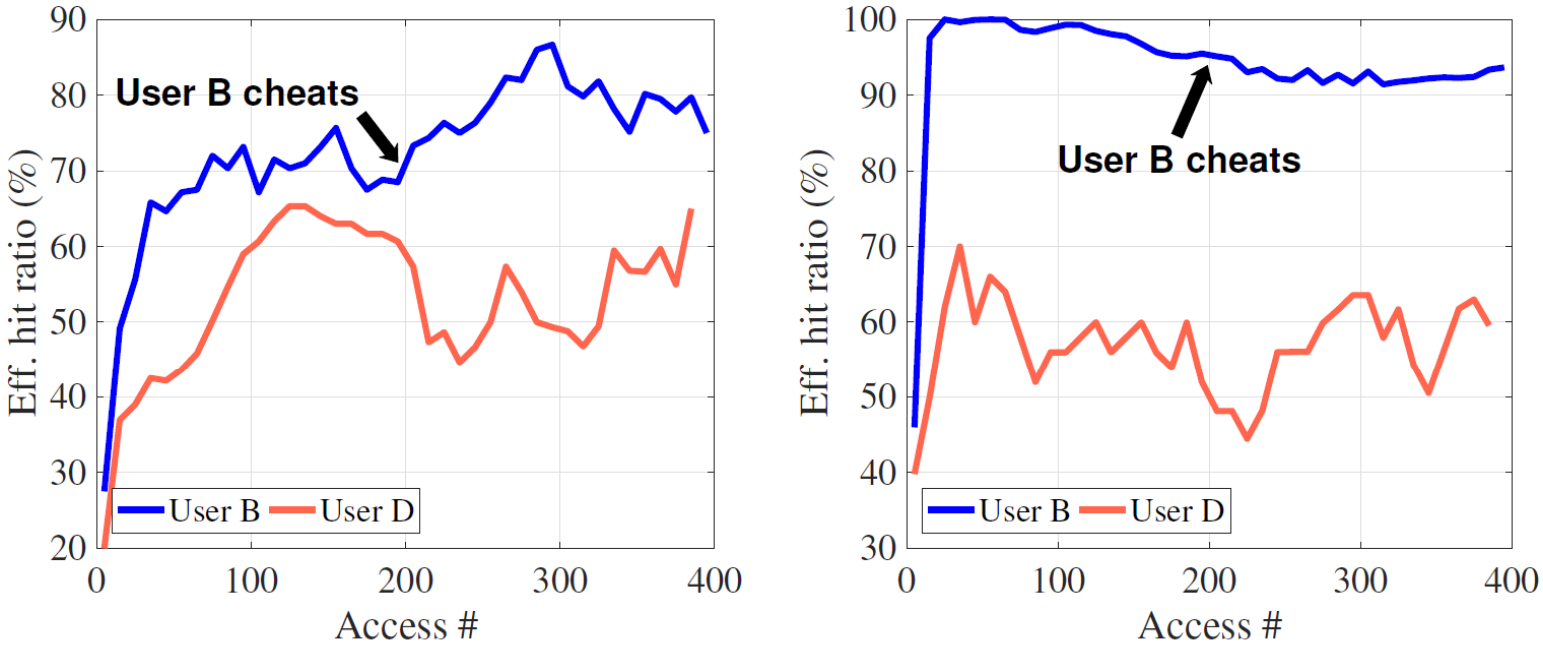
- Isolation guarantee.
- Immune to any form of cheating.
- Near-optimal efficiency.

Implementation



Evaluations

- Strategy-proofness

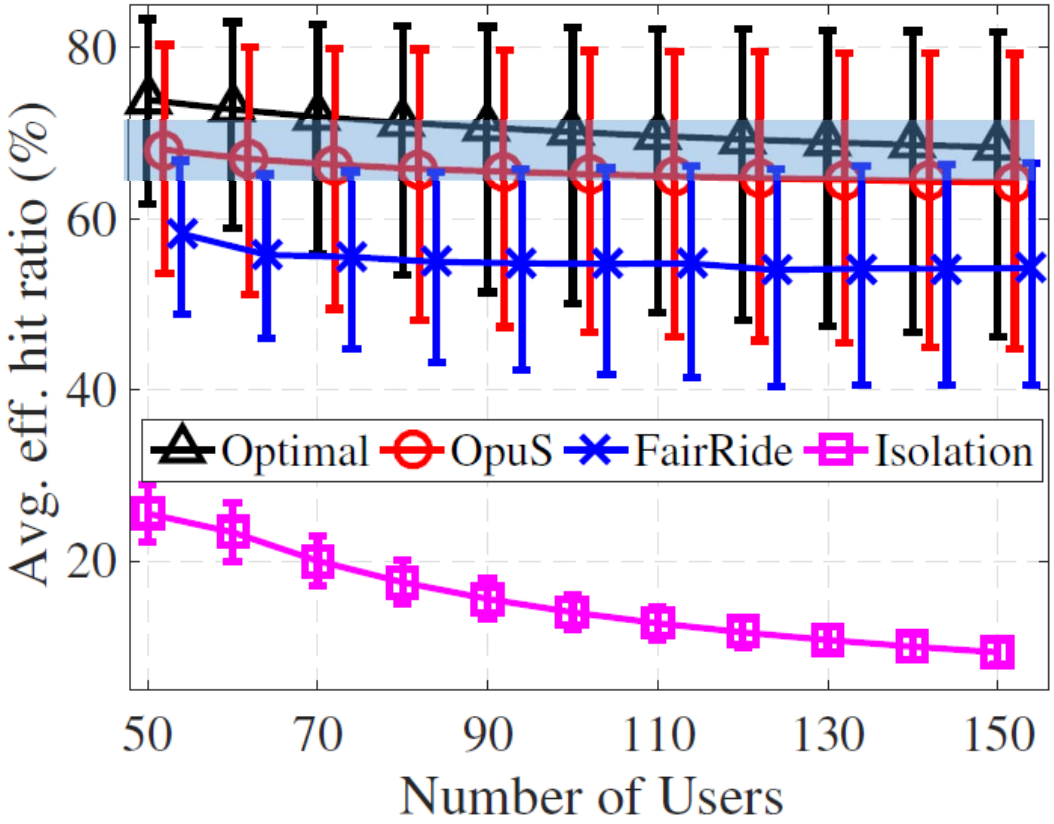


(a) FairRide (b) OpuS

Cheating only leads to worse performance

Evaluations

- Near-optimal efficiency



$\leq 7\%$

Wrap up – Fair Cache Sharing

- Cache sharing uniquely faces the free-riding challenge.

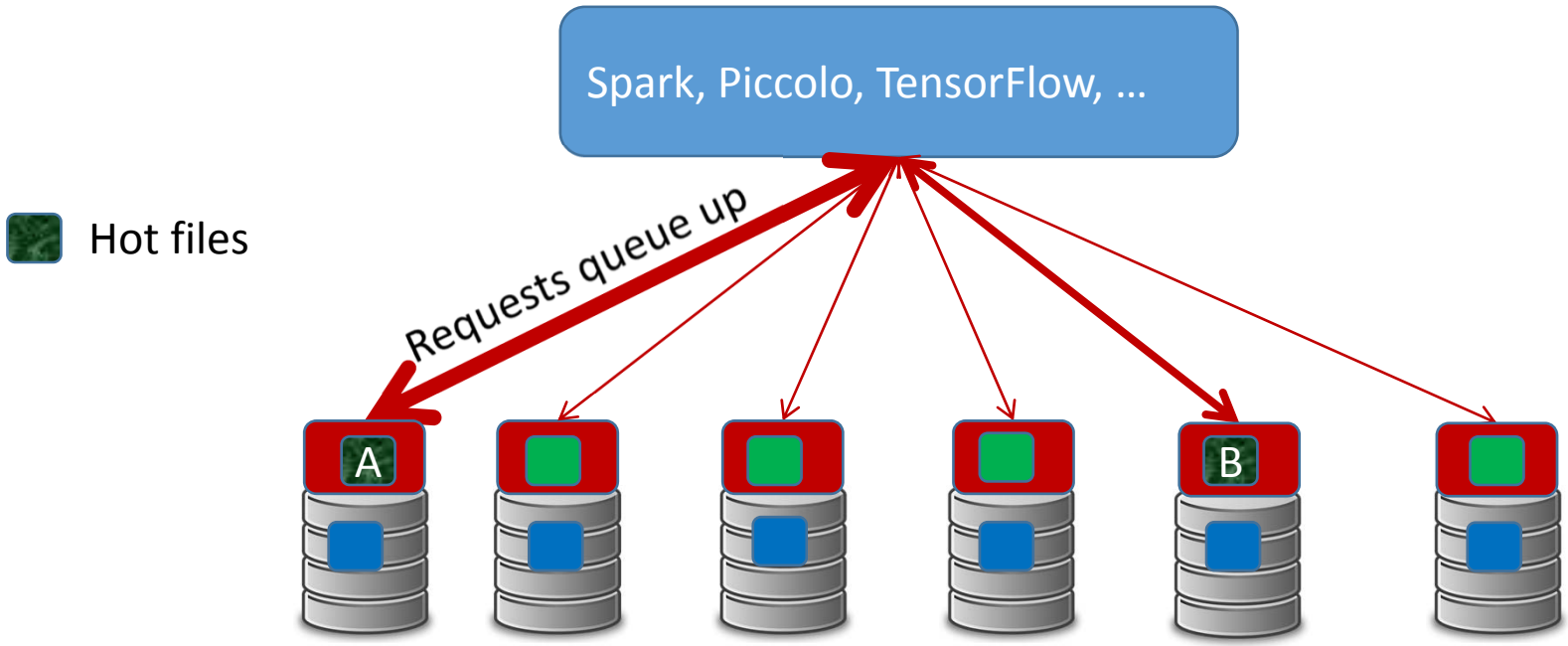
- OpuS: a cache allocation algorithm for shared clouds.
  - Dis-incentivize all cheating strategies, including free-riding
  - Near-optimal efficiency

3. Load Balancing

SP-Cache: Load-balanced, Redundancy-free Cluster Caching with Selective Partition

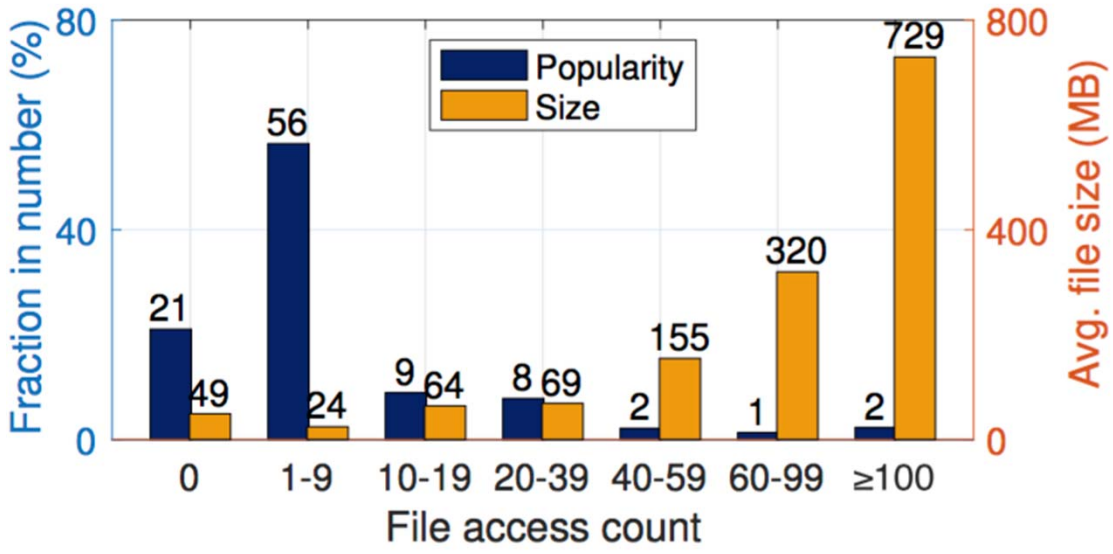
[7] Y. Yu, R. Huang, W. Wang, J. Zhang, and K. B. Letaief, "SP-Cache: Load-balanced, Redundancy-free Cluster Caching with Selective Partition," accepted by *IEEE/ACM Int. Conf. High Perform. Comput., Netw., Storage, and Analysis (SC), Nov. 2018*.

The Challenge of Load Imbalance



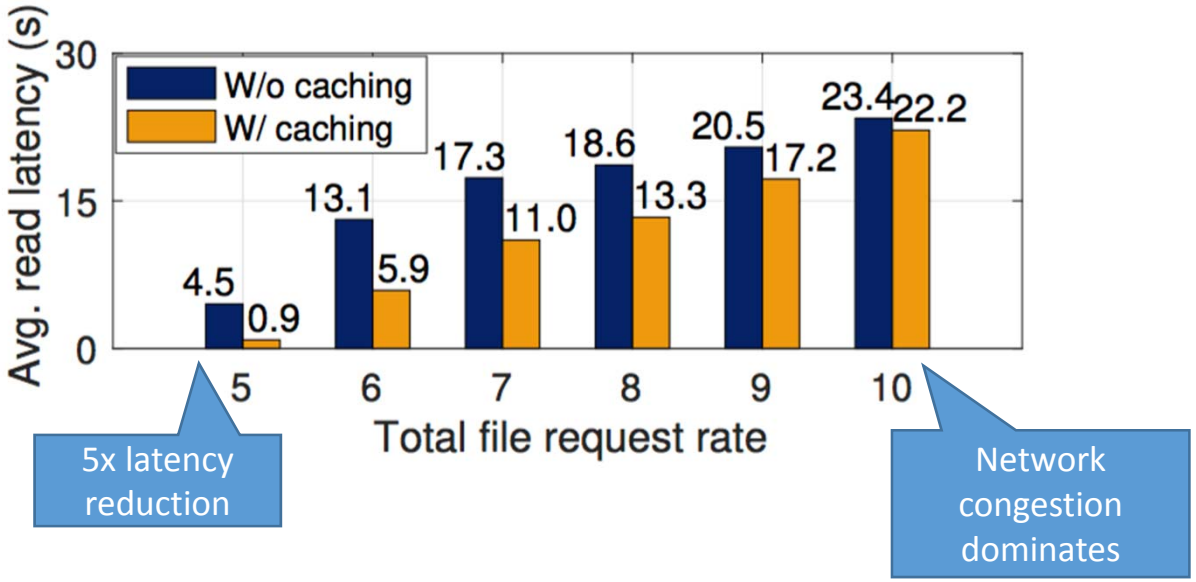
Root Causes

1. Skewed file popularity.
2. Hot files are usually of large sizes.



Distribution of file popularity and size observed in a Yahoo! cluster.

Load-Imbalance Cancels out the Caching Benefit



The read latencies with and without caching as the load of hot files increases.

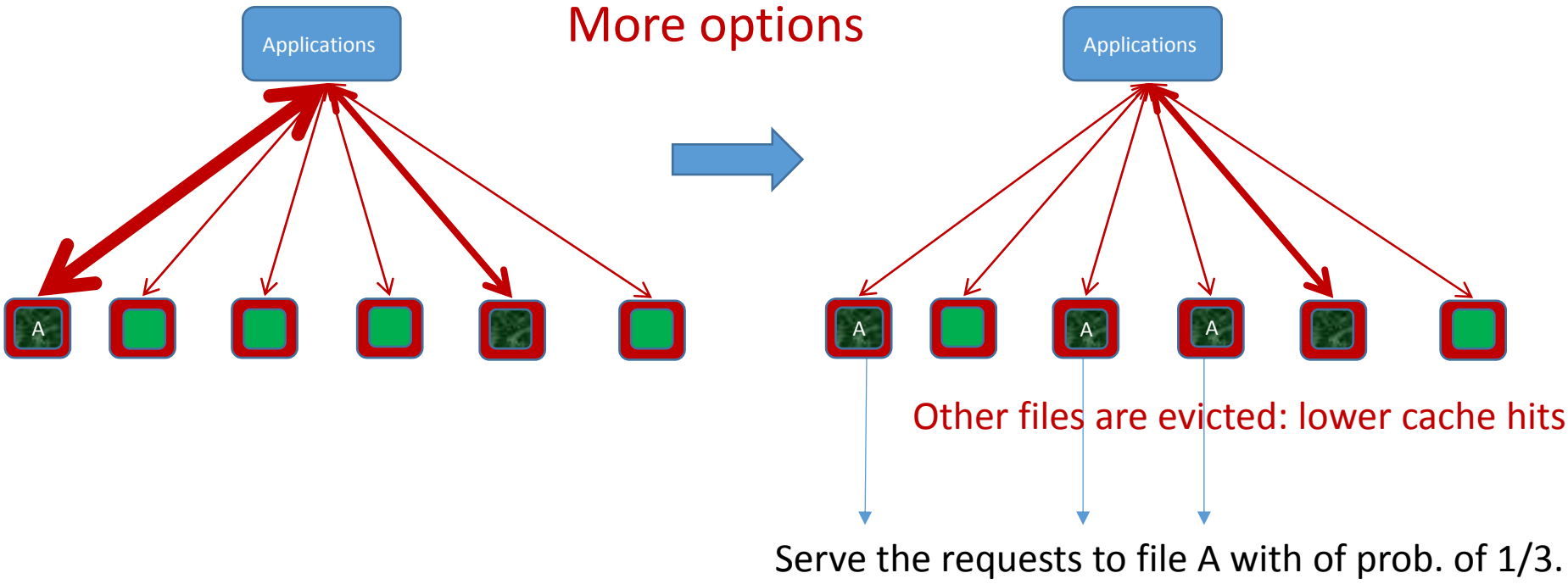
Problems of Existing Solutions

Resort to redundant caching:
(1) copy multiple replicas
(2) create coded partitions.

Replication: High cache overhead
Coding: High compute overhead.

Existing Solutions – 1:
Selective Replication

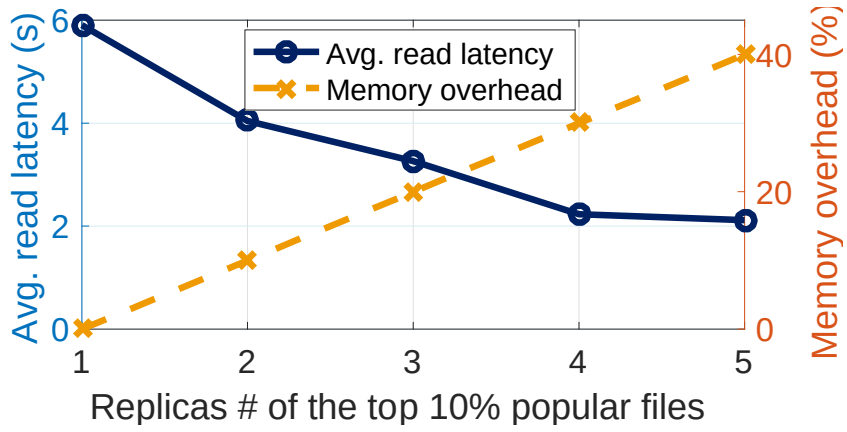
Replicate hot files, e.g., Scarlett [8].



[8] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: coping with skewed content popularity in MapReduce clusters," in *Proc. ACM Eurosys*, 2011.

Existing Solutions – 1:
Selective Replication

High memory redundancy: each replica incurs an additional memory overhead of 1×.



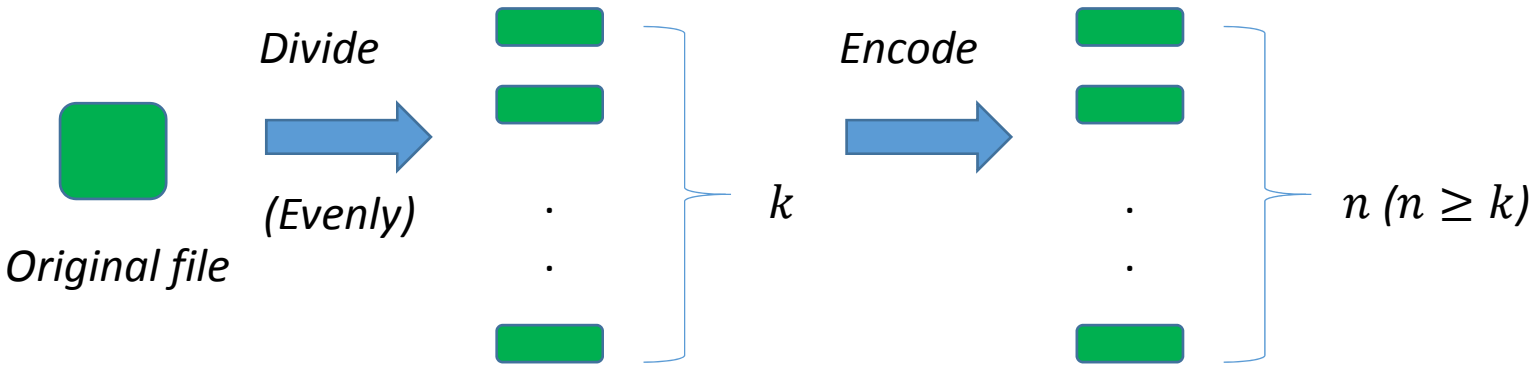
Linearly increasing cost

Sublinear improvement

Existing Solutions – 2:
Erasure Coding

Storage codes, e.g., EC-Cache [9].

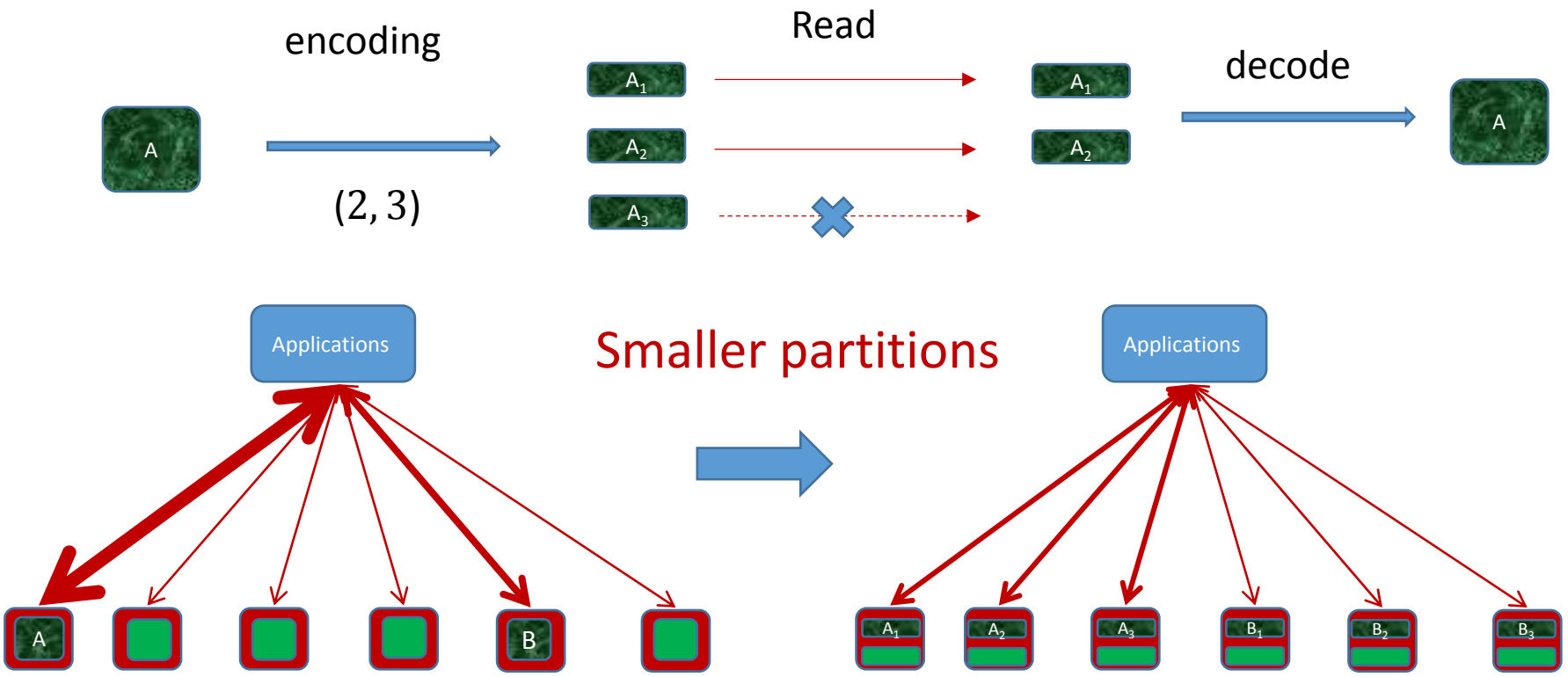
(k, n) coding scheme:



Any k out of the n partitions can recovery the original file.

[9] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding." in *Proc. USENIX OSDI*, 2016.

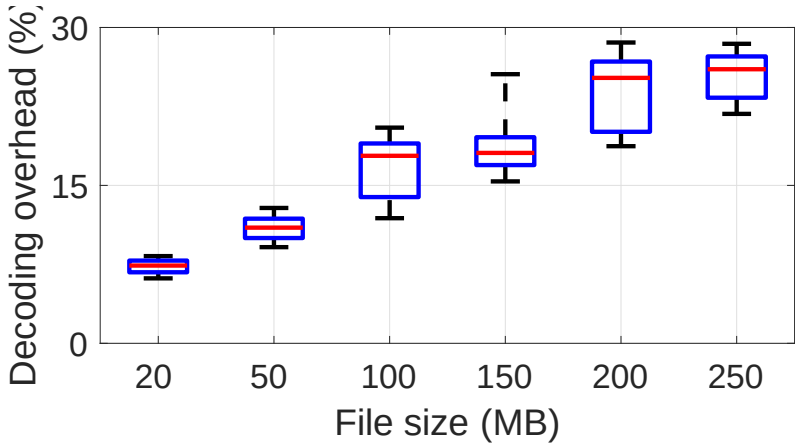
Existing Solutions – 2:
Erasure Coding



The traffic to file A is distributed to three servers.

Existing Solutions – 2:
Erasure Coding

Decoding/encoding overhead



Our Solution: File Partition

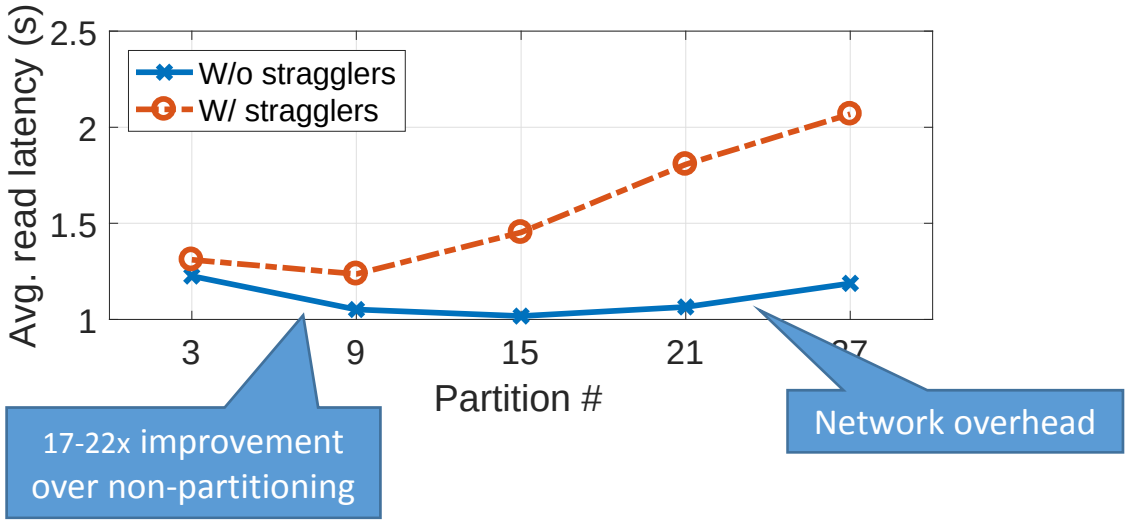
A simple yet effective approach –

splitting files without parity partitions.

No redundancy.
No decoding/encoding overhead.

Spreading the load to multiple partitions.
Read/write parallelism.

Challenges



Partition is unnecessary (and even troublesome) for the *trivial many* small files. → Selective partition

Network overhead: TCP connections.

Stragglers.

→ No need for too many partitions

SP-Cache: Selective Replication

Divide files *in proportion to* their expected loads.

size popularity

$$k_i = \lceil \alpha L_i \rceil = \lceil \alpha S_i P_i \rceil$$

Scale factor(design parameter)

Intuition: Differentiate the vital few from the trivial many.

- A few hot files contribute the most to the load imbalance.
- Mitigate the hotspots without amplifying the straggler effects.

Configuration of the Scale Factor

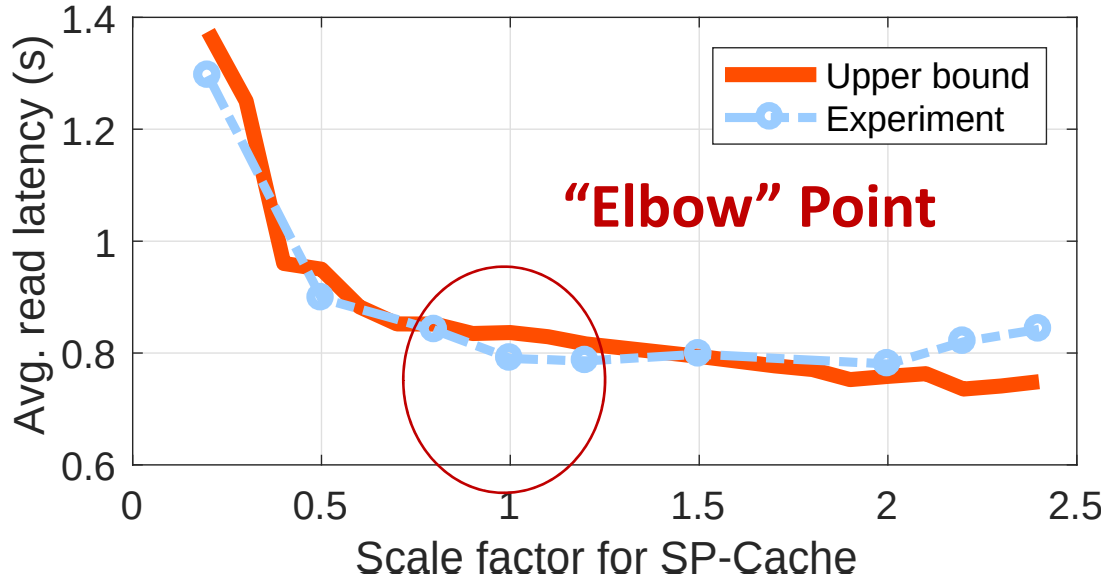
To minimize the straggler effects, α should be **just large enough** to mitigate the hotspots.

Solution:

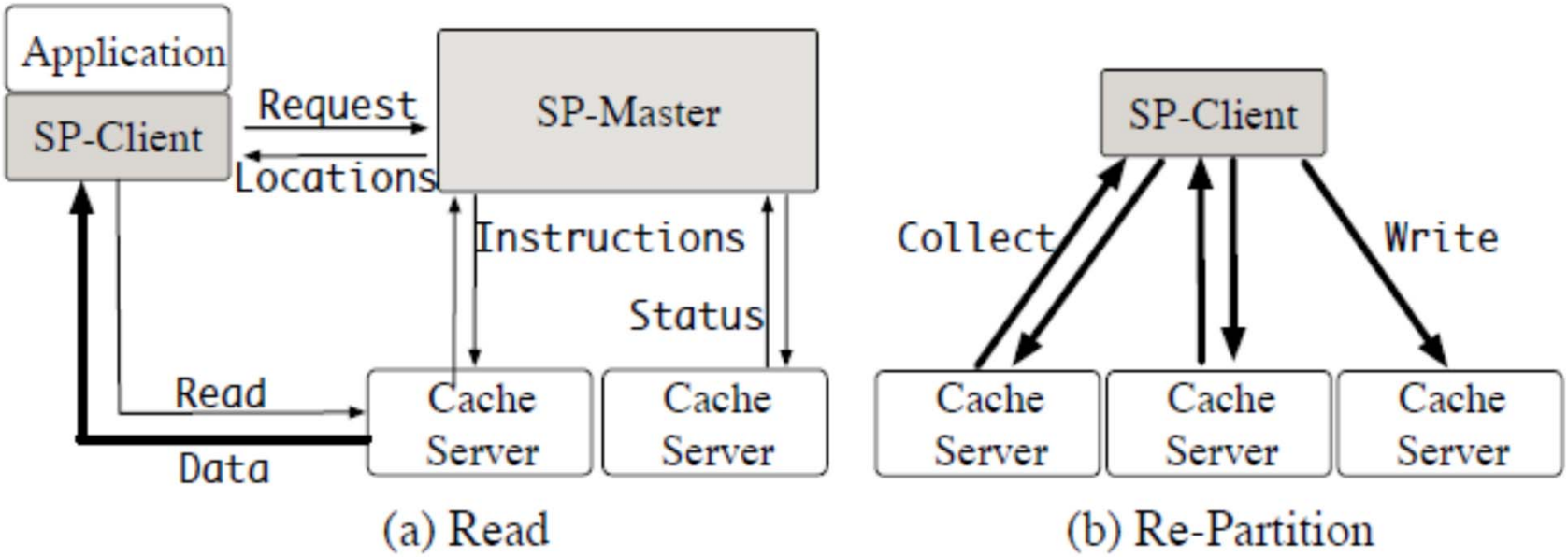
1. Set up a model without stragglers.
2. Initialize α.
3. Gradually increase α, until the improvement of average read latency become marginal.

Find the "Elbow" Point

A tight upper bound to approximate the average read latency.

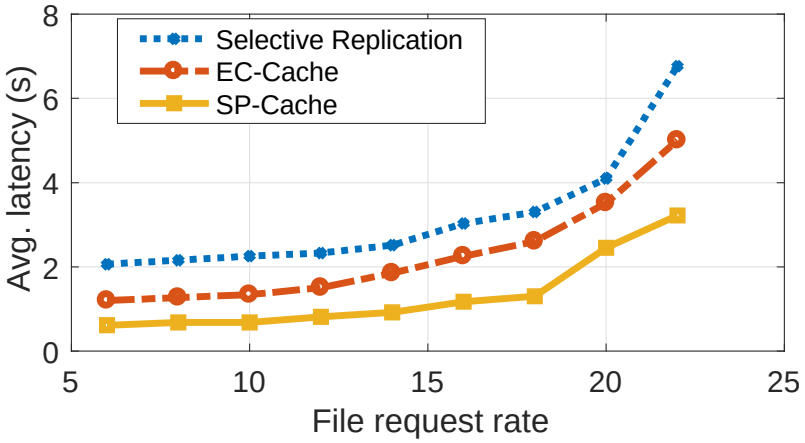


Implementation



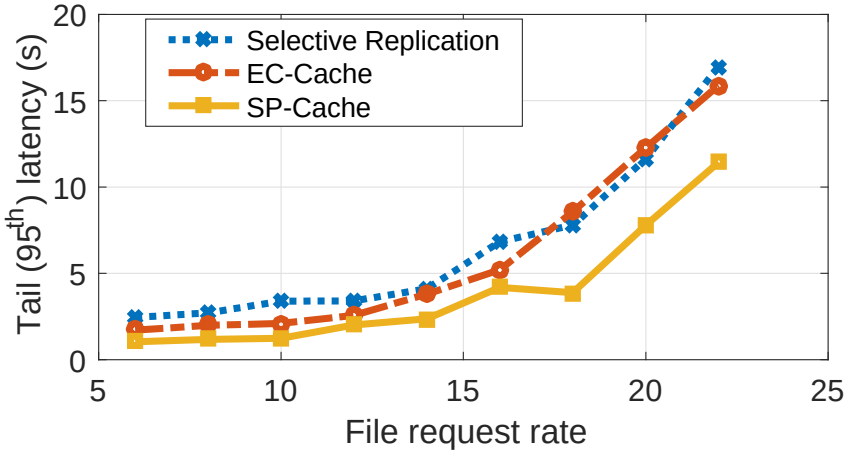
Evaluations – Read Latency

Average latency



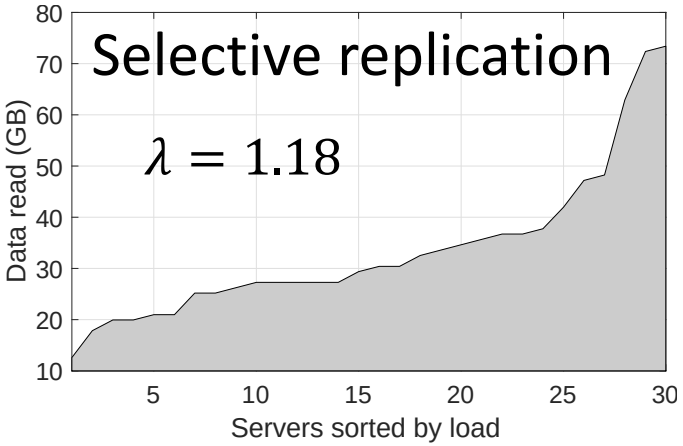
Improvement up to 50% and 70%, respectively

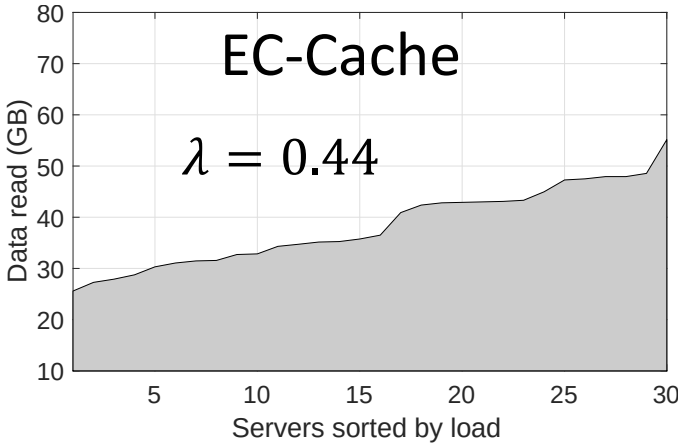
Tail latency



Improvement up to 55% and 60%, respectively

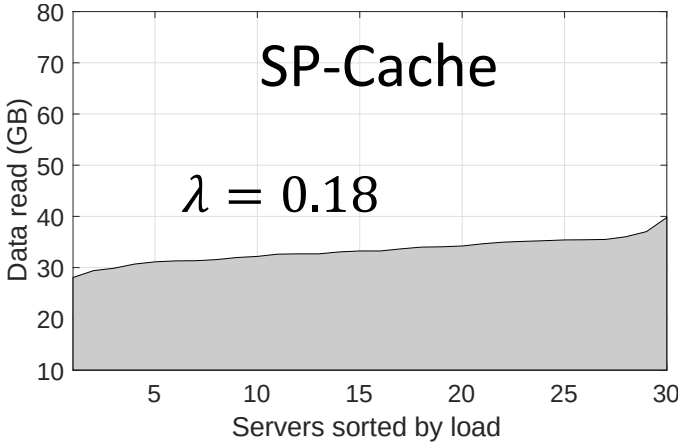
Evaluations – Loads





Imbalance factor

$$\lambda = \frac{l_{max} - l_{avg}}{l_{avg}}$$



Wrap up – Load Balanced Caching

- Load imbalance: reasons and impacts.
- Selective partition: algorithms and analysis.
- Compared to the-state-of-the-art:
  - improve both average and tail latencies by up to **50%**;
  - improve the load-balance by **2.45x**.

Summary

LRC: dependency-aware caching management

- Exploit dependency DAGs.
- Effective cache hit ratio as the metric.

Opus: fair cache sharing built atop the VCG mechanism

- Strategy-proof.
- Isolation guarantee.
- Near optimal efficiency.

SP-Cache: efficient load balancing with selective partition

- Redundancy-free and coding-free.
- Reduces latency and improves load balancing with less memory footprint.