交叉信息研究院
Institute for Interdisciplinary Information Sciences

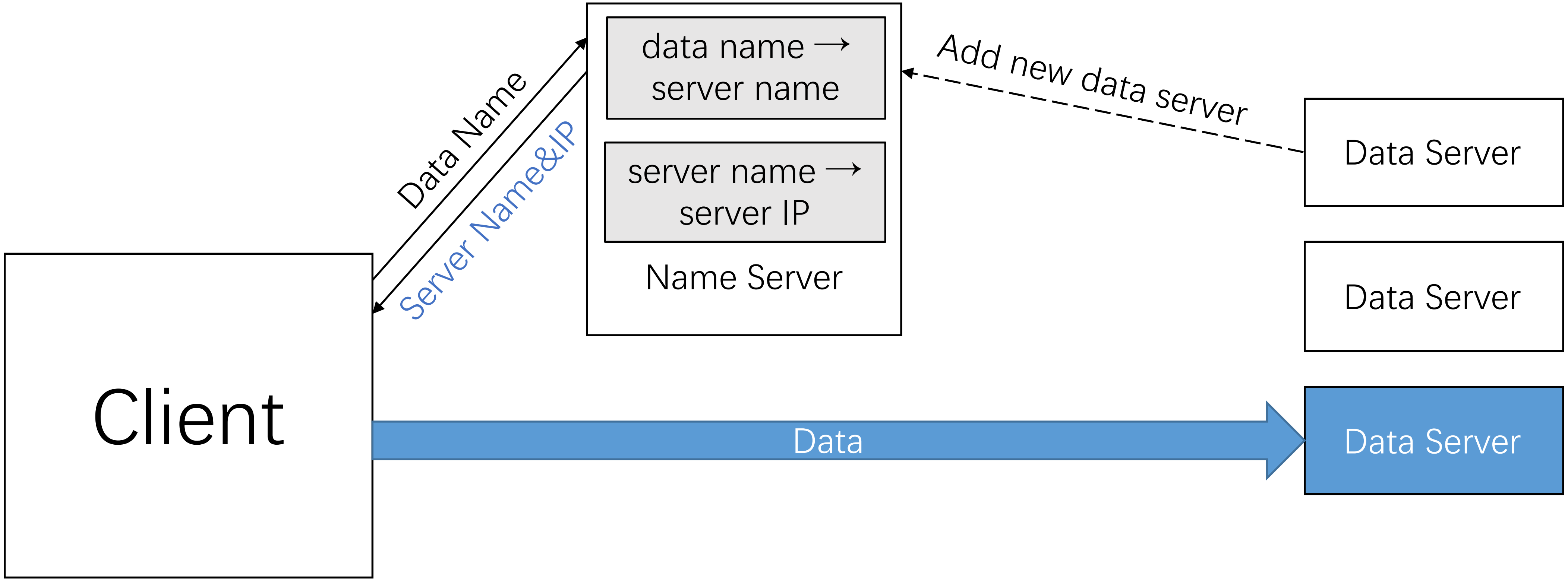
Optimizing Hash-based Distributed Storage Using Client Choices

Peilun Li and Wei Xu

Institute for Interdisciplinary Information Sciences, Tsinghua University

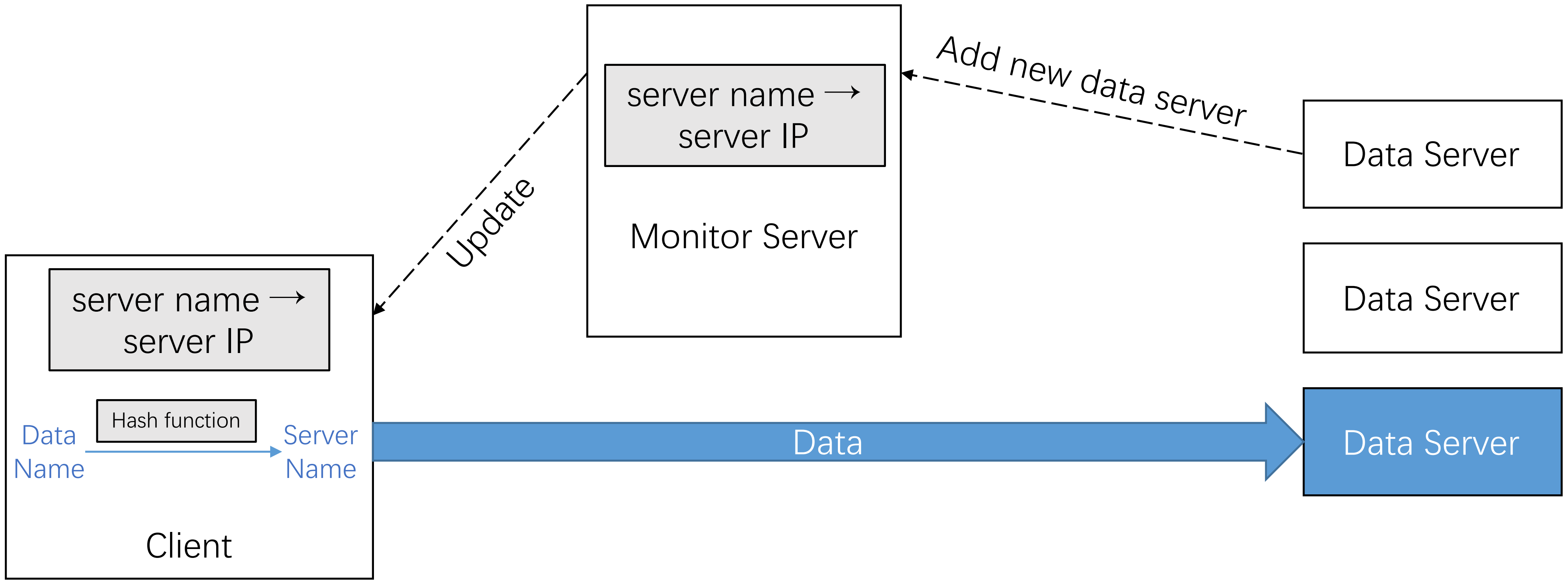
Data Placement Design #1

- Centralized management: GFS, HDFS, …



Data Placement Design #2

- Hash-based distributed management: Ceph, Dynamo, FDS, …



Pros and Cons of Different Designs

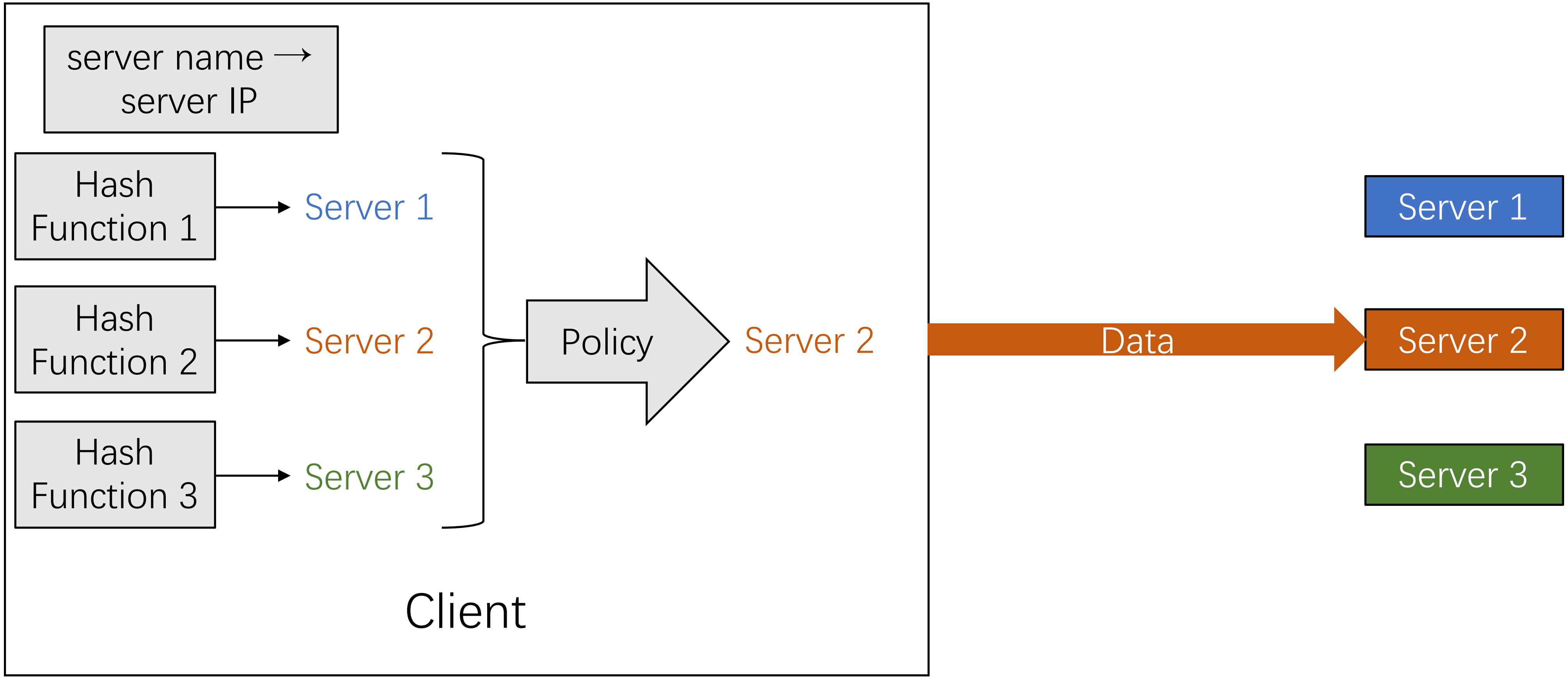
	Pros	Cons
Centralized Management	Global performance optimization.	Centralized name server can become bottleneck.
Hash-based Distributed Management	Avoid centralized server bottleneck.	Fixed placement makes it hard to do optimization. Some optimization is vulnerable to change of lower-level storage architectures.

Motivation

- We want to use server information to improve system performance in hash-based distributed management.
  - Static information: network structure, failure domain, …
  - Dynamic information: latency, memory utilization, …

- We want a flexible system so that new optimizations for specific applications can be added easily.
  - Do not want to redesign the whole placement algorithm or hash function.

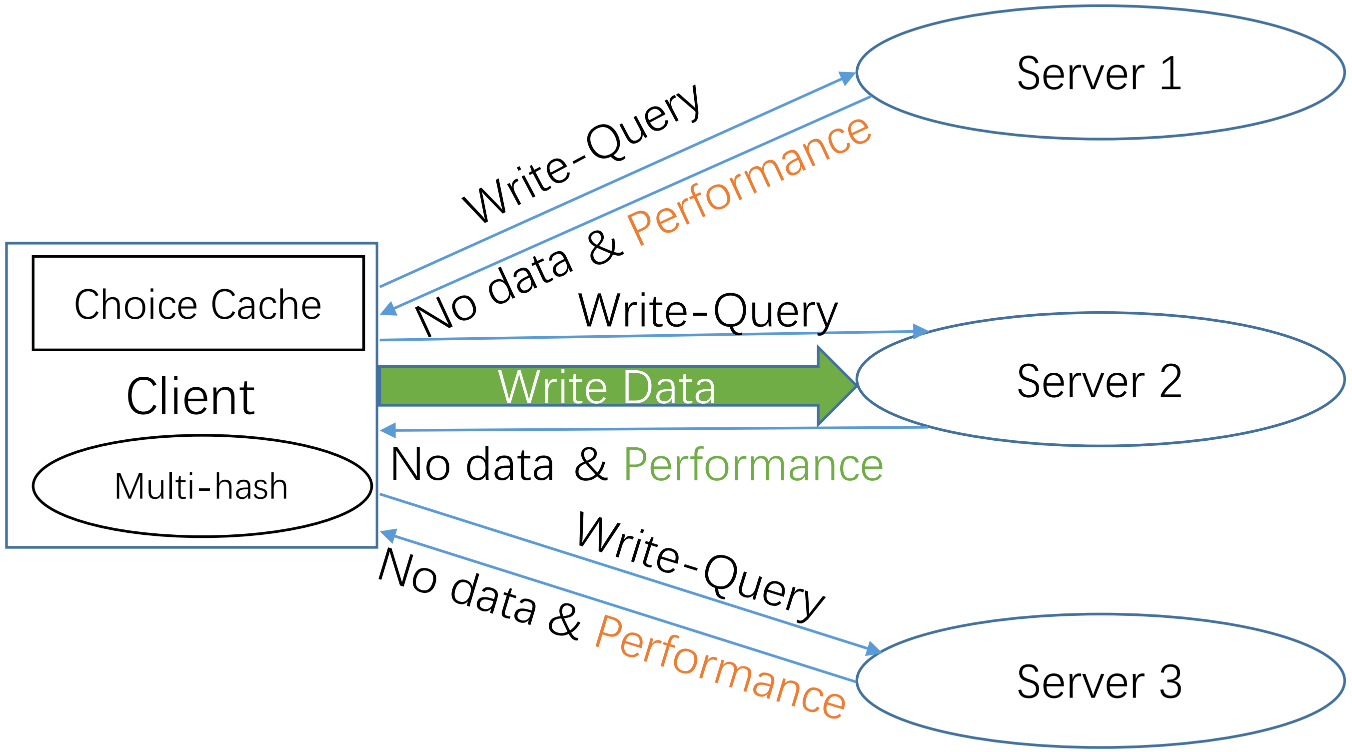
Solution: Multiple Hash Functions



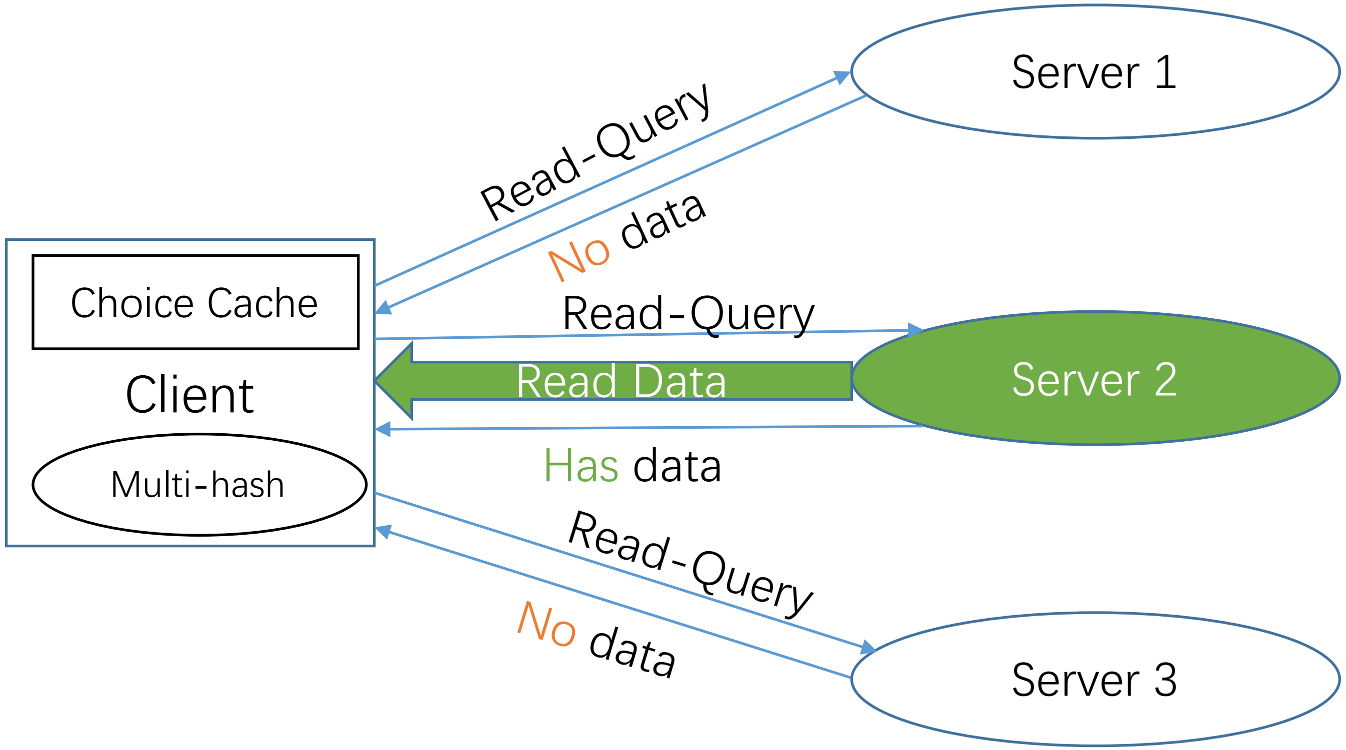
Solution: Multiple Hash Functions

- We can use multiple hash functions to provide multiple choices, and choose the best one with a fixed policy.
  - Different servers provide different performance.
- A performance requirement or even a specific application can have their own optimization policy.
  - Easy to be implemented as an independent module.

How does Write Work Now?



How does Read Work Now?



Simple Server

- Gather server performance metrics.
  - CPU/memory/disk utilization, average read/write latency, unflushed journal size, …
- Answer client probing.
  - Check whether the requested data exist on this server or not.
  - Piggyback server metrics with probing results.

Clever Client

- Provide multiple choices.

- Probe server choices before the first access.
  - Make a choice if need to write new data.

- Cache the choice after the first access.

Making the Best Choice

- A **policy** gets server information as input and output the best choice.
- Example policies:

Choice Type	Choose the server with ...
`local`	closest distance to the client
`memory`	lowest memory utilization
`cpu`	lowest cpu utilization
`space`	lowest disk utilization
`latency`	lowest recent latency
`journal`	least unflushed data in journal

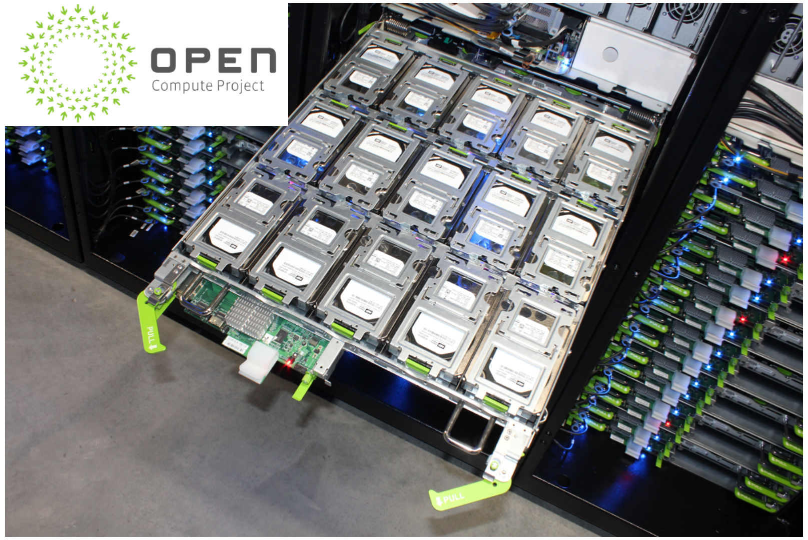
Implementation

- We implement it based on Ceph.

- About 140 lines of C++ codes for server module.
  - Easy to be implemented on other systems.

- Only support block device interface now.
  - It ensures that only one client is accessing the block device data.

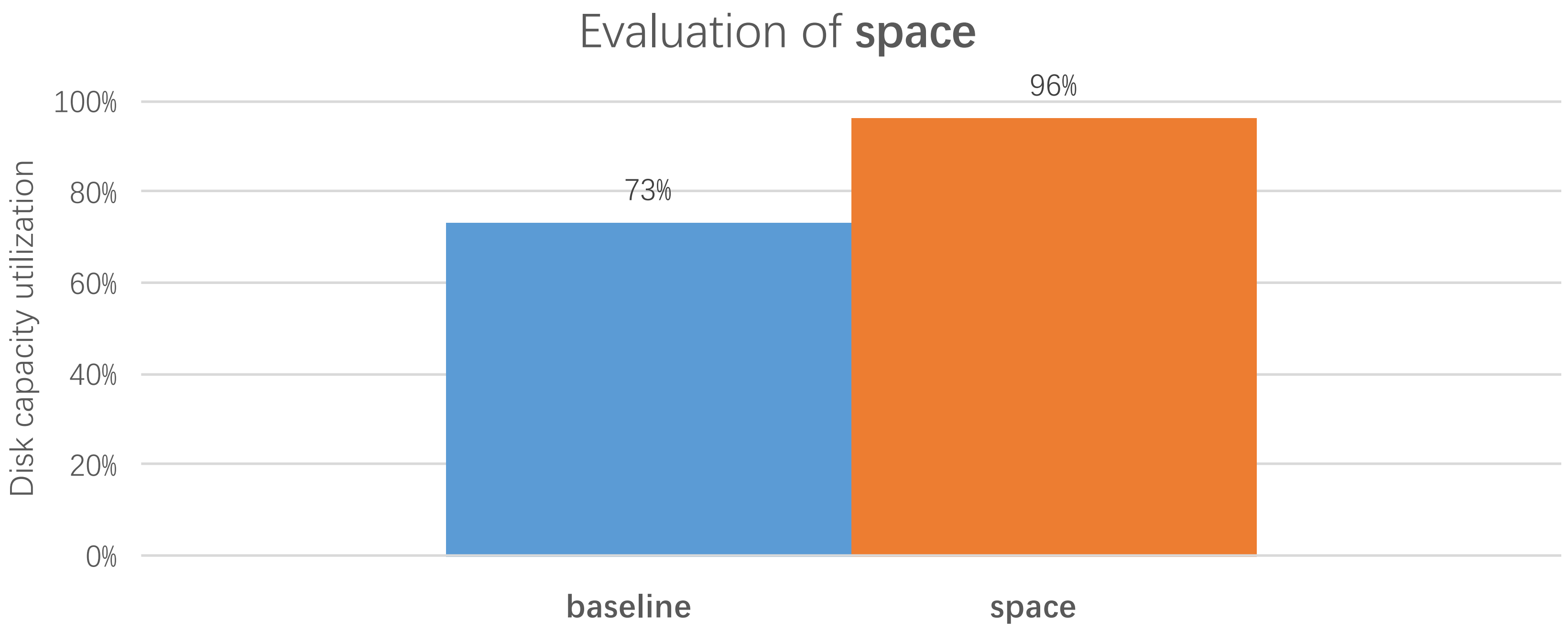
Evaluation Setup



- Testbed cluster.
  - 3 machines.
    - 15*4TB hard drives
    - 2*12 cores 2.1GHz Xeon CPU
    - 128 GB memory
    - 10Gb NIC.
  - Workloads are generated with librbd engine of FIO. 8 images are read/written with 4MB block size concurrently on the same machine.
- Production cluster.
  - 44 machines.
    - 4*4TB hard drives and 256GB SSD.
    - 2 10Gb NICs.
  - Workloads are generated with webserver module of FileBench.
- The number of choice is fixed to 2.

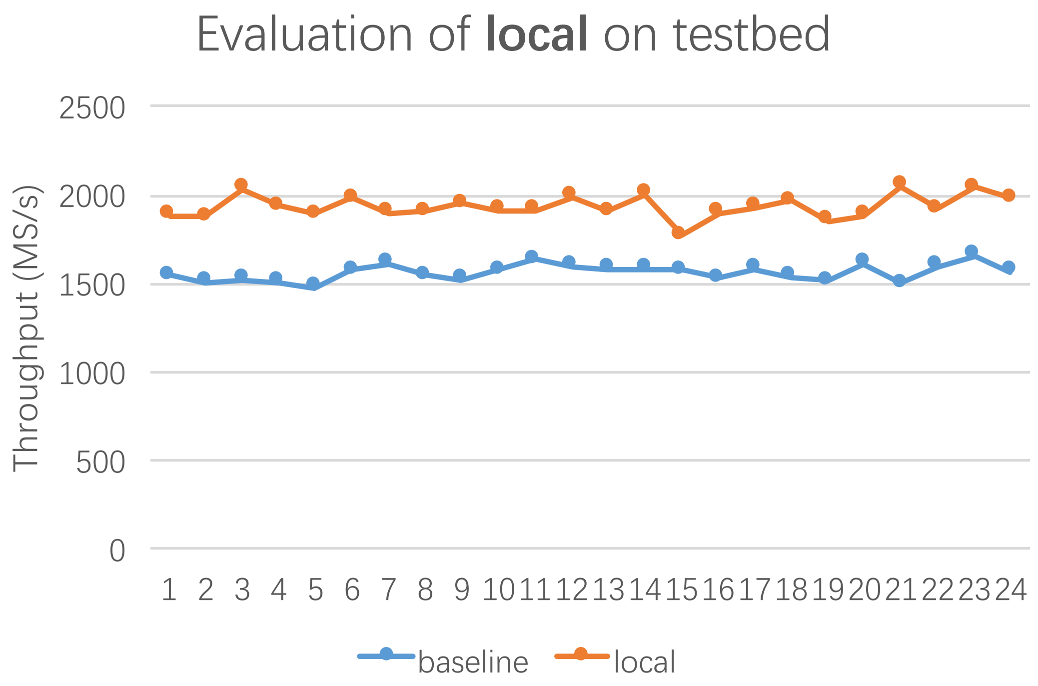
Policy **space** Saves Disk Space

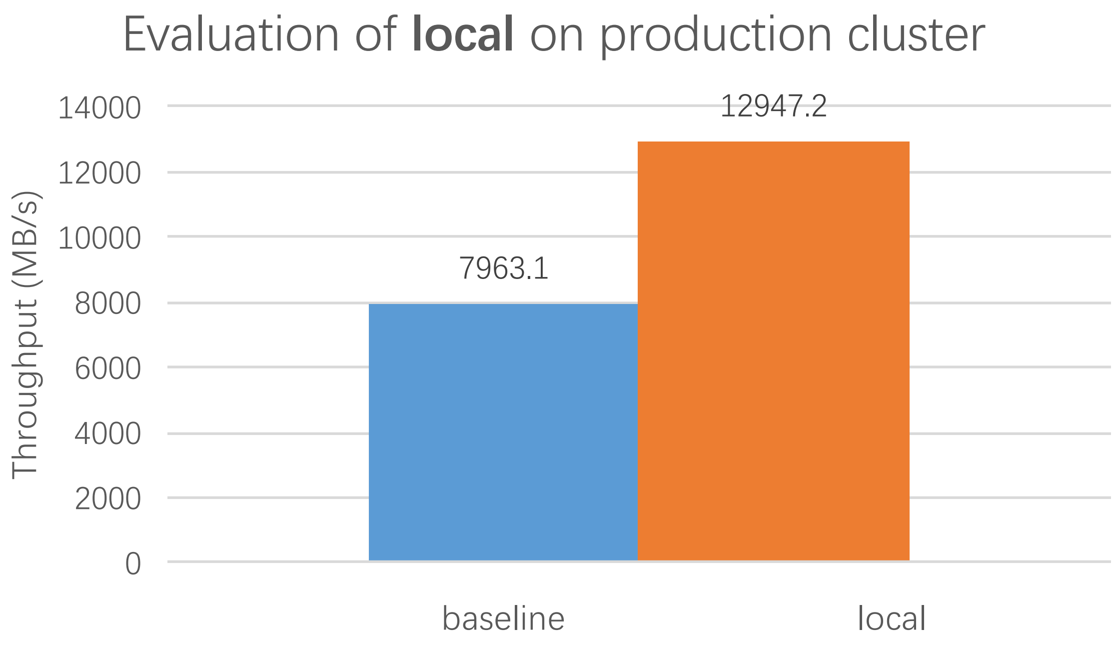
- **space** chooses the server with most free space to store data.
  - A hash-based storage system is full when there is one full disk.



Policy **local** Reduces Network Bottleneck

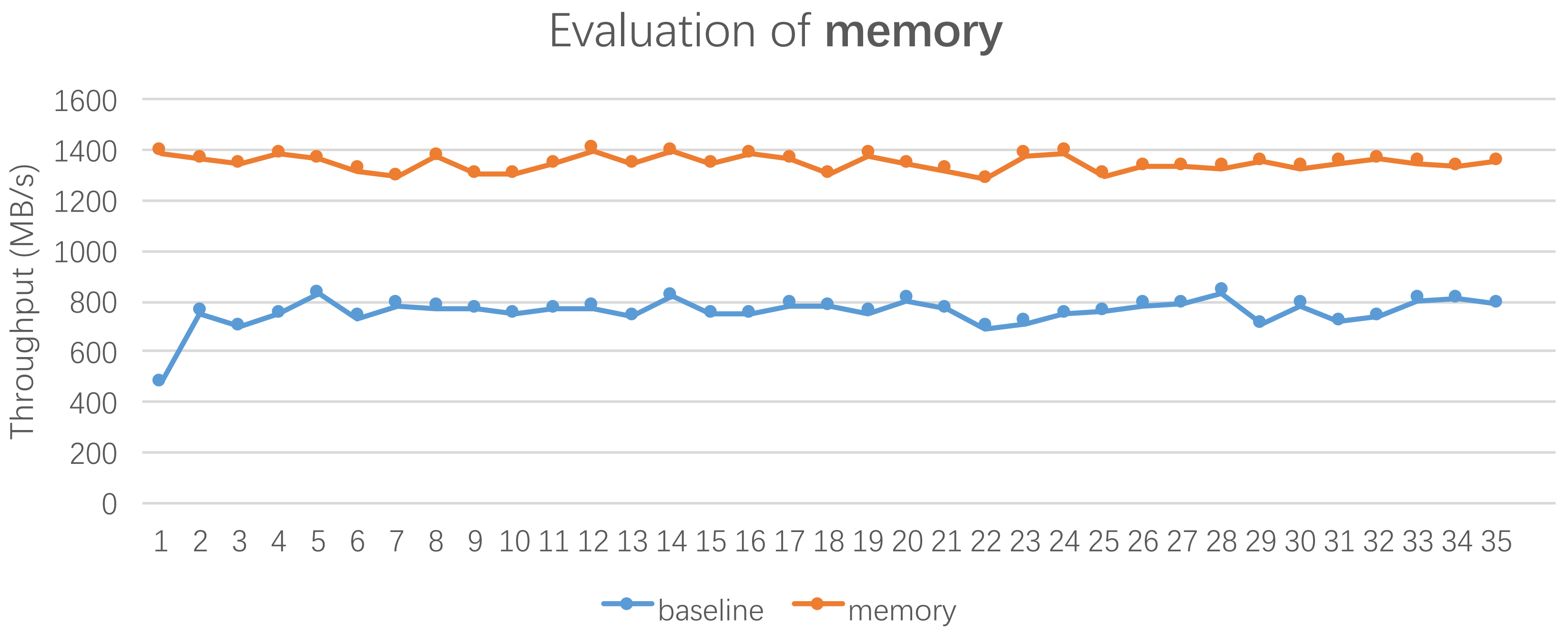
- **local** chooses the closest server to store data.
  - Can save cross-rack network bandwidth.





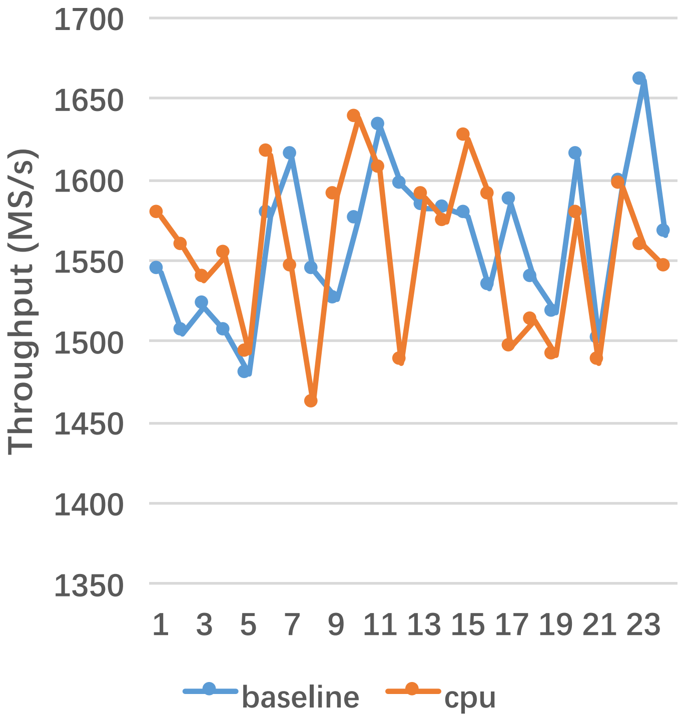
Policy **memory** Improves Read Throughput

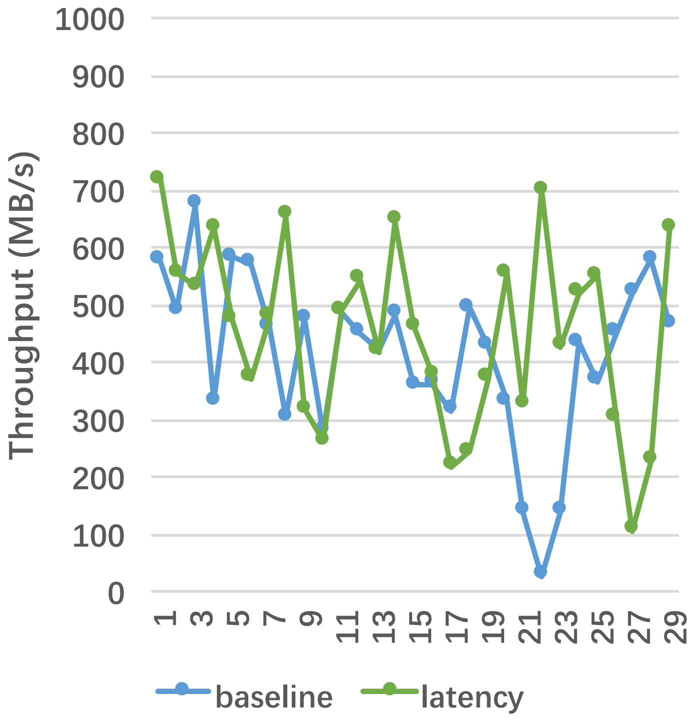
- **memory** chooses the server with the most free memory.
  - Coexist with other running programs
  - More free memory => more file systems buffer => better read perf.

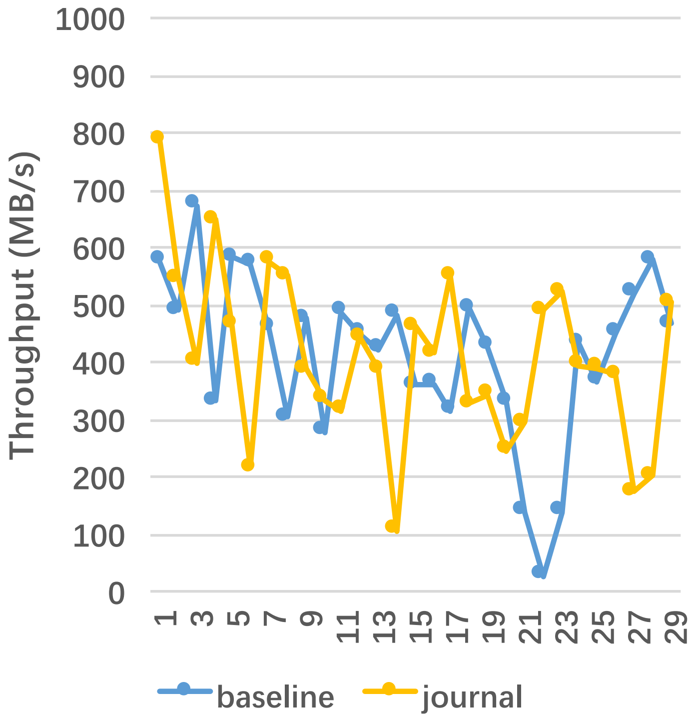


Inefficient Policies

- Policies **cpu**, **latency**, and **journal** do not work well.







Why are They Inefficient?

- The Ceph server is not CPU intensive under this hardware configuration.
- Queue-based transient metrics, e.g. unflushed journal size, changes too fast, so we can not have a consistent measurement.

- However, applying ineffective policies still provide similar performance of the baseline!

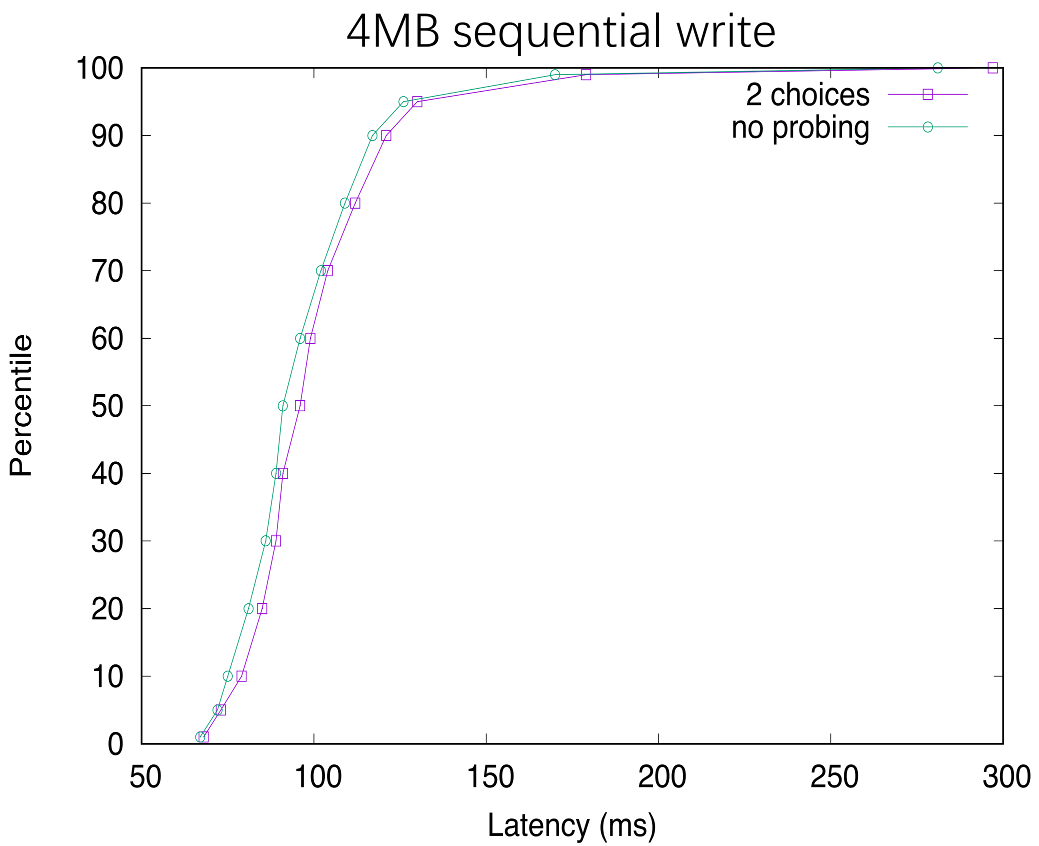
Summary of Different Policies

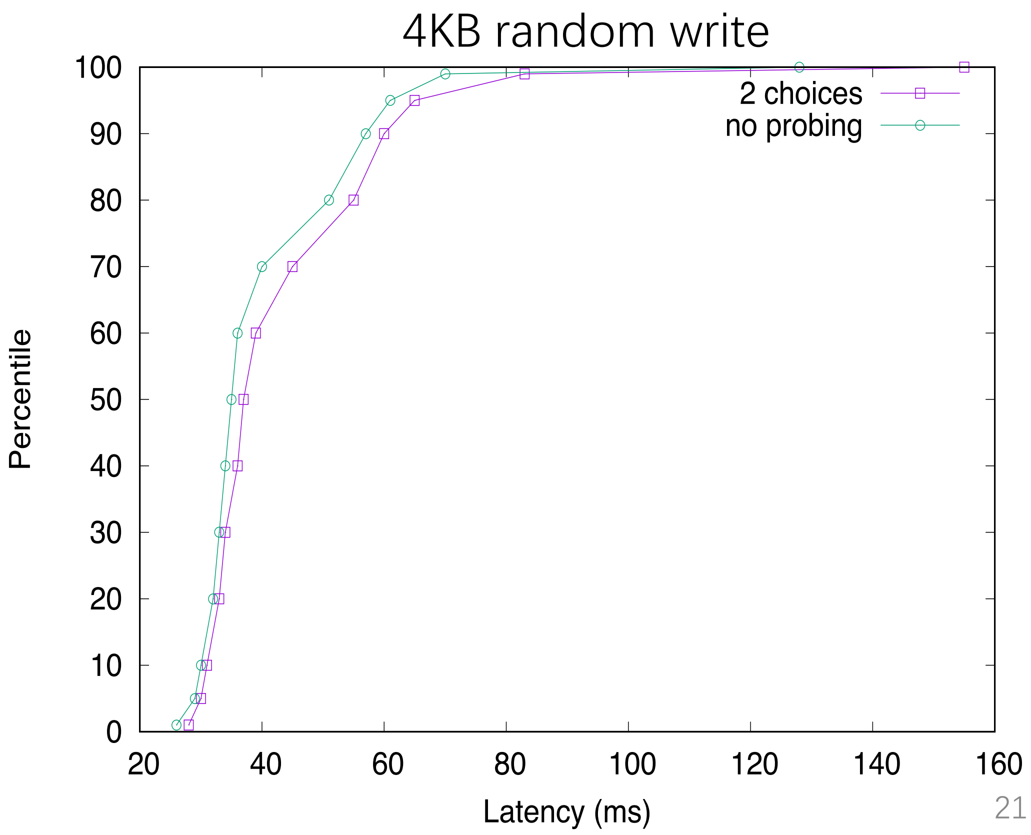
- General improvement:

Policy	Performance Change	Improvement
local	1545 MB/s → 1900 MB/s	23.0%
memory	778 MB/s → 1403 MB/s	80.3%
space	73% → 96%	31.5%
cpu	1545 MB/s → 1513MB/s	-1.9%
latency	402 MB/s → 396MB/s	-1.5%
journal	402MB/s → 396MB/s	-1.5%

Probing Overhead

- The most significant overhead is server probing.





Discussion about Probing Overhead

- It has 2.7ms average latency overhead for probing because of an extra round trip time.

- Latency is increased by 2.7% for large sequential write and 6.9% for small random write.

- The probing is only done in the first access at a client.
  - The overhead is distributed to all subsequent accesses of an object.

Future Work

- Develop more advanced choice policies based on multiple metrics.

- Provide an application-level API, so the application itself can make the choices.

- Exploring different ways to collaboratively cache the choice information, in order to reduce the number of probing.

Conclusion

- Hash-based design in distributed systems can be flexible as well.
- Statistic optimization with best efforts can be both simple and efficient.
- Without significant queueing effects, the power of two may not work well in a real computer system.

Thank You

清華大學
Tsinghua University

交叉信息研究院
Institute for Interdisciplinary Information Sciences

We are hiring: faculty members, postdocs in any CS field
contact: weixu@tsinghua.edu.cn