Optimizing Hash-based Distributed Storage Using Client Choices

Peilun Li Wei Xu

Institute for Interdisciplinary Information Sciences, Tsinghua University

lpl15@mails.tsinghua.edu.cn weixu@mail.tsinghua.edu.cn

Abstract

Many distributed storage systems use hash-based methods for block placement. While hashing improves scalability, it lacks the flexibility that modern applications need for performance optimization. We propose CHOICE, a design allowing clients to have multiple choices for block placement. It also provides the client with relevant server performance metrics so the clients can implement their own choice policy for performance optimization such as choosing better locality or less busy servers. CHOICE requires minimal changes to the storage server and thus easy to deploy. We have implemented it in Ceph, a popular open-source distributed storage system. On two real Ceph clusters with 45 and 176 disks respectively, we show that we can greatly improve performance using the right placement policy.

1. Introduction

In a distributed storage system, in addition to storing all the data blocks, it is essential to manage the metadata, such as the block placement. There are two typical ways to handle metadata management. The first way is to use a logically centralized server. For example, GFS [10] and HDFS [4] all use a single master to manage block placement. The central server provides better opportunity to optimize the block placement, but it can be a scalability bottleneck. GFS uses large block sizes to reduce the amount of metadata. However, the later Colossus [7] has to change the master to a distributed object storage to scale further.

To handle lots of small data objects in a storage system, consistent hashing [11] is another commonly used method for block placements. Systems like Dynamo [8] or FDS (Flat Datacenter Storage) [16] all use consistent hashing. As the locations of the data blocks are computed using a hash function, there are fewer states to be maintained on the master, leading to better scalability. There are many optimized versions of consistent hashing, for example, CRUSH [24] optimizes the disaster recovery efficiency in consistent hashing.

One problem with consistent hashing is that the hash function determines each block placement, and thus there is almost no flexibility in where to place a data block. Unfortunately, this flexibility is becoming more important nowadays, mainly for two reasons:

1) With the development of virtualization and cloud technology, many applications run on top of the distributed storage. These applications may have quite different storage access patterns. For example, if a block is used as a part of a volume attached to a virtual machine, it will be desirable to place the block closer to the virtual machine to exploit better locality, while keeping the blocks of the same volume on separate disks to increase aggregated bandwidth. This is because the virtual machine is likely to be the only one accessing the block.

2) Many data centers now put storage services and user-facing applications on the same rack, or even on the same set of servers. They also share the same network fabric for both application communication and storage accesses. Thus, there might be a resource contention, e.g., for CPU, memory or network bandwidth, between the storage and the applications, leading to variable performance at each storage node. The applications with higher service level objectives may want to choose the nodes with lower workload to achieve better performance, while other applications may want to choose any node with low cost.

In this paper, we propose CHOICE, a design that provides certain flexibility in storage systems, while retaining all the scalability benefits with consistent hashing. Our solution also allows each application to use customized placement policies to meet its own requirements, making the storage system "software defined".

Our key idea is using *multiple hashing* [12] to provide an application with several choices for block placement. These possible choices are annotated by various metrics describing the server's status, such as current average latency, CPU/disk/memory utilization, location in the network and so on, therefore the application can make an informed choice. We can either let the storage system client make the choice for the application to avoid modifying the application, or we

APSys '16, August 04-05, 2016, Hong Kong, Hong Kong

DOI: http://dx.doi.org/10.1145/2967360.2967365

can provide an interface at the client, letting the user application code to make the choice itself.

Of course, we do not provide the application with *arbitrary* choices, but just a limited set. Note that the problem is different from the *power-of-two-choices* design usually used in network traffic engineering [27] and also in storage [2]. This is because the choices are not based on queue lengths, which is a transient state and quite predictable for a short term. The block placement decision is a longer-term decision, considering many potential factors.

We have implemented CHOICE on Ceph [23], a popular open source distributed storage system. We choose Ceph not only because it is widely used, but also because it uses a complex and highly optimized two-level consistent hashing that represents an advanced case for CHOICE. We performed our evaluation on both a 172-disk production Ceph cluster and a 45-disk testing cluster. The preliminary result shows that even with a small number of choices (2-3), we can achieve improvements in different types of applications. For example, by optimizing the block locality, we can improve the average throughput of the virtual machine volume storage by 62.6%; by optimizing the load balancing in block placement, we can increase overall storage capacity by 31.5%; and by choosing less busy nodes for block placement, we can improve the average read throughput by 80.4%. Interestingly, many seemed-plausible placement policies do not work well, we report these negative results and their reasons in this paper too.

In summary, our contributions in this paper include:

1) We propose a multi-hashing algorithm that adds significant flexibility in block placement with only small changes to existing storage servers.

2) We propose and evaluate several potential block placement policies, including three that work well and the other three that do not.

3) We provide a CHOICE implementation on Ceph, and performed real evaluations on a 44-node, 172-disk cluster.

The rest of the paper is organized as following: Section 2 reviews the related work. We represent our system design in Section 3 and detailed implementation on Ceph in Section 4. Then we evaluate our policies with different workloads in Section 5 and conclude in Section 6.

2. Related Work

Consistent Hashing and Multiple Hashing. Hashing is commonly used for data or metadata placement in distributed storage systems like FDS [16], ShardFS [28], CalvinFS [21], IndexFS [18], Giga+ [17], Dynamo [8], and GlusterFS [9]. It is also used to place data in single-machine key-value storage systems [13, 26].

The 2-choice hashing has been used to reduce collision rate in hash tables for a long time, and the power of two choices is also known to be able to provide good performance improvement in queueing theory [14]. At the same time multiple hashing is used in load balancing [27] and IP lookup [5]. The power of choices is also adopted in distributed computing framework to dynamically schedule computing tasks [22]. To our knowledge, using multiple hashing to improve data placement has not been well studied.

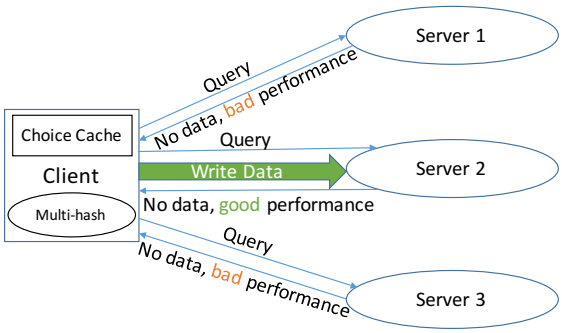


Figure 1. Overview of CHOICE

Data placement optimization. Many storage systems optimize data placement to improve performance, but most of them require centralized control. HDFS [4] moves the computation closer to the data to improve locality, but this method is not usually an option in general storage systems. F4 [15] optimizes data placement using metrics like space utilization. Pileus [20] allows the client to choose the server for latency or consistency. Based on Pileus, Tuba [1] uses automatic reconfiguration to provide more choices. CFS [6] provides choices of data placement. All these systems require a centralized master, while our design allows clients to make optimization decisions in a fully distributed system.

FDS [16] and other similar systems focus on distributing data more evenly to provide high throughput and fast recovery. It gets good improvement in parallelizing data access by modifying the placement algorithm, but it is not flexible enough to handle other application requirements. We use multiple hashing to provide a limited set of choices to improve the *probability* of having a good placement, which is an effective trade-off between performance and complexity. The choice strategies work as hints instead of strict requirements, so the applications can tolerate occasional mismatch between requirements from strategy and the underlying resources.

3. System Design

In order not to add too much complexity to the already heavy-duty servers, we decide to implement most of the CHOICE functionalities on the client side only. This choice also makes it easier to add CHOICE to production software as the modification to the server code is simple and easy to verify by the operators.

Figure 1 shows the main components of CHOICE. There is almost no change to the servers, as they handle the client

queries as before. The client is responsible for generating multiple choices and choosing the best one using a placement policy engine. The client also has to ensure that the choices are consistent, both among different clients and over time. We discuss about the design details in this section.

3.1 Server Module

Our key idea is to keep the changes to the servers as small as possible so that we can implement CHOICE on different consistent hash-based storage systems. At minimal, the server module only needs to provide performance monitoring.

In order to implement meaningful choice policies on the client side, the server needs to provide performance status to the client. For example, we let the servers to provide CPU utilization, memory utilization, location of the server in the network, read/write latency and so on.

3.2 Client Module

In CHOICE, we put most of the program logic in the client code, in order to avoid changing the already-too-complex servers. There are three main considerations in the client design: 1) providing multiple hash functions, 2) making good choices using a policy engine; 3) avoiding inconsistencies with previous and other clients' choices.

1) Providing the choices

We use multiple hash functions to map the data block to multiple servers. By applying multiple hash functions in the client module, we can provide the application with multiple server choices. Note that different clients must use the same set of hash functions, so they see the same set of choices throughout the life of the system for a given object ID. This remains true even if we are adding or removing servers from the system (similar to consistent hashing).

2) Making the choice

After providing the choices, the clients should make the choice when it accesses the block for the first time. If it is a read operation, the client needs to probe all choices to find which one actually contains the data.

For writes, it depends on a policy engine (Section 3.3) to choose a good one. In order to make informed decisions, it needs to query the servers for their performance. The client can query in-band within the storage system or out-of-band using a dedicated monitoring system. It is also important to choose an appropriate data collection frequency as a trade-off between server load and data accuracy. Our current implementation performs these queries together with other client-server communication to improve performance (details in Section 4.4).

The client caches the query results to avoid repeated probing during a read. Note that the cache here is only for block locations, and we do not change any data caching mechanism in the existing client.

Choice Type	Choose the server with ...	Effective
`local`	closest distance to the client	yes
`memory`	lowest memory utilization	yes
`cpu`	lowest cpu utilization	no
`space`	lowest disk utilization	yes
`latency`	lowest recent latency	no
`journal`	least unflushed data in journal	no

Table 1. Choice Strategies.

3) Making consistent choices

One important consistency requirement in the system is that we have to make the same choice for the same block at different time and across different clients. That is, if some client has made a choice on block A, all subsequent choices for block A must be the same - otherwise there will be multiple inconsistent copies of block A. This is hard as only the client itself knows about the choice.

We design a probing mechanism on client writes to avoid inconsistency. Before the client writes to an object, it probes every server in all choices to see whether this object has already existed. If so, the client will skip the policy choice and use the existing one.

Similarly, we need to make sure that different clients make consistent choices about a single data block. The same probing mechanism helps except for two cases: concurrent writes to the same block from different clients, and client failure in the middle of a write. Fortunately, many storage clients, including the Ceph client we use, already have mechanisms to prevent writes to a single block concurrently. If a client recovers from failure, it may not remember its previous choice that has not been done yet. Thus we force it to re-probe all blocks in doubt upon recovery.

3.3 Policy Engine

Using the mechanism discussed above, a client may have different choices, and each choice comes with a number of server metrics. We provide an extensible interface allowing the client to implement different policy engines.

In this paper, we implement some simple policies described in Table 1. We choose these policies based on intuition about both transient and persistent system properties that we thought may affect performance. However, while some policies work very well, others do not work as we expect, and we discuss the facts in Section 5.1. None of our policies use historical workload data, and we leave the history data based policies for future work.

How many choices shall we offer? Although more choices mean a higher probability of finding a "good" placement, it comes with performance cost of more probing and book keeping on the clients.

We emphasis that the problem here is different from the power-of-two-choices in queuing theory, where the queue is the biggest factor impacting performance. In our case, as the writes and subsequent reads are separated in time, there is

no significant queuing effect in many cases. Instead, we are optimizing for specific goals such as more balanced block placement for a better overall system load balancing.

In some cases, we can calculate the number of choices. For example, with the `local` policy, we can write the number of choices p as a function of the number of local / remote disks, available bandwidth for remote access etc. We omit the calculation in this paper due to space constraints. In other more dynamic cases that depend on workload metrics, we need to obtain the number with empirical experiments.

4. Implementing CHOICE on Ceph

In this section, we discuss the implementation of CHOICE in Ceph [23], a popular open source distributed storage system that runs on commodity servers. We first introduce basic concepts in Ceph, and then we focus on how we modify the Ceph server and client to support CHOICE features. We show that even with a complicated storage system like Ceph, our change is not only effective, but also simple to implement. In fact, we only change 142 lines of Ceph server codes for 6 different types of choice strategies, and many of the changes are for server status monitoring.

4.1 Background: Ceph Block Placement

Ceph offers three interfaces, filesystem, block device and object storage, using a common object store backend called RADOS. Every file or image stored in Ceph is striped into many small objects (default is 4MB), and every object belongs to a placement group (PG). Every disk in Ceph corresponds to an object storage device (OSD), and a PG is replicated to multiple (default is 3) OSDs.

Each client computes the location of each block by itself. Specifically, each object has a unique ID [25] and the object ID is hashed to a placement group ID (PG-ID). Then, a CRUSH [24] function computes the primary OSD number from the PG-ID. This mechanism prevents overloading the Monitors, the metadata managers in Ceph.

4.2 Providing Multiple Choices

Ceph has two levels of block placement mapping. One is from object ID to PG-ID with normal hash function, and the other is from PG-ID to OSD with CRUSH. We choose to provide multiple hashes in the former step, leaving the latter step unchanged, so we can reuse the delicate disaster recovery or data integrity check mechanisms provided by Ceph placement groups.

The only drawback is that we are no longer able to provide choices on a single OSD granularity. We have to choose 3 OSDs as a single unit instead of choosing individual OSDs directly, which limits our flexibility to some degree, but still enough in most cases.

4.3 Server Module

The queries from clients are sent as `read` messages with some `QUERY` flags set, and they will be processed by the server like normal messages. The server always keeps object metadata in memory, so it is fast to check the object existence. After the checking, the server will see if the request carries `QUERY` bits, and reply with corresponding metrics directly. The reply from server will have the same flags set, so the client can check all `QUERY` bits of `read` request replies to see if they are answering queries.

4.4 Client Module

Providing the choice. The number of choices `pg_choice_num` is a configurable parameter, and we need a hash function for every choice. Thus we need a way to automatically generate a number of hash functions.

Algorithm 1 shows the client procedure to provide placement group choices. For the first choice, the function gets the original object ID as the input. For the next i-th choice, it appends the number i to the object ID. If we have a collision on the output ID (the collision rate is low), we discard it, increase i by 1 and continue. The end result is a set of different PG-IDs with size `pg_choice_num`. No matter how the cluster state changes, the choices will remain the same.

Algorithm 1 Generating PG choices

```
1:  procedure GENERATINGPGID(oid)
2:      pgid[0] ← hash(oid)
3:      j ← 0
4:      for i = 1 to pg_choice_num − 1 do
5:          while True do
6:              j ← j + 1
7:              pg ← hash(oid + string(j))
8:              if pg not in pgid then
9:                  pgid[i] ← pg
10:                 break
11:     return pgid
```

Choosing PG metrics based on strategy. The client reads metrics from each server, but it makes decisions based on PG. We need to combine metrics from different servers, and we take a strategy-dependent approach. For `cpu, journal` and `latency` strategies, we use the average metrics for the entire PG. For `memory` strategy, we only use the memory utilization of the primary OSD. This is because we aim to improve read performance, and read operations only involve the primary OSD. For `space` strategy, we take the highest disk space utilization of all OSDs in this PG. For `local` strategy, we only consider the location of the primary OSD, for the same reason as the `memory` strategy.

Making consistent choices. We combine the query for performance metrics and probing for consistency, as Section 3.1 describes, into a single message. Once we know the PG choice, the rest is no difference from stock Ceph.

The data placement cache is stored as a hash table within every client context. For block device interfaces, every opened image has its own context. The hash table maps

Choice Type	Read Throughput	Average Read Latency
baseline	1545.9MB/s	44.23ms
`local`	1900.2MB/s	33.06ms

Table 2. The average throughput and latency of the reading experiments using baseline, `local` policy and `cpu` policy. We use $1 * 8$ client threads to access 8 images.

Choice Type	Read Throughput	Average Read Latency
baseline	778.0MB/s	41.46ms
`memory`	1403.2MB/s	23.26ms

Table 3. The average throughput and latency of the reading experiments using baseline and `memory` policy. We use $2 * 8$ client threads to access 8 images.

an object ID to a corresponding placement group. A mapping entry is less than 50 bytes. On a 4TB image with 4MB blocks, the choice cache uses less than 50MB of memory.

RADOS block device interface prevents concurrent access. Clients access the entire image instead of individual objects through the interface, and it ensures that each image is accessed exclusively.

Currently we write the placement policy configuration in the clients' Ceph configuration file, so all images a client opens use the same set of configuration. It will be more flexible if we provide these choices to the application, but it requires modifying the user-level applications. We plan to support this API as future work.

5. Evaluation

In this section, we first describe the microbenchmarks of all the choice policies we have implemented on a test cluster, and analyze their performance. We deploy the policies that perform well and show their performance in a production cluster.

We evaluate CHOICE on a Ceph cluster with 44 machines on 3 racks. Each machine has four 3TB hard disks to store data and one 256GB SSD to store journal files. It uses two 10Gb NICs, one for cluster network and the other for public network, respectively.

As many load-related evaluations use too much resource for a production cluster, we also build a small-scale testbed with 3 hosts, each containing fifteen 4TB hard drives. Each server has $2 * 12$ cores 2.1GHz Xeon CPU, 128 GB memory, and one 10Gb NIC. The `librbd` engine of FIO [3] is used to generate workloads, with different number of threads and other parameters. We use a single 10Gb NIC for both public and cluster networks. For simplicity, we deploy the journals on the same disks as the OSD data, so the write operation can only achieve half of the disk throughput, but it is enough to show the performance differences with or without choices.

5.1 Microbenchmarks

We first conduct a series of microbenchmarks on our testing cluster to evaluate the performance of each policy in Table 1. We compare the performance of the policy with the stock Ceph (the baseline), on exactly the same set of hardware. Between each test, we re-create the local file system to ensure a fresh start. We find that `local`, `memory` and `space` work well, while `cpu`, `latency` and `journal` do not.

The `local` policy. The intuition behind the `local` policy is to reduce cross-rack network bandwidth. It is especially useful when there are many disks per server and the network bandwidth is limited.

During the experiment, we first write 8 images with 6GB each. Then we read back these blocks using the same set of clients. Each client sequentially reads an image with 2 threads and 4MB block size. The result is shown in Figure 2(a) and Table 2, we see that the `local` policy improves the throughput by 23.1%, and reduces the latency by 25%.

We believe one reason for the big improvement is that with a total of 48GB data size, most data are cached in the sever memory when read back, and thus the network latency plays the most important role. We believe with the adoption of flash and non-volatile memory, the network is more likely to become the bottleneck and thus locality can be important, even within the same cluster . Note that there are no other processes using the network during the evaluation, and the improvement maybe even better on a shared network.

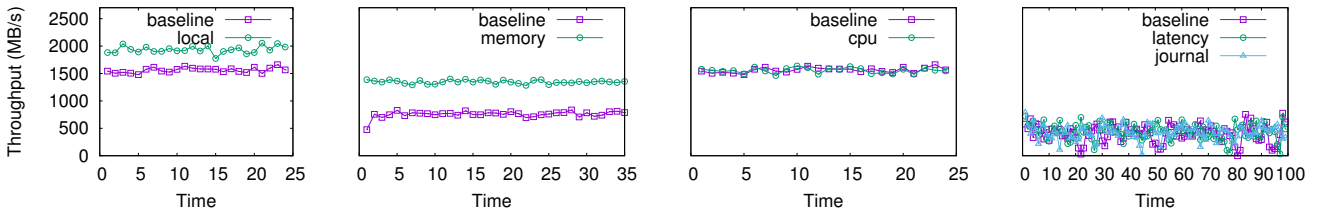
The `memory` policy. The previous `local` policy only uses static topology information, but does not consider the server workload. During the experiment, there is no other workload running on these nodes.

Now we want to evaluate the effect of coexisting applications on the same server. To emulate coexisting applications, we run a program that occupies 48GB of memory on two servers (the testing client runs on one of them), and repeat the same experiment as the `local` case.

Figure 2(b) and Table 3 shows the results. We see that `memory` policy almost doubles throughput and reduces the latency by half. On the contrary, the baseline does not perform well, due to memory contention. This is because the local machine has limited memory space for cache, causing many disk accesses. `memory` policy chooses the one with the highest amount of free memory for cache, and the benefit outweigh the performance loss of using a remote server.

Of course, memory utilization changes in real system. We observe that the memory utilization changes slowly in cloud systems as many tasks (such as a virtual machine) stay for a long time period. Also, a data block is likely to get accessed again in a short period of time. Thus, the memory policy would help if the memory utilization can last long enough.

Load balance. In Ceph, the cluster is considered full if any of the OSD becomes full. In our cluster, we found that we are not able to put in more data after the average disk utilization reaches about 70%, as some data start to be assigned to full



(a) Evaluation of `local` policy (b) Evaluation of `memory` policy (c) Evaluation of `cpu` policy (d) Evaluation of `latency` policy and `journal` policy

Figure 2. (a) and (c) are reading throughputs with 1 client thread for each image. (b) is reading throughput with 2 client threads for each image. (d) is writing throughput with 10 client threads for each image.

Choice Type	Write Throughput	Average Write Latency
`baseline`	401.7MB/s	723.49ms
`latency`	396.1MB/s	765.22ms
`journal`	396.2MB/s	727.46ms

Table 4. The average throughputs and latencies of the writing experiments with 10 threads for each of 8 images of baseline, `latency` policy and `journal` policy.

Choice Type	Read Throughput
`baseline`	7963.1MB/s
`local`	12947.2MB/s

Table 5. FileBench `webserver` reading throughput in production environment.

OSDs and cause errors. Thus, we would like to keep the disk utilization balanced in order to increase the overall capacity, using the `space` policy to choose the lowest utilized disks.

Due to the time taken to fill all disks, we use simulation for the experiment. We simulated 100 disks with capacity of 1 million data blocks each. We simulated the baseline policy and the `space` policy with 2 choices.

In the baseline, the average utilization is only 73%, consistent with the result in our production. With `space` policy, the average utilization increases to 96%. The result shows that we can store 31.5% more data by just having 2 choices.

The reason is clear, with a single uniform random choice, the most utilized disk has the same probability be used as others, and thus more likely to get full. Even with a single extra choice, the client has a chance to avoid the worst case, and thus we can achieve better balance in disk utilization.

Policies that do not work well. We originally expect other three policies, `cpu`, `latency` and `journal` to work as well, following the network power-of-two analysis, but in our experiments we find that it is not the case.

To evaluate the `cpu` policy, we make the CPU utilization 99% at two of the storage servers, and repeat the experiment as the `local` policy. We do not find significant improvement over the baseline as indicated in Figure 2(c). The reason is that Ceph server is not a CPU intensive task, and insensitive to the CPU disturbances. We confirm the fact with a single node Ceph server experiment with lots of CPU disturbances, and see no change on its performance.

More interestingly, although `journal` (choose the smallest uncommitted journal) and `latency` (choose the server with lowest average latency) policies look similar to the well-studied queuing case, as the metric reflects the transient workload, they do not work well. We run a similar set of experiments as before, but we increase the workload by using 10 threads per client, hoping to create more transient queuing effects. Results in Figure 2(d) and Table 4 show no improvement from the baseline.

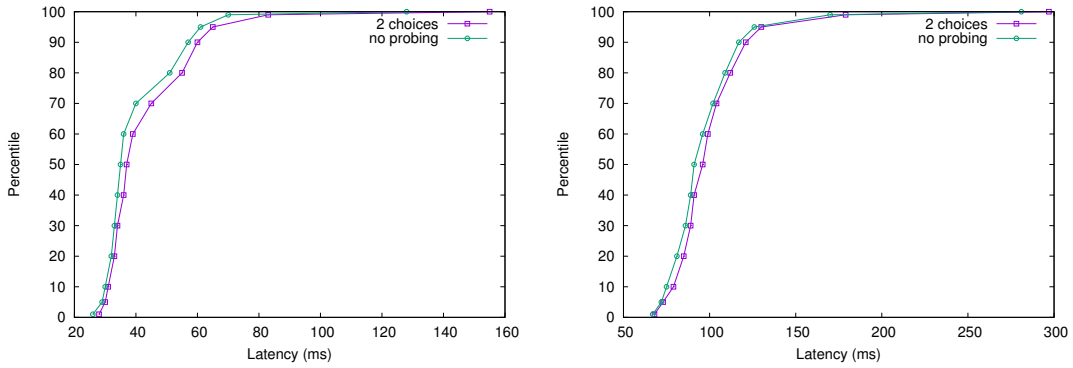
We believe it is because the queue-based transient metrics changes too fast, and it is impossible for us to take a consistent measurement over all the replicas. Plus, as the blocks are large (4MB comparing to the network packets of KBs), when the next block is written to the server, the transient workload might have already changed. Thus, we do not have a better chance of choosing a better one.

Note that even with cases that do not improve performance, CHOICE does no worse than the baseline, showing that it is only adding a negligible overhead to the system.

5.2 Overhead of Server Probing

The most obvious performance overhead is the need of probing multiple servers, and we provide a quantitative evaluation in this section. The most significant performance metric influenced by probing is latency. Thus we performed the following two experiments. First, we run a 4KB-random-write test to see the latency overhead introduced by probing. We randomly choose a 4MB block, and write the first 4KB bytes of the block. We do this repeatedly and we explicitly avoid choosing the same block, causing a probe for every single access. Second, we run a 4MB-sequential-write evaluation to see its overhead on sequential accesses. For each case, we compare our system with probing and the off-the-shelf Ceph without probing. Figure 3(a) and Figure 3(b) show the cumulative distribution of latency, with or without probing. Table 6 shows the average latency numbers in both cases.

We can see that the probing will increase the latency by 2.7 ms, or 6.9% for 4KB random write and 2.7% for 4MB



(a) 4KB random write

(b) 4MB sequential write

Figure 3. Cumulative latency distribution with 2 choices or without probing.

	4K Random Write	4M Sequential Write
no probing	39.19ms	99.69ms
2 choices	41.90ms	102.39ms

Table 6. Average latency for 4KB random write and 4MB sequential write.

sequential write. The increase of latency comes from the extra round trip of probing packets. It is acceptable for small random accesses, and can be ignored for large sequential accesses. Note that our evaluation represents *the worst case* of requiring a probe at every single access. Normally, the probing is only done in the first access at a client, as it can use cached results for subsequent accesses, and thus amortizing the probing cost to multiple accesses. However, if an application keeps writing new small piece of data or reading non-existent data, whose places are not determined so can't be cached, it will suffer such latency overhead constantly.

5.3 Overall Performance in the Production Environment

We migrate the three policies `local`, `memory`, `space` to the production cluster. This Ceph cluster is used primarily as the volume storage backend of Openstack, an open-source platform for cloud computing. We launch two virtual machines on each host of the first rack, and mount two images as volumes on each VM. Then we run FileBench [19] with module `webserver` for each volume, with 10MB average file size, 5000 files, 4MB reading block size. The experiment lasts 1000 seconds. We run one baseline and one configured with `local` policy, and the result is shown in Table 5.

The disks in a rack can provide about $15 * 4 * 100$MB/s = 6GB/s total read throughput. With stock Ceph, when the clients in a rack are running, 1/3 data are written to the local rack. When the network bandwidth limits our reading throughput from other racks to about 5GB/s, we can only utilize half of that on the local disks, which is 2.5GB/s, so the local disks are relatively free when the load is high. Our choice of `local` exploits these free disk throughputs to achieve better performance. And when clients are distributed on all racks to make the speed limited only by disks, our choice does not make things worse because it just prevents the network from becoming the bottleneck, but does not put more pressure on the disks from a global view.

6. Conclusion and Future Work

Consistent hashing greatly reduces the amount of metadata, especially the data block placement information to manage, and thus greatly improves storage system scalability. We enable application-specific data placement flexibility in the hashing framework, using multiple hashing, by increasing the *probability* of finding a good choice. Different applications can customize their choice strategies that fit their performance goals. Although we only provide a small number of choices, and current implementation only allows simple policies, we show the right policy choice can greatly improve the average latency and throughput. Providing just a *higher probability* rather than a *guarantee* of finding a good choice greatly simplifies the implementation.

As future work, we will explore more advanced choice policies based on multiple metrics, and we will provide an application-level API, so the application itself, instead of the storage client embedded in the application, can make the choices. We are also exploring different ways to collaboratively cache the choice information, in order to reduce the number of probes to the servers, further improving performance.

Acknowledgments

We are grateful to the anonymous reviewers for their extensive comments that substantially improved this work. This research is supported in part by the National Natural Science Foundation of China Grants 61361136003, 61379088, China 1000 Talent Plan Grants, Tsinghua Initiative Research Program Grants 20151080475, and a Google Faculty Research Award.

References

[1] M. S. Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 367–381, 2014.

[2] F. M. auf der Heide, C. Scheideler, and V. Stemann. Exploiting storage redundancy to speed up randomized shared memory simulations. *Theoretical Computer Science*, 162(2):245–281, 1996.

[3] J. Axboe. Flexible i/o tester. `https://github.com/axboe/fio`.

[4] D. Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop.apache.org/common/docs/current/hdfs_design.pdf*, 2008.

[5] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve ip lookups. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1454–1463. IEEE, 2001.

[6] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 231–242. ACM, 2013.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[9] G. Developers. The gluster web site. *http:/www. gluster. org*, 2008.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[11] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[12] R. M. Karp, M. Luby, and F. M. auf der Heide. Efficient pram simulation on a distributed memory machine. *Algorithmica*, 16(4-5):517–542, 1996.

[13] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami. Nvmkv: A scalable, lightweight, ftl-aware key-value store. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 207–219, 2015.

[14] M. Mitzenmacher. The power of two choices in randomized load balancing. *Parallel and Distributed Systems, IEEE Transactions on*, 12(10):1094–1104, 2001.

[15] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebooks warm blob storage system. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 383–398. USENIX Association, 2014.

[16] E. B. Nightingale, J. Elson, J. Fan, O. S. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *OSDI*, pages 1–15, 2012.

[17] S. Patil and G. A. Gibson. Scale and concurrency of giga+: File system directories with millions of files. In *FAST*, volume 11, pages 13–13, 2011.

[18] K. Ren, Q. Zheng, S. Patil, and G. Gibson. Indexfs: scaling file system metadata performance with stateless caching and bulk insertion. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 237–248. IEEE, 2014.

[19] F. system and S. L. at Stony Brook University. Filebench. `http://sourceforge.net/projects/filebench/`.

[20] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 309–324. ACM, 2013.

[21] A. Thomson and D. J. Abadi. Calvinfs: consistent wan replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 1–14, 2015.

[22] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 301–316, 2014.

[23] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[24] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 122. ACM, 2006.

[25] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*, pages 35–44. ACM, 2007.

[26] X. Wu, Y. Xu, Z. Shao, and S. Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *USENIX ATC 15*, pages 71–82, 2015.

[27] Y. Xia, S. Chen, and V. Korgaonkar. Load balancing with multiple hash functions in peer-to-peer networks. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 1, pages 10–pp. IEEE, 2006.

[28] L. Xiao, K. Ren, Q. Zheng, and G. A. Gibson. Shardfs vs. indexfs: replication vs. caching strategies for distributed metadata management in cloud storage systems. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 236–249. ACM, 2015.