CloudFS Report

Milestone 1: Hybrid Flash-Cloud System

Goals

In this part, key idea of CloudFS is to implement a File System which combines Cloud and SSD. It ensures that only small files and files that users want access are stored in SSD, while other files(big files) always are stored in Cloud. CloudFS utilizes Fuse and S3 Cloud like apis to achieve its functionality.

Design

In this part, my implementation can be divided into two steps. First step mainly focused on implementing a local file system, and second step implemented cloud storage service.

In first step, I implemented all necessary FS functions mentioned in writeup, like write, open, release,etc. In the part, nothing is special. I only utilized system calls in my functions, in other words, my functions are like wrappers of system calls.

In second step, Cloud service was integrated into system. When a file creates in CloudFS, I know how big ti is .So if the size is greater than threshold, CloudFS will 1) send content of file to cloud. 2) delete file itself 3) create a proxy file with 0 size, but saved original attribute like m_time and st_szie by extended attribute on this proxy file. Then if user wants read or write this big file again, content of file will be retrieved from cloud into corresponding proxy file. Then user can operate this file as usual. Once user finishes operations and release this file, it will be either repush to cloud or truncate to 0 size, depending on if it is modified or not.

Evaluation

In this part, the main focus is correctness. The way to evaluate part 1, is a) test if CloudFS supports UNIX POSIX interface, perfectly. b) test if big file can be put, get from cloud, perfectly.

Since in part 1, correctness is most importance, not only I test CloudFS with scripts, but I also manually test CloudFS in command line. I test commands include mkdir, rm, cp, cat, ls, head, tail, rmdir, etc. CloudFS passes all the tests.

Cost/Performance trade off

Complexity and Simplicity. The first trade off I need to make, is trade off between complexity and simplicity. If I want my CloudFS support more features, utilze tricks to decrease cloud cost, I cannot avoid increasing complexity of software. The obvious drawbacks of complexity, are high running time (e.g. more code to handle more cases) and more space requirement (e.g. increase size of metadata file). More complexity, more bugs, and more time to make software work. So what I have done is keep thing simple, in the meantime, add slightly complexity to reduce necessary cloud cost.

Milestone 2: Deduplication

Goal

Cloud computing is a way to decrease IT cost, if we use it appropriately. Because Cloud service providers usually charge based on storage capacity, operations and data transfer, in order to reduce unnecessary cost, some kind of data duduplication, or compression should be used to reduce data transfer and data capacity.

Design

In CloudFS implementation, Rabin fingerprinting is used to find segment level identify of different files. CloudFS always does rabin fingerprinting on every files whose size are bigger than threshold. It builds index on unique segments and mapping between files and segments. This feature can significantly reduce cost of cloud capacity. It also supports

segment level read and write, that is, only provides minimum data for read and write functions. This feature can reduce cost of data transfer.

Another design is segment reference count. Since multi files may have segments with same MD5 code, one unique copy of segment in cloud can satisfy all files need. However, multi files reference make everything complicated, and raise issues like garbage collection. So final design of my system is, maintain reference count on each segment, and if reference count becomes 0 for one segment, delete this segment.

Key issue here is to decide segment size. I am going to discuss it later.

Usage of Deduplication

Deduplication can be used in any data that need saving in cloud. In my design, files whose size are bigger than threshold will be applied fingerprinting. As for snapshot, they can be applied fingerprinting, as well.

Deduplication is critical in CloudFS, it not only reduces storage capacity cost, but reduces data transfer cost. However, it is possible that deduplication would increase cloud requests, because files are split into small pieces, and s3 uses object model. For example, if user access every file and read their content in CloudFS, all segments would be pulled down, causing lots of requests.I create a new data set, which combine test files with my own files. This set is 6.1MB.

Here is comparison on cloud cost between deduplication and without deduplication:

	Bytes read from cloud	Capacity usage in cloud in byte	Cloud cost	Requests to cloud
No deduplication	6303782	6303786	1.402275674	11
Deduplication	1601574	1601578	3.378323034	305

Table 1: test cost/performance by reading whole files.

Table 1 shows statistics when put files in CloudFS and sequentially read content of files in CloudFS. We can see that deduplication decrease storage capacity and data transfer, but increase cloud requests. Deduplication increases cloud cost in my test.

Table 1 shows an issue in CloudFS. Small, random read and write is very common in File System, bring challenges on file system and hard drives or SSDs. This issue especially becomes critical in CloudFS. If one small read leads to whole file downloaded from cloud, cloud cost will significantly increase. So it is necessary to avoid unnecessary cloud access and data transfer. Segment level IO is a choice. Each time, only get minimum data that can satisfy user's need from cloud. Cost of this feature is, running time grows from O(1) to O(n), where n is number of segments which a file has, since segment level IO request comparison segment by segment.

Here is putting test files into CloudFS and only read last 10 bytes of every file, thing get changed.

	Bytes read from cloud	Cloud Capacity usage(Byte)	Cloud cost	Requests to cloud
No deduplication	6303782	6303786	1.402275674	11
Deduplication	84730	1601578	.365402818	21

Table 2: test cost/performance by reading last 10 bytes of each file.

To conclude, in general, deduplication reduces cloud capacity usage, data transfer while it increases cloud requests. Deduplication may or may not increase cloud cost, depends on read/write type and pattern.

Fingerprinting segment size decision

Segment size decision affects space efficiency deeply. I need to construct complicated index and save metadata of fingerprinting segments. In one hand, If segment size is small, there is no doubt that deduplication works well, while size of metadata becomes big. In the other hand, if size is big, deduplication may lose its power. Consider this case: if we set segment size as 1 byte, deduplication will work extremely well. However, size of metadata will much larger than data itself. Another case gives us opposite trend: set segment size as file size can reduce space cost of metadata, but hurt effect of deduplication.

segment size (KB)	metadata size(Byte)	Cloud Capacity usage(Byte)	Cloud cost
2	87784	1568810	.388290824
4	44553	1601578	.365402818
8	23202	1667114	.378801332
16	12318	1814570	.401914012

Table 3: segment size with metadata size, cloud capacity usage and cloud cost by testing read last 10 bytes of files.

Table 3 shows clearly the trade off between segment size and cloud capacity usage. It is hard to conclude the effect on cloud cost. It looks like cloud cost is necessarily related to segment size.

Evaluation

In this part, the main focus is performance and cloud cost. I measure my system in terms of cloud capacity usage, cloud requests, running time, data transfer, metadata size, etc. I create a test file set that mix given test files and my own files, and use this file set to compare deduplication and no-deduplication under different IO pattern, i.e. randon read/write, sequential read/write, etc.

I discussed and tested random IO, sequential IO, random files, and identical files with deduplication. Evaluation result can be showed above in Table1, 2 and 3. You can see the effort I made to show the influence of deduplication on CloudFS.

Milestone 3: Snapshot

Goal

Snapshot can help CloudFS increase ability on fault-tolerance. From common view, snapshot can tolerant user level errors, like delete files that should not be deleted. What's more, in CloudFS, because cloud storage is available, actually CloudFS can make snapshots and put them in cloud storage, which let CloudFS gain ability to tolerant faults like broken drive.

Design

In CloudFS implementation, it provides make, delete, list and restore operations on snapshots. The trickiest ones are make and restore while delete and list are straightforward. Make asks for two operations, a) tar local metadata and small files, and upload them to cloud, b) increase segment reference count in cloud. Restore asks for a) restoring status at the time when snapshot made, b) delete snapshots made after restored snapshot.

Snapshot design and effort on reducing cost

The most naive way to make snapshot is, every time make a full copy of all bits in file system. However, this way need lots of space. In order to reduce space cost, I come up with two techniques.

First technique is based on cloud storage. Because for big files, they are split into small chunks and saved in cloud. When making snapshots, it is no need to save these chunk again, instead, increase reference count of chunks in cloud is good enough. So when making snapshot, only tar local small files, metadata, and current status of CloudFS and upload this data to cloud. Because of increment of reference count, if a file is deleted after snapshot, this file will still be saved in cloud. When restoring snapshot, I restore status of CloudFS in the time when made snapshot, as well as cloud reference count. This technique is similar to copy-on-write semantics, that is, only alloc minimum space for every snapshot.

Second technique is based on dedeuplication. Because when making snapshot, there are still some data need to transfer to cloud, therefore these data can be applied deduplication.

Evaluation

Case 1: Make snapshot. It turns out, incremental snapshot efficiently decrease extra space requirement.

	Cloud requests	Cloud usage in byte
put test files	307	1601578
make first snapshot	309	1698871
add more files	310	1698875
make second snapshot	312	1798739

Case 2: Restore first snapshot. Restore will triger deleting follow-up snapshots and save space.

	Cloud requests	Cloud usage in byte	Cloud read in byte
before restore	308	1698871	0
after restore	316	1698912	97304

Case 3: delete snapshot. Delete snapshot will reduce cloud capacity usage.

	Cloud requests	Cloud usage in byte
before delete	308	1798739
after delete	316	1698899