CMU 18-746/15-746 **Storage Systems** **Assigned: 24 March 2014**

Spring 2014 **Project 2: CloudFS** **Due: 2 May 2014**

Contents

Overview

In this project, you will build a file system, called CloudFS, to integrate solid-state devices (SSDs) and cloud storage systems (AmazonS3, etc.). To ease development, this file system will be built using the "file system in user-space" (FUSE). CloudFS includes three dimensions: a core file system that leverages the properties of SSDs and cloud storage for making data placement decision, a second dimension that takes advantage of redundancy in datasets to reduce storage capacity and a third dimension that uses local caching to improve performance and reduce (cloud) costs.

The deadline to submit your CloudFS source and project report is **11:59pm EST on May 2, 2014**. To help you make steady progress, we have imposed three **graded** intermediate milestones:

Part1: Hybrid Fuse filesystem	11:59pm EST on Thursday April 3, 2014
Part2: Deduplication and Compression	11:59pm EST on Thursday April 17, 2014
Part3: Caching	11:59pm EST on Thursday May 1, 2014
Project Report	11:59pm EST on Friday May 2, 2014

Note that there is a day between final code handin (Part 3) and the final project report handin. This will leave you a little time to think through and carefully write your project description and performance evaluation report. You must submit your code to Autolab for each milestone; some automated testing will be available through Autolab, but all sections will be evaluated by more than the automated tests.

The rest of this document describes the specification of CloudFS in detail: Section 1 gives you tips about FUSE and VirtualBox setup; Section 2, 3 and 4 present the specification for CloudFS on an SSD/cloud hybrid storage device and service, data reduction via fine grain data deduplication and compression and finally cache management, respectively. Section 5 provides logistics for the code and report handins for this project. Finally, details for the Amazon S3-like cloud service API, Rabin Segmenting APIs and zlib APIs for compression are presented in the Appendix.

Project Environment and Tools

CloudFS will be developed using the file system in user-space (FUSE) toolkit. FUSE provides a framework for implementing a file system at user level. FUSE comes with most recent implementations of Linux and is available in other operating systems as well. FUSE has a small kernel module (FUSE in Figure 1) which plugs into the VFS layer in the kernel as a file system and then communicates with a user-level process that does all the work, using the FUSE library (libfuse in Figure 1) to communicate with the kernel module. IO requests from an application are redirected by the FUSE kernel module to this user level process for execution and the results are returned back to the requesting application. You will implement CloudFS as user-level code that uses libfuse to communicate with test applications (found in the project distribution we provide, or new tests that you write yourself), and uses regular file system calls (through the appropriate mount point) to access the SSD.

By default, FUSE is multi-threaded, so multiple system calls from user applications can be running at the same time in the user-level process, allowing higher parallelism and (probably) faster performance. This requires careful synchronization, and it is **not** required to accomplish this project. We recommend that you

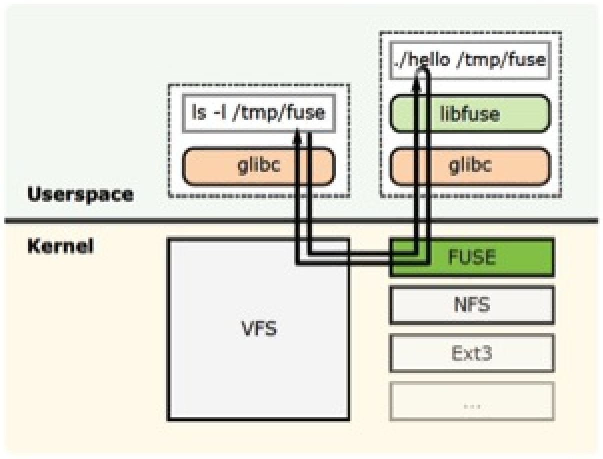

Figure 1: FUSE based filesystem (from http://en.wikipedia.org/wiki/Filesystem_in_Userspace)

use the FUSE option -s which limits the number of threads (concurrent operations from the application through the VFS) to one, to make your debugging simpler.

To enable CloudFS development on your own machine (without a dedicated SSD or external cloud service), you will use a virtual machine environment (for more information, see the appendix). You will use VirtualBox, a free product from Oracle and supported on Windows, Linux and Macintosh machines.

Inside the virtual machine, you will run a Linux OS with two virtual disks: one for the Linux OS (Ubuntu 10.10) and one for your CloudFS SSD. Both virtual disk images (.vdi) will be available on the project website. It is inside this virtual machine that you will run your CloudFS user-level hybrid file system that will make file system calls on the SSD virtual disk (not the OS virtual disk) and a local Cloud storage service.

Note that virtual machine images are 5-10 GB in size, and you will need at least this much free space on the machine you use to run Virtual Box. We have given you two compressed disk images in the project distribution (along with the README with instructions). Please make sure you can run Virtual Box with the root OS partition as soon as possible. If you have problems with this early, we may be able to help. If you have problems with this at the end of the semester, you may be in big trouble.

We will be giving you a few test scripts that you run at user level in the virtual machine with a pathname that resolves in the FUSE filesystem. These scripts will typically extract a TAR file containing sample files and directories into CloudFS, and then do something with the resulting file system. They will detect the effect of CloudFS on storage components using the command vmstat and cloud storage service's statistics report. These statistics are useful for seeing the effect of splitting the CloudFS file system between the two components, of deduplication, and of caching. We will also give you a script to summarize the SSD and cloud storage statistics output. We will also provide a object storage server binary which implements the Amazon S3 interface. You will run this in your virtual machine, so http accesss to this web server use the loopback network stack connecting back into the same machine. The disk storage for the cloud will actually be in the Linux OS virtual disk, although it could in principle be in the cloud.

Few thoughts about the design of your project:

- We suggest an approach to the design of CloudFS that simplifies the overall system design. Think of the various parts (fuse-ssd, deduplication, cache management) as layers of function stacked on top of the cloud storage. Designing clear and pre-documented interfaces between layers should simplify the design and save you lots of wasted development effort.
- DO NOT rely ONLY on in-memory data-structures; this would be an unrealistic design for most real-world file systems because you lose the contents of memory on crash-n-reboot and because you may not have enough memory to hold all of the file system metadata. Although the test cases for this project are not too big for an in-memory structure of all metadata, we will not accept in-memory only as a correct solution and you will get a poor grade for using such an approach.
- You should design out-of-core data structures that are persistent through crash-n-reboot. Your CloudFS can use storage space on the SSD to store any state associated with your approach (including special files that are known only to CloudFS), but certainly not the data of all large files.

Part 1 : Hybrid file system spanning SSD and cloud storage

The first part of this project is to implement a hybrid file system with two different storage components: a SSD and a cloud storage service such as Amazon S3 (or Windows Azure). Cloud storage services provide a large storage capacity with “pay as you go” costs, dynamic scalability and high availability, as long as you are connected to the network.

In this part of the project, you extend your limited personal storage (the SSD) with cloud storage. Compared to cloud storage, SSDs, particularly NAND flash devices, are (1) very much faster, (2) wear out if written too often, (3) have limited capacity and (4) induce no operation charge (billing) for actions or data transfer. Users want the best of both worlds: the extensibility of cloud storage, the small random access speed of SSDs, no per-operation cost of SSDs and storage that does not wear out, all at a minimal cost. In this project, we assume that the SSD storage controller (and its flash translation layer) completely handles wear-leveling to improve the lifespan of SSDs (so you dont need to worry about wear out).

The goal of the first part of this project is to build a basic hybrid file system, called CloudFS, that realizes the properties described above for a system that uses both an SSD and cloud storage for storage. The basic idea is to put all the small objects on the SSD and all the big data objects on the cloud storage. A real implementation strategy would be to modify the local Linux Ext (2, 3 or 4) file system to be mounted on a device driver that offers an SSD. This modified Ext file system would manage lists of free blocks on SSD, allocating an object on the appropriate storage component, and have pointers pointing between the two as appropriate. Since building an in-kernel file system is complex and beyond the scope of a course project, you should use the more layered approach we recommend.

In our project, the SSD device will have a dedicated local file system (e.g. ext2) mounted on it (and you will not modify the code for this local file system). Cloud storage can be accessed via the Amazon S3 object store interface. You will write the higher level interposition layer, using the FUSE API, that plugs in as a file system, but rather than use a raw device for its storage, it uses the one local file system for SSD storage and the Amazon S3 interface for cloud storage.

The cloud storage has a different interface from the traditional file system API. To understand a cloud storage service, let's take Amazon S3 as an example. Amazon S3 provides a simple web interface that can be used to store and retrieve arbitrary objects (files). Objects are organized into buckets as a flat namespace. Each bucket is essentially a directory that stores a number of objects, which are identified within each bucket by a unique, user-assigned key. Buckets names and keys are to be chosen so that objects are addressable via an HTTP URL in the form: http://s3_sever_hostname/bucket/key. The namespace is not hierarchical, and only has one level of directory. Amazon S3 uses a simple APIs such as LIST, PUT, GET and DELETE to access objects. The LIST operation retrieves all the bucket names in a S3 server, or the names (key) and attributes of all objects in a bucket. The PUT operation puts an entire object into the S3 server. The GET operation reads the whole object. The DELETE operation can delete an object or a bucket.

Unlike SSD, cloud storage costs money each month. Cloud storage services have different pricing models. For instance, Amazon S3 charges for the capacity of disk space you consume, the amount of data transferred, and each retrieval/command you submit. Dropbox has a different model. Dropbox charges only for the capacity of disk space. For this project, we choose an Amazon style cost model, as shown in Table 1. One of your design goals is to minimize the cost incurred for as wide a range of workloads as possible, especially the ones in our tests.

Type	Price used in our tests	Real price of Amazon S3
Capacity	$ 0.095 per MB (max usage during one test)	$ 0.085 per GB per month for the first 1 TB
Operation pricing	$ 0.01 per request	PUT, COPY, POST, LIST: $ 0.005 per 1,000 requests. GET: $ 0.004 per 10,000 requests.
Data transfer pricing	$ 0.12 per MB (out from S3 only)	$ 0.12 per GB for up to 10 TB (out from S3 only).

Table 1: Cost model for Cloud Storage API (For more real price information: refer to "http://aws.amazon.com/s3/pricing")

Design Specifications

Your CloudFS will provide two primary features:

i) size-based placement to leverage the high IO per second (IOPS) provided by SSDs and

ii) attribute replication to avoid performing small IOs on Cloud storage.

Size-based data placement

CloudFS data placement - small objects on a SSD and large objects on a cloud storage - is implemented through redirection when a file is created and is written to. In CloudFS, the file system namespace is created on the SSD, small files are written to the SSD and big files (files that grow larger than a threshold) are moved to the cloud storage. This migration replaces a small file, which was previously stored on the SSD, with a user defined link pointing to the location of the big file in the cloud storage. When opening such file, CloudFS parses the path through the user-defined link, creates a temporary file (on SSD) and copies an entire cloud object into the temporary file on SSD. Finally CloudFS returns the file descriptor of this temporary file as the return value of open system call. When a big file is closed, you shall flush the temporary file into cloud storage if the file is dirty (changed), or delete it if the big file is clean (unchanged). For design simplicity, you can assume that the SSD always has enough space for keeping copies of all currently open files.

In part 2 you may need to re-design the handling of open files, because in part 2 you cannot assume that all open files will fit in the SSD.

Because CloudFS is a FUSE-based file system, most of the code will be implementations of functions called through the VFS interface. In fact, for this project, you don't need to support all the VFS functions; you can build a working prototype using a subset of VFS calls including getattr, getxattr, setxattr, mkdir, mknod, open, read, write, release, opendir, readdir, init, destroy, access, utimens, chmod, unlink, rmdir (and maybe truncate). Your implementation of CloudFS will have to make various design decisions including deciding whether and when a file is placed on the SSD or the cloud storage, detecting when a file gets big, copying it to the cloud storage and updating the file system namespace on the SSD using a user-defined link. Your CloudFS file system should run from the command line as follow:

```
./CloudFS --hostname Hostname --threshold MigrateThreshold
  --ssd-path SSDMount --fuse-path FUSEMount --ssd-size SSD_size
 [--more_args]
```

where `MigrateThreshold` is the maximum size of a file that should be stored (permanently) in the SSD (specified in KB, defaulting to 64 KB), `SSDMount` is the mount point for the SSD device in the virtual machine (defaulting to `/mnt/ssd`) and `FUSEMount` is the mount point for CloudFS (defaulting to `/mnt/fuse`). Hostname is hostname of the web server that runs the Amazon S3 simulation (it should probably always be set to `localhost:8888`). `SSD_size` specifies the capacity of SSD (specified in KB, not counting open temporary files). You can assume that there is no capacity limit on the cloud storage.

A key goal is to migrate a file from the SSD to the cloud storage based on its size. While it is possible to implement the correct behavior of a single operation in a FUSE file system by opening the path, seeking, performing the operation, and closing the file on every operation, it is very inefficient. A better way is to open the path on the FUSE `open()` call, save the file descriptor, and re-use that file descriptor on subsequent `read()` or `write()` operations. FUSE provides a mechanism to make this easier: the `open()` call receives a `struct fuse_file_info` pointer as an argument. CloudFS may set a value in the `fh` field of `struct fuse_file_info` during `open()` and that value will be available from all future `read()`/`write()` calls on the open file. On `close()`, you make the decision to migrate the file to cloud depending on the file size. If the file does need to be stored in the Cloud, then it has to be done in a manner transparent to users of the filesystem. This means you have to store all the attributes of the file as well as the mapping to an object name in the cloud somewhere on the SSD.

Saving the attributes of migrated files

Based on the CloudFS description so far, the attributes (such as size, timestamps and permissions) of a big file need to be stored with a stub or proxy for the file on the SSD. You don't need to store attributes of big files in the cloud as well. CloudFS needs to work hard to make sure these small accesses go to the SSD, not to the cloud. In particular, `ls -lR` reports attributes of all files, a small amount of information per file, and it does not read the data of any file, so we would like it to not to incur the cost and latency of cloud storage access.

The metadata about files that have been migrated to the cloud can be managed on SSD in a couple of ways. One of them is to have a proxy file on the SSD to represent the file in cloud. Reporting the attributes of this proxy file is **not** the correct information. So if a user makes a `stat()` call to get the attributes of a file in the cloud, CloudFS should look into the proxy file, somehow, and fetch the cloud file's information from the proxy file's user defined meta structure. There are many other ways that you could store the attributes of the big files in the SSD — in a stand-alone database, as a special directory representation with extended attributes, or as data in an hidden file (such as resource forks in OS X). Whichever technique you use, it is important to tolerate machine failures and CloudFS process crashes, so that the file system is in a consistent state after rebooting and restarting. If you decide to keep meta information in extended attributes of a proxy file, you could invent user-defined attribute names, beginning with the string "user", for example, one for every attribute of the migrated file and then make sure they are updated on non-volatile storage in the future as well.

Mapping files to objects in the cloud

The cloud storage is a specific key-value store. So in order to put a file in the cloud, you have to first come up with a key (a name in the cloud) for the file you want to upload. You also need to be able to regenerate or remember the key in order to read the file back from the cloud into SSD. One way to do this is to use the original path name of the file as a name in the cloud storage so that you can infer location of the file in the cloud storage from the original path name correctly. This might be desirable because it does not incur extra storage space for a mapping table and you dont need to take special care of consistency. You might need to replace "/" in the original path name with other characters eligible for Amazon S3 naming rule such as "+". Another way is to use an unique value, such as a creation sequence number or creation timestamp as the object name. Other approach is to use the MD5 hash value of the data contents. You may need to revisit this decision in part 2 though. Refer to the appendix for more information on the usage of the cloud API.

Test and Evaluation

To get you started with FUSE, the project distribution includes skeleton code for CloudFS. These source files compile without errors and include comments with a few pointers about how to use the FUSE library to build a file system. You are expected to build your code using this skeleton code and include a Makefile to compile it. Source code documentation will be a part of the grading; please write useful and readable documentation (without generating an excess amount of it).

The project distribution also includes scripts to facilitate code testing. Note that these scripts will help you with correctness and performance checks, but you may have to dig deeper to debug performance issues (e.g. you should develop your own tests). And your projects will be graded using a different set of scripts and data-sets.

The initial test script for Part 1 is called `test_part1.sh`. This script performs three operations:

1) extract a TAR file into the CloudFS mount point,
2) read all the files in the directory to compute md5sum for each file and
3) read attributes of all objects in the system using `ls -laR`.

We also provide three TAR files of different sizes to help you test your CloudFS. Each of these three operations is wrapped around two measurement related actions. The script will unmount and then re-mount the CloudFS filesystem to eliminate the benefits of the OS buffer cache from the performance numbers. Second, each operation will generate statistics of block IO and cloud storage access that happened during each operation. We use `vmstat -d` for the SSD and cloud S3 stats (number of operations, bytes transferred, maximum capacity usage) before and after each of the three operations and then parse the output using a helper program. Read the man page for vmstat to understand its output format. The README file in the scripts directory has details about using this script.

In addition to performance, there are three correctness requirements for Part 1:

1) The files in CloudFS should be identical to the files in the tarball,
2) no closed file larger than the threshold is stored in SSD and
3) reading metadata only (e.g. `ls -laR`) should not cause any cloud storage access.

These correctness requirements lead to a large portion of the grade in Part 1. Extra score will be given according to how efficiently you keep the meta data in SSD.

Part 2 : Data Reduction

In Part 1, you were able to transparently increase the size of the file system by backing it up with infinite storage provided by the cloud. Every file that was stored in the cloud increased the cost of storage. If you have a lot of files that have the same or nearly the same content, you pay the price of storing the duplicate content again and again. Can we devise a way to reduce these duplicate storage costs? Duplication can exist within a file as well as across many files. Two common techniques to reduce duplication as well as achieve efficient data reduction are:

a) Segment level Deduplication
b) Data Compression

Part 2a : Segment Level Deduplication

Deduplication is a term used when a storage system tries to discover duplication among unrelated files and store identical content only once.

The simplest way to think about deduplication is to compute a checksum or hash of the entire file, or perhaps each file block, storing the files or blocks hash in a large lookup table. When a new file (or block) is "written" to the storage system, compute its hash and look up this hash in the lookup table. If the lookup table does not contain the hash, then this data is new, because identical content ensures identical hashes, so store the new data, insert the new hash and the location of the new data into the lookup table and set the file (or block) metadata to point to the newly stored data. If the lookup table does contain the hash, then the data *might* already be stored in the storage system. Some deduplication systems do a bit by bit comparison of the new data and the stored data with the same hash, but others select hashes with a lot of bits and strong randomness properties (MD5 for example) so that they can assert that the chance of an accidental collision of two different data objects with the same hash is much much lower than the chance of the data being returned by storage devices incorrectly (1 mis-read bit in 10ˆ21 bits read on many hard disks, for example) – these systems do not verify a hash collision, but instead assume identical hashes mean identical data without checking.

If the hashes and data match, the lookup table also provides the location of the stored copy of the data, so you can drop the new copy and use the stored copys location in the file (or block) metadata. Storage used in the cloud will now be less than if we had not computed and checked hashes.

The "unit" or granulatity of deduplication

Using MD5 it is pretty easy to notice identical files. For example:

```
$ ls -l bigfile
-rw-r--r-- 1 palampal palampal 20480 Mar 6 05:03 bigfile

$ cp bigfile copy-of-bigfile

$ md5sum bigfile copy-of-bigfile
7fbc9616d4d275d05629e5cf9415495e bigfile
7fbc9616d4d275d05629e5cf9415495e copy-of-bigfile
```

You could build a deduplicating cloud storage system based on this style (whole file) of defining the "object" to be deduplicated. It is the easiest to implement, but if the file is an email message with an embedded 10 MB powerpoint presentation, and the 10 MB powerpoint presentation is already in a file in the cloud, then the mail message including the incoming 10 MB powerpoint presentation is not identical to the 10 MB powerpoint file in the cloud, and we end up with 20 MB in the cloud. And if the presentation was broadcast to everyone in the department, 1 GB of the same 10 MB powerpoint presentation is not unlikely.

But whole file deduplication works only for identical files that are bitwise equivalent. Consider the following not unusual file editing scenarios. Often we create files by gradually appending data to the file over time. We also make copies of files and edit them. A single bit change (bit flip / addition / deletion) in the file will cause the MD5 hash values to differ, thereby forcing us to store two copies of nearly identical content.

```
$ ls -l smallfile
-rw-r--r-- 1 palampal palampal     1 Mar  6 05:06 smallfile

$ cat ./bigfile ./smallfile > ./bigfile-smallfile
$ md5sum ./bigfile ./bigfile-smallfile
7fbc9616d4d275d05629e5cf9415495e  bigfile
3367778e0f263a7879daf956439439d8  bigfile-smallfile
```

One solution to the above problem is to split the file into smaller blocks of fixed size and compute the hash for every block. Now the granularity of detecting duplicated content comes down to the block-size, typically something like 4KB. Note that most of the content in the two files above are identical, except for the last byte that was appended. We can apply the same dedup (deduplication is often nicknamed dedup) technique as before, but at the block level.

In the below example, we split the files into chunks of 4KB each. The first five chunks from each file will be the same and have the same MD5 hash. So we need to only store the sixth chunk of the second file. All other chunks can be deduped, thus saving us the cost of storage.

```
$ split -b 4096 -d ./bigfile bigfile

$ md5sum ./bigfile0*
463829a1a37bb5fbd36197d1b4b459bc  bigfile00
98e8b11cc2c8702066d2323d0ad5c3fa  bigfile01
ccaad733a231d62c5fdd777d808f15b2  bigfile02
130702081e45ffd7facaab4a52da91f2  bigfile03
e13dd6730e63115d52afe056939c5573  bigfile04

$ cat ./bigfile ./smallfile > ./bigfile-smallfile

$ split -b 4096 -d ./bigfile-smallfile  bigfile-smallfile

$ md5sum bigfile-smallfile0*
463829a1a37bb5fbd36197d1b4b459bc  bigfile-smallfile00
98e8b11cc2c8702066d2323d0ad5c3fa  bigfile-smallfile01
ccaad733a231d62c5fdd777d808f15b2  bigfile-smallfile02
130702081e45ffd7facaab4a52da91f2  bigfile-smallfile03
e13dd6730e63115d52afe056939c5573  bigfile-smallfile04
9fe0f7244a7da1d3f5b3d21f9b1e1ea8  bigfile-smallfile05
```

Block level deduplication has solved our problem in the above case, but this (naive) style of block level deduplication (i.e dividing the file into fixed size 4KB blocks) does not work as well if content is duplicated but misaligned.

Lets revisit the above bigfile example. Instead of appending data to the end of bigfile, we will prepend (i.e. add at the beginning) a little data to bigfile and see how the fixed block size solution works.

```
$ cat ./smallfile ./bigfile  > ./smallfile-bigfile

$ split -b 4096 -d ./smallfile-bigfile  smallfile-bigfile

$ md5sum ./bigfile0*
```

```
463829a1a37bb5fbd36197d1b4b459bc  bigfile00
98e8b11cc2c8702066d2323d0ad5c3fa  bigfile01
ccaad733a231d62c5fdd777d808f15b2  bigfile02
130702081e45ffd7facaab4a52da91f2  bigfile03
e13dd6730e63115d52afe056939c5573  bigfile04

$ md5sum smallfile-bigfile0*
f7290d75e8f81c8acd21ee350f759bfe  smallfile-bigfile00
353d2c12147583f287de0576768f957f  smallfile-bigfile01
0c8d327ab961c780d74b2bbd9d880b07  smallfile-bigfile02
fc69cbbec0683ed5250cfe59091a0d5e  smallfile-bigfile03
dbd09bbbbf0fa8657fcb3c01f7203ed6  smallfile-bigfile04
15f41a2e96bae341dde485bb0e78f485  smallfile-bigfile05
```

You can clearly see in the above output that all the MD5 hashes are now different, even though we prepended just one byte of data to the beginning of the bigfile.

The problem is that arbitrarily splitting a file into fixed size chunks is very easily misaligned. What we need is an algorithm that divides up data sequences into subsequences based on the content itself, so that files with subsequences that are the same are likely to be split so the parts that are the same becomes separate identical chunks.

We are *NOT* asking you to invent such a content specific splitting algorithm. That invention, **Rabin Fingerprinting**, was a big deal. The key idea is that a specific pattern/value of a short hash of a rolling window on the file can be declared to be the start of a content subsequence, called a segment, so any content that is identical through at least two of these "start subsequence" hash values will decompose into segments where at least the middle segments are the same and not misaligned.

In this project we will not require you to understand Rabin Fingerprinting at a deep level. We will provide you with a library for that. However, for more historical and advanced reading, please see the following papers:

Udi Manber, "*Finding Similar Files in a Large File System*", USENIX Winter 1994 Technical Conference Proceedings, Jan. 17-21, 1994, San Francisco, CA.

Benjamin Zhu, Kai Li, Hugo Patterson, "*Avoiding the Disk Bottleneck in the Data Domain Deduplication File System*", 6th USENIX Conference on File and Storage Technologies, Feb 26-29, 2008, San Jose, CA

An important consequence of using a splitting scheme like Rabin Fingerprinting is that segments are *NOT* all the same size. Each segment might be a very different size, in fact. An important parameter to Rabin Fingerprinting is the average segment size, which can be tuned to different values. This average segment size is the granularity for this scheme. For practical reasons it is also likely that segments will be required to be between a minimum and a maximum size.

One of the fundamental choices when designing a deduplication system is the granularity over which you wish to detect duplicate content. Larger granularity requires less metadata (pointers in the files and entries in the lookup table) and less frequent lookups, but larger granularity also leads to more identical data that ends up combined with non-identical data in the same segment, or less space savings.

Design Specification

The goal here is to implement variable size segment deduplication for the data content stored in CloudFS. To simplify the implementation, we ask you to apply deduplication only to files that are to be stored in the cloud (i.e files bigger than a given threshold, not all files). So instead of automatically shipping the large file to the cloud (as in Part 1), you will hand it over to a dedup code layer, which should then do its magic and only store unique segments in the cloud, thereby reducing the data capacity consumption on cloud storage and the corresponding data transfer costs. Your design and implementation of a deduplicating system will be measured against lowering these cost metrics.

You should be able to enable/disable the de-duplication for one run of your operation through an argument on this invocation command line. Deduplication should be enabled by default. Specifying '`--no-dedup`' should run the system without deduplication. This will aid you in debugging and us in testing your program. A layered design with clearly defined interfaces will help you here and throughout the project. You can assume that we will not be switching dedup modes during the running of your file system.

```
./CloudFS [--no-dedup] [--other-args]
```

There are various approaches to designing the dedup subsystem, but all designs should be able to cope with addition and deletion of a few bytes in the files without losing (all of) the cost savings benefit. The implementation will be measured against the total cost savings in the cloud.

Identifying the segment boundaries

We will be providing you with an implementation of the Rabin Fingerprinting algorithm along with the code to partition the file stream into segments. You are free to use this or write you own (we do not intend to give bonus points for doing your own Rabin Fingerprinting implementation). A sample program that prints out the MD5 checksums of segments in a given file will also be included along with the support code.

To understand the Rabin Fingerprinting algorithm first look at the way it uses a rolling hash algorithm, hashing a window of data in the file at every byte offset. A fast implementation incrementally calculates the hash values over a given window of bytes (say 'w' bytes). We start at the first byte and slide the window forward, one byte at a time, while calculating the rabin fingerprint at every byte offset in the file. To determine a segment boundary, we look for a specific bit pattern in the generated rabin fingerprints. For example we can mark a segment boundary every time all 'b' least significant bits (LSB) of the rabin fingerprint are equal to zero, so that 2^b bytes will be the average segment size. To guard against a string of nulls in the data or a huge segment, there will also be a minimum segment size and for ease of implementation, a maximum segment size.

Your CloudFS implementation should support passing in the average segment size and window size for the rabin fingerprinting algorithm to override the defaults.

```
./CloudFS [--avg-seg-size <bytes>] [--rabin-window-size <number>] [--other-args]
```

Identifying the duplicated segments

Consider a file system having 40 GB of data with an average segment size of 4 KB. In this case you will end up storing and searching through 10 million hash values. And 40 GB, to be fair, is not a lot of data for a file system that is backed by the cloud. So you need a good algorithm to search through the hash values efficiently. The choices range from a simple hash table implementation to databases with bloom filters or B+ trees.

Another problem is that you may (in fact, should) not be able to fit everything in this lookup table into main memory. In the real world this data structure needs to be mostly on storage, with only as much in memory as it takes to go fast. The cost of reconstructing this data structure at startup can also be prohibitive, so many implementations of this data structure have to be persistently updated on disk.

To simplify the design and implementation, we allow you to assume that all the hash values of all segments in the filesystem fit into main memory. In this case a simple hash table should be sufficient, if it is persistently backed on SSD. We will also allow you to recreate the data structure by doing a LIST on the segments stored in the cloud, provided you have a good naming scheme for the segments.

We are not looking for a fancy data structure; make it fast, persistent, small and above all simple. Note that the hash values that are being stored and searched in this lookup table are MD5 hashes of the content in the segments; these hashes have nothing to do with the hash that Rabin Fingerprinting algorithm generates internally to select the segment boundaries.

Because we allow C++ solutions to use the STL library hash table function, if you choose to use C instead you may ask use to approve your use of a pre-existing hash table package. You must name exactly what code you are using in email, you must have our written approval (not verbal) and your design document must name and describe any pre-existing code used that we did not give you. Except for this hash function, we do not plan to approve use of existing code.

Mapping files to segments

Last piece of the dedup puzzle is to tie down the newly created segments to the file they belong to. Segment naming and management is internal to your file system and are not exposed to the user. You have to build and maintain a mapping of CloudFS "big file" name to a list of segments. You can come up with various ways of achieving this mapping. You could create a on-SSD data structure that maps a key (filename) to values (all the segments that belong to the file). One of the simplest approaches is to use the hidden proxy file that you may have used in Part 1 to store the attributes of the file in cloud. You could store the segment identifiers in this file. Do note that you need to also maintain the correct ordering of the segments.

Note also that if the file being accessed is bigger than the free space on the SSD, it is still possible to access just one or two segments at a time. This is particularly challenging when a file is freshly created and being written the first time. YOU MAY ASSUME THAT ALL FILES ARE WRITTEN SEQUENTIALLY STARTING AT THE FIRST BYTE (because it is much harder to dynamically segment a new file that is being written in a random order).

Deleting segments on file deletion

You must be able to recover the cloud storage space used by a segment whenever all files to which the segment belongs gets deleted. To achieve this, you may come up with a reference counting scheme for the segments stored in the cloud. You may choose to store the reference counts as part of the lookup table that is used to search for duplicated segments.

Part 2b : Data compression

Compression is the process of reducing the size of data by using algorithms to find short redundant bit sequences within data or by using an encoding technique that can discover predictabilities or differential frequencies that can be exploited to represent the original data using fewer bits. Two general classes of compression algorithms exist, lossy and lossless. A lossless data compression algorithm represents data without loosing information, whereas lossy data compression algorithm is the opposite, where some loss of information is acceptable, for example in pictures (JPEG) or movies (MPEG). You will be using lossless algorithm in this project.

Design Specification

The goal here is to leverage benefits of compression for further data reduction and hence reduce the cost of storing data on the cloud. All data stored on SSD can be compressed as well. The computation costs must always be considered while using compression. A lossless compression library ”zlib” has been provided along with the support code. The zlib compression library provides in-memory compression and decompression functions, including integrity checks of the uncompressed data. Its APIs and example usage are explained in the next section. Design and implement well defined interfaces using this library so that it can interact well with other components of the system such as fuse-ssd, deduplication, cache and cloud as appropriate.

You should be able to enable/disable compression for one run of your operation through an argument on this invocation command line. Compression should be enabled by default. Specifying ‘`--no-compress`’ should run the system without compression. This will aid you in debugging and us in testing your program. A layered design with clearly defined interfaces will help you here and throughout the project. You can assume that we will not be switching compression modes during the running of your file system.

```
./CloudFS [--no-compress] [--other-args]
```

zlib Compression library

zlib is a free lossless data-compression library for use on virtually any computer hardware and operating system. The zlib data format is itself portable across platforms. It’s compression method, an LZ77 variant called deflation, emits compressed data as a sequence of blocks. For advanced reading, detailed explaination of its deflation and inflation process can be found at

- zlib’s deflate algorithm: http://www.zlib.net/feldspar.html

- zlib technical details: http://www.zlib.net/zlib_tech.html
- zlibs manual: http://www.zlib.net/manual.html

The zlib APIs for compression/deflate and decompression/inflate are given below.

zlib compress/deflate api :-

```
source - input file pointer
dest  - destination file pointer
len - length of the file to be compressed
level  - level of compression
return  - the length of the compressed segment

int def(FILE *source, FILE *dest, int len, int level);
```

zlib inflate/decompress api :-

```
source  - input file pointer
dest  - destination file pointer
return  - 0 if success, negative otherwise

int inf(FILE *source, FILE *dest);
```

zlib example usage

Following is an example of how this library can be used. Appendix section presents a more detailed example.

File before compression

```
mukuls@MUKUL:~/TA/cloudfs/src/cloudfs$ ls -l compress_example.c
-rw-r--r-- 1 mukuls mukuls 2408 Mar 13 18:37 compress_example.c
```

Compiling and linking with zlib library.

```
mukuls@MUKUL:~/TA/cloudfs/src/cloudfs$ gcc compress_example.c compressapi.c
../lib/libz.a -o compress_example
```

Compressing "compress_example.c"

```
mukuls@MUKUL:~/TA/cloudfs/src/cloudfs$ ./compress_example compress_example.c
compressed_file
compressing
```

Verify compression : reduced file size

```
mukuls@MUKUL:~/TA/cloudfs/src/cloudfs$ ls -l compressed_file
-rw-r--r-- 1 mukuls mukuls 1034 Mar 13 18:37 compressed_file
```

Decompressing

```
mukuls@MUKUL:~/TA/cloudfs/src/cloudfs$ ./compress_example -d compressed_file
decompressed_file
decompressing
```

Stats of the decompressed file

```
mukuls@MUKUL:~/TA/cloudfs/src/cloudfs$ ls -l decompressed_file
-rw-r--r-- 1 mukuls mukuls 2408 Mar 13 18:38 decompressed_file
```

Diff original file and decompressed file to verify integrity

```
mukuls@MUKUL:~/TA/cloudfs/src/cloudfs$ diff decompressed_file compress_example.c
```

You may also find following resources from zlib website useful

- http://www.zlib.net/zlib_how.html

Mixing Deduplication and Compression

An interesting design decision that you have to make and justify in the report is how you use both deduplication and compression to achieve efficient data reduction. For e.g. using deduplication followed by compression or compression followed by deduplication or any other policy. You might find the following IBM Research paper useful for this purpose:

```
Constantinescu, C., J. Glider, D. Chambliss, Mixing Deduplication and Compression
on Active Data Sets, IEEE Data Compression Conference (DCC), 2011, DOI: 10.1109/DCC
```

Again, a clean and layered design of interfaces for both deduplication as well as compression shall help you to test your theory as well as implement efficient data reduction.

Test and Evaluation

Your Part 2 program will be tested for the amount of capacity and cost savings that you can bring about by implementing deduplication and compression. While correctness, performance and cost are all important, correctness is the most important.

The test script for Part 2 generates a large number of big files. The files will sometimes share data chunks with other files. At the end of test, capacity consumption of cloud storage is measured as well computation cost shall be measured. This computation cost will include computation for compression and decompression. As in Part 1 tests, the Part 2 test is also wrapped around two measurements (vmstat and cloud stats). Capacity savings by deduplication and compression with minimal computation cost is most important for grading Part 2, but a part of the grade will depend on how well you manage segment metadata on SSD and minimize cloud operation costs.

Part 3 : Segment-level Caching on SSD

In this part, you will implement caching of CloudFS segments in the SSD. These segments are the ones identified by rabin-fingerprinting. As mentioned in Part 1, cloud storage charges for operations and it has a very much longer average access latency compared to SSD. Also, the entire file may not always be needed, because only a few segments may be used by the file access operations. Note that this opens up the opportunity for files to be larger than the (available space or even the entire) SSD.

Your goal for caching is to improve performance and minimize cloud operation costs. This is possible by caching such segments on the SSD itself, so that the cost of accessing such segments from cloud is reduced. This can be quite complicated because segments are variable sized. Thus moving medium sized segment out of the cloud might cost more, per byte moved, than a larger segment, because of the fixed per-operation cost.

The best policy is workload dependent, so we will be interested in your reasons for selecting a specific policy, and your explanation for the tradeoffs.

Design Specification

The direct goal of this part is to build a segment-based cache layer between the data reduction layer in part 2 and the S3 Cloud. This layer caches one or many segments of or many files in a hidden directory, for example "`.cache`" on SSD.

You should be able to enable/disable the cache for one run of your operation through an argument on this invocation command line. Caching should be enabled by default. Specifying '`--no-cache`' should run the system without caching. This will aid you in debugging and us in testing your program. You can assume that we will not be switching modes during the running of filesystem.

The size limit of the hidden cache directory is given as an input parameter. You should make sure that the size of hidden directory for caching does not exceed the given cache size.

```
./CloudFS [--no-cache] [--cache-size cache_size] [--other-args]
```

Cache replacement policy

In general, a cache is never big enough for all segments in the cloud. And until the cache is (near) full it is reasonable (and probably cost-effective) not to migrate any segments out of the SSD and to the cloud.

What segments should be moved to the cloud when you must move something, and when? This is a cache replacement policy. You can imagine that frequently accessed segments should reside in local storage, to prevent unnecessary transfers from the cloud, to keep medium sized segment access response times small as well as limiting data transferring costs. Note that moving many, small, rarely read segments to the cloud may cost more than moving a slightly more recently used large segment. Keeping contiguous segments in cache for better locality is another possibility. You are encouraged to design more sophisticated replacement algorithms to reduce response time and cloud storage costs.

Unlike Part 1, open files need not be entirely in SSD, and can be considered part of the cache.

Persistency of cache contents

The segment cache in a SSD is persistent storage space. Therefore, you can reuse the cache contents and get benefits from caching even across multiple remounts. To make this work, you must be able to recognize each segment in the cache without having a record in memory of putting it there. This should sound a bit like fsck. At mount time, CloudFS can scan its cache to figure out what is contained there and build an in-memory lookup table describing the cache contents. Figuring out what is contained in the cache will require you to define and manage on-SSD metadata for the cache. This might be in the construction of a name, or attributes, or even in the links in the small stub files in the SSD namespace. Also, this metadata should be consistent with other metadata you maintain for segments in Part2 as mapping of files to these segments is necessary.

For simplicity of crash recovery cases, you can assume that the cloud storage service never crashes and never loses, damages or adds to things you have given it.

Evaluation and Testing

Your Part 3 CloudFS program will be tested for the amount of performance and cost savings that you can bring about by implementing caching.

Part 3 test consists of two sub tests: a stress test and a cache test. In the stress test, the script will generate random sized files which may share chunks of data, and verify correctness. In the cache test, the test will run a set of workloads and performance metrics are measured. In Part 3, the cost savings provided by the cache is the most important part in grading, after correctness.

Project Logistics

Resources

The following resources are available in the project distribution on the course website.

- CloudFS skeleton code:

  The files inside `src/cloudfs/` are the skeleton code that you will modify and extend for your project. "`cloudfs.h`" and "`cloudfs.c`' contain the skeleton code for FUSE file system. "`cloudapi.h`" and "`cloudapi.c`" contain the wrapper functions of libs3 C library. The file cloud-example.c gives you an example of how to use our wrapper of libs3 to communicate with the Amazon S3 simulation server. The file rabin-example.c gives you an example for Rabin Segmentation API. Use the "`make`" command under `src/cloudfs/` to create the binary code of "`cloudfs`" in the directory "`src/build/bin/cloudfs`".

- Amazon S3 simulation web server:

  The file "`src/s3-server/s3server.pyc`" is the compiled python code that simulates Amazon S3 storage service. To run the web server, you can use the command line: "`python s3server.pyc`". The web server depends on the Tornado web server framework (http://www. tornadoweb.org/), which has been already installed inside the Virtual Box Image. It stores all the files by de- fault in `/tmp/s3/` (do not change this). To enable logging, you can simply run it with parameter "`--verbose`". More options can be checked out by using "`--help`" option.

- VirtualBox disk images:

  There are two images used by Virtual Box: ubuntu10.10-OS.vdi which is the OS disk image and SSD.vdi which is the SSD disk image. Instructions to setup VirtualBox using these .vdi files are included in the `README` file after you unpack the vbox images.tar.gz files.

- VirtualBox setup scripts:

  There are three scripts, `format_disks.sh`, `mount_disks.sh` and `umount_disks.sh`, that are required to manage the VirtualBox environment with the SSD. All the scripts are placed in the `scripts/` directory and have a `README` file that describes their usage in details. NOTE that these scripts are provided for your assistance; it is a good idea to write your own scripts to debug and test your source code.

Deliverables

The homework source code and project report will be graded based on the criteria given below (note: this is the rough criteria and is subject to change).

- 25 % Part 1 (Hybrid FS)
- 25 % Part 2 (Data Reduction: Deduplication + Compression)

- 25 % Part 3 (Caching)
- 25 % Project report, source code documentation, etc. Target size should be 5 or fewer pages.

For each of the Part 1, 2 and 3 milestones, you should submit a tar file that contains only a directory "`src/`" with the source files of CloudFS. You should use the same code structure as given in handout, and make sure that there exists a Makefile that can generate the binary "`src/build/bin/cloudfs`". We will test this in Autolab, and your code should compile correctly (Test this yourself!).

In your final "*report*" submission, you should submit a file "`andrewId.tar.gz`" which should include at least the following items (and structure):

- `src/` directory with the source files, and any test suites you used for evaluation in your report. (Please keep the size of test suite files small (1MB), otherwise omit them.)
- `andrewId.pdf` containing your 5-page report
- `suggestions.txt` file with suggestions about this project (what you liked and disliked about the project and how we can improve it for the next offerings of this course).

Source code documentation:

The `src/` directory should contain all your source files. Each source file should be well commented highlighting the key aspects of the function without a very long description. Feel free to look at well-known open-source code to get an idea of how to structure and document your source distribution.

Project report:

The report should be a PDF file no longer than 5 pages in a single column, single-spaced 10-point Times Roman font with the name `AndrewID.pdf`. You should be now know how annoyed instructors can be when you forget to present your Andrew ID front and center!

The report should contain design and evaluation of all three parts, i.e. discuss Part 1, Part 2 and Part 3 goals, design, evaluation and evidence of success. The design should describe the key data-structures and design decisions that you made; often figures with good descriptions are helpful in describing a system. The evaluation section should describe the results from the test suite provided in the hand-out and your own test suite.

Your report must answer the following questions explicitly:

- Explain all the cost/performance tradeoffs that you did as part of this project
- Explain the tradeoffs in choosing the right average segment size in deduplication
- Explain how deduplication and compression are used in your design?

How to submit ?

All your submissions will go to Autolab. Autolab's results will be used in your grade, but we will also do additional tests that Autolab does not run and return to you.

For the final submission, include both source code and project report as described above. Please use the same directory structure provided in handout to ease our grading.

Useful Pointers

- http://fuse.sourceforge.net/ is the de facto source of information on FUSE. If you download the latest FUSE source code, there are a bunch on the examples included in the source. In addition, the documentation about FUSE internals is helpful in understanding the behavior of FUSE and its data-structures:

  http://fuse.sourceforge.net/doxygen/

  You can Google for tutorials about FUSE programming. Some useful tutorials can be found at:

  http://www.ibm.com/developerworks/linux/library/l-fuse/

  http://www.cs.nmsu.edu/pfeiffer/fuse-tutorial/

- Disk IO stats are measured using `vmstat -d`. More information on can be found using the man pages. btrace and blktrace are useful tools for tracing block level IO on any device. Read their man pages to learn about using these tools and interpreting their output.

- We have provided instructions to setup VirtualBox in a `README` file in the `vbox_images.tar.gz` for this project. More information about VirtualBox can be found at the following URLs:

  http://www.virtualbox.org

  http://www.virtualbox.org/wiki/End-user_documentation

- We have provided instructions on Amazon S3 API specifications. More information about Amazon S3 API specifications can be found at the following URLs:

  http://libs3.ischo.com.s3.amazonaws.com/index.html

  http://aws.amazon.com/s3/

- IBM's research paper on mixing deduplication and compression Mixing Deduplication and Compression on Active Data Sets

- zlib references

  http://www.zlib.net/

  http://en.wikipedia.org/wiki/Zlib

  http://www.zlib.net/feldspar.html

  http://www.zlib.net/zlib_tech.html

  http://www.zlib.net/manual.html

Appendix

Amazon S3 API Specifications

To simulate the Amazon S3 cloud storage environment, we provide you with a web server running locally in Virtual Box. This web server supports basic Amazon S3 compatible APIS including: `LIST, GET, PUT, DELETE` on buckets and objects. On the client slide, you will use a open-source S3 client-library called "`libs3`" in FUSE to allow CloudFS to communicate with web server.

The libs3 C library (http://libs3.ischo.com/index.html) provides an API for accessing all of S3's functionality, including object accessing, access control and so on. However, in this project, we only need to use a subset of its full functionality. For your convenience, we provide a wrapper of libs3 C libaray in files "`cloudapi.h`" and "`cloudapi.c`", although you are free to use original libs3 C library for better performance. All functions in the wrapper are listed in "`cloudapi.h`". The following example shows how to use these wrapper functions:

```
1  FILE *outfile;
2  int get_buffer(const char *buffer, int bufferLength) {
3      return fwrite(buffer, 1, bufferLength, outfile);
4  }
5
6  void test() {
7      cloud_init("localhost:8888");
8      outfile = fopen("./test", "wb");
9      cloud_get\_object("test_bucket", "test", get_buffer);
10     fclose(outfile);
11     cloud_destroy();
12 }
```

To use any wrapper function, you first have to initialize a lib3 connection by calling "`cloud_init(HOSTNAME)`" (shown in line 7), where `HOSTNAME` specifies the IP address that the S3 web server binds to. Line 8 uses the call "`cloud_get_object`" to download the file "`S3://test_bucket/test`" from cloud to a local file "`./test`". The "`cloud_get_object`" call, takes a bucket name, a file name, and a callback function as input parameters. In the internal implementation of "`cloud_get_object`" call, it retrieves data from S3 server into a buffer, and once the buffer is full or the whole object is downloaded, it will then pass the buffer to the callback function for data processing. Line 2 to 4 shows a callback function that simply writes the received data into the local file system. For more examples of using the wrapper of libs3, look at the sample code "`src/cloudfs/cloud-example.c`".

Rabin Segmentation API Specifications

The Rabin segmentation API should be used to define the segment boundaries. First you need to initialize the data structures by calling `rabin_init()`. Once initialized, use `rabin_segment_next()` to run the contents of a file through the Rabin fingerprinting algorithm. You may have to call the latter function multiple times in a loop. Once you are done with all the fingerprinting for one file, you can call `rabin_free()`

at the end. You can user `rabin_reset()` re-initialize the datastructure. We suggest you use initialize only once at startup and then use `rabin_reset()` to use the same datastructure for multiple files. An example program that uses this API is provided with the source code. It prints out the segment lengths and their MD5 sums. To build the example do “`make rabin-example`”.

```
cloudfs $ make rabin-example
build/obj/rabin-example.o: Compiling object
build/bin/rabin-example: Building executable

cloudfs$ ls -l /tmp/bigfile /tmp/smallfile
-rw-r--r-- 1 guest guest 20480 2013-03-17 17:06 /tmp/bigfile
-rw-r--r-- 1 guest guest     1 2013-03-17 17:07 /tmp/smallfile

cloudfs$ cat /tmp/bigfile | ./build/bin/rabin-example
3190 cb26f4d170a93009e0d1c5b29b31796e
7862 409348f2fdd9aa2d18641a0d3d113108
3868 5e27fd72f47bb5a263f6443e39a8c5d7
5560 d61085ec3b8749ff57c4bbb4590760b3

cloudfs$ cat /tmp/smallfile /tmp/bigfile | ./build/bin/rabin-example
3191 c62235153f6148d2cc9fb94ef576b57b
7862 409348f2fdd9aa2d18641a0d3d113108
3868 5e27fd72f47bb5a263f6443e39a8c5d7
5560 d61085ec3b8749ff57c4bbb4590760b3
```

That’s the magic of rabin fingerprinting at work!!!

Below is the API Interface header `dedup.h` :

```
/**
 * @file dedup.h
 * @author Pavan Kumar Alampalli
 * @date 15-mar-2013
 * @brief Interface header for dedup library
 *
 * This header file defines the interface for the dedup library.
 * The interface follows the basic init, call-in-a-loop, free
 * pattern.  It also declares an opaque structure (rabinpoly_t)
 * that will be used by all the functions in the library to maintain
 * the state.
 *
 * Note that the memory allocated by rabin_init() has to be freed
 * the calling rabin_free() in the end. You can reuse the same
 * rabinpoly_t structure by calling a rabin_reset().
 */
```

```
#ifndef _DEDUP_H_
#define _DEDUP_H_

/**
 * Rabin fingerprinting algorithm structure declaration
 */
struct rabinpoly;
typedef struct rabinpoly rabinpoly_t;

/**
 * @brief Initializes the rabin fingerprinting algorithm.
 *
 * This method has to be called in order to create a handle that
 * can be passed to all other functions in the library. The handle
 * should later be free'ed by passing it to rabin_free().
 *
 * The window size is the size of the sliding window that the
 * algorithm uses to compute the rabin fingerprint (~32-128 bytes)
 *
 * @param [in] window_size Rabin fingerprint window size in bytes
 * @param [in] avg_segment_size Average desired segment size in KB
 * @param [in] min_segment_size Minumim size of the produced segment in KB
 * @param [in] max_segment_size Maximum size of the produced segment in KB
 *
 * @retval rp Pointer to a allocated rabin_poly_t structure
 * @retval NULL Incase of errors during initialization
 */
rabinpoly_t *rabin_init(unsigned int window_size,
                        unsigned int avg_segment_size,
                        unsigned int min_segment_size,
                        unsigned int max_segment_size);

/**
 * @brief Find the next segment boundary.
 *
 * Consumes the characters in the buffer and returns when it
 * finds a segment boundary in the given buffer. The segments
 * defined by this function will never be longer than max_segment_size
 * and will never be shorter than min_segment_size.
 *
 * It returns the number of bytes processed by the rabin fingerprinting
 * algorithm. Note that it can be <= the number of bytes in the
 * input buffer depending on where the new segment was found (similar
 * to shortcounts in write() system call). So you may need to call
 * this function in a loop to consume all the bytes in the buffer.
 *
 * @param [in] rp Pointer to the rabinpoly_t structure returned by rabin_init
 * @param [in] buf Pointer to a characher buffer containing data
 * @param [in] bytes Number of bytes to read from the buf
 * @param [out] is_new_segment Pointer to an integer flag indicating segment
 *                             boundary. 1: new segemnt starts here
 *                                       0: otherwise.
 *
 * @retval int Number of bytes processed by the rabin algorithm.
 *         -1  Error
 */
int rabin_segment_next(rabinpoly_t *rp,
                       const char *buf,
                       unsigned int bytes,
                       int *is_new_segment);

/**
 * @brief Resets the Rabin Fingerprinting algorithm's datastructure
 *
 * Call this function to reuse the rp for a different file or stream.
 * It has the same effect as calling rabin_init(), but does not do
 * any allocation of rabinpoly_t.
 *
 * @param [in] rp Pointer to the rabinpoly_t structure returned by rabin_init
 *
 * @retval void None
 */
void rabin_reset(rabinpoly_t *rp);

/**
 * @brief Frees the Rabin Fingerprinting algorithm's datastructure
 *
 * This function should be called at the end to free all the memory
 * allocated by rabin_init.
 *
 * @param [in] p_rp Address of the pointer returned by rabin_init()
 *
 * @retval void None
 */
void rabin_free(rabinpoly_t **p_rp);

#endif /* _DEDUP_H_ */
```

zlib Compression API Specifications

This section presents the zlib API and a sample program that uses this API to perform compression and decompression

zlib APIs

Below is the zlib API header `compressapi.h` :-

```
/**
 * @file compressapi.h
 * @author Mukul Kumar Singh
 * @date 13-MAR-2014
 * @brief Interface header for compression library
 *
 * This header file defines the interface for the compress library.
 *
 * This file contains the deflate and infalte interface to compress and
 * decompress a file.
 */


#ifndef _COMPRESSAPI_H
#define _COMPRESSAPI_H

#include <stdio.h>

/** @brief This api is used to compress/deflate a file
 *
 * This method compresses a file depending upon the given compression level.
 *
 * @param source source file pointer of file to be compressed.
 * @param dest output file pointer where the compresed file should be saved
 * @param len length of the segment of the file, which should be compressed,
 *            this is relative to the current offset of the file.
 * @param level level of compression to be used while compressing the file
 *
 * @return returns the length of the compressed segment
 */
int def(FILE *source, FILE *dest, int len, int level);

/** @brief This api is used to decompress/inflate a file
 *
 * This method decompresses a file.
 *
 * @param source source file pointer of file to be compressed.
 * @param dest output file pointer where the compresed file should be saved
 *
 * @return returns 0 if success, negative otherwise
 */
int inf(FILE *source, FILE *dest);

#endif
```

Sample usage program for zlib library.

Below is the sample usage program `compress_example.c` :-

```
/* compress_example.c : Reference from zpipe.c: example of proper use of zlib's
                        inflate() and deflate()
   Not copyrighted -- provided to the public domain
   Version 1.4  11 December 2005  Mark Adler */

/* Version history:
   1.0  30 Oct 2004  First version
   1.1   8 Nov 2004  Add void casting for unused return values
                     Use switch statement for inflate() return values
   1.2   9 Nov 2004  Add assertions to document zlib guarantees
   1.3   6 Apr 2005  Remove incorrect assertion in inf()
   1.4  11 Dec 2005  Add hack to avoid MSDOS end-of-line conversions
                     Avoid some compiler warnings for input and output buffers
 */

#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "zlib.h"
#include "compressapi.h"

#if defined(MSDOS) || defined(OS2) || defined(WIN32) || defined(__CYGWIN__)
#  include <fcntl.h>
#  include <io.h>
#  define SET_BINARY_MODE(file) setmode(fileno(file), O_BINARY)
#else
#  define SET_BINARY_MODE(file)
```

```
#endif

#define CHUNK 16384

/* report a zlib or i/o error */
void zerr(int ret)
{
    fputs("zpipe: ", stderr);
    switch (ret) {
    case Z_ERRNO:
        if (ferror(stdin))
            fputs("error reading stdin\n", stderr);
        if (ferror(stdout))
            fputs("error writing stdout\n", stderr);
        break;
    case Z_STREAM_ERROR:
        fputs("invalid compression level\n", stderr);
        break;
    case Z_DATA_ERROR:
        fputs("invalid or incomplete deflate data\n", stderr);
        break;
    case Z_MEM_ERROR:
        fputs("out of memory\n", stderr);
        break;
    case Z_VERSION_ERROR:
        fputs("zlib version mismatch!\n", stderr);
    }
}

/* compress or decompress from stdin to stdout */
int main(int argc, char **argv)
{
    int ret;
    int len = 0;
    FILE *infp = NULL;
    FILE *outfp = NULL;

    if (argc == 3) {
        printf("compressing\n");
        infp = fopen(argv[1], "r");
        outfp = fopen(argv[2], "w");
    } else if (argc == 4) {
        printf("de compressing\n");
        infp = fopen(argv[2], "r");
        outfp = fopen(argv[3], "w");
    } else {
        printf("invalid\n");
        return 0;
    }

    /* do compression if no arguments */
    if (argc == 3) {
        fseek(infp, 0, SEEK_END);
        len = ftell(infp);
        fseek(infp, 0, SEEK_SET);

        ret = def(infp, outfp, len, Z_DEFAULT_COMPRESSION);
    }

    /* do decompression if -d specified */
    else if (argc == 4 && strcmp(argv[1], "-d") == 0) {
        ret = inf(infp, outfp);
    }
}
```