### 1.Source code

Download the source codes from: <https://github.com/iotlpf/HiDeStore>

### 2.Datasets

We use 4 public datasets in our evaluations.

1. linux kernel: <https://mirrors.edge.kernel.org/pub/linux/kernel/>

2. gcc: <https://ftp.gnu.org/gnu/gcc>

3. fslhomes: <http://tracer.filesystems.org/traces/fslhomes/>

4. macos: <http://tracer.filesystems.org/traces/macos/>

You can download some versions for evaluations. We take the dataset ‘linux kernel’ as an example to show how to organize the data, and other datasets are similar.

1. Download the compressed data into the directory: data/kernel/tarbz.

2. Modify the following lines of file kernel/jieya.sh:

Line 3: the correct path to kernel/tarbz

Line 7: the correct path to kernel/tarbz

Line 9: the correct path to kernel/kernel

3. chmod +x jieya.sh

4. ./jieya.sh

Modify the data paths in script files:

vim script/dbenchmark.sh

vim script/run\_benchmark.sh

vim script/restore.sh

vim script/drestore.sh

all the above files should have the correct paths to access the uncompressed data:

kernel\_path="correct path to/kernel/kernel"

gcc\_path=" correct path to /gcc/gcc"

fsl\_path=" correct path to /fslhomes/fsl/"

macos\_path=" correct path to /macos/macos"

run the command:

chmod +x ./script/\*

### 3.Environment

Linux 64bit

### 4.Dependencies

1. zlib, libffi, libssl, glib

2. libssl-dev is required to calculated SHA-1 digest

3. GLib 2.32 or later version

* + libglib.so and glib.h may not be found when you first install it.
  + The header files (that originally locate in /usr/local/include/glib-2.0 and /usr/local/lib/glib-2.0/include) are required to be moved to a searchable path, such as "/usr/local/include".
  + Also a link named libglib.so should be created in "/usr/local/lib".

### 5.Running

#### 0.The metadata directory

Build a new directory to store the metadata.

mkdir myworking (or other name)

Change the third line in File destor.config:

Line 3: working-directory “the correct path to the above directory”

Change the third line in Files rebuild and drebuild:

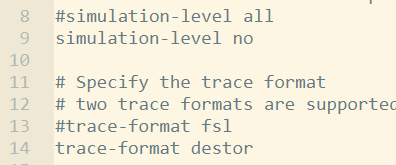
Line 3: DIR= the correct path to the above directory

Running the command:

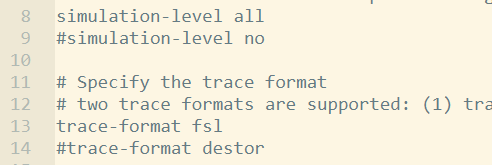
chmod +x reduild drebuild

Notice:

To evaluate the datasets linux kernel and gcc, we need to change the File destor.config:



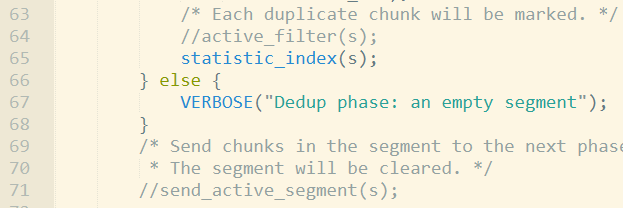
To evaluate the datasets fslhomes and macos, we need to change the File destor.config:



#### 1.The chunk number distribution

To obtain the chunk number distribution over different backup versions like Figure 3 in our paper, we need to modify the files as shown below:

Lines 64-71 in File src/lpf/active\_dedup\_phase.c:



Line 135 in File src/lpf/active\_index.c:



Compile:

make

Run the code:

./script/run\_benchmark.sh kernel [or gcc, fsl, macos]

The results are shown in the File log:

vim log

At the line after “flushing active cache” in the log, we obtain a sequence of chunk numbers, which represent the numbers of chunks of different versions. For example, after processing the first version, we obtain 1728, which represent that there are 1728 V1 chunks.



After processing the second version, we obtain 793, 1810, which represent that there are 793 V1 chunks and 1810 V2 chunks.



After processing the third version, we obtain 792, 479, 1889, which represent that there are 792 V1 chunks, 479 V2 chunks and 1810 V3 chunks.

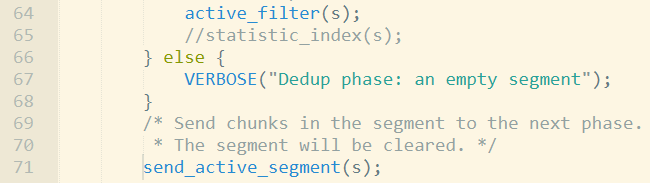


After processing the subsequent backup versions, we observe that the chunks not appearing in the current backup version have a low probability to appear in the subsequent backup versions.

#### 2.The deduplication ratios

To obtain the deduplication ratios, we need to modify some files as shown below:

Lines 64-71 in File src/lpf/active\_dedup\_phase.c:



Compile:

make

Run the code:

./script/dbenchmark.sh kernel [or gcc, fsl, macos]

./script/run\_benchmark.sh kernel [or gcc, fsl, macos]

The results are shown in the File log:

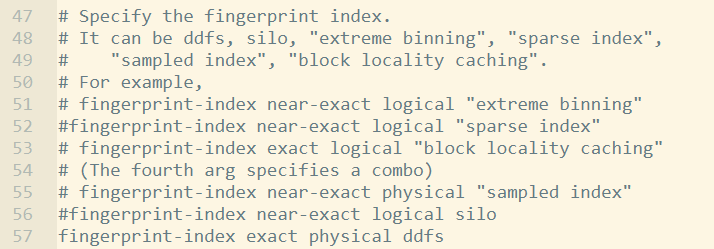
vim log

The line “dedup ratio” represents the deduplication ratio after processing each version. We calculate the average deduplication ratio of all backup versions.

The deduplication ratios of other schemes are evaluated by destor:

<https://github.com/fomy/destor>

We can obtain the results by change File destor.config:



Run the codes:

./script/dbenchmark.sh kernel [or gcc, fsl, macos]

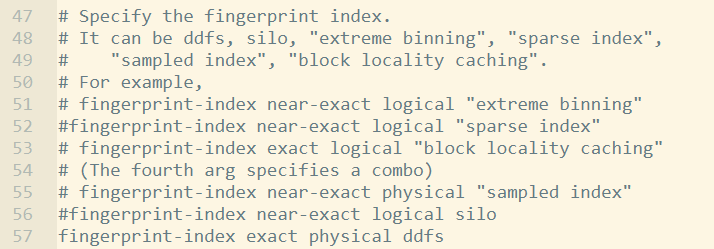
And the results are shown in File dlog：

vim dlog

#### 3.Lookup overhead

Existing schemes are evaluated by destor: <https://github.com/fomy/destor>

We can obtain the results by change File destor.config:



Run the codes:

./script/dbenchmark.sh kernel [or gcc, fsl, macos]

And the results are shown in File dlog：

vim dlog

The line “index\_overhead.lookup\_request” represents the number of request to access the full index table. The line “total size(B)” represents the total size of the processed data. We obtain the metric “Lookup request per GB” by the following equation:

Lookup request per GB = (lookup\_request)\*1024\*1024\*1024 / total size(B)

HiDeStore doesn’t need to access the full index table during deduplicating, since the chunks that have the high probability to be deduplicated have been stored in the fingerprint cache before deduplication. To facilitate fair comparisons, we use the same unit size of a lookup request (i.e., each request obtains 1024 indexes) as the traditional schemes, and evaluate the number of the lookup requests.

Run the code:

./script/run\_benchmark.sh kernel [or gcc, fsl, macos]

The results are shown in the File log:

vim log

The line “Init double cache buffer size” represents that how many indexes are accessed. We obtain the lookup\_request by the following equation:

Lookup\_request = ceil((init double cache buffer size)/1024);

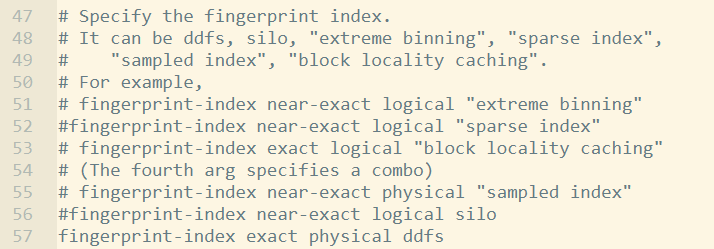
We obtain the metric “Lookup request per GB” by the following equation:

Lookup request per GB = (lookup\_request)\*1024\*1024\*1024 / total size(B)

#### 4.The index table overhead

Other schemes are evaluated by destor: <https://github.com/fomy/destor>

We can obtain the results by change File destor.config:



Run the codes:

./script/dbenchmark.sh kernel [or gcc, fsl, macos]

And the results are shown in File dlog：

vim dlog

The line “init\_kvstore\_htable” represents the number of indexes in the index table, each index is a 20 byte hash value calculated by SHA-1. The line “total size(B)” represents the total size of the processed data. The size of index table increases as the processed data increase. We obtain the metric “Space overhead per MB” by the following equation:

Space overhead per MB = (init kvstore\_htable)\*20\*1024\*1024 / total size(B)

Unlike them, HiDeStore doesn’t need extra space to store the indexes, since HiDeStore deduplicates one backup version against its previous one, and the fingerprint indexes of all chunks in previous backup version have been stored in the recipe. We check the recipe of the previous backup version during deduplication, rather than maintaining an index table. Hence, the space overhead per MB for storing hash table is 0.

#### 5. Running overhead

Updating recipes:

When we deduplicate 10 backup versions, we can run the command:

./build/destor -u9

We update 9, since the number starts from 0. We obtain the “total time” to update all recipes, and we calculate the metric “Latency per version” by the following equation:

Latency per version = (total time) / 10

Removing cold chunks:

Change the file:

Line 135 in File src/lpf/active\_index.c:



Compile:

make

Running:

./script/run\_benchmark.sh kernel [or gcc, fsl, macos]

The results are shown in the File log:

vim log

The line “remove time” represents the time overhead to remove the cold chunks.

#### 6.Restore performance

The data are restored after deduplication:

./script/dbenchmark.sh kernel [or gcc, fsl, macos]

./script/run\_benchmark.sh kernel [or gcc, fsl, macos]

./script/drestore.sh

./script/restore.sh

The results of existing schemes are shown in File drlog:

vim drlog

The results of HiDeStore are shown in File rlog:

vim rlog

The line “speed factor” represents the restore performance of each version.