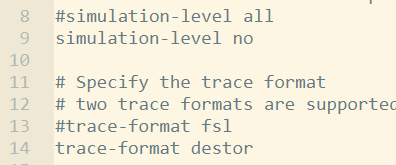
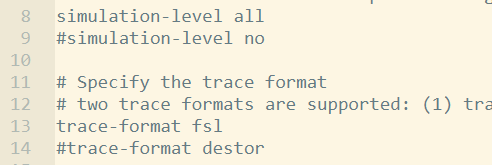
#### 1.Different configurations for various workloads

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



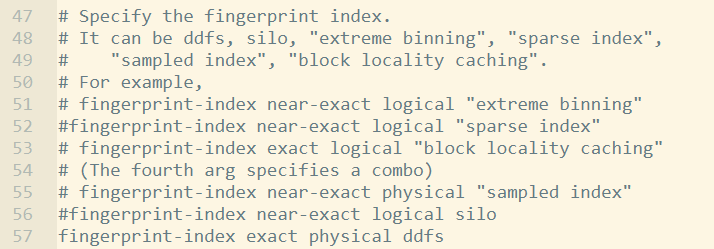
To evaluate the datasets fslhomes and macos, we need to change the File destor.config:



#### 2.Evaluate the related work (the compared schemes)

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

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



Run the commands:

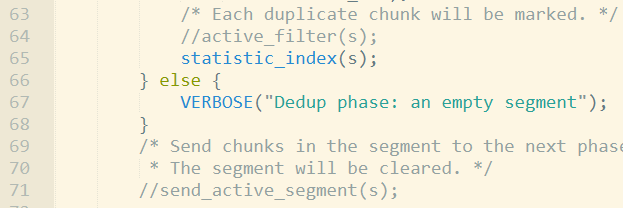
./script/dbenchmark.sh kernel [or gcc, fsl, macos]

./script/drestore.sh

#### 3.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.

#### 4.The deduplication ratios

**Other schemes (i.e., the compared schemes)**

Change the file destor.config and run the codes:

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

And the results are shown in File dlog：

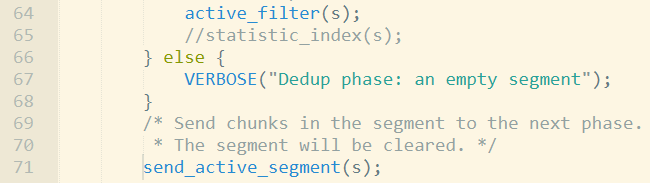
vim dlog

The line “deduplication ratio” represents the deduplication ratio after processing each version.

**HiDeStore**

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.

#### 5.Lookup overhead

**Other schemes**

The results are shown in File dlog：

vim dlog

The line “index\_overhead.lookup\_requests” 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 requests per GB” by the following equation:

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

**HiDeStore**

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.

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\_requests by the following equation:

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

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

Lookup requests per GB = (lookup\_requests)\*1024\*1024\*1024 / total size(B)

#### 6.The index table overhead

**Other schemes**

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)

**HiDeStore**

Unlike existing schemes, HiDeStore doesn’t need extra space to store the full index table, 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.

#### 7. Running overhead

**Updating recipes:**

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

./build/destor -u9

We update 9, since the version number starts from 0. For example, the version numbers [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] represent 10 different backup versions in HiDeStore. W 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.

#### 8.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 9

./script/restore.sh 9

The number 9 represents that we restore 10 backup versions.

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.