DEDUPLICATION AND COMPRESSION BENCHMARKING SUPPORT IN FILEBENCH

CSE506 Project Report

By
Rami Al-Rfou'
Nikhil Patwardhan
Phanindra Bhagavatula

December 11, 2010

Contents

Abstract

Deduplication systems look for repeating patterns of data at the block and bit levels. When multiple instances of the same pattern are discovered, the system stores a single copy of the pattern. However, most of the popular file-system benchmarks generate data in a manner that does not enable realistic benchmarking of deduplication systems. Controlling the entropy of the data that is used while benchmarking deduplication systems is one way of overcoming this limitation. In this project, we integrated entropy-based data generation in the Filebench file-system benchmarking suite.

By assuming the emission of a byte in the data stream as an event, we generate data that can take values from 0 to 8 bits/byte. The user is able to control the value of entropy by specifying it in the workload to be run. We show that by varying the amount of entropy of data that is either written to or read from a disk using a deduplicated file-system, the benchmarking results are more pertinent to the actual behavior of such a file-system.

Chapter 1

Introduction

1.1 Filebench

Filebench is a framework for simulating different file systems workloads. Such simulation is helpful to identify performance weaknesses and bottlenecks in file systems. Filebench interface relies on an interpreter that reads workload Definition language scenarios in `f` scripts files. The model language is expressive which allows Filebench to run so complicated testing workloads. Filebench at the same time hides the complexity of running multi threaded systems and synchronizing them. Moreover, it offers an integrated system to measure the latency and throughput for each operation[1].

Nowadays, complex applications such as relational databases have sophisticated relations in the system. Setting up each application in environment that already has other applications can complicate the benchmarking and make it hard to evaluate the results. Filebench gives the power to simulate an application from the perspective of the filesystem isolating any other interactions that the application can have with other components of the system.

1.2 SDFS

Deduplication becomes more important because of the new features that nowadays file systems are offerings as backups and snapshots. Such features keeps a lot of redundant data on the storage medium. Cloud computing is forcing more challenges on file systems running on servers in terms of scalability and storage footprint. Many of the virtualization environments are keeping copies of the operating systems images running on the system. Most of those images are similar to large extent that deduplication can be so useful[2].

SDFS is a open source project to implement a file system with deduplication support. Deduplication is a technique used to save disk space by keeping track of redundant data and save only one copy of that data. Deduplication differs from compression that the scope of redundancy that has to be avoided is across the file system and on file or at least chunks of the file basis. On the other hand compression tries to reduce the size of a file by looking at its byte stream.

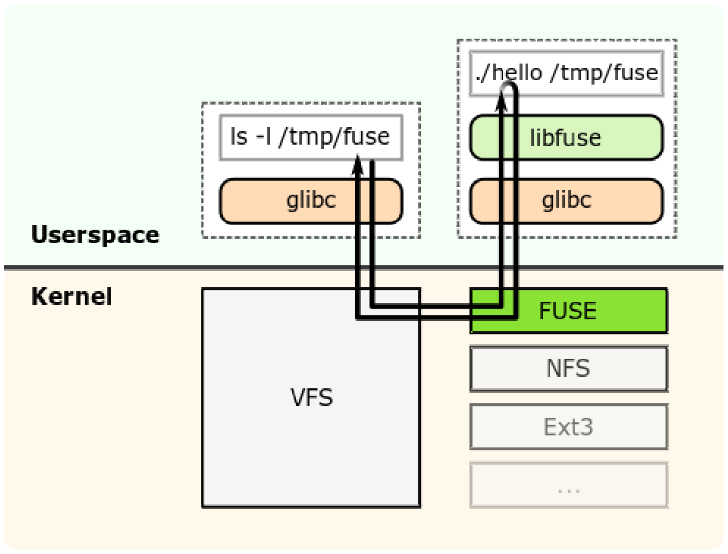

Figure 1.1: Userspace file system architecture in linux[4]

SDFS offers two mechanism to achieve deduplication: inline and batch mechanisms. The inline mechanism stores chunks of the files and take a finger print of each chunk by calculating the hash function of the data and then write it directly to storage medium. In case a similar chunk or file has to be written again, SDFS does not write it instead just references to the existing chunk on the disk. The batch mode applies a post-process deduplication. The data is stored directly to the disk then the file system perdiodically checks for any chance of duplicated chunks.

SDFS is implemented as a userspace file system and it is written in JAVA. This gives flexibility to use SDFS without updating the system. Beside that implementing it in JAVA makes it easier to support multiple operating systems.

From figure 1.1, it is clear the userspace file systems has a lot of similarity of stackable file systems. They allow the developer to add functionality to an existing file system without changing it. Moreover, they give the flexibility to make such improvements without updating the operating system. Userspace performance may suffer due to the extra communication overhead and context switching between kernel and userspace. However, it has a security advantage by reducing the size of code base runs in the kernel mode.

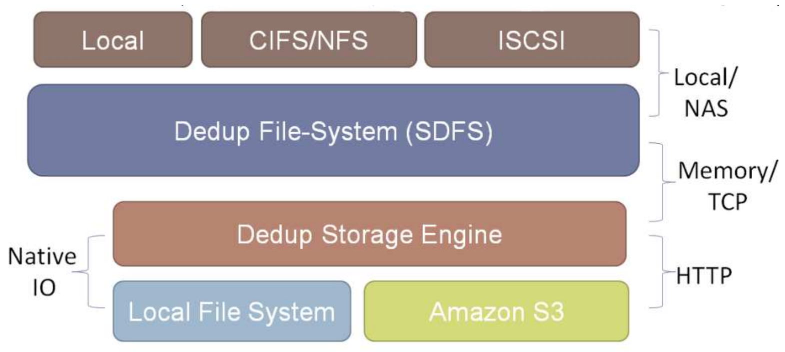

Figure 1.2: SDFS Architecture[3]

Figure 1.2 shows that SDFS has a server client design model. The client can be a process on the host machine or a network socket that writes data to the disk. SDFS integrates with NFS and VFS to support network and local operations.

Chapter 2

Design

The design of our project can be logically divided into three parts. Firstly, we need a mechanism to specify what we want Filebench to do, viz. to generate files with specific entropy values. Secondly, we need a way to generate data with arbitrary entropy. Lastly, we need a way to populate this data into files in the existing workflow of Filebench. The following sections describe and also explain the rationale behind the design of each of these parts.

2.1 Filebench Interpreter

We modified the Filebench Interpreter to enable Filebench to optionally accept various datasources. These datasources are meant to be used to populate files created during benchmarking process. The Interpreter has thus been modified to accept new attributes called `datasource` and `entropy` as a part of the fileset command. For example a valid fileset command is:

```
define fileset name=bigfileset,...,datasource=entro,entropy=3.4
```
The attribute `entropy` is not directly an attribute of `fileset` command. It is a subattribute of the datasource type. For example the below is an invalid fileset definition.

```
define fileset name=bigfileset,path=\$dir,...,entropy=3.4
```

All attributes for command `fileset` have a place in the `struct fileset`. The `datasouce` attribute has also been placed in the structure. A new place has been created in this structure for `datasource` attribute alone and not for `entropy` attribute. The logic is that, the only attribute that makes sense to be part of `fileset` is the type of data defined by datasource. The attributes which define the data itself of the `datasource` are irrelevant to be part of `fileset` datastructure.
Since the a datasource can have various attributes of the data constituting it, `datasource` object has a pointer to a list of attribute objects relevant to it. This is the reason `datasource` is of type `attr` and not `avd_t`. To accommodate a list of sub-attributes inside an attribute(like the `datasource`), `struct attr` has been modified.

2.2 Entropy Generator

2.2.1 Entropy function

The entropy is quantity that is defined for a set of data to quantify how much random it is. For any stream of data that is composed of n symbols, its entropy is given by the following equation:

$$Entropy = -\sum_{i=1}^{n} P(s_i) \lg P(s_i) \tag{2.1}$$

$P(s_i)$ is the probability that a symbol s_i is generated by the data generator. To calculate the probability of a symbol in a generated data, can be done by calculating the experimental probability according to equation 2.2

$$P(s) = \frac{number\, of\, s\, occurences}{size\, of\, data} \tag{2.2}$$

The maximum entropy that can be obtained from a data generated using n symbols is when the probability distribution of symbols is uniform. That entropy of a uniform distribution can by calculated as following

$$\begin{aligned} Entropy &= -\sum_{i=1}^{n} P(s_i) \lg P(s_i) \\ &= -\sum_{i=1}^{n} \frac{1}{n} \lg \frac{1}{n} \\ &= \frac{-1}{n} \sum_{i=1}^{n} - \lg n \\ &= \lg n \end{aligned} \tag{2.3}$$

To decrease the value of entropy, we can imbalance the uniform distribution. Using this method we can reach zero by increasing the probability of the first symbol to 1 and decreasing the others to zero. However, calculating the ϵ_i that should be added to subtracted from every $P(s_i)$ to reach a specific entropy value less than $\lg n$ can be hard. To simplify the situation we can focus on changing the probability of two symbols at a time. Our target to generate all values of $x \ni \lg(n-1) < x < \lg n$ just by changing the probability of two symbols.

$$\begin{aligned} Entropy = -[&(P(s_1) + \epsilon) \lg(P(s_1) + \epsilon) + (P(s_2) - \epsilon) \lg(P(s_2) - \epsilon) \\ &+ \sum_{i=3}^{n} P(s_i) \lg(P(s_i))] \end{aligned} \tag{2.4}$$

$$\begin{aligned} = -[&(\frac{1}{n} + \epsilon) \lg(\frac{1}{n} + \epsilon) + (\frac{1}{n} - \epsilon) \lg(\frac{1}{n} - \epsilon) \\ &+ \frac{n-2}{n} \lg(\frac{1}{n})] \end{aligned} \tag{2.5}$$

To prove that the entropy as a function of ϵ equal to any value between $\lg n, \lg(n-1)$, the maximum entropy using n, $n-1$ symbols respectively. We notice the following observations:

- Equation 2.5 shows that entropy is a function in one variable, ϵ.
- Equation 2.5 also shows that the entropy is a continuous function of ϵ on the interval $\epsilon \in [0, 1/n)$ as it is a result of adding and multiplying continuous functions on the same interval.
- Entropy is equal $\lg n$ when $\epsilon = 0$.
- Entropy is equal to $\lg(n) - \frac{2}{n}$ when ϵ reaches $1/n$
- $\lg(n) - \frac{2}{n} < \lg(n-1) \;\; \forall n > 2$ [1].

Given the above and using the median value theorem the Entropy function spans over the interval $(\lg(n-1), \lg n)$ using subinterval of ϵ values. To get the value of ϵ we can apply any numerical method to find the roots of the equation.

2.2.2 Random generator

In 2.2.1 we showed that any entropy value can be obtained using a slightly modified uniform distribution. Now, given that the probability distribution function (PDF) of our symbols is already calculated. How can we build a data source that generates the data with the given entropy ? Again we will use a uniform random source to help us.The idea that we will calculate the cumulative distribution function (CDF) of the symbols table first. Then use the output of a uniform random generator to search for the corresponding symbol of the random value. Every symbol has different size interval that correspond to its probability. Because the CDF is an increasing function, the CDF table is increasing also which allow us to use in our search a binary search algorithm.

Algorithm 1 Random Generator

```
Solve the equation to get the value of $\epsilon$
Calculate PDF
Calculate CDF
for $i = 1$ to Buffer size do
    $x$= Random number in $[0, 1)$
    index = search in which interval of CDF $x$ lie.
    return SymbolTable[index]
end for
```

[1]Can be verified by visiting
`http://www.wolframalpha.com/input/?i=lg\%28n-1\%29+-+\%28lg\%28n\%29++-+2\%2Fn\%29&a=*FunClash.lg-_*Log2.Log10-`

Because the symbol table has constant size, the cost of the binary search is also constant. This guarantees that the time complexity of our algorithm is linear. However, in 3.2 we will show that in practice the constant factors of such algorithm is not good enough. Moreover, will will present other different modifications of the algorithm.

2.3 Design Principles

During the process of designing we tried our best to maintain the following principles.

- **Minimal change**

  The scope of the changes is as minimal as possible. This applies to the size of the patch counted by number of lines and the number of files modified that were modified.

  The files that are modified

  - fileset.c/fileset.h
  - vars.c/vars.h
  - flowop_library.c
  - parser_lex.l
  - parser_gram.y
  - filebench.h
  - Makefile.am

  Moreover, we used any already available structure instead of reinventing the wheel.

- **Extensibility**
  The design of the parser has been made such that a new datasouce can be easily defined and any number of sub-attributes of this datasource can be specified without any significant amount of code change.

- **Backward compatibility**
  The patched Filebench runs all the old workload model files without modification. The patch is triggered only when the data source attribute is specified. The patch is surrounded by conditional compilation preprocessors,`CONFIG_ENTROPY_DATA_EXPERIMENTAL` , that enables the user to switch the functionality on or off at the compilation time.

- **Modularity**
  Any code that do not change the flow of the current Filebench code base, is separated and kept in separate `C` modules. Files added

  - sources.c/sources.h
  - entropy.c/entropy.h

2.4 Filebench Workflow Integration

The integration in Filebench workflow can be logically divided into the following parts:

2.4.1 Data Specification

Data intended to be written to or read from files could be specified in many ways for benchmarking purposes. One approach is by specifying its *entropy*. However, this is not the only way to specify a property of data. We currently support only entropy based data population, but keeping possible future requirements in mind we chose to create a separate structure which is dedicated to store all the specifications about the data itself. This way, more attributes of the data can be captured in this structure itself as and when new requirements arise. One of the fields of this structure is a function pointer that is set *dynamically* depending on the data specification. In our current implementation, this function pointer can be made to point to a function that populates a buffer with a certain entropy value.

2.4.2 Fileset Initialization

The `struct fileset` structure holds information that is relevant to a set of files. Entropy, and other information spceific to a *fileset* is recorded in this structure by the parser, as explained in section **??**. Hence, we put the above described structure as one of the members of `struct fileset`. The information recorded by the parser is used to appropriately initialize this structure by pointing its function pointer dynamically to a function that provides data in the desired format, while also initializing other data members as necessary. Once initialized, this function pointer can be used by the corresponding *flow operations* to populate their buffers in a generic manner.

2.4.3 Flow Operations

Flow operations represent workload actions and they carry out buffered I/O on the files. Using the dynamically set function pointer, we populate the buffer (in this case with the specified entropy) just before writing it to an open file in the following flow operations:

1. createfiles
2. write
3. writewholefile
4. appendfile
5. appendfilerand

Chapter 3

Implementation

3.1 Filebench Interpreter

The implementation of the Interpreter has been done in fashion that future extensibility of Filebench to accept different datasources with optional and variable number of sub-attributes is easy. Most of the code changes have been made to the `parser_gram.y` and `parser_lex.l` files. `parser_lex.l` file has a list of valid tokens. Two tokens (`datasource` and `entropy`) have been added to this file. Since the parser never recognized decimal numbers e.g. 3.4 earlier, the file has also been modified to accept decimal values. New tokens in the `parser_lex.l` are as below:

```
..
%%
..
datasource               { return FSA_DSRC;}
entropy                  { return FSA_ENTROPY;}
..
<INITIAL>[0-9]*\.[0-9]+  {  .. } // parse decimal values.
..
%%
..
```

`parser_gram.y` file has been modified to accept `datasource` as a parameter. `entropy` attribute is accepted by the grammar only if `datasource` parameter is present.
Following are the key rules in the grammar.

```
..
%%
..
//define fileset command supports only source_type and not entropy

files_define_command: FSC_DEFINE FSE_FILE { .. }
| FSC_DEFINE FSE_FILESET { .. }
| files_define_command files_attr_ops { .. }
| files_define_command files_attr_ops FSK_SEPLST \bf{source_type} { .. }
..
```

```
source_type: FSA_DSRC FSK_ASSIGN FSV_STRING { .. }
| FSA_DSRC FSK_ASSIGN FSV_STRING FSK_SEPLST source_define_params { .. }

..

// Support for multiple sub-attributes for datasource
source_define_params: source_define_param { .. }
| source_define_params FSK_SEPLST source_define_param { .. }
..

source_define_param: source_params_name FSK_ASSIGN attr_value { .. }

%%
..
```

To keep the parser generic, it was decided that the parser won't verify if a sub-attribute (like `entropy`) is valid for a particular type of `datasource` (like "entro").
The sub-attributes are stored in a list of type `struct attr` and the `datasource` object has a pointer to this list. Following is the new variable in `fileset` structure pointing to the `datasource` object.

```c
typedef struct fileset {
struct fileset  *fs_next;
avd_t       fs_name;
avd_t       fs_path;
avd_t       fs_leafdirs;
..
avd_t       fs_dirwidth;
..
struct attr *fs_datasource;
..
};
```

3.2 Entropy Generator

The entropy generator is organized into two modules `entropy.c` and `source.c`. `entropy.c` is a library that contains a group of helping functions to build data sources with random generation capabilities. `source.h` has the definition of the `source` and `source_operations` structures.

```c
struct source {
    double s_entropy;
    struct source_operations *s_ops;
};

struct source_operations {
    int (*fill)(struct source * datasource, void *buf, unsigned int size);
};
```

In `sources.c` we can find few declared instances of the later structure. Filebench uses `dummy_operations` by default unless an entropy is specified as a parameter to the fileset declaration statement. In that case, Filebench will assign `entropy_operations` to the fileset source structure. Both structures are minimal and can be expanded for any future needs.

3.2.1 Calculating the PDF

We explained in section 2.2.1 the algorithm that calculates the PDF. However, the algorithm generates an equation that is not easily solvable analytically. To overcome this problem, numerical methods can be used to solve the equation. We implemented the secant method to find the roots of the equation. The secant method was chosen as it is easy to implement, converge fast enough, only 20 iterations needed !. Not to mention that the requirements are easy to prepare, you have to specify the range that you expect the solution to be in.

3.2.2 Populating algorithm

Filebench is calling `entropy_fill` operation to fill the buffer with data with the specific entropy. `entropy_fill` is an interface that call the actual algorithm that calculates the PDF and populate the data.
Many algorithms were proposed to generate data, differs mainly in the way they generate the data stream out of the calculated PDF.
In section 2.2.2 we explained how we can map the PDF to generate symbols according to their probability. This method is called in literature the roulette selection algorithm. The function that implments that method is `entropy_search_fill`. Although the time complexity of such algorithm is linear. The constant factors can go up to 10; the binary search takes at most 8 comparisons as the symbols table size is 256.
In an effort to minimize the time used to fill the buffer the following methods are implemented:

- `entropy_cont_fill`.
  This algorithm fills the buffer with random data according to the PDF creating contiguous segments of symbols to minimize the computational power needed, it generates contiguous segments of symbols in the buffer. Every segment length is proportional to the symbol probability. To make same size buffers look different, we shuffle the symbols table before using them. This algorithm is the only one that does not affect Filebench behavior, it keeps the disk busy all the time as the default case. On the other hand, the level of randomization obtained is per file basis and the entropy is not homogeneous over the segments/pages of the file.

- `entropy_permutate_fill`
  It relies on `entropy_cont_fill` to fill the buffer with the data with the given entropy. After that it generates a random permutation using the `permutate` function of the given buffer and return it back. In this way, the entropy over the array is made homogeneous and not only per file basis. The permutation function has linear time complexity. However, the permutate function operates on randomly selected elements of the buffer at each step

it iterates over the buffer. The algorithm does not have local spatiality behavior, so it does not take advantage of the caching behavior of the memory hierarchy. This shows clearly as the overhead of the permutation step is 30x.

- `entropy_4k_fill`
  To help the `entropy_permutate_fill` function overcome the spatiality problem we can fill and then permutate 4 KiB segments of the file every time. This guarantees that the page will be cached all the time in the CPU caches. This reduces the overhead of the `permutate` function to 10x. This also does not guarantee the entropy to be homogeneous across segments that are not aligned with the pages offsets. The error in the entropy gets smaller as the segments we are considering get larger. The function should be sure that the pages filling algorithm is using the same set of symbols for each page, otherwise the entropy of the whole file will increase more than the specified value. If you are using `entropy_permutate_fill` that is calling `entropy_cont_fill` to fill the 4KiB pages then comment the symbols shuffle step.

- `entropy_lookup_fill`
  This algorithm initializes a vector using any fill method. The buffer is filled by looking up randomly different elements in the lookup table. It is the fastest algorithm that guarantees homogeneous entropy over the file. The overhead that this algorithm has is 4x, which means that `entropy_cont_fill` it is faster by a factor of 3.

3.3 Filebench Workflow Integration

We used an `#ifdef CONFIG_ENTROPY_DATA_EXPERIMENTAL` construct to optionally include our code at make time. We primarily modified three of the existing files in filebench to integrate our entropy generation logic into the main workflow:

3.3.1 fileset.h

We added a `struct source fs_ds` as a member of `struct fileset`. Although certain data is recorded by the parser into another member of `struct fileset` which is `struct attr* fs_datasource`, we found it necessary to store this information again in `struct source fs_ds`. The reasons for this were twofold. Firstly, all other attributes of a fileset are captured using the `struct attr` data structure. We decided not to modify this structure to remain compliant with the exising implementation. Secondly, it might be necessary to store derived results in the `struct source fs_ds`. To store such derived results would modify the original `struct attr* fs_datasource` even more.

3.3.2 fileset.c

We defined an additional function `fileset_init_datasource` that will initialize the `struct source fs_ds` data structure depending on the information available in `struct attr* fs_datasource`. A call to this function was added to two existing functions in this file:

1. fileset_create
2. fileset_alloc_file

The function fileset_alloc_file performs an I/O operation to populate the created file with data. Initializing the struct source fs_ds member ensures that preallocated files are created with the desired entropy. Initializing the struct source fs_ds in fileset_create ensures that all operations that will be done on files in this fileset will use the appropriate function to populate data in their buffers. This provides a generic way of populating data in files that is independent of the actual mechanism of populating data.

3.3.3 flowop_library.c

We support the following filesets with entropy specifications:

1. createfiles
2. write
3. writewholefile
4. appendfile
5. appendfilerand

These flow operations use the following functions in flowop_library.c to populate data in buffers:

1. flowoplib_write
2. flowoplib_writewholefile
3. flowoplib_appendfile
4. flowoplib_appendfilerand

We inserted a call using the dynamically set function pointer to fill the buffer with the specified entropy value. An example of this is shown below:

```
int fd = flowop->fo_fdnumber;
struct fileset *fs = threadflow->tf_fse[fd]->fse_fileset;
fs->fs_ds.s_ops->fill(&fs->fs_ds, iobuf, iosize);
```

In the example shown above, the existing flowop and threadflow variables are used to get the corresponding fileset. Once a handle on the fileset is procured, its fill function is used to populate the preallocated buffer iobuf with the entropy value specified in fs_ds.

Chapter 4

Setup

The simulations were run under the following conditions:

1. The machine specifications:
   - 2 Intel Xeon 2.8 GHz, 2GB RAM
   - 2 73G SCSI Disks. The second disk was used for benchmarking.
   - Ubuntu 10.04 LTS Server Edition 64bit.
   - Software stack: JDK 7 Early Access Build 117, Fuse 2.8.1, SDFS 1.0.1
2. The base filesystem that SDFS read and write from is formatted as ext2 partition.
3. we mount the SDFS partition with default parameters.
4. Between any two runs of Filebench, prepare.sh and cleanup.sh are called to the do the following:
   - The SDFS partition is unmounted
   - The chunkstore is cleared.
   - The ext2 partition is unmounted and formatted.
   - Clear any filebench side effects stored at /var/tmp
   - Drop memory caches by running `sync && echo 3 > /proc/sys/vm/drop_caches`
5. The write workload scenario is

```
set $dir=/mnt/sdfs
define fileset name=rami_fileset,path=$dir,size=1m,entries=50000,
    dirwidth=100000,prealloc=0,datasource=entro,entropy=7.0
define process name=filewriter,instances=1
{
  thread name=filewriterthread,memsize=10m,instances=10
  {
    flowop createfile name=createfile,filesetname=rami_fileset,fd=1
    flowop write name=writtfile,fd=1,iosize=1m
    flowop closefile name=closefile,fd=1
  }
}
run 300
```

6. The read workload scenario is

```
set $dir=/mnt/sdfs
define fileset name=rami_fileset,path=$dir,size=1m,entries=50000,
    dirwidth=100000,prealloc,datasource=entro,entropy=3.0
define process name=filereader,instances=1
{
  thread name=filereaderthread,memsize=10m,instances=10
  {
    flowop read name=readfile,filesetname=rami_fileset,iosize=1m
  }
}
run 300
```

7. The python script `robot.py`, is a command line tool that reads the workload files and replace the entropy to the specified value in a file named values, then generates a temporary file that will be fed to Filebench. For example:

```
$./robot.py -v ./values -l results2 -d /mnt/sdfs/rami_fileset+ \
-x /mnt/sdbdrive/dedupfs/files/rami_fileset -w work_write.f
```

Chapter 5

Results

5.1 Contiguous fill method results

The following results were generated by using `entropy_cont_fill`[1] method.

5.1.1 Writes: Bandwidth v/s Entropy

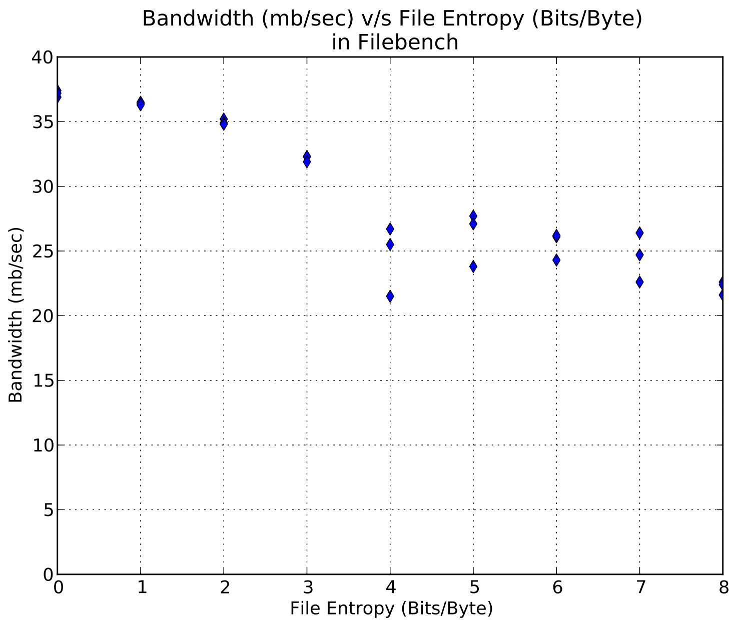

Figure 5.1: The bandwidth of all the write runs

[1]section 3.2

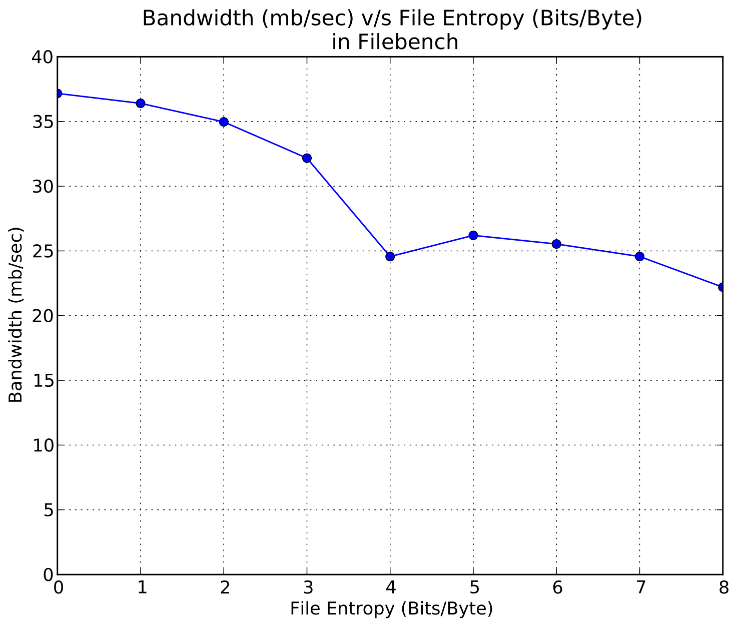

Figure 5.2: The average bandwidth over all the different write runs

Figures 5.1 and 5.2 show that bandwidth of the write operation is decreasing as the entropy is increased. This can be explained by the idea that the larger the entropy, the more is the ”randomness” and hence, the more is the effort SDFS spends to deduplicate the files. Figure 5.1 shows all values of bandwidth collected from 27 runs with small deviations. The average of all these runs for each entropy is calculated and presented in figure 5.2.

5.1.2 Writes: Latency v/s Entropy

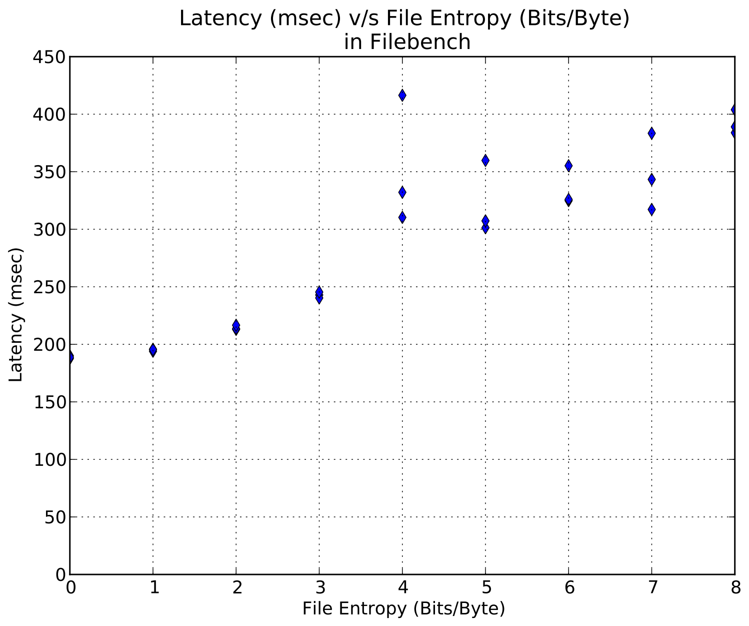

Figure 5.3: The latency of all the different write runs

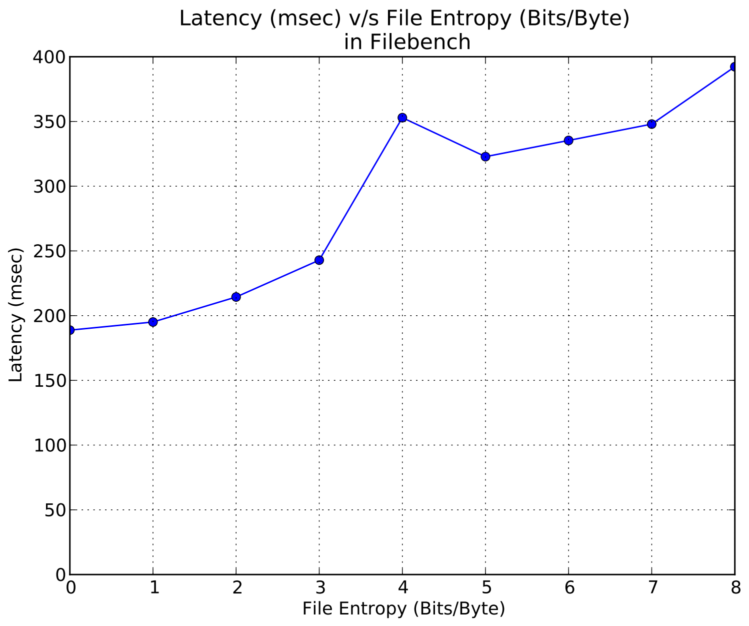

Figure 5.4: The average latency over all the write runs

Figures 5.3 and 5.4 show that latency of the write operation is increasing as the entropy increased. The larger the entropy the more effort that SDFS spend to deduplicate the files and the larger the latency will be for each write operation. Figure 5.3 shows all values of latency collected from 27 runs with small deviations. The average of all these runs for each entropy is calculated and presented in figure 5.4.

5.1.3 Writes: Operations/sec v/s Entropy

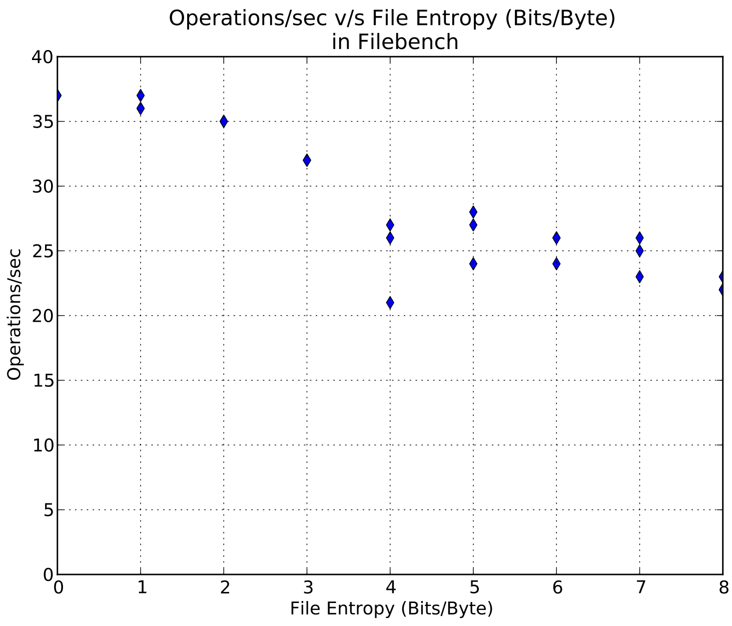

Figure 5.5: The operations execution-speed of all the write runs

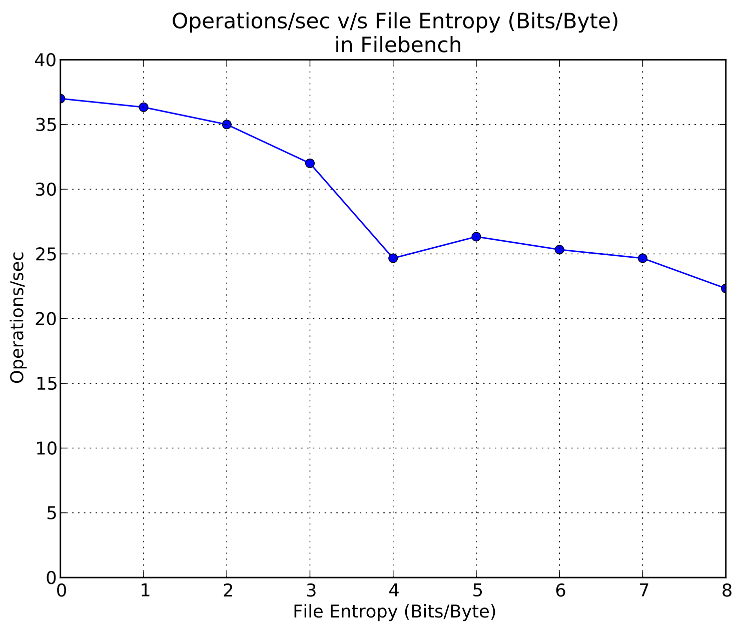

Figure 5.6: The average operations execution-speed over all the write runs

The results depicted by figure 5.5 show that the rate of operations goes down as entropy is increased. They also indicate the effect of ops/sec on bandwidth and latency. The less operations we can execute per second the less the size of data we can write which decrease the bandwidth. In the same manner if we want more time to execute a specific amount of operations, this means that the latency of the write operations is increased. Hence, these results are as expected. Figure 5.6 shows the average of the three readings obtained for each entropy value.

5.1.4 Reads: Bandwidth v/s Entropy and Latency v/s Entropy

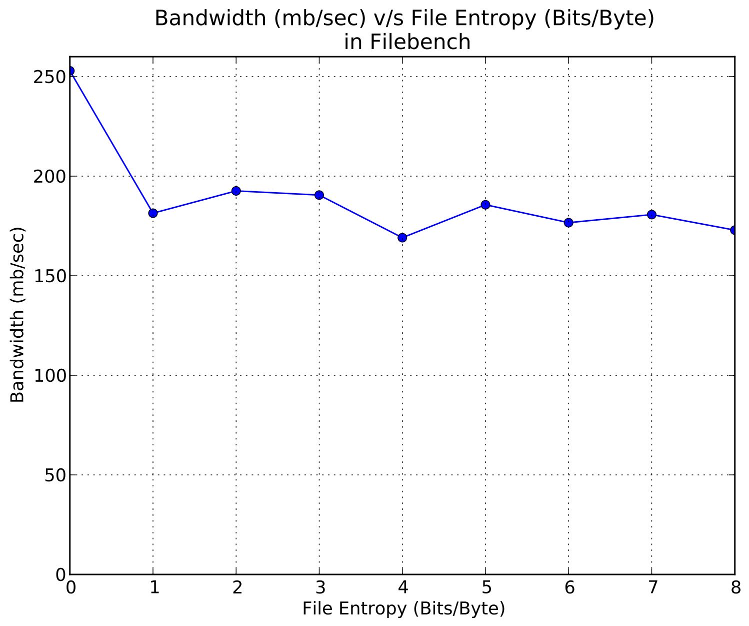

Figure 5.7: The bandwidth of all the read runs

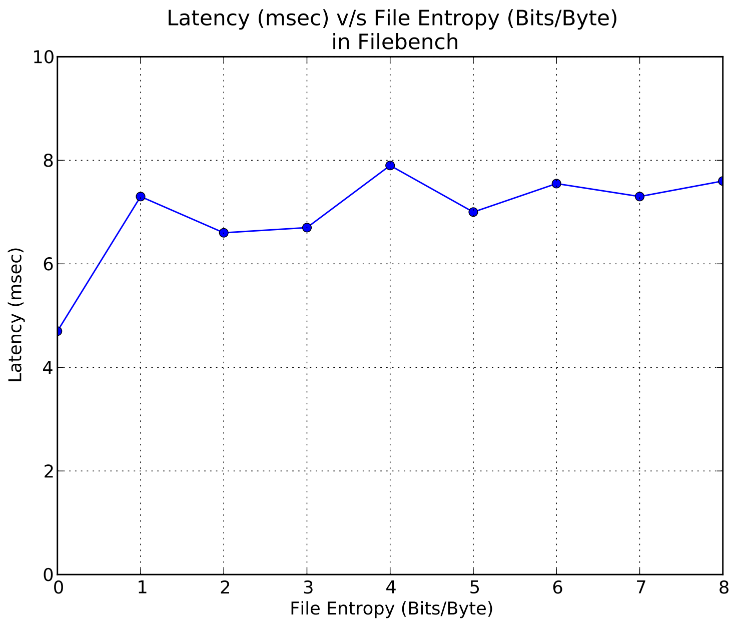

Figure 5.8: The latency over all the read runs

The read workload preallocates around 50 GiB for each run and this takes a considerable amount of time. For this reason we could not manage to run multiple rounds of operations and then compute the average readings. We managed to run a sequence of reads with entropies 0 through 8 once, and once again from 5 through 7. Figures 5.7 and 5.8 show that the trends in the read workload closely follow the those of the write workload as the entropy increases.

5.2 Lookup fill method results

The following results were generated by using `entropy_lookup_fill`[2] method.

[2]section 3.2

5.2.1 Writes: Bandwidth v/s Entropy

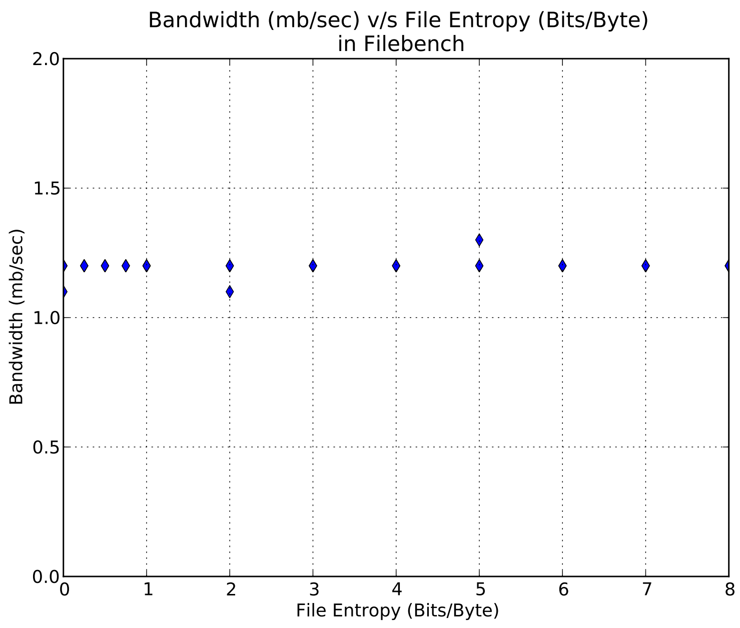

Figure 5.9: The bandwidth of all the write runs

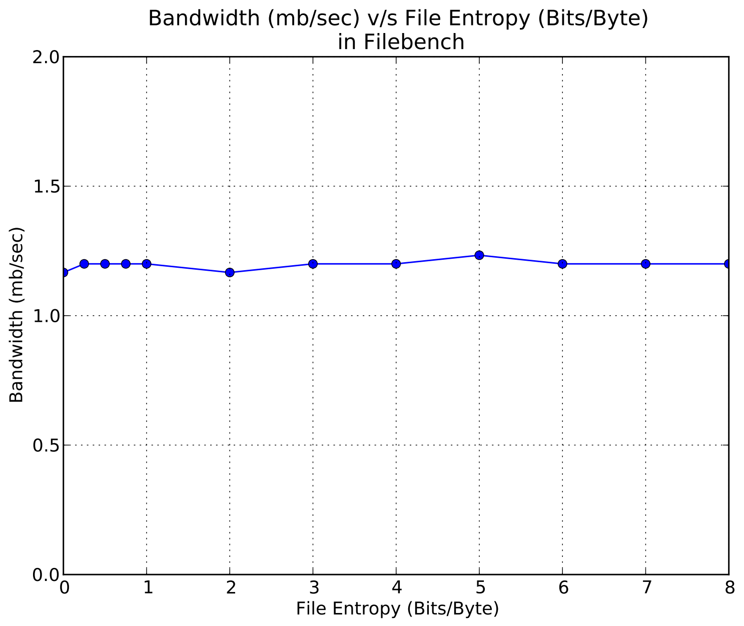

Figure 5.10: The average bandwidth over all the different write runs

Figure 5.9 and 5.10 show that the bandwidth of the write operation dropped dramatically to low values compared to the values using `entropy_cont_fill`. This behavior can be explained based on the idea that the new method takes 3 times the time required by the first method to generate the data which makes the disk idle most of the time, so the average bandwidth achieved is so low. Moreover, the lookup generates random data with homogeneous entropy over the file which makes it so hard for the SDFS to find deplicates chunks in the fileset.

5.2.2 Writes: Latency v/s Entropy

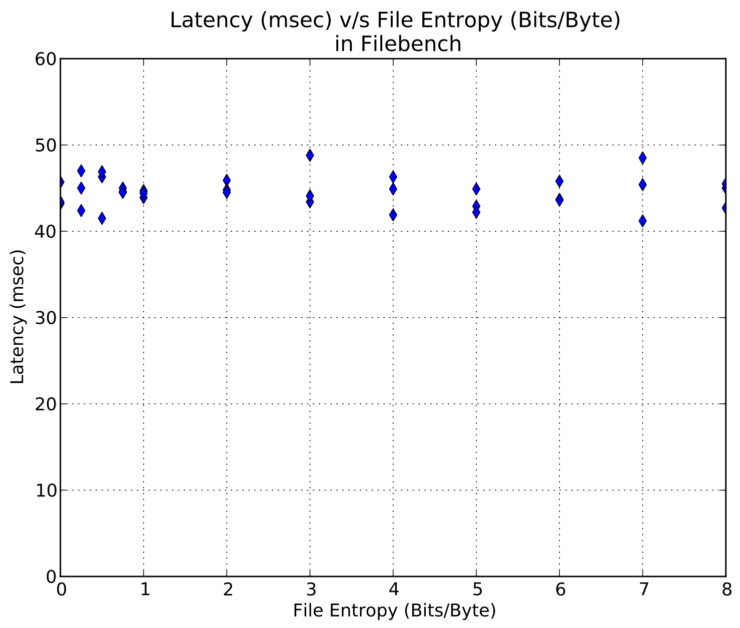

Figure 5.11: The latency of all the different write runs

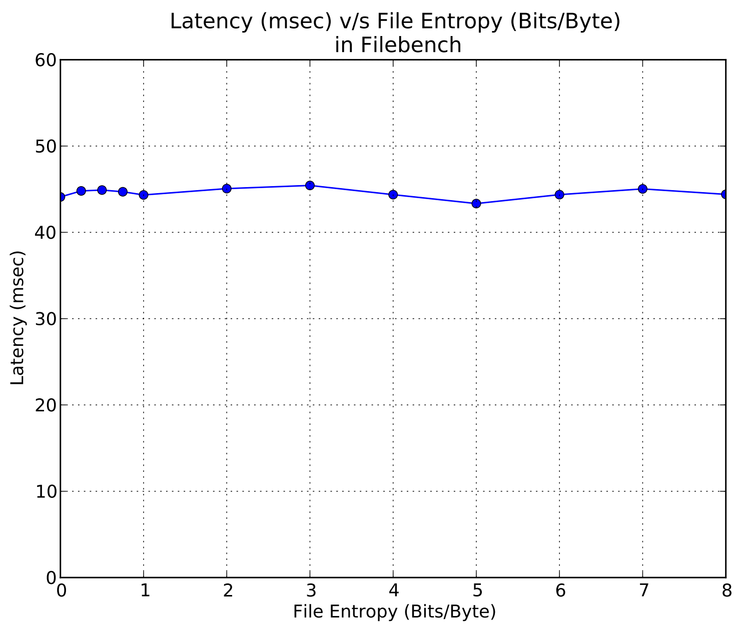

Figure 5.12: The average latency over all the write runs

The results in these graphs are not as expected. Based on the premise that the newer method of entropy generation is slower, we expected the latency values to be higher than those achieved in the previous method. However, we do not have a convincing argument to explain these results and hence we cannot hypothesize about the reasons for these and for now, we leave them for future investigation.

5.2.3 Writes: Operations/sec v/s Entropy

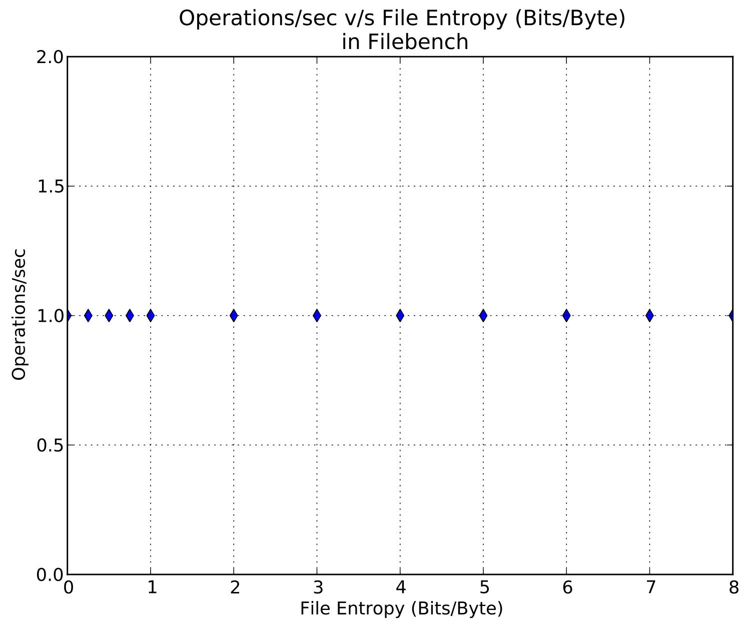

Figure 5.13: The operations execution-speed of all the write runs

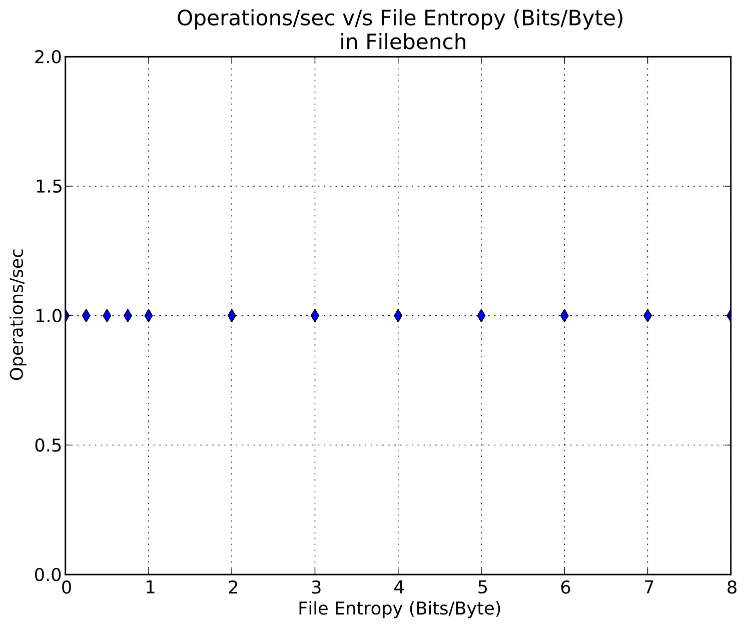

Figure 5.14: The average operations execution-speed over all the write runs

The results achieved here are similar to Bandwidth and can explained based on the same ideas.

5.2.4 Writes: Compression v/s Entropy

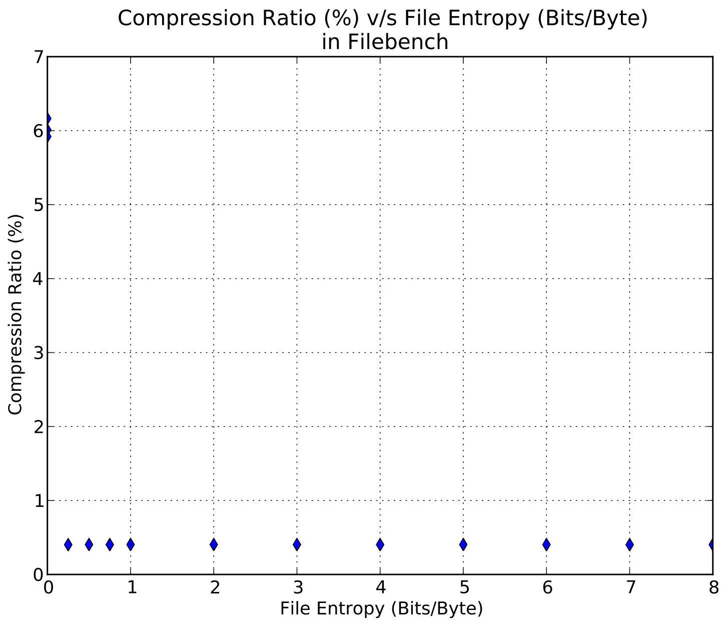

Figure 5.15: The compression ratio of all the write runs

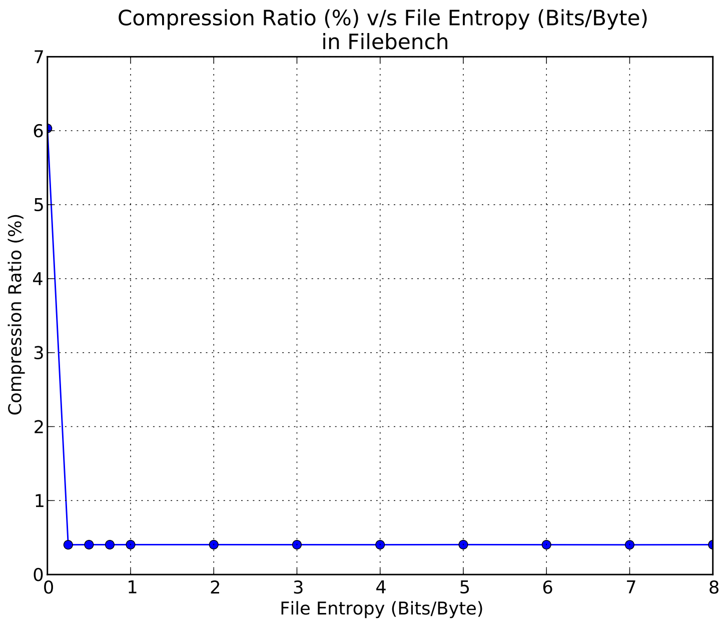

Figure 5.16: The compression ratio average over all the write runs

In Figures 5.15 and 5.16 we see that as entropy is increased, compression levels decrease. As entropy is increased, the number of duplicate chunks on the disk become fewer and hence deduplication is not as effective anymore.

Chapter 6

Conclusions

We successfully implemented a mechansim in Filebench to incorporate entropy based data generation that can take arbitrary values that the user can specify in the range 0.0 to 8.0. We designed our implementation in such a way as to accomodate any future ways of specifying data and in doing so, made it easily extensible. We provide 5 ways of generating entropy, and each one differs in the amount of randomness that is generated. We provide a choice of using any of these methods to better benchmark deduplicating filesystems.

We showed that using entropy-based data generation, various metrics of a deduplicating filesystem can be studied better against different entropy values of the data that is actually written to or read from such a filesystem.

Chapter 7

Future Work

We realized a tradeoff between generating random data and performing read/write operations. While generating data takes a lot of time, except for the contiguous method explained in section 3.2, this makes the processor lags behind the disk, which keeps the disk idle. However, this should not affect the results when we compare the readings obtained from different levels of an entropy used with a deduplication file system. However, testing other non-deduplication filesystems with entropy generation enabled will cause unexpected results because of the way the statistics are collected in Filebench.
Filebench calculates the averages bandwidth and operations per second. Therefore, if the entropy is enabled while running filebench on ext2 for example, most of the time the disk will be idle which will result in lower readings than the case if entropy is switched off, although ext2 is not aware of the data that is being written the previous scenario
To fix this, Filebench has to be modified to calculate the time the disk is actually used and use only that time to calculate the average bandwidth and operations per second.
Another point to mention is the model we are using to model practical loads. We are using entropy as the only characteristics of the file that is actually varied. Moreover, it is the same value across all the files. In normal loads like the ones in cloud computing servers where there is a lot of backups and snapshots, the entropy can high per file but the redundancy is also high. Using true randomization makes it so hard to generate any chunk of the file twice. Maybe it is better to generate a more realistic model by profiling large amount of storage and construct the PDF from that data. We can use such PDF to to calculate the probability that a page will be redundant to one already have been written on the disk.

Bibliography

[1] Open Solaris Community. Filebench. `http://www.solarisinternals.com/wiki/index.php/FileBench`, December 2010.

[2] Petros Efstathopoulos and Fanglu Guo. Rethinking deduplication scalability. *HotStorage '10 Proceedings*, 2010.

[3] Opendedup. Sdfs architecture. `http://www.opendedup.org/`, December 2010.

[4] Wikipedia. Filesystem in userspace. `http://en.wikipedia.org/wiki/Filesystem_in_Userspace/`, December 2010.