DEDUPLICATION AND COMPRESSION BENCHMARKING SUPPORT IN Filebench

Operating Systems CSE506 Final Project Report

By
Rami Al-Rfou'
Nikhil Partwardhan
Phanindra

December 10, 2010

Contents

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

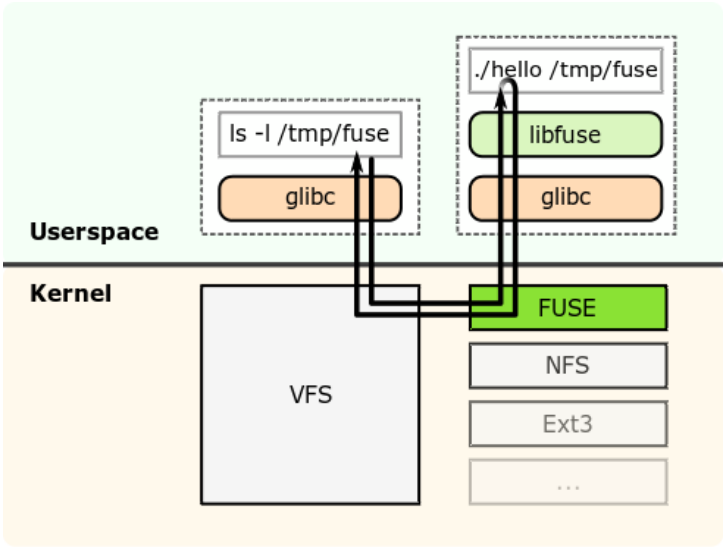

Figure 1.1: Userspace file system architecture in linux[4]

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.

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 1.2, 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 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

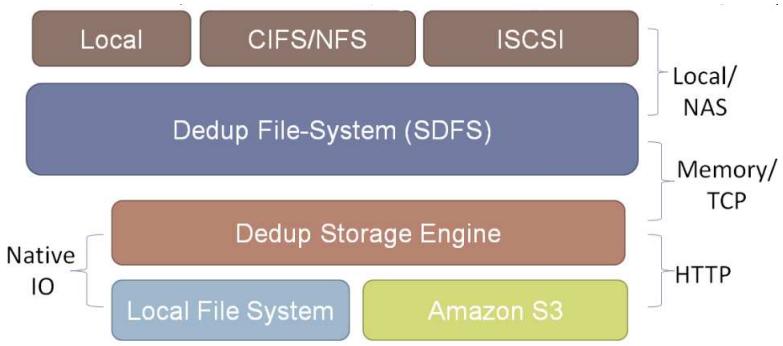

Figure 1.2: SDFS Architecture[3]

VFS to support network and local operations.

Chapter 2

Design

In our design we tries the following principles:

- **Minimal change**

  We tried our best to make the scope of the changes as minimal as possible. This applies to the size of the patch counted by number of lines and the number of files modified.

  The files that are modified

  - fileset.c
  - flowop_library.c
  - parser_lex.l
  - parser_gram.y

- **Extensibility**

- **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 that enables the user to switch the functionality on or off at the compiling 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

2.1 Filebench Interpreter

The changes in the Filebench Interpreter have been designed 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,path=\$dir,...,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. 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.

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.

2.2 Filebench workflow

2.3 Entropy Generator

2.3.1 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.

2.3.2 Random generator

In 2.3.1 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 ε
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 3.3 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.

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:

```
1 #include "utils.h"
  #include "parser_gram.h"
  ..
  %%
  ..
6 datasource                { return FSA_DSRC;}
  entropy                   { return FSA_ENTROPY;}
  ..
  <INITIAL>[0-9]*\.[0-9]+   {  .. } // parse decimal values.
  ..
11 %%
  ..
```

`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.

```
  ..
3 %%
  ..
  //define fileset command supports only source_type and not entropy

  files_define_command: FSC_DEFINE FSE_FILE { .. }
8 | 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.

```
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 Filebench workflow

3.3 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.

```
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.

3.3.1 Calculating the pdf

We explained in section 2.3.1 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.

3.3.2 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 2.3.2 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.

Chapter 4

Evaluation

Chapter 5

Results

Chapter 6

Conclusions

Chapter 7

Future Work

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.