Deduplication and compression benchmarking support in Filebench

Submitted by: **Nikhil Patwardhan, Rami Al-Rfou', Phanindra Bhagavatula**
{npatwardhan, ralrfou, pbhagavatula}@cs.stonybrook.edu

November 14, 2010

1 Project Goal

Data deduplication and compression techniques are widely used to improve storage system performance and space requirements. Most of the popular file system benchmarks, however, generate data in a very simplistic way: files are simply filled with fixed (hard-coded) byte patterns.

Such an approach cannot provide the realistic benchmarking of the deduplication systems. In this project we modify the Filebench file-system benchmarking suite so that it can benchmark deduplication and compression systems. Currently, Filebench writes `0x00` bytes when any write occurs. The entropy (a.k.a. "randomness") of such data stream is `0`, since the same character appears throughout the data.

The objective of the project is to implement a data stream source that generates data with a predefined entropy.

2 Theoretical Background

If X is a random event with N possible states $X_1, X_2, ..., X_n$ and probability of each state is $P(X_1)$, $P(X_2)$, ..., $P(X_n)$, then the entropy of X is defined as:

$$H(X) = -\sum_{i=1}^{n} p(X_i) \times \log_2 p(X_i) \tag{1}$$

In this project we assume that the event is an emission of a new byte by a data source. Since each byte [8 bits/byte] can have 256 states, according to the equation above, the range of entropy is from 0 to 8 bits/byte. The more the value of the entropy, the more random the data is, which means it is less predictable.

The entropy of a typical English text is about 0.6 - 1.5 bits/character. By picking the distribution with which the data source generates bytes, we can produce data streams with various entropy levels. This allows us to control the quality of the data and accordingly use it while benchmarking filesystems.

3 Existing Implementation

The user first creates a profile in Filebench where he specifies various general parameters for running the benchmark like the execution time, location of the report, etc. The user also specifies what different types of operations need to be executed like creation of files, reading files, appending to files in a flow language file saved with the extension `.f`.

Depending on such a flow file scenario, Filebench runs its benchmark tests and displays a report to the user. The specification of these settings is read by a implemented in Filebench using a parser that reads the various options and accordingly decodes the action to be taken.

4 High Level Design

We will add another option to the file and filesets definitions for specifying entropy so that the user has more fine grained control over the kind of data that is generated while running the benchmarks. This will require us to make changes in the parser so that Filebench can understand what needs to be done and invokes our code with the parameter values that the user will specify for this new feature in Filebench.

Currently, a lot of the implementation related code in Filebench exists in the `flowop_library.c` file. It lists out different kinds of operations that can be executed. Some examples are:

1. write/writewholefile
2. read/readwholefile
3. createfile/appendfile
4. openfile/closefile

When writing to file(s), the following function is invoked:
`static int flowoplib_iobufsetup(threadflow_t *threadflow, flowop_t *flowop, caddr_t *iobufp,fbint_t iosize)`. This function allocates memory for a buffer to be written out to a file. We will add functionality to handle user-specified entropy levels and use those to generate random data patterns whose entropy will match the user-specified entropy levels.

5 Design Challenge

There is a mathematical limitation on the maximum amount of entropy that can be generated in data of a certain size [buffer]. This is dictated by the size of the buffer. For example, if the size of the buffer is less than 256 Bytes, it will be impossible to have entropy of 8 bits/Byte. Hence, given any buffer size, we have to find the maximum possible entropy for it and check it against the user-specified value of entropy. If it is greater than the maximum possible value, the algorithm applied for populating the buffer with data will have to reject the specified value and instead generate the maximum possible entropy following best effort delivery strategy.

Also, the size of the buffer decides the resolution of the entropy that can be achieved after populating it with data that follows a certain Probability Density Function(PDF). For example, it is not possible to achieve an exact entropy value of 1.23456789 using a buffer that is only 8 Bytes. Instead, it may only be possible to generate entropy having the value 1.2346, thus sacrificing some resolution.

6 Plan of Action

6.1 Step 1

Instead of selecting a number of distributions and calculating entropy for them, we choose to not pick the distribution and instead generate it according to specification (entropy).
Given a specific entropy we will implement a generator that will create on the fly a discrete probability distribution that generates data with the given entropy. The discrete probability distribution will have an entropy that is equal to the requested entropy within 0.01 bits/byte resolution. However, this computed entropy will be subject to the Design Challenge mentioned above. Note that even if the actual resolution of entropy in the buffer does not match the requested level, the buffer will still be filled. However, the user will be notified by a warning message.

6.2 Step 2

The second step is to program the generator and incorporate it in the Filebench as explained in the High Level Design section above. This will involve modifying the parser and adding C code to generate distributions of data described in Step 1 and integrate this functionality into the existing Filebench implementation.

6.3 Step 3

The third step is to benchmark SDFS file system with modified Filebench. (SDFS is an open source deduplication file system). We will use SDFS to test the efficiency of the Deduplication file systems. SDFS is an open source project that is maintained by openDedup (http://www.opendedup.org/). SDFS is a cross platform file system that runs as a user space file system in Linux.

We will deploy SDFS on a Linux machine and not inside a virtual machine, to avoid any virtualization effects and to be close as much as possible to the real usage scenarios.

The requirements of deploying the SDFS file system are the following:

- JAVA 7
- 64 bit Linux distribution
- Fuse 2.8+
- 2 GB of RAM

We are planning to use our laptops that already have Ubuntu 10.10 64-bit with Java 7 installed. Ubuntu 10.10 has fuse 2.8.4 by default. Each machine has 2 GB of memory. We will test the SDFS by writing files with different sizes and entropies on a partition mounted by SDFS and measuring the actual capacity used to store those files. We will study the effect of entropy on SDFS performance and plot graphs of entropy versus deduplication efficiency.

7 References

1. Filebench - `http://www.solarisinternals.com/wiki/index.php/FileBench`
2. SDFS - `http://www.opendedup.org/`
3. Entropy - `http://en.wikipedia.org/wiki/Entropy`