Single File Compression Through Successive Bit Shifted XOR Functions

Rosie Bartlett

Abstract Compression is undoubtedly an exceptionally important part of data storage. Throughout this paper I will be exploring a new compression method that utilises XOR functions and bit shifting to compress data. This method also gives rise to compound compression, resulting in incresed efficiency.
Key words Compression; Compund compression.

1 Preface

To preface this paper, I would like to first explain the notation used throughout this paper. The method being discussed throught this paper is refered to as SBX, for successive bit shifted XOR compression. SBX uses bit shifting to compress correction keys into pairs of shorter keys and bit shift lists, or BSLs which are simply lists of integers. Since there are multiple BSLs, the BSL related to slice n will be refered to as s_n, where the i^{th} value, starting from 1, is refered to as $s_{n,i}$. the same naming method also applies to keys, with k_n being the key associated with slice n. One of the measures used is in relation to the length of all of the BSLs for a given file. In this instance, s refers to all BSLs for the given file.

The term *slice* is used extensively throughout the paper. A *slice* refers to a section of the file being compressed. This is not equivalent to a *file*. A *file* is comprised of *slices*. The term *pair* is used extensively throughout and refers to a BSL calculation class and key class, and not the actual values contained within the classes. For mathematical notation, see table 1.

2 Compression

SBX uses a parallel iterative process to compress single files. This process (see figure 1) is only run once per file and returns the optimal BSLs, keys, and F_{0c} for the respective file. This process is split into two parts, preprocessing and file compression. The file compression method is outlined in figure 1. Please note however, that each node is representative of an iterative loop, that when ended returns to the previous node. For the child BSL iteration nodes however, once these have ended, they return to the master pair key iteration node.

Symbol	Description	Type
s_n	Bit shift list for file n	List of integers
k_n	The associated key for file n	Bit string
$D(k_n, s_n)$	Decoding function. See equation (3)	Function
Ξ	Bitwise XOR shorthand. See equation (4)	Operation
$<<$	Bit shift left	Operation
I_n	Correction key. See equation (2)	Bit string
F_n	File n, where n$\neq$ 0	Bit string
F_{0C}	Calculated F_0 for a given key and BSL pairing. This may be an incorrect F_0	Bit string
F_0	Compressed file	Bit string
$L(s)$	Number of items in all bit shift strings	Integer

TABLE 1. *Notation used throughout this paper, in order of occurrence*

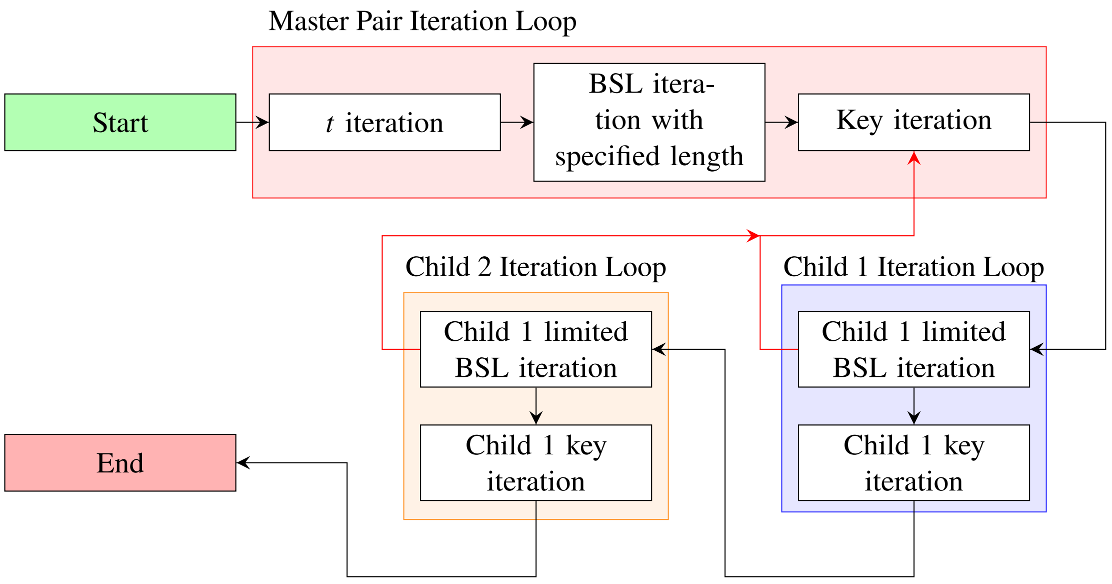

FIGURE 1. *File compression method for three files excluding preprocessing*

2.1 Proprocessing

Before the file can be considered, a small amount of preprocessing must be performed. This preprocessing consists of populating two empty lists, which will be referred to as s_0 and t, as well as initialising classes. The first list to be populated, s_0, is an exhaustive two dimensional list of all the possible BSL values. When this list is being populated, each item added is an exhaustive list of all possible BSL values of a specific length, with this length starting at one and ending with a length of l. The second list, t, is all the different combinations of the lengths of the different BSL values for all the slices. This is ordered in ascending order by considered the sum of each list value within t.

For example, when n is 2 and l is 3, s_0 would be:
`[[[0], [1], [2]], [[0, 1], [0, 2], [1, 2]], [[0, 1, 2]]];`
and t would be:
`[[1, 1], [2, 1], [1, 2], [3, 1], [2, 2], [1, 3], [3, 2], [2, 3], [3, 3]].`

Once this has been completed, the file can now be considered. This consists of the file being converted into slices[1], and then each slice being allocated to classes that all hold their respective slice. The first class is different to the others, as it is a master pair class instance which controls all of the compression. Each class contains more data needed for the compression such as the key length, and references to the master pair node class instance for the child nodes, and a list of references to the child nodes for the master pair node.

2.2 File Compression

Once the preprocessing has finished, then the compression can begin. The compression is controlled by the master pair node, and begins begins by iterating over each of the items in t. For each of these, it iterates over each of the possible BSL values with a length specified by the first item in the current iteration of t. For each of these iterations, every possible value for the key is iterated over and F_{0c} is calculated, see equation (1), using the current value of the BSL and the current value of the key.

$$F_{0c} = F_1 \oplus D(k_1 \oplus s_1) \tag{1}$$

$$I_n = F_n \oplus F_{0c} \tag{2}$$

Once F_{0c} has been calculated, the master pair begins iterating over each of the child nodes. For each child node iteration, a small amount of preprocessing must be performed first to enable checks later on. This preprocessing contains calculating I_n, and then the offset of the first and last non-zero bit in I_n. Once these have been calculated, the child begins iterating over all possible BSL values with a length specified in the current iteration of t in the master class. The index of the child within the master pair class' list of children is the index used in retrieving the current BSL length to iterate over, with the first child accessing the second item in the current iteration of t, the first value having already been used by the master pair class.

$$D(k_n, s_n) = \mathop{\Xi}_{i=1} (k_n << s_{n,i}) \tag{3}$$

$$\mathop{\Xi}_{i=1}^{m} (x_i) = x_1 \oplus x_2 \oplus \cdots \oplus x_m \tag{4}$$

It is here that the first efficiency check is performed. The first value in the current BSL value has to be less than or equal to the index of the first non-zero bit in I_n, and the last value in the current

[1]Each slice is the same length, and padding is added if required to allow this

BSL added to the length of the key must be larger than or equal to the index of the last non-zero bit in I_n. This check ensures the the current BSL is able to produce a non-zero bit in the lowest and highest index required. If this check is passed, then the child node then attempts to calculate a key that, when paired with the respective BSL and then passed into equation (3), would result in I_n.

To attempt to calculate the required key for the current BSL, the process first begins by considering the first bit in I_n. This bit is then saved as the current calculated key, and the precess moves onto the next bit. From this point onwards the calculation becomes slightly more complicated. After the first bit, the process first calculates $D(k_n, s_n) \oplus I_n$, and then considers the current bit being calculated. This bit is then added to the end of the previously saved key. This is a bitwise process that relies upon the previous bits having been calculated so that there is a maximum of one unknown bit that can be calculated. This process does include checks to improve efficiency, such as only using values in the BSL that could affect the current bit being calculated.

However, this calculation method is not perfect and errors can be introduced throughout. Because of this, once the key calculation has been performed, the result of the calculation is passed into the key expansion method, D, along with the respective BSL, the result of which is compared against I_n. If these are identical then the child returns true to the master pair node, which then continues iterating over the remaining children. If none of the child BSL values could be used, then a new key is generated and the process begins from the first child. Once all children have successfully been iterated over, the master pair and all child nodes save their values and the iterative process is ended. This can be done due to the sorting of t in the preprocessing. Since t was sorted by the sum of each of the elements, then once all children have successfully been iterated over, then there cannot be a lower values of $L(s)$ for the current set of files, and so iteration is ended.

2.2.1 Example of SBX Compression of a Correction Key

Given F_{0c}=`10110101` and F_n=`11001000`, therefore the long key must be `01111101`. However, this long key can be compressed, and represented using the key `11001` and the BSL `[0, 2]`. To expand the key and BSL pair into the long, or intermediate, key, which is XOR-ed with the master file to produce the file related with this specific BSL and key pair, the BSL is used one item at a time in the following manner once the long key has been initialised to 0. First the key is bit shifted left by the current value in the BSL. This is then XOR-ed with the long key and replaces the previously stored long key value. Using the example previously given, this would result in `11001` being first bit shifted by 0 to the left resulting in `11001`. This is then XOR-ed with the initialised long key value of 0, resulting in the long key now having a value of `11001`. The next value in the BSL is 2, so the key is bit shifted left by 2 resulting in `1100100`. This is then XOR-ed with the long key resulting in `01111101`, which is the desired long key for this F_0 and F_n.

3 Expansion

Due to the simplicity of SBX, file expansion is an exceptionally quick process. The expansion begins by parsing the compressed file and splitting this into the relevant sections detailed in section 5.

Once this has been completed, the program now has access to F_0, k, and s, and can therefore expand the file. This expansion, using equation (5), is fairly simple and does not require any complex and time consuming calculation, thus making expansion exceptionally fast. The speed of expansion has been quantified in section 6.

$$F_n = F_0 \oplus D(k_n, s_n) \tag{5}$$

4 Compund Compression

Since SBX can compress any size of data, it only makes sense that to improve the efficiency of the compression, the keys usd withing SBX would be compressed themself, thus giving rise to compund compression. Compund compression is only applied to the keys of a compressed file.

5 Compressed File Data Layout

	Bit offset	Length	Description
🟥	0	4	Length of each file passed into the compression function, $\frac{l}{8}$, in bytes
🟦	4	1	Boolean representing a nested compression of the key
🟦	5	$l+1$	Key k_n
🟦	$6+l$	2	Length of s_n, $L(s_n)$
🟦	$8+l$	$L(s_n)\log_2 l$	BSL s_n
🟧	-4	4	Padding to remove

TABLE 2. *Bit offsets and their related description within an SBX file*

The data layout of SBX compressed files has been constructed to alow for a maximum efficiency in compression, but this does come at a slight cost. SBX can only compress data that is stored in bytes, and thus cannot handle data that includes a non integer value of bytes. This would seem to contradict some of the resuts in section 6, however the results given are only for compression and do not include data storage.

In table 2, there are two sections, indicated by the two different colours. The first section, indicated with red, is a header section that contains the value of $\frac{l}{8}$ used when the file was compressed; or alternatively the number of bytes in each of the compressed files. This section is included at the start of all compressed sections, not just at the start of the compressed file. The second section, indicated in blue, is a repeatable section containing data on a single compressed file. The first bit is a boolean that, if true, indicates that the key of this compressed file has itself been compressed. The next part is the key which may itself need to be expanded. The length of the key in table 2 is given as $l+1$, however this may not be true. The key will have a length of $l+1$ only if the key has not been compressed, otherwise the key will have a different length.

After the key is the length of the BSL related to the current section, followed by the BSL itself. This section is repeatable. After all of the compressed file sections have been parsed, this leaves the final 4 bits, which represent the padding added to the first compressed file to allow for the most efficient compression. This is removed from the first file before all the compressed files are concatenated together. The offset for the padding to remove section is a negative since this cannot be given as a general formula for all compressed files, and as such has been given relative to the end of the file.

6 Results

To evaluate the effectiveness of SBX, it was run on several different sets of data, each set having a different batch and file size. The code was run on a three node raspberry pi cluster, appendix time me thinks for detailed specs, and times for each compression have been included. Each set of data contained 1000 pseudo-randomly generated files. The generated files were not individually generated, rather a single integer was generated from 0 to nl inclusively, which was then sliced into n seperate files.

7 Future Work

Whilst SBX can be efficient for large files, it is still exceptionally slow in comparison to common compression systems, such as ZIP files, which will require further development to achieve similar compression times. I am continually developing SBX to improve efficiency, and as sure that with further developemnt, SBX will become singificantly more efficient

Appendix

A Pseudo Code

This appendix is to give a general overview of the code used for SBX in the form of pseudo code. For the source code of SBX, a link is provided in the footer on the first page of the paper.

B Raspberry Pi Cluster Specifications

This appendix details the specifications of the Raspberry Pi cluster that SBX was tested upon.

- **Master Node**:
  - Raspberry Pi model B
  - Something else?

Author

Rosie Bartlett is a currently a student and is still developing SBX. She can be reached at abc@xyz.edu. GitHub is also a good way to keep up with the current version of SBX.

Acknowledgements

I would like to thank the developers of MPI for Python [1] which was used extensively throughout this project. I would also like to thank the peoples who helped me with writing this paper and their advice given in relation to the subject. Merci beaucoup Sophie.

References

[1] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. "MPI for Python". In: *Journal of Parallel and Distributed Computing* 65.9 (2005), pp. 1108–1115. ISSN: 0743-7315. DOI: `https://doi.org/10.1016/j.jpdc.2005.03.010`. URL: `https://www.sciencedirect.com/science/article/pii/S0743731505000560`.

Current version: V0.1(March 13, 2021); Date accepted: MMMM DD, YYYY.