UNIVERSITAT POLITÈCNICA DE CATALUNYA

FAPEC integration as an HDF5 filter

by

Sergi Dueñas Pedrosa

in the
Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona
Departament de Física

Advisor: Enrique Garcia-Berro Montilla
Supervisor: Marcial Clotet Altarriba
Co-Supervisor: Jordi Portell i de Mora

July 2016

Acknowledgements

Curiosament, les primeres línies que es llegeixen en aquest projecte, són les últimes que s'han escrit. I també les més fàcils i alhora les més difícils.

Són les més fàcils perquè aquestes línies estan dedicades a les persones que sempre he tingut a prop en el transcurs d'aquest projecte que es diu vida, i les més difícils perquè no es pot expressar amb paraules el que sento cap a elles.

Aquestes línies estan dedicades al meu pare i a la meva mare. Al meu *mein*. Al meu iaio i els seus enginyosos acudits i a la meva iaia i els seus dinars dels divendres. A la meva tia i els seus "achaques". I per últim a l'Andrea, la persona amb la qual vull viure mil i una aventures i a la que estimo amb bogeria.

Sóc qui sóc gràcies a tots vosaltres. Us estimo moltíssim a tots.

També vull agrair a l'Enrique, el Marcial i el Jordi la manera tan magistral en com m'han anat guiant en el transcurs del meu projecte final de carrera. Són tres persones que tant de bo tothom tingues l'oportunitat de conèixer i treballar amb elles.

Aquest projecte també és per a vosaltres.

Sergi

Contents

List of Figures

List of Tables

Acronyms

AO Astro Observation.

API Application Program Interface.

ASTER Advanced Spaceborne Thermal Emission and Reflection Radiometer.

AVX2 Advanced Vector Extensions 2.

BSC Barcelona Supercomputing Center.

CCSDS Consultative Committee for Space Data Systems.

CUs Coordination Units.

DICOM Digital Imaging and Communication in Medicine standard.

DPAC Data Analysis and Processing Consortium.

DPCB Data Processing Center of Barcelona.

DPCs Data Processing Centers.

EDAC Error Detection and Correction.

ESA European Space Agency.

FAPEC Fully Adaptive Prediction Error Coder.

FITS Flexible Image Transport System.

GBIN Gaia Binary File Format.

HDF Hierarchical Data Format.

HDF4 Hierarchical Data Format version 4.

HDF5 Hierarchical Data Format version 5.

IDU Intermediate Data Updating.

IEEC Institute for Space Studies of Catalonia.

NASA National Aeronautics and Space Administration.

PEC Prediction Error Coder.

SIMD Single Instruction, Multiple Data.

SKA Square Kilometre Array.

SSE2 Streaming Single Instruction Multiple Data Extensions 2.

UB Universitat de Barcelona.

UPC Universitat Politècnica de Catalunya.

Chapter 1

Introduction

By definition, supercomputers are hardware and software computing systems that provide a sustained performance close to the best one achievable nowadays. Clearly, the performance of these systems is much better than that of home and office systems. Supercomputers and supercomputing centers are used to solve very complex problems, including simulations of physical phenomena such as weather predictions, supernova explosions, genome sequencing, astronomical observations or the data packets traffic in mobile networks. All these processes and simulations require a large amount of data to correctly model the problem. A clear example is the European Space Agency (ESA) Gaia mission, which will generate over 100 Terabytes of raw data by the end of its five years of nominal duration. Given these huge data volumes, one of the first problems is the storage and data management. Moreover, database systems or similar solutions are required in order to keep data organized. Data organization and indexing is extremely important to allow efficienty access, data transfers among computing elements or organizations (either inside or outside the computing premises) and to ensure data integrity. Finally, backups are obviously mandatory for such large projects, so the volume of data can be easily doubled or tripled.

1.1 Background

Gaia is an ambitious astrometric space mission integrated within the scientific program of ESA in October 2000 and launched in December 2013. It measures with very high accuracy the position and velocities of a large number of stars, and galactic and extragalactic objects. For each object observation (also called *transit*) the brightness, color and position of the object are recorded. Gaia will map more than 1 billion stars, which

means that data processing centres will have a huge amount of complex data to store and process.

Gaia data storage and processing constitutes a challenging task in terms of efficiency, effort and computing power. For this reason, a large team of experts including scientists, engineers and software developers has been set up, constituting the Data Analysis and Processing Consortium (DPAC). It is responsible for the data processing and the elaboration of the final Gaia cataloge. This consortium is divided into nine Coordination Units (CUs), each of them specialized on a given set of data processing tasks, as well as in six Data Processing Centers (DPCs). CU3 is the unit in charge of the development and implementation of the core data processing pipelines, which process the raw telemetry data coming from Gaia. Within CU3 there are different systems with specific tasks. One of these systems is the Intermediate Data Updating (IDU), a cyclic system that processes the entire set of raw data using the updated calibrations obtained so far during the mission. IDU will provide higher coherence between all the scientific results and will correct any errors or bad data interpretations from previous iterations. IDU is executed in the Data Processing Center of Barcelona (DPCB) using the MareNostrum supercomputer of the Barcelona Supercomputing Center (BSC).

The files used to run IDU in MareNostrum are serialized Java objects compressed with `zip`. This format is called Gaia Binary File Format (GBIN). However, a transition from the GBIN format to Hierarchical Data Format version 5 (HDF5) is being studied to increase the processing and computing efficiency, thus optimizing the usage of MareNostrum. The HDF5 library implements some compression algorithms such as `zlib` or `szip` as *filters*, but users can register their own compression algorithm as a third-party filer and use it in their systems. Taking advantage of this feature, the main goal of this project is to integrate a high-performance compression algorithm, the Fully Adaptive Prediction Error Coder (FAPEC) into HDF5 to use it as a compression filter.

FAPEC is a compression algorithm that was especially designed for Gaia, so the computational efficiency, the data storage management and data transmission performance could be considerably improved if this integration is done. However, not only the Gaia mission could benefit from this work. Many other organizations or work groups use HDF5 for different purposes. In particular, HDF5 has been successfully used in the field of geophysics, in remote sensing applications, in the field of Finantial Engineering, or in oceanography, to put just a few examples. All of them could make use of FAPEC once integrated as an HDF5 filter.

1.2 State of the art

HDF5 is not the only data management suite available for scientific or engineering projects. A first example is ROOT. It is a modular scientific software framework that provides all the functionalities needed to deal with big data processing, statistical analysis, visualization and storage. Users can store their data in a compressed binary form in a ROOT file. These files are self-descriptive and data is organized in a tree structure, similarly to HDF5. Moreover, powerful mathematical and statistical tools are provided to operate on the data, and a powerful C++ application and parallel processing is available for any kind of data manipulation.

Another file format to store data is Flexible Image Transport System (FITS), which is the most common format used in astronomy. FITS includes many tools to describe photometric and spatial calibration information, together with image origin metadata. This file format also supports a large variety of programming languages such as C, C++, Fortran or Java.

Finally, the Common Data Format (CDF) is another example of data storage and management libraries. This solution is an interface for storage and manipulation of multi-dimensional data sets.

Regarding high-performance data compressors, there is a large variety of algorithms such as `LZO`, `bzip2`, `blosc`, `snappy`, `MAFISC`, `LZ4`, `LPC-Rice`, `zlib` or `szip`. All of them can be used as a compression filter in HDF5. Each has its strengths and weaknesses, so for illustrative purposes we will select and evaluate only a few of them in this work.

1.3 Motivation

In this work we propose a solution to some of the problems found in supercomputing environments by combining an extremely efficient, standard, open-source data manager suite with a high-performance data compressor. We do not intend to use such an efficient file format and, later, compress the resulting files or data sets without further ado, as we would be losing in the compression process the file format benefits. Our aim is to compress the little portions of data that conform the data sets stored inside the file (which are named *chunks*), thus without losing any of the functionalities offered by the mentioned data management suite.

HDF5 is our choice for the data storage and management format, and FAPEC is the high-performance data compressor chosen. By integrating FAPEC as an HDF5 filter we

will offer a solution that can solve in a smart, clean and efficient way the storage and management problems in supercomputing environments.

It is worth mentioning that a first tentative of integrating the Prediction Error Coder (PEC) into HDF5 exists [1], but in that case the solution was not actually a filter but an adhoc application just for the Gaia case. Furthermore, PEC is a *static* data compressor, whereas in the present work we intend to use the fully adaptive version (FAPEC).

1.4 Structure and plan of this project

The final goal of this project is to integrate FAPEC as an HDF5 filter. To do this, we will first study the HDF5 Application Program Interface (API), FAPEC in itself, and design the software implementation. We will analyze the performance of the solution and compare it with the alternatives available in HDF5, such as `szip` or `gzip`, as well as third-party filters such as `bzip2` or `blosc`.

The project is structured as follows. First, in Chapter 2 an introduction to HDF5 and its main concepts and benefits is given, as well as a description of the *filters* concept. Chapter 3 introduces FAPEC, its main features and the API that allows its use by other software. It follows Chapter 4 where we explain in detail how FAPEC has been integrated as an HDF5 filter, the feasibility study that has been done, and how the code has been structured. Chapter 5 presents the tests done and discusses the results obtained. Finally, Chapter 6 presents the main conclusions obtained in the course of this work and proposes some improvements that could be implemented in the future.

Chapter 2

The HDF5 file format

2.1 Overview

Hierarchical Data Format (HDF) technologies consist of two data storage and management formats, Hierarchical Data Format version 4 (HDF4) and HDF5, and their associated libraries. The HDF Group also provides tools, such as a data browser or editor, as well as command line tools. Both HDF4 and HDF5 were designed to be adaptable to virtually any scientific or engineering application. Another of the advantages of this file format is the availability of an API in different programming languages. Such compatibility allows HDF to work across a large variety of computational systems. Actually, HDF technologies are intended to be used in super-computing environments.

HDF5 allows to express very diverse and large amounts of data in a natural manner. In contrast to databases that work with tables, HDF5 supports n-dimensional datasets, and each element in the dataset may be a complex object (such as an image, an array of integers, strings, or floating point numbers). This fact allows users to develop powerful data processing systems that work on HDF-stored data, either with sequential or parallel access. This might be an advantage in some cases as filed-matching queries are not straightforward on traditional databases. Currently, the HDF format is used by many scientific and engineering communities like atmospheric physics, or Astrophysics [2], to put just two examples.

2.2 HDF5

HDF5 is a suite that allows the management of extremely large and complex data collections. It addresses important deficiencies of HDF4 taking advantage of the latest

computing advances, to deliver users and developers an extra efficient data processing and management system [3]. Abstractly, an HDF5 file is a container of an organized collection of objects. These objects are the following ones. First, Groups and Datasets are used for the data in itself. Then, Dataspaces, Datatypes and Attributes, are employed for the description of these data or as meta-data. Lastly, Property lists and Links describe the way objects are created and related with each other. A deeper view of Datasets and Datatypes is presented in Sects. 2.4 and 2.5.

Owing to its powerful API [4], developers can implement their own HDF5 solutions for their projects. The HDF5 API provides resources to work with all HDF5 objects and their properties. An example of this is the H5D API that allows developers to deal with datasets, or the H5G API that allows to deal with groups. Moreover, HDF5 also implements a high-level API to ease common HDF5 operations when working with images or tables, among others. As previously mentioned, HDF5 has C, C#, C++, Fortran and Java API.

One of the most powerful features of HDF5 is that it includes powerful storage arrangement and processing options such as chunking and compression. These allow very efficient read and write data routines, such as writing or reading just a portion of a very large dataset, and parallel I/O operations over the same file.

Lastly, regarding size limits, the HDF5 format has no practical limit on the size of its files. The library allows up to 32-dimension dataspaces (that defines the size of a dataset), with each dimension able to allocate up to the limit of an unsigned 64-bit value [5]. Most likely, the maximum file size limitation will be imposed by the maximum file size that can be allocated in a single file.

2.3 File structure

An HDF5 file can be easily compared to a Unix file system. It has a hierarchical structure, at the top of which we find the *root* group, represented by the "/" symbol. This group is the parent or ancestor of the rest of the objects that conform the file. An HDF5 file will always have at least one object, that is, the root group.

The objects generated are linked to their associated parent. It is important to note that the objects themselves have no name, but instead names are provided through links. As in an UNIX file system, */foo* represents an object (named *foo*), which is linked directly to the root group. Objects have a unique object identifier, but because of the possible file structure, an object can be named in as many ways as paths to the object exist. Figs. 2.1 and 2.2 show examples of possible structures in an HDF5 file.

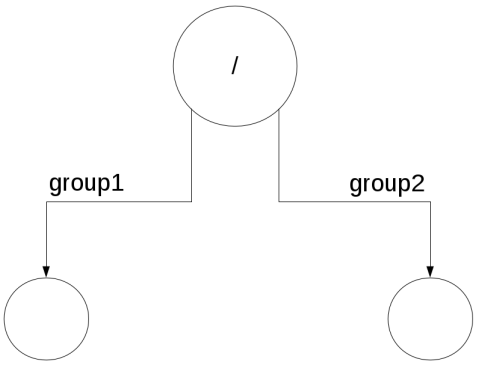

FIGURE 2.1: Possible structure of an HDF5 file. The descendant objects of the root group have */group1* and */group2* names.

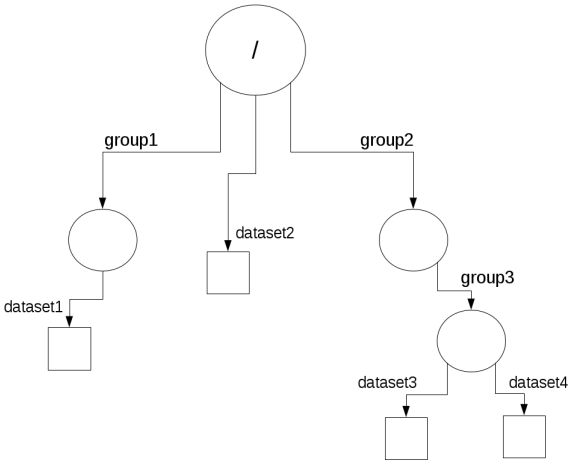

FIGURE 2.2: A more complex structure of an HDF5 file. *Group1*, *dataset2* and *group2* are members of the root group. *Dataset1* is member of *group1*. *Dataset3* and *dataset4* are members of *group 3*, which in turn is member of *group2*.

2.4 Datasets

An HDF5 dataset is an object composed of a collection of data elements or raw data. Data is stored as one-dimensional or multi-dimensional arrays of elements, the characteristics of which is described by the dataspace. A data element stored into the dataset is a single unit of data, which is a set of bits with a certain layout. This layout is described by the datatype. Data in different datasets may have different datatypes (this is, different layouts) such as integer numbers, floating point numbers, strings, arrays, references or even compound elements. However, data in a given dataset must always have the same datatype.

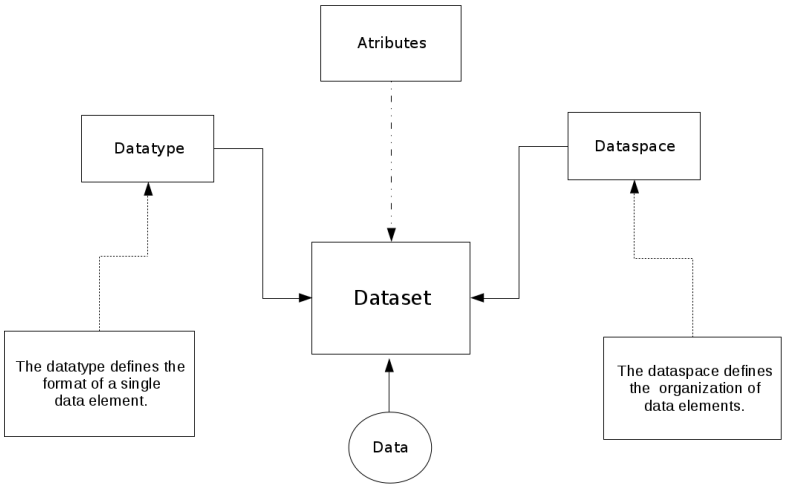

FIGURE 2.3: Application view of a dataset. A dataset contains some given data with the type specified by the datatype. A dataset may have user-defined attributes.

Datatype objects are closely linked to a dataset. When a dataset is created, the datatype is set and it cannot be changed for the lifetime of the dataset. For more information about datatypes, see Sect. 2.5. When a dataset is generated, a list of creation properties is set, which includes storage properties such as chunking and a list of filter to be applied (which might include compression). Like the datatype the properties list cannot be changed once the dataset is created. This properties are stored as metadata within the HDF5 file.

HDF5 allows users to store custom metadata within the same file. This metadata is managed through *attributes*. An attribute can have any datatype that the user defines, and thus allows complex data structure, and can be attached to any HDF5 object. Fig. 2.3 shows a possible schematic view of a dataset and its elements.

A dataset object is stored in a file in two parts: a header and a data array. The header contains information that is needed to interpret the array portion of the dataset, as well as meta-data that describes or annotates the dataset. Header information includes the name of the object, the dimensions, the datatype, information about how the data in itself is stored on disk, and other information used by the library to speed up access to the dataset or maintain the integrity of the file. For instance, HDF5 automatically stores the filters and the order in which they have been applied when writing the data to a given dataset. In this way, the library will transparently process the stored data with the same filters in the inverse order when reading.

2.5 Datatypes

Datatype objects implement a mechanism to specify the storage layout of data elements, which determines how to interpret the elements, and allows to convert data from different compatible layouts. The idea behind datatype objects is similar to Objected Oriented classes. A generic datatype class with some properties and parameters is given, but its determination into specific values is equivalent to instantiating a datatype object. Once the dataset is created, the datatype object is set and linked to the dataset and cannot be changed during the life of the dataset.

A datatype class is defined as a set of one or more datatype properties. The datatype properties are defined by the logical model of the datatype class. For example, the integer class has properties such as "signed or unsigned", "length" and "byte-order". Table 2.1 presents the properties of each datatype class.

When a read or write operation is performed, the HDF5 library must know the exact datatype that describes the layout of the data. The library provides the *NATIVE* types, which are mapped to the corresponding datatype of each platform. That means, for example, that the integer type of any platform such as Linux will be mapped to `H5T_NATIVE_INT`, where H5T refers to the API portion to deal with datatypes. These platform dependent types are used to read or write data from memory of the host system. There are as well *fixed* datatypes, as `H5T_IEEE_F64LE` which corresponds to an eight-byte, little-endian, IEEE floating-point.

2.6 Filters

When a dataset is generated, the user can specify if data has to be processed by filters prior to being written or after being read from the filesystem. These filters are added to the *data pipeline* or data flow. The HDF5 standard library implements many of them, such as data compression, shuffling or error detection filters. Additional user-defined filters can also be used. In this section we describe how the data flows through the HDF5 library in order to understand how filters are applied.

2.6.1 The data pipeline

When data is written or read from an HDF5 file, the HDF5 library passes the data through a sequence of steps. Each of these steps processes the data according to the meta-data located at the header of the dataset when this was created. This data processing can be, for example, a simple byte swapping (to adjust the data to a Little

Class	Description	Properties	Notes
Integer	Twos complement integers	Size (bytes), precision (bits), pad, byte order, signed/unsigned	
Float	Floating Point numbers	Size (bytes), precision (bits), offset (bits), pad, byte order, sign position, exponent position, exponent size (bits), exponent sign, exponent bias, mantissa position, mantissa size (bits), mantissa sign, mantissa normalization, internal padding	See IEEE 754 for a definition of these properties. These properties can describe non-IEEE 754 floating point formats as well.
Character	Array of 1-byte character encoding	Size (characters), Character set, byte order, pad/no pad, pad character	Currently, ASCII and UTF-8 are supported.
Bitfield	String of bits	Size (bytes), precision (bits), offset (bits), pad, byte order	A sequence of bit values packed into one or more bytes.
Opaque	Uninterpreted data	Size (bytes), precision (bits), offset (bits), pad, byte order, tag	A sequence of bytes, stored and retrieved as a block. The "tag' is a string that can be used to label the value.
Enumeration	A list of discrete values, with symbolic names in the form of strings.	Number of elements, element names, element values	Enumeration is a list of pairs (name, value). The name is a string; the value is an unsigned integer.
Reference	Reference to object or region within the HDF5 file		See the Reference API, H5R
Array	Array (1-4 dimensions) of data elements	Number of dimensions, dimension sizes, base datatype	The array is accessed atomically: no selection or sub-setting.
Variable-length	A variable-length 1-dimensional array of data elements	Current size, base type	
Compound	A Datatype of a sequence of Datatypes	Number of members, member names, member types, member offset, member class, member size, byte order	

Table 2.1: Datatype classes and their properties, extracted from the HDF5 User's Guide [6].

Layers	Description
I/O initiation	Initiation of HDF5 I/O activities (H5Dwrite and H5Dread) in a user application program.
Memory hyperslab operation	Data is scattered to (for reading), or gathered from (for writing) the application memory buffer (bypassed if no datatype conversion is needed).
Datatype conversion	Datatype is converted if it is different between memory and storage (bypassed if no datatype conversion is needed).
File hyperslab operation	Data is gathered from (for reading), or scattered to (for writing) file space in memory (bypassed if no datatype conversion is needed).
Filter pipeline	Data is processed by filters when it passes. Data can be modified and restored here (bypassed if no datatype conversion is needed, no filter is enabled, or dataset is not chunked).
Virtual File Layer	Facilitate easy plug-in file drivers such as MPIO or POSIX I/O.
Actual I/O	Actual file driver used by the library such as MPIO or STDIO.

TABLE 2.2: Stages of the data pipeline, extracted from the HDF5 User's Guide [6].

Endian or Big Endian system), alignment, scatter-gather or hyperslab selections (which consist of selecting a subset of a dataset, for example if this is too large).

The HDF5 library determines automatically which operations are needed on the data buffers. Each operation transforms its input buffer, writes the transformed data into an output buffer, and passes the output buffer to the next processing stage. Table 2.2 lists the stages of the Data Pipeline.

To apply any filter to the data such as the mentioned byte swapping or, for example, a high performance compressor, the HDF5 library must know about the filter to be able to add it into the data pipeline. The process of registering a filter is explained with full detail in Section 2.6.2.

2.6.2 Registering a third-party filter

The integration of a user-defined filter is quite simple. First, the HDF5 file must be created. After that, the filter has to be *registered*. Finally, the last step is to create the dataset into the HDF5 file with the properties specifying that the registered filter shall be applied. Once this is done, the library will automatically call the filter for each data chunk and the filter output data will be stored into the actual file. Once all data is written, the dataset and the file have to be closed. After the file creation and write operations, the file and the dataset can be re-opened in order to read and display the

data stored into the dataset. To do so, the program that performs the read operation must have the filter registered as well. Again, once the file and the dataset have been used they must be closed properly.

With this procedure in mind, the first step is to define the filter function. Although it could be defined in the main program, defining it in a separate file and compiling it as a shared library makes the code more modular, clean and elegant. The filter definition prototype is defined by the HDF5 API, with the following prototype:

```
/* Declaration of the filter function */
size_t my_filter_function(unsigned int flags, size_t cd_nelmts,
          const unsigned int cd_values[], size_t nbytes,
          size_t *buf_size, void **buf);
```

Note that the interface deals with one single data pointer (`**buf`), which means that we must use it for both receiving and returning the data. Once the filter is defined, which should return a value of 0 in case it fails, it has to be registered into the main program so that the HDF5 library knows that it can be called. The way to do this operation is invoking the H5Z API, which deals with filter operations:

```
/* Register the filter */
herr_t H5Zregister(const H5Z_class_t *filter_class)
```

Finally, since filters in HDF5 only work with chunks, both chunking and the desired filters must be set up in the dataset creation properties list. This process is done by means of the H5P API, which deals with property lists:

```
/* Create the dataset creation property list */
hid_t H5Pcreate(hid_t cls_id)

/* Set up chunking to the properties list */
herr_t H5Pset_chunk(hid_t plist, int ndims, const hsize_t * dim)

/* Set the filter to the properties list */
herr_t H5Pset_filter(hid_t plist_id, H5Z_filter_t filter_id,
        unsigned int flags, size_t cd_nelmts,
        const unsigned int cd_values[])
```

Applications that want to read and display the data that has been processed by the filter and has been stored into the HDF5 file, such as for example HDF5 tools, have to be recompiled linking the filter library in the process. This will allow such applications to access the contents processed by the custom filter. Otherwise, an error message will appear.

Before attempting the integration of FAPEC (see Chapter 3 below) as an HDF5 filter, a test has been performed to demonstrate how a third-party filter can be used with an HDF5 file. This test has also been used to create the base of the adaptation layer that allows the integration of FAPEC in the HDF5 API. The filter used in this demonstration test follows the described procedure. Specifically, it implements an example proposed at the HDF5 web page [7], which simply adds a checksum to the data by using the `MD5()` function of the OpenSSL cryptographic libary [8].

2.7 Available compression filters in HDF5

A large variety of compression filters can be integrated into HDF5 files. Such filters allow to reduce the size of the datasets and contribute to improve the efficiency of the storage and data transfers. Sect. 2.6 explains how filters work in HDF5. To evaluate and compare the performance of FAPEC as an HDF5 filter, we have considered some of the compression algorithms that can be used as an HDF5 filter. These are `deflate`, `szip`, `bzip2` and `blosc`. Other filters available in HDF5 include `LZO`, `LZF`, `LZ4`, `snappy` and `LPC-Rice` [9].

2.7.1 Deflate or gzip

Gzip is a data compressor relatively popular that uses the Zlib library as a compression/decompression algorithm. The algorithms were written by Jean-Loup Gially and Mark Adler [10] and is fully integrated by default into the with HDF5 Library and it offers nine compression levels. It is worth mentioning that deflate never expands the data [11].

2.7.2 Szip

Szip is a stand-alone library that can be set as a filter in HDF5. It uses a maximum blocksize of 4.1 MB, which means that the compressor works better with larger files [12]. Szip implements a quick algorithm that achieves good compression ratios. It is able to adapt rapidly to the statistical variations of the data to be compressed. Actually, Szip is an implementation of the extended Rice lossless compression algorithm [13, 14] which the Consultative Committee for Space Data Systems (CCSDS) adopted more than a decade ago as an international standard for space applications. As it will be seen later, it means that this is probably the algorithm that is more comparable to FAPEC. In the

case of Szip, it has been adopted by the NASA Earth Observatory System (EOS) to compress the data generated [15].

Table 2.3 illustrates a comparison between different compression techniques including Szip algorithm. The table is extracted from the tests performed [15] with the HDF4 Szip integration. Szip also has been integrated and is being distributed with HDF5 Library since Release 1.6.0.

Technique	RLE	Adaptative Huffman	Szip	Gzip
Compression Ratio	1,60	2,28	2,80	2,37
Compression Time (sec)	85,7	558,4	71,6	273,1
Decompression Time (sec)	41,6	574,9	63,6	68,3

TABLE 2.3: Lossless compression results using various compression techniques. Tested on Pentium II 300Mhz processor.Table extracted from Earth Science Technology Conference [15].

2.7.3 Bzip2

Bzip2 library was written by Julian Seward, and it compresses the data using the Burrows-Wheeler block-sorting text compression algorithm and Huffman coding [16]. Bzip2 is usually more effective than Deflate in terms of compression ratio, but it performs slower in terms of time. However, decompression is relatively fast.

2.7.4 Blosc

Blosc was written by Francesc Alted Resumé. It is a high performance compressor optimized for binary data. It has been designed to compress data extremely fast, at the expense of achieving lesser compression ratios [17, 18].

Chapter 3

FAPEC

3.1 Overview

The Data Compression Group of the Institute for Space Studies of Catalonia (IEEC) has developed FAPEC, the Fully Adaptive Prediction Error Coder. It is currently being commercialized by DAPCOM Data Services S.L., a spin-off company participated by the Universitat Politècnica de Catalunya (UPC) and the Universitat de Barcelona (UB).

FAPEC is a high performance data compressor, which can be applied as a compression solution for satellite payloads owing to its resiliency in front of data outliers and its autonomous operation. Moreover, FAPEC is prepared to operate in a large variety of on-ground engineering and scientific projects, be these related to radiotelescopes, geophysical probes, radars, genomics, text from web or huge log files, images and video including High-Definition, or medical projects working with the Digital Imaging and Communication in Medicine standard (DICOM) format [19].

Having in mind the benefits of using HDF5 in a computational environment as a very efficient data storing and managing library, and the advantages of using FAPEC in any kind of environment and data type, the integration of FAPEC as an HDF5 is a logical step. This is the main motivation of this project. The aim is to achieve an excellent performance both in compression ratio and I/O speed.

3.2 PEC and its Fully Adaptive layer

PEC is a fast and noise-resilient semi-adaptive entropy coder. It offers a high performance in presence of noise or when the input data contains a substantial fraction of outliers, while requiring very few processing resources. PEC offers an excellent coding

efficiency for a wide range of data statistics. In order to obtain the best performance, an adequate pre-processing stage should be used for this coder [20]. Moreover, PEC requires some calibration parameters in order to reach optimum ratios. However, in some cases, even with a simple fixed calibration, it can provide still good compression ratios. In spite of the assets previously mentioned, PEC must be trained for each particular case to get the optimum performance. This is due to the fact that significant variations in statistics of the data to be compressed can lead to a rapid decrease in the compression ratio. To solve this, the Fully Adaptive implementation of PEC (FAPEC) was devised. FAPEC adds an adaptive layer in order to configure the PEC coding options according to the data statistics of each data block. This powerful adaptive layer allows PEC to achieve nearly-optimal ratios. With FAPEC, prior knowledge of the statistics of the data to be compressed is not needed anymore.

Fig. 3.1 shows a performance comparison between PEC, FAPEC and the current recommendation for lossless data compression in space (CCSDS 121.0) which is based on Rice codes. The Shannon Limit, or the maximum theoretical compression ratio, is also represented.

3.3 Configuration and modes

FAPEC is distributed as a software library and as a command-line data compressor. If we call `fapec` with the `-h` switch we get additional information on the compression options, shown in Fig. 3.2 to give an idea on the capabilities and configurability of FAPEC. In our case we will have to configure it through the adequate API functions.

In the case of command-line execution, FAPEC can be invoked without any option, leading to an automatic configuration. However, in the routines that we will use for its integration in HDF5 the adequate configuration options must be passed. Most of these are optional, but there is a specific option that is mandatory, namely, the data type. This is related to the HDF5 data types described in Sect. 2.5, and in the command-line case of FAPEC it is indicated with the `-dtype <t>` switch. This option only supports from 4 to 28 bits per sample, but then we can go beyond with *interleaving* (indicated with the `-il <s>` switch in the command-line case). For example, if a Dataset has a 64-bit integer datatype, a 16-bit sample with an interleaving of 4 can be indicated. Also, since the current version of FAPEC does not include yet an adequate handling of floating-point data, in those cases we can also improve compression through interleaving. For example, single-precision 32-bit floats can be compressed as 16-bit samples with an interleaving of 2. Depending on the nature of the data another size and interleaving configuration could perform somewhat better.

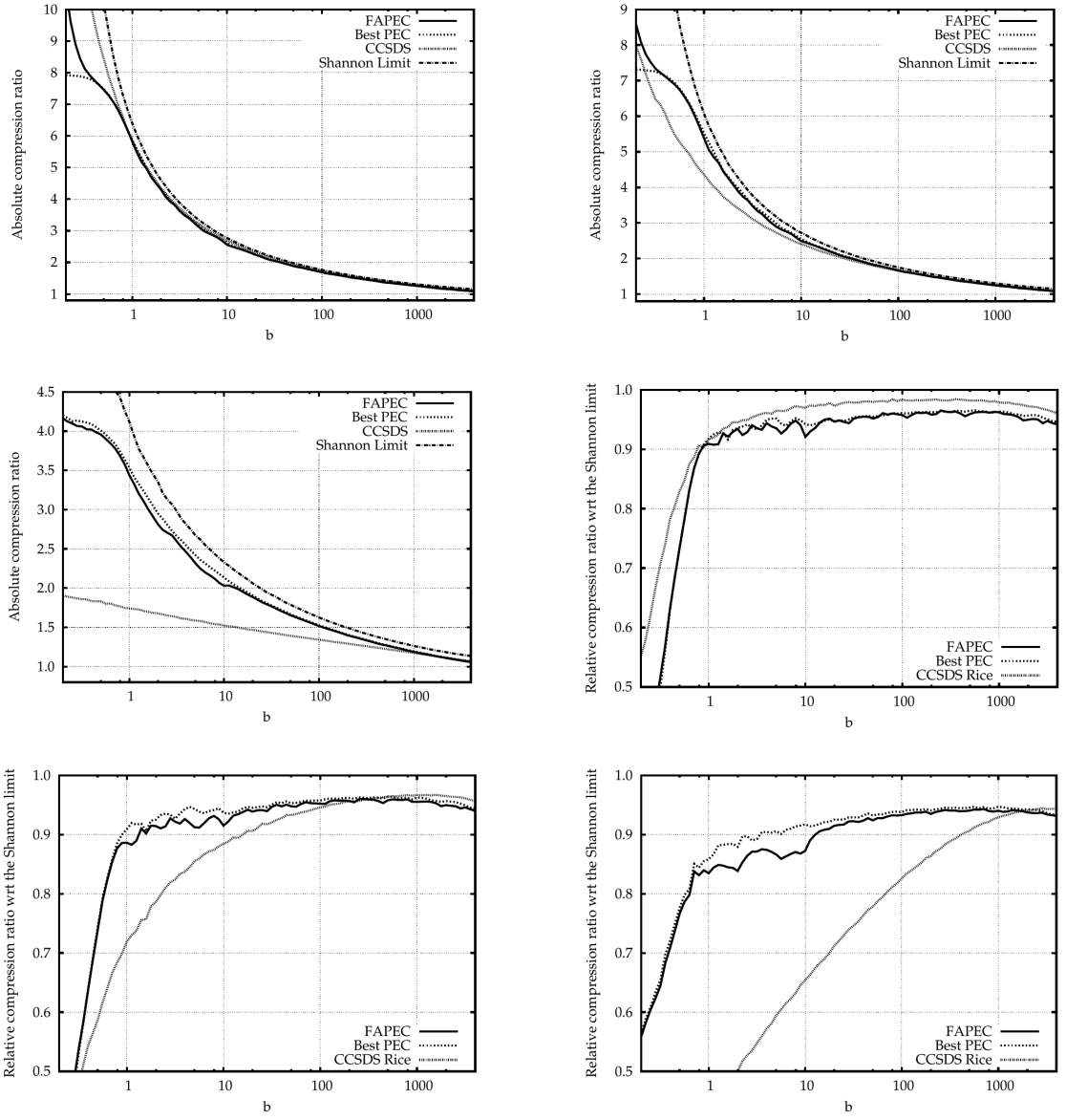

FIGURE 3.1: FAPEC compression ratios for two-sided geometric distributions. This figure has been taken from Ref. [20]. Left panels: absolute compression ratios. Right panels: efficiency of the coders. From top to bottom, tests with 0.1%, 1% and 10% of outliers (flat noise). The ranges of the distributions (values of b) are representative of real applications. A small value of b means a low dispersion of the data (that is, a low entropy) and, hence, an accurate prediction in the pre-processing stage, and vice versa.

Obviously, interleaving can also be used to obtain a better compression ratio in cases where there is some kind of pattern in the data, such as an array of integer values which are more correlated across the columns than along the rows. In such a case, being M the number of columns in the array, an interleaving of M can be used to improve compression.

The case of text data is peculiar, because it has to be indicated with `-dtype txt` in

```
OPTIONS:

 General switches:
  -chunk <b>  Chunk size (default: 4 MB)
              K and M suffixes can be used
  -edac <o>   EDAC option (0-3, default 1)
  -rawout     Do not include compression options in output
              (Warning: it requires manual decompression setup)
 Common compression options:
  -dtype <t>  Data type or symbol size of the data
              4 to 28 bits per sample, or 'txt' for text compression
  -hd <b>     File header to bypass (up to 524294), in bytes
  -chd <b>    Header to bypass in every chunk (up to 524294), in bytes
  -be         Big Endian data (only for 16-bit and 24-bit symbols)
              Default is Little Endian
  -bl <s>     FAPEC adaptiveness block length (32 to 1024 samples)
 Basic or filtered pre-processing:
  -il <s>     Interleaving to be used in pre-processing (up to 32775 samples)
  -od <o>     Pre-processing filter order (default 1, max. 4)
 Prediction-based multi-band image pre-processing:
  -bands <b>  Number of image bands (up to 32771)
 No pre-processing:
  -np <f>     No pre-processing. Data format must be indicated:
              'tc'  typical 'twos-complement'
              'ab'  sign in an additional bit (MSB)
              'us'  unsigned (positive) values
 Simple differential pre-processing with losses:
  -lossy <b>  Number of least-significant bits to remove from each sample
 Image pre-processing algorithms:
  -dwt <width> <height> <bands> <losses>
  -hpa <widht> <height> <bands> <losses> <realBpp> <bil>
              -dwt      Discrete Wavelet Transform algorithm
              -hpa      Hierarchical Pixel Averaging algorithm
              <width>   Image width, up to 8396800 pixels
              <height>  Image height, up to 8396799 pixels
              <bands>   Image bands (colors), up to 32771, 1 for monochrome
              <losses>  0 for lossless, up to 8 for lossy
              <realBpp> Significative number of bits per pixel
              <bil>     1 for Bands-Interleaved-by-Lines, 0 for B-I-by-Pixel

 If '-dtype' is not specified, FAPEC automatically selects the options
 (excluding the lossy and image options)
```

FIGURE 3.2: FAPEC command-line switches.

the command-line case. In the library that is activated with a specific configuration function call. That leads to 8-bit samples which are then processed with a special text compression algorithm.

The last option which is relevant in the context of this project is the `-be` command-line switch (also handled adequately in the API). Most of the Datatypes that are presented in Table 2.1 of Sect. 2.5 indicate a byte order property. This FAPEC switch only affects 16-bit and 24-bit settings, for which FAPEC assumes Little Endian coding by default unless this switch is indicated.

3.4 The FAPEC API

The FAPEC API [21] allows to easily integrate FAPEC in our software. It defines two major configuration sets. The first one is the *User Options* set, encoded into a single 32-bit integer value. This value combines user parametres such as privacy, file handling, console logging, or error detection and correction, among others. The following function allows the users to encode this information into an integer:

```
/* User Options function */
int newFapecUsrOpts(int verbLevel, int askOverwrite, int deleteInput,
int enforcePriv, int streamMode, int keepAttr, int noCompHead,
int edacOpt, int cryptOpt, int threadPool, int decMode);
```

For simplicity, this value can be set to 0 (zero) and it will lead to the default options: no logging, force output file overwrite, no deletion of input file when done, no privacy enforcement, no streaming, generate default compressed header, no Error Detection and Correction (EDAC), no encryption, single thread, and do not store original file attributes.

In a similar way, the user options can be retreived from a given encoded value using the following function:

```
/* Function to retreive User Options */
void getFapecUsrOpts(int fapecUsrOpts, int *verbLevel, int *askOverwrite,
int *deleteInput, int *enforcePriv, int *streamMode, int *keepAttr,
int *noCompHead, int *edacOpt, int *cryptOpt, int *threadPool,
int *decMode);
```

The second configuration set corresponds to the *FAPEC Compression Options*. Here, user can configure the specific parameters of FAPEC such as the symbol size (data type), pre-processing algorithm, adaptative block size, and other. This set of options include the main configuration options described in the previous subsection. All this information is encoded in the `t_fapecOpts` type which can be created as follows:

```
/* Function to create the FAPEC Options variable */
t_fapecOpts* newFapecOpts();
```

It is mandatory to generate this variable before either compression or decompression. Once this is done, it can be directly used as far as we operate at a high enough level, that is, working with files instead of FAPEC *chunks* — which are the basic data block allowed for FAPEC compression. Here the concept of chunk is very similar to that of HDF5. If we operate on files (which is not our case), the default configuration options include auto-configuration and a default chunk size of 1 MB. However, as already said, for a better HDF5 integration we will use the chunk-based compression functions (leaving the file-level handling to {glsHDF5), which means that we must update the `t_fapecOpts` variable with, at least, the data type. The following function allows us setting some common options, such as the chunk size we want (which can be adjusted to the HDF5 chunk size):

```
/* Function to configure the FAPEC Options variable */
int setCommonFapecOpts(t_fapecOpts *fapecOpts, int32_t chunkSize,
```

```
uint32_t headerOffset, uint32_t chunkOffset, uint16_t fapecBlockLength);
```

After this, the user can define a specific compression configuration, which includes the data type and the pre-processing algorithm. There are several algorithms available: basic pre-processing, basic lossy pre-processing, filter pre-processing, text pre-processing, or image pre-processing algorithm. In this work, basic pre-processing compression algorith will be used. The reason for this choice will be explained in Chapter 4. To set up such basic pre-processing compression algorithm, as well as the data type to be used (and any eventual interleaving value), the following function must be used:

```
/* Set basic pre-processing compression algorithm */
int activateBasicFapecOpts(t_fapecOpts *fapecOpts, int8_t symSize,
bool bigEndian, uint16_t interleaving);
```

Once the user has set up the User and FAPEC Options, data can finally be compressed. The FAPEC API provides compression functions to operate at three different levels: chunk-based functions, memory-based functions and file-based. According to Sect. 2.6.2, filters in HDF5 only work whith chunks. Therefore, the chunk-based function of FAPEC will be used in this project:

```
/* Chunk-based compression function */
int fapecChunkCompression(unsigned char **buff, size_t *buffSize,
int userOpts, t_fapecOpts *inputConf);
```

The FAPEC decompression API is equivalent to the compression case. To recover the original data from a compressed chunk the following function must be used:

```
/* Chunk-based decompression function */
int fapecChunkDecompression(unsigned char **buff, size_t *buffSize,
int userOpts, t_fapecOpts *inputConf);
```

Chapter 4

Integration

4.1 Feasibility study

Before starting the integration work in itself, which requires quite some development effort, we want to know if compressing data stored in HDF5 files with FAPEC can really be done, and if so, which improvement can we expect with respect to other compressors. To answer this question, we have carried out a simple feasibility study. The idea is to extract, from an HDF5 file, every single dataset into separate files (using tools available in the HDF5 suite). Then we will compress them with FAPEC and with `gzip` level 1 (fastest), 4 (average/fast) and 9 (best), and compare the size and compression time. Finally, we will reassemble the outputs into a single HDF5 file again (just considering the level 9 gzip in this case), to compare the original HDF5 file size with this reassembled one. As we will see, the results of this feasibility study prove the benefits of FAPEC integration as an HDF5 filter.

4.1.1 Astro Observation File Structure

The HDF5 file selected to do this comparison is an Astro Observation (AO) file from the Gaia space mission [22]. The `AO.h5` file structure is presented in Fig. 4.1. Inside the `AO.h5` file there are five groups (in fact, there are six groups if the *root* group is taken into account): Class0, Class1, Class1T, Class2 and Class2T, containing 3 datasets each (Af, Header and Sm). These correspond to the different *window classes* (of pixels on the Gaia focal plane), window truncation, and data provenance (astrometric field, sky mapper or ancillary data). The data types are arrays of 16-bit unsigned integers for both `Af` and `Sm`, and compound types for `Header`.

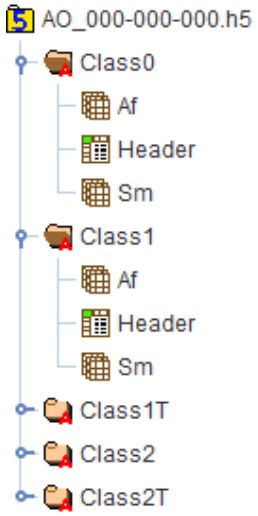

FIGURE 4.1: HDF5 Astro Observation file structure. The hierarchical structure with 5 groups and 3 datasets each group can be seen.

4.1.2 ASCII procedures and results

As a first approach, we have dumped the above mentioned datasets in ASCII format using the `h5dump` command-line tool. Once the datasets have been dumped, they have been compressed with gzip using level 1, 4 and 9. Gzip lower levels offer quick compression, but less compression ratio while higher levels offer the best compression ratio, but higher compression times. These compression levels have been chosen to have a broad view of the different compression ratios and times. Finally, the dumped files also have been compressed with the FAPEC command-line executable program in its auto configuration mode.

In order to simplify the analysis of the results obtained with this procedure, Class1 and Class2 groups have been ignored (we have chosen test files where these groups do not contain any data). Table 4.1 shows the original sizes of the ASCII files extracted, their size once compressed with FAPEC, and their size with gzip levels 1, 4 and 9. The best results in terms of size are highlighted using boldface.

As can be seen, FAPEC does not achieve good compression results when compared with gzip, not even with the lowest gzip level. We should recall that FAPEC is a binary-optimized high-performance compressor, whereas this test has been performed on ASCII (text) files.

	Size (bytes)				
Dataset	Original	FAPEC	Gzip (lev.1)	Gzip (lev.4)	Gzip (lev.9)
Class0/AF	16204992	7656546	5829566	5088653	**4571501**
Class0/Header	846688	133583	115484	97539	**78703**
Class0/Sm	2124654	896534	706014	616063	**543246**
Class1T/AF	8015445	4390097	3366319	2922941	**2670382**
Class1T/Header	5505488	700519	623998	524593	**426959**
Class1T/Sm	3080665	1230628	1018785	891936	**755098**
Class2T/AF	134286752	53943574	44644963	39283718	**32849861**
Class2T/Header	139804160	17191764	15449932	12563645	**10045645**
Class2T/Sm	81035772	23845852	20379312	17131143	**14231032**

TABLE 4.1: Feasibility test results for the ASCII dumps. Since FAPEC is a binary-optimized compressor, the compression sizes are not competitive in front of gzip performance.

4.1.3 Binary procedures and results

As in the previous section, every single dataset has been extracted from the `AO.h5` file, but this time the datasets have been dumped using the binary switch in the command-line `h5dump` tool. As this is a more realistic case, we provide more detailed results including compression times. Tables 4.2, 4.3 and 4.4 show the ratios and times for the several dumped files. This is, the size and compression times of the original dumped files, of gzip-compressed files, and of FAPEC-compressed files. To better illustrate the differences in performance for the several compression solutions, some performance ratios are also shown. Again, the best performances in terms of size (and also time, in this case) are highlighted in boldface.

These three tables show that FAPEC always performs better than gzip when compressing AF datasets. In all cases, FAPEC achieves better compression ratios and also performs a faster compression. FAPEC also produces excellent results when compressing SM datasets. Again, FAPEC always performs better than gzip in terms of compression ratios, and also better than gzip level 9 in terms of compression speed. Nevertheless, gzip level 1 and 4 perform faster than FAPEC, notwithstanding the compression ratios are far from the ones achieved by FAPEC. On the other hand, for Header datasets FAPEC is not the best option. The compression ratios are reasonably good, but they are significantly better with gzip using either of its levels. Also, FAPEC performs slower when processing these files. It is worth mentioning that Header datasets have a particular, very heterogeneous structure with several datatypes such as unsigned 64-bit integers, 8-bit, 16-bit and 32-bit integers, and arrays of 8-bit and 16-bit integers.

Finally, we should also mention that to obtain good results compressing Class2T/Sm dataset, FAPEC autoconfig feature had to be turned off to manually pass the compression parameters. With autoconfig turned on, the current FAPEC release incorrectly

Dataset	Original	Gzip (lev.1)	Gzip (lev.4)	Gzip (lev.9)	FAPEC
			Size (bytes)		
Class0/AF	5372136	3862598	3823948	3801570	**3366311**
Class0/Header	172634	58704	55517	**52201**	65086
Class0/Sm	702240	461424	455444	451258	**406024**
			Compression time (ms)		
Class0/AF	-	365	511	678	**221**
Class0/Header	-	**10**	12	25	162
Class0/Sm	-	**45**	61	126	123
			Time ratio (t/t_{min})*		
Class0/AF	-	1,652	2,312	3,068	**1,000**
Class0/Header	-	**1,000**	1,200	2,500	16,200
Class0/Sm	-	**1,000**	1,356	2,800	2,733
			Compression ratio ($Size_{orig}/Size_{comp}$)**		
Class0/AF	-	1,391	1,405	1,413	**1,596**
Class0/Header	-	2,941	3,110	**3,307**	2,652
Class0/Sm	-	1,522	1,542	1,556	**1,730**
					* Lower is better
					** Higher is better

TABLE 4.2: Class0 group test results. The table shows the comparison of four parameters for the dumped Class0 files. Namely, size in bytes, compression time in ms, a time ratio and a compression ratio.

selects a pre-processing stage based on ASCII. By manually indicating the correct parameters to FAPEC, the compression ratio increases from 1.52 to 2.30. This means that, as otherwise expected, it is better to use a configuration based on the data types to be compressed (in case this can be known) rather than an automatic configuration. This conclusion is important for an adequate integration of FAPEC in HDF5, as it will be shown below.

4.1.4 Reassembling the Astro Observation file

Once the datasets have been compressed using FAPEC and gzip, they have been reassembled into a single HDF5 file again, using the HDF5 command line tool, to compare the overall file size. For this purpose, a HDF5 file has been generated from the separate dataset files compressed with gzip level 9, and another HDF5 file with the datasets compressed using FAPEC. We have chosen level 9 of Gzip because it delivers the best compression ratio compared to the rest of levels. The results are presented in Table 4.5.

As can be seen, the overall compression ratio obtained using FAPEC is 2.41, in contrast to just 2.12 achieved by gzip in its best version (level 9). However, the main advantage of FAPEC is in the processing time to compress the data. Gzip takes almost 30 seconds to compress all datasets. FAPEC has done this operation in less than 10 seconds.

Dataset	Original	Gzip (lev.1)	Gzip (lev.4)	Gzip (lev.9)	FAPEC
			Size (bytes)		
Class1T/AF	2630160	2192647	2186181	2178683	**1934781**
Class1T/Header	1146480	304337	291326	**272610**	309767
Class1T/Sm	1011600	646446	633019	627698	**568492**
			Compression time (ms)		
Class1T/AF	-	198	302	928	**147**
Class1T/Header	-	**35**	49	106	218
Class1T/Sm	-	**64**	87	255	112
			Time ratio (t/t_{min})*		
Class1T/AF	-	1,347	2,054	6,313	**1,000**
Class1T/Header	-	**1,000**	1,400	3,029	6,229
Class1T/Sm	-	**1,000**	1,359	3,984	1,750
			Compression ratio ($Size_{orig}/Size_{comp}$)**		
Class1T/AF	-	1,200	1,203	1,207	**1,359**
Class1T/Header	-	3,767	3,935	**4,206**	3,701
Class1T/Sm	-	1,565	1,598	1,612	**1,779**
					* Lower is better
					** Higher is better

TABLE 4.3: Class1T group test results.

From these results we can conclude that, indeed, FAPEC can not only be integrated in HDF5 to compress its datasets, but also that we can expect a good performance both in compression ratios and times.

4.2 Identification of data formats

To properly integrate FAPEC as an HDF5 filter, one of the main goals of this work consists of identifying the data types of the datasets that can be contained in an HDF5 file. By knowing the data type of a particular dataset, we will be able to adjust the FAPEC compression parameters correctly to obtain the best possible compression performance when invoking it as a filter from the HDF5 Data Pipeline (see Sect. 2.6.1).

The HDF5 Library implements two datatype models: *atomic* datatypes and *composite* datatypes. We have focused the attention on atomic datatypes because they correspond to commonly used storage formats. In addition, composite datatypes are an aggregation of one or more atomic or composite datatypes. According to that explained in Sect. 2.5, a datatype class is defined as a set of one or more datatype properties, so we have taken into account every single combination of these properties to set up a robust relationship between a specific datatype with its defined properties, and the compression parameters that will be set in FAPEC. For instance, a `H5T_STD_I32LE` (standard 32-bit, little endian integer) has to be mapped to different compression parameters than a `H5T_IEEE_F64BE`

Dataset	Original	Gzip (lev.1)	Gzip (lev.4)	Gzip (lev.9)	FAPEC
			Size (bytes)		
Class2T/AF	45385716	29156378	28718163	28503094	**22887364**
Class2T/Header	28922270	7736744	7144833	**6625418**	6802527
Class2T/Sm	26697480	12510143	12070230	12002927	**11605445**
			Compression time (ms)		
Class2T/AF	-	2781	4002	9850	**1201**
Class2T/Header	-	**839**	1297	3486	2154
Class2T/Sm	-	**1188**	1798	14520	5400
			Time ratio (t/t_{min})*		
Class2T/AF	-	2,316	3,332	8,201	**1,000**
Class2T/Header	-	**1,000**	1,546	4,155	2,567
Class2T/Sm	-	**1,000**	1,513	12,222	2,733
			Compression ratio ($Size_{orig}/Size_{comp}$)**		
Class2T/AF	-	1,557	1,580	1,592	**1,983**
Class2T/Header	-	3,738	4,048	**4,365**	4,252
Class2T/Sm	-	2,134	2,212	2,224	**2,300**
					* Lower is better
					** Higher is better

TABLE 4.4: Class2T group test results.

File	Size (bytes)	Ratio ($Size_{orig}/Size_{com}$)*	Time (ms)	Ratio (t/t_{min})**
AO	115848704	1	-	-
AO (Gzip lev.9)	54546775	2,124	29995	3,064
AO (FAPEC)	**47977129**	**2,415**	**9789**	**1**
				* Higher is better
				** Lower is better

TABLE 4.5: Overall file size comparison. FAPEC is 3 times faster than gzip level 9, and it also delivers a higher compression ratio.

(64-bit, big endian IEEE floating point) to obtain in both cases the optimum compression performance. Sect. 3.3 explains the most important FAPEC switches involved in the context of the present project. We have checked which characteristics of an HDF5 dataset can be translated into a FAPEC compression parameter, and found the following ones:

- **Size** in bits of the sample, which affects the datatype and interleaving to be used in FAPEC.
- **Byte order**, which affects the endianess.
- **Text** format case, which then requires the selection of text-based FAPEC pre-processing.

Note that the first two datatype properties correspond to compression parameters that FAPEC expects to receive when basic pre-processing compression algorithm is selected:

```
/* Set basic pre-processing compression algorithm. */
int activateBasicFapecOpts(t_fapecOpts *fapecOpts, int8_t symSize,
bool bigEndian, uint16_t interleaving);
```

That is the main reason why the basic pre-processing algorithm is selected in FAPEC.

In order to determine this datatype information, the HDF5 API offers excellent methods to retrieve it in a clean and simple way. To retrieve the datatype of a dataset we have used the following function:

```
/* Returns the the datatype class identifier */
H5T_class_t H5Tget_class(hid_t dtype_id)
```

On the other hand, to retrieve the sample size of a dataset and the byte order, we have used the following functions:

```
/* Returns the size of the actual datatype in bytes */
size_t H5Tget_size(hid_t dtype_id)

/* Returns the byte order of an atomic datatype */
H5T_order_t H5Tget_order(hid_t dtype_id)
```

The optimum value for the interleaving parameter has been assessed with several tests with different sample sizes and interleaving configurations. The chosen values offer the best performance, but depending of the nature of the data, other values could perform even better. Table 4.6 shows the mapping implemented in our integration layer that allows us to map the datatype properties into compression parameters that FAPEC currently supports. We have indicated the configuration as command-line switches for a simpler description, but the actual invocation and configuration will obviously be done through the adequate API functions and parameters. For example, float 16-bit Little Endian (LE) will be compressed with FAPEC using 8-bit samples and an interleaving of 2 samples.

4.3 Integration approach and description

As described in Sect. 2.6.2, to integrate FAPEC as an HDF5 filter we need to implement a software program (or better said, library) that calls the filter in the precise moment that data is being written into a dataset. That can be considered a *wrapper* of the FAPEC compressor in itself, adapting it to the exact HDF5 API.

FAPEC obviously provides both compression and decompression routines, so we have divided the implementation in two parts: the *Writer* (or compression) and the *Reader*

Class	Size (bits)	Byte Order	FAPEC switch
Integer/Bitfield/Opaque	8	N/A	`-dtype` 8 `-il` 1
	16	LE	`-dtype` 16 `-il` 1
		BE	`-dtype` 16 `-il` 1 `-be`
	32	LE	`-dtype` 16 `-il` 2
		BE	`-dtype` 16 `-il` 2 `-be`
	64	LE	`-dtype` 16 `-il` 4
		BE	`-dtype` 16 `-il` 4 `-be`
Float	16	LE	`-dtype` 8 `-il` 2
		BE	`-dtype` 8 `-il` 2 `-be`
	32	LE	`-dtype` 16 `-il` 2
		BE	`-dtype` 16 `-il` 2 `-be`
	64	LE	`-dtype` 16 `-il` 4
		BE	`-dtype` 16 `-il` 4 `-be`
Char	N/A	N/A	`-dtype` txt
String	N/A	N/A	`-dtype` txt

TABLE 4.6: Mapping between HDF5 datatype properties and FAPEC parameters.

(or decompression). Additionally, some other filters (see Sect. 2.7) have been integrated into our software to be able to compare FAPEC performance against that of other high-performance algorithms.

4.3.1 Writer or Compression

The writer program expects two command-line parameters: an HDF5 input file name, and the name of the filter to be applied to the data. Optionally, an output file name can be indicated. The input file is opened with read-only permissions. After that, the filter registration takes place. Because we need an output file to store the data read from the input file, the program creates a new empty HDF5 file. Finally, the writer traverses the input file in search of all objects in an iterative way. A pseudocode listing of the main part of the writer is shown here:

```
/* Open the input file */
H5Fopen(input_file, H5F_ACC_RDONLY, H5P_DEFAULT);

/* Register the filter */
H5Zregister(FAPEC_FILTER);

/* Create the destination file using default properties */
H5Fcreate(output_file, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

/* Traverse the input file */
H5Lvisit(input_file_id, H5_INDEX_NAME, H5_ITER_NATIVE,
         op_func, (void *) &od);
```

One of the input parameters of the `H5Lvisit()` function is a pointer to another function (the *callback function*) which defines the operations to be done when an HDF5 object is

found. It is worth mentioning that `H5Lvisit()` does not search for objects in a file, but it searches for links. It means that it can discover groups and datasets, but it cannot discover other objects such as attributes, which are inserted as a header information and do not have any link to any object. To simplify the integration of FAPEC in HDF5, only datasets and groups are handled. The prototype of the callback function is defined by the HDF5 API as follows:

```
herr_t (*H5L_iterate_t)(hid_t input_file_id, const char *name,
                        const H5L_info_t *object_info, void *op_data)
```

When the callback function finds a group, it opens the output file previously created and generates a new group inside the output file with the same name as the group found, and finally it closes the objects. A pseudocode of the group handling part of the callback function is listed here:

```
/* Open the output file */
H5Fopen(FILE, H5F_ACC_RDWR, H5P_DEFAULT);

/* Copy the group of the input file to the output file */
H5Gcreate(file_id, name, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);

/* Close the objects */
H5Gclose(group_id);
H5Fclose(file_id);
```

The first step when a dataset is found, as in the case of groups, is to open the output file created before. After that, the input file dataset that has been discovered has to be opened and its creation properties list must be retrieved. Afterwards, the callback function adds the registered filter to this list. The number of dimensions, its size and its data type must be known in order to create a new compressed copy of the current dataset, so the function also retrieves this information from the input dataset. Finally, the callback function reads the data from the input dataset, creates a new datatset to the output file already opened, and writes the data with the filter set up into the output dataset. When done, all objects must be closed. A pseudocode of the dataset handling part of callback function is listed below:

```
/* Open the output file */
H5Fopen(output_file, H5F_ACC_RDWR, H5P_DEFAULT);

/* Open the input file dataset to retrieve data */
H5Dopen(input_file, name, H5P_DEFAULT);

/* Retrieve the dcpl */
H5Dget_create_plist(input_dataset_id);
```

```
/* Set the filter to the dcpl */
H5Pset_filter(p_list, FAPEC_FILTER_ID, H5Z_FLAG_MANDATORY,
              NUM_ELEMENTS, cd_values);

/* Retrive the numer of dimension and its size of the input file */
H5Dget_space(input_dataset_id);
H5Sget_simple_extent_dims(dataspace_id, dims, NULL);

/* Retrieve the dataset type */
H5Dget_type(input_dataset_id);

/* Read the input file data using the default properties */
H5Dread(input_dataset_id, datatype_id, H5S_ALL, H5S_ALL,
        H5P_DEFAULT, rdata[0]);

* Create the new dataset into the output file */
H5Dcreate(output_file, name, datatype_id, dataspace_id,
          H5P_DEFAULT, p_list, H5P_DEFAULT);

/* Write the filtered data into the new dataset */
H5Dwrite(output_dataset_id, datatype_id, H5S_ALL, H5S_ALL,
        H5P_DEFAULT, rdata[0]);

/* Close the objects */
H5Fclose(input_file);
H5Dclose(input_dataset_id);
H5Pclose(p_list);
H5Sclose(dataspace_id);
H5Tclose(datatype_id);
```

Before the writing operation, the HDF5 library sends the data automatically to the filter in order to process it. The filter function prototype is defined by the HDF5 API as follows:

```
/* Declaration of the filter function */
size_t my_filter_function(unsigned int flags, size_t cd_nelmts,
          const unsigned int cd_values[], size_t nbytes,
          size_t *buf_size, void **buf);
```

Our implementation of this function manages all FAPEC configuration processes and ultimately calls FAPEC through its API (see Sect. 3.4). First of all, the FAPEC common options are configured: chunk size, chunk offset, header offset and block length. The chunk and header offsets are set to 0, and a default value of 128 samples is used for the block length. These are the values recommended by the FAPEC API and the User Manual, and confirmed by FAPEC developers. Finally, the chunk size must be set to

the nominal chunk size that can be found in an HDF5 dataset, making sure that it is not smaller than the minimum recommended FAPEC chunk size (which is 1024 bytes). These common options are stored into an array and passed as an argument to FAPEC. The next pseudocode listing shows the way to do that operation:

```
/* Register common options */
setCommonFapecOpts(fapecOpts, chunk_size, header_offset,
                   chunk_offset, block_length);
```

After configuring and registering FAPEC, it is time to configure its actual compression options. The values of these options are the result of implementing Table 4.6, and depending on the data type, sample size, interleaving and byte order, they will have one or another set up. According to the discussion of Sect. 4.2, the basic pre-processing compression algorithm has to be used because its input parameters are the ones that we have retrieved from the HDF5 dataset:

```
/* Configure basic fapec options: size, endianness, interleaving */
activateBasicFapecOpts(fapecOpts, size,
                       endianness, interleaving);
```

An exception to this is the case of text (char or string) datasets, for which we will use the text-based FAPEC pre-processing instead.

Finally, the FAPEC chunk compression function is invoked, passing all the adequate parameters, stored into two variables:

```
/* Call FAPEC compression */
fapecChunkCompression((unsigned char **) buf,
                      &nbytes, fapecUsrOpts, fapecOpts);
```

With this last operation, the HDF5 original chunks become compressed with FAPEC.

We have added some mechanisms to ensure the correct execution of the writer regardless of the complexity of the input HDF5 file. According to what was stated in Sect. 4.2, only atomic datatypes are processed and compressed with FAPEC, but the writer can also identify and handle other situations. The first complex situation occurs when a dataset exists, but does not contain data, or in other words, the storage size of the dataset is equal to zero. In this case, the filter call by HDF5 Library returns an error. Another complex situation occurs when an HDF5 file contains complex datatypes such as `H5T_COMPOUND`, `H5T_ENUM`, `H5T_REFERENCE`, `H5T_TIME` or `H5T_VLEN`, which are not supported in compression yet. Finally, the third complex situation occurs when a dataset does not have a chunked layout. According to that explained in Sect. 2.6.2, filters in HDF5 only work when a dataset implements a chunked layout.

Our writer software can identify these three situations and handle them by simply implementing a bypass. When any of these cases are detected, the writer copies the input dataset into the output file as-is, without any kind of processing. The `H5O` API handles this:

```
/* Copy the input object to the output file */
H5Ocopy(input_location, input_name, output_location, output_name,
        H5P_DEFAULT, H5P_DEFAULT);
```

4.3.2 Reader or Decompression

Decompression is quite similar to the compression case described before. The reader functionality is the same as that of the writer in the sense that the eader expects an HDF5 input file, but with its datasets compressed with FAPEC or other compression techniques that we currently support in the present work. The reader generates a new empty HDF5 file, reads the data of the input file, decompresses the data, and writes it from the input file to the output file.

Some considerations have to be taken into account. The first one is that we do not have to set the FAPEC filter into the new dataset creation properties list. This list is retrieved from the incoming dataset, which already has the filter in it. If we set up again the FAPEC filter to the retrieved creation properties list, we would add twice the filter into the data pipeline causing a bad decompression operation. The second consideration is that instead of calling the FAPEC chunk compression function, we obviously have to call the FAPEC chunk decompression function. Exactly the same common and compression options as in the compression case must be used except the chunk size, which must be set to 0 (as indicated by the FAPEC API), because we cannot know the compressed chunk sizes beforehand in HDF5:

```
/* Register common options */
setCommonFapecOpts(fapecOpts, 0, header_offset,
                   chunk_offset, block_length);
```

Once again, a bypass has been implemented to detect and handle the complex situations explained at the end of Sect. 4.3.1. With these considerations in mind, the compressed HDF5 file is now decompressed.

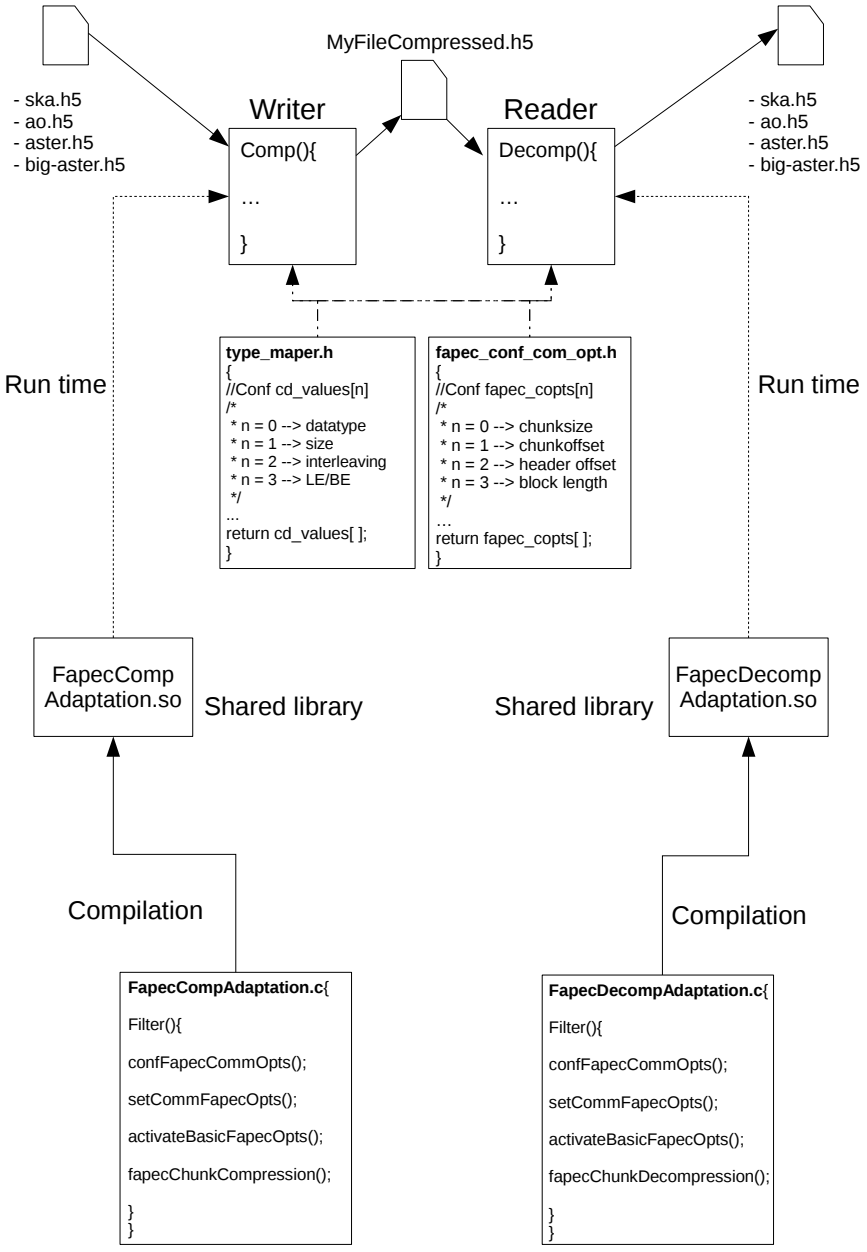

FIGURE 4.2: Overall code structure, illustrating the modular approach taken to optimize compilation time and memory usage.

4.4 Implementation and code structure

In Sect. 4.3 we have explained with some technical detail how FAPEC has been integrated as an HDF5 filter through the corresponding APIs. Here we present the overview of the implementation and code structure, which we have tried to do as clean, simple, modular and efficient as possible. Fig. 4.2 shows the overall structure of our software project.

We have split the actual filter implementation into two files. One file defines the FAPEC compression operations, and the other file defines the decompression operations. We have chosen to compile these two files as shared libraries to provide more versatility and because it contributes to reduce the size of the executable file and the compilation time of the whole program. Benefits in memory management are another argument to do that, and lastly, and most important, if changes have to be done on the filter definition or configuration, updating to a new version of the shared library will solve the problem without having to modify any other code. This is specially interesting in case we wish to keep some parts of the code confidential, as in the case of the FAPEC code.

In two header files we implement the mentioned mapping of datatype properties to FAPEC compression parameters (see Table 4.6) and the function that computes FAPEC common options (chunk size, chunk offset, header offset and block size). Having them implemented as header files avoids duplicate code, since these functions will be used with both writer and reader executables.

Finally, the writer and reader programs are also split into two executable files. The shared library corresponding to the FAPEC filter compression implementation is linked to the Writer, and the shared library corresponding to FAPEC filter decompression implementation is linked to the reader.

Chapter 5

Tests and results

5.1 Test case description

Once FAPEC has been integrated as an HDF5 filter, we have designed and implemented a test to check the performance of the compression algorithm within the file manager. The test consists on executing our writer and reader software over four different files. This is, to compress and decompress them, measuring the compression ratios and the time spent to do the operation for every single compression algorithm that we have integrated: blosc, bzip2, deflate, FAPEC and szip. For bzip2 and deflate we have used compression level 5, and for szip we have configured the "Pixels per Block" compression parameter to 16.

The files used to run these tests come from the Square Kilometre Array (SKA) project [23], from the Gaia space mission of ESA, and from the Advanced Spaceborne Thermal Emission and Reflection Radiometer (ASTER) mission [24] of National Aeronautics and Space Administration (NASA). It is worth to comment that some of these files had to be decompressed before running the test because they were originally compressed using deflate, szip or bit shuffling techniques. Thus, in order to have a raw file (and hence a comparable test), we have first executed our reader to decompress all files. Table 5.1 shows the original and decompressed sizes of these files and the type of filter originally used.

Tables 5.2, 5.3, 5.4 and 5.5 show a description of the SKA.h5, AO.h5, ASTER.h5 and BIG-ASTER.h5 files. The "empty file" size shown in these tables correspond to the size of the file with only the internal structure without datasets. In other words, just with the groups and HDF5 headers. This measure allows us to determine the size of the data for a given file.

Name	Filter	Original size (bytes)	Raw size (bytes)
AO.h5	Shuffle	115858160	115850216
ASTER.h5	Gzip lev.6	422985016	768164128

TABLE 5.1: HDF5 test files (originally compressed). In order to have raw input files, without any filter applied to their datasets, an initial decompression of AO.h5 and ASTER.h5 file was required. The decompression operation has been done using our reader software.

Name	Size(bytes)	Groups	Datasets	Datatypes
SKA.h5	75843628	30	247	32-bit and 64-bit floating-point, 64-bit integer, 8-bit unsigned integer, compound and string.
SKA.h5 (empty file)	23512	30	0	N/A

Datatype	Amount	Total size (bytes)	Data size (bytes)	(%) of file size
H5T_INTEGER_8	1	8328688	8305176	11,0
H5T_INTEGER_64	36	397376	373864	0,5
H5T_FLOAT_32	5	66287408	66263896	87,4
H5T_FLOAT_64	50	387856	364344	0,5
H5T_STRING	3	33160	9648	0,0
H5T_COMPOUND	152	530028	506516	0,7

TABLE 5.2: SKA test file structure.

Name	Size(bytes)	Groups	Datasets	Datatypes
AO.h5	115850216	6	15	Array of 16-bit unsigned integer and compound.
AO.h5 (empty file)	4648	6	0	N/A

Datatype	Amount	Total size (bytes)	Data size (bytes)	(%) of file size
H5T_ARRAY	10	82838128	82833480	71,5
H5T_COMPOUND	5	33018376	33013728	28,5

TABLE 5.3: Gaia AO test file structure.

Name	Size(bytes)	Groups	Datasets	Datatypes
ASTER.h5	768164128	75	239	32-bit and 64-bit floating-point, 32-bit integer and string.
ASTER.h5 (empty file)	66400	75	0	N/A

Datatype	Amount	Total size (bytes)	Data size (bytes)	(%) of file size
H5T_INTEGER	38	80520	14120	0,0
H5T_FLOAT_32	4	768093584	768027184	99,9
H5T_FLOAT_64	74	93112	26712	0,0
H5T_STRING	123	103096	36696	0,0

TABLE 5.4: ASTER test file structure.

Our implementation of the writer allows selecting just a given datatype to be compressed. Note that the entire structure of the file will be created, including information on all groups. This means that the selected datatype will be processed and copied to the output file, but the rest of datatypes will be ignored and not copied to the output file. By knowing the size of the empty structure of the file we can deduce the actual size of the data of a given dataype, by simply subtracting the empty file size to the total file size (which only contains one datatype). Thus, we are able to compare not only the compression ratio and compression/decompression time for the entire file, but also for

Name	Size(bytes)	Groups	Datasets	Datatypes
BIG-ASTER.h5	2332814752	1	4	64-bit floating-point and 16-bit integer.
BIG-ASTER.h5 (empty file)	1832	1	0	N/A

Datatype	Amount	Total size (bytes)	Data size (bytes)	(%) of file size
H5T_INTEGER	1	51845048	51843216	2,2
H5T_FLOAT	3	2280971864	2280970032	97,8

TABLE 5.5: BIG-ASTER test file structure.

Test platform specifications	
OS	Linux 3.16.7-35-desktop openSUSE 13.2 (x86_64).
CPU	AMD Turion(tm) 64 X2 Mobile Technology TL-58 @ 1900MHz.
RAM	2,63GiB of total physical memory + 2GiB of Swap memory.

TABLE 5.6: Equipment specifications to run the tests.

a specific datatype. That will give us a precise measure of how each filter performs for every datatype.

To measure the compression and decompression times we have compiled HDF5 with debugging turned on for the H5Z layer. Such layer is the portion of the API that handles filter operations, so by turning the debugging mode on for this layer, an output from the HDF5 library is given when any kind of filter is used in our program. In this way the code supplies information such as the filter used and its direction (input or output), the total number of bytes processed by the filter including errors, how many of these bytes can be attributed to errors, the elapsed time and the throughput. Note that since the elapsed time is subject to system load, the throughput, which is the total processed bytes divided by elapsed time, may not be completely reliable.

Finally, an HDF5 tool has been used to check if the input file of the writer (that is, the original file) is equal to the output file of the reader (that is, the decompressed or restored one). This tool compares the whole file, its structure, every single dataset, the dataset dataspace and datatype, its contents,... The tool is called *h5diff*, which works in the same way than the Linux command-line tool *diff*.

5.2 Results

The equipment used to run the test is a end-user laptop with the characteristics specified in Table 5.6, whereas Table 5.7 shows a brief description of each parameter that has been measured in order to better understand the results obtained.

Tables 5.8, 5.9, 5.10, 5.11 and 5.12 present the results obtained from the execution of each filter on each file. The best performance for each case is, as usual, highlighted in

Parameter	Description
Proc. in comp.(bytes)	Total number of bytes processed by the filter including errors in compression.
Errors (bytes)	Number of processed bytes that could not be correctly handled by the filter, and thus were output as-is.
Size after comp. (bytes)	Size of the file after applying compression.
Ratio	Reduction of the data size after compression, defined as the uncompressed size divided by the compressed size.
Comp. time (s)	Elapsed time to compress the data.
Comp. throughput (MB/s)	Total number of bytes processed divided by the elapsed time in compression.
Proc. in decomp. (bytes)	Total number of bytes processed by the filter, including errors, in decompression.
Decomp. time (s)	Elapsed time to decompress the data.
Decomp. throughput (MB/s)	Total number of bytes processed divided by the elapsed time in decompression.

TABLE 5.7: Description of the metrics evaluated in the tests.

SKA.h5					
OVERALL compression/decompression results					
	Blosc	Bzip2	Deflate	FAPEC	Szip
Proc. in comp.(bytes)	74582561	74592511	74584144	74594985	74579968
Errors (bytes)	59180897	0	0	0	66060288
Size after Comp. (bytes)	67082657	50945168	**50935237**	66653315	67788462
Ratio	1,131	1,489	**1,489**	1,138	1,119
Comp. Time (s)	**0,78**	27,78	6,62	1,47	0,91
Comp. throughput (MB/s)	**90,88**	2,56	10,73	48,28	78,46
Proc. in decomp. (bytes)	15401664	161345503	110992382	74594985	8519680
Decomp. Time (s)	**0,05**	5,19	1,05	0,74	0,05
Decomp. throughput (MB/s)	**314,90**	29,64	101,10	96,60	167,00

TABLE 5.8: SKA overall test results.

bold face. Table 5.13 presents the compression ratio and compression throughput parameters normalized to the worst case, offering an easy view of how much compression an algorithm achieves and how fast it is with respect to that with the worst performance. Finally, Table 5.14 presents the weighted average for compression throughput and compression ratio (according to the file and dataset sizes), only taking into account the test executions without errors.

5.3 Discussion

The tests that we have executed consist of running our writer program (see Sect. 4.3) over the four files described in Sect. 5.1 using each of the filters that we have integrated. After that, we ran the reader to decompress the files. The results obtained present the performance of each filter when compressing data in an HDF5 file, and specifically, they show the advantages and drawbacks of the FAPEC integration as an HDF5 filter in front of the rest of high-performance algorithms.

SKA.h5					
INTEGER 8-bit compression/decompression results					
	Blosc	Bzip2	Deflate	FAPEC	Szip
Proc. in comp.(bytes)	8257536	8257536	8257536	8257536	8257536
Errors (bytes)	0	0	0	0	0
Size after Comp. (bytes)	407905	**222754**	222870	342378	508889
Ratio	20,418	**37,390**	37,370	24,326	16,366
Comp. Time (s)	**0,02**	0,42	0,22	0,07	0,06
Comp. Bandwidth (MB/s)	**451,70**	18,60	35,06	119,00	125,40
Proc. in decomp. (bytes)	8257536	12470520	11590656	8257536	8257536
Decomp. Time (s)	**0,02**	0,09	0,06	0,06	0,05
Decomp. Bandwidth (MB/s)	**398,00**	128,40	175,90	141,00	170,80
INTEGER 64-bit compression/decompression results					
	Blosc	Bzip2	Deflate	FAPEC	Szip
Proc. in comp.(bytes)	263080	267657	263756	264016	262144
Errors (bytes)	936	0	0	0	0
Size after Comp. (bytes)	139339	140129	**136212**	176584	165065
Ratio	2,852	2,836	**2,917**	2,250	2,407
Comp. Time (s)	**0,00**	0,06	0,09	0,00	0,00
Comp. Bandwidth (MB/s)	**154,60**	4,21	2,85	78,32	54,97
Proc. in decomp. (bytes)	262144	293648	363084	264016	262144
Decomp. Time (s)	**0,00**	0,01	0,00	0,00	0,00
Decomp. Bandwidth (MB/s)	**156,80**	19,86	74,17	56,96	52,59
FLOAT 32-bit compression/decompression results					
	Blosc	Bzip2	Deflate	FAPEC	Szip
Proc. in comp.(bytes)	66060288	66060288	66060288	66070816	66060288
Errors (bytes)	59179008	0	0	0	66060288
Size after Comp. (bytes)	65704922	49746529	**49744875**	65303169	66287408
Ratio	1,009	1,333	**1,333**	1,015	1,000
Comp. Time (s)	**0,76**	28,38	6,42	1,40	0,85
Comp. Bandwidth (MB/s)	**83,12**	2,22	9,81	45,11	74,24
Proc. in decomp. (bytes)	6881280	148561299	99035510	66070816	N/A
Decomp. Time (s)	**0,03**	4,78	0,81	0,68	N/A
Decomp. Bandwidth (MB/s)	**241,20**	29,61	117,00	92,08	N/A
FLOAT 64-bit compression/decompression results					
	Blosc	Bzip2	Deflate	FAPEC	Szip
Proc. in comp.(bytes)	952	6289	1851	190	N/A
Errors (bytes)	952	0	0	0	N/A
Size after Comp. (bytes)	N/A	392097	387856	**387856**	N/A
Ratio	N/A	0,989	1,000	**1,000**	N/A
Comp. Time (s)	N/A	0,07	0,03	0,00	N/A
Comp. Bandwidth (MB/s)	N/A	1,07	0,09	1,87	N/A
Proc. in decomp. (bytes)	N/A	18986	1851	1904	N/A
Decomp. Time (s)	N/A	0,01	0,00	0,00	N/A
Decomp. Bandwidth (MB/s)	N/A	2,86	0,93	2,22	N/A
STRING compression/decompression results					
No chunked layout detected. Filtering not applicable.					
COMPOUND compression/decompression results					
Not atomic datatype. Filtering not implemented.					

Table 5.9: SKA detailed test results.

First of all, FAPEC (and our HDF5 adaptation) appears to be a robust algorithm, because in all the tests FAPEC has performed without any error. Such errors mean that a chunk cannot be compressed or processed by the filter (because a given filter may not support some specific datatype or chunk size, for example). In those cases, the HDF5 library bypasses the filter, and the chunk is copied to the output file without any kind

AO.h5					
OVERALL compression/decompression results					
	Blosc	Bzip2	Deflate	FAPEC	Szip
Proc. in comp.(bytes)	82812672	82812672	82812672	82812672	N/A
Errors (bytes)	0	0	0	0	N/A
Size after Comp. (bytes)	85128993	**70661195**	80820485	72858829	N/A
Ratio	1,361	**1,640**	1,433	1,590	N/A
Comp. Time (s)	**0,85**	22,62	10,38	1,23	N/A
Comp. throughput (MB/s)	**92,91**	3,49	7,61	64,45	N/A
Proc. in decomp.(bytes)	82812672	115435952	123122982	82812672	N/A
Decomp. Time (s)	**0,30**	9,73	1,16	1,29	N/A
Decomp. throughput (MB/s)	**265,50**	11,30	101,20	61,18	N/A
ARRAY compression/decompression results					
	Blosc	Bzip2	Deflate	FAPEC	Szip
Proc. in comp.(bytes)	82812672	82812672	82812672	82812672	N/A
Errors (bytes)	0	0	0	0	N/A
Size after Comp. (bytes)	52116905	**37649107**	47808397	39846741	N/A
Ratio	1,589	**2,200**	1,733	2,079	N/A
Comp. Time (s)	**0,75**	24,76	10,42	1,21	N/A
Comp. throughput (MB/s)	**104,80**	3,19	7,58	65,00	N/A
Proc. in decomp. (bytes)	82812672	115435952	123122982	82812672	N/A
Decomp. Time (s)	**0,28**	9,52	1,32	1,28	N/A
Decomp. throughput (MB/s)	**278,30**	11,55	88,75	61,92	N/A
COMPOUND compression/decompression results					
Not atomic datatype. Filtering not implemented.					

TABLE 5.10: AO file test results.

of processing. That is the reason why we could always recover the original file, despite of the errors, such as in the case of blosc or szip in SKA, or szip in ASTER.

Looking closer at the results for SKA (Tables 5.8 and 5.9), it can be seen that FAPEC performs very well offering a compression throughput of 48MB/s, although the compression ratio is low. It is worth commenting that blosc seems to be the fastest option in this case, but taking into account the number of errors of the filter when dealing with float and even with integer data, and the number of decompressed bytes that is less than the rest of filters, it can be said that FAPEC is the fastest and safer option, since it is almost 5 times faster than deflate, and 19 times faster than bzip2. On the other hand, deflate obtains the best compression ratio together with bzip2, but with a much lower compression throughput. The partial results for the float datatype reveal some difficulties in FAPEC to compress float data efficiently, but also in blosc and szip. This causes a low overall compression ratio for the entire file due to the large fraction of float data in the SKA file (88% of the total data size). In the case of FAPEC it is worth mentioning that there is no pre-processing stage specific for float data yet, but future releases will include it and, thus, we a large improvement can be expected here.

Moving to the Gaia AO file (Table 5.10), FAPEC performs even better than in the SKA case. The compression ratio achieved for the Array datatype is 2,08, whereas the best option, bzip2, achieves a ratio of 2,2. However, the compression throughput is 20 times

ASTER.h5					
OVERALL compression/decompression results					
	Blosc	Bzip2	Deflate	FAPEC	Szip
Proc. in comp.(bytes)	768000000	768000000	768000000	768000000	768000000
Errors (bytes)	0	0	0	0	512000000
Size after Comp. (bytes)	**296633233**	351638465	424306640	328723343	764778061
Ratio	**2,590**	2,185	1,810	2,337	1,004
Comp. Time (s)	**4,89**	298,64	92,91	9,63	18,95
Comp. throughput (MB/s)	**149,60**	2,45	7,88	76,06	38,65
Proc. in decomp.(bytes)	768000000	1429635358	1252280592	768000000	256000000
Decomp. Time (s)	**2,85**	104,40	10,01	10,55	4,53
Decomp. throughput (MB/s)	**257,30**	13,05	119,30	69,41	53,92
FLOAT 32-bit compression/decompression results					
	Blosc	Bzip2	Deflate	FAPEC	Szip
Proc. in comp.(bytes)	768000000	768000000	768000000	768000000	768000000
Errors (bytes)	0	0	0	0	512000000
Size after Comp. (bytes)	**296562689**	351567921	424236096	328652799	764707517
Ratio	**2,590**	2,185	1,811	2,337	1,004
Comp. Time (s)	**4,90**	306,11	94,82	9,88	17,25
Comp. Bandwidth (MB/s)	**149,50**	2,39	7,72	74,16	42,47
Proc. in decomp.(bytes)	768000000	1429635358	1252280592	768000000	256000000
Decomp. Time (s)	**3,43**	105,73	12,61	13,22	4,57
Decomp. Bandwidth (MB/s)	**213,60**	12,89	94,74	55,42	53,43
FLOAT 64-bit compression/decompression results					
No chunked layout detected. Filtering not applicable.					
INTEGER compression/decompression results					
No chunked layout detected. Filtering not applicable.					
STRING compression/decompression results					
No chunked layout detected. Filtering not applicable.					

Table 5.11: ASTER file test results.

faster than with bzip2. Blosc also yields excellent results combining an acceptable ratio with an extremely low compression and decompression time. Regarding the Compound datatype, being this a complex datatype it cannot be compressed yet by our software. Note that one of the limitations of szip is the impossibility to compress complex datatypes such as array datatypes, which is an important limitation that FAPEC does not have.

The ASTER file (Table 5.11) is one of the largest files in this test with almost 770 MB. Almost all of the datasets have a float datatype. The rest of the data that includes integer and string datatypes cannot be compressed because they do not have a chunked layout, as indicated in Sect. 2.6. Again, the results show an excellent performance of FAPEC, which in this case is a bit surprising considering that they are float datatypes. It is almost 32 times faster than bzip2, and 10 times faster than deflate. It also achieves a high compression ratio of 2,34. The reason this is that in this file the float datatypes are compressed much better than in the case of SKA. This, in turn, could be the much smaller dispersion in the actual values. In contrast, the dispersion of float values in SKA was much larger, which the values oscillating in a wide range and even changing the sign often. To conclude the analysis of the ASTER file, we see that blosc obtains

BIG-ASTER.h5					
OVERALL compression/decompression results					
	Blosc	Bzip2	Deflate	FAPEC	Szip
Proc. in comp.(bytes)	2332800000	2332800000	2332800000	2332800000	2332800000
Errors (bytes)	0	0	0	0	0
Size after Comp. (bytes)	75370792	**22043926**	43726608	76763906	255854286
Ratio	30,951	**105,826**	53,350	30,389	9,118
Comp. Time (s)	**5,54**	1061,30	42,63	10,97	127,78
Comp. throughput (MB/s)	**401,50**	2,10	52,10	202,80	17,41
Proc. in decomp.(bytes)	2332800000	2448673416	3455996688	N/A	2332800000
Decomp. Time (s)	**8,27**	82,66	13,32	N/A	70,05
Decomp. throughput (MB/s)	**268,80**	28,25	247,40	N/A	31,76
INTEGER compression/decompression results					
	Blosc	Bzip2	Deflate	FAPEC	Szip
Proc. in comp.(bytes)	51840000	51840000	51840000	51840000	51840000
Errors (bytes)	0	0	0	0	0
Size after Comp. (bytes)	6999831	5405443	7536114	**5354389**	6796587
Ratio	7,406	9,591	6,879	**9,682**	7,628
Comp. Time (s)	**0,16**	4,73	1,63	0,28	0,22
Comp. throughput (MB/s)	**308,00**	10,44	30,31	173,60	224,20
Proc. in decomp.(bytes)	51840000	64804744	60248528	51840000	51840000
Decomp. Time (s)	**0,19**	1,64	0,43	0,35	0,27
Decomp. throughput (MB/s)	**266,00**	37,75	132,70	142,60	179,80
FLOAT compression/decompression results					
	Blosc	Bzip2	Deflate	FAPEC	Szip
Proc. in comp.(bytes)	2280960000	2280960000	2280960000	2280960000	2280960000
Errors (bytes)	0	0	0	0	0
Size after Comp. (bytes)	68373121	**16640643**	36192654	71411677	249059859
Ratio	33,361	**137,072**	63,023	31,941	9,158
Comp. Time (s)	**4,65**	959,96	41,91	9,94	97,08
Comp. throughput (MB/s)	**467,70**	2,27	51,90	218,90	22,40
Proc. in decomp.(bytes)	2280960000	2383868672	3395748160	N/A	2280960000
Decomp. Time (s)	**5,70**	79,82	15,05	N/A	64,28
Decomp. throughput (MB/s)	**381,30**	28,48	215,10	N/A	33,83

Table 5.12: BIG-ASTER file test results.

the best results in this case, since it is 2 times faster in compression than FAPEC with a high compression ratio, and almost 3 times faster in decompression. As described in the blosc documentation, such excellent speeds are achieved because this compressor employs Single Instruction, Multiple Data (SIMD) techniques, such as Streaming Single Instruction Multiple Data Extensions 2 (SSE2) and Advanced Vector Extensions 2 (AVX2) [25]. On the contrary, FAPEC just uses plain standard C instructions.

Finally, in the BIG-ASTER file test (Table 5.12) bzip2 achieves an amazing compression ratio of 105,83, but the compression time is far from being competitive when compared to the other filters. Again, FAPEC performs excellent and is not far from the results obtained by blosc, achieving the best compression ratio for the dataset with an integer datatype. For the float datatype (with almost 2,1 GB) the compression ratio achieved is 31,94 in less than 10 seconds. It is worth mentioning that we have not been able to decompress the FAPEC-compressed file because of limitations in our testing platform.

Normalized compression ratio*					
	Blosc	Bzip2	Deflate	FAPEC	Szip
SKA.h5	N/A	**1,31**	**1,31**	1,00	N/A
AO.h5	1,00	**1,20**	1,05	1,17	N/A
ASTER.h5	**1,43**	1,21	1,00	1,29	N/A
BIG-ASTER.h5	3,39	**11,61**	5,58	3,33	1,00
Normalized compression throughput*					
	Blosc	Bzip2	Deflate	FAPEC	Szip
SKA.h5	N/A	1,00	4,19	**18,85**	N/A
AO.h5	**26,61**	1,00	2,18	18,46	N/A
ASTER.h5	**61,01**	1,00	3,21	31,02	N/A
BIG-ASTER.h5	**191,56**	1,00	24,86	96,76	8,31
					* Higher is better

TABLE 5.13: Normalized compression results with respect to the worst value. Executions with errors in the tests are not taken into account.

Weighted average of compression results*					
	Blosc	Bzip2	Deflate	FAPEC	Szip
Ratio	23,11	**75,58**	38,30	22,16	9,12
Throughput	**330,23**	2,24	39,27	161,81	17,41
					* Higher is better

TABLE 5.14: Weighted average of compression results. We have only considered the test executions without errors.

As already said at the beginning of this section, FAPEC has performed all the tests without errors, in contrast to the case of blosc or szip filters that have reported many errors when compressing SKA.h5 and ASTER.h5 files. With this in mind, and seeing the excellent results in compression ratio and bandwidth in all cases, it is clear that FAPEC is the most balanced compression algorithm. Finally, it is important to remark that in all cases, the use of the HDF5 tool `h5diff` has reported no differences between the original file and the decompressed one.

Chapter 6

Conclusions

6.1 Conclusions

In this work we have integrated FAPEC as an HDF5 filter aimed at super-computing environments that need to optimize the size of their HDF5 files, the computational cost and the file transfer time between nodes and storage systems. This integration is also a clear solution for those users that need a high performance compression algorithm in their projects, specially in cases where the compression techniques integrated by default in the HDF5 library do not satisfy their needs.

The integration of FAPEC as an HDF5 filter has been done in a clear and simple way. We have chosen to compile the compression and decompression routines of FAPEC as shared libraries because it contributes to reduce the size of the final executable file and the compilation time of the entire program, but specially it leads to a more modular implementation. Benefits in memory management are another argument to do that, and lastly, and most important, if changes have to be done on the filter definition or configuration, updating to new version of the shared library will solve the problem without requiring modifications to any other code parts.

The results of the tests reveal a good performance of FAPEC on HDF5 files in general. In the SKA, AO and ASTER file cases, FAPEC provides a faster compression speed than bzip2 and deflate. In the most extreme case, the compression throughput is increased by 20 times with respect to bzip2, and by 4 times with respect to deflate, while achieving very similar compression ratios. The case of BIG-ASTER file is a bit different because bzip2 and deflate achieve extremely high compression ratios at the expense of not being competitive in terms of compression times. In this case, FAPEC works extremely well when compressing integer data, and it also achieves very good results in ratios and speed

when compressing float data. Specifically, it is 100 times faster than bzip2, and 4 times faster than deflate.

In contrast to szip, the resiliency of the algorithm and the integration in itself allows FAPEC to operate without errors in a wide range of datatypes, even in a specific complex datatype such as arrays. Since data can be very heterogeneous in HDF5, the robustness of the filter and of its integration is a critical point in order to allow users to process their files without the risk of data corruption or data incompatibilities. On the other hand, we have found that the blosc compressor is a tough competitor. Yielding very similar compression ratios, blosc is typically two times faster than FAPEC in all cases, and in decompression blosc works even better despite it is not the most robust algorithm.

6.2 Future work

Several issues might be improved in order to improve even further the FAPEC integration as an HDF5 filter. First of all, it would be interesting to implement a solution to compress complex datatypes with FAPEC, and specially, compound datatypes. They seem to be frequently found in HDF5 files. To do that, the integration layer has to discover all datatypes of a compound dataset, and read the data of every portion of the compound datatype so they can be sent to the filter.

Regarding the implementation of the data buffer where data is stored when a dataset is read, it could be improved in order to be more efficient in terms of memory usage. Currently, the memory usage when a file is considerably large increases proportionally to the file size, so a circular data buffer could be implemented to solve this issue.

Lastly, The HDF Group allows to register third-party filters in their website. The Group would then assign a filter identifier, which could be used by our filter. That would mean that the HDF5 API may include the FAPEC filter, so other users could know about our solution and use it in their systems. We should check with the HDF5 Group which requirements should be met in order to perform such request.

Appendix A

MD5 filter

A.1 Filter definition

The MD5 filter used in Section 2.6.2 is defined as follows. Note that the input part and the output part have been defined into the same structure.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/md5.h>
#include <assert.h>
#include "hdf5.h"

#define HAVE_MD5

/* Define MD5 filter function */

size_t md5_filter(unsigned int flags, size_t cd_nelmts,
            const unsigned int cd_values[], size_t nbytes,
            size_t *buf_size, void **buf)
{
#ifdef HAVE_MD5
    unsigned char       cksum[16];

    if (flags & H5Z_FLAG_REVERSE) {
      /* Input */
        assert(nbytes>=16);
        MD5(*buf, nbytes-16, cksum);
```

```
        /* Compare */
        if (memcmp(cksum, (char*)(*buf)+nbytes-16, 16)) {
          return 0; /*fail*/
        }

        /* Strip off checksum */
        return nbytes-16;

    } else {
        /* Output */
        MD5(*buf, nbytes, cksum);

        /* Increase buffer size if necessary */
        if (nbytes+16 > *buf_size) {
            *buf_size = nbytes + 16;
            *buf = realloc(*buf, *buf_size);
        }

        /* Append checksum */
        memcpy((char*)(*buf)+nbytes, cksum, 16);
        return nbytes+16;
    }
#else
    return 0; /*fail*/
#endif
}
```

Appendix B

FAPEC filter

B.1 Filter definition for FAPEC Core 2016.0 Release

The next listing shows the compression part of FAPEC filter definition:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "fapec_opts.h"
#include "fapec_comp.h"
#include "hdf5.h"
#include "fapec_conf_com_opt.h"

#define DEBUG 0
#define STRING_TYPE 3

#define HAVE_FAPEC

/* Define FAPEC compression filter function */

size_t fapec_call(unsigned int flags, size_t cd_nelmts,
                  const unsigned int cd_values[], size_t nbytes,
                  size_t *buf_size, void **buf)
{
  #ifdef HAVE_FAPEC

  int status, fapec_copts[4] = {0, 0, 0, 0};
  int fapecUsrOpts = 0; /* Default user config /
  size_t uncomp_nbytes;
  t_fapecOpts *fapecOpts = NULL;
```

```
/* Initialize fapec options var */
fapecOpts = newFapecOpts();

if (flags & H5Z_FLAG_REVERSE) {
  /* NOT AVAILABLE. Return error to HDF5 */
  printf("Error. Not available\n");
  return 0;

} else {
  /* Output */

  /* Size of the chunk before compression */
  uncomp_nbytes = nbytes;

  /*
   * Set FAPEC common options
   */
      status = configureFapecCommonOptions(cd_values, cd_nelmts,
                                           fapec_copts, nbytes);
  if(status != 0){
    printf("Error coding common FAPEC options: %d\n", status);
    // Return failure to HDF5
    return 0;
  }

  status = setCommonFapecOpts(fapecOpts, fapec_copts[0], fapec_copts[1],
                              fapec_copts[2], fapec_copts[3]);
  if(status != 0){
    printf("Error configuring common FAPEC options: %d\n", status);
    // Return failure to HDF5
    return 0;
  }

  /*
   * Set FAPEC compression options
   */
  if(cd_values[0] == STRING_TYPE){
    status = activateTextFapecOpts(fapecOpts, 3);
    if(status != 0){
      printf ("Error configuring FAPEC text options: %d\n", status);
      // Return failure to HDF5
      return 0;
    }
  } else{
    /* Configure basic fapec options: size, endianness, interleaving */
    status = activateBasicFapecOpts(fapecOpts, cd_values[1],
                                    cd_values[3], cd_values[2]);
```

```
      if(status != 0){
        printf ("Error configuring basic fapec options: %d\n", status);
        // Return failure to HDF5
        return 0;
      }
    }

    /*
     * Compress the chunks
     */
    status = fapecChunkCompression((unsigned char **) buf,
                                   &nbytes, fapecUsrOpts, fapecOpts);

    free(fapecOpts);
    return nbytes;
  }

  #else

  return 0; /* failure */
  #endif
}
```

The next listing shows de decompression part of FAPEC filter definition:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "fapec_opts.h"
#include "fapec_decomp.h"
#include "hdf5.h"
#include "fapec_conf_com_opt.h"

#define DEBUG 0
#define STRING_TYPE 3

#define HAVE_FAPEC

/* Define FAPEC compression filter function */

size_t fapec_call(unsigned int flags, size_t cd_nelmts,
                  const unsigned int cd_values[], size_t nbytes,
                  size_t *buf_size, void **buf)
{
  #ifdef HAVE_FAPEC

  int status, fapec_copts[4] = {0, 0, 0, 0};
```

```
int fapecUsrOpts = 0; /* Default user config */
size_t comp_nbytes;
t_fapecOpts *fapecOpts = NULL;

/* Initialize fapec options var */
fapecOpts = newFapecOpts();

/* Define maximum "verbosity" of FAPEC operation */
fapec_usrCfgSet_verbosity(fapecUsrOpts,2);

if (flags & H5Z_FLAG_REVERSE) {
  /* Size of the chunk before decompression */
  comp_nbytes = nbytes;

  /*
   * Set FAPEC common options
   */
  status = configureFapecCommonOptions(cd_values, cd_nelmts,
                                       fapec_copts, nbytes);
  if(status != 0){
    printf("Error coding common FAPEC options: %d\n", status);
    // Return failure to HDF5
    return 0;
  }

  /*
   * Beware: In Decompression we must set a chunksize of 0,
   * because we cannot know it beforehand in the case of HDF5
   */
  status = setCommonFapecOpts(fapecOpts, 0, fapec_copts[1],
                              fapec_copts[2], fapec_copts[3]);
  if(status != 0){
    printf("Error configuring common FAPEC options: %d\n", status);
    // Return failure to HDF5
    return 0;
  }

  /*
   * Set FAPEC decompression options
   */
  if(cd_values[0] == STRING_TYPE){
    status = activateTextFapecOpts(fapecOpts, 3);
    if(status != 0){
      printf ("Error configuring FAPEC text options: %d\n", status);
      // Return failure to HDF5
      return 0;
    }
  } else{
```

```
      /* Configure basic fapec options: size, endianness, interleaving */
      status = activateBasicFapecOpts(fapecOpts, cd_values[1],
                                      cd_values[3], cd_values[2]);
      if(status != 0){
        printf ("Error configuring basic fapec options: %d\n", status);
        // Return failure to HDF5
        return 0;
      }
    }

    /*
     * Decompress the chunks
     */
    status = fapecChunkDecompression((unsigned char **) buf,
                                     &nbytes, fapecUsrOpts, fapecOpts);

    free(fapecOpts);
    return nbytes;

  } else {
    /* Output */
    /* NOT AVAILABLE. Return error to HDF5 */
    printf("Error. Not available\n");
    return 0;
  }

  #else

  return 0; /* failure */
  #endif
}
```

Bibliography

[1] J. Portell, E. García–Berro, C. Estepa, J. Castaeda, and M. Clotet. Efficient data storage of astronomical data using hdf5 and pec compression. *High-Performance Computing in Remote Sensing*, 1, 2011. doi: 10.1117/12.898203. Prague 2011, SPIE.

[2] The HDF GROUP. Who uses hdf? HDF web page, January 2016. URL `https://www.hdfgroup.org/users.html`. Last modified on 29 January 2016.

[3] The HDF GROUP. HDF products. HDF web page, March 2016. URL `https://www.hdfgroup.org/products/`. Last modified on 24 March 2016.

[4] The HDF Group. HDF5: API Specification Reference Manual. URL `https://www.hdfgroup.org/HDF5/doc/RM/RM_H5Front.html`.

[5] The HDF GROUP. Limits in HDF5. HDF web page, November 2015. URL `https://www.hdfgroup.org/HDF5/faq/limits.html`. Last modified on 19 November 2015.

[6] The HDF GROUP. *HDF5 User's Guide*, March 2016. URL `https://www.hdfgroup.org/HDF5/doc/UG/HDF5_Users_Guide-ResponsiveHTML5/index.html#t=HDF5_Users_Guide%2FHDF5_UG_Title%2FHDF5_UG_Title.htm`. HDF5 Release 1.10.0.

[7] The HDF GROUP. Filters in HDF5. HDF web page, August 2001. URL `https://www.hdfgroup.org/HDF5/doc/H5.user/Filters.html`. Last modified on 2 August 2003.

[8] OpenSSL Cryptography and SSL/TLS Toolkit. URL `https://www.openssl.org/`.

[9] The HDF GROUP. Filters. HDF web page, June 2016. URL `https://www.hdfgroup.org/services/filters.html`. Last modified on 8 June 2016.

[10] Jean loup Gailly. Zlib homepage. URL `http://zlib.net/`.

[11] Antaeus Feldspar. An Explanation of the Deflate Algorithm. *comp.compression forum*, 1997.

[12] Michael Schindler. Szip homepage. URL `http://www.compressconsult.com/szip/`.

[13] R.F. Rice. Some practical universal noiseless codin techniques. *JPL Tech Rep.*, pages 22–79, 1979. Jet Propulsion Laboratory.

[14] Consultative Committee for Space Data Systems. Lossless Data Compression, Blue Book. Technical Report CCSDS 121.0-B-1, CCSDS, 1993.

[15] Pen-Shu Yeh, Wei Xia-Serafino, Lowell Miles, Ben Kobler, and Daniel Menasce. Implementation of CCSDS lossless data compression in HDF. In *Earth Science Technology Conference-2002*, 2002. URL `http://esto.nasa.gov/conferences/estc-2002/Papers/A3P2(Yeh).pdf`.

[16] Julian Seward. Bzip2 user manual. URL `http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.html`.

[17] Francesc Alted. Blosc homepage. URL `http://www.blosc.org/blosc-in-depth.html`.

[18] Francesc Alted. Why modern CPU's are starving and what can be done about it. *IEEE computing now*, 2010.

[19] Dapcom Data Services. URL `http://www.dapcom.es/fapec_core_2016_0.html`.

[20] J. Portell, A. G. Villafranca, and E. García–Berro. Quick outlier-resilient entropy coder for space missions. *Journal of Applied Remote Sensing*, 4:339–363, 2010. doi: 10.1051/0004-6361:20010085.

[21] DAPCOM Data Services. *FAPEC Core 2016.0 API Reference*, April 2016. DDS-HARC-UM-02.

[22] M. A. C. Perryman, K. S. de Boer, G. Gilmore, E. Høg, M. G. Lattanzi, L. Lindegren, X. Luri, F. Mignard, O. Pace, and P. T. de Zeeuw. GAIA: Composition, formation and evolution of the Galaxy. *Astron. & Astrophys.*, 369:339–363, Apr 2001. doi: 10.1051/0004-6361:20010085.

[23] P. Diamond. The Square Kilometre Array: A Physics Machine for the 21st Century. *SPIE Newsroom*, July 2014. doi: 10.1117/2.3201407.12.

[24] M. Abrams. The Advanced Spaceborne Thermal Emission and Reflection Radiometer (ASTER): Data products for the high spatial resolution imager on NASA's Terra platform. *International Journal of Remote Sensing*, 21:847–859, 2010. doi: 10.1080/014311600210326.

[25] David A. Patterson and John L. Hennessey. *Computer Organization and Design: the Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., 2nd edition edition. San Francisco, California, 1998, p.751.