2021 7th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD-7) | 978-1-7281-8672-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/DRBSD754563.2021.00005

Productive and Performant Generic Lossy Data Compression with LibPressio

Robert Underwood *, Victoriana Malvoso †, Jon C. Calhoun †, Sheng Di‡, and Franck Cappello‡
* *School of Computing*, *Clemson University* Clemson, USA
† *Holcombe Department of Electrical and Computing Engineering*, *Clemson University* Clemson, USA
‡ *Mathematics and Computer Science Division Argonne National Laboratory* Lemont, USA

***Abstract*—In recent years, lossless and lossy compressors have been developed to cope with the ever increasing volume of scientific floating point data. However not all compression techniques are appropriate for all data-sets, and determining which one to use can be time consuming requiring code modifications and trial and error. We present LibPressio – a generic library for the compression of dense tensors that minimizes the code changes scientists need to make to take advantage of new and improved compression techniques. We compare LibPressio to 9 different competing libraries and measure the overhead of their design decisions as well as overall run time overhead showing insignificant overhead. We further show an improvement in usability as measured by a reduction in lines of code compared to native code by 50-90%. The value of this tool can be seen by integration into Z-Checker and ADIOS2.**

***Index Terms*—Error Bounded Lossy Compression, LibPressio**

I. Introduction

In recent years, modern compression methods such as ZSTD, SZ, ZFP, and MGARD have begun to revolutionize the way that application scientists transport data across the network, store data to persistent storage, and process data in memory. This is because error bounded lossy compression methods achieve much higher compression ratios than what can be typically achieved with lossless compression methods, but also provide a bound on the errors introduced making lossy compression viable for scientific applications where data integrity is key. These thrusts have enabled advances in check-pointing, cosmology codes, climate codes, physics simulations, and more [1].

In practice, scientists who use compression methods need a consistent, flexible, and high performance interface to use and understand the effects of compression regardless of what programming language they use. Existing compressors suffer from a proliferation of interfaces and semantics, making it difficult to perform comparisons between methods [2], [3], [4]. As such, users need one interface so that they can focus on their science and allow compression researchers to develop their methods independently. This is essentially the same line of argument that decades ago led to the MPI specification from proliferating and inconsistent message passing interfaces.

In this paper, we present LibPressio – a uniform, low overhead, productive interface which applications can use to automatically configure, perform in parallel, and analyze the results of compression. This work is challenging because (1) the sheer number and variety of interfaces have been provided by various compressors, (2) the uniform interface to be developed requires careful attention such that all compressors can be executed efficiently, and (3) the interface implementation should induce minimal (almost non-measurable) overhead.

LibPressio enables a wide array of compressor-agnostic uses of compression including: 1) generic Command Line Interface (CLI), 2) IO library plugins for HDF5 and ADIOS2 3) configuration optimizer, 4) distributed and parallel compression experiments 5) compression quality analysis tools like Z-checker 6) Correctness testing tools like fuzzers 7) bindings for other languages including Julia, R, Python, and Rust, and 8) and exascale scientific applications. These are tools that previously would need to be rewritten for each compressor.

This paper presents the design, implementation, and some of the applications of LibPressio. Our contributions are: 1) an extensive analysis of the API designs of 9 competing libraries highlighting their strengths and weaknesses for use in high performance parallel and distributed computing. 2) measurement of 6 areas where existing compressor interfaces introduce overhead when using modern compressors and demonstrate no statistically significant overhead relative to calling the native APIs directly 3) a 50% - 90% reduction in the volume of client code as measured across 11 different applications implemented with at least feature parity and often additional features. Finally existing tools such as Z-Checker[5] and ADIOS-2[6] have already integrated our library.

II. Background

Designing a uniform effective interface for both lossless compression and lossy compression requires a deep understanding of different kinds of operations in compression and decompression.

For lossless compression, designing a uniform interface seems somewhat straightforward but is actually complex. For lossless compressors, there are some minor differences in the semantics relating to initialization of global structures, what data in memory is held constant during compression, whether to accept inputs from the file system or from memory also, memory management for the underlying buffers, and the passing of additional configuration parameters. Furthermore, some specialized lossless compressors like fpzip[7] only

This research was funded grants from the National Science Foundation and the US Department of Energy

978-1-7281-8672-6/21/$31.00 ©2021 IEEE
DOI 10.1109/DRBSD754563.2021.00005

accept floating point inputs. This means that a compressor interface that abstracts between different lossless compressors needs to pass metadata for the data being compressed.

With lossy compression, this problem is compounded. The decompression process for most lossy compressors preserves the "structure" of the output while the values are often not preserved but instead are approximations of the originals. The loss of information requires users to conduct tests and experiments with different lossy compressors to ensure that their data is preserved sufficiently for their applications. However, the lack of consistent implementations for this often leads to a proliferation of metrics interfaces and implementations as well. Additionally, many of the leading lossy compressors for dense tensors take advantage of spatial information requiring a more sophisticated metadata to describe dimension ordering, type information for their inputs and the leading compressors again all differ on how to best do this.

III. Related Work

There have been multiple prior attempts at compression interface libraries.

One example of such a uniform compression interface is libarchive – the library that underlies many modern implementations of the Linux/UNIX `tar` command. It supports a number of what it calls filters which are lossless compressors such as gzip, lz4, lzma, sx, zstd, and others. It also supports a number of formats such as cip, zip, tar, rar and other container formats which encapsulate separate files stored within the buffer. One of its key portability features is that it uses callback functions so that archives can be read or written from sockets, files, memory, or other custom resources. Lastly libarchive expects a "record" organized layout – that an archive file consists of one or more "named" records. However in scientific codes, this is not always true, but share some similarities with HDF5 which is commonly used in HPC. The weakness of libarchive is that it only supports lossless formats, has no concept of the underlying type or structure of the data being stored, and does not allow third party filters.

There are two examples of libraries dedicated to compressing/decompressing lossy artifacts: `imagemagick` [8] and `ffmpeg` [9]. These compress/decompress images and video respectively. However not all scientific data can be neatly characterized either as a 2D image or even a video. So while these interfaces could be used, they do not necessarily represent a universal interface for lossy compression, but rather specialized interfaces for specific domains. ImageMagick further supports a variety of features that generally feel out of place in scientific computing such as image transforms or color mapping. The same is true for the various libraries such as the one in VLC media player application called `libcompression`. In addition to these libraries there are also specialized libraries for specific algorithms such as `libjpeg-turbo`. These libraries use still different interfaces which optimize for there use-case by providing a "by-scanline" interface which allows reading images by row for ease of rendering. This falls short of the various random and parallel strides access patterns used by HPC codes.

In Python, there are tools that are closer to LibPressio. For example, NumCodecs [10] is a Python library that provides a set of "codecs" which implement various lossless and lossy compressors. An important limitation of NumCodecs is that it is a Python library, and a vast majority of HPC codes are not written in Python. The translation between Python and C++, even with the Python buffer protocol introduced in Python 3.2 and improved in 3.3 is still moderately expensive. The further restrictions of the global interpreter lock for multi-threading in C Python make it unlikely that applications will embed Python to perform compression in a multi-threaded C or Fortran application. Moreover, the interface requirements of NumCodecs are overly strict. For example, at time of writing, SZ – one of the leading error bounded lossy compressors cannot fully implement NumCodecs' interface correctly due to SZ's use of global memory to store some configuration parameters. Additionally, NumCodecs does not support all of the kinds of options modern lossy compressors in HPC require. For example, it doesn't support passing structures or opaque pointers that cannot be serialized as JSON such as `MPI_Comm` or `sycl::queue` structures used by some compressors to control parallelism. Finally, NumCodecs does not have a uniform way to query certain kinds of important information about compressors such as their thread safety level.

Analysis tools such as Z-checker[5] and Foresight[2] also provide there own lossy compression interface. Older version of Z-Checker as well as current versions of CBench – the compression library behind Foresight – provide their own compressor interface layers. Both of these interface are limited in that they aim to adapt only a subset of compressor options. Z-checker is notable in that it provides adapters to convert the native bounds kept by error bounded lossy compressors those supported by SZ using mathematical relationships between the bounds. Both of these tools use "string-ly typed" configuration parameters. Neither of these libraries provide run-time information such as thread safety that can be used to safely parallelize workloads. Furthermore, only parts of Zchecker [5] which are designed to collect and store metrics are designed to be embedded into other applications. While for CBench the actual compressor interface can be embedded, Foresight was designed as a standalone application cannot.

IV. Design Overview

The architecture of LibPressio was developed considering all the strengths and weaknesses of the existing compressor libraries. Our goal is to develop a library which would fit the specific needs of users in the HPC community who have some of the most complex needs of any compression user.

LibPressio has six major components, as illustrated in Figure 1. The `pressio` component is used to create references to, enumerate, and handle errors while creating compressors, metrics and IO modules. The `pressio_data` is an abstraction for handling memory management, and different shapes

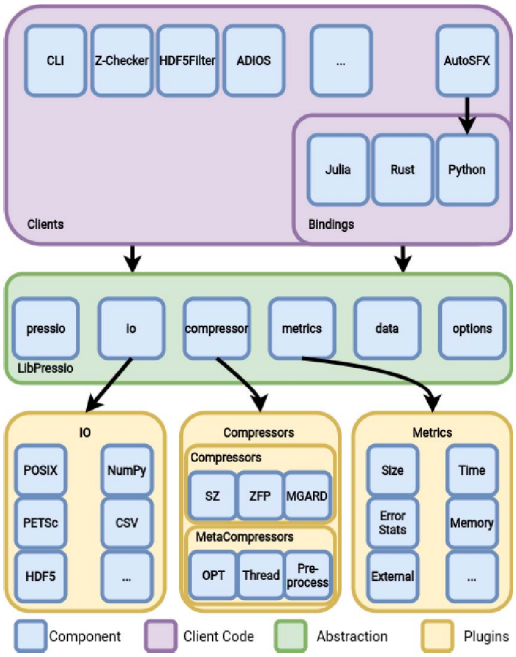

Fig. 1. Major Components of LibPressio

and types of data buffers. The `pressio_compressor` component is used to compress and decompress data. The `pressio_options` structure holds introspect-able configuration for compressors, metrics, and IO objects. The `pressio_io` component provide convenience functions for reading data from various sources into our out of `pressio_data` buffers. The `pressio_metrics` component provides functions for measuring the performance of compression and the quality of compression.

To illustrate the usage of the library, we provide a complete example of the LibPressio API in Appendix A. In the remainder of this section, we mainly discuss some of the key architectural decisions in LibPressio.

A. Data Abstraction

A compressor interface library ought to have an abstraction to describe the type and layout of data. This is because every compressor library that we studied has a different understanding of what it should be passed. This also allows the underlying implementation to use the dimension and type information if it can be used, and ignore it if it cannot be.

Memory management is another key aspect of the data abstraction because of diverse use-cases in practice. The compression library may need to put compressed/decompressed data into a user-provided memory space, may need to allocate the compressed or decompressed buffer for compression and decompression respectively, or it may need to run on a GPU or on persistent memory [11]. If the library does not take responsibility for memory management, users would not know how to pass memory to or from each application and it would leak through the abstraction.

In our design, we adopt the most flexible solution, which is essentially a pointer, with an array to store the dimension information, and an enum to store the datatype, a function pointer to a deleter method, and an optional void pointer to state for the deleter method. As an alternative to an enum the address of a fully specialized template function could be used; this design is used by many implementations of `std::any` to allow it to be used without Run-time Type Information (RTTI). The advantage of this design is that it trivially supports new types. The disadvantage of this design is that this address could differ from compiler to compiler, and could foil network serialization in a heterogeneous environment. The design allows users to use persistent or GPU unified memory memory function with APIs like `mmap` or `sycl::malloc_device` easily. The deleter can be a static function to `memunmap` or `sycl::free` respectively. Likewise, this design allows for shallow copies where the deleter function is a noop.

B. Compression and Decompression

Compressor interfaces themselves differ in a many ways. A good interface needs to handle all compressors.

Ensuring users use the correct ordering of dimension in the lossy compression is a critical prerequisite of getting expected compression quality. In fact, passing the wrong dimension ordering can result in a seriously poor compression quality because of incorrect strides to be used to represent the data. Whereas, the existing lossy compressors adopt diverse dimension orderings in their interfaces. For instance, ZFP, image and video libraries tend to use a Fortran based dimension ordering whereas SZ and MGARD expect a C based dimension ordering in their interfaces. For the sake of simplicity and credibility, LibPressio provides a uniform interface with consistent dimension ordering across different compressors, and the underlying ordering corresponding to different compressors is handled transparently to users.

Compressors have different construction methods, which could be another heavy burden for users to call different compressors correctly. SZ, for example, has a single shared configuration store which is created by `SZ_Init` and deallocated by `SZ_Finalize`. ZFP can have multiple independent configuration stores. This has impacts on thread safety since a thread can only call SZ_Finalize if they are confident no other thread (possibly in a different library) is still using SZ. The safest approach is reference count instances of compressors and to provide an interface to indicate if the instance returned is a shared instance or not – allowing use of multi-threading if it is not shared.

Compressors may not encode the compressor configuration into the compressed byte stream. LibPressio allows users to provide the configuration outside of the use of the compressor, and provide interfaces to use the stream encoded metadata if it exists, which is similar to NumCodecs.

Compressors may or may not clobber the buffers passed to them by the user. Some versions of SZ and MGARD, for example, treat the input data as mutable to save on memory during compression. However, clobbering the input data is unusual amongst compressors, and could surprise the user. Enforcing `const`-ness of the input data is preferable from a

consistency perspective, so compressors that clobber the input data should generally make a copy and compress on the copy.

Another diversity amongst various error bounded compressors is they generally have different notions of error bounds or options. SZ for example has 27+ different configuration parameters, where as some lossless compressors have either zero or one parameter. The compressor interface – LibPressio allows compressors to have arbitrarily many options, while at the same time providing a list of "common" options understood by one or more compressors. Since LibPressio also provides introspection, users can select the compressors that meet their specific needs programmatically.

C. Option Abstractions

Now that we have discussed the ways in which compressors can differ, we consider the abstractions for representing these configuration options. Introspection of types is key to an interpret-able interface between compressors. Users need to know what type the compressor expects in order to supply arguments of the correct type. In LibPressio, each option reports its type as one of 9 options: signed and unsigned integers of size 8, 16, 32, and 64, IEEE 32-bit single precision floating point, IEEE 64-bit double precision floating point, string, array of string, data, user data, and unset. The first 5 store a scalar of the specified type. The array of string option stores a dynamically sized list of string. It can be used for compressors which support multiple error bounds at a time. The data option stores a full `pressio_data` buffer. It can be used for compressors which need a mask such as SZ's ExaFEL mode [12]. The user data mode stores a void pointer. It is used to pass opaque types that represents parallel resources such as MPI_Comm. The unset type is used to indicate an invalid error state and does not actually contain data.

D. Plugins

At time of writing, we have developed over 54 public first-party plugins in LibPressio, based on the connection with researchers and scientists from across 6 different institutions. These plugins include all of the leading error bounded lossy compressors as well as common lossless and image compressors. LibPressio also supports common IO formats used in scientific computing such as flat binary files, HDF5 files, and character-delimited files such as CSV. These plugins enabled researchers to quickly adopt LibPressio into their existing work flows. Figure 2 gives an overview of the available plugins. Descriptions of these are found in the glossary.

LibPressio also supports a number of meta compressors. Some of these plugins allow performing common, useful pre/post processing steps such as transposition, resizing, and linear quantization. Others provide more robust capabilities such as fault or statistical error injection, auto tuning, and automatic task-based parallelization. This allows users or even compressor developers to experiment with different compressor designs out of their consistent functional parts such as quantization, transform, prediction, and encoding stages. This even allows new approaches to compressor architecture by allowing compressor developers to focus on providing specific pieces of the compression pipeline.

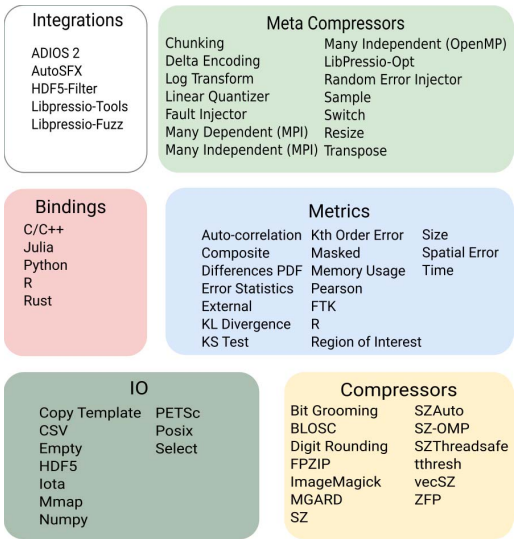

Fig. 2. List of Plugins, Applications and Language Bindings using LibPressio. Descriptions can be found in the Glossary. A up to date list and description can be found at https://github.com/robertu94/libpressio

TABLE I
FEATURE COMPARISON TABLE

library	lossless compression	lossy compression	n-d data aware	datatype-aware	embeddable design	arbitrary configuration	option introspection	third party extensions
ADIOS-2 [6]	✓	✓	✓	✓	✓	✗	✗	✗
ffmpeg [9]	✓	✓	❑	✓	✓	✓	✓	✓
Foresight/CBench [2]	✓	✓	✓	✗	❑	✓	✗	✗
HDF5 [13]	✓	✓	✓	✓	✓	✗	✓	✓
imagemagick [8]	✓	✓	❑	✓	✓	✓	✗	✓
libarchive [14]	✓	✗	✗	✗	✓	✗	✓	✗
NumCodes [10]	✓	✓	✓	✓	✗	✗	✓	✓
SCIL [3]	✓	✓	✓	✓	✓	✗	✓	✗
Z-checker (0.7) [5]	✓	✓	❑	✓	❑	✗	✗	✓
LibPressio	✓	✓	✓	✓	✓	✓	✓	✓

Beyond meta compressors, unlike prior approaches of developing tools for compression which were tied to specific compressors, meta compressors and external tools built upon LibPressio benefit the entire compression community. Abstractions such as ZFP's inline arrays, python bindings for a compressor, HDF5 plugins no longer have to be developed for a single compressor, but all of them simultaneously.

V. Design Impacts on Performance

In this section, we compare compression interface libraries in terms of the following criteria that impact runtime and compression performance and explained in subsequent paragraphs: (1) Does it support lossless compressors? (2) Does it support lossy compressors? (3) Is it dimension aware? (4) Is it datatype aware? (5) Is it embedable in-process? (6) Does it allow arbitrary pointers for configuration? Table I provides an

overview of the features of the compressor interface libraries. The remaining columns are discussed in Section VII.

The first two categories are relatively self explanatory. If a compressor interface provides any lossless compressors, it has a ✓ for lossless compressors. If a compressor interface provides any lossy compressor, it has a ✓ for lossy compressors.

Most lossy compressors and some lossless compressors support multi-dimensional data in their compression [15], [16], [17], which strongly depend on the layout (or spatial features) of the data during the compression. We give a ✓ to libraries which support arbitrary dimensions, a ❑ to have dimensions, but do not support arbitrary ones, and a ✗ to libraries consume only 1d data. Passing the information about dimensions correctly is critical to getting high compression quality for three reasons. First, Incorrect ordering of dimension may significantly degrade the compression quality. According to our measurements, on the CLOUD field of the hurricane dataset, mistakenly reversing the order of the dimensions in SZ would lower the compression ratio between 1.4× to 1.8× for the value range relative error bounds of 1e-5 to 1e-2. Second, using mismatched number of dimensions may also significantly degrade the compression quality. Although most compressors supporting high-dimensional compression can often treat the contiguious higher dimensional datasets as lower dimensions with a larger stride, the corresponding compression would suffer from significantly lower compression quality. Our measurements show that treating the same multi-dimensional data buffers/files as 1D reduces compression ratios between 1.2× and 1.3×. Third, data cannot always be treated as 1D, e.g., MGARD requires at least 3 rows in each dimension or it returns an error rather than compressing the data [17]. Likewise while compression may not fail, passing incorrect information about dimensions can produce inefficient compression. For example, with ZFP, passing any one dimension smaller than the blocksize (i.e $3 \times N$) results in inefficient compression due to required zero padding for the algorithm.

If a compressor interface is data-type aware, it requires information about data type and supports at least two data types. This information is critical to correctly preserving data especially for lossy compression. One cannot preserve a data type to a non-zero error tolerance if he/she does not know how the data is stored. Lossless compressors can allow multiple data types without being aware of data-type because they treat all input types as a stream of bytes regardless of the underlying structure of the data. However, they also typically do not accept information about type information [1].

If a compressor interface is embeddable, it can be embedded into an application written in native languages such as C or C++ without the use of the `exec` or loading an interpreter. Zchecker and Foresight/CBench get a ❑ because only portions of their API are embeddable. This is important because many HPC environments and frameworks (such as MPI) limit the use of `exec` to start other processes, and running interpreters can be expensive overhead for running an application [18]. In our measurements, spawning an external process and copying the data back and forth across process boundaries (i.e. NumCodecs/ZChecker) takes on the order of 174ms where compressing the CLOUD field of Hurricane takes on the order of 993ms; meaning that ignoring embedding can have a performance penalty of $\approx 17.5\%$ on each compression operation preformed. Some compressors can take much longer if they have complex initialization (i.e. if the specific compressor uses MPI [4]) on the order of 1997ms overhead or 201.1%.

If a compressor interface supports arbitrary configuration, it can accept arguments of arbitrary type. This is essential to support compressors that can be configured with non-serializable native types such as `MPI_Comm` or `cudaStream_t` to control the degree and placement of parallel resources [11], [19]. These compressors can be dramatically faster that variants that do not use these types. This can also be important for compressors such as SZ which require structs be passed for configuration of certain modes that may not have native serializable representations. For this reason, compressor interface which are string-ly typed (use strings to store configuration [20], [2], and parse the string to the appropriate type at runtime) or JSON typed are not appropriate for existing lossy compressors since they cannot accurately configure these compressors.

VI. Overhead Evaluation

In this section, we run an experiment to measure the overhead of LibPressio relative to the native compressor interfaces. For this evaluation we use one 40 core Intel Xeon 6148G processor with 372 GB of RAM. We used SZ 2.1.10, ZFP 0.5.5, MGARD 0.1.0, and LibPressio 0.70.4 compiled with the default flags from Spack and the system GCC 8.3.1 compiler.

To measure the timings, we placed calls to `std::chrono::steady_clock::now()` around the invocation to the compressor's compress and decompress function which on our platform reads from a monotonic timing register on the processor. For LibPressio based usages, we place the timings around the call to `pressio_compressor_compress` and `pressio_compressor_decompress`. This means that we capture any translation overhead from LibPressio's interface relative to the native API. We execute the experiments in matched pairs – one native, one using LibPressio – to measure the overhead.

We considered 3 datasets from SDRBench [21]: ScaleLetKF, NYX, HACC choosen as some of the largest single buffers. We also consider several error bounds for each of the compressors. We apply the value range based relative error bound, which calculates the absolute error bound based on a percentage (1e-4 $\sim$ 2e-2 in our setting) of the dataset's value range. We run each configuration 30 times and compute the median and largest overhead to account for variation between runs induced by the system.

The largest overhead we measured across all configurations was 2.08% slower in a single observation, the largest median overhead across configurations was 0.47%. The distribution

[1]For two exceptions consider fpzip [7] and ZFP's reversible mode [16]

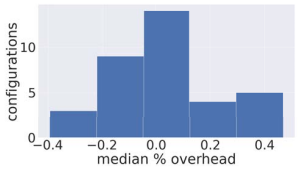

Fig. 3. Distribution of Median Percent Overheads Across all 35 configurations tested. Each configuration was run 30 times

of the median overheads is show in Figure 3. The largest variation range from 1.6% slower to 1.7% faster. We assess the statistical significance of this using a Wilcoxon sign rank test. We find there is insufficient evidence to conclude that these overheads meaningfully differ from 0 (p=.600) [2].

VII. Productivity Evaluation

There are two other columns in table I relating to developer productivity: (7) Can options of compressors be introspected? (8) Does it allow 3rd party extensions? If a compressor interface is introspect-able, it allows users to query options with types a compressor supports. This is more important the more compressor plugins that an interface provides. Users need to have some common ways of enumerating the options supported by a compressor in order to programmatically configure them. The compressor interface which allow this for all non-arbitrary types are introspect-able. If a compressor interface supports third party extensions, it allows additional implementations to be added to the interface interface without modifying the code for the interface. This is important because it allows developers to create and distribute their plugins without modifying the library so that they can be used experimentally before being released publicly.

In this section, we evaluate productivity improvements from executing lossy compression/decompression operations by LibPressio in three ways. First, we consider the number of lines of normalized client code which has long been used as an estimate of effort for developing new applications and maintenance effort required for an implementation and show a 50% to 90% reduction in lines of client code.

We assess the effort of developing or maintaining a code base supporting multiple compressors by the number of lines of code. We started with a number of use cases each supporting at least one of the leading lossy compressors: ADIOS2, Julia bindings, Python bindings, Rust bindings, command line interfaces (CLI), HDF5 filters, a configuration optimizer, and Z-Checker. We added to our list a few use cases that were requested by our collaborators: R bindings, an experimental test harness written in C++ distributed with MPI, a fuzzer which provides random inputs to the compressor to identify implementation flaws in the compressors. We then implemented each of these facilities in LibPressio to at least feature parity with the native tool. In some cases such as

[2]We choose the non-parametric Wilcoxon sign rank test to be robust to possible machine run-to-run variance which is known to possibly be large relative to the observations at this timescale. While this does not prove that there is no overhead (obviously there is some), it suggests that it is *de minimis* relative to machine noise for most uses.

TABLE II
Lines of Client Code for Various Usages, † indicates no native multi-compressor implementation exists

Task	Compressors	Lines Native	Lines LibPressio	Improvement	Relative Improvement
ADIOS2 [20]	3	744	367	377	50.67%
BindingJulia [22]	1	299	25	274	91.64%
BindingPython [23], [12] †	2	768	363	405	52.73%
BindingR	-	-	793	-	-
BindingRust [24]	1	112	34	78	69.64%
CLI [17], [16], [12] †	3	1649	756	893	54.15%
Configuration Optimizer [25]	1	4683	1869	2814	60.09%
DistributedExperiment	-	-	613	-	-
Fuzzer	-	-	24	-	-
HDF5 filter [16], [12] †	2	1469	438	1031	70.18%
Z-Checker [5]	7	3052	405	2647	86.73%

the LibPressio CLI, the LibPressio version implements many more features – for example, the libpressio CLI can compress and decompress HDF5 datasets where as the SZ, ZFP, and MGARD cannot. Additionally, in some cases the LibPressio bindings use the compressors in a more correct way such as passing dimensionality information correctly. Finally, in there of these cases – the CLI, Python bidings and HDF5 filter – the implementation do not have competing a multi-compressor implementation: in these cases, we simply sum the lines of code in each implementation. While this will over-count some code, like command line argument parsing code, it is often less than the code required to implement a correct interface. For clarity, we mark these entries in Table II. To account for differences in formatting/style, clang-format was applied to all files, and we then measured the number of lines of code example applications utilities using the `cloc` utility. In the case of larger libraries like ADIOS2 [20] or Z-checker ([5], we include only the files that directly include compressor library headers. Table II provides a summary of our results. We consistently find that LibPressio decreases the number of lines of code required between 50-90%.

VIII. Conclusions and Future Work

In this paper, we present our developed LibPressio, and describe how they can be used to enable a wide array to tools to advance the state of compression by simplifying existing work flows. We demonstrate a at least a 50% reduction in client code while maintaining insignificant overhead. These improvements brought many new features to existing compressors and new compressors to new languages and tools while simplifying the existing compressor work flows and reducing redundant work. To this end tools such as Z-Checker and ADIOS2 have integrated LibPressio for interfacing with compressors.

For future work on the interface of compressors, we plan to extend LibPressio to account for the following use cases. 1. Better support for accelerators both in plugins and core,

2. Better support for asynchrony and streaming Compression, and 3. Better support for sparse data Compression.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under grant numbers: NRT-DESE 1633608, OAC-2003709, and SHF-1910197, SHF-1617488, and CSSI-2104023/2104024.

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations - the Office of Science and the National Nuclear Security Administration,responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative.

The material was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract DE-AC02-06CH11357.

REFERENCES

[1] F. Cappello, S. Di, S. Li, X. Liang, A. M. Gok, D. Tao, C. H. Yoon, X.-C. Wu, Y. Alexeev, and F. T. Chong, "Use cases of lossy compression for floating-point data in scientific data sets," vol. 33, no. 6, pp. 1201–1220. [Online]. Available: http://journals.sagepub.com/doi/10.1177/1094342019853336

[2] P. Grosset, C. M. Biwer, J. Pulido, A. T. Mohan, A. Biswas, J. Patchett, T. L. Turton, D. H. Rogers, D. Livescu, and J. Ahrens, "Foresight: Analysis That Matters for Data Reduction," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2020, pp. 1–15.

[3] J. Kunkel, A. Novikova, E. Betke, and A. Schaare, "Toward Decoupling the Selection of Compression Algorithms from Quality Constraints," in *High Performance Computing*, J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf, Eds. Cham: Springer International Publishing, 2017, vol. 10524, pp. 3–14.

[4] R. Underwood, S. Di, J. C. Calhoun, and F. Cappello, "FRaZ: A Generic High-Fidelity Fixed-Ratio Lossy Compression Framework for Scientific Floating-point Data." IEEE.

[5] D. Tao, S. Di, H. Guo, Z. Chen, and F. Cappello, "Z-Checker: A framework for assessing lossy compression of scientific data," vol. 33, no. 2, pp. 285–303. [Online]. Available: http://journals.sagepub.com/doi/10.1177/1094342017737147

[6] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, K. Wu, and S. Klasky, "ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, Jul. 2020.

[7] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1245–1250, 2006.

[8] I. S. LLC, "ImageMagick," https://imagemagick.org/.

[9] "FFmpeg," https://www.ffmpeg.org/.

[10] "Numcodecs — numcodecs 0.9.2.dev0+dirty documentation," https://numcodecs.readthedocs.io/en/stable/.

[11] J. Tian, S. Di, K. Zhao, C. Rivera, M. H. Fulp, R. Underwood, S. Jin, X. Liang, J. Calhoun, D. Tao, and F. Cappello, "cuSZ: An Efficient GPU-Based Error-Bounded Lossy Compression Framework for Scientific Data," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. Virtual Event GA USA: ACM, Sep. 2020, pp. 3–15.

[12] S. Di, "Error-bounded Lossy Data Compressor (for floating-point/integer datasets): Disheng222/SZ." [Online]. Available: https://github.com/disheng222/SZ

[13] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the HDF5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, ser. AD '11. New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 36–47.

[14] "Libarchive - C library and command-line tools for reading and writing tar, cpio, zip, ISO, and other archive formats @ GitHub," https://www.libarchive.org/.

[15] S. Di and F. Cappello, "Fast Error-Bounded Lossy HPC Data Compression with SZ," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 730–739.

[16] P. Lindstrom, "Fixed-Rate Compressed Floating-Point Arrays," vol. 20, no. 12, pp. 2674–2683.

[17] M. Ainsworth, O. Tugluk, B. Whitney, and S. Klasky, "Multilevel techniques for compression and reduction of scientific data—the univariate case," vol. 19, no. 5-6, pp. 65–76.

[18] "MPI: A Message-Passing Interface Standard," Jun. 2015.

[19] J. Tian, S. Di, C. Zhang, X. Liang, S. Jin, D. Cheng, D. Tao, and F. Cappello, "waveSZ: A hardware-algorithm co-design of efficient lossy compression for scientific data," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, pp. 74–88. [Online]. Available: https://dl.acm.org/doi/10.1145/3332466.3374525

[20] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, K. Wu, and S. Klasky, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711019302560

[21] Scientific Data Reduction Benchmark, https://sdrbench.github.io/, online.

[22] A. Sengupta, "Zfp_jll · JuliaHub," https://juliahub.com/ui/Packages/zfp_jll/DIPUA/0.5.5+0.

[23] N. Kukreja, T. Greaves, G. Gorman, and D. Wade, "Pyzfp," 2018.

[24] C. Zapart, "Zfp-sys - crates.io: Rust Package Registry," https://crates.io/crates/zfp-sys.

[25] K. Zhao, S. Di, X. Liang, S. Li, D. Tao, Z. Chen, and F. Cappello, "Significantly Improving Lossy Compression for HPC Datasets with Second-Order Prediction and Parameter Optimization," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. Stockholm Sweden: ACM, Jun. 2020, pp. 89–100.

[26] "Libpressio," Codesign Center for Online Data Analysis and Reduction. [Online]. Available: https://github.com/CODARcode/libpressio

Appendix A
LibPressio Usage Example

A basic example of using LibPressio with error handling omitted for conciseness. Adapted from the example on [26]. It takes a buffer in memory, and compresses it with the SZ compressor using an absolute error bound of 0.5. To adapt this example for ZFP or another supported compressor, only lines 10, 20, and 21 would need to be changed.

```
1  #include <libpressio.h>
2
3  float* make_input_data();
4
5  int
6  main(int argc, char* argv[])
7  {
8    // get a handle to a compressor
9    struct pressio* library =
     ↪  pressio_instance();
10   struct pressio_compressor* compressor =
     ↪  pressio_get_compressor( library,
     ↪  "sz");
11
12   // configure metrics
13   const char* metrics[] = { "size" };
14   struct pressio_metrics* metrics_plugin
     ↪  pressio_new_metrics( library,
     ↪  metrics, 1);
15   pressio_compressor_set_metrics(
     ↪  compressor, metrics_plugin);
16
17   // configure the compressor
18   struct pressio_options* sz_options =
     ↪  pressio_compressor_get_options(
     ↪  compressor);
19
20   pressio_options_set_string( sz_options,
     ↪  "sz:error_bound_mode_str", "abs");
21   pressio_options_set_double( sz_options,
     ↪  "sz:abs_err_bound", 0.5);
22   pressio_compressor_check_options(
     ↪  compressor, sz_options);
23   pressio_compressor_set_options(
     ↪  compressor, sz_options);
24
25   // load a 300x300x300 dataset into data
     ↪  created with malloc
26   double* rawinput_data =
     ↪  make_input_data();
27   size_t dims[] = { 300, 300, 300 };
28   struct pressio_data* input_data =
     ↪  pressio_data_new_move(
     ↪  pressio_double_dtype, rawinput_data,
     ↪  3, dims, pressio_data_libc_free_fn,
     ↪  NULL);
29   // setup compressed and decompressed
     ↪  data buffers
30   struct pressio_data* compressed_data =
     ↪  pressio_data_new_empty(
     ↪  pressio_byte_dtype, 0, NULL);
31   struct pressio_data* decompressed_data =
     ↪  pressio_data_new_empty(
     ↪  pressio_double_dtype, 3, dims);
32
33   // compress and decompress the data
     pressio_compressor_compress( compressor,
     ↪  input_data, compressed_data);
34   pressio_compressor_decompress(
     ↪  compressor, compressed_data,
     ↪  decompressed_data);
35
36   // get the compression ratio
37   struct pressio_options* metric_results =
38   ↪  pressio_compressor_get_metrics_results(
     ↪  compressor);
     double compression_ratio = 0;
39   pressio_options_get_double(
40   ↪  metric_results,
     ↪  "size:compression_ratio",
     ↪  &compression_ratio);
41   printf("compression ratio: %lf\n",
     =  ↪  compression_ratio);
42
43   // free the input, decompressed, and
     ↪  compressed data
44   pressio_data_free( decompressed_data);
45   pressio_data_free( compressed_data);
46   pressio_data_free( input_data);
47
48   // free options and the library
49   pressio_options_free( sz_options);
50   pressio_options_free( metric_results);
51   pressio_compressor_release( compressor);
52   pressio_release( library);
53   return 0;
54 }
```

Appendix B
Glossary

k^{th} order error
: Metrics module that computes The size of k^{th} largest absolute value of the differences observed between the uncompressed and decompressed data.

ADIOS2
: A parallel IO, data movement, and data processing framework.

auto-correlation
: Metrics module that computes the Pearson's correlation coefficient between the data and itself shifted

by one or more "lags". For example for the points $\vec{v} = \{1, 2, 3, 4, 5\}$ with a lag of 2 would compute the correlation between $\vec{v_1} = \{1, 2, 3\}$ and $\vec{v_2} = \{3, 4, 5\}$.

AutoSFX
: An automated crystallography analysis and processing framework being developed at the Stanford Linear Accelerator Center.

Bit Grooming
: Compressor that applies various manipulation techniques to increase comparability of IEEE floating point numbers.

BLOSC
: A family of lossless compressors that have been optimized for performance.

chunking
: a meta-compressor which divides a dataset into contiguous chunks dispatching each of them to a another meta-compressor. This is useful for automatic parallelization.

CSV
: IO plugin that consumes character delimited values.

delta encoding
: Meta-compressor that applies a delta encoding a preprocessing step. Delta encoding encodes values encodes the values using adjacent differences. For example $\vec{v} = \{1, 2, 3, 4, 5\}$ would be encoded as $\vec{v} = \{1, 1, 1, 1, 1\}$.

dense tensor
: a multi-dimensional generalization of an array with a large number of non-zero values often stored contiguously in memory. In C/C++, these are stored in row-major order which has indicies that advance from slowest to fastest. In Fortran, column major order is used where indicies advance from fastest to slowest.

differences-probabilities densities function (pdf)
: Metrics module that generates an empirical probability density function of the differences between the uncompressed and decompressed values.

Digit Rounding
: Compressor that applies various rounding techniques to increase comparability of IEEE floating point numbers.

error statistics
: Metrics module that computes basic descriptive statistics using algorithms that can be computed in a single pass.

Fault Injector
: Meta-compressor that applies a sequence of single bit errors into the compressed data. Useful for implementing fuzz testing.

fpzip
: A specialized lossless and lossy compressor for IEEE floating point values.

HDF5
: IO plugin that uses the HDF5 parallel IO library and file format.

HDF5 Filter
: A feature of the HDF5 IO library that allows compression to be preformed inline to dataset access. Supports plugins to support different compressors.

Image Magick
: A extensive library for image manipulation and compression.

Iota
: A IO plugin that generates synthetic data using C++'s `std::iota` which fills a buffer with sequentially increasing values.

Kolmogorov-Smirnov (KS) Test for Goodness of Fit
: Metrics module that compute a non-parametric statistical hypothesis test which test the hypothesis that two distributions are two samples are drawn from the same distribution that operates by determining the largest difference between the empirical cumulative density function.

Kullback-Liebler (KL) Divergence
: A metrics module that computes A measure of relative entropy from one distribution to another. It is defined as $D(P||Q)_k l = \sum_{x \in X} P(x) \log\left(\frac{P(x)}{Q(x)}\right)$. It is used in information theory and machine learning.

LibPressio-Fuzz
: A Fuzzer developed for this paper that use LibPressio and Clang/LLVM's libfuzzer.

LibPressio-Opt
: A meta-compressor that implements an optimizer that can be used to determine an optimal configuration. Previous version of this were named FRaZ and OptZConfig..

LibPressio-Tools
: A set of tools developed for this paper that use LibPressio to implement a command line interface for LibPressio compressors and meta-compressors.

linear quantization
: Meta-compressor that preforms linear quantization. Quantization is a transformation that maps a contiguous domain (i.e. floats) to a countable domain (i.e. integers). Linear Quanitization, does so with a mapping like $Q(x) = \lfloor \frac{x-m}{\Delta} \rfloor$ where x is the value, $Q(x)$ is the quantized value, Δ is a scaling factor, and m is some centering term. Quantization is often used in lossy compression because coutable domains often have lower entropy than contiguous domains and are thereby more compress-able.

Many Dependent
: A Meta-compressor that implements a parallel pipeline that does the following. The first buffer is operated upon and metrics are gathered from it. Metrics from the first buffer are passed to one or more compression that are done in parallel as configuration options. As each buffer finishes, the value of the latest indexed buffer to complete is stored to be passed to future invocations. This is used for forwarding a guess for a configuration to subsequent time steps.

Many Independent
: A meta-compressor that implements a embarrassingly parallel compression of multiple data-sets.

masked
: Metrics module that removes specified points from a data set prior to computing another metric.

Meta-Compressor
: A concept within LibPressio. Meta-Compressors implement the compressor interface, but are not compressors. Examples may include pre/post processing steps, parallel run-times, optimizer, etc....

MGARD
: A multi-grid based error bounded lossy compressor.

mmap
: IO plugin that uses the UNIX system call `mmap` that maps the contents of a file or memory of a device into memory via the virtual memory of a process.

NumPY
: IO plugin for the custom file format used by the python numeric library NumPY for storing n-dimensional arrays.

Pearson's Correlation
: Metrics module that computes Pearson's Correlation Coefficient (often denoted r) measures the strength of a linear relationship between two values .

PETSc
: A IO plugin that reads file created by PETSC, the "Portable, Extensible Toolkit for Scientific Computation".

posix
: IO plugin that uses the POSIX functions `read` and `write` to read in an array in a native data format.

R
: Metrics module that uses the scripting language R that is specialized in statistical analysis.

Random Error Injector
: A meta compressor that applies randomly generated noise to each element of the input dataset according to some specified distribution.

Region of Interest
: Metrics module that Computes the arithmetic mean of a region of interest within a dataset.

resize
: A meta-compressor which modifies the dimensions of the data without modifying the values. This is useful for compressors which sometimes benefit from being told the data shape is different than it actually is – i.e. ZFP if you have a 3d dataset that is $A \times B \times 1$ so you can treat it as 2D.

sample
: A meta-compressor which applies data-sampling techniques such as uniform sampling with and without replacement prior to compression.

select
: IO plugin that selects a sub-region of an input dataset read in by another IO plugin for compression/analysis.

Spatial Error
: The percentage of elements of a dataset that exceed some specified threshold.

switch
: A meta-compressor which allows runtime switching between different compressors based on a configuration setting. This is useful because it allow tools like LibPressio-Opt to select between multiple different compressors types dynamically.

SZ
: a prediction based error bounded lossy compressor.

SZ-OMP
: the parallel CPU version of the SZ prediction based error bounded lossy compressor.

SZ-Threadsafe
: the threadsafe serial version of the SZ prediction based error bounded lossy compressor.

the Feature Detection Toolkit (FTK)
: Metrics module that uses the library FTK that tracks features such as maxima, minima, an saddle points in data between time-steps of a simulation.

transpose
: A meta-compressor which applies a multi-dimensional abstraction of a transpose to the data prior to compression.

tthresh
: A compressor that uses the principles of singular value decomposition to compress data.

vecSZ
: A version of SZ optimized to leverage SIMD vector instructions.

ZFP
: A transform based error bounded lossy compressor.