An Intelligent, Adaptive, and Flexible Data Compression Framework

Hariharan Devarajan, Anthony Kougkas, and Xian-He Sun
Illinois Institute of Technology, Department of Computer Science
{hdevarajan, akougkas}@hawk.iit.edu, sun@iit.edu

***Abstract*—The data explosion phenomenon in modern applications causes tremendous stress on storage systems. Developers use data compression, a size-reduction technique, to address this issue. However, each compression library exhibits different strengths and weaknesses when considering the input data type and format. We present Ares, an intelligent, adaptive, and flexible compression framework which can dynamically choose a compression library for a given input data based on the type of the workload and provides an appropriate infrastructure to users to fine-tune the chosen library. Ares is a modular framework which unifies several compression libraries while allowing the addition of more compression libraries by the user. Ares is a unified compression engine that abstracts the complexity of using different compression libraries for each workload. Evaluation results show that under real-world applications, from both scientific and Cloud domains, Ares performed 2-6x faster than competitive solutions with a low cost of additional data analysis (i.e., overheads around 10%) and up to 10x faster against a baseline of no compression at all.**

I. Introduction

Modern applications produce and consume data at an unprecedented rate [1]. The proliferation of data allows advances in sciences such as climate simulation [2], molecular biology [3], and many more. Also, industry gains competitive advantages by leveraging these data. However, these capabilities do not come for free. One of the significant challenges big data applications face is the efficiency of storage infrastructures. Scientists have proposed many techniques to ease this stress, with data subsampling [4], moving computation to data [5], and data transformations [6] being some examples where data is reduced to minimize I/O cost. One such technique that has been adapted to reduce the data size in many modern applications is *data compression*.

Compression techniques are well-explored in the literature. There are two categories of compression techniques, lossy and lossless algorithms. The former loses information to efficiently reduce the data and hence, prohibiting data reconstruction whereas the latter reduces the data while having a mechanism for reconstruction. The lossless algorithms are a defacto in the scientific and cloud applications for data-reduction [7]. They are used in several situations such as archival [8], data-movement [9], reducing memory utilization [10], etc. Furthermore, there are two classes of lossless compression, general-purpose and specialized algorithms. Some examples of general-purpose compression libraries include Bzip, Zlib, and 7z. Even though these algorithms promise good performance, they do not exploit data representation's nuances. Hence, researchers developed more specialized algorithms such as Snappy, SPDP [11] , LZO, and others. These algorithms show a great promise by boosting the overall application performance by minimizing the data footprint.

Although there have been several domain-specific developments in compression libraries, in this work, we identify several challenges when performing data-reduction using compression: a) data-dependency, due to the specialization of each library towards a type of data, it often is not general enough for other cases. Even if one chooses a library, most applications use different types of data and hence, using just one library does not yield the best performance. b) library-choice, different libraries have different strengths and weaknesses and often choosing the correct library for a use-case is hard. Even within the application, various parts of the application might have different compression needs. For instance, archival storage needs high compressibility whereas data-sharing between processes require high compression and decompression speed. c) API and usability: each compression library has its own set of parameters and APIs. It is often difficult to transition to or adopt a new library. In this study, we highlight that there is no "one compression for all" approach. Practically, no compression algorithm can offer the best performance for all types of data, file formats, or application requirements. Due to these challenges, there is a need for an intelligent framework which can seamlessly unify multiple compression libraries. Furthermore, based on the given scenario, it should have the ability to dynamically choose the "best" compression algorithm.

In this work, we present Ares: a dynamic, adaptive, and flexible compression framework that addresses the above challenges. Ares intelligently chooses the appropriate compression library for a given data type and format. Ares uses several components to achieve its goals. These include: an *input analyzer* which predicts the data type (i.e., integer, float, character, etc.,) and the data format (i.e., binary, textual, HDF5, CSV, etc.,); a *main engine* which decides the best compression library for the given situation (i.e., data-type, data-format and workload characteristics); a *library pool* which unifies and manages all the compression libraries; and an *output manager* that decorates the compressed data with Ares metadata and delivers them to the destination. Ares is a modular framework which allows developers to offload the complexity of manually choosing the "best" compression algorithm for the task-at-hand to the framework. The contributions of this work are:

1) Performing a comprehensive study of the performance of several compression algorithms for different data types and formats, and workload characteristics (Section III).
2) Showing the benefits of an intelligent, adaptive, and flexible approach to data compression by designing and implementing Ares framework (Section IV).

II. Background and Motivation

A. Data-intensive Applications and Data Compression

Modern applications, in a variety of domains, have become data-intensive. In fact, the Vs [12] broadly characterize the big data as follows: *volume*, *velocity*, *variety*, and *veracity*. Each one of those Vs dictates a growing need to manage, if not control, the explosion of data growth. For instance, the amount of data generated from Square Kilometer Array (SKA) [13] is estimated to reach 25 PiB/sec, which can drain the entire memory of supercomputers extremely fast and overwhelm existing storage solutions. In the business world things are even worse with the data production estimated to reach 44 zettabytes by 2020 [14], [15]. This data growth is driven by various sources such as mobile, sensors, video, audio, and social networks [16]. This unprecedented data volume poses significant challenges to both computing and storage systems. Collecting massive amounts of data does not result in better knowledge extraction. Data must remain consolidated, cleansed, consistent, and current to make scientific discovery feasible. This data explosion has led to several techniques to manage, process, and store massive datasets. Data partitioning techniques [17], data filtering [18], and data streaming [19] are some examples where data can be reduced before getting processed. One such technology that has been around for a long time and which acquired a new sense of importance in this data-driven world is *data compression*.

Various data compression techniques have been explored to reduce the stress that data-intensive workloads pose to memory and storage systems. These techniques can be broadly categorized as *lossy* and *lossless*. The former aim to achieve performance at the cost of losing a part of input data and so are irreversible. On the other hand, lossless data compression allows perfect reconstruction of the original data. Mission-critical workloads, in both scientific and cloud domains, cannot tolerate data loss, and hence, they use lossless data compression algorithms [20] including Run Length Encoding (RLE) [21], Huffman encoding [22], Shannon-Fano [23], Rice [24], Burrows-Wheeler [25], and Lempel-Ziv (LZ77 and LZ78) [26]. Several general-purpose data compression techniques (e.g., gzip, bzip2, 7z, etc.,) do not consider the nature of data they compress, which may lead to significant performance penalties or missed opportunities. Type-specific data compression techniques such as GTZ [27], Genoox [28] and MAFCO [29] try to alleviate this issue by being specific to the data type being compressed. These techniques have been proven to be both fast and highly compressible than general-purpose data compressors. Some instances showcase the importance of choosing the right compressor for a given use-case. Pied Piper [30], a compression algorithm, reduces data movement and storage cost for PiedPiperCoin by transporting highly-compressed complex data files in seconds. SPDP [11] aims at achieving best compression ratio for floating point data demonstrating 30% higher performance than a typical bzip.

B. Motivating Examples

Data compression techniques are mostly used to minimize the data footprint and applications use it in a variety of workloads. For instance, NASA uses sentiment analysis [31] to identify human trafficking in web data. During sentiment analysis, a web crawler collects various textual (e.g., xml, json, csv) and visual (e.g., images, videos) cues from the web and passes the data to the main sentiment model which in turn produces a set of intermediate data describing the correlations between relevancy and sentiment. Finally, the output analyzer merges the intermediate data to produce the final result. Another example is weather forecasting applications such as WRF [32]. The entire workflow starts by gathering data (e.g., temperature, pressure, coordinates, timestamps, and area name) from sensors and other environmental instruments. These data are stored in various formats ranging from flat files (e.g., binary, csv) to specialized ones (e.g., pNetCDF, HDF5, MOAB etc.,). The workflow continues by converting the collected data to different formats for analysis. The last step is to produce the final results that may be further analyzed by different programming frameworks. Modern applications demonstrate complex workflows where information can be accessed, shared, and stored in a variety of data types and formats.

In such complicated workflows various compression priorities may arise. Data compression is characterized by three main metrics: compression speed, decompression speed, and compression ratio. Each compression algorithm performs differently in each category. The performance of various compression libraries depends on the data type and format (e.g., LZ4 [33] performs better for integers whereas QuickLZ [33] for floats). Further, each phase of a workflow may require different prioritization over the above metrics. For instance, input data (i.e., from sensors or static storage silos) must be quickly decompressed, intermediate results for process-to-process data sharing must be compressed fast, and final results sent to the archival storage system must be heavily compressed to minimize the footprint of data-in-rest. This motivates us to realize a new compression framework which can: a) unify all compression libraries, b) intelligently select the appropriate compression library based on the workflow, and c) utilize the best compression library for a given workload.

III. Approach

Based on our motivation we see that different compression libraries excel in different situations (i.e., data-type, data-format, and workload priority). We empirically evaluate a wide selection of compression libraries through several comprehensive benchmarks to understand their characteristics.

Benchmark Configurations: all tests run on a single core on a Intel Xeon(R) CPU E5-2670 v3 @ 2.3 GHz, with 128 GB RAM, and a local 200GB HDD. We wrote our

Metric	Data Type	brotli	bsc	bzip2	lz4	lzma	lzo	huffman	pithy	quicklz	snappy	zlib	Best
Compression Ratio	char	2.99	4.78	3.33	1.75	3.77	2.03	2.60	1.92	1.91	1.77	2.71	bsc
	integer	2.16	2.39	2.14	1.42	2.53	1.53	1.94	1.49	1.53	1.44	1.97	lzma
	sorted int	3.00	4.47	3.42	1.92	3.51	2.22	2.62	2.20	1.83	1.89	2.69	bsc
	float	14.88	9.62	8.09	2.75	19.90	2.53	4.59	2.45	3.32	2.42	5.00	lzma
	double	14.49	17.82	11.45	6.06	16.70	5.55	8.42	6.60	5.45	5.46	9.93	bsc
Compression Speed (MB/s)	char	25.66	6.83	13.06	237.27	1.70	118.01	35.52	195.64	195.45	186.28	31.40	lz4
	integer	18.96	8.40	12.48	255.57	2.77	97.44	32.53	202.34	202.41	198.13	27.32	lz4
	sorted int	26.16	8.75	12.97	273.78	2.74	135.89	38.46	269.15	236.91	267.47	44.57	lz4
	float	36.28	16.59	19.14	320.09	2.78	273.17	60.09	342.06	381.52	331.55	60.31	quicklz
	double	70.06	14.60	8.02	436.27	6.99	312.26	96.56	434.61	379.60	400.34	105.23	lz4
Decompression Speed (MB/s)	char	339.52	11.16	32.74	539.73	70.61	287.64	259.05	495.60	275.99	452.94	272.94	lz4
	integer	201.55	10.76	27.29	518.79	37.92	264.22	164.16	456.14	229.00	449.28	217.57	lz4
	sorted int	352.92	12.52	33.75	582.23	56.27	428.23	246.67	600.78	279.19	541.34	306.33	pithy
	float	461.05	34.50	72.63	590.17	143.34	579.03	410.52	594.82	429.97	560.21	491.02	pithy
	double	1014.62	19.41	53.84	862.05	264.44	615.72	395.81	864.72	578.98	763.05	737.25	brotli

(a) Performance by data type

Metric	Data Format	brotli	bsc	bzip2	lz4	lzma	lzo	huffman	pithy	quicklz	snappy	zlib	Best
Compression Ratio	POSIX	2.99	4.78	3.33	1.75	3.77	2.03	2.60	1.92	1.91	1.77	2.71	bsc
	HDF5	2.99	4.48	4.03	1.83	3.62	2.16	2.73	1.97	2.09	1.84	2.70	bsc
	CSV	4.07	4.49	3.89	2.09	4.45	2.34	3.13	2.41	2.31	2.09	3.14	bsc
	JSON	3.17	3.31	3.23	2.06	2.22	2.32	2.85	2.12	2.17	2.08	3.07	bsc
	XML	9.80	13.85	11.81	4.35	11.04	4.57	6.53	4.83	4.75	4.09	7.32	bsc
	AVRO	3.31	3.16	2.96	2.09	4.06	2.12	2.83	2.08	2.07	2.04	2.93	lzma
Compression Speed (MB/s)	POSIX	20.53	5.46	10.45	189.82	1.36	94.41	28.42	156.51	156.36	149.02	25.12	lz4
	HDF5	18.90	8.26	14.69	232.13	2.04	138.14	36.73	188.09	245.26	191.11	22.98	quicklz
	CSV	26.85	9.72	10.08	239.56	4.50	100.52	36.50	204.91	192.47	206.42	35.27	lz4
	JSON	20.72	5.31	8.12	284.40	2.26	90.68	40.19	159.07	166.11	213.30	37.69	lz4
	XML	52.28	16.24	7.41	300.53	5.64	198.69	60.10	293.26	270.72	256.63	65.34	lz4
	AVRO	21.69	6.08	8.72	275.20	2.61	95.66	40.74	182.34	174.70	204.02	25.84	lz4
Decompression Speed (MB/s)	POSIX	271.62	8.93	26.19	431.78	56.49	230.11	207.24	396.48	220.79	362.35	218.35	lz4
	HDF5	228.00	12.58	35.60	460.63	44.33	326.21	213.11	415.14	226.20	394.66	231.63	lz4
	CSV	293.66	15.54	28.66	484.31	53.76	267.69	220.25	454.33	240.31	428.18	235.97	lz4
	JSON	213.30	7.95	30.47	552.08	36.74	312.84	208.56	586.58	257.13	536.30	208.56	pithy
	XML	564.84	28.70	43.30	592.92	142.37	408.59	247.94	607.95	413.65	532.26	477.92	pithy
	AVRO	177.90	7.94	30.42	470.56	40.52	283.66	183.49	530.45	255.92	494.49	198.47	pithy

(b) Performance by data format

Priority in percentage			Workload Examples	Top library based on data-type				
Compression Speed	Decompression Speed	Compression Ratio		char	int	sorted int	float	double
100%	0	0	Asynchronous communication	lz4	lz4	lz4	quicklz	lz4
0	100%	0	Multicast in network	lz4	lz4	pithy	pithy	brotli
0	0	100%	Archival store	bsc	lzma	bsc	lzma	bsc
50%	50%	0	Synchronous communication	lz4	lz4	pithy	pithy	lz4
0	50%	50%	Dequeue operation	lz4	lz4	lz4	quicklz	pithy
50%	0	50%	Queue operation	lz4	lz4	lz4	pithy	lz4
33%	33%	33%	Mixed workload (balanced)	lz4	lz4	pithy	pithy	pithy

(c) Performance based on priorities

Fig. 1. Compression Libraries' Evaluation based on Data Type and Format

synthetic benchmarks to test the three dimensions of our observation: *data-type*, *data-format*, and *workload priority*. The benchmark first generates 8 GB of datasets, one for each test configuration (i.e., various data types and formats), and then, compresses and decompresses it once. We measure compression/decompression speed (CS, DS) in MB/seconds and compression ratio (CR) as original by compressed dataset size. We test the following implementations of lossless compression algorithms [33]: bzip2, zlib, huffman, brotli, bsc, lzma, lz4, lzo, pithy, snappy, and quicklz. We collect results from a total over 1000 test cases and report the average values over five repetitions. Due to space limitations, we only present the highlights of our comprehensive benchmarks. More results can be found in the technical report [34]. Figure 1 shows our findings.

Data type: In this first test, we evaluate how different data types affect the performance of the compression libraries. We load the pre-generated dataset into a memory buffer and compress/decompress it using each library's API. We only capture the compression/decompression time excluding loading times. The tested data-types are: *characters*, *integers* along with their modifiers (short, long, signed, unsigned), *sorted integers*, *floating point*, and *double floating points*. The selection of the above data-types is based on popular programming languages used today. Figure 1(a) summarizes the results of this data-type benchmark. We can make the following observations. First, compression libraries exhibit different performance for different data-types. For instance, `brotli` compresses integers with CS of 18.96 MB/s achieving CR of 2.16x. However, the same library compresses doubles with CS of 71 MB/s for a much higher CR of 14.49x. Second, there is substantial performance variability between each library for the same data-type, even when targeting the same CR. As an example, `quicklz` compresses integers with CS of 202.41 MB/s for a CR of 1.53x whereas `lzo` achieved the same CR with CS of 97.44 MB/s making it 2x slower than `quicklz`. Third, some libraries are optimized for sorted data. For example, `lzma` library achieved the highest CR for integers, but if data is sorted then `bsc` takes the lead. Lastly, it is worthy to note that achieving the best compressibility (i.e., maximum CR) is not the ultimate metric. There is a direct relationship between the time spent on compression and the achieved CR, and hence, workload priorities could affect the choice of an ideal library.

Data format: In this test, we evaluate how different data formats affect the performance of compression libraries. We pass the pre-generated dataset files to the library's executable for compression/decompression. We capture the overall time each library took to compress/decompress the input files that include 8 GB of character data. The tested formats are: a) *binary data* (e.g., POSIX), b) *scientific data* (e.g., HDF5), c) *textual data* (e.g., csv, json, xml), and d) *columnar data* (e.g., Avro, Parquet). The selection of the above formats represents typically used data representations. From Figure 1(b) we can make the following observations. First, the data format affects the performance of a compression library. For instance, `lzo` compresses and decompresses the HDF5 file with CS of 138.14 MB/s and DS of 326.21 MB/s respectively, but for the AVRO file, the performance drops to CS 95.66 MB/s and DS 283 MB/s. In both cases `lzo` library achieves similar CR of about 2.14x. Second, different compression libraries exhibit different performance for the same data-format. This is apparent in the results where for characters represented in a binary format (i.e., POSIX file), `pithy` compresses and decompresses the data with CS of 156.51 MB/s and DS of 396.48 MB/s, and achieves a CR of 1.92x, whereas, `quicklz`, for the same data, achieves similar CR and CS but only half the DS in comparison to `pithy`. Lastly, we observe a direct relationship between compression speed and achieved compression ratio i.e., heavier compression takes more time.

Workload priority: In this part, we analyze how different workload priorities are affected by the choice of a compression library. We normalize the data from the previous tests and create a weighted sum of the compression metrics (CS, DS, and

CR). The formula for this weighted sum is given in equation 1.

$$Y_{d,f} = \sum_{i=1}^{m} W_i * X_i \quad (1)$$

where Y is the final score, d is the data-type, f is the data-format, m is the metrics (i.e., CS, DS and CR), W_i is the weight of i^{th} metric, and X_i is the normalized value of i^{th} metric. Figure 1(c) shows some examples of how the workload dictates the prioritization of a certain metric and which library can offer the best overall performance based on equation 1. The results reflect data stored in a binary format along with the ideal library per data-type. We can observe the following: first, when compressing character data, if we only prioritize CS, `lz4` has the best performance. However, if we need to compress the data for archival purposes, we would need to maximize CR, and thus, we should use the `bsc` library which offers 4x more compressibility. This shows that depending on the workload type and the set compression metric priority, different libraries will perform differently. This is an important observation since there is no "one compression library for all purposes" approach. There is observed performance variability between libraries, and each workload can benefit from an intelligent, dynamic compression framework.

IV. Ares

A. Design

Based on our findings in Section III, we introduce Ares, a new intelligent, adaptive, and flexible data compression framework. Ares takes into consideration the input data type and format as well as the user intention, expressed in compression metric priorities based on the workload characteristics, to offer an overall easy-to-use, high-performance data compression solution. Moreover, Ares is a flexible framework which can efficiently utilize a plethora of compression libraries to achieve its objectives. Our design is modular allowing further additions of compression libraries in the future. We guide the design of Ares with the following principles:

Intelligent: the framework should be able to learn and adjust itself to the input data compression characteristics. The input data may be of any type and in any format. The framework should be able to identify its nature efficiently and make appropriate decisions to lead to a better compression performance.

Adaptive: the framework should be able to reconfigure itself, dynamically, to various compression needs of an application. The application may consist of multiple phases in its workflow, with each phase having a different data compression requirement. The framework should transparently manage this diversity to the end-user.

Flexible: the framework should be able to unify all interfaces of the compression libraries it contains. Each compression library might have its API requirements. The presence of multiple libraries, with their API requirements, should be abstracted from the end-user. Additionally, the framework should be able to incorporate new libraries and benchmarking

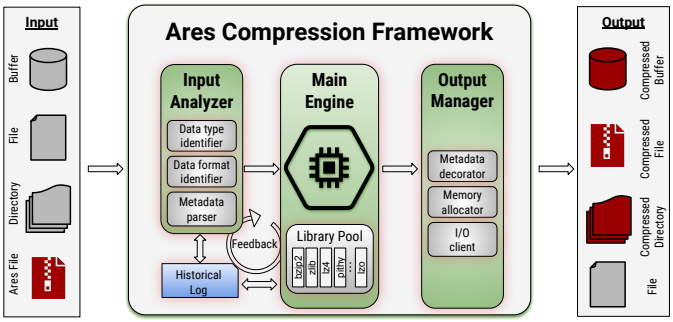


Fig. 2. High-Level Architecture

information dynamically. This feature allows the growing of the framework to support new libraries.

The primary objective of Ares is to *transparently abstract the complexity of choosing the best compression technique for a given workload dynamically*. To achieve this, Ares aims for: 1) Efficient analysis of input data, 2) Transparent management of a collection of compression libraries. 3) Flexible application interface. 4) Adaptive compression requirements.

B. Architecture

The core of Ares architecture is plug and play and can be seen in Figure 2. The framework is a middleware library that encapsulates several compression libraries abstracting their complexity from the user. Applications can use Ares either as a tool (i.e., via CLI) or as a library (i.e., via Ares API). In both cases, the flow within Ares is the same. First, Ares analyzes the input data to identify data types and formats that are involved. The input to Ares can be a memory buffer, a file, a directory, or a previously compressed file (e.g., `file.ares`). It then passes the analysis results to the main engine that decides which compression library is best for the given situation (passed as flags to Ares). Then, based on the decision, Ares utilizes a library pool, that includes pre-compiled compression libraries (i.e., 11 such libraries in our prototype), to perform the compression/decompression operation. Lastly, Ares decorates the compressed data with its metadata (i.e., for future decompression) and writes the final output as *.ares* files to the disk.

Ares framework is comprised of three components: *input analyzer*, *main engine*, and *output manager*. These components work together to realize Ares' design principles and objectives.

Input Analyzer: this component is responsible for describing the input data, inferring its type and format. The accuracy of this information is performance critical for the framework. The analyzer aims to create the best possible data inference. It uses a hybrid approach to determine the data-type and data-format of the input data. The approach is a combination of static analysis and a dynamic feedback mechanism. The main engine updates the log with the actual performance results which are processed by the analyzer to identify the difference between expected and actual measurements. This feedback allows the analyzer to improve over time by matching its prediction with the historical results. For data-type detection,

we classify the data into binary data and descriptive data. To extract data-type from binary data we use static binary decoding techniques [35] whereas, for descriptive data (e.g., high-level libraries such as HDF5, NetCDF, Parquet or Avro), the data itself contains the data-type information. For data-format extraction, if the extension is known, then we can determine the data format (e.g., *.h5* for HDF5 files, *.xml* for XML files, etc.,). Otherwise, we utilize the `mime-type` information within each file to identify the data-format. The description of input data allows the main engine to make better decisions on the choice of compression library. The major challenge this component faces is the type inference of binary data. As shown in the binary decoding techniques [35], the decoding is an approximation and could lead to erroneous decisions. However, Ares can learn as it compresses more and more data by the feedback loop between the analyzer and its main engine. Lastly, for compressed input data, the analyzer first extracts the metadata and passes the information (e.g., if data were previously compressed by Ares using *bzip2* or *gzip*) to the main engine for decompression.

Main Engine: this component is responsible for choosing the best compression engine for the given description of data and users priorities. During bootstrap of Ares framework, the engine loads the initial seed data, from the comprehensive benchmarking done in Section III, to build a forest of decision trees. This seed is a JSON file which can be configurable by the user. This data structure makes the engine lightweight and easily tunable. For instance, given a workload type (i.e., compression metric priority), the main engine selects the appropriate decision tree from the forest in constant time, and, traverses the tree until it reaches the right decision. The complexity of this operation is insignificant, and in the order of $O(log(n+m))$ where n is the data-type, and m is the data-format. The main challenge that the engine faces is that the effectiveness of its decision is directly related to the accuracy of the analysis of the input data. For instance, if the input analyzer passes a buffer of integers (instead of floats) due to erroneous identification, the main engine might not select the best compression library and, therefore, would not achieve the best compression performance. However, this can be mitigated by Ares' learning capabilities through the feedback mechanism. The main engine of our prototype implementation is equipped with the eleven compression libraries tested in Section III. It unifies their interfaces by using a template pattern and two general functions: a) `compress()`, b) `decompress()`. Each library specific implementation is defined in a client class which implements the above template. To make the library pool extensible and modular, we use adapter pattern and factory pattern so that we can decide the implementation of the compression clients at runtime. This modular design also allows Ares to easily add new compression libraries to the pool and use them at runtime. Modular design and adapter pattern allow Ares to link new compression libraries easily and allow conditional compilation. Ares' main engine implements a light-weight mechanism to switch libraries at runtime boosting the overall performance of the framework.

Output Manager: this component has three primary responsibilities. First, it decorates the compressed data with some additional information, in the form of headers, regarding the compression library used. This allows Ares to decompress data, previously processed by the framework, quickly. We kept the size of this metadata minimal with a total size of 8 bytes per data-type (e.g., if a dataset has 4 data-types, it uses 32 bytes for metadata). In case of compressing a directory, Ares flattens its structure and adds the appropriate metadata in the form of a directed acyclic graph (DAG). The benefit of small metadata footprint is that it keeps overheads minimal relative to competitive solutions (e.g., bzip2 adds 50 bytes). Second, the output manager is also responsible for checking the correctness of the Ares format using parity checking that ensures accurate data encoding. Lastly, the output manager performs the final I/O of the compressed data. It incorporates several memory allocators (i.e., `malloc` and `tcmalloc`), and I/O clients (i.e., POSIX calls to `fwrite()` and `fsync()`) to pass the processed data to the user. We enabled known optimizations to Ares' I/O capabilities by implementing clients for both local file systems (e.g., ext4, xfs, etc.,) as well as parallel file systems (i.e., Lustre, GPFS). The parallel I/O client can also use MPI-IO, by enabling a flag, to further boost the performance.

C. Implementation Details

Ares framework prototype implementation is written in C++ in around 1K lines of code [1]. Additionally, Ares contains wrappers for C/C++ and JAVA applications supporting a wide range of applications ranging from scientific computing to MapReduce. Ares framework supports two modes: a) Ares tool: which compresses files and folders passed to its executable via the command line and can be used in batch files or scripts, and b) Ares library: which can be simply linked to an application (e.g., using `LDFLAGS` or `LDPRELOAD`) to compress/decompress data in buffers, files, or folders using the Ares API. The Ares library exposes a compress/decompress API with either buffer, file, or folder as an input as well as the ability to pass the user-defined priorities as flags.

V. Evaluation

A. Methodology

Testbed and Configurations: all experiments were conducted on a bare metal configuration on Chameleon systems [36]. Each node has a dual Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz (i.e., a total of 48 cores per node), 128 GB RAM, 10 Gbit Ethernet, and a local 200GB HDD. We setup two kinds of hardware configurations: scientific and cloud computing. The scientific setup uses 32 client nodes to run the application and 8 parallel file system (PFS) servers running OrangeFS 2.9.8 [33]. For the cloud setup, we build a 40-node Hadoop cluster. We set the replication factor to 1 to minimize the additional data movements (i.e., no need for fault tolerance). The input data for both setups are pre-loaded into the storage

[1] Ares source code can be found in https://bitbucket.org/hdevarajan/ares/

systems. The cluster OS is Ubuntu 16.04, the MPI version is MPICH 3.2, and the Apache Hadoop distribution is 2.9.2.

Workloads and Datasets: to evaluate Ares framework, we use a collection of workloads spanning from our synthetic benchmarks to scientific and cloud applications. Specifically, our benchmark can generate data with compound data-types and a set of diverse data formats. Once data are generated, the benchmark compares Ares performance to other state-of-the-art data compression libraries. The benchmark uses two distinct datasets as input: a) a collection of binary, netCDF, and HDF5 files representing *scientific data*, b) a collection of textual data formats such as CSV, JSON, and XML representing *cloud data*. We measure the time to perform compression/decompression in seconds, excluding the data generation time. We also use a collection of applications to evaluate the Ares framework in real situations. Specifically, for science applications we use: "Vector Particle-In-Cell" (VPIC) [33], a general purpose simulation code for modeling kinetic plasmas in spatial multi-dimensions, and "Hardware Accelerated Cosmology Code" (HACC) [33], a cosmological simulation that studies the formation of structure in collision-less fluids under the influence of gravity in an expanding universe. Both of these simulations periodically produce output files that are stored in the PFS. We used 16 timesteps for both simulations resulting in total I/O of 1.5 TB. This data gets compressed to minimize the data footprint and boost the overall performance. We only report compression/decompression time and I/O time excluding simulation time. For Cloud workloads, we run popular MapReduce kernels: *sort* [33], and *word count* [33]. The sorting application uses the MapReduce framework to sort the input data and store them into the output directory. The inputs and outputs should be Sequence files where the keys and values are BytesWritable. We generate a dataset of 1.5 TB of binary integers for this application. The word count application reads text files and counts how often words occur [33]. The input is text files, and the output is also text files, each line of which contains a word and the count of how often it occurred, separated by a tab. A possible optimization reduces the amount of data sent across the network by combining each word into a single record. The input dataset for this application data is a 1.5 TB Wikipedia article data. For both these applications, we store the data into HDFS file systems.

B. Experimental Results

In our evaluation, we compare Ares' performance characteristics with the following high-performing compression libraries: snappy, bzip2, lz4, quicklz, and bsc. All tests are repeated five times, and we report the average. It is to be noted that in the figures we use "CT" for compression time, "DT" for decompression time, and "CR" for compression ratio.

*1) **Overheads and Resource Utilization**:* every compression/decompression library demonstrates overheads in the form of additional CPU cycles and memory space with the promise of minimizing working datasets. Ares adds an extra overhead of analyzing the input data to achieve a balanced

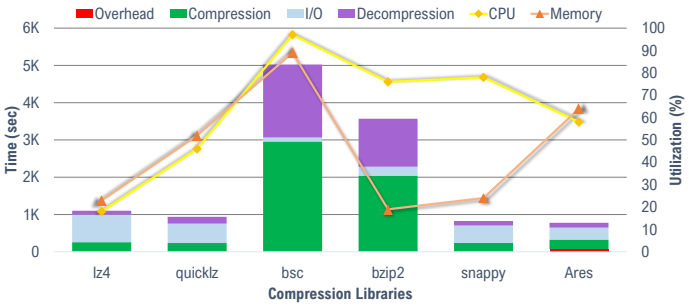
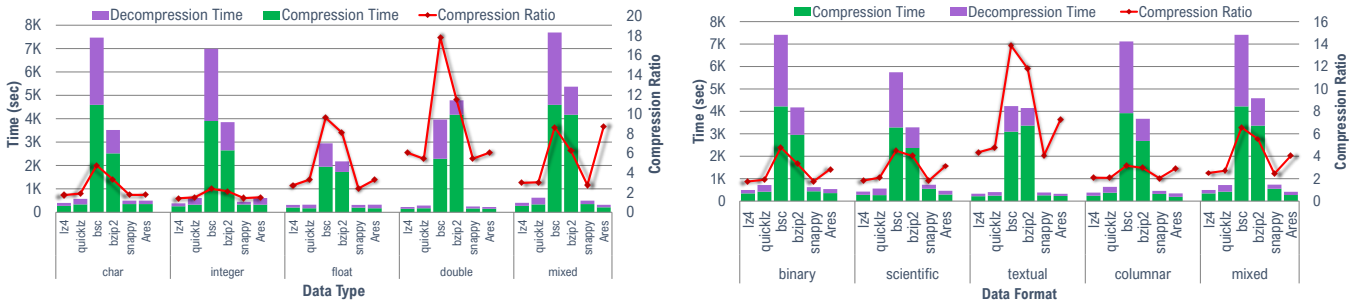


Fig. 3. Ares Overheads and Resource Utilization

CT and CR. This analysis allows Ares to choose the best library for the given scenario (i.e., data type and format). In this test, we use 64GB input data organized in an HDF5 file with four datasets: characters, integers, sorted integers, and doubles. The workflow of this test is: read input data from the file system, compress data, write compressed data back, read compressed data, and lastly, decompress the data. We measure the time spent in each of the above phases, and we compound all I/O operations in one I/O time, excluding the time to read the initial data since it is the same between all tested libraries. It is to be noted that I/O time is directly related to the achieved CR since fewer data are read/written due to compression. Lastly, Ares' analysis of input data is depicted as "Overhead". For system monitoring, we use Intel's Performance Analysis Tool (PAT) tool [33] to capture the CPU and memory utilization. Figure 3 shows the results. We can observe that each of the tested libraries demonstrates different overheads. For instance, `lz4`, `quicklz`, and `snappy` all achieved similar time for CT, I/O, and DT but with different system utilization (e.g., `snappy` is CPU intensive with low memory footprint). In contrast, `bsc` offers the highest CR of 8.6x but also is the slowest library with high overheads of more than 90% CPU and memory utilization. `bzip2` has a lower memory footprint but maintains high CPU utilization for a CR of 6.2x. On the other hand, Ares balances the trade-off between CT/DT and CR by analyzing the input data. This additional overhead is only about 10% of the overall time (i.e., Ares spent 74 sec to perform the data type and format detection). Even with this extra overhead, Ares performed all operations faster than all libraries and achieved the best overall time. Specifically, Ares is 6.5x faster than `bsc`, 4.6x faster than `bzip2`, 5-40% faster than `lz4`, `quicklz`, and `snappy` while hitting a 58% CPU and a 64% memory utilization. Ares performance and overheads are a result of Ares' ability to perform those operations using a collection of libraries combining the strengths of each one of them.

*2) **Compression/Decompression Intelligence**:* in this test, we quantitatively evaluate the importance of data compression based on data type and format, as discussed in Section III. We first evaluate how different data types affect the performance of compression libraries, and we compare Ares to several libraries. The benchmark places 64 GB of input data to a buffer in memory. We tested five configurations of this buffer: characters, integers, floats, doubles, and a mixed case with all above types. This test starts by passing the buffer to each



(a) Data Type (b) Data Format

Fig. 4. Ares Compression/Decompression Intelligence

library for compression and decompression. We measure the time spent in these operations, and we calculate the CR of the compressed data. As it can be seen in Figure 4(a), different libraries excel in different data types. For instance, `lz4` offers the best CT for integers with CR of 1.42x whereas `snappy` is best in decompressing integers. One observation we make is that there is a clear trade-off between CT and CR: the more time spent compressing data, the more compressed the final output would be. For instance, `bsc` and `bzip2` are the slowest libraries but they both achieved superior compression ratio. Another finding is that when a mixed input is passed for compression, each library takes a hit in performance since the algorithms implemented inside of each library might be optimized or simply more suitable for certain data types. This observation is exactly what Ares exploits and offers better performance when compared to other frameworks. This is apparent from our results. By spending additional time to analyze the input data and detect the data type, Ares can boost performance by using the best compression algorithm for the given input. In case of the mixed data type buffer, Ares decomposes the buffer in smaller buffers of homogeneous data types (e.g., one for characters, one for integers, etc.,) and compress them separately using the best compression library for each case. This approach results in 26-50% performance boost over `lz4`, `quicklz` and `snappy` and up to 24x over slower libraries such as `bsc`. In the mixed data type case, Ares also offers the best CR of 8.79x. In Figure 4(b) we can see the results of comparing Ares to other state-of-the-art compression libraries when various data formats are used as input. In this test, we pass an entire directory containing data represented in different formats. We used the following data formats: binary data (e.g., flat files like POSIX), scientific data (e.g., HDF5, pNetCDF), textual data (e.g., HTML, XML, JSON), and columnar data (e.f., Avro, Parquet). We also used a mixed data format where the directory contains all the above files. In all test configurations the total data size of the input is 64 GB organized in 64 files (i.e., each file has 1GB of char, integers, float, and double). The benchmark's workflow is the same: get input data and compress/decompress them while capturing CT/DT and calculating CR. As can be seen from the results, binary data are compressed faster using `lz4` with a CR of 1.75x. For heavier compression, `bsc` offers a CR of more than 5x, but it is significantly slower in CT and DT. For columnar data, `snappy` takes the lead by offering the best DT with a CR of 2.04x. A similar trend can be observed in these results. Compression time and ratio are directly related. `lz4`, `quicklz` and `snappy`, while the fastest, only achieved CR of 2.54x on average. In contrast, `bsc` and `bzip2` compressed heavily the data taking more time to do so. In the mixed directory, each library performed slightly slower when compared to the homogeneous input data format. As before, this can be attributed to internal optimizations (or lack thereof) of each compression algorithm implemented by each library. Ares takes advantage of this observation, and by analyzing the input directory, it extracts the data format and picks the best library for the task at hand. This approach results in 19-35% performance boost over `lz4`, `quicklz` and `snappy` and 17x over `bsc` while offering competitive CR of 4.12x.

*3) **Compression/Decompression Adaptiveness**:* Applications data management needs vary across domains and platforms. Modern applications consist of a complex set of phases organized into one workflow. In each of those phases, different data compression techniques might be needed. For instance, during a communication heavy workload, a quick compression/decompression scheme is preferred than a heavy compression one. However, a phase that writes out to the archival storage the final results of an application will be greatly benefited by a strong compression ratio minimizing the final footprint of data-in-rest. These observations, on how applications use compression/decompression, partially motivated us to create Ares, a framework that provides the infrastructure to perform application-specific compression schemes. In this test, we pass a CSV file containing 64 GB of data organized in four columns: sorted integer as an index, char as a location ID, integer as population size, and double as the median income. The diversity of data types in the input does not help the generality of each compression library. Ares, on the other hand, takes advantage of the input characteristics to offer the best performance for the given scenario. In this test, we configured Ares to prioritize speed (i.e., CT or DT respectively), and compressibility (i.e., CR) and we compare the results with other compression frameworks. As can be seen in Figure 5, Ares' ability to adapt to the workload can boost the performance relative to the objective. When Ares is configured to prioritize compression speed, for instance, Ares

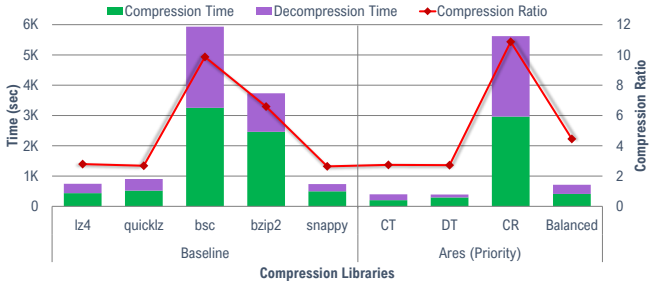
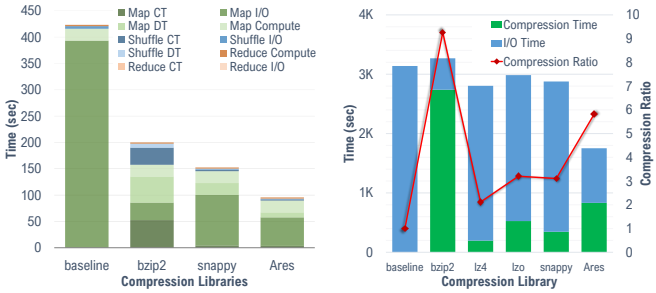


Fig. 5. Tuning Workload Compression Metric Priorities



(a) Read-intensive: Word Count (b) Write-intensive: VPIC

Fig. 6. Applications

decomposes the .csv input file in individual buffers containing homogeneous data type (i.e., one buffer per column from the csv file), and use the library that offers the best CT to perform the compression. This approach results in a performance boost between 2.1-15.8x for CT and 1.6-13.9x for DT compared to all other libraries tested. Ares leverages the data types to achieve this specialized compression and ultimately lead to better performance. We can see a similar picture when Ares is configured to boost the compression ratio offering the highest CR of 10.87x using a combination of `bsc` and `huffman` algorithms. More interestingly, if we focus on the *balanced* mode, Ares does not provide the fastest compression/decompression speeds or the heaviest compression ratio, but instead it provides the most balanced approach of all. It can offer a respectable 4.45x CR while only being between 16-30% slower than `lz4`, `quicklz` and `snappy`. As a result, Ares offers a higher score in equation 1 which reflects the real value of a compression engine: as fast as possible with the higher ratio as feasible. This test highlights the need of an adaptive compression framework that can offer custom performance characteristics based on the user's priorities by passing a flag to the framework's API.

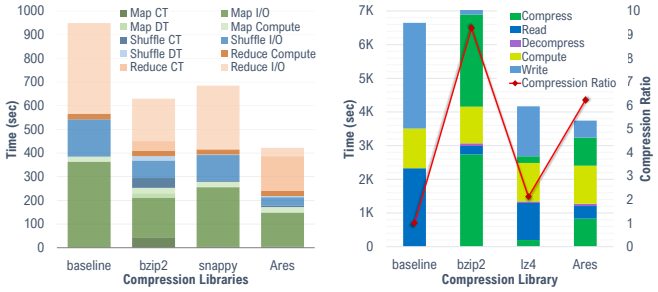
*4) **Compression/Decompression Flexibility**:* Ares' strength comes from its ability to perform compression based on the input data type and format. Furthermore, Ares provides the infrastructure to prioritize certain compression characteristics given a workload. Ares aims to support both scientific and cloud workloads through its C/C++ and Java bindings. Further, Ares abstracts the details of each compression library it contains in its engine, which makes it simple to use and flexible to extend to more compression libraries if needed. In this set of tests, we evaluate Ares with real applications, and we demonstrate the benefits of such an approach. We test Ares' performance with four distinct applications both scientific (i.e., VPIC and HACC) and cloud workloads (i.e., word-count and integer sorting). We investigate three types of workloads: *read-intensive*, *write-intensive*, and *mixed read/write*.

Read-intensive: to test a typical read-intensive workload, we use a Map-Reduce implementation of the word-count kernel, running on 40 nodes (i.e., 32 mappers and 8 reducers). The input to this application is a 1.5 TB of HTML files containing Wikipedia articles written in English. The flow of this application is as follows: a) each map task first reads its input data from HDFS, counts individual word occurrences, and produces intermediate files that contain a word-to-count mapping (i.e., intermediate files data size is an order of magnitude smaller than input files), b) during the shuffle phase, all intermediate files are sorted and sent to the reducers, c) reduce tasks read the intermediate files to merge the final count across all files and write the final word count back to a file in HDFS (i.e., final data size is an order of magnitude smaller than the intermediate files). The majority of the data is moved from the file system to computation during the map phase making this workload read-intensive. Compression techniques can be beneficial to the overall performance by minimizing the data size. The data compression requirements of this workload vary in each phase with map tasks benefited the most by a compressed input and a high decompression speed. Shuffling needs a quick compression to minimize I/O traffic. Reducers do not benefit much from compression since the final output is already too small. Figure 6(a) shows the results broken down to each phase. As it can be seen, by applying compression on the input data, the map phase I/O time is reduced significantly. There is a trade-off, however, between how much time is spent to perform the compression and the data size saved by it. For instance, `bzip2` applies heavy compression in 52 seconds while minimizing the I/O time by 10x over the baseline. `Snappy` on the other hand, is faster in compression with less CR which results in more I/O time over `bzip2`. Ares strikes a balance between the metrics, CT/DT and CR, achieving the best overall performance. Ares is 4.4x faster than the baseline and 1.6-2.1x faster than other compression libraries since it leverages the data type and format being compressed. Lastly, the shuffle phase is also faster in libraries that offer fast compression/decompression speed, such as `snappy` and Ares, especially when compared to libraries that aim to offer high CR such as `bzip2`. This result is not surprising since reduce phase can start as soon as shuffle sends the intermediate data. If compression is expensive in CPU cycles, then the reduce phase gets stalled. Overall, this test highlights the importance of striking a balance of compression speed and ratio. General-purpose libraries do not offer dynamic adaptiveness based on the workload type. Ares thrives because of the data type and format awareness and the flexibility to choose the best library for the task at hand.

Write-intensive: we evaluate Ares in write-intensive workloads using the VPIC simulation. In this workload, each process is producing 1 GB of data that need to be written to storage at the end of each time step. The overall data size used in our test is 1.5 TB organized in an HDF5 file with

7 datasets (i.e., two datasets of integers, two of floats, and three of doubles). We compare Ares with built-in compression filters provided by HDF5. Results are shown in Figure 6(b). In this test configuration, the baseline writes out the data uncompressed in 3139 seconds. When compression is applied, the I/O time is reduced since fewer data are written out. The overall performance of this application is directly related to how fast we can store the final results to the file system. Hence, the compression ratio is an important metric since it minimizes the data footprint in the storage system. However, heavy compression is costly, and a balance must be found to be beneficial to the application. For example, `bzip2` shrinks the data size from 1.5 TB down to about 160 GB, and thus, it reduced I/O time by more than 6x. However, it spent 2608 seconds in compressing the data making it 5% slower in overall completion time than the baseline. An opposite picture can be seen when using `lz4`, `lzo`, and `snappy` as compression filters. These libraries performed data compression significantly faster (i.e., in 350 sec on average) but with fewer data savings (i.e., 2.7x ratio) resulting in a similar overall performance with `bzip2`. On the other hand, Ares offers a solution that is optimized to compress data by specific type while prioritizing both CT and CR. In this test, Ares achieved a balance between CT and I/O time. It compressed data in 832 seconds (i.e., faster than `bzip2` but slower than `lz4`, `lzo`, and `snappy`) while reducing the data size almost 6x. This resulted in a 38-47% faster overall performance.

Mixed Read/Write: we evaluate Ares in scenarios with mixed read/write operations using two applications: HACC simulation and Integer Sort implemented in the MapReduce framework. Figure 7(a) describes the results of integer sorting. The input is 1.5 TB of integers stored in a collection of data formats (i.e., csv, xml, and json). This application sorts the integers in phases (i.e., out-of-core) and produces 1.5 TB of intermediate data (i.e., partially sorted data) before it goes into the reduce phase where the final sorted output is created by merging the intermediate data. As it can be seen by the results, compression has an impact in performance in all phases. Specifically, `bzip2` that compresses data heavily, boosted the overall performance by 1.5x while `snappy`, which has a smaller CR, by 1.38x. In contrast, Ares performed better by offering a 2.25x performance improvement over the baseline of uncompressed data. This improvement comes for two main reasons: first, Ares take into consideration the data format before choosing which compression library to use, and second, Ares uses different library per phase. Specifically, during map phase, where data are compressed before sent to map tasks, Ares used `huffman` encoding that offers quick compression speed for a CR between `bzip2` and `snappy`. During the shuffle phase, where decompression speed is essential, Ares used `lzma` that excels in integer compression. Finally, during reduce phase, Ares used a heavy compression library, `bsc` to achieve maximum compressibility and therefore small I/O time over the file system. Similar results can be seen in Figure 7(b), where Ares' choice of compression library between



(a) Mixed Read/Write: Integer Sort (b) Mixed Read/Write: HACC

Fig. 7. Applications

the phases of the HACC simulation positioned its performance in between `bzip2` and `snappy`. Specifically, HACC first reads 1.5 TB of particles (i.e., a collection of integers and floats)from binary files, then goes into its computation phase, and lastly writes 1.5 TB of the new values of the simulated particles. Ares once again leveraged the different data type between the input and offered smaller CT with a competitive CR (i.e., lower than `bzip2` but higher than `snappy`). In summary, the results highlight our initial hypothesis: different data types and formats affect compression characteristics, and each application might benefit from a dynamic selection of compression algorithms based on its specific type of workload.

VI. Related Work

Several efforts have been made to provide specialized algorithms for a given workload. MAFCO [29] is a lossless compression tool which specializes in compressing MAF (Multiple Alignment Format) files. It aims to gain a better compression ratio than Bzip2 (a general purpose compression library). GTZ [27] is a compression and transmission tool for FASTQ files. For this workload, it outperforms other tools such as DSRC2, QUIP, and LW-FQZip in compression ratio and speed. FPcrush [37] and SPDP [11] are compression tools that are specialized for single and double precision floating point data. They highlight their better speed and compressibility against general-purpose compression libraries such as Bzip2 and Zlib. These libraries implement a specific algorithm to improve performance. By nature, Ares is a modular framework for managing compression libraries smartly. Hence such libraries can be easily incorporated into the framework.

There are several efforts in making compression context-aware. Barr et al. [38] use asymmetric compression schemes to reduce energy consumption. The authors propose the use of suitable compression libraries based on current network monitoring and processor resources. This work focuses on optimizing energy in contrast to Ares which is performance driven, offering compression effectiveness by choosing the most effective compression algorithm based on the workload. Finally, [39] suggest identifying the data-type to choose the best compression libraries and they are the closest to Ares. However, they do not account for the data-format or the workload characteristics of the application.

There have been several efforts in detecting type and format of the data. Caballero et al. in [35] perform a comprehensive survey on type interference of data. They categorize techniques

and present their pros and cons. In the area of file-format detection, McDaniel et al. in [40] present various techniques to identify the data format of a file. These works were directly utilized in the Ares framework's input analyzer module to build data-type and data-format detection engines.

VII. Conclusion

In this work, we performed a comprehensive benchmarking to investigate how different data-types, data-format, and workload characteristics affect the choice of the "ideal" compression library for a given use case. Furthermore, we have developed Ares, a dynamic, adaptive, and flexible compression framework, that can transparently meet various compression needs of big data applications. Ares integrates multiple compression libraries under a single and easy to use framework. Ares has low overhead in analyzing the nature of input data leading to the choice of an appropriate compression algorithm. Results show that Ares can boost performance when compared to traditional compression libraries. Specifically, under real-world applications, from both scientific and Cloud domains, Ares performed 2-6x faster than competitive solutions with a low cost of additional data analysis (i.e., overheads around 10%). Ares leverages different compression algorithms for different application needs and provides a flexible infrastructure for the users to customize their compression characteristics based on the task-at-hand. As a future step, we plan to utilize machine learning to enhance Ares' input analyzer improving the robustness of the input analysis process.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grants no. OCI-1835764, CSR-1814872, CCF-1744317, CNS- 1730488, and CNS-1526887 .

References

[1] M. Mohammadi and A. Al-Fuqaha, "Enabling cognitive smart cities using big data and machine learning: Approaches and challenges," *IEEE Communications Magazine*, vol. 56, no. 2, pp. 94–101, 2018.

[2] "Community Earth Simulation Model (CESM)," https://www2.cesm.ucar.edu/, [Online; accessed November-2018].

[3] "Storage Systems and Input/Output for Extreme Scale Science (Report of The DOE Workshops on Storage Systems and Input/Output)," https://bit.ly/2K5PpjB, [Online; accessed November-2018].

[4] U. Fugiglando, E. Massaro, P. Santi, S. Milardo, K. Abida, R. Stahlmann, F. Netter, and C. Ratti, "Driving Behavior Analysis through CAN Bus Data in an Uncontrolled Environment," *IEEE Transactions on Intelligent Transportation Systems*, no. 99, pp. 1–12, 2018.

[5] E. Riedel, G. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia applications," in *Proceedings of 24th Conference on Very Large Databases*. Citeseer, 1998, pp. 62–73.

[6] D. R. Cox, *Analysis of survival data*. Routledge, 2018.

[7] F. Puig-Castellví, Y. Pérez, B. Piña, R. Tauler, and I. Alfonso, "Compression of multidimensional NMR spectra allows a faster and more accurate analysis of complex samples," *Chemical Communications*, vol. 54, no. 25, pp. 3090–3093, 2018.

[8] R. A. McLeod, R. D. Righetto, A. Stewart, and H. Stahlberg, "MRCZ–A file format for cryo-TEM data with fast compression," *Journal of structural biology*, vol. 201, no. 3, pp. 252–257, 2018.

[9] H. Shan, S. Williams, and C. W. Johnson, "Improving MPI Reduction Performance for Manycore Architectures with OpenMP and Data Compression," *ICPP 2018*, pp. 58:1–58:11, 2018. [Online]. Available: http://doi.acm.org/10.1145/3225058.3225114

[10] J. Binas and Y. Bengio, "Low-memory convolutional neural networks through incremental depth-first processing," *arXiv preprint arXiv:1804.10727*, 2018.

[11] Steven Claggett and Sahar Azimi and Martin Burtscher, "SPDP: An Automatically Synthesized Lossless Compression Algorithm for Floating-Point Data," in *2018 Data Compression Conference*. IEEE, 2018, pp. 335–344.

[12] John Gantz and David Reinsel, "Extracting value from chaos," *IDC iview*, vol. 1142, no. 2011, pp. 1–12, 2011.

[13] P. C. Broekema, R. V. Van Nieuwpoort, and H. E. Bal, "Exascale high performance computing in the square kilometer array," in *Proceedings of the 2012 workshop on High-Performance Computing for Astronomy Date*. ACM, 2012, pp. 9–16.

[14] IDC, "Rich data and the increasing value of IoT," https://www.emc.com/leadership/digital-universe/2014iview/index.htm, 2014, [Online; accessed November-2018].

[15] S. IDC, "Data Age 2025," https://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf, 2017, [Online; accessed November-2018].

[16] Oracle, "The rise of data capital," https://bit.ly/2nj6ZV6, 2016, [Online; accessed November-2018].

[17] M. M. Theimer, G. D. Ghare, J. D. Dunagan, G. Burgess, and Y. Xiong, "Dynamic partitioning techniques for data streams," 2017, uS Patent 9,720,989.

[18] Galit Shmueli and Peter C Bruce and Inbal Yahav and Nitin R Patel and Kenneth C Lichtendahl Jr, *Data mining for business analytics: concepts, techniques, and applications in R*. John Wiley & Sons, 2017.

[19] R. Duvignau, V. Gulisano, M. Papatriantafilou, and V. Savic, "Piecewise Linear Approximation in Data Streaming: Algorithmic Implementations and Experimental Analysis," *arXiv preprint arXiv:1808.08877*, 2018.

[20] J. Uthayakumar, T. Vengattaraman, and P. Dhavachelvan, "A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications," *Journal of King Saud University - Computer and Information Sciences*, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1319157818301101

[21] A. Husseen, S. S. Mahmud, and R. Mohammed, "Image compression using proposed enhanced run length encoding algorithm," *Ibn AL-Haitham Journal For Pure and Applied Science*, vol. 24, no. 1, 2017.

[22] A. Al-Okaily, B. Almarri, S. A. Yami, and C.-H. Huang, "Toward a Better Compression for DNA Sequences Using Huffman Encoding," *Journal of Computational Biology*, vol. 24, no. 4, pp. 280–288, 2017.

[23] S. Chattopadhyay and G. Chattopadhyay, "Conjugate gradient descent learned ANN for Indian summer monsoon rainfall and efficiency assessment through Shannon-Fano coding," *Journal of Atmospheric and Solar-Terrestrial Physics*, vol. 179, pp. 202–205, 2018.

[24] K. Sayood, *Introduction to data compression*. Morgan Kaufmann, 2017.

[25] A. Khan, A. Khan, M. Khan, and M. Uzair, "Lossless image compression: application of Bi-level Burrows Wheeler Compression Algorithm (BBWCA) to 2-D data," *Multimedia Tools and Applications*, vol. 76, no. 10, pp. 12 391–12 416, 2017.

[26] R. Rahim, M. Dahria, M. Syahril, and B. Anwar, "Combination of the Blowfish and Lempel-Ziv-Welch algorithms for text compression," *World Trans. Eng. Technol. Educ*, vol. 15, no. 3, pp. 292–297, 2017.

[27] Yuting Xing and Gen Li and Zhenguo Wang and Bolun Feng and Zhuo Song and Chengkun Wu, "GTZ: a fast compression and cloud transmission tool optimized for FASTQ files," *BMC bioinformatics*, vol. 18, no. 16, p. 549, 2017.

[28] "100,000 GENOMES:Genoox Selected to Serve the Israeli Genome Project," https://tinyurl.com/y8bdcfck, [Online; accessed September-2018].

[29] Luís MO Matos and António JR Neves and Diogo Pratas and Armando Pinho J, "MAFCO: A compression tool for MAF files," *PloS one*, vol. 10, no. 3, p. e0116082, 2015.

[30] "Pied Piper compression algorithm," http://www.piedpiper.com/, [Online; accessed September-2018].

[31] A. Mensikova and C. A. Mattmann, "Ensemble sentiment analysis to identify human trafficking in web data," in *Proceedings of ACM workshop on Graph Techniques for Adversarial Activity Analytics (GTA32018)*. ACM, 2018, p. 6.

[32] "National Center for Atmospheric Research (NCAR) - Weather Research and Forecasting (WRF) Model," https://www.mmm.ucar.edu/weather-research-and-forecasting-model, [Online; accessed November-2018].

[33] H. Devarajan, A. Kougkas, and X.-H. Sun, "List of libraries and workloads tested," https://tinyurl.com/ybtgule2, 2018, [Online; accessed December-2018].

[34] Hariharan Devarajan, Anthony Kougkas, Xian-He Sun, "A comprehensive study of the compressibility of various data type and formats." http://www.cs.iit.edu/ scs/publications.html, Illinois Institute of Technology, Tech. Rep., 11 2018.

[35] J. Caballero and Z. Lin, "Type inference on executables," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 65:1–65:35, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2896499

[36] Chameleon.org, "Chameleon system," https://www.chameleoncloud.org/about/chameleon/, 2017, [Online; accessed November-2018].

[37] M. Burtscher, H. Mukka, A. Yang, and F. Hesaaraki, "Real-time synthesis of compression algorithms for scientific data," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 23.

[38] K. C. Barr and K. Asanović, "Energy-aware lossless data compression," *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 3, pp. 250–291, 2006.

[39] J. J. Fallon, "Data compression systems and methods," May 11 2010, uS Patent 7,714,747.

[40] M. McDaniel and M. H. Heydari, "Content based file type detection algorithms," in *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*. IEEE, 2003, pp. 10–pp.