Data Compression in Resource Constrained Devices

Finding efficient Data Compression algorithms for IoT devices

Hitarth Kothari
Birla Institute of Technology and Science Pilani, Goa
f20190178@goa.bits-pilani.ac.in

ABSTRACT

The aim of this research project is to find efficient data compression algorithms for resource constrained devices such as IoT(Internet of things) devices. These devices have limited processing and memory capabilities. Applying traditional data compression algorithms is not be suitable for the data generated on these devices. Our study attempts to find a suitable compression algorithm/tool that gives a reasonable trade off between factors such as compression/decompression speed, compression/decompression ratio and resource consumption on these devices.

1 INTRODUCTION

Motivation. This research is especially important in the context of federated learning. Federated Learning removes the need to send data from IoT devices to cloud, for applying machine learning algorithms. However to accomplish the training of the model on the IoT device itself, the enormous data generated by the device needs to be effectively compressed and stored in the device. This poses a problem as the IoT devices have limited memory and processing capabilities.

Related Work. According to the paper published in Hotedge 2020 relating to adaptive compression for resource constrained data [3], several factors need to be taken into account for resource constrained devices. These include CPU usage, memory usage, number of cores in the device, type of dataset and network bandwidth. The authors perform several tests on different types of data and arrive to the conclusion that an adaptive compression tool is the best solution. They propose a tool built by them known as 'Zipmate' that takes into account the aforementioned parameters on similar data as the data to be compressed, to suggest the best compression tool. Although 'Zipmate' is in the development state, their results convince us that adaptive compression is the best possible way.

Another study published in the 2018 IEEE conference on Big Data [1] suggests that specific type of data can be dealt specifically in order to get the best results. The authors compress a graph using huffman coding. They visualize a graph as an adjacency matrix, then they find the recurring patterns of 1 and 0. From this data they build a metadata list and a compressed adjacency matrix using huffman coding. Although their resource utilization is not ideal they point out the fact that type of dataset makes a significant difference.

Supervised by Dr. Arnab Paul.

Project in Computer Science, BITS Pilani Goa,
2022.

2 BACKGROUND

Compression can be lossy or lossless. Our study focusses on lossless compression. All compression algorithms are built upon certain compression techniques that leverage the fact that repeating chunks of data can be stored as metadata so that the actual data can be compressed. These techniques include huffman coding, run length encoding, entropy encoding and dictionaries among others. The combination of these techniques have led to the advent of several popular compression algorithms such as the Lempel-Ziv(LZ) series of algorithms, deflate algorithm etc [2]

These compression algorithms in turn, are the backbone of popular and widely used compression tools in modern day computers such as bzip, gzip, zip, zstd and brotli among others. Our study focusses on three tools - zip (deflate), gzip (deflate) and zstd (finite state entropy encoder + dictionary compression). Zip represents the traditional way of data compression using the deflate algorithm. Gzip works in a similar way but uses tar command alongside the compression to archive the files first. Zstd is a new tool developed by meta that uses latest techniques.

3 EXPERIMENT METHOD

We designed a bash script alongside a python script that gives the CPU usage and RAM usage of the system during the compression and decompression. See figure 1.

(1) First, the bash script takes the compression tool input - zip, gzip or zstd. It also has levels or operation modifiers that can be specified along with the tool as input.
(2) Then it takes the path of the directory which needs to be compressed. Using these inputs, it calls the python script for compression.
(3) The python script executes the bash command for the compression and simultaneously starts the monitoring.
(4) Step 3 is invoked again for decompression in place of compression by the bash script.
(5) After the decompression ends, 4 graphs are generated - CPU usage during compression, CPU usage during decompression, RAM usage during compression and RAM usage during decompression.

3.1 Bash script

The script takes directory path and the tool as input, the tools have the following additional modifiers that can be taken as input.

- *zip* - level from 0 to 9 (1 being fastest and 9 being best ratio)
- *gzip* - level from 1 to 9 (0 being fastest and 9 being best ratio)
- *zstd* - fast mode (fast compression speed), ultra mode (best compression ratio) and adapt mode (adapt speed and ratio based on resource availibilty)

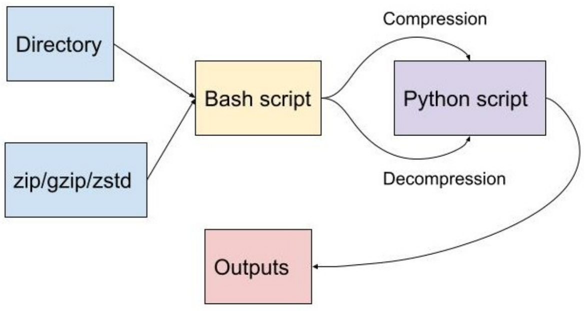


Figure 1: Flow of testing

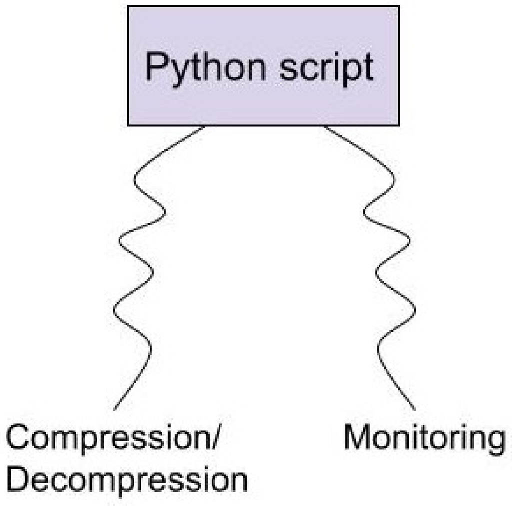


Figure 2: Python script multithreading

3.2 Python script

The python script uses **multithreading** to monitor the CPU,RAM usage alongside the compression/decompression. One thread handles the compression/decompression and the other thread handles the resource monitoring. Once the compression/decompression thread ends, it signals the monitoring thread to end. See the python script visualization in figure 2

The outputs are generated as bar graphs using matplotlib. The x-axis represent the 0.5 second time intervals and the y-axis represent the CPU/RAM usage as a percentage in that interval.

4 EXPERIMENTAL EVALUATION

4.1 Experimental setup

The experiment was carried out on **Raspberry Pi Model 3B+**. The raspberry pi has a 64 bit quad core processor and 1GB SDRAM. It serves as the ideal resource constrained IoT device prototype for the experiment.

We chose **2 1GB datasets** to carry out the compression on the pi. The first dataset had just 1 csv file of size 1 GB, the second dataset had 25862 images that totalled to 1GB. This was done to understand how difference in distribution of data would affect the results.

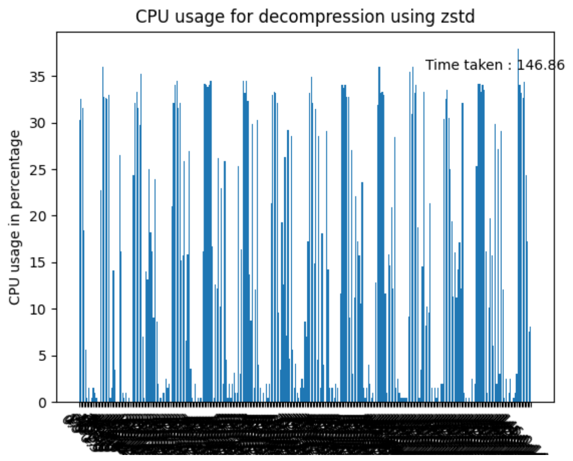
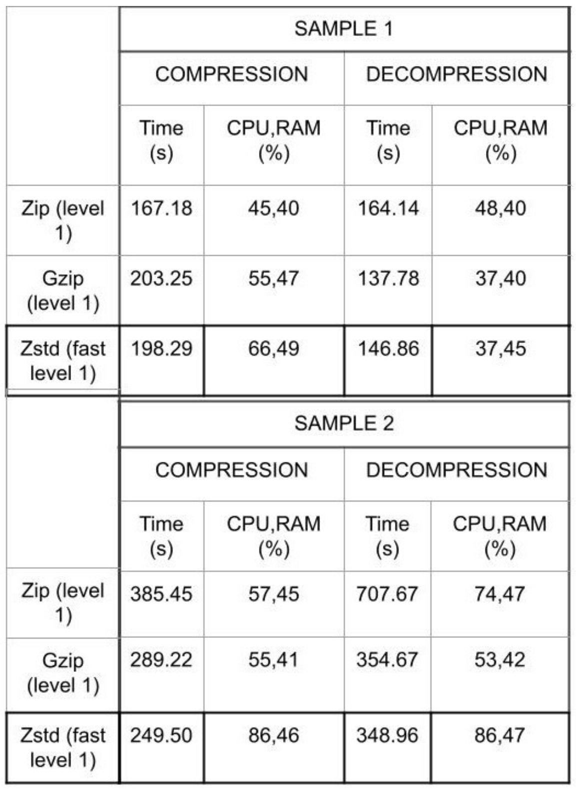


Figure 3: CPU usage for decompression using zstd



	SAMPLE 1			
	COMPRESSION		DECOMPRESSION	
	Time (s)	CPU,RAM (%)	Time (s)	CPU,RAM (%)
Zip (level 1)	167.18	45,40	164.14	48,40
Gzip (level 1)	203.25	55,47	137.78	37,40
Zstd (fast level 1)	198.29	66,49	146.86	37,45

	SAMPLE 2			
	COMPRESSION		DECOMPRESSION	
	Time (s)	CPU,RAM (%)	Time (s)	CPU,RAM (%)
Zip (level 1)	385.45	57,45	707.67	74,47
Gzip (level 1)	289.22	55,41	354.67	53,42
Zstd (fast level 1)	249.50	86,46	348.96	86,47

Figure 4: Results

4.2 Results

The results output were generated as graphs as shown in the example 3. The table in figure 4 shows the complete results for both sample datasets (sample 1 - 1 file and sample 2 - 25862 files), using zip, gzip and zstd. For each compression tool, the fastest level of compression was used to get the reading. The CPU and RAM usage are the peak percentage usages over the entire compression/decompression time.

Inferences

- zstd provided the best compression ratio and speed on an average.
- *T*he sample with over 25000 small files took much more time and resource utilization to be compressed/decompressed on an average.
- *A*lthough zstd gave the best results, its CPU utilization for sample 2 was relatively higher.

5 CONCLUSION

Our study shows that zstd is the best compression tool for resource constrained devices. It provides the best tradeoff between compression speed and ratio. It also boasts of several compression modes that can be utilized based on the need. The dictionary compression model of zstd, proves that some form of adaptive compressing is the best way to proceed for data compression in resource constrained devices. However, there are aspects of zstd that can be changed to further suit the resource constraints of IoT devices [4]. We suggest a few noteworthy points to be explored regarding zstd:

(1) The dictionary for compression is not persistent, but it can be made so. Would using a pre-existing dictionary for compression reduce the burden on processing?
(2) There is no bound on the window size which accesses the memory. This improves compression speed but again comes at the cost of resources used. Would setting an upper bound on the window size reduce resource use to an optimum.

REFERENCES

[1] Amlan Chatterjee, Rushabh Jitendrakumar Shah, and Khondker S. Hasan. 2018. Efficient Data Compression for IoT Devices using Huffman Coding Based Techniques. In *IEEE International Conference on Big Data (IEEE BigData 2018), Seattle, WA, USA, December 10-13, 2018*, Naoki Abe, Huan Liu, Calton Pu, Xiaohua Hu, Nesreen K. Ahmed, Mu Qiao, Yang Song, Donald Kossmann, Bing Liu, Kisung Lee, Jiliang Tang, Jingrui He, and Jeffrey S. Saltz (Eds.). IEEE, 5137–5141. https://doi.org/10.1109/BigData.2018.8622282

[2] ETHW. 2019. History of Lossless Data Compression Algorithms. https://ethw.org/History_of_Lossless_Data_Compression_Algorithms#Dictionary_Algorithms

[3] Tao Lu, Wen Xia, Xiangyu Zou, and Qianbin Xia. 2020. Adaptively Compressing IoT Data on the Resource-constrained Edge. In *3rd USENIX Workshop on Hot Topics in Edge Computing, HotEdge 2020, June 25-26, 2020*, Irfan Ahmad and Ming Zhao (Eds.). USENIX Association. https://www.usenix.org/conference/hotedge20/presentation/lu

[4] Chip Turner Yann Collet. 2016. Smaller and faster data compression with Zstandard. https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/