TSXor: A Simple Time Series Compression Algorithm

Andrea Bruno[1], Franco Maria Nardini[2], Giulio Ermanno Pibiri[2], Roberto Trani[2], and Rossano Venturini[1,2]

[1] University of Pisa, Italy
[2] ISTI-CNR, Italy

Abstract. Time series are ubiquitous in computing as a key ingredient of many machine learning analytics, ranging from classification to forecasting. Typically, the training of such machine learning algorithms on time series requires to access the data in temporal order for several times. Therefore, a compression algorithm providing good compression ratios and fast decompression speed is desirable. In this paper, we present TSXor, a simple yet effective lossless compressor for time series. The main idea is to exploit the redundancy/similarity between close-in-time values through a window that acts as a cache, as to improve the compression ratio and decompression speed. We show that TSXor achieves up to 3× better compression and up to 2× faster decompression than the state of the art on real-world datasets.

1 Introduction

In this paper, we focus on compressing *time series* that have become the de-facto data format for monitoring systems sharing content through the Internet [1]. As a result, time series are heavily used in several machine learning applications. In fact, machine learning algorithms learn analytics on time series data by accessing the data in temporal order and for several times during training. Fast and lossless decompression of time series is important to reduce training time without compromising the accuracy of the process.

We present TSXor, a simple yet effective encoder/decoder for time series that achieves high compression ratios and fast decompression speed. TSXor leverages on the similarity between values in a window. This permits to reference recently seen values using few bytes and, at the same time, to achieve fast decompression by using the window of decompressed values as a data cache. We measure the performance of TSXor in comparison to two state-of-the-art compression algorithms (Gorilla [2] by Facebook and FPC [3]) on seven public, real-world, time series datasets. Results show that TSXor achieves a compression ratio of up to 3× better compared to its competitors while decompressing up to 2× faster.

2 Background

A *uni-variate* time series is a collection of key-value pairs $\langle t_n, v_n \rangle$ for a single time-dependent variable, where the key t_n denotes the time at which the n-th

Table 1. Cost in bits of a value Δ using the range-based encoding by Gorilla.

	Range	Value bits	Total bits
$\Delta \in [-0, 0]$	`0`	0	1
$\Delta \in [-2^6 + 1, 2^6]$	`10`	7	9
$\Delta \in [-2^8 + 1, 2^8]$	`110`	9	12
$\Delta \in [-2^{11} + 1, 2^{11}]$	`1110`	12	16
$\Delta \in [-2^{31} + 1, 2^{31}]$	`1111`	32	36

observation was made and v_n is the corresponding measured value. A *multivariate* time series has m time-dependent variables, hence each point can be regarded as a tuple $\langle t_n, [v_{n,1}, \ldots, v_{n,m}] \rangle$. In our experiments, in Section 4, we consider both types of time series. Refer to the book by Hamilton [4] for an introduction to time series.

FPC [3] is a lossless compression algorithm for double-precision floating-point data. FPC compresses sequences of IEEE 754 double-precision floating-point values by sequentially predicting each value. It uses variants of an FCM [5] and a DFCM [6] value predictor to predict the doubles. Both predictors are implemented using hash tables. The more accurate of the two predictions, i.e., that sharing the largest number of most significant bits with the true value, is XOR-ed with the true value. The XOR operation turns identical bits into zeros. Hence, if the binary representation of the predicted and that of the true value are similar, the result has many leading zeros. FPC then counts the number of leading zero bytes, encodes the count in a 3-bit value, and uses an extra bit to specify which of the two predictions was used. The resulting 4-bit code and the nonzero residual bytes are written to the output.

Gorilla [2] is an in-memory time-series database developed at Facebook. It uses compression techniques based on delta-encoding timestamps and values. The n-th timestamp t_n is turned into a "delta of a delta" as $\Delta = (t_n - t_{n-1}) - (t_{n-1} - t_{n-2})$ and encoded using the simple range-based encoding illustrated in Table 1: if Δ belongs to the k-th range $[\ell, r]$, first k is coded in unary, followed by the binary representation of Δ using $\lceil \log_2(r - \ell + 1) \rceil$ bits. Since most measurements occur at regular and constant intervals, this results in a very small difference between consecutive timestamps (with often $\Delta = 0$), thus achieving good compression effectiveness. Instead, the n-th value v_n is XOR-ed with the previous v_{n-1} and the result of the XOR, say x_n, is encoded as follows (the first value v_0 is written explicitly in 64 bits). If $x_n = 0$, then output a `0` bit. If $x_n \neq 0$, then output a `1` bit and calculate the number of leading and trailing zeros: if these quantities are the same as those of the previous XOR value x_{n-1}, then just output the different bits; otherwise store the number of leading zeros (in 5 bits), the number of different bits (in 6 bits), followed by the different bits themselves.

3 TSXor

Inspired by the XOR-based approach adopted by both FPC and Gorilla, we now present a novel lossless compressor, TSXor. We aim at improving the compression

Table 2. Examples of pairs of values and their corresponding IEEE 754 double-precision representation.

Value	Double-Precision Representation
11.3	0100000000100110100110011001100110011001100110011001100110011010
11.5	0100000000100111000000000000000000000000000000000000000000000000
-6.6	1100000000011010011001100110011001100110011001100110011001100110
-3.8	1100000000001110011001100110011001100110011001100110011001100110
15.9	0100000000101111100110011001100110011001100110011001100110011101
12.4	0100000000101000110011001100110011001100110011001100110011001101

ratios of FPC and Gorilla, while achieving very fast decoding speed. In this preliminary version of the work we focus on compressing the values v_n, that are more challenging to compress effectively compared to the timestamps t_n.

Good compression has to necessarily exploit the empirical property of time series data in that close-in-time measurements are very similar if not exactly the same. To understand how to best exploit this property, we first study how the IEEE 754 double-precision binary representation of two values varies in comparison to their decimal representation. We contribute the following insight: *floating-point values that are very close in decimal format do not necessarily have a similar binary representation.* Table 2 illustrates some concrete examples. The first two rows are relative to 11.3 and 11.5 that are very close in decimal format but only share 16 bits out of 64 (25%). Instead, although the difference between -6.6 and -3.8 is larger than $11.5 - 11.3 = 0.2$, the binary format of -6.6 and -3.8 share 61 bits out of 64 (more than 95%).

As a result of this observation, it is not always effective to compress v_n relative to v_{n-1} (as Gorilla does). Better compression can instead be achieved by enlarging the number of values that should be compared to v_n as to select the one with most common bits. To achieve this, we compare v_n with its preceding $W \leq 127$ values, logically corresponding to the values seen in the time range $[t_{n-W}, t_{n-1}]$. Our goal is to compress v_n relative to this "window" containing the previous W values. We distinguish between 3 cases, namely *Reference*, *XOR*, and *Exception*.

Reference. If v_n is equal to a value in the window, just output its position p in the window. Since the window contains at most 127 values, 1 byte suffices to write the position with the most significant bit always equal to `0`.

If the window does not contain v_n, then we search for the value u in the window such that $x = v_n \oplus u$ has the largest number of leading and trailing zeros bytes. Let p be the position of u in the window. We first write $p + 128$ using 1 byte. In this case the most significant bit will always be `1` because of sum, which allows us to distinguish this case from the Reference case. Let LZ and TZ indicate the number of leading and trailing zero bytes of x respectively.

XOR. If $LZ + TZ \geq 2$, we output a byte where 4 bits are dedicated to TZ and the other 4 bits to the length (in bytes) of the segment of x between the leading and trailing zero bytes. We then write such middle bytes.

Table 3. Basic statistics of the datasets: number of time series, size of each time series, and percentage of distinct values.

Dataset	Time Series	Size	Distinct Values
AMPds2 [7]	14 629 292	11	5.01%
Bar-Crawl [8]	14 057 564	4	12.45%
Max-Planck [9]	473 353	32	0.54%
Kinect [10]	733 432	80	41.07%
Oxford-Man [11]	143 397	19	79.85%
PAMAP [12]	3 127 602	44	0.38%
UCI-Gas [8]	2 841 954	18	0.63%

Exception. Otherwise, we output an exception code, i.e., the value 255 using 1 byte, followed by the plain double-precision representation of v_n using 8 bytes.

The decoding algorithm just reverts the encoding procedure. In particular, during decoding, the last W decoded values are cached in a separate data structure that represents the sliding window. If the Reference case occurs frequently, as we are going to show for several real-world datasets, decoding v_n defaults to an inexpensive lookup in the window, which is small and likely to be kept in the processor cache. Moreover, the encoding of v_n requires just 1 byte which is not possible with neither FPC nor Gorilla. The byte-level alignment maintained by the algorithm further contributes to keep the decoding process simple and efficient.

4 Experiments

In this section, we present the results of an experimental evaluation that compares the performance of TSXor, FPC, and Gorilla on seven public time series datasets. All experiments are carried out on a server machine equipped with Intel i7-7700 cores (@3.60GHz), 64 GB of RAM, and running Ubuntu 18.04. The implementation of TSXor is written in C++ and available at `https://github.com/andybbruno/TSXor`. The code was compiled with gcc 9.1.0 with the `-O3` optimization flag.

We test all algorithms on datasets belonging to different scientific fields so as to not introduce any bias in the results. The datasets comprehend uni-variate as well as multi-variate time series. We do not apply any normalization nor further pre-processing to the datasets. Table 3 reports some basic statistics.

Compression Effectiveness. TSXor achieves a higher compression ratio than FPC and Gorilla, compressing from 1.0 to 3.5× better than Gorilla and from 1.2 to 5.8× better than FPC (which is always outperformed by Gorilla). Furthermore, TSXor achieves a 6.4× compression ratio on the AMPds2 dataset, while the best competitor achieve only a 2.0× compression ratio on this dataset. Here, the strength of TSXor is the use of a single byte in 85% of the cases (see Table 5) to reference an identical 8-byte value that occurred in the sliding window.

On the dataset containing the highest percentage of distinct values, i.e., Oxford-Man, TSXor is still able to beat the other two algorithms. Interestingly

Table 4. Performance of TSXor, FPC, and Gorilla. The best performance on each dataset is highlighted in bold.

	Compr. Ratio			Decompr. Speed (MB/s)			Compr. Speed (MB/s)		
	TSXor	FPC	Gorilla	TSXor	FPC	Gorilla	TSXor	FPC	Gorilla
AMPds2	**6.39**×	1.10×	2.03×	**1174**	411	666	67	339	**704**
Bar-Crawl	**2.36**×	1.20×	1.44×	**710**	436	447	29	424	**466**
Max-Planck	**4.84**×	1.06×	2.97×	**1057**	355	859	52	313	**871**
Kinect	1.37×	1.09×	**1.41**×	**665**	287	636	17	166	**696**
Oxford-Man	**1.30**×	1.06×	1.28×	**604**	222	574	15	170	**630**
PAMAP	**4.85**×	1.01×	1.38×	**949**	224	487	45	182	**521**
UCI-Gas	**3.50**×	1.19×	1.23×	**642**	455	578	22	287	**654**

enough, on this dataset only 23% of the values has been compressed using 9 bytes (see Table 5), thus spending an extra byte with respect to the 8 bytes needed by the uncompressed representation. In this case, our advantage comes from the 17% of the values that are compressed with only one byte (Reference case), while the remaining 59% of the values are compressed using 6.94 bytes on average.

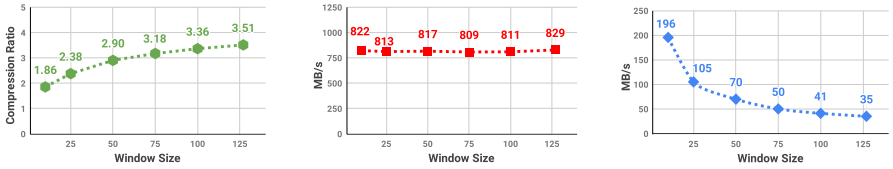
Decompression Speed. Since the time series are compressed once but read several times, the most critical evaluation metric is decompression speed. Therefore, we start by analyzing the decompression speeds (reported in MB/s) of the different algorithms. Gorilla is from 1.0 to 2.6× faster than FPC. TSXor is the fastest algorithm, consistently on all datasets. In particular, TSXor is from 1.0 to 1.9× faster than Gorilla and from 1.4 to 4.2× faster than FPC. The byte granularity helps the algorithm to avoid bit shifts and costly functions calls. In particular, 92% of the times (see Table 5) we end up either in the Reference case or in the XOR case, by consuming only 2.62 bytes on average instead of the 8 bytes of the original representation. This means that TSXor heavily leverages on the window of cached values.

Compression Speed. Regarding the compression speeds (in MB/s), which is the less interesting case, Gorilla outperforms both FPC and TSXor. The reason lies in the simplicity of the algorithm. Indeed Facebook's approach requires neither table lookups nor complicated calculations. The second fastest algorithm is FPC, which compresses the values exploiting two hash functions as predictors. TSXor trades compression speed for better compression and faster decoding speed. In fact, for each value to encode, the whole window is scanned.

Varying the Window Size. We now examine the performance achieved by TSXor when varying the window size W. We show this analysis in Figure 1, which reports the average performance over all datasets. We did not observe noteworthy variations among the different datasets. Figure 1a shows that the compression ratio improves when increasing the window size. Not surprisingly, the larger the window, more compression opportunities are created for Reference and XOR cases. This improvement is balanced by the fact that the compression algorithm needs to look at more values when compressing. Indeed, Figure 1c shows that the compression speed slows down when increasing the window size.

Table 5. Percentage of TSXor cases (Reference, XOR, and Exception) over each dataset. For the XOR case, it is evident that TSXor spends less than 8 bytes for a double-precision value.

	Reference (1 byte)	XOR		Exception (9 bytes)
	%	%	bytes	%
AMPds2	84.87	14.87	3.19	0.26
Bar-Crawl	50.53	28.25	5.53	21.22
Max-Planck	77.93	21.94	4.15	0.13
Kinect	28.01	62.95	7.66	9.04
Oxford-Man	17.44	59.44	6.94	23.12
PAMAP	75.95	23.13	3.63	0.92
UCI-Gas	45.36	54.63	3.57	0.01
Average	54.30	37.89	4.95	7.81



(a) Compr. Ratio (b) Decompr. Speed (c) Compr. Speed

Fig. 1. Compression ratio, decompression speed, and compression speed of TSXor by varying the size of the window. Each point represents the average of each metric over all datasets.

An interesting finding is that the window does *not* affect the decompression speed (Figure 1b). The main reason is that during the decoding phase no searches are performed in the window, but only direct access to individual elements.

5 Conclusion and Future Work

In this short communication we introduced a lossless compression scheme for time series, TSXor, achieving good compression ratios and very fast sequential decoding. Despite its simplicity, TSXor provides very promising results: therefore, we think that room for improvement is possible with more sophisticated mechanisms. One defect of TSXor is certainly its encoding time, as it requires to scan the window for each value to encode. Future work will tackle this issue, e.g., by exploiting vectorized instructions. We will also explore the concrete applicability of TSXor to machine learning applications.

Acknowledgments. This work was partially supported by the projects: MobiDataLab (EU H2020 RIA, grant agreement Nº101006879), OK-INSAID (MIUR-PON 2018, grant agreement NºARS01_00917), and "Algorithms, Data Structures and Combinatorics for Machine Learning" (MIUR-PRIN 2017).

Bibliography

[1] Shivangi Vashi, Jyotsnamayee Ram, Janit Modi, Saurav Verma, and Chetana Prakash. Internet of Things (IoT): A vision, architectural elements, and security issues. pages 492–496, February 2017. https://doi.org/10.1109/I-SMAC.2017.8058399.

[2] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: a fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, August 2015. ISSN 2150-8097. https://doi.org/10.14778/2824032.2824078.

[3] Martin Burtscher and Paruj Ratanaworabhan. FPC: A High-Speed Compressor for Double-Precision Floating-Point Data. *IEEE Transactions on Computers*, 58(1):18–31, January 2009. ISSN 0018-9340. https://doi.org/10.1109/TC.2008.131.

[4] James Douglas Hamilton. *Time series analysis*. Princeton university press, 2020.

[5] Y. Sazeides and J.E. Smith. The predictability of data values. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 248–258, December 1997. https://doi.org/10.1109/MICRO.1997.645815. ISSN: 1072-4451.

[6] B. Goeman, H. Vandierendonck, and K. de Bosschere. Differential FCM: increasing value prediction accuracy by improving table usage efficiency. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 207–216, January 2001. https://doi.org/10.1109/HPCA.2001.903264.

[7] Stephen Makonin, Bradley Ellert, Ivan V Bajić, and Fred Popowich. Electricity, water, and natural gas consumption of a residential house in canada from 2012 to 2014. *Scientific data*, 3(1):1–12, 2016.

[8] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical*, 215:618–629, 2015.

[9] Max Planck Institute for Biogeochemistry. Max-Planck-Institut fuer Biogeochemie - Wetterdaten, 2019. URL `https://www.bgc-jena.mpg.de/wetter/`.

[10] Simon Fothergill, Helena Mentis, Pushmeet Kohli, and Sebastian Nowozin. Instructing people for training gestural interactive systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, page 1737–1746, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310154. https://doi.org/10.1145/2207676.2208303. URL `https://doi.org/10.1145/2207676.2208303`.

[11] Gerd Heber, Asger Lunde, Neil Shephard, and Kevin Sheppard. Oxford-man institute's realized library, 2009.
[12] Attila Reiss and Didier Stricker. Introducing a new benchmarked dataset for activity monitoring. In *2012 16th International Symposium on Wearable Computers*, pages 108–109. IEEE, 2012.