RoLZ - The Reduced Offset LZ Data Compression Algorithm

Sabin Belu[1], Daniela Coltuc[2]
[1]*Doctoral School ETTI, Univ. Politehnica of Bucharest, Romania*
[2]*Faculty of Electronics, Tc. and Information Technology, Univ. Politehnica of Bucharest, Romania*

***Abstract*— The paper unveils an exotic data compression algorithm, called *Reduced Offset Lempel Ziv* (RoLZ). Unlike classical Lempel Ziv implementations, RoLZ uses a 'reduced' subset from which a possible 'match' set is chosen and also it minimizes the information needed to describe this match-length set. The big advantage of such approach is higher compression ratio, at some decompression speed expense. Our three algorithm embodiments for LZSS, LZP and RoLZ are tested against five types of data, and in all our tests, the compression ratio of RoLZ is far superior to LZSS' or LZP's.**

I. Introduction

The appearance of RKive in the early 90's as de facto one the best data compression tools on the market, created a lot of commotion and rumours in the bbs/online communities. RKive was written by Malcolm Taylor, a New Zeeland developer and researcher in data compression and it was immediately adopted by researchers and engineers all over the world, since it was achieving compression ratios never seen before. This was a world that has been dominated for years by PkZIP/PkArc, LHa/LHarc or Arj (all trademarks are the property of their respective owners).

Also, during the '90s and mostly in the DotComs boom era, a lot of patents were issued in Data Compresion. RoLZ, the acronym from *Reduced Offset Lempel Zip*, which was later discovered to be the undelying compression algorithm within RKive, is even today free of patents, to the best of the authors' knowledge. It is worth mentioning Robert Jung's US Patent 5,140,321 [7], also known as the *LZ77 limited search patent*. This idea played an important role in shaping RoLZ and it will be discussed later. Meanwhile, RKive was closed-source and its ROLZ algorithm remained undocumented for years; some clues emerged from the readme file associated with RKive compression package. Within the document, Malcolm acknowledges Charles Bloom for his noticeable insights and helps on writing his program [1].

Going back one year, in 1995, Charles Blooom invented a new algorithm that he will be presenting at The Data Compression Conference in Utah, in the following year [2]. This algorithm was called Lempel Zip Prediction (LZP). In his paper describing LZP, appears from the first time the following line: "This algorithm works by reducing the set of available window position for an LZ77 encoder to match from". Furthermore, comparing LZP with LZ77, it turns out that fewer bits are used to indicate a match-length pair, since only the length is transmitted to output location. Charles Bloom presented four versions for LZP algorithms: from LZP1, which used a fixed order-3 context hash table lookup and a 16k byte LZ-window to LZP4, which used an order-5 context and no hashing. However, not even the best LZP variant could come close to Rkive results in compression ratios[2]. It turned out that RKive was a successful combination of compression methods and heuristics, from solid compression to optimal LZ parsing. But RKive's powerful compression algorithm will remained a mistery for many years to come.

Our article's purpose is to depict the power that lies within this algorithm and maybe, once for all, re-enact RoLZ towards wider usage within software programs.

II. LZ77 and two subsequent versions

Lempel-Ziv's 1977 algorithm (LZ77) is a dictionary compression algorithm. The algorithm's core functionality is replacing substrings of commonly seen successions of symbols from the input stream into pairs of position and length. As shown in Fig. 1, LZ77 splits the input stream into history and lookahead buffer; any substring portion of a $< match, length >$ pair points to a copy of it in the history part of the buffer. The output encoding consists of a triple $\langle d, l, s\rangle$ meaning *distance, length, symbol*, where *symbol* is the first literal or unmatched symbol following the match-length pair $\langle d, l\rangle$.

Suppose there is a string S, which starts at ith position of the input stream, called current pointer (Fig. 1):

- The string $S_{i\cdots i+l}$ of length l has another occurrence P, which starts d positions earlier in the text, $S_{i-d\cdots i-d+l}$,
- This earlier occurrence of S, should be always less than l_{max} and should start within a window $S_{i-d_{max}\cdots i-1}$,
- The values d and l must satisfy the following constraints: $d \leq d_{max}$ and $l \leq l_{max}$,
- Strings $S_{i\cdots i+l}$ and $S_{i-d\cdots i-d+l}$ overlap only if $d \leq l$, which makes LZ77 a self-compressible algorithm,
- When in greedy mode, LZ77 always try to maximize l from all the possible occurences of P_j in history,
- The triple $\langle d, l, s\rangle$ could be further encoded using $log_2(d_{max})$ + $log_2(l_{max} - LZ_{MIN\_LEN})$ bits. LZ_{MIN_LEN} is the minimum encoding length which helps decide if a match is accepted or not.

It was until Storer and Szymansky made their LZ77 version famous, with a variant called LZSS. With LZSS, the authors introduced the concept of *coding flags*, used to differentiate a literal from a distance/length pair. This *1-bit flag* eliminates

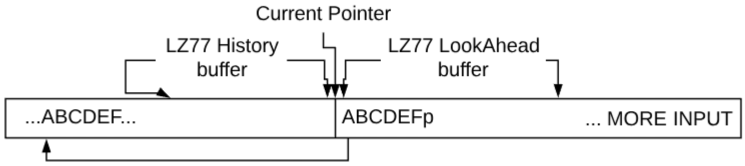

Fig. 1. Diagram of LZ77 algorithm. Symbols at current pointer are compared with similar symbols from the LZ History or symbols that have the same hash value when combined in a "string". There is no context involved in chosing the LZ-History string to be matched against. Only the current pointer makes the separation between the LZ-History and LZ-Lookahead buffers, within the input stream.

the need to output a *triple* at each iteration of the algorithm, thus saving a *literal* for a new search and possibly, a new match.

Years later, LZP rewrote history changing the rules, just like LZSS did for LZ77.

In LZP, offsets from the match-pairs are selected from a subset generated by a finite state Markov chain. The order $cntxN$ of the Markov chain is given by the number of symbols used to predict the matches (Fig. 2).

LZP algorithm is implemented into 5 distinct steps:

1) In the first step, LZP calculates the context value $hashIndex$ for the current position i, by inserting the previously seen $cntxN$ symbols into a hash function hF;
2) In the 2nd step, LZP checks this value against a hash table. If the hash table at requested value, $hash\_table\_value$, is empty or uninitialized, LZP goes straight into step 4. If $hash\_table\_value$ is not null, LZP steps into the 3rd step;
3) In the 3rd step, LZP performs a match trying to find the length of a possible match between the string at the current position i and a string at the position pointed out by the $hash\_table\_value$; the length l of the longest match is stored;
4) In the 4th step, LZP replaces $hash\_table\_value$ with the current position;
5) In the final step, LZP writes to the output location, regardless of the value of l.

LZP output sequence is made up only of lengths and literals. No position/distance bits are sent to output location. The length l will always appear, regardless if its value. However, a literal will only follow if l is null (0), meaning that length l acts also as just like LZSS $codingflag$, helping to distinguish between a literal and a length.

III. RoLZ - Reduced Offset algorithm outlined

Malcolm Taylor took the basic idea from LZP, generalized and improved it by creating RoLZ. While LZP was designed to be simple, fast, memory-friendly and to achieve a somewhat good compression ratio, RoLZ was built to achieve the best compresion ratio of its time.

Instead of a single value hash table, Malcolm Taylor created RoLZ so that its hash table is linked to a $hash\_colision\_node\_list$. The hash collision node list is controlled in the exact same way Robert Jung has depicted in his *LZ limited search patent* and this become a powerful advantage in achieving very good compression ratio. This idea alone shaped LZP into becoming RoLZ.

RoLZ algorithm is implemented into 7 distinct steps, of which the first 5 are derived from LZP:

1) In the first step, RoLZ calculates the context value $hashIndex$ for the current position i, by inserting the previously seen $cntxN$ symbols into a hash function hF.
2) In the 2nd step, RoLZ checks this value against a hash table. In RoLZ case, this value is a pointer to a $collision\_nodes\_list$; if the pointer is null or the list is empty, RoLZ goes straight to step 4. If the pointer is not null or the $collision\_nodes\_list$ is not empty, RoLZ steps into the 3rd step.
3) In the 3rd step, RoLZ performs a match trying to find the longest length of a possible match between the string at current position i and the string at position pointed out by $collision\_nodes\_list$'s first node; the length l_node_1 specifying the longest match found during this current search is stored.
4) In the 4th step, RoLZ tries to maximize the longest match by advancing to the next node from this $collision\_nodes\_list$; so step 3 is repeated for each node until the list has depleted or $max\_nodes\_searched$ is reached.
5) In the 5th step, RoLZ calculates the longest length out of all matches, $l_{max} = max(l\_node_N, l\_node_{N-1}, ..., l\_node_1)$; this l_{max} is also stored, along with the index where it was found, named herein n_{max};
6) In the 6th step, RoLZ inserts the current position, i into the current $collision\_nodes\_list$;
7) In the final step, RoLZ writes l_{max} to the output location. if l_{max} is not null, n_{max} is also sent to output location.

The pseudocode of the complete algorithm with an order-4 context is in Table I.

IV. Experimental results

For the experiments, we have implemented LZSS, LZP and RoLZ variant with the specifications in Table II. As test results, we provide data compression ratios and decompression speed timing which are available from tests performed with applications we have implemented for this article; these applications are embodiments of LZP, LZSS and RoLZ algorithms. We have used test data sets which we believe to be generically

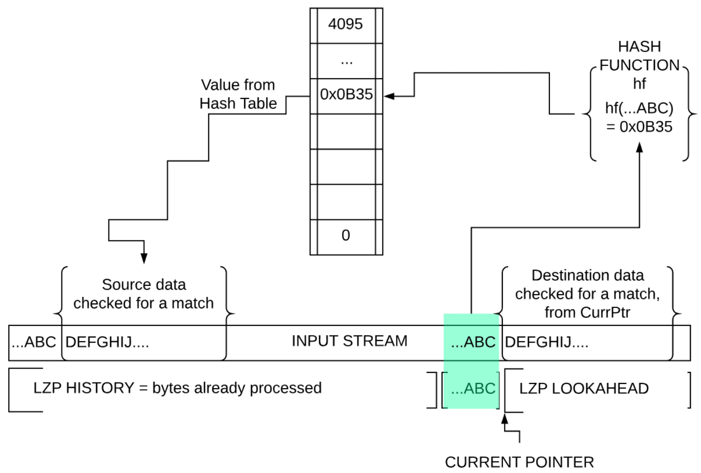

Fig. 2. Diagram of LZP algorithm. The string at the current pointer from LZ-Lookahead buffer's ith position is matched only with strings $following$ the same context from the LZ-History. The longest match is encoded only by its length, l, not by a pair $< d, l >$. The current pointer is always written in the hash table. If no match is found, the current symbol, S_i, is sent to output location and the current pointer advances to the next location. If a match of length l is found, it is sent to output using only l and the current pointer advances $l + 1$ locations.

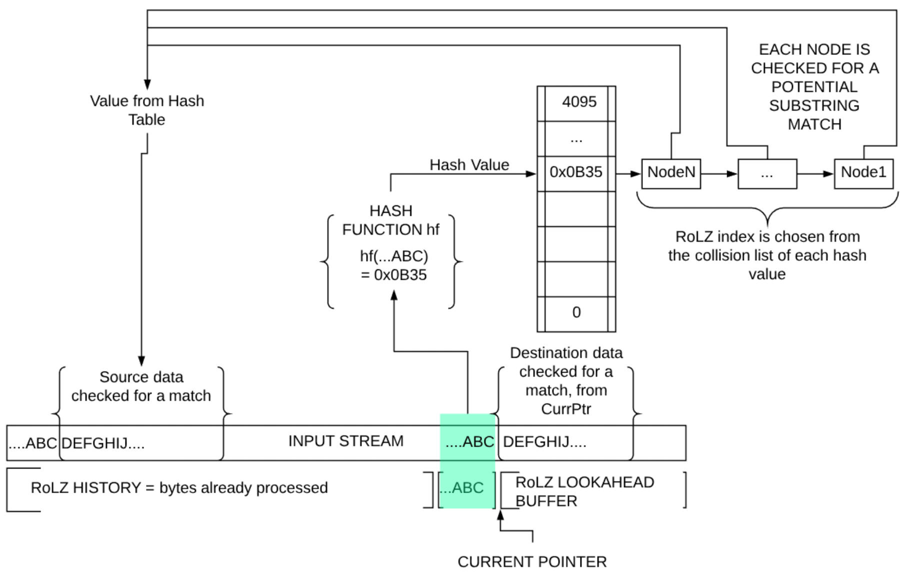

Fig. 3. Diagram of RoLZ algorithm. The string at the current pointer from LZ-Lookahead buffer's ith position is matched only with strings $following$ the same context from the LZ-History. All matches l_p are stored and the maximum length l_{max} is computed. The current pointer is always added to the $collision\_nodes\_list$. If l_{max} is zero, no match is found, the current symbol S_i is sent to output location and the current pointer advances to the next location. If l_{max} is not zero, it is sent to output and the current pointer advances l_{max}+1 locations.

applicable for any data compression application, as explained in details:

1) Executable files include a collection of Windows PE files (portable executables), 735,974,314 bytes in size.
2) Formated Text files include a collection of log files and metadata files with binary headers and text, text configuration files, 142,498,905 bytes in size.
3) Object Files include various debug binaries, debug information data, totalling 552,312,846 bytes.
4) English Text files include a collection of text files, readme, plain english text and log files, with a total size of 90,551,771 bytes.
5) Miscelaneous Binary files include various release binaries, binary images, pre-formated binary headers, binary configuration files, 582,719,354 bytes in size.

Tests were performed on Intel(R) Xeon(R) E5-CPU 2.80 GHz, 16GB RAM, 64-bit Windows 10 OS; test applications were 64bit windows applications compiled with MsVisual C++ Community Ed. 2017. RoLZ performs extremely well on text and object files as compared with LZSS and LZP (Table III). On executable and object files, RoLZ advancement over LZSS is extremely good, and this can be explained by frequent

TABLE I

RoLZ Pseudocode

```
RoLZ Initialization
  hashBits ← 12
  N ← 2^hashBits
  contxN ← 4
  i ← contxN − 1
  cntx ← S_{i...i−contxN}
  max_nodes_searched ← 64
  hashTable[0...N] ← empty_list : 0...%max_nodes_searchedzeros
  cntx → OUTPUT

RoLZ Compression: Repeat
  n ← 0
  i ← i + 1
  l_max, n_max ← 0
  cntx ← S_{i...i−contxN}
  hashIndext = hF(cntx)
  collision_list = hashTable[hashIndex]
  collision_list ← i          % insert current position into collision list
   for p = collision_list.begin...end && n < max_nodes_searched
     l ← 0
     while (S_{i+k} is equal S_{p+k}) { l ← l + 1}
     l_max, n_max ← l, n
     n ← n + 1
   if l_max > 0
     l_max, n_max → OUTPUT
   else
     n_max, S_i → OUTPUT          % l_max is zero
   i ← i + l_max
Until depletion of input stream
```

TABLE II

LZSS, LZP and RoLZ specifications

	Window Size	Hash Bits	Min Length	CntxN
LZSS	2MiB	16	3	0
LZP	2MiB	16	1	3
RoLZ	2MiB	16	1	3

shorter matches within executables, which are not favoring LZSS results (Table IV).

TABLE III

LZSS, LZP and RoLZ Compression rates

	Original[MiB]	**LZSS**	**LZP**	**RoLZ**
	MiB	%	%	%
Executables	701	68.63	65.04	**64.43**
Formated Text	135	25.15	23.00	**21.87**
Object Files	526	43.68	37.87	**37.33**
TXT Files	86	58.95	58.17	**54.68**
Misc. Binaries	555	56.46	54.84	**53.92**

LZP and RoLZ take advantage of LZ_{MIN_LEN}, the LZ minimum encoding length. LZSS is usually implemented with a minimum encoding length of 3: 2 bytes for position/distance and 1 byte for length. Shorter matches of 3 symbols or less are not encoded by LZSS but they are encoded by LZP and RoLZ, which define LZ_{MIN_LEN} as 1

LZSS results suffers from the same disadvantage in case of formatted text files. Files containing logging output tend to display a lot of information from iterations. The lines of text are highly similar, with some small changes. Breaking large matches into smaller ones which add extra information is extremely poisonous for LZ77 & LZSS algorithms.

TABLE IV

RoLZ vs LZSS, LZP Compression gain

	RoLZ vs LZSS[%]	RoLZ vs LZP[%]
Executables	-4.20	-0.61
Formated Text	-3.28	-1.13
Object Files	-6.35	-0.54
TXT Files	-4.27	-3.49
Misc. binaries	-2.54	-0.92

TABLE V

Decompression times for RoLZ vs LZSS and LZP

	RoLZ	RoLZ vs LZSS	RoLZ vs LZP
	ms	x time	x time
Executables	13,312	1.03	2.55
Formated Text	1,037	3.02	2.46
Object Files	4,413	2.26	2.08
TXT Files	954	2.04	2.26
Mis. binaries	7,933	3.58	2.40

Decompression results vary, but LZSS is more suitable where fast decompression is a must (Table V).

V. Conclusion

Based on our research, studies and empirical research, RoLZ was the best algorithm on all cases in terms of compression ratio. RoLZ seems to adapt much better than LZP and LZSS, mostly due to locality and contextual information stored in the $collision\_list$ and $cntx$. Short match-files are favoured by RoLZ, along with formatted text files, which seem to give much better results in terms of compression ratio. We find RoLZ to be an exceptional algorithm because, as compared to LZSS, the output of $cntx$ bytes allows us to preserve a certain degree of context in the output stream. This implies that RoLZ output is suitable for a $cntxN$-order compression modelling, providing even better compression ratio, once the output is to be further compressed.

While LZSS is usually bound to finite LZ window, LZP and RoLZ can work in an environment where position and distance of matches are infinite in values, with RoLZ providing the best compression rate, among all three algorithms.

References

[1] Malcolm Taylor, *"RKIVE file archiver"* http://files.mpoli.fi/unpacked/software/dos/utils/diskfile/rkv190b1.zip/

[2] Bloom, Charles. *"LZP: A New Data Compression Algorithm."* Data Compression Conference. 1996.

[3] Kolmogorov, A. (1968). *"Logical basis for information theory and probability theory"*. IEEE Transactions of Information Theory, IT-14:662–664, 1968

[4] Ziv, Jacob; Lempel, Abraham (May 1977). *"A Universal Algorithm for Sequential Data Compression"*. IEEE Trans. on Info. Theory, 23:337–343, 1977.

[5] Storer, James A.; Szymanski, Thomas G. (October 1982). *"Data Compression via Textual Substitution"*. Journal of the ACM. 29 (4): 928–951. doi:10.1145/322344.322346.

[6] James A. Storer, 1992, *"Method and apparatus for data compression"* US Patent US5379036A, United States

[7] Robert K. Jung, *"Data compression/decompression method and apparatus"* https://patents.google.com/patent/US5140321/