An innovative algorithm for data differencing

Sabin Belu
Doctoral School ETTI, Univ.
Politehnica of Bucharest,
Bucharest, Romania

Daniela Coltuc
Faculty of Electronics
Tc. and Information Technology
Univ. Politehnica of Bucharest,
Romania

***Abstract*—We are presenting a novel algorithm to create software updates using Normalized Compression Distance (NCD) where we define the Z compression function as a modified LZ77[1] compression algorithm. Our test results presented are based on three algorithmic strategies and they surpass commercial software available on the market with a factor of almost 2:1.**

Keywords—software update, data differencing, LZ77

I. INTRODUCTION

A. Software Updates

When software does not work according to specifications or when new features are required and the effort to redownload and install an entire application is too high, a software update is the right solution. A software update is a small downloadable file 'interpreted' by an application installing the update. It contains simple commands like ADD / CUT / INSERT / COPY symbol(s). These commands get translated into changes that are applied on software files in order to improve or fix features that do not work according to the technical specifications the software was built against. Most of the time software updates or patches are released to address security issues, to remove bugs in the software revealed by continous testing (testing that is done even after the software is released).

B. Data differencing

Applying an update improves or "modernize" a software file not only by updating its version information but by changing the data within. Therefore, software updates or patches consists in the sum or differences between the old and the improved version of the software file. The algorithms that produces them are data differencing algorithms.

Our new differencing engine produces a binary "description" of the differences between the two files, old and new, source and target. Formally, it takes as input, source data and target data, and produces the binary or technical description of the differencing data such as given the source data and the difference data, one can construct a binary identical target data. It simply updates or patches the source data with the difference data to produce the target data.

II. NCD USING LZ77

A. Normalized Compression Distance (NCD)

When applied to computer software, NCD defines a way of measuring the similarity between two files. NCD does not depend on the data contained by the files, thus it is arbitrary. Using NCD, one could solve the problem of finding how hard could be to transform one file into another one, and vice-versa. In other words, given two files or two strings, x and y, with NCD one could find the shortest program p such as p computes x into y or y into x:

$$p(x) = y \text{ or } p(y) = x$$

It has been shown in [7] that in order to express the length of p, one could use Kolmogorov complexity.

$$|p| = \max\{K(x|y), K(y|x)\}$$

In [2], Vitanyi and Cilibrasi define the Normalized Information Distance or the similarity metric, as follows:

$$NIDz(x,y) = \frac{\max\{K(x|y),K(y|x)\}}{\max\{K(x),K(y)\}}$$

where K(x | y) is the algorithmic information of x given y as input [2, 3]. Furthermore, Vitanyi and Cilibrasi redefine NCD by approximating K with real data compression and Z(f) as the binary length of file f encoded by an arbitrary compression algorithm Z

$$\mathrm{NCDz}(x,y) = \frac{Z(xy)-\min\{Z(x),Z(y)\}}{\max\{Z(x),Z(y)\}}$$

Our novel algorithm uses a modified sliding window LZ77 as the preferred data compression algorithm for creating delta differencing.

B. Sliding window LZ77

Fig. 1 shows the ideal case in which the input stream is endless, and the current pointer moves incrementally, thus giving the impression that the LZ buffer slides along

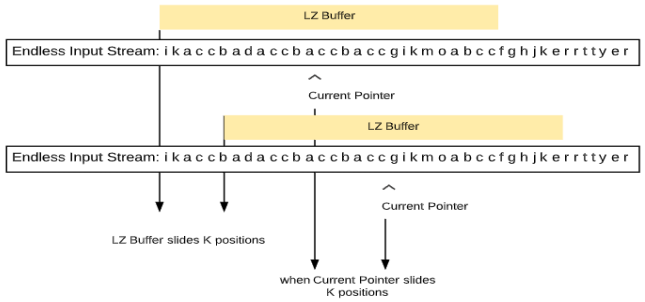


Fig. 1. LZ BUFFER and Current_Pointer located at half of LZ_ BUFFER size.

In reality, due to the fact the resources are always limited and because we want to optimally implement algorithms, the input stream is cut into chunks, and they slide by simply replacing each other's place.

When current_pointer reaches LZ_WINDOW_SIZE, the upper half of the input buffer is moved in the lower half of the

input buffer, initializing current_pointer with LZ_ LZ_WINDOW_SIZE / 2.

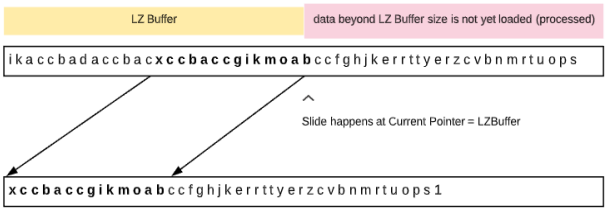


Fig. 2. LZ77 window sliding. When Current_Pointer reaches LZ_WINDOW_SIZE, the upper half window moves into lower half window and the process resumes from Current_Pointer = LZ_WINDOW_SIZE / 2

Current_pointer becomes LZ_WINDOW_SIZE /2 so that the lower half of the LZ window is not processed once more and it is considered now the LZ_HISTORY. The LZ_LOOKHEAD buffer will be from now on only the upper half of the window. The process repeats, strings are again searched and matched, and when the current_pointer reaches again LZ_WINDOW_SIZE, sliding occurs again, until no more input data. See Fig. 3

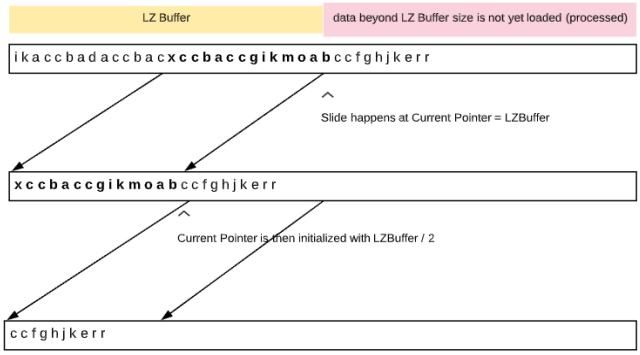


Fig. 3. Continous sliding occures whenever Current_Pointer reaches LZ_BUFFER until no more input data.

III. KEOPS – OUR DATA DIFFERENCING ALGORITHM

We call our new algoritm for delta differencing, KEOPS. KEOPS algorithm creates a data differencing file from two files given as parameters in the console command line. The two files are called herein the old and the new file. f1 is the old file, the initial or the source file. We also name f1 the dictionary file. f2 is the new file, the target or the destination file.

We have designed KEOPS to use the LZ77 sliding window algorithm. As presented above, the current_pointer divides the input buffer into two portions. The LZ-Dictionary portion and the "lookahead buffer" portion. Moving along with LZ77, initially, current pointer is zero, since nothing has been encoded in the input buffer. See Fig. 4

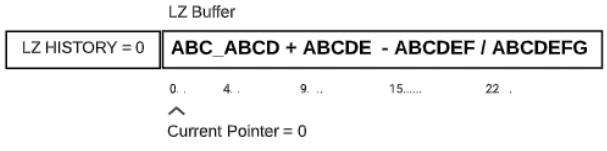


Fig. 4. LZ77 and the two areas in LZ Window: LZHistory and LZLookAhead

Previously processed data, the input data buffer that lies below Current_Pointer (index < Current_pointer) forms the LZ History. The next data to process that lies beyond Current_Pointer (index >= current_pointer) forms the LZ LookAhead buffer. While the current pointer advances incrementally, new substrings are found and encoded.

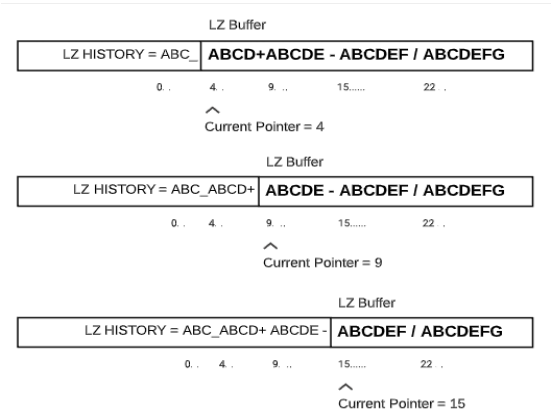


Fig. 5. Examples of moving LZ Current_Pointer and the division on LZ Window into LZ History and LZ LookAhead buffers.

As our purpose is creating data differencing files, things are a bit different in reality. Since the file we need to update is already on the target machine, we will not need to encode it, therefore, we will consider all the data that comprises file f1 as LZ-History, or previously encoded data. This is the reason why our algorithm will position the starting pointer LZ_WINDOW_SIZE / 2. See Fig. 6

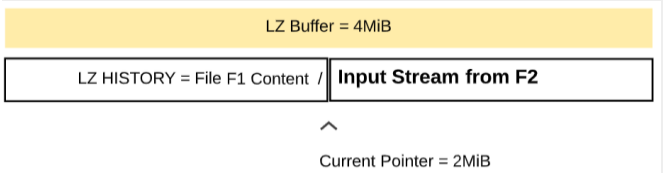


Fig. 6. KEOPS Algorithm starting point.

Fig. 6 shows the ideal scenario in which the entire content of the file f1 fits the LZ HISTORY size. Since this is an ideal case, we will have to ensure our algorithm works without this limitation. The main idea was to use the lower half of the LZ WINDOW area only for the file f1. Since we need to ensure that the whole file will be processed, an easier solution is to work with chunks of both files. We assume the LZ Window is 2MiB in size and the working chunks are 1MiB in size. This will be the layout of f1, 3MiB in size and f2, also 3 MiB in size. The current pointer will always be initialized at half of the LZ_WINDOW_SIZE, since the lower half which now belongs to file f1, has been already processed after the loading of the data. See Fig. 7

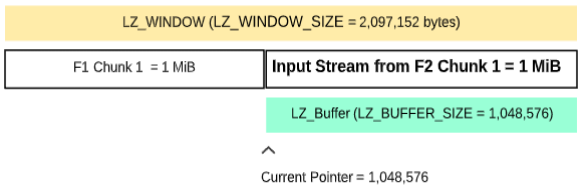


Fig. 7. KEOPS algorithm LZ History and LZ Lookahead buffer sizes for LZ Buffer Size = 2 MiB.

Instead of sliding the upper half of the LZ Buffer into the lower half of it, where the LZ History relies, we simply load

the next chunks from both files. After the first chunk of f2 is depleted, the next chunk from f1 is loaded. The condition for re-loading the chunks is completely related to the upper half of the buffer, since current_pointer only activates in the upper half of the LZ_BUFFER. When current_pointer reaches LZ_WINDOW_SIZE, a new chunk from f1 is loaded into the lower half of the LZ window and the next chunk from file f2 is loaded in the upper half. See Fig. 8

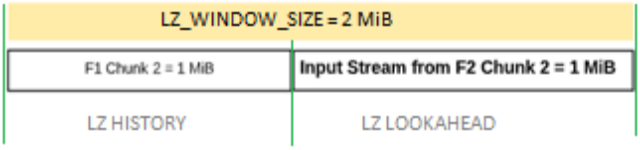


Fig. 8. Loading of the next chunk from file f1 into the lowe half of the LZ Window, the LZ History.

The process continues until file f2 is depleted meaning the current chunk size is less than the half of the LZ_Buffer. The representation in a more compressed form is in Fig. 9.

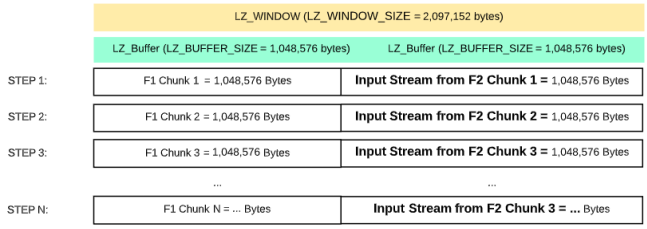


Fig. 9. For all steps 1 to K, chunks from f1 are read and processed into the lower half of LZ Buffer (the LZ History) and chunks from f2 are processed into the upper half of the LZ Window.

B. *Operating with KEOPS*

As noted above, we are devising an algorithm that works on limited resource environments therefore, the input stream is sliced into buffers or chunks. Furthermore, we need to define a similarity function which in our terms, means how many substrings the two buffers have in common, as a percentage (1 to 100). We call our function BEST. BEST(xy) returns the compression ratio when chunks (buffers) x and y are concatenated and encoded as a single buffer. We rewrite the function BEST, such as BEST(x) returns an index k where y = kth buffer when BEST(xy) returns the highest compression ratio in the above condition. BEST(x) = k where y = k^{th} buffer such as LZ77(xy) returns the best compression ratio. We replace Z(xy) with our BEST function, so our NCD equation becomes:

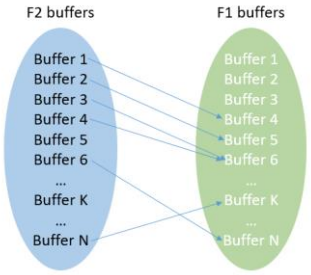
$$NCDz(x,y) = \frac{BEST(xy) - min\{Z(x), Z(y)\}}{max\{Z(x)\ Z(y)\}}$$

This is our called similarity function. See ref [9].

We note the first function, BEST(data_buffer_k) = i. That is for the kth buffer of file f2, the most similar buffer is buffer ith. The higher the similarity of the two buffers, the higher the compression ratio when compressing the two buffers.

C. *We are now able to ink down all KEOPS steps.*

We have defined our N and K values for our empirical studies and noted that N = 2MB and K=1MB tend to give the higher speed at no expense of good results.



$$Fkeops(data\_buffer\_k) = i$$

Fig. 10. Function BEST for an arbitrary index.

Step 1: Define N and K values; N is the LZ Buffer size, K is the size of the chunks;

Step 2: Split File f1 into chunks;

Step 3: Calculate BEST(1) = i and read the ith chunk from file f1;

Step 4: Split File f2 into chunks and read the first chunk from file f2;

Step 5: Start processing buffer 1 of f2;

Step 6: Finalize encoding of buffer k^{th} from f2 with buffer i^{th} from file f2, given by the function BEST(k) = I;

Step 7: Repeat Step 6 until f2 is depleted;

In a nutshell, KEOPS algorithm is in fact best dscribed by the following equation:

LZ77(buffer $_{f2}$(k), buffer $_{f1}$(BEST(buffer $_{f2}$(k)))

where the prefered ENCODE function is the LZ77 algorithm and BEST is the resultant of the similarity functions, Fkeops based on the selected strategy.

D. *Strategies for the BEST (Fkeops) function*

We have devised three strategies for the BEST function, as follows:

A. *Strategy 1: The 1-to-1 strategy.*

This is the fastest method which sometimes achieves a good result. If a better results is expected, the other two strategies should be considered. In this 1 to 1 strategy, we simply assume that Best(k) = k, which means that the best buffer to encode buffer k from file f2 is the kth buffer from file f1. The only problem in this strategy is when file f1 is smaller than file f2 file size, which in this case the formula that we use is Best(k) = k % (Z(f1) / K) where k is the index of the buffer, the kth buffer and K is the chunk (buffer) size.

B. *Strategy 2: The Brute-Force strategy*

In this strategy, hence the name, we compute all possible values of similarities between buffers. This is proven by design to achieve the best possible result. For any buffer k of file f2, we try out all possibilities by applying the similarity function for all chunks of file f1.

Best(k) = MAX (Simi(k, 1..N))

C. *Strategy 3: The eXtreme Strategy*

In this strategy we analyze file f2 chunks against other chunks from file f2 and also against chunks from file f1.

D. TEST RESULTS

We tested our KEOPS strategies to see how it compares when real data sets are applied. We have also tested out all three KEOPS strategies and . The data set 1 is made up of log files which have been collected in two separate ocasions, in f1 and f2. Second data set consists of two versions of mingw software application, version 4.4 and 4.5. Third data set consists of a windows software package, each representing different versions of the same software. The difference in size is considerable.

TABLE I. KEOPS RESULTS BASED ON DIFFERENT LZ_WINDOW_SIZE

	KEOPS Optimal Strategy on different LZ_WINDOW_SIZE values				
Data Sets	***F1 Size***	***F2 Size***	***4 MiB***	***8 MiB***	***16 MiB***
Log files	213,403,728	211,206,693	3,460,306	1,548,156	**655,420**
Mingw Binaries	116,858,661	159,723,368	75,523,864	73,232,888	**69,581,586**
SW Package	340,375,413	319,794,721	119,374,426	63,878,617	**54,932,363**

By increasing LZ WINDOW_SIZE, KEOPS is able to find more similarities between chunks of data from f1 and f2. Increasing the size implies less chunks (or buffers) therefore less substrings are shared between chunks. This leads to fewer instructions to describe a common substring(match) between chunks.

TABLE II. HOW KEOPS STRATEGIES COMPARE

	KEOPS Strategies on LZ_WINDOW_SIZE = 16 MiB				
Data Sets	***F1 Size***	***F2 Size***	***The optimal Strategy***	***Brute Force***	***eXtreme***
Log files	213,403,728	211,206,693	655,420	655,459	**655,459**
Mingw Binaries	116,858,661	159,723,368	69,581,586	68,662,418	**49,239,565**
SW Package	340,375,413	319,794,721	119,374,426	87,559,537	**86,913,955**

On the log files data sets, the optimal strategy achieves a 99.69% reduction in size as compared to file f2. Changing the stratey to brute force strategy or eXtreme, we are able to do more comparisons between f1 chunks and f2 chunks, which lead to better results and a smaller data differencing file.

E. Comparisons with commercial applications

We compare KEOPS with commercial application including DeltaMAX (© Indigo Rose Corporation) and X3Delta.

Data Sets	% vs f2		
	KEOPS	DeltaMAX	X3 Delta
Log files	**0.41**	0.35	0.12
Mingw Binaries	**30.83**	63.96	46.55
SW Package	**54.42**	63.07	51.21

KEOPS is able to deliver results even 50% better than established comercial data differencing software. The results on the windows sw package show a 13.7287% smaller delta file than Indigo Rose's DeltaMAX software. When applied on Mingw compiler binaries, KEOPS results are with more than 33% better than DeltaMAX.

ACKNOWLEDGMENT

The work has been funded by the Operational Programme Human Capital of the Ministry of European Funds through the Financial Agreement 51675/09.07.2019, SMIS code125125.

REFERENCES

[1] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding" IEEE Trans. on Information Theory, IT-24(5):5306, September 1978

[2] Calude, C.S. (1996). "Algorithmic information theory: Open problems"

[3] Vitanyi, P. "Obituary: Ray Solomonoff, Founding Father of Algorithmic Information Theory"

[4] "Delta Algorithms: An Empirical Analysis", James J. Hunt University of Karlsruhe, Karlsruhe, Germany and Kiem-Phong Vo AT&T Laboratories, Florham Park, NJ, USA and Walter F. Tichy University of Karlsruhe, Karlsruhe, Germany.

[5] David G. Korn and Kiem-Phong Vo, "Engineering a Differencing and Compression Data Format" AT&T Laboratories – Research 180 Park Avenue, Florham Park, NJ 07932, U.S.A. dgk,kpv@research.att.com.

[6] James W. Hunt and M.D. McIllroy, "An algorithm for differential file comparison" Technical Report Computing Science Technical Report 41, Bell Laboratories, June 1976.

[7] Webb Miller and Eugene W. Meyers, "A file comparison program. Software|Practice and Experience" 15(11):1025{1039, November 1985.

[8] C.H. Bennett, P. Gacs, M. Li, P.M.B. Vitányi, and W. Zurek, Information Distance, IEEE Trans. Inform. Theory, IT-44:4(1998) 1407–1423

[9] Li, Ming; Chen, Xin; Li, Xin; Ma, Bin; Vitanyi, P. M. B. (2011-09-27). "M. Li, X. Chen, X. Li, B. Ma, P.M.B. Vitanyi, The similarity metric, IEEE Trans. Inform. Th., 50:12(2004), 3250–3264". IEEE Transactions on Information Theory. 50 (12): 3250–3264. doi:10.1109/TIT.2004.838101