Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Dirk Habich, Wolfgang Lehner

Conflict Detection-Based Run-Length Encoding: AVX-512 CD Instruction Set in Action

Erstveröffentlichung in / First published in:

2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW). Paris, 16.-20.04.2018. Springer, S. 96-101. ISBN 978-1-5386-6306-6.

DOI: http://dx.doi.org/10.1109/ICDEW.2018.00023

Final edited form was published in "2018 IEEE 34th International Conference on Data Engineering Workshops. Paris 2018", S. 96-101, ISBN 978-1-5386-6306-6
http://dx.doi.org/10.1109/ICDEW.2018.00023

Conflict Detection-based Run-Length Encoding — AVX-512 CD Instruction Set in Action

Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Dirk Habich, Wolfgang Lehner
Database Systems Group, Technische Universität Dresden
Dresden, Germany
{firstname.lastname}@tu-dresden.de

***Abstract*—Data as well as hardware characteristics are two key aspects for efficient data management. This holds in particular for the field of in-memory data processing. Aside from increasing main memory capacities, efficient in-memory processing benefits from novel processing concepts based on lightweight compressed data. Thus, an active research field deals with the adaptation of new hardware features such as vectorization using SIMD instructions to speedup lightweight data compression algorithms. Following this trend, we propose a novel approach for run-length encoding, a well-known and often applied lightweight compression technique. Our novel approach is based on newly introduced conflict detection (CD) instructions in Intel's AVX-512 instruction set extension. As we are going to show, our CD-based approach has unique properties and outperforms the state-of-the-art RLE approach for data sets with small run lengths.**

I. Introduction

The continuous growth of data volumes is still a major challenge for efficient data processing. This applies not only to database systems [1], [2], but also to other areas, such as information retrieval [3], [4] or machine learning [5]. With growing capacities of the main memory, efficient analytical in-memory data processing becomes viable [1], [2], [6]. However, the gap between computing power of the CPUs and main memory bandwidth continuously increases being now the main bottleneck [1]. To overcome this issue, the mentioned application domains have a common approach: (i) encode values of each data attribute as a sequence of integers using some kind of dictionary encoding [7], [8] and (ii) apply lightweight lossless data compression to each sequence of integers. Besides reducing the amount of data, operations can be directly performed on compressed data [5], [7], [9].

For the lightweight lossless compression of a sequence of integers, a large corpus of algorithms has been developed [7], [3], [4], [10], [11], [12], [13], [14], [15]. In contrast to heavyweight algorithms, like arithmetic coding [16], Huffman [17], or Lempel Ziv [18], lightweight algorithms achieve comparable or even better compression rates [7], [3], [4], [10], [11], [12], [13], [14], [15]. Moreover, the computational effort for (de)compression is lower than for heavyweight algorithms. To achieve these unique properties, each lightweight compression *algorithm* employs one or more basic compression *techniques* such as frame-of-reference [10], [12], run-length encoding (RLE) [7], [14] or null suppression [7], [14], that allow the appropriate utilization of contextual knowledge like value distribution, sorting, or data locality. In particular, RLE is the only technique tackling uninterrupted sequences of occurrences of the same value, so called runs. In its compressed format, each run is represented by its value and length. Thus, the compressed data is a sequence of such pairs.

In recent years, the efficient *vectorized* implementation of these algorithms using SIMD (Single Instruction Multiple Data) instructions has attracted a lot of attention [4], [13], [19], [20], since it further reduces the computational effort. Generally, SIMD extensions such as Intel's SSE (Streaming SIMD Extensions) or AVX (Advanced Vector Extensions) have been available in modern processors for several years. SIMD instructions apply one operation to multiple elements of so-called *vector registers* at once. The available operations include parallel arithmetic, logical, and shift operations as well as permutations. Although the vectorization of RLE by means of these common operations is possible [21], [22], this implementation shows poor performance for sequences of integers with small run lengths as already shown in [21]. The reason is that uninterrupted sequences of occurrences of the same value have to be determined and this data dependency within the input sequence makes vectorization challenging. To overcome that, we developed a novel vectorization concept for the compression part of RLE using new *conflict detection (CD) instructions* which have been introduced in Intel's AVX-512 instruction set extension. In detail, our **main contributions** in this paper are:

1) In Section II, we recap the state-of-the-art RLE vectorization concept as presented in [21], [22]. Based on that, we clearly describe the shortcomings of this concept. As we are going to show, these shortcomings increase with increasing vector sizes, which is a current hardware trend.
2) Section III starts with a description of the newly introduced *conflict detection (CD)* instructions in detail. Then, we describe the application of these *CD* instructions for the RLE encoding of a sequence of integers.
3) We exhaustively evaluated our *CD*-based RLE approach to clearly show the benefits and the unique properties of our approach. In Section IV, we highlight selective results of this evaluation. In particular, we will show that our *CD*-based implementation concept is up to 3.2 times faster for sequences of integers with short run lengths.

Finally, we review related work in Section V. Then, we conclude the paper by summarizing our lesson learned in Section VI.

Final edited form was published in "2018 IEEE 34th International Conference on Data Engineering Workshops. Paris 2018", S. 96-101, ISBN 978-1-5386-6306-6
http://dx.doi.org/10.1109/ICDEW.2018.00023

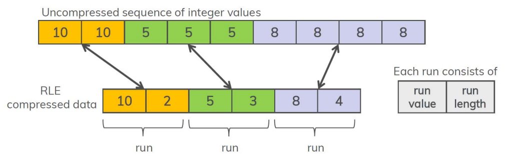

Fig. 1. Example for input and output of RLE compression.

II. State-of-the-Art Vectorization of RLE

Run-Length Encoding (RLE) is a well-known lightweight compression technique [7], [14] and Fig. 1 shows an example. As illustrated, uninterrupted sequences of occurrences of the same integer value in the input data are represented by a run value and a run length in the compressed output format in a lossless way. As we can see, a compression at run level starting at run length 3 is achieved, while the compression rate improves with increasing run lengths. That means, as long as the average run length over all runs is greater than 2, we get a data size reduction with RLE. In our example, the average run length is 3 with a variance of 1. Thus, these data properties mainly influence the achievable compression rate.

A. Comparison-based Vectorization

In this paper, we mainly focus on the compression part, because the state-of-the-art comparison-based vectorization has shortcomings in this area. Generally, to compress a sequence of integers with RLE, the corresponding runs have to be determined and this can be done by comparing each element with its predecessor. If they are equal, a run continues. If they are not equal, a new run starts. These comparisons can be done for more than one element at once using SIMD instructions as shown in [21], [22]. In detail, this state-of-the-art RLE comparison-based vectorization works as follows, whereby the authors used 128-bit vector registers:

1) One 128-bit vector register v_1 is loaded with four copies of the current input element.
2) The next four input elements are loaded into a vector register v_2.
3) The intrinsic `_mm_cmpeq_epi32()` is employed for a parallel comparison, so that the four elements in v_1 and v_2 are pair-wise compared at once. The result is stored in a vector register.
4) Next, a 4-bit comparison mask is obtained using the intrinsic `_mm_movemask_ps()`. Each bit in the mask indicates the (non-)equality of two corresponding vector elements. The number of trailing one-bits in this mask is the number of elements for which the run continues. If this number is 4, then a run's end has not been reached and the execution continues at step 2 (new iteration). Otherwise, a run's end is reached that means that run value and run length are appended to the output. The execution continues with step 1 at the next element after the run's end (new iteration).

The execution behavior of this vectorization concept is depicted in Fig. 2 for two different data sets. A detailed description follows in the next section. In the remainder of this paper, we refer to this 128-bit implementation as RLE128. Since only common intrinsics are used, this comparison-based implementation can easily be adapted to 256 and 512 bit-wide registers by loading more elements in the wider registers and by using the appropriate intrinsics of AVX2 (256 bit) or AVX-512. Additionally, step 3 and 4 can be merged into one step in AVX-512, because there is an intrinsic producing a bitmask directly from the comparison. The corresponding implementations are denoted as RLE256 and RLE512.

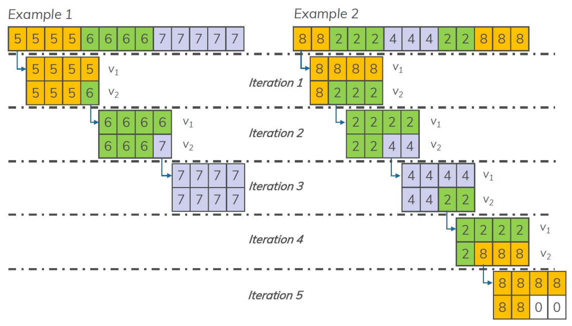

Fig. 2. Execution behavior of the comparison-based implementation. As illustrated, the number of necessary iterations depends on data characteristics.

B. Shortcomings of this Comparison-based Vectorization

As already mentioned, Fig. 2 highlights the resulting execution behavior for two different input sequences of integers. Both have in common that in each iteration, four integers are loaded and compared with a vector containing the current run value. If this comparison for equality is not true for all elements, the current run ends. In this case, the register with the run value is filled with four copies of the new value and the next four elements after the beginning of the new run are loaded. Obviously, the number of necessary iterations is data dependent and Fig. 2 shows that clearly. In detail, *Example 1* in Fig. 2 depicts a *fully vectorized* execution behavior. Fully vectorized means that each integer value is only processed once. In contrast to that, in *Example 2* several integers are loaded and compared multiple times. The redundant processing is usually negligible as long as the overhead is not dramatic.

To analyze the magnitude of this redundant processing, we counted the load instructions for different average run lengths and all possible variances for each average run length, whereby we used an input sequence with 100 million integers in all experiments. For instance, the maximal variance for an average run length of 5 is ± 4 resulting in the interval $[1, 9]$ for the possible run lengths. Then, we selected the minimal and the maximal number of load instructions and visualized them in Fig. 3(a) for RLE128, RLE256, and RLE512. The x-axis shows the average run length and the y-axis shows the number of loaded elements as a percentage of the elements in the input sequence, e.g. 200% means that on average every element is loaded twice. The colored area shows the range between the maximal and minimal number of load instructions. Fig. 3(b) shows a close up of Fig. 3(a) with the y-axis ranging only until 200%. From these experiments, we can conclude:

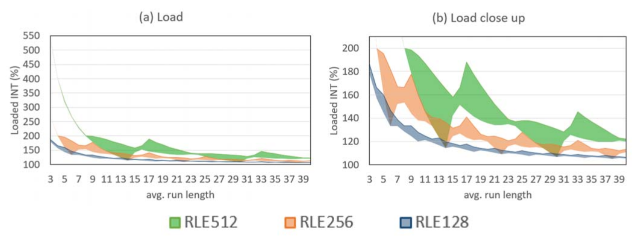

Fig. 3. (a) The number of loaded integers as a percentage of the integers in the uncompressed data set. Depending on the vector width, the average run length, and the variance of the run length, the number of loaded integers differs heavily. In particular for data sets with small average run lengths. (b) A close up of (a) to show the repeating pattern at every vector size.

1) The state-of-the-art RLE vectorization uses a significantly higher number of load operations for sequences with short runs than for sequences with long runs.
2) The redundant processing dramatically increases with increasing vector widths. For example, RLE512 processes each element 5 times on average when the average run length is 3. Furthermore, not only the absolute number increases, but also the size of the covered area grows.
3) There is a pattern with a minimum and a maximum spanning exactly one vector width, which is repeated for every vector width (in number of elements).
4) For large run lengths, the number of loaded integers approaches more or less 100%, i.e. every value is only processed once, which is the optimal scenario.

C. Impact on Performance

The presented high proportion of redundancy for sequences with small runs has a negative effect on the performance–measured in million integers per second (mis)–as illustrated in Fig. 4(a). In this experiment, we used again input sequences with 100 million integers and varied the average run length, whereby we used a fixed variance of ± 5. However, the results were the same for other data characteristics. As shown, only run lengths, which are greater than ~ 150 reach the peak performance for all vector widths, while small run lengths reach only a fraction of the peak performance. Additionally, the performance increases not even smoothly for RLE512. This becomes more obvious when looking at the speed up in Fig. 4(b). The speed up of RLE512 compared to RLE128 increases until a run length of ~ 8 is reached and decreases afterwards. The sampled run lengths in this region are 20 and 36, both being shortly after a maximum load number in Fig. 3. For larger run lengths, the number of loaded values becomes smaller and the speed up becomes constant.

III. Conflict Detection-based Vectorized RLE

Intel's latest version of their vectorization extension is AVX-512. In addition to an increased vector width of 512-bit (16 x 32-bit), AVX-512 also offers a variety of new instructions. One of the new instruction feature sets is called *Conflict Detection* (AVX-512 CD) which allows the vectorization of loops with possible address conflicts. This instruction feature

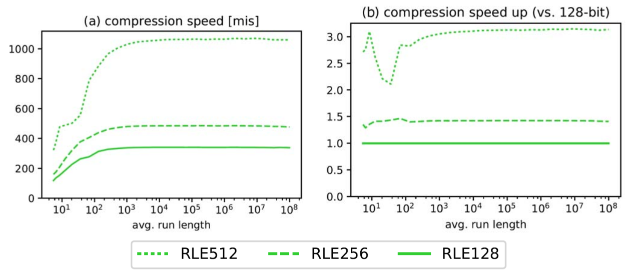

Fig. 4. RLE compression speed and speed up for different average run lengths, a fixed run length variance of ± 5, and different vector widths.

set is currently supported by Intel Xeon Phi Knights Landing (KNL) and will be available in future Xeon processors.

Some key features of AVX-512 CD are (i) the generation of conflict free subsets, i.e. subsets which contain no equal elements, and (ii) the count of leading zeros of the elements in a vector. For example, the intrinsic `_mm512_conflict_epi32` creates a vector register containing a conflict free subset of a given source register. An example for this is shown in Fig. 5. In other words and as illustrated in this figure, this intrinsic transforms a vector register with 16 32-bit elements (illustrated by A, B and C) in a new vector register with 16 bitmasks (each represented by 32-bit values). Each bitmask encodes the positions of equal previous elements in the vector. The bitmasks for the first three elements A, B, and C are zero in our example, because there are no equal previous elements. The A element at the third position in the input register is in conflict (equal to) with the element at position 0 in the input register. Thus, the least significant bit of the corresponding bitmask is set to 1, the rest of the bitmask is filled with zeros. The element A at position 4 is in conflict with the previous elements at positions 3 and 0 (equal previous elements). Therefore, the corresponding bits in the bitmask are set to 1, all other bits are zero. Another CD-feature is the intrinsic `_mm512_lzcnt_epi32`, which counts leading zeros. Given a vector of 16 values, this intrinsic counts the number of leading zeros for all values at once and writes the results in a vector register with 16 values.

A. RLE Implementation Concept with AVX-512 CD

To overcome the presented shortcomings of the comparison-based RLE vectorization, our novel approach–called *RLE512-CD*–uses the conflict detection innovations of AVX-512 in

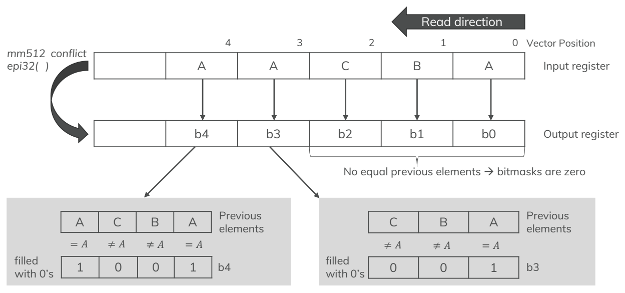

Fig. 5. Example for the *_mm512_conflict_epi32* intrinsic.

Final edited form was published in "2018 IEEE 34th International Conference on Data Engineering Workshops. Paris 2018", S. 96-101, ISBN 978-1-5386-6306-6
http://dx.doi.org/10.1109/ICDEW.2018.00023

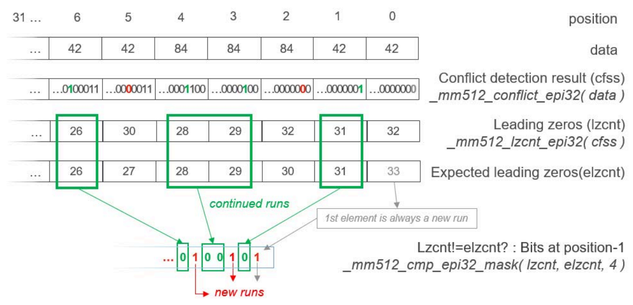

Fig. 6. Run detection using *conflict detection* instructions

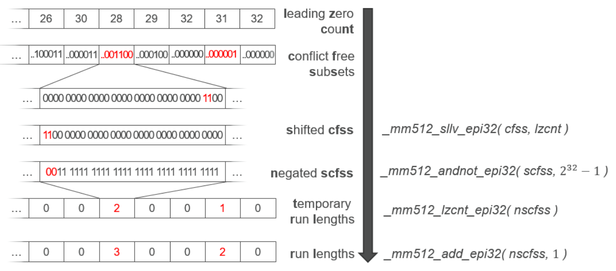

Fig. 7. Run length determination using *conflict detection* instructions.

an appropriate way. Generally, our novel approach consists of four steps, which are repeated until all input elements are processed:

Loading Step: In this first step, 16 input elements are loaded into a 512-bit vector register.

Run Detection Step: In the second step, we detect if there are any runs beginning in this register and where they begin.

Run Length Detection Step: The run length of all finished runs have to be determined in the third step.

Storage Step: The determined runs are written to memory.

While the *loading step* is trivial, the steps 2-4 are explained in more detail below.

B. Run Detection Step

To avoid any redundant element processing for small run lengths, the main challenge of this step is to detect *all* runs included within the loaded 16 elements. This challenge can effectively be realized using the AVX512-CD innovations as illustrated in Fig. 6:

In the first sub-step, we create a new vector register containing a **c**onflict **f**ree **s**ub**s**et (*cfss*) of the given source register with the 16 loaded elements using the `_mm512_conflict_epi32` intrinsic. The example in Fig. 6 shows the first 7 values of a vector register containing two different values spread over 3 runs. As described above, the newly created vector register consists of 16 bitmasks, where each bitmask shows the equality to all previous elements. However, for detecting a run it is sufficient to know if the direct predecessor of an element is equal because all elements are either the beginning of a new run or the continuation of another run. If an element is equal to its direct predecessor, the element continues a run. If they are not equal, a new run starts. Hence, only one bit in every bitmask of *cfss* is of interest, i.e. the bit which indicates the equality with the direct predecessor. To find this bit for all elements in parallel, two more operations are necessary:

First (second sub-step), we count the leading zeros of all bitmasks in *cfss* (*lzcnt*). The number of leading zeros should decrease with every element if a run is continued because there is always one more bit set in the subsequent element, e.g. the bitmask at position 1 should have $32 - 1 = 31$ leading zeros, the bitmask at position 2 should have $32 - 2 = 30$ leading zeros and so on. If a run is not continued, the next bit is not set and the number of leading zeros does not decrease. To find out, if the number of leading zeros is decreasing, we compare *lzcnt* with a predefined vector, containing decreasing numbers, for inequality (third sub-step). As shown in Fig. 6, this comparison returns 0 for every element which continues a run. Vice versa, it returns 1 for all elements which start a new run. Thus, the position of the ones in the final bitmask indicates the position of the start of all runs in this register. Note that the first element always starts a new run.

C. Run Length Detection Step

With the previous step, we know the start positions and the run values of all runs within the register. The next challenge is to determine the run length of each run. Fundamentally, the run length is already encoded in the results of the conflict detection (*cfss*) operation, because each continuous sequence of 1s in the bitmasks indicates a subsequent occurrence of equal numbers. Hence, the number of the most significant subsequent 1s in the bitmask of every last element of a run indicates the length of the run. To get this number, at first the position of the last element of every run has to be determined. This can be done by using the bitmask generated as the result of the run detection (*cfss*). Since every 1 in this bitmask indicates the beginning of a new run, we can get the end of the runs by shifting this mask one bit to the right. Now every 1 indicates the end of a run. Then, the bitmasks at these end positions in the output of the conflict detection (in *cfss*) are selected. In Fig. 7, which continues the example from Fig. 6, one bitmask is selected as an example. In this example the second run, consisting of 3 elements is treated. The continuous sequence of 1s is highlighted. There are only 2 instead of 3 set bits because the bit of the first element, i.e. the least significant element, of a run is always set to 0. We will add this bit later. In order to retrieve the number of subsequent set bits in this bitmask, 3 sub-steps are executed:

1) Shift the elements in the result of `_mm512_conflict_epi32` by the number of leading zeros (leading zeros were derived during run detection). In Fig. 7 we shift by 28 bits. Now, the sequence is at the beginning of the bitvector.
2) There is no intrinsic for counting leading 1s, so the result from the previous sub-step is inverted.
3) Then, the leading zeros are counted in the third sub-step. In the example, there are two leading zeros.

Since the bit for the first element of a run is always set to

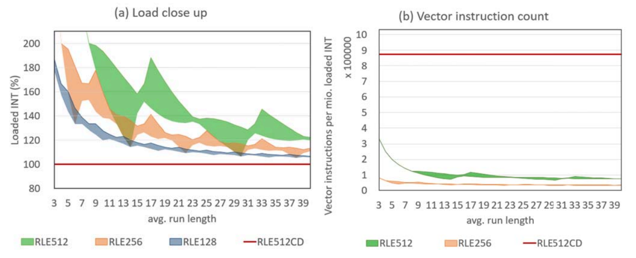

Fig. 8. (a) The number of loaded integer as a percentage of the integers in the data set. Only *RLE512-CD* shows a constant behavior. (b) The number of vector instructions per million loaded integers (excluding loading and storing) is significantly higher for *RLE512-CD* compared to RLE512 and RLE256. This shows that the lower number of loaded integers do not come for free.

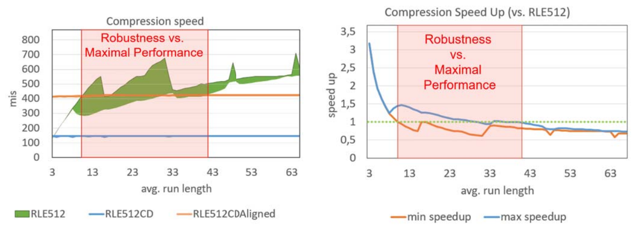

Fig. 9. (a) The compression speed for RLE512 varies depending on the run length and the variance of the run length, while our novel implementation shows a constant compression speed. The costs of the scatter store for RLE512CD are clearly visible. (b) The minimum and maximum speedup for RLE512-CDAligned compared to RLE512.

0 during conflict detection, the result has to be increased by 1. Hence, the run length for the second run is $2+1=3$. In our implementation, these steps are executed in parallel for all runs by using the intrinsics shown in Fig. 7.

D. Storing the Result

Before storing the results, it must be checked whether the first run of a register is a continuation of the last run of the previous register. If it is a continuation, the run lengths are added and the run is stored once. While these are trivial steps, we avoided branching by applying bitmasks instead of conditions, i.e. the comparison between the last and the first run of two registers returns whether the last bit in a bitmask is set. This bitmask is then used to add the run lengths by applying `_mm512_mask_add_epi32`, which adds the content of two registers only if the corresponding bit in the bitmask is set.

Finally, the run values and run lengths must be written back to main memory. For this, there are two possible cases: (a) per integer or (b) per vector. Case (a) represents the output format proposed by the state-of-the-art implementation [21], where a sequence of (value, run length)-tuples is stored. An advantage of option (a) is that the output is independent from the vector word size. The disadvantage is that the values and run lengths cannot be loaded sequentially into a vector register again for processing the compressed values, e.g. for aggregating. Case (b) stores sequences of values and run lengths which are as long as a vector word, e.g. 16 values followed by 16 run lengths. Option (b) requires the vector word width as necessary meta data but it is also ideal for processing the compressed data with vector instructions. We implemented both cases, case (a) using a scatter store provided by AVX-512 and case (b) using the result of the *run detection* as a write mask and the `_mm512_mask_storeu_epi32` intrinsic.

IV. Evaluation

In this section, we evaluate and compare our novel CD-based RLE implementation (RLE512-CD) with the state-of-the-art implementation. For this evaluation, all implementations are done with C/C++ and we compiled them with g++ 7.0.1 using the optimization flag *-O3*. Then, all experiments were executed on a Intel Xeon Phi KNL 7250 with a main memory capacity of 192GB supporting all vector widths of 128 (SSE), 256 (AVX2), and 512 (AVX-512). The maximum core frequency is 1.6 GHz. Moreover, all experiments were performed with our benchmark framework [23] running entirely in main memory and single-threaded.

A. Data Processing Behavior

In Section II-B, we analyzed the processing behavior of the state-of-the-art implementation. For this analysis, we also used the above mentioned evaluation setting. As we have shown, this implementation suffers from a significantly higher number of redundant load operations for sequences of integer values with short average run lengths. In contrast to that, our novel *RLE512-CD* implementation is branch-free and every integer sequence value is only loaded once as illustrated in Fig. 8(a). The y-axis shows the number of loaded integers as the percentage of the integer count in the uncompressed data for RLE128, RLE256, RLE512, and for *RLE512-CD*. It is clearly visible that our novel implementation loads the input data set only once, independent of the data characteristics, and that the amount of loaded data is smaller than for the state-of-the-art implementation. Additionally, we observe that the difference is smaller when the average run-lengths are longer.

However, this constant data loading behavior comes at a cost. The total number of executed vector instructions of *RLE512-CD* is higher than for the state-of-the-art implementation. Fig. 8(b) shows the number of vector instructions per million loaded integers (excluding operations on masks and other scalar operations) for RLE512, RLE256 and *RLE512-CD*. Thus, it comes down to the number of loaded and processed integers versus the amount of executed instructions. Depending on the system, this can have different effects on the compression speed.

B. Performance

Fig. 9(a) shows the compression speed for RLE512 and RLE512-CD with two different storage options: RLE512-CD stores the result data integer-wise like RLE512 with a scatter store while RLE512-CDAligned stores the result vector-wise. Again, each run length has been tested with all possible run length variances. The first obvious finding is that RLE512-CD shows an almost constant compression speed as expected. However, the scatter store used in RLE512-CD is too slow to compete with RLE512. For RLE512-CDAligned, there are 3 different regions: (1) RLE512-CDAligned always outperforms RLE512 for very small run lengths (<12). (2) Between the run lengths of 11 and 40, there is no binary decision possible between RLE512 and RLE512-CDAligned. RLE512 shows the highest peak performance but also the lowest possible performance. RLE512-CDAligned does not reach the peak performance but guarantees a constant compression speed, i.e. it is robust. (3) for run lengths greater than 40, the state-of-the-art implementation always shows the highest compression speed. Hence, at the transitions of these regions, the applied implementation should be changed. Additionally, in region (2) a decision between maximal peak performance and robustness must be made.

The same regions as for the compression speed can be shown for the speed up in Fig. 9(b). Here, the base line is RLE512 and the maximal and the minimal speed up is shown for RLE512-CDAligned. The lower curve compares to the maximal compression speed of RLE512 and the upper curve compares against the minimal compression speed. The graph shows that the chances to gain a speed up greater than 1 are higher, the lower the run length is. As already mentioned, this graph can look different on another system, where the execution of vector instructions or the loading of a vector register is faster or slower. Additionally, in a multi-threaded scenario the loading of integers might become a bottleneck earlier, e.g. because of shared caches (focus of future research).

V. Related Work

The efficient utilization of SIMD (Single Instruction Multiple Data) instructions in database systems is a very active research field [24], [25]. On the one hand, these instructions are frequently applied in lightweight data compression algorithms [20]. In this domain, null suppression (NS) is the most studied lightweight compression approach, whereby the basic idea is the omission of leading zeros in the bit representation of integers [13], [19]. However, none of these approaches uses the leading zero count intrinsic of the *Conflict Detection* feature set of AVX-512. The application would be very interesting and should be definitely investigated. On the other hand, SIMD instructions are also used in other database operations like scans [26], aggregations [25] or joins [27]. To best of our knowledge, none of these approaches uses AVX-512 CD, although the operations could benefit from CD.

VI. Conclusion

In this paper, we described the newly introduced *conflict detection (CD)* instructions which are available in Intel's AVX-512 instruction set extension. Furthermore, we proposed the application of these CD instructions to speedup the RLE compression of sequences of integers with small run lengths. However, our novel concept is not suitable for integer sequences with long run lengths because too many instructions have to be executed in comparison to the state-of-the-art approach. Nevertheless, new instructions in addition to wider vectors are useful, but they require new implementation concepts.

Acknowledgments

This work is partly funded by the German Research Foundation (DFG) in the Collaborative Research Center 912 and by the individual DFG project LE-1416/26-1.

References

[1] P. A. Boncz, M. L. Kersten, and S. Manegold, "Breaking the memory wall in monetdb," *Commun. ACM*, vol. 51, no. 12, pp. 77–85, 2008.
[2] H. Plattner, "A common database approach for OLTP and OLAP using an in-memory column database," in *SIGMOD*, 2009, pp. 1–2.
[3] D. Arroyuelo, S. González, M. Oyarzún, and V. Sepulveda, "Document identifier reassignment and run-length-compressed inverted indexes for improved search performance," in *SIGIR*, 2013.
[4] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi, "Simd-based decoding of posting lists," in *CIKM*, 2011.
[5] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed linear algebra for large-scale machine learning," *PVLDB*, vol. 9, no. 12, 2016.
[6] T. Kissinger, T. Kiefer, B. Schlegel, D. Habich, D. Molka, and W. Lehner, "ERIS: A numa-aware in-memory storage engine for analytical workloads," in *ADMS*, 2014.
[7] D. J. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD*, 2006.
[8] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in *SIGMOD*, 2009, pp. 283–296.
[9] J. Hildebrandt, D. Habich, P. Damme, and W. Lehner, "Compression-aware in-memory query processing: Vision, system design and beyond," in *IMDM@VLDB*, 2016, pp. 40–56.
[10] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz, "Super-scalar RAM-CPU cache compression," in *ICDE*, 2006.
[11] V. N. Anh and A. Moffat, "Index compression using 64-bit words," *Softw., Pract. Exper.*, vol. 40, no. 2, 2010.
[12] J. Goldstein, R. Ramakrishnan, and U. Shaft, "Compressing relations and indexes," in *ICDE*, 1998.
[13] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Softw., Pract. Exper.*, vol. 45, no. 1, 2015.
[14] M. A. Roth and S. J. Van Horn, "Database compression," *SIGMOD Rec.*, vol. 22, no. 3, 1993.
[15] F. Silvestri and R. Venturini, "Vsencoding: Efficient coding and fast decoding of integer lists via dynamic programming," in *CIKM*, 2010.
[16] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, no. 6, 1987.
[17] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, 1952.
[18] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theor.*, vol. 23, no. 3, 1977.
[19] J. Plaisance, N. Kurz, and D. Lemire, "Vectorized vbyte decoding," *CoRR*, vol. abs/1503.07387, 2015.
[20] W. X. Zhao, X. Zhang, D. Lemire, D. Shan, J. Nie, H. Yan, and J. Wen, "A general simd-based approach to accelerating compression algorithms," *ACM Trans. Inf. Syst.*, vol. 33, no. 3, 2015.
[21] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner, "Lightweight data compression algorithms: An experimental survey (experiments and analyses)," in *EDBT*, 2017, pp. 72–83.
[22] A. Ungethüm, P. Damme, J. Pietrzyk, A. Krause, D. Habich, and W. Lehner, "Balancing performance and energy for lightweight data compression algorithms," in *ADBIS Short Papers*, 2017, pp. 37–44.
[23] P. Damme, D. Habich, and W. Lehner, "A benchmark framework for data compression techniques," in *TPCTC*, 2015.
[24] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking SIMD vectorization for in-memory databases," in *SIGMOD*, 2015, pp. 1493–1508.
[25] J. Zhou and K. A. Ross, "Implementing database operations using simd instructions," in *SIGMOD*, 2002, pp. 145–156.
[26] Z. Feng, E. Lo, B. Kao, and W. Xu, "Byteslice: Pushing the envelop of main memory data processing with a new storage layout," in *SIGMOD*, 2015, pp. 31–46.
[27] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," *PVLDB*, vol. 7, no. 1, pp. 85–96, 2013.