Dirk Habich, Patrick Damme, Annett Ungethüm, Wolfgang Lehner

Make Larger Vector Register Sizes New Challenges? Lessons Learned from the Area of Vectorized Lightweight Compression Algorithms

Erstveröffentlichung in / First published in:

SIGMOD/PODS '18: International Conference on Management of Data, Houston 15.06.2018. ACM Digital Library, Art. Nr. 8. ISBN 978-1-4503-5826-2

DOI: https://doi.org/10.1145/3209950.3209957

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 8, ISBN 978-1-4503-5826-2
https://doi.org/10.1145/3209950.3209957

Make Larger Vector Register Sizes New Challenges?

Lessons Learned from the Area of Vectorized Lightweight Compression Algorithms

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

ABSTRACT

The exploitation of data as well as hardware properties is a core aspect for efficient data management. This holds in particular for the field of in-memory data processing. Aside from increasing main memory capacities, in-memory data processing also benefits from novel processing concepts based on lightweight compressed data. To speed up compression as well as decompression, an active research field deals with the specialization of these algorithms to hardware features such as vectorization using SIMD instructions. Most of the vectorized implementations have been proposed for 128 bit vector registers. However, hardware vendors still increase the vector register sizes, whereby a straightforward transformation to these wider vector sizes is possible in most-cases. Thus, we systematically investigated the impact of different SIMD instruction set extensions with wider vector sizes on the behavior of straightforward transformed implementations. In this paper, we will describe our evaluation methodology and present selective results of our exhaustive evaluation. In particular, we will highlight some challenges and present first approaches to tackle them.

CCS CONCEPTS

• **Information systems → Data compression**; **Main memory engines**;

KEYWORDS

in-memory, database systems, lightweight data compression, vectorization, experimental evaluation

ACM Reference Format:
Dirk Habich, Patrick Damme, Annett Ungethüm, Wolfgang Lehner. 2018. Make Larger Vector Register Sizes New Challenges?: Lessons Learned from the Area of Vectorized Lightweight Compression Algorithms. In *DBTest'18: Workshop on Testing Database Systems , June 15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3209950.3209957

1 MOTIVATION

Data compression as well as vectorization are important keystones in modern state-of-the-art in-memory column store systems [1, 10, 14, 21, 22]. On the one hand, *data compression* is used to tackle the continuously increasing gap between computing power of CPUs and main memory bandwidth (also known as memory wall [5]) [1, 10]. On the other hand, *vectorization* is used to improve the processing performance by parallelizing computations over vector registers [14, 21]. This vectorization is done using SIMD extensions (Single Instruction Multiple Data) such as Intel's SSE (Streaming SIMD Extensions) or AVX (Advanced Vector Extensions) and have been available in modern processors for several years. SIMD instructions apply one operation to multiple elements of so-called *vector registers* at once.

State-of-the-art in-memory column stores have more or less a common compression approach: (i) encode values of each column as a sequence of integers using some kind of dictionary encoding [1, 4] and (ii) apply lightweight lossless data compression to each sequence of integers resulting in a sequence of compressed codes. For the lossless compression of sequences of values (in particular integer values), a large variety of lightweight algorithms has been developed [1–3, 8, 11, 15–18, 22]. Each lightweight compression *algorithm* employs one or more basic compression *techniques* such as frame-of-reference [8, 22] or null suppression [1, 15] allowing the appropriate utilization of contextual knowledge like value distribution, sorting, or data locality. In recent years, the efficient *vectorized* implementation of these lightweight compression algorithms has attracted a lot of attention [6, 11, 13, 16, 18, 20], since it further reduces the computational effort. M (corresponding to Intel's SIMD extension SSE).

However, hardware vendors have introduced new SIMD instruction set extensions operating on wider vector registers. For instance, Intel's Advanced Vector Extensions 2 (AVX2) operates on 256-bit vector registers[1] and Intel's AVX-512 uses even 512-bit vector registers. The wider the vector registers, the more data elements can be stored and processed in one vector. For example, while an SSE 128-bit vector register can store four uncompressed 32-bit data elements, an AVX2 256-bit vector can store eight (*2x*) and an AVX-512 512-bit vector can store 16 (*4x*) of such data elements. Consequently, the SIMD instructions on these wider vector registers can also process *2x* respectively *4x* the number of data elements *in one instruction*, which promises significant speed ups.

Our Contribution. To obtain implementations of lightweight data compression algorithms for wider vector sizes (AVX2 and AVX-512), the 128-bit implementation can be used as foundation. In a straightforward transformation, the 128-bit SIMD operations can

[1]Note that 256-bit vector registers had already been introduced with Intel's AVX. However, most instructions relevant to lightweight data compression were only introduced with AVX2.

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 8, ISBN 978-1-4503-5826-2
https://doi.org/10.1145/3209950.3209957

be substituted by the corresponding operations for 256 or 512-bit vectors. This is possible in almost all cases, since many instructions offered by SSE are also offered by AVX2 and AVX-512 *on wider vectors*. So far, there is no evaluation as to whether this transformation really exploits the potential of larger vector sizes. Thus, we systematically investigate the impact of different SIMD instruction set extensions with vector sizes of 128, 256, and 512 bits on the behavior of lightweight data compression algorithms. In detail, we make the following contributions in this paper:

(1) We start our description with an introduction of our evaluation methodology in Section 2.
(2) Based on this evaluation methodology, we present selective results of our evaluation in Section 3.
(3) Then, we summarize our lessons learned and present some thoughts about our ongoing research activities in this direction in Section 4.

Finally, we conclude the paper with related work in Section 5 and a summary in Section 6.

2 EVALUATION METHODOLOGY

The focus of this work is the large corpus of *lossless lightweight data compression algorithms* and to investigate the influence of wider vector registers on the behavior of these algorithms. Generally, the input of every lightweight compression algorithm is a finite sequence of uncompressed values (usually integer values [1, 22]) and the output is a compressed representation. The goal is to represent the input with as few as possible bits. To achieve that, each specific algorithm employs one or more basic compression techniques. There are currently five basic techniques known and frequently used: frame-of-reference (FOR) [8, 22], delta coding (DELTA) [11, 15], dictionary compression (DICT) [1, 22], run-length encoding (RLE) [1, 15], and null suppression (NS) [1, 15]. While FOR, DELTA, and DICT consider the mapping to smaller values, the goal of RLE is to reduce the number of values on the logical level, and NS addresses the *physical* level of bits or bytes to reduce the number of bits per value. This explains why every lightweight data compression algorithm can be usually described as a cascade of one or more of these basic techniques. In the following, we also denote the techniques FOR, DELTA, and DICT as preprocessing techniques.

Generally, the NS technique has been studied most extensively. There is a very large number of specific algorithms showing the diversity of the implementations for a single technique. The pure NS algorithms can be divided into the following classes [20]: (i) bit-aligned, (ii) byte-aligned, and (iii) word-aligned.[2] While bit-aligned NS algorithms try to compress an integer using a minimal number of *bits*, byte-aligned NS algorithms compress an integer with a minimal number of *bytes* (1:1 mapping). The word-aligned NS algorithms encode as many integer values as possible into one 32-bit or 64-bit word (N:1 mapping). The logical-level techniques have not been considered to such an extent as the NS technique *on the algorithm level*. In most cases, the preprocessing steps have been investigated in connection with the NS technique. For instance, PFOR-based algorithms implement the FOR technique in combination with a bit-aligned NS algorithm [22].

[2] [20] also defines a *frame-based* class, which we omit, as the representatives we consider also match the *bit-aligned* class.

2.1 Considered Implementations

An implementation of a compression algorithm is a hardware-specific executable code, whereby a *vectorized* implementation using SIMD instructions is state-of-the art. Most of the developed vectorized implementations have been developed for a fixed vector width of 128 bits (e.g., corresponding to Intel's SIMD extension SSE). For these 128-bit vectorized implementations, we already presented an experimental survey [6]. As we have shown, performance and compression ratio vary greatly depending on data properties. Even algorithms that are based in the same technique, show a very different behavior. In this underlying survey [6], we considered all five basic techniques, whereby we investigated implementations of a single technique as well as cascades of one preprocessing technique and one physical compression. In this survey, we decided to re-implement the logical-level techniques on our own in order to be able to freely combine them with all considered NS algorithms.

To obtain implementations of lightweight compression algorithms for wider vector register sizes (256-bit for AVX2 and 512-bit for AVX-512), we did a *straightforward* re-implementation of most of the vectorized algorithms used in [6]. By a straightforward re-implementation, we mean that we tried to stick with the original source code for 128-bit instructions as much as possible and applied only intuitive changes. In particular, we mainly substituted the SSE intrinsics for 128-bit vectors by the corresponding AVX2 or AVX-512 intrinsics for 256 or 512-bit vectors, respectively. This is possible in many cases, since many instructions offered by SSE are also offered by AVX2 and AVX-512 *on wider vectors*.

2.2 Evaluation Setup

We compiled our C++ source code using `g++-7.0.1` with the optimization flag `-O3`. All experiments have been executed on the same hardware machine to be able to compare the results. To evaluate the influence of the vector width on the behavior of lightweight compression algorithms, an evaluation platform supporting SSE, AVX2, *and* AVX-512 is required. This holds only for very recent Intel processors. We choose a system equipped with an Intel Xeon Phi 7250 with 68 active cores, each of them running up to 4 hyperthreads and with 196GB DDR4 RAM, whereby all experiments ran single-threaded. The cores support a frequency scaling ranging from 1 GHz to 1.6 GHz. Each core features a 32kB L1 instruction cache and a 32kB L1 data cache. These cores are organized on tiles and each tile contains two cores, two vector processing units, and a 1MB L2 cache, which is shared between both cores. The tiles are connected by a 2D mesh cache-coherent interconnect, such that each row and column forms a half ring. The chip features six DDR4 channels, which operate at a clock speed of 2400 MHz. While earlier generations of the Intel Xeon Phi were designed as co-processors only, the model we employ is of the most recent generation at the time of this writing and is as the main processor of a system.

All experiments happened entirely in main memory. The whole evaluation is performed using our specific benchmark framework [7] using synthetic datasets. During the executions, the single-threaded runtimes—reported in million integers per second (mis)—and the compression ratio were measured. Furthermore, we emptied the cache before each algorithm execution. All time measurements were carried out by means of the wallclock-time and were repeated 12

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 8, ISBN 978-1-4503-5826-2
https://doi.org/10.1145/3209950.3209957

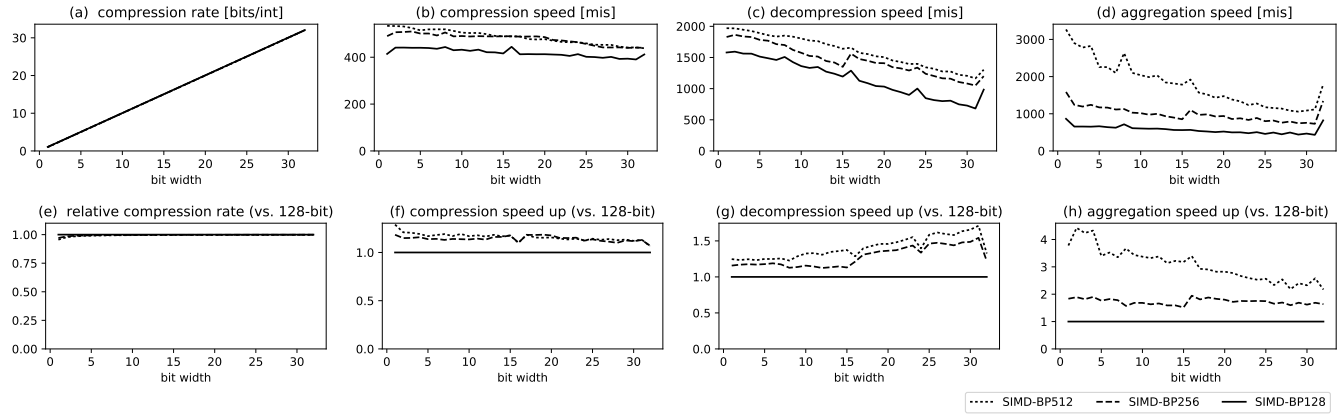

Figure 1: The variants of SIMD-BP on dataset $D0$. Row 1 reports absolute measurements, while row 2 reports the measurements of each algorithm variant relative to the measurement of the classical 128-bit variant.

times to receive stable values, thereby we only report average values. Aside from time measurements for compression and decompression, we measured also the time for an aggregation (summation).

3 EVALUATION RESULTS

In this section, we present some selective results of our evaluation with regard to various algorithms and different vector register sizes. Based on that, we draw lessons learned in the following section.

3.1 NS Compression Techniques

We start with results for a well-known and well-performing bit-aligned null suppression algorithm called *SIMD-BP128* [11], which is available in the FastPFor-library [12]. The original *SIMD-BP128* approach subdivides the data into blocks of 128 integers each. For each such block, the number of bits required for the largest element is determined. Then, all 128 integers in the block are stored using the vertical layout with that many bits for each value. The used bit width is stored in a single byte, whereby 16 of these bit widths are followed by 16 compressed blocks.

The original 128-bit implementation is based on SSE shift and mask operations on 128-bit vector registers, for each of which there are equivalent operations for 256-bit and 512-bit vectors in AVX2 and AVX-512, respectively. Thus, the SIMD intrinsics are exchanged in a straightforward way. As a consequence, while SIMD-BP128 determines a common bit width for a block of 128 data elements at a time, SIMD-BP256 and SIMD-BP512 determine the bit width for a block of 256 and 512 integers at a time, respectively. SIMD-BP128 uses one byte to store the bit width used for a particular block. In memory, 16 of these descriptor bytes are stored subsequently and are followed by 16 compressed blocks. This is necessary, because SSE's load and store instructions require an alignment of 16 bytes. Since this alignment requirement naturally increases to 32 and 64 bytes in AVX2 and AVX-512, respectively, we had to slightly and intuitively adapt the storage format for our re-implementations. In the formats of SIMD-BP256 (SIMD-BP512), 32 (64) descriptor bytes are stored subsequently followed by 32 (64) compressed blocks.

Figure 1 (a,e) show the results for the compression ratio of the variants of SIMD-BP on dataset $D0$. Dataset $D0$ contains 32 unsorted generated single datasets, such that all data elements in the i-th dataset have exactly i effective bits, i.e., the value range is $[0, 1]$ for $i = 1$ and $[2^{i-1}, 2^i)$ for $i = 2, \ldots, 32$. That is, we effectively vary the number of bits required by the values, which is the key factor involved in NS-algorithms. Within these ranges, the 100 million 32-bit integer values are uniformly distributed. Thus, all three variants of SIMD-BP yield *nearly* the same compression ratio, since each of them can perfectly adapt to the bit width in the same way. However, the blocks of SIMD-BP256 and SIMD-BP512 are twice respectively four times as large as the blocks of SIMD-BP128. At the same time, all three variants need the same amount of meta data, namely one byte, per block. Thus, the compression ratios achieved by SIMD-BP256 and SIMD-BP512 are in fact *minimally* better than those of SIMD-BP128. Regarding the performance (Figure 1 (b-d)), we can make the general observation that the speed increases as the vector width is increased. Figure 1 (f-h), show the speed ups of each variant compared to SIMD-BP128. With respect to the (de)compression performance, it becomes visible that the performance gains achieved through the use of wider vector registers are lower than they could be expected. In the ideal case, SIMD-BP256 and SIMD-BP512 could yield speed ups of 2x and 4x compared to SIMD-BP128, respectively. However, the true speed ups are far less than that for both, the compression and the decompression. Moreover, it is worth noting that, while SIMD-BP256 is significantly faster than SIMD-BP128, SIMD-BP512 can add only little to the speed of SIMD-BP256. The use of SIMD extensions with wider vector registers enables faster computations. As a consequence, the algorithms become increasingly memory-bound, which explains the sub-optimal speedups. However, if we have a look at the aggregation speeds and speedups in Figure 1 (d, h), we observe that SIMD-BP256 and SIMD-BP512 can indeed reach the expected speedups of about 2x and 4x respectively, at least for small bit widths. Unlike the decompression, the aggregation does not store the decompressed values to memory, but adds them to a running sum. Thus, the aggregation is generally rather compute-bound then memory-bound and can, in consequence, profit much more from AVX2 and AVX-512. Finally,

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 8, ISBN 978-1-4503-5826-2
https://doi.org/10.1145/3209950.3209957

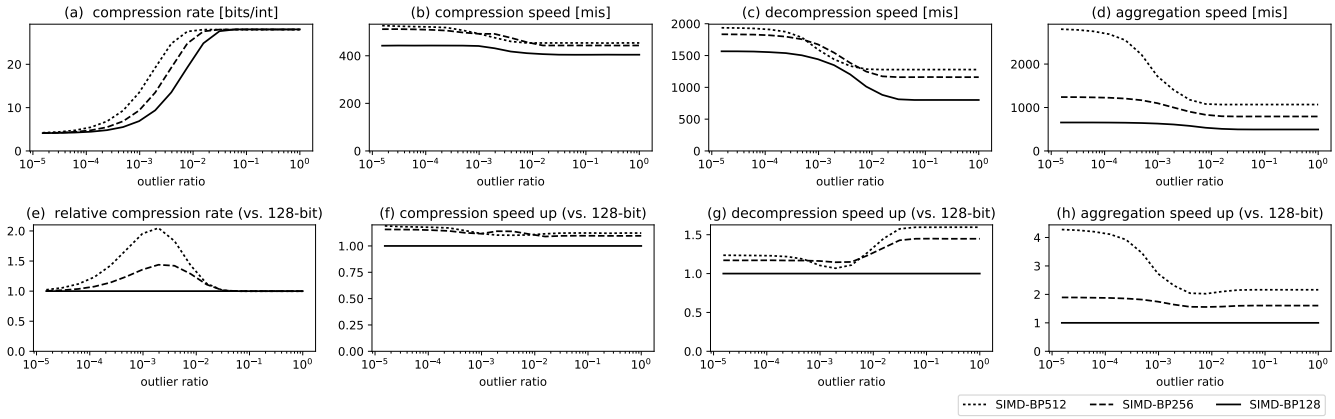

Figure 2: The variants of SIMD-BP on dataset $D1$. Row 1 reports absolute measurements, while row 2 reports the measurements of each algorithm variant relative to the measurement of the classical 128-bit variant.

the speedups achieved also depend on the bit width, respectively the data characteristics in general.

Our experimental survey in [6] has revealed that especially outliers in the data play a crucial role for the behavior of lightweight data compression algorithms. Thus, we now investigate the interplay of outliers and vector widths. For this purpose, we introduce a new synthetic dataset $D1$. $D1$ is an unsorted dataset consisting of 100 million uncompressed 32-bit integers. Each data element is either a 4-bit value or a 28-bit outlier, whereby we vary the outlier ratio. Figure 2 display the results of the variants of SIMD-BP on this dataset $D1$. Regarding the compression ratio (Figure 2 (a, e)), the three variants differ significantly subject to the outlier ratio. More precisely, as the vector width grows, so does the algorithm's vulnerability to outliers. In the worst cases, SIMD-BP256 and SIMD-BP512 might yield approximately 1.5x and 2x of the compressed data size of SIMD-BP128, respectively. The reason for this is that these ported variants of the algorithm use blocks of larger sizes. Even one outlier per block suffices to force the use of the outlier bit width *for all data elements in the entire block*. Thus, the larger the blocks, the more blocks are affected even by low outlier ratios. Regarding the performance we can make the same observations as for D0. However, with respect to the outlier ratio, we see that the (de)compression speed of SIMD-BP512 is even less than that of SIMD-BP256 for those outlier ratios at which it suffers most regarding compression ratio.

Generally, similar effects are observable for other NS algorithms as well. To summarize, when porting NS algorithms in a straightforward way to SIMD extensions using wider vector registers, the performances can generally be increased. However, the algorithms quickly become memory-bound, resulting in sub-optimal speed ups compared to the classical 128-bit variants. In terms of compression ratio, the algorithms tend to use larger blocks for larger vector widths, which results in less meta data to be stored, but also increases the vulnerability to outliers in the data.

3.2 Preprocessing Techniques

On the preprocessing level, run-length encoding (RLE) is an interesting 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. 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 [6]. In detail, this state-of-the-art RLE comparison-based 128-bit vectorization works as follows:

(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).

We denote this 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.

Figure 3 shows the results for the variants of RLE on dataset $D2$. Dataset $D2$ contains 100 million uncompressed 32-bit integers whose data elements are uniformly drawn from the range $[0, 2^{16}-1]$

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 8, ISBN 978-1-4503-5826-2
https://doi.org/10.1145/3209950.3209957

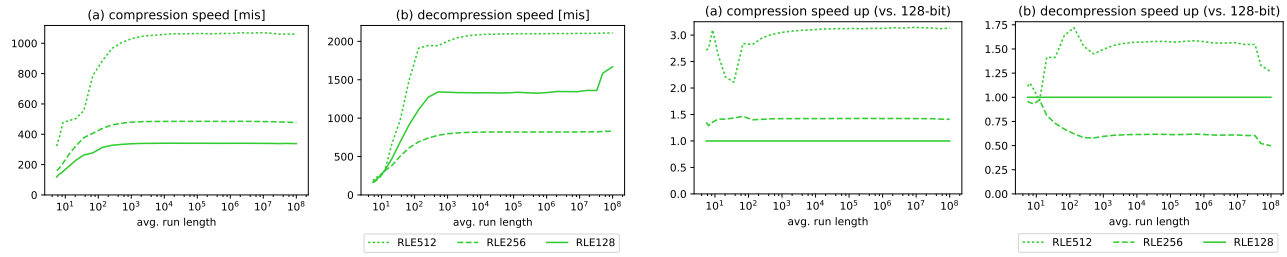

Figure 3: The compression and decompression performance of the three variants of RLE on dataset $D2$. (a-b) report the absolute speeds, while (c-d) report the speeds relative to RLE128.

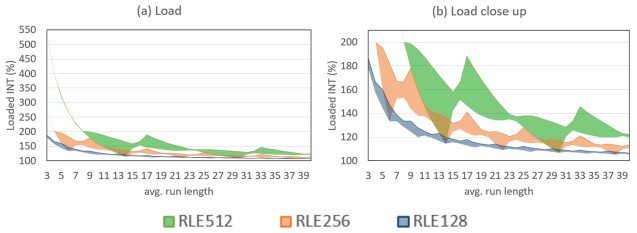

Figure 4: (a) Loaded integers as percentage of the integers in the uncompressed data set. (b) A close up of (a).

while varying the average run length. We omit the compression ratio, since all three variants of RLE always have the same output. Regarding the compression, RLE256 and RLE512 achieve considerable speed ups of about 1.4x and 3x, respectively. Regarding the decompression, however, RLE256 performs significantly worse than RLE128 and yields only about half of the classical variant's speed for long runs. As we can see in Figure 3, the speedup of RLE256 and RLE512 is very marginal for short run lengths. The reason is that some data elements in the input are loaded very frequently for small run lengths. 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. 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. 4(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. 4(b) shows a close up of Fig. 4(a) with the y-axis ranging only until 200%. From these experiments, we can conclude:

(1) The comparison-based RLE vectorization uses a significantly higher number of load operations for sequences with short runs than for sequences with long runs, which negatively effects the performance.
(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.

Similar effects are observable for the other preprocessing techniques. Thus, the speedups are below the optimal goals.

3.3 Cascades of Techniques

We also investigated cascades of logical-level (RLE, FOR, DELTA, and DICT) and pyhsical-level (NS) techniques. Due to space constraints, we only sum up our observations. When SIMD extensions with wider vector registers are employed, the cascades of logical-level and physical-level algorithms still yield compression ratios superior to those of stand-alone NS algorithms, if the data characteristics are suitable. Moreover, the cascades become faster than with SSE (128-bit) in most cases. Finally, we could confirm our initial idea, at least in several cases: when 256-bit or 512-bit SIMD operations are used, the cascades perform better compared to the stand-alone NS algorithm using the same SIMD extension than for 128-bit SIMD operations.

4 LESSONS LEARNED AND NEXT STEPS

As described above, many lightweight compression algorithms can be ported to newer SIMD extensions in a *straightforward* way, but for some algorithms, this is not possible. For instance, 4-WS NS requires auxiliary permutation masks (stored in a lookup table) for compression with a total size of 4 KiB. Additionally, the decompression requires the same amount of space for the masks of the inverse permutations, so 8 KiB are required in total. Note that this amount of data can easily fit into the L1 data cache of modern processors. However, when going to AVX-512, the situation changes *dramatically*. Now a vector register fits 16 uncompressed 32-bit integers, such that the lookup table contains $4^{16} = 4\ Gi$ entries, each of which is a 512-bit vector, resulting in a size of 256 GiB for one table and 512 GiB for both tables. It is self-evident that storing 512 GiB of auxiliary information for a compression algorithm in practically infeasible.

If a straightforward transformation is possible, two effects are observable. First, the larger block sizes of NS algorithms resulting from straightforward re-implementations increase the vulnerability of these algorithms to outliers in the data, which can affect both, the compression ratio as well as the performance negatively. Second, both, logical-level and physical-level algorithms become the faster the wider the employed vector registers are. However, the speed ups are sub-optimal in most cases, since the algorithms quickly become memory-bound when the computations are accelerated through wider vectors registers processing more data elements at once. This has two implications when employing AVX2 and AVX-512: Firstly, accessing uncompressed data in main memory should be avoided even more strictly then with SSE, which we showed with

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 8, ISBN 978-1-4503-5826-2
https://doi.org/10.1145/3209950.3209957

our aggregation of compressed data. Secondly, cascades of logical-level and physical-level algorithms become even more promising alternatives to stand-alone NS algorithms, since they still yield superb compression ratios, but perform much more competitive to stand-alone NS algorithms when implemented with wider vector registers. We have conducted our evaluation using 128, 256, and 512-bit SIMD extensions, which are the ones currently available in general-purpose processors. However, the increase of the vector width is an obvious trend in processor evolution. Thus, we will likely see SIMD extensions with even wider vector registers in the future. We expect the effects we observed in our evaluation be become even more important then. Finally, we would like to highlight that the vector width of the employed SIMD extension has an impact on the relative ranking of the lightweight compression algorithms regarding compression ratio, performance, and any trade-off of these two. Therefore, a strategy for selecting the best lightweight compression algorithm should also be aware of the employed SIMD extension in order to make a wise decision.

Thus, a open challenge is still the development of appropriate approaches which better exploits the capabilities of newer SIMD extensions to a the maximum extent. For example, 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. 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. In [19], we described the application of these *CD* instructions for RLE encoding. In particular, we have clearly shown that the *CD*-based implementation is up to 3.2 times faster for sequences of integers with short run lengths. That means, researchers should still adapt algorithm implementations to new available SIMD instructions.

5 RELATED WORK

The efficient utilization of SIMD (Single Instruction Multiple Data) instructions in database systems is a very active research field [14, 21]. 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 [11]. On the other hand, SIMD instructions are also used in other database operations like scans, aggregations or joins [14, 21]. However, in spite of their great potential, the newer SIMD extensions of AVX2 or AVX-512 have received only little attention in the literature on lightweight data compression. Some papers [16, 20] propose approaches to vectorize lightweight compression algorithms, which essentially treat the vector width as an adjustable parameter. However, non of these has actually discussed wider vectors in detail nor evaluated their proposed algorithms using SIMD extensions beyond 128 bits. While there are papers [9] which employ 256-bit SIMD in their evaluation, to the best of our knowledge, a systematic investigation of 256-bit and 512-bit SIMD extensions for lightweight data compression has never been published.

6 CONCLUSION

Data compression as well as vectorization are important keystones in modern state-of-the-art in-memory column store systems. In particular, the efficient *vectorized* implementation of lightweight compression algorithms has attracted a lot of attention. Most of the developed vectorized implementations have been developed for a fixed vector width of 128 bits In this paper, we investigated the influence of wider vector registers on the behavior of these algorithms and highlighted some challenges.

ACKNOWLEDGMENT

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] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *SIGMOD*.
[2] Vo Ngoc Anh and Alistair Moffat. 2010. Index compression using 64-bit words. *Softw., Pract. Exper.* 40, 2 (2010).
[3] Diego Arroyuelo, Senén González, Mauricio Oyarzún, and Victor Sepulveda. 2013. Document identifier reassignment and run-length-compressed inverted indexes for improved search performance. In *SIGIR*.
[4] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*. 283–296.
[5] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008).
[6] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *EDBT*. 72–83.
[7] Patrick Damme, Dirk Habich, and Wolfgang Lehner. 2015. A Benchmark Framework for Data Compression Techniques. In *TPCTC*.
[8] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *ICDE*.
[9] Abdullah Al Hasib, Juan M. Cebrian, and Lasse Natvig. 2015. V-PFORDelta: Data Compression for Energy Efficient Computation of Time Series. In *HiPC*. 416–425.
[10] Juliana Hildebrandt, Dirk Habich, Patrick Damme, and Wolfgang Lehner. 2016. Compression-Aware In-Memory Query Processing: Vision, System Design and Beyond. In *ADMS-IMDM Workshop at VLDB*. 40–56.
[11] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.* 45, 1 (2015).
[12] D. Lemire, L. Boytsov, O. Kaser, M. Caron, L. Dionne, M. Lemay, E. Kruus, A. Bedini, M. Petri, and Robson Braga Araujo. [n. d.]. The FastPFOR C++ library: Fast integer compression, `https://github.com/lemire/FastPFOR`. ([n. d.]). https://github.com/lemire/FastPFOR
[13] Jeff Plaisance, Nathan Kurz, and Daniel Lemire. 2015. Vectorized VByte Decoding. *CoRR* abs/1503.07387 (2015).
[14] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *SIGMOD*. 1493–1508.
[15] Mark A. Roth and Scott J. Van Horn. 1993. Database Compression. *SIGMOD Rec.* 22, 3 (1993).
[16] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. 2010. Fast integer compression using SIMD instructions. In *DaMoN*.
[17] Fabrizio Silvestri and Rossano Venturini. 2010. VSEncoding: Efficient Coding and Fast Decoding of Integer Lists via Dynamic Programming. In *CIKM*.
[18] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. 2011. SIMD-based Decoding of Posting Lists. In *CIKM*.
[19] Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Dirk Habich, and Wolfgang Lehner. 2018. Conflict Detection-based Run-Length Encoding – AVX512-CD Instruction Set in Action.. In *HardBD and Active Workshop at ICDE 2018*.
[20] Wayne Xin Zhao, Xudong Zhang, Daniel Lemire, Dongdong Shan, Jian-Yun Nie, Hongfei Yan, and Ji-Rong Wen. 2015. A General SIMD-Based Approach to Accelerating Compression Algorithms. *ACM Trans. Inf. Syst.* 33, 3 (2015).
[21] Jingren Zhou and Kenneth A. Ross. 2002. Implementing database operations using SIMD instructions. In *SIGMOD*. 145–156.
[22] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *ICDE*.