BOUNCE: Memory-Efficient SIMD Approach for Lightweight Integer Compression

Juliana Hildebrandt
TU Dresden
Database Research Group
Dresden, Germany
juliana.hildebrandt@tu-dresden.de

Dirk Habich
TU Dresden
Database Research Group
Dresden, Germany
dirk.habich@tu-dresden.de

Wolfgang Lehner
TU Dresden
Database Research Group
Dresden, Germany
wolfgang.lehner@tu-dresden.de

Abstract—**Integer compression plays an important role in columnar database systems to reduce the main memory footprint as well as to speedup query processing. To keep the additional computational effort of (de)compression as low as possible, the powerful *Single Instruction Multiple Data (SIMD)* extensions of modern CPUs are heavily applied. While a scalar compression algorithm usually compresses a block of N consecutive integers, the state-of-the-art SIMD implementation scales the block size to $k*N$ with k as the number of elements which could be simultaneously processed in an SIMD register. On the one hand, this scaling SIMD approach improves the performance of (de)compression but can lead to a degradation of the compression ratio compared to the scalar variant on the other hand. Within this paper, we analyze this degradation effect for an heavily applied and well-performing integer compression algorithm called *BitPacking (BP)* and present a novel SIMD concept to overcome that effect. Our novel SIMD idea called *BOUNCE* is to concurrently compress k different blocks of size N within SIMD registers achieving the same compression rate as the scalar variant. As we are going to show, our proposed SIMD idea works well for *BP* and may offer a new generalized approach to optimize further algorithms.**

Index Terms—**integer compression, SIMD, memory-efficiency**

I. Introduction

Cloud computing has become mainstream and we observe that modern data analysis as well as management continues to move to cloud environment. The biggest challenge for cloud providers of data management solutions is to make efficient use of the underlying hardware to reduce the overall costs. This holds especially for in-memory database systems in the cloud, because main memory is a driving factor for hardware costs [1]. Thus, the optimization of main memory consumption for base data as well as intermediate data during query processing in these systems plays an important role [2], [3]. Generally, this aspect has been an integral part of in-memory database systems from the very beginning. For example, in-memory column-stores encode every base column as a sequence of integer values and the necessary memory space for storing these integer sequences is reduced with the help of some additional lightweight computations for integer compression. Various works have shown that this drastically reduces the main memory footprint [2], [4]. Moreover, these compressed integer values also offer advantages for query processing [2], [4], [5]. However, compression as well as decompression requires additional computational effort.

To keep the computational effort as low as possible, the *Single Instruction Multiple Data (SIMD)* extensions of modern CPUs are heavily applied [6]–[8]. The SIMD objective is to increase the single-thread performance by executing an identical operation on multiple data elements in an SIMD register simultaneously (data parallelism) [9]. The general state-of-the-art SIMD approach for integer compression works as follows: [8]: *While a scalar compression algorithm would compress a block of N consecutive integers, the state-of-the-art SIMD approach scales this block size to $k*N$ with k as the number of integers that can be simultaneously processed with an SIMD register.* This scaling approach increases the performance of (de)compression but can lead to a degradation of the compression ratio compared to the scalar variant.

Our Contribution and Outline. In this extended paper of [10], we analyze this degradation effect and present an alternative SIMD concept called *BOUNCE* to overcome that effect. For that, we mainly focus on a heavily-used and well-performing representative integer compression called *BitPacking (BP)*. The idea behind *BOUNCE* is to concurrently compress k different blocks of size N within SIMD registers to achieve the same compression ratios as the scalar variant in all cases. To present our memory-efficient *BOUNCE* concept, the rest of the paper is structured as follows: In Section II, we briefly summarize the state-of-the-art and we theoretically analyze the degradation effect for *BP*. Then, we present our alternative SIMD concept *BOUNCE* in Section III. We (i) introduce the general idea, (ii) describe the application to *BP* and discuss one specific optimization aspect. Section IV presents representative evaluation results. Finally, we discuss related work in Section V and present a summary in Section VI.

II. Analyzing State-of-the-Art

The objective of lossless, lightweight integer compression algorithms is to represent a sequence of finite integer values with as few bits as possible [6]–[8]. Over the past decades, a large corpus of different algorithms has evolved [6]–[8]. The algorithms have in common that the compressed representation for blocks of integer values often consists of control patterns and data snips [8]. Data snips represent the compressed integers in binary format, while control patterns store the auxiliary information to interpret the data snips.

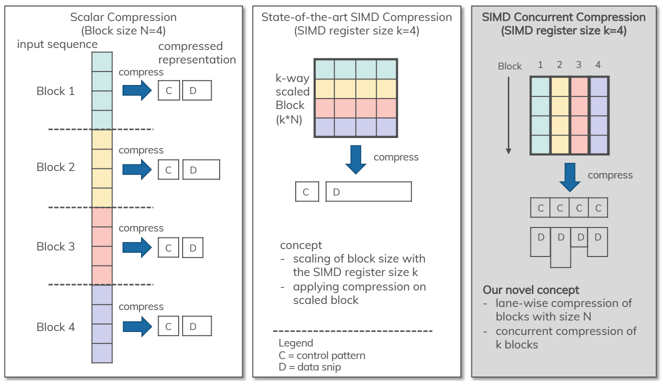

Scalar Compression
(Block size N=4)
input sequence
compressed representation
compress
Block 1
Block 2
Block 3
Block 4
C
D
State-of-the-art SIMD Compression
(SIMD register size k=4)
k-way scaled Block (k*N)
concept
- scaling of block size with the SIMD register size k
- applying compression on scaled block
Legend
C = control pattern
D = data snip
SIMD Concurrent Compression
(SIMD register size k=4)
Block
1 2 3 4
Our novel concept
- lane-wise compression of blocks with size N
- concurrent compression of k blocks

Fig. 1: Overview of compression processing concepts.

A. *Scalar BitPacking*

In this paper, we focus on *BitPacking (BP)* as a heavily used and well-performing integer compression algorithm [7]. *BP* belongs to the class of null suppression algorithms by omitting leading zero bits [7]. This type of compression is, for example, the basis to efficiently execute scans [11], [12]. The scalar version of *BP* for 64-bit integer values is called *BP64* and works as follows: The input sequence of integer values is subdivided into blocks of 64 integers each. For each block, the minimal number of bits required for the largest element is determined. Then, all 64 integers in each block are stored in a data snip with the respective number of bits for each value. The used bit width is stored in a single 64-bit integer as control pattern. Other scalar compression algorithms operate in a similar way and Fig. 1 gives a schematic overview of this procedure with a block size of four.

B. *State-of-the-art SIMD Approach*

The state-of-the-art SIMD approach for integer compression is characterized by the fact, that (i) the block size is scaled by the SIMD register size k – k is the SIMD register size in number of integer values – and (ii) the application of the compression on this larger block sizes as depicted in Fig. 1 [8]. For example, *SIMD-BP256* is the SIMD implementation of *BP64* for SIMD register sizes of 256-bits. Here, we are able to store and process 4 64-bit integer values at once, so that the block size is scaled by $k = 4$. Based on that, the k-way scaled SIMD block contains 256 integer values and for these elements, the minimal number of bits required for the largest element is determined. Then, all 256 integers in each block are stored in a data snip with that many bits for each value and the used bit width is stored as common control pattern. The compressed values in the data snips are organized using a k-way vertical layout distributing N consecutive integers to k different groups [7]. *SIMD-BP256* offers superior performance compared to other compression algorithms [7]

C. *Analyzing Memory Footprint*

A main drawback of the k-way scaling is that the compression factor

$$cf(k) = \frac{|k\text{-way compressed data}|}{|\text{uncompressed data}|} \tag{1}$$

mostly increases with an increasing SIMD register and block size. On the one hand, fewer control patterns need to be stored due to larger block sizes. On the other hand, a number of larger integer values may be compressed with a larger bit width. To precisely analyze this effect, we derive the expected compression factor for different integer bit width distributions.

Our analysis framework works as follows: The scalar algorithm *BP64* encodes blocks of 64 64-bit values with the least possible common bit width and the bit width as control pattern itself with 64 bits. The SIMD-based implementations with SIMD register size k encode $64 \cdot k$ values with the same approach. Given is a data distribution for 64-bit integer values characterized by the probability for the bit widths $0 \leq b \leq 64 : p(b)$ and an SIMD register size k. Now, we can distinguish 65 cases corresponding to blocks of $64 \cdot k$ values that are encoded with bit width $0 \leq b \leq 64$. Each of these cases (i) occurs with a probability $p'(b, k)$, which depends on the given data distribution and the SIMD register size k, and (ii) is characterized by a block compression factor $cf'(b, k)$. The expected compression factor for a k-way SIMD-based implementation of *BP* can be calculated by

$$cf(k) = \sum_{b=0}^{64} p'(b, k) \cdot cf'(b, k). \tag{2}$$

The block compression factor $cf'(b, k)$ is given by

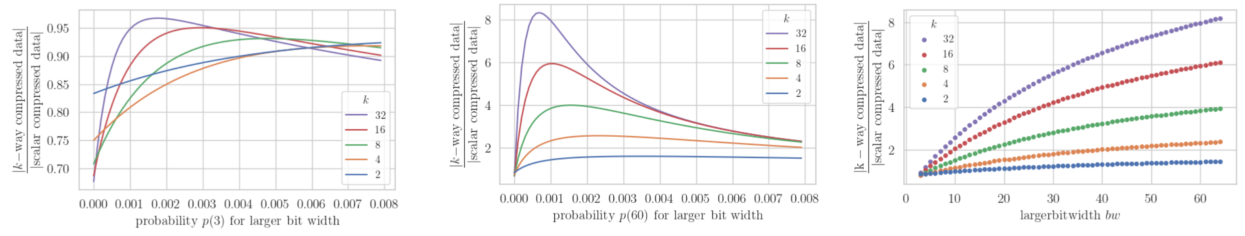
$$cf'(b, k) = \frac{|k\text{-way compressed block}|}{|\text{uncompressed block}|} = \frac{1 + b \cdot k}{64 \cdot k} \tag{3}$$

and the block probability can be derived by the following consideration. The probability of the occurrence of a block of size $64 \cdot k$ containing only zero values is $p'(0, k) = p(0)^{64 \cdot k}$. The probability of a block encoded with one of the bit widths $0 \leq b$ is $\left(\sum_{bw=0}^{b} p(bw)\right)^{64 \cdot k}$. The probability for the occurrence of a block encoded with bit width b is the difference of the above probability and all probabilities for the occurrences of blocks with a smaller bit width than b:

$$p'(b, k) = \left(\sum_{bw=0}^{b} p(bw)\right)^{64 \cdot k} - \sum_{bw=0}^{b-1} p'(bw, k). \tag{4}$$

For the compressed size ratio between a k-way SIMD implementation and the scalar implementation of *BP*, we calculate $\frac{cf(k)}{cf(1)}$ with $cf(1)$ corresponding to the scalar compression factor (scaling factor $k = 1$).

In the following, we apply these formulas on two different data distributions where most integer values are characterized by a bit width of 2, but we also have a probability x for integer values with a larger bit width. While in the first case the larger bit width is 3, the bit width in the second case is 60. Fig. 2a and Fig. 2b depict the compressed size ratio $\frac{cf(k)}{cf(1)}, k = \{2, 4, 8, 16, 32\}$ for the SIMD implementations for both cases and different probabilities. As we can observe in Fig. 2a, all lines are below 1 for case one with a larger bit width of 3. That means, each SIMD implementation (using different k-way scalings) has a lower compression factor than



(a) Bit widths 2 and 3 with varying probability $p(3)$.

(b) Bit widths 2 and 60 with varying probability $p(60)$.

(c) Varying larger bit widths bw and $p(bw) = 0.001$.

Fig. 2: Theoretical analysis results comparing scalar and state-of-the-art SIMD implementations for *BP*.

the scalar algorithm. Thus, the memory footprint is further optimized compared to the scalar variant. The reason is the lower number of control patterns for larger blocks and the more or less homogeneous bit widths for all integer values. Moreover, the value k for the best SIMD implementation yielding the best compression ratio depends on the probability of the larger bit width.

In contrast to that, for the second case with the larger bit width of 60 and lower outlier probabilities, we see that all lines are above 1 (cf. Fig. 2b). This means, the compression ratio of the scalar variant is much better than of the SIMD implementations. For example, a compressed representation of 8-way SIMD implementation – SIMD register size 512-bit with 64-bit integer values – is 4 times larger than for the scalar variant. Those disturbing effects happen and destroy the advantages of the SIMD-based integer compression, especially since a small number of outliers has such large effects.

Finally, we examined a variety of larger bit widths with a fixed probability of $p = 0.001$ and different values of k. Again, most integers are characterized by a bit width of 2. As we can see in Fig. 2c, the memory footprint of the SIMD compression is significantly worse than for the scalar variant in most of the situations. And the factor grows with increasing larger bit widths and vector sizes k.

III. Memory-Efficient SIMD Concept

Our analysis has shown that the state-of-the-art SIMD approach has shortcomings from a memory footprint point of view. To overcome that, we propose an alternative SIMD concept called *BOUNCE*, explain the application to *BP*, and discuss a specific optimization in this section.

A. Block Concurrent SIMD Concept

Instead of scaling the block size by the SIMD register size k, we propose a block concurrent compression concept *BOUNCE* as alternative SIMD approach as depicted in Figure 1. In *BOUNCE*, each SIMD register place – also called SIMD lane – compresses its own data block. Thus, the number of available SIMD lanes determines the number of blocks that will be compressed simultaneously. The advantages are (i) the same block size of the scalar compression algorithms is maintained and (ii) the control patterns as well as data snips are calculated lane-wise. That means, we apply the scalar compression algorithm on each SIMD lane on different data

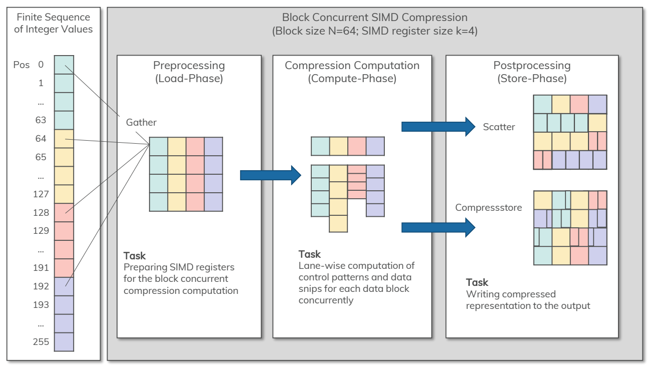

Finite Sequence of Integer Values
Pos 0
1
...
63
64
65
...
127
128
129
...
191
192
193
...
255
Block Concurrent SIMD Compression (Block size N=64; SIMD register size k=4)
Preprocessing (Load-Phase)
Gather
Task
Preparing SIMD registers for the block concurrent compression computation
Compression Computation (Compute-Phase)
Task
Lane-wise computation of control patterns and data snips for each data block concurrently
Postprocessing (Store-Phase)
Scatter
Compressstore
Task
Writing compressed representation to the output

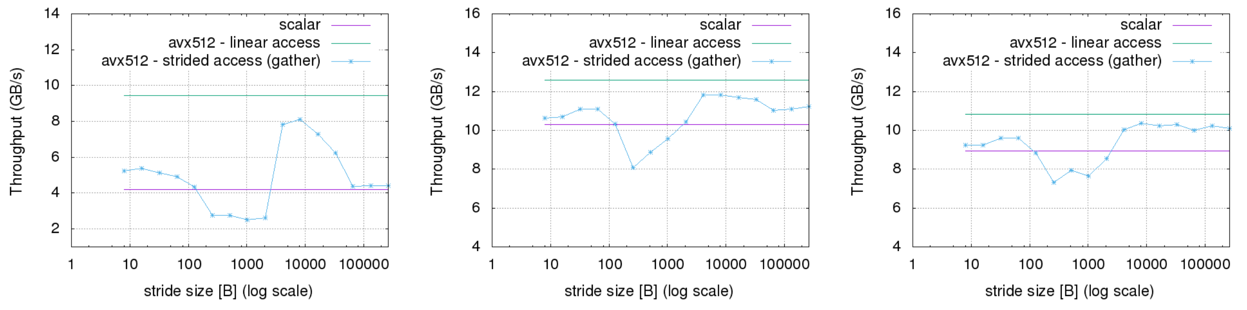
Fig. 3: Block concurrent BP compression.

blocks concurrently. Thus, we are able to guarantee the same compression ratios as the scalar variant in all cases.

B. Application to BP

Compression. The *BP* compression with *BOUNCE* is illustrated in Fig. 3. Here, we assume integer values of size 64-bit, which will be compressed with the scalar variant *BP64*. The assumed SIMD register size k is 8, so that eight different data blocks of 64 values are compressed simultaneously. That means, we are processing 512 integer values in total of the input sequence of integer values and compute eight control patterns and data snips as compressed output. The *BP* compression is done in two phases, thereby each phase iterates over all 512 integer values. In the first phase, the bit width of the largest integer value within each block is determined, while the second phase uses the determined bit widths to shorten the values accordingly and to write out the compressed output.

As shown in Fig. 3, we distinguish (i) a preprocessing step to load the data from the input into the vector registers, (ii) a computation step to shorten the values, and (iii) a postprocessing step to write the data into the output area. These steps are executed in each phase and each phase executes 64 iterations for 64 values per block. That means, for the nth iteration, we require the integer value of the nth position of each considered data block in the SIMD register. Assuming that the input sequence contains correct ordered data (*horizontal data layout*), we simply could use an `SIMD-gather` instruction to load the corresponding values of the different data blocks into the SIMD register. The `SIMD-gather` seems to be expensive, because it can be used for random memory access.



(a) Xeon Phi 7250 (b) Xeon Gold 6126 (Skylake) (c) Xeon Gold 6240R (Cascade Lake)

Fig. 4: Evaluating best-performing stride distance for AVX-512 (64-bit) on different Intel hardware platforms.

However, in our case, we realize a block-strided access pattern with a distance of 64.

In the computation step, we apply the appropriate SIMD functions for each phase. In the first phase, we apply the SIMD functions to compute the number of leading zeros for the largest value per lane (per block). Based on the number of leading zeros, we compute the minimal number of bits for the compression. This bit widths are used in the second phase to concatenate the shortened data values while using an appropriate SIMD-bitshifting instruction, which can be applied for each lane individually. Because in the single lanes, the data is concatenated with a different bit width, the lanes are filled at different loop passes. For example, a lane is full after 2 iterations for a bit width of 30 (assuming 64 bit integer values), but for a bit width of 2, we need 32 iterations to fill a lane. In any case, if one of the lanes is full, it has to be written to the output (postprocessing step). Here, we see two alternatives. The first alternative is to use an `SIMD-compressstore` instruction to consecutively write out lanes as soon as they are full. In this case, data snips of the different blocks are intertwined. The second alternative is to use an `SIMD-scatter` instruction. Since the bit width for each block is determined at first, the bit widths can also be used to calculate the position for each full lane in the output. In this case, we are able to organize the data snips for each of them in a consecutive manner.

Decompression. The decompression routine of *BOUNCE* can be built nearly straightforward the other way around. The preprocessing starts with loading of the control patterns by the application of an `SIMD-load` intruction and they are required for the correct decompression. For the preprocessing of the compressed values, the instructions complementary to `SIMD-gather` and `SIMD-compressstore` – namely `SIMD-scatter` and `SIMD-expand` – are applied to load the compressed data snips into the vector registers. During the decompression computation step, the compressed data snips are expanded to 64-Bit integers by prepending leading zeros. For the postprocessing, an `SIMD-scatter` instruction complementary to the `SIMD-gather` in the preprocessing phase of the compression is applied. For simplicity, the `gather/scatter` alternative might be preferred. The `compressstore/expand` requires a calculation intensive mapping from consecutive compressed data to the different lanes. This can be avoided by decompressing the data from the last to the first value – at any point in time one or several lanes are discharged, the next one or several 64-bit words holding compressed data are accessed via `SIMD-expand` and loaded in the discharged lanes.

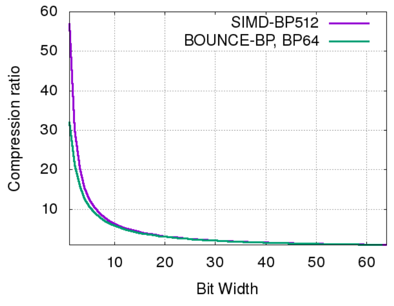
C. Optimization

A drawback of our *BOUNCE* concept may be the utilization of expensive SIMD instructions like `gather`, `scatter`, or `compressstore`. Thus, the performance will probably be worse compared to the state-of-the-art SIMD approach. To overcome that, there are enough optimization knobs, hence we haven taken a closer look at one knob as an example.

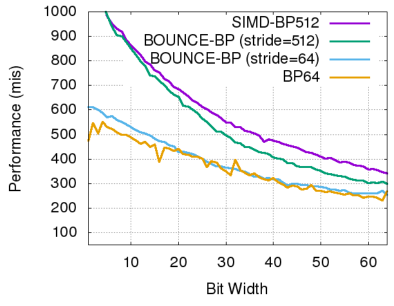
For the *BOUNCE-BP* compression, the integer values from different memory regions must be loaded twice into vector registers using `gather` instructions. In contrast to that, the state-of-the-art SIMD approach also loads values twice but always consecutively by means of a `load` instruction. However, a special feature of our approach is that we realize a block-strided access pattern with the `gather`, whereby the stride distance is 64. That means – for vector length of $k = 4$ –, the first vector register is filled with integer values at positions 0, 64, 128 and 192. Then, the integers at positions 1, 65, 129, and 193 are loaded and so on. The stride distance does not necessarily have to be 64, but must be a multiple of 64 and thus represents an optimization knobs.

In these micro-benchmarks, we executed a sum-aggregation (mainly to focus on reading with little computation and one write operation with the sum at the end) over a 4 GiB input array of randomly generated 64-bit unsigned integer values. We implemented this sum-aggregation in C++ as a scalar variant, as a SIMD variant using the `load` instruction (linear access pattern), and as a SIMD variant using the `gather` instruction with a variable stride distance. We compiled the variants using `g++` (version 9.3.0) with the optimization flags `-O3 -fno-tree-vectorize -mavx512f -mavx512cd`.

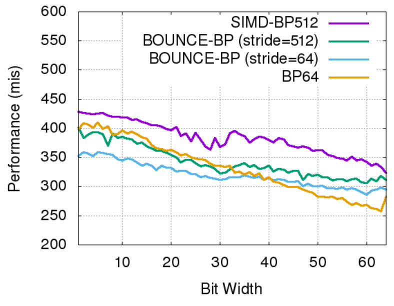
We executed these micro-benchmarks using AVX-512 on different Intel platforms as depicted in Table I and the results are shown in Fig. 4. All micro-benchmarks are executed single-threaded, happened entirely in-memory, were repeated 10 times, and we averaged the results. In the diagrams, the stride distance or stride size in bytes is plotted in log scale on



(a) Compression ratio



(b) Compression speed



(c) Decompression speed

Fig. 5: Experimental results for data sets with fixed bit widths.

Processor Type	L1 Cache	L2 Cache	L3 Cache	Main Memory
Xeon Phi 7250	32KiB	1MiB	-	204GB
Xeon Gold 6126	32KiB	1MiB	19MiB	92GB
Xeon Gold 6240R	32KiB	1MiB	35.75MiB	384GB

TABLE I: Hardware platforms with cache information.

the x-axis, while the y-axis shows the throughput in GB/s. As we can see, the curves are similar on the different platforms. The scalar variant achieves always the slowest throughput, while the SIMD with the linear access pattern always achieves the best throughput (expected behavior). The `gather`-variant is in one stride distance range much worse than the scalar variant, but for certain stride distances it comes very close to the SIMD-variant with the linear access pattern. Especially, a stride distance of 64 values (512 Bytes) is in the very unfavorable range, where the achievable throughput is much lower compared to the scalar variant. However, very good throughput values are achieved for the stride distances of 512 values (4096 Bytes). These well-performing stride distances match the page size of 4KB, so that all SIMD lanes in *BOUNCE* load integer values from different pages that are cached after the first access and thus can be accessed optimally afterwards. We conclude, we should use a stride distance of 512 instead of 64 for *BOUNCE*. In this case, each SIMD lane operates on its own page containing 512 values and then 64 values — one after the other — are compressed with BP64.

IV. Evaluation

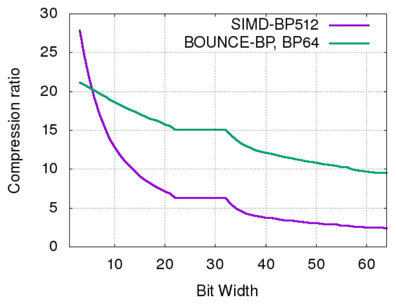
To evaluate whether and when our *BOUNCE* concept for *BP* is suitable, we implemented *BP* for 64-bit integer values in its scalar form (denoted as *BP64*), and compared it with the state-of-the-art SIMD approach (called SIMD-BP512)[1] and with our novel *BOUNCE* concept (denoted as *BOUNCE-BP*) using Intel's latest SIMD extension AVX-512. For all variants, we implemented the compression as well as decompression routines. For *BOUNCE* ($k = 8$), we implemented all possible variants as described in Section III, but the evaluation in this paper focuses only on the variant using `SIMD-gather` and `SIMD-scatter` instructions. The state-of-the-art SIMD implementation uses `SIMD-load` and `SIMD-store` operations with a block scaling factor of $k = 8$. We used the same compiler and optimization flags as described in the previous section. We ran this evaluation on an Intel Xeon Gold 6240R (Cascade Lake architecture) with 768 GB main memory capacity (cf. Table I). All experiments are executed single-threaded, happened entirely in-memory, were repeated 10 times, and we averaged the results. In all cases, we report the compression ratio[2] and the performance in *million integers per second (mis)* for the (de)compression, so that higher values are always better.

Data sets with fixed bit widths. In the first set of experiments, we created different synthetic data sets with randomly generated unsigned integer values. Each data set only contains values of a fixed bit width. Then, we applied all compression as well as decompression routines and the results are shown in Fig. 5. As we can see in Fig. 5a, these data sets are perfectly suited for the state-of-the-art SIMD approach, because they achieve higher compression ratios and higher performance for compression as well as decompression. However, the *BOUNCE* BP compression implementation (cf. Fig 5b) with a stride distance of 512 closely matches the performance of the state-of-the SIMD approach, while a stride distance of 64 has similar performance to the scalar variant. This clearly shows that the optimization of the stride distance is very important. Decompression speeds as illustrated in Fig. 5c are all in the same range, with the *BOUNCE* implementation being slowest for small bit widths. However, the *BOUNCE* decompression routines are not optimized yet.

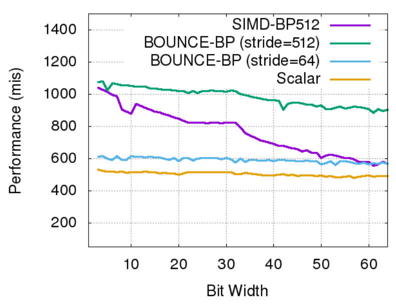
Data sets with different bit widths. In the second set of experiments, we created synthetic different data sets similar to the setting in Section II. That means, the integer values are mainly characterized by a bit width of 2, but we have a probability of $p(bw) = 0.001$ for integer values with a larger bit width bw. We varied this larger bit width bw from 3 to 64 for the data sets and the results are shown in Fig. 6. As we can see in Fig. 6a, the *BOUNCE-BP* compression achieves much higher compression ratios resulting in a smaller compressed output. Since we have to write out less, the performance for compression also improves as depicted in Fig. 6b. In this case, *BOUNCE-BP* with a stride distance of 512 clearly outperforms

[1]Both implementations for 64-bit integers are inspired by existing implementations of Daniel Lemire for 32-bit integers published on Github: https://github.com/lemire/LittleIntPacker/blob/master/src/bitpacking32.c, https://github.com/lemire/simdcomp/blob/master/src/avx512bitpacking.c

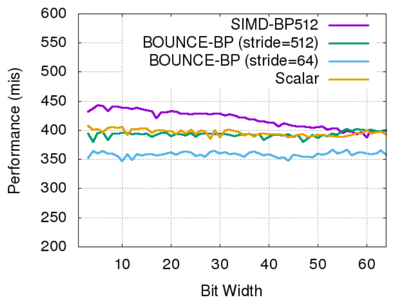
[2]compression ratio is computed by $\frac{|\text{uncompressed}|}{|\text{compressed}|}$



(a) Compression ratio



(b) Compression speed



(c) Decompression speed

Fig. 6: Experimental results for data sets with a mix of two different bit widths.

the state-of-the-art SIMD implementation and the speedup increases with increasing bit widths. Moreover, *BOUNCE-BP* with a stride distance of 64 is slightly better than the scalar variant which shows again the importance of the stride distance optimization. The decompression performance is similar to the previous set of experiments as the decompression is not optimized yet. To summarize, our evaluation results are promising and encourage further research towards the generalization of *BOUNCE* for further compression algorithms.

V. Related Work

A comprehensive overview of the field of lossless lightweight integer compression algorithms and SIMD implementations is given by the following papers [6]–[8], [13]. In addition to that, we presented a meta-model to specify integer compression algorithms in a descriptive and abstract way with the ability to derive executable code from that description [14]. An integration of our presented generalized SIMD approach into the transformation to the executable code is in the focus of our ongoing research activities. Moreover, the selection of the best-fitting integer compression variant is a research field with a very dynamic development [5], [6]. With our alternative generalized SIMD approach *BOUNCE*, we extend the variety of variants increasing the importance of the selection. From a SIMD execution point of view, our presented *BOUNCE* concept is in line with the idea of sharing vector registers for concurrently running queries as described in [15]. Nevertheless, the application as well as the specific challenges differ. However, both approaches show that an alternative use of SIMD execution can be profitably employed.

VI. Conclusion

Integer compression plays an important role to reduce the memory footprint and to speedup query processing in column-stores. While a scalar compression algorithm usually compresses a block of N consecutive integers, the state-of-the-art SIMD implementation usually scales the block size to $k \cdot N$ with k as the number of elements that could be simultaneously processed in a SIMD register. However, this means that as the SIMD register size increases, the block of integer values for compression also grows, which can have a negative effect on the compression ratio. In this paper, we analyzed this effect and showed that the compressed output could be many times larger than the result of a scalar implementation. To overcome that, we presented an alternative SIMD concept called *BOUNCE* which concurrently compress k different blocks of size N within SIMD registers of size k. Due to the promising results for the heavily used integer compression representative *BitPacking*, we want to intensify our work in this area. In particular, we will further optimize our concept in combination with investigating different integer compression algorithms. In general, *BOUNCE* can lead to more responsible usage of main memory resources which is necessary for cloud environments.

References

[1] D. Lomet, "Cost/performance in modern data stores: How data caching systems succeed," in *DaMoN*, 2018, pp. 1–10.

[2] D. J. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD*, 2006, pp. 671–682.

[3] L. Landgraf, F. Wolf, A. Boehm, and W. Lehner, "Memory efficient scheduling of query pipeline execution," in *CIDR*, 2022.

[4] P. Damme, A. Ungethüm, J. Pietrzyk, A. Krause, D. Habich, and W. Lehner, "Morphstore: Analytical query engine with a holistic compression-enabled processing model," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2396–2410, 2020.

[5] M. Boissier and M. Jendruk, "Workload-driven and robust selection of compression schemes for column stores," in *EDBT*, 2019, pp. 674–677.

[6] P. Damme, A. Ungethüm, J. Hildebrandt, D. Habich, and W. Lehner, "From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms," *ACM Trans. Database Syst.*, vol. 44, no. 3, pp. 9:1–9:46, 2019.

[7] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Softw. Pract. Exp.*, vol. 45, no. 1, pp. 1–29, 2015.

[8] W. X. Zhao *et al.*, "A general simd-based approach to accelerating compression algorithms," *ACM Trans. Inf. Syst.*, vol. 33, no. 3, pp. 15:1–15:28, 2015.

[9] C. J. Hughes, *Single-Instruction Multiple-Data Execution*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2015.

[10] J. Hildebrandt, D. Habich, and W. Lehner, "Towards a general simd concurrent approach to accelerating integer compression algorithms," in *EDBT Short Paper*, 2022.

[11] Y. Li and J. M. Patel, "Bitweaving: fast scans for main memory data processing," in *SIGMOD*, 2013, pp. 289–300.

[12] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner, "Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units," *PVLDB*, vol. 2, no. 1, pp. 385–394, 2009.

[13] 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.

[14] J. Hildebrandt, D. Habich, P. Damme, and W. Lehner, "Model kit for lightweight data compression algorithms," in *EDBT*, 2016, pp. 692–693.

[15] J. Pietrzyk, D. Habich, and W. Lehner, "To share or not to share vector registers?" in *DaMoN*, 2020, pp. 12:1–12:10.