Mikhail Zarubin, Patrick Damme, Thomas, Kissinger, Dirk Habich, Wolfgang Lehner, Thomas Willhalm

Integer Compression in NVRAM-centric Data Stores – Comparative Experimental Analysis to DRAM

Erstveröffentlichung in / First published in:

SIGMOD/PODS '19: International Conference on Management of Data, Amsterdam 01.07.2019. ACM Digital Library, Art. Nr. 11. ISBN 978-1-4503-6801-8.

DOI: https://doi.org/10.1145/3329785.3329923

Integer Compression in NVRAM-centric Data Stores — Comparative Experimental Analysis to DRAM

Mikhail Zarubin, Patrick Damme, Thomas Kissinger, Dirk Habich, Wolfgang Lehner
Database Systems Group
Technische Universität Dresden
first.last@tu-dresden.de

Thomas Willhalm
Intel Deutschland GmbH, Germany
[illegible]

ABSTRACT

Lightweight integer compression algorithms play an important role in in-memory database systems to tackle the growing gap between processor speed and main memory bandwidth. Thus, there is a large number of algorithms to choose from, while different algorithms are tailored to different data characteristics. As we show in this paper, with the availability of byte-addressable non-volatile random-access memory (NVRAM), a novel type of main memory with specific characteristics increases the overall complexity in this domain. In particular, we provide a detailed evaluation of state-of-the-art lightweight integer compression schemes and database operations on NVRAM and compare it with DRAM. Furthermore, we reason about possible deployments of middle- and heavyweight approaches for better adaptation to NVRAM characteristics. Finally, we investigate a combined approach where both volatile and non-volatile memories are used in a cooperative fashion that is likely to be the case for hybrid and NVRAM-centric database systems.

CCS CONCEPTS

• **Information systems** → **Data compression**; **Main memory engines**; **Phase change memory**.

KEYWORDS

in-memory; data compression; nvram; analysis; database systems

ACM Reference Format:
Mikhail Zarubin, Patrick Damme, Thomas Kissinger, Dirk Habich, Thomas Willhalm, Wolfgang Lehner. 2019. Integer Compression in NVRAM-centric Data Stores — Comparative Experimental Analysis to DRAM. In *International Workshop on Data Management on New Hardware (DaMoN'19), July 1, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3329785.3329923

1 INTRODUCTION

Data compression is a well-known optimization technique in database systems [1, 15, 17, 18, 34]. On the one hand, data compression has been extensively used to optimize the disk access bottleneck in disk-centric database systems [15, 34] using classical or so-called heavyweight generic data compression schemes such as Lempel-Ziv [44], Huffman [19] or arithmetic coding [39]. On the other hand, lightweight integer compression algorithms are heavily used to optimize the in-memory processing in DRAM-located column store systems [1, 7, 17, 18, 24]. The difference between heavy- and lightweight compression algorithms is their computational complexity.

DOI: https://doi.org/10.1145/3329785.3329923

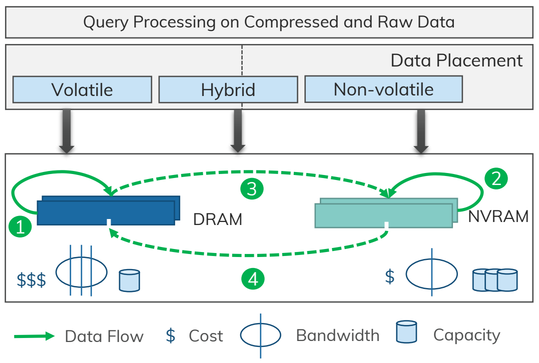


Figure 1: Possible data flows in hybrid main memories: (1)-(2) *intra-memory* and (3)-(4) *inter-memory* data flows.

With the long-awaited availability of *non-volatile random-access memory (NVRAM)* – i.e., Intel Optane DC Persistent Memory [20] – a novel player between disk and main memory has finally turned up. Due to the mixture of DRAM-like characteristics (e.g., low latencies, direct load/store access semantics, higher bandwidth compared to flash memory) and non-volatility in a single device, significant changes on the architectural level of database systems have been proposed. NVRAM is primarily expected to complement or replace block-based secondary storage (e.g., HDDs or SSDs) for storing the *primary data* [3, 4, 23, 29, 32] which can be at the same time the *working copy*. Despite the fact that persistent memory can potentially provide higher (up to 4*x* of DRAM) capacities [20], data compression is still reasonable due to the following reasons: (1) the amount of data to be stored still can be large even for NVRAM-offered volumes, (2) the NVRAM bandwidth is significantly lower than that of DRAM and data compression is a way to increase the effective throughput.

Contributions and Outline

Our main focus in this paper is on integer compression in NVRAM-centric and hybrid data stores, which has not been investigated before. These storage architectures implicitly assume that for certain

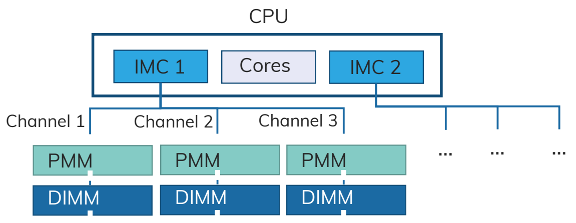


Figure 2: Schematic view of a single socket of our platform.

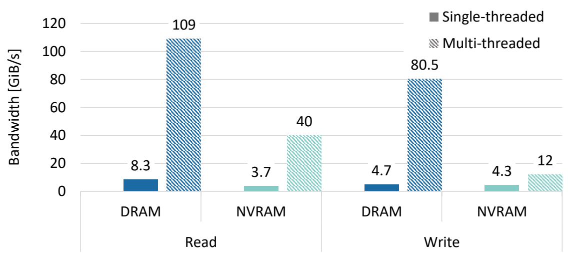


Figure 3: Read and write bandwidths for a sequential access pattern on DRAM and NVRAM.

scenarios data can be located and processed not only in NVRAM but also in DRAM (before actually being placed in persistent memory) and vice versa since these mediums are only distinguished on virtual addressing level and both are accessed via load/store instructions. This allows intra-memory and inter-memory data flows as schematically shown in Figure 1. The following four combinations are possible: (1) both source and destination buffers are accommodated by DRAM, (2) both source and destination buffers are placed in NVRAM, (3) source is located in DRAM, while destination in NVRAM (one possible use case is flushing the primary data or delta to storage in compressed form), (4) source is located in NVRAM, while destination in DRAM (one use case is compressing the bulk NVRAM data to fit in the DRAM volumes). While the flow (1) is traditional and can be projected on persistent memory as flow (2), the flows (3)-(4) are only enabled for the hybrid DRAM-NVRAM data stores. Using these data flows, we will analyze and discuss the behavior of different data compression algorithms in this paper. In detail, we make the following contributions:

(1) We provide a thorough evaluation of state-of-the-art lightweight compression schemes on NVRAM with respect to their single- and multi-threaded performance and compare it with DRAM.
(2) We reason about possible usages of middle- and heavyweight approaches. Particularly, we deploy cascades of lightweight strategies and one traditional heavyweight algorithm (a variation of Lempel-Ziv [44]) to better utilize NVRAM characteristics.
(3) We consider the approach specific for hybrid DRAM-NVRAM data stores where both volatile and persistent memories are used in a cooperative way to support inter-memory data operations and show that we can bridge the performance gap between DRAM- and NVRAM-only setups. Moreover, we demonstrate that moving data inter-memory can be accelerated by (de)compressing it on-the-fly.
(4) We investigate the impact of the SIMD extension employed by the lightweight compression algorithms, depending on the kind of memory.
(5) We show that aggregations directly on the compressed data are even more beneficial on NVRAM than on DRAM, while the selection scan speed is weakly correlated with the memory performance.

The remainder of the paper is structured as follows: Section 2 provides a general comparison of related NVRAM and DRAM metrics. Subsequently, Section 3 introduces our investigated data compression algorithms in more detail, followed by Section 4 providing an in-depth analysis of their single- and multi-threaded behavior on DRAM, NVRAM and hybrid cases. Afterwards, we consider the performance of important database operations on compressed data in Section 5. Finally, Section 6 briefly summarizes related work and Section 7 concludes the paper.

2 HARDWARE PLATFORM

In this paper, we use a novel dual-socket system equipped with (1) a second generation Intel Xeon Scalable processor (codenamed Cascade Lake) clocked at 2600 MHz and (2) a hybrid main memory consisting of 384 GiB DDR4 DRAM memory and 1.5 TiB Intel Optane DC Persistent Memory as a foundation. This is pre-release hardware and performance measurements may differ from the final product. Each processor has 24 physical cores (48 with Hyper-Threading). The persistent memory modules (PMMs) are plugged into the system in a shared fashion meaning that each of the three channels of an integrated memory controller (IMC) is attached to both DRAM and NVRAM memory modules as shown in Figure 2. We run a Fedora 27 with kernel version 4.15.

To better understand this hardware with respect to integer compression, we now provide a general comparison of DRAM and NVRAM. In particular, we focus on sequential read and write bandwidth, since for most of the data compression algorithms, the input data is loaded sequentially, processed and then stored sequentially as well. For the purpose of comparison, we use synthetic, but appropriate workloads generated by our internal benchmark tool.

Read Bandwidth. Figure 3 shows the maximum read bandwidth achieved by single and multiple threads on local sockets using a sequential memory access pattern for both DRAM and NVRAM. We observe that the sequential read access DRAM bandwidth is 2.25*x* and 2.75*x* higher compared to the NVRAM bandwidth for the single-threaded and multi-threaded scenario respectively.

Write Bandwidth. Figure 3 also depicts the write bandwidth for a sequential access pattern for DRAM and NVRAM measurements. Similar to the read workload, the peak bandwidth for both memory types is reached using a multi-threaded access pattern. While the DRAM write throughput is about 74 % of the peak read throughput, this asymmetry amounts to a 3.4*x* lower write bandwidth on NVRAM compared to the read measurements. With regard to single-threaded performance, we measured only a few percent difference between DRAM and NVRAM. Surprisingly, the single-threaded NVRAM write bandwidth outperforms its read counterpart by approximately 15 %.

Final edited form was published in "SIGMOD/PODS '19: International Conference on Management of Data. Amsterdam 2019", Art. Nr. 11, ISBN 978-1-4503-6801-8
https://doi.org/10.1145/3329785.3329923

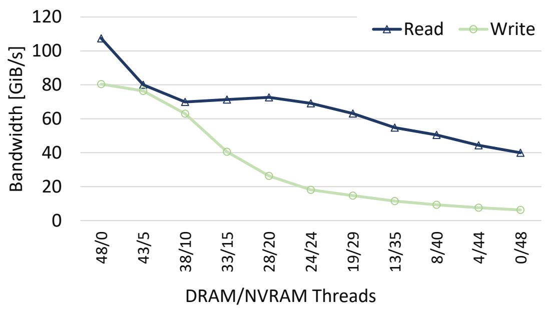


Figure 4: Shared DRAM and NVRAM read and write bandwidth for a sequential access pattern.

Concurrent DRAM-NVRAM Access. To figure out whether the two memory types can be used simultaneously with the same maximum performance, we performed the measurements reflected for the 48-threaded case by Figure 4. This experiment revealed that concurrent accesses to both memories are not able to exceed the maximum reachable bandwidth (independently of the NVRAM thread count) of stand-alone DRAM workloads and actually hurt the performance of both involved components compared to separate results. We expected this since both memory types in our platform share the same memory controllers and channels.

Conclusions. From the read and write bandwidth experiments we can conclude that NVRAM is $2.75x$ (read) to $7x$ (write) slower compared to DRAM and exhibits a high read-write asymmetry of up to $3.4x$. Interestingly, single-threaded measurements show only a small difference in the case of writes. From the concurrent DRAM-NVRAM access experiments we infer that intensive simultaneous usage of both memories is not favoured by the shared IMCs.

3 INTEGER COMPRESSION

As already mentioned, there is a large variety of data compression algorithms available ranging from lightweight to heavyweight algorithms. Based on the different read and write bandwidths of DRAM and NVRAM, we decided to investigate the whole spectrum of algorithms in our experimental study. Moreover, we focus our study on the (de)compression of integer data since most in-memory column store systems encode values of each column as a sequence of integers using some type of dictionary encoding [6].

3.1 Investigated Algorithms

To cover the whole spectrum, we focus on three different categories: light-, middle-, and heavyweight algorithms.

Lightweight Algorithms. This category of algorithms is usually used to optimize query processing of in-memory column stores. These algorithms are not only optimized for compression rate, but also for performance and processing capabilities [11]. We already conducted an exhaustive experimental survey for this category on DRAM and we showed that the performance and compression rate of these algorithms vary greatly depending on the underlying data properties [11, 13]. Based on that, we decided to consider two representatives of the state-of-the-art of Null Suppression algorithms which aim at representing each integer value using a minimal number of bits.

SIMD-BP128 [25] subdivides the data into blocks of 128 32-bit integers each and encodes all data elements in a block using the number of bits required for the block's *largest* data element. Thus, this algorithm profits from small integers. In fact, among all algorithms we consider, SIMD-BP128 is the one investing the least effort into compressing the data. In [11, 13], we have shown that SIMD-BP128 is a very good choice regarding both compression rate and performance if the data contains no or only few outliers. However, if the data contains many outliers, these dominate the block bit widths and, thus, lead to a degradation of the compression rate [11, 13]. In contrast, **SIMD-FastPFOR** [25] is able to adapt to outliers by choosing a bit width suitable for *most* data elements in a block and storing the remaining *exceptions* separately. While this special treatment improves the compression rate in the presence of outliers, it can also yield a degradation of the performance due to the extra effort spent on the outliers [11, 13].

Middleweight Algorithms. SIMD-BP128 and SIMD-FastPFOR only achieve good compression rates if the data consists of small integers. In practice, this might not always be the case, which motivates a preprocessing of the data to obtain small integers. We consider two well known and frequently used representatives of such preprocessing techniques: delta coding [34] and frame-of-reference [15], which replace each data element by the difference to its predecessor or to the minimum data element, respectively. The actual compression of the preprocessed data can be achieved by *cascading* the preprocessing with a Null Suppression algorithm [11, 13]. We call such cascades middleweight algorithms since the computational effort increases compared to lightweight algorithms, but it may improve the compression rate significantly on suitable data sets. In particular, we employ **DELTA + SIMD-BP128** and **FOR + SIMD-BP128**. Our cascades subdivide the data into pages of 4096 32-bit integers each (which fits into the L1 data cache of our machine) and apply for each page first the preprocessing and then SIMD-BP128 (compression) or vice versa (decompression).

Heavyweight Algorithms. While the previous two categories were mainly designed for in-memory (DRAM) systems, the different characteristics of NVRAM motivate us to also investigate general-purpose heavyweight data compression algorithms. Thus, we also employ **LZ4** [10] and **Snappy** [16] as two state-of-the-art representatives. Both LZ4 and Snappy belong to the LZ77 family of byte-oriented compression schemes. The LZ4 source code we use offers a recommended standard version, which we refer to as *LZ4* in this paper. Besides that, it also provides a number of variants targeting different trade-off levels between compression speed and rate. We also employ a variant optimized for highest speed, which we call *LZ4s*, and a variant for maximum compression, which we call *LZ4c*. We applied these algorithms on our integer data without any modification.

Final edited form was published in "SIGMOD/PODS '19: International Conference on Management of Data. Amsterdam 2019", Art. Nr. 11, ISBN 978-1-4503-6801-8
https://doi.org/10.1145/3329785.3329923

	# 32-bit ints	sorted	data distribution
D1	100 M	no	normal($\mu = 2^5$, $\sigma = 20$)
D2	100 M	no	normal($\mu = 2^{25}$, $\sigma = 20$)
D3	100 M	yes	uniform($min = 0$, $max = 10^6 - 1$)
D4	100 M	no	90% normal($\mu = 2^5$, $\sigma = 20$) 10% normal($\mu = 2^{25}$, $\sigma = 20$)

Table 1: The synthetic data sets used in our evaluation.

3.2 Implementation Remarks

All algorithms are written in C/C++ and compiled using g++-7.1 with the -O3 optimization flag. Our considered light- and middleweight algorithms are vectorized using Intel's SIMD instruction set extension SSE working on 128-bit vector registers. For SIMD-BP128 and SIMD-FastPFOR, we used the implementations from the FastPFor-library [26]. For the cascades, we used our own hand-tuned and vectorized implementations of DELTA and FOR [11, 13]. Moreover, we utilized the original source for the heavyweight algorithms. To enable the correct NVRAM usage, we modified all algorithms using two aspects: first, we provisioned an access to persistent memory via memory mapped files on a DAX-enabled file system (where DAX stands for Direct Access, meaning that physical NVRAM regions are mapped into the virtual address space of an application while bypassing kernel page cache), second, we used a combination of `CLFLUSHOPT` plus `SFENCE` to ensure persistency [43]. Due to the *DAX* feature, no system calls are required to propagate the modifications to NVRAM. However, the presence of CPU caches requires explicit cache line flushes to guarantee the persistency of store operations.

4 COMPRESSION EVALUATION

In this section, we present the evaluation results of our investigated compression algorithms on DRAM and NVRAM. Previous works have shown that the data characteristics determine which algorithm is the most suitable one [11, 13]. Thus, we experiment with several synthetic data sets being summarized in Table 1. We used our compression benchmark framework as a solid execution environment [12, 14]. Figure 5—we use short names such as BP128 for SIMD-BP128 or DELTA for DELTA+SIMD-BP128 in all our figures—reports the compression rates achieved by each algorithm on each data set and clearly shows that there are significant differences even within each of the categories light-, middle-, and heavyweight algorithms. Obviously these compression rates are independent of the type of memory as well as of the number of threads used, so they serve as a reference throughout our evaluation.

We measured the performance of an algorithm by running it on a data set for few seconds and counting how many times it could process the entire data set. This procedure is especially important to guarantee that all threads execute simultaneously in the multi-threaded scenarios. Thus, loading/storing data and computations are included in the measurement. We report (i) performances in *million integers per second* (mis) referring to the underlying number of logical data elements, not to the physical data size and (ii) throughput/bandwidth in GiB/s. We omit Snappy in our evaluation, since its results were mostly similar to those of standard LZ4. However, we additionally include LZ4s and LZ4c, which are pre-configured for higher execution speed and for better compression rate, respectively, by the developers.

4.1 Single-Threaded Experiments

Figure 6 presents the performances of the single-threaded execution of all algorithms on all data sets. First, we compare the performance on NVRAM to that on DRAM. Figure 6(a-b) shows the speeds achieved on NVRAM relative to those on DRAM. As a general observation, the performance on NVRAM is never better than that on DRAM. As a consequence of the lower bandwidth of NVRAM compared to DRAM, the algorithms reach only between 25 % and 100 % in case of compression, or 17 % and 39 % of the DRAM performance in case of decompression.

In the following, we focus on the impact of the memory type on the choice of the most suitable compression algorithm. This topic has already been investigated in detail for DRAM [13]. However, the lower bandwidth of NVRAM suggests to prefer algorithms investing more cycles for computation if this can improve the compression rate appropriately, since that way, the effective bandwidth may be improved. We check this hypothesis by comparing the DRAM and NVRAM performances of the low-effort SIMD-BP128 to those of the other more compute-intensive algorithms. We focus on the compression in Figure 6(c), since it is generally more compute-intensive than decompression as shown in Figure 6(d).

On data set D1 (small integers), no algorithm achieves a significantly better compression rate than SIMD-BP128 and in fact, this algorithm achieves the best compression speed on both DRAM and NVRAM. However, on data set D2 (large integers in a narrow range), the situation changes. Here, FOR + SIMD-BP128 achieves by far the best compression rate. While on DRAM this cascade achieves only 87 % of the speed of SIMD-BP128, it is 72 % faster than SIMD-BP128 on NVRAM. The heavyweight variants of LZ4 also achieve much better compression rates than SIMD-BP128. Nevertheless, they fail to outperform it with respect to compression speed on both mediums, since they are to compute-intensive. On the sorted data set D3, the three variants of the heavyweight algorithm LZ4 achieve the best compression rates. However, regarding compression speed, only the standard LZ4 and LZ4s perform superb: they are fastest compressors on both DRAM and NVRAM, while LZ4c is the slowest of all algorithms on both mediums, due to its high computational effort. Regarding the lightweight algorithms, SIMD-BP128 again is the fastest on DRAM, but again it is outpaced by up to 83% by the more compute-intensive cascades on NVRAM. One tenth of data set D4 are large outliers. Here, the best compression rate is achieved

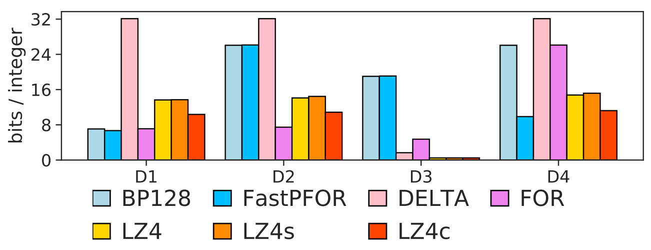


Figure 5: Compression rates.

Final edited form was published in "SIGMOD/PODS '19: International Conference on Management of Data. Amsterdam 2019", Art. Nr. 11, ISBN 978-1-4503-6801-8
https://doi.org/10.1145/3329785.3329923

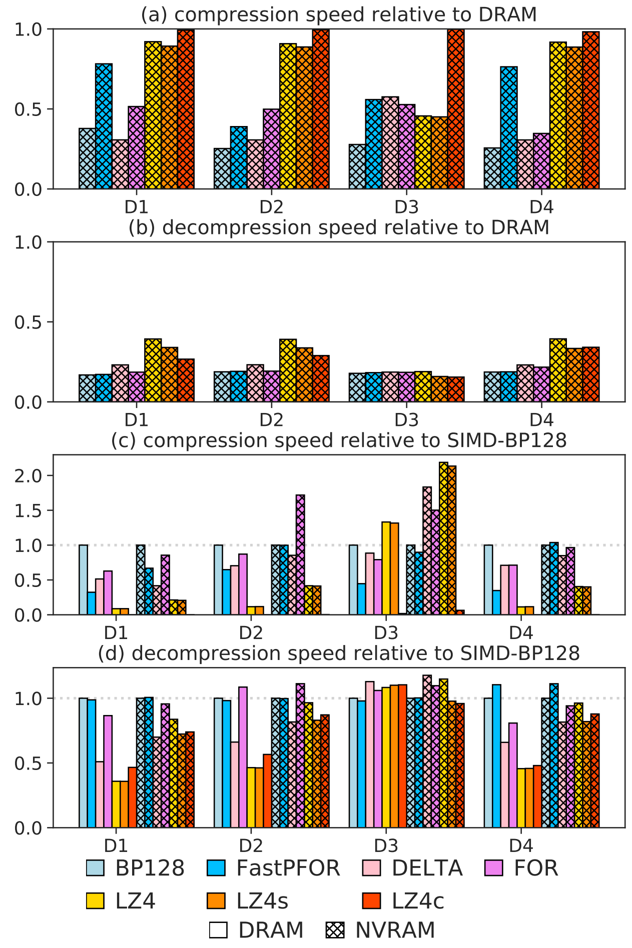


Figure 6: Single-threaded speeds: NVRAM relative to DRAM and algorithms relative to SIMD-BP128 *on the same medium.*

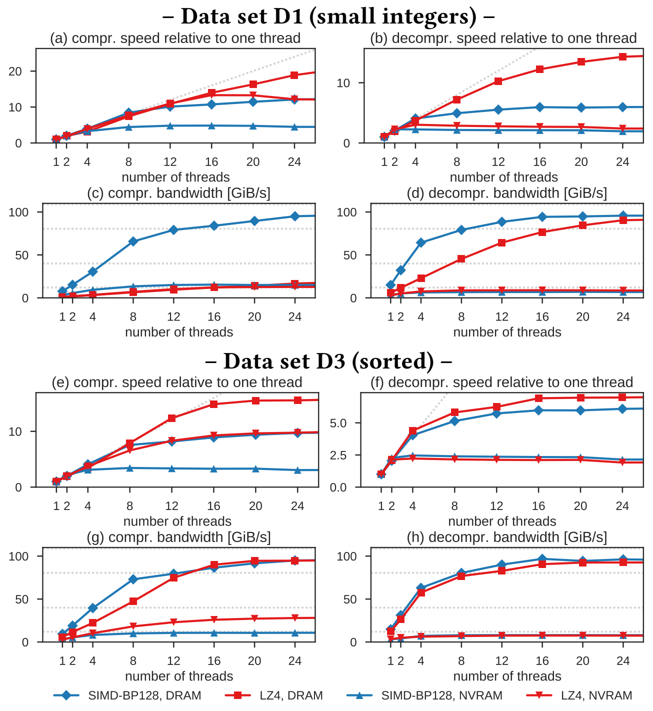


Figure 7: Multi-threaded performances on D1 and D3: speed relative to single-threaded (dotted line is linear scaling), absolute bandwidths (dotted lines are read and write bounds).

by SIMD-FastPFor, but LZ4c as almost equally good. Regarding SIMD-FastPFor, the price for its special treatment of outliers is a speed of only 35 % of that of SIMD-BP128 on DRAM. However, it achieves a small speed-up of 4 % on NVRAM. On the contrary, LZ4c has a compression speed close to zero on both mediums.

Conclusions. We conclude that the special characteristics of NVRAM necessitate a rethinking of the trade-offs involved in the selection of the fastest compression algorithm. On NVRAM, it is recommendable to invest more computations for a better compression rate. However, while the size reduction achieved by general-purpose heavyweight algorithms does not always balance for their computational cost, the middleweight cascades of lightweight algorithms are a good choice.

4.2 Multi-Threaded Experiments

In the following, we present our observations about the DRAM and NVRAM experiments in the multi-threaded scenario. We show the compression speeds relative to a single thread in Figure 7(a-b, e-f) and the respective absolute memory bandwidth consumed per second in Figure 7(c-d, g-h). However, the latter does not distinguish between write and read components, as it was obtained through the Intel *PQoS* monitoring tool [21]. The analysis is based on two selected algorithms and data sets (SIMD-BP128, standard LZ4 on D1, D3) as the most interesting cases.

As mentioned above, the compression rate, of course, is not affected by the number of threads. The performance on NVRAM is never better than that on DRAM for the same number of threads. A further observation is that both compression algorithms exhibit different scalability behaviors on DRAM and NVRAM. The LZ4 DRAM performance grows nearly linearly until 16 and 12 threads on D1 and D3, respectively. For SIMD-BP128 this result is lower, which is explained by the fact that the corresponding consumed bandwidth increases faster: 8 threads on both data sets. However, both approaches demonstrate a stable speed growth up to 20 threads independently of the data characteristics. At this point the DRAM bandwidth limitations are actually reached (cf. Section 2). One exception here is the LZ4 (de)compression on D1, which demonstrates a constant growth up to 24 threads (and even further until 48), and is, in contrast to the others, not memory-bound, but compute-bound. The NVRAM scalability diverges from its ideal linear case at 8 (D1) and 4 (D3) threads for heavyweight LZ4 compression and already at 2 for the lightweight SIMD-BP128, while a certain performance increase is still observed until 16 and 8 threads, respectively.

Regarding the decompression, we observe a mostly similar situation with a certain degradation of the thread count: on DRAM, the linear scalability is bounded by 4 threads (except for LZ4 as mentioned above) and stable growth hits the limit at around 16 threads,

Final edited form was published in "SIGMOD/PODS '19: International Conference on Management of Data. Amsterdam 2019", Art. Nr. 11, ISBN 978-1-4503-6801-8
https://doi.org/10.1145/3329785.3329923

while on NVRAM, even 2 threads cannot achieve a linear scaling anymore after 8 threads. There is no further performance increase because decompression is more write-intensive than compression and, therefore, is more dependent on write bandwidth boundaries, which are lower than that of the read counterpart on both mediums.

Overall, we detected the following accelerations via the deployment of multiple threads (best performance in a range from 1 to 24 threads) for DRAM compression/decompression: SIMD-BP128 on D1 – 11.5x / 5.3x, SIMD-BP128 on D3 – 7x / 6x, LZ4 on D1 – 18.5x / 14x, LZ4 on D3 – 11.9x / 6.7x. For NVRAM the following numbers are observed: SIMD-BP128 on D1 – 3.7x / 1.2x, SIMD-BP128 on D3 – 2.1x / 1.4x, LZ4 on D1 – 11.2x / 1.8x, LZ4 on D3 – 9.5x / 1.2x.

Conclusions. Our general outcome is that the multi-threaded scalability of (de)compression algorithms is much better on DRAM, due to higher throughput limits, though there is an exception – the compute-intensive standard LZ4 compression, which scales well also on NVRAM. Moreover, the scalability property depends not only on the medium but also on the algorithm and input data.

4.3 Interplayed NVRAM-DRAM Placement

In the following series of experiments we investigate the influence of interplayed data placement on the performance of our (de)compression workloads. Again we provide the results only for two data sets – D1 and D3. The thread count is set to 24 as the generally most performable configuration for all participating algorithms. Figure 8(a-b) shows the execution speed relative to the DRAM-only case for the three other possible data placements: NVRAM-only and the two interplayed cases DRAM→NVRAM and NVRAM→DRAM, where the left side of "→" corresponds to the input buffer location and the right side to the output buffer location.

From Figure 8(a), we see that the compression speed of the interplayed setups fills the gap between DRAM- and NVRAM-only allocations, whereby NVRAM→DRAM usually incurs the least slowdown by reaching between 41 % and 99 % of the DRAM-only performance. This workload leverages the high write bandwidth of DRAM and is not required to flush the caches of a volatile output buffer. Further, we observe that the compression speed of the DRAM→NVRAM scenario, though it is quite close to NVRAM-only, still outperforms it in all but one experiment due to the higher DRAM read bandwidth. The only exception to this is LZ4c, whose high computational effort causes the combination of mediums to have no visible effect on its performance. Moreover, the middleweight DELTA + SIMD-BP128 and the heavyweight standard LZ4 and LZ4s prefer this placement to NVRAM→DRAM for D3. The decompression results in Figure 8(b) generally demonstrate a similar trend, however NVRAM→DRAM converges closer to the DRAM-only and DRAM→NVRAM closer to the NVRAM-only settings, since the decompression has a higher sensitivity to the write throughput of the output medium than the compression.

In hybrid DRAM-NVRAM-systems, moving data between mediums is a natural idea. Assuming uncompressed data on the source medium, a trivial approach is to copy the data as-is. In the following, we investigate if it is beneficial to use a compressed format on the destination medium, i.e., to include compression into the transfer to increase the effective bandwidth, and to include decompression into the back-transfer to restore the original data format. Such data

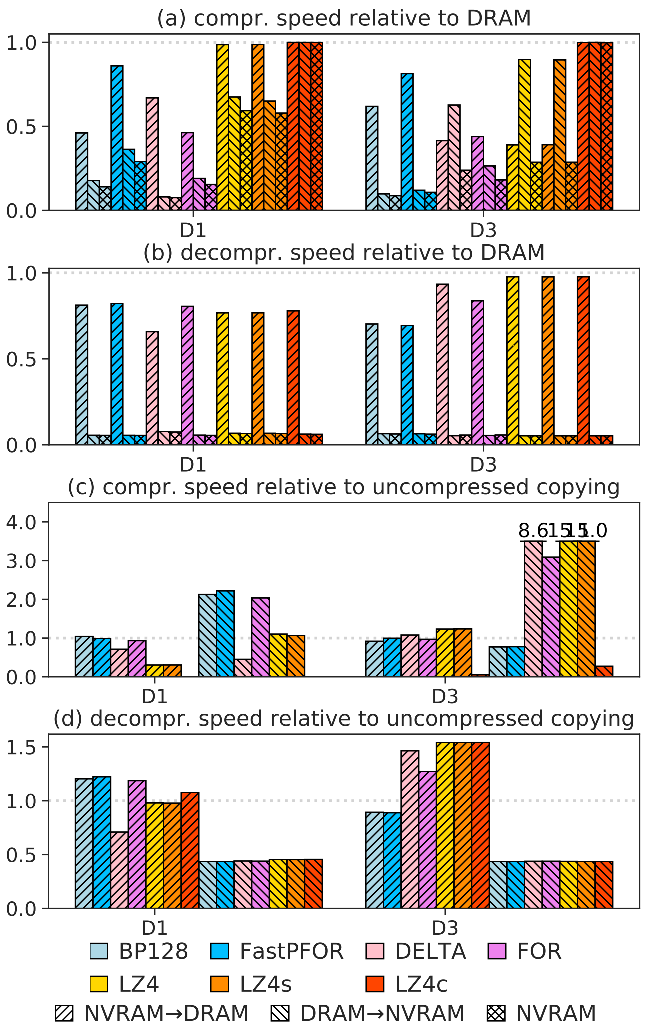


Figure 8: Interplay setups: 24-thread speeds relative to DRAM and relative to copying uncompressed data.

flows are easily enabled by allocating the input and output buffers of a (de)compression algorithm on different mediums. Figure 8(c-d) shows the speed of (de)compression relative to copying uncompressed data between the mediums. When the data is moved from NVRAM to DRAM, using a suitable compression algorithm can result in speed-ups of up to 4 % (D1) and 23 % (D3). However, when moving the data back to NVRAM, a decompression achieves only 44 % of the performance of copying uncompressed data. The reason is that the low write bandwidth of NVRAM is saturated in both cases, while the decompression adds its computational overhead on top. On the contrary, when the data is moved from DRAM to NVRAM, the use of a suitable compression algorithm can yield significant speed-ups of up to 2.2x (D1) and even 15x (D3). Due to the low write bandwidth of NVRAM, extra computations for compression are able to increase the effective bandwidth. However, too heavyweight algorithms such as LZ4c fail to achieve the right balance and are, thus, not beneficial in such settings. Moreover, even for the back-transfer of the data to DRAM, decompression achieves speed-ups of up to 1.2x (D1) and 1.5x (D3) compared to copying back uncompressed data. While both cases materialize the same data in DRAM, reading compressed data is less costly than reading uncompressed data due to the low read bandwidth of NVRAM.

Conclusions. We conclude that interplayed allocation is a good way to bridge the performance gap between basic settings. Moreover, compression can be employed to accelerate inter-memory data flows, especially on the way to NVRAM.

4.4 SIMD Extensions

In recent years, the vectorization of lightweight and middleweight compression algorithms has been a main driver in this research field. As a consequence, most of these algorithms are tailored to a specific SIMD instruction set extension. For instance, the light- and middleweight algorithms we investigated so far are tailored to Intel's SSE, whose vector registers have a size of 128 bits. In fact, many light- and middleweight compression algorithms can be ported to more recent SIMD extensions, such as Intel's AVX2 and AVX-512. These newer extensions provide 256-bit and 512-bit vector registers, respectively. An obvious expectation could be that these larger vector registers yield speed-ups of 2x or 4x, respectively, such that more data elements can be processing in one instruction. However, in our recent article [13], we have already proven this expectation not to be fulfilled by current hardware, since the algorithms quickly become memory-bound as the vector size increases. Our setting was an Intel Xeon Phi using only DRAM in a single-threaded way. In the following, we extend our scope to cover also NVRAM and the multi-threaded case.

For these experiments, we ported the SSE-targeted implementations of the light- and middleweight algorithms used in this paper to AVX2 and AVX-512 in a *straightforward* way. This means, we replaced all SSE intrinsics in the code by their AVX2 or AVX-512 counterparts for wider vector registers. Furthermore, we adapted the storage layouts in a straightforward way to the processing with wider vector registers, e.g., by increasing block sizes. We refer to our ported implementations using the name of the original algorithm followed by the vector width in bits.

In the following, we again focus on data sets D1 and D3. Figure 9 shows the results. First of all, Figure 9(a) reveals that the compression rates achieved by the different variants of the lightweight algorithms SIMD-BP and SIMD-FastPFor differ only insignificantly on these data sets, while for DELTA + SIMD-BP on the sorted data set D3 the compression rate gets worse as the vector width increases.

Next, we compare the speeds achieved using the newer SIMD extensions AVX2 and AVX-512 to those achieved with SSE. Figure 9(b-c) shows the (de)compression speeds relative to those achieved by the SSE-variant of the same algorithm. At first glance, we can see that the desired speed-ups of 2x and 4x cannot be achieved at all, neither on DRAM, nor on NVRAM. Regarding the compression speeds, increasing the vector size often results in a slow-down of up to 17 % on DRAM and up to 8 % on NVRAM. This is most likely a consequence of the fact, that only reads, but not writes can benefit

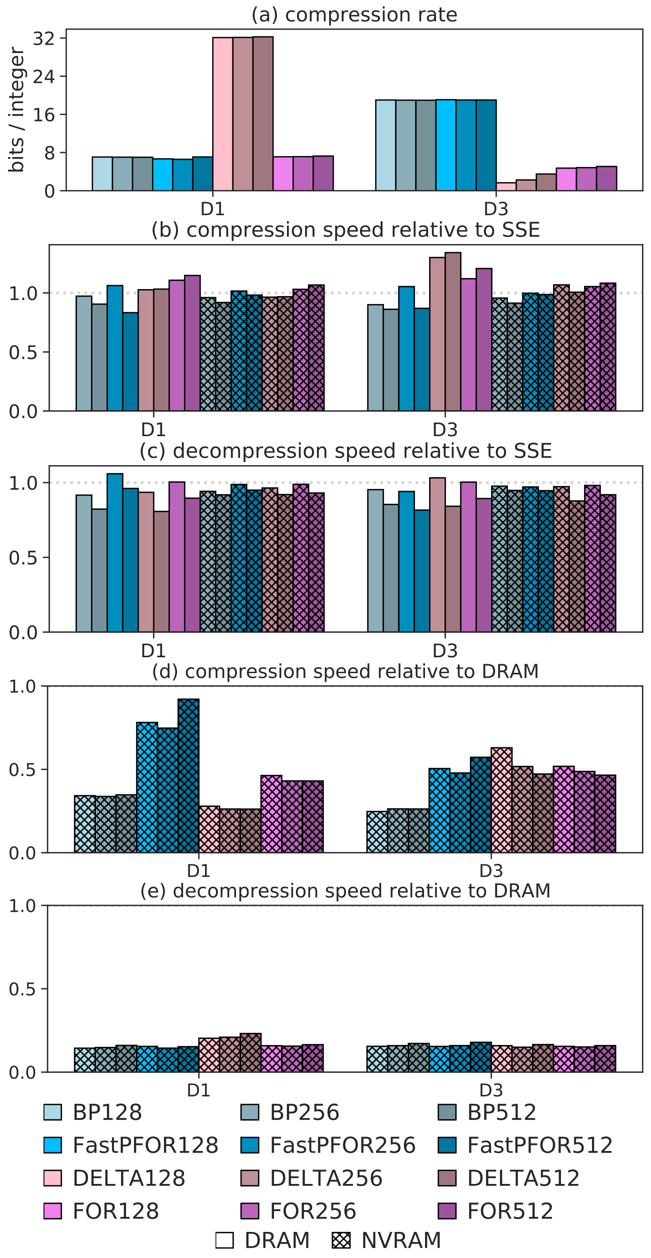


Figure 9: Compression rates and single-threaded relative performances of the light- and middleweight compression algorithms tailored to different SIMD extensions.

from larger vector registers on our experimentation platform. Consequently, these effects propagate to SIMD-BP, which does only a little amount of computation compared to load/store operations. On the contrary, the middleweight cascades do achieve speed-ups compared to their SSE implementations, since they involve more computations, which can, indeed, profit from wider vectors. Nevertheless, these speed-ups reach only up to 34 % on DRAM and up to 8 % on NVRAM. Regarding the decompression, the SSE variant

Final edited form was published in "SIGMOD/PODS '19: International Conference on Management of Data. Amsterdam 2019", Art. Nr. 11, ISBN 978-1-4503-6801-8
https://doi.org/10.1145/3329785.3329923

achieves the best performance for most algorithms, while AVX-512 yields slow-downs of up to 19 % on DRAM and 12 % on NVRAM. The only exceptions are SIMD-FastPFor and DELTA + SIMD-BP, where the AVX2 variant is the fastest on DRAM. We also varied the number of threads and recognized that the performance differences between the SIMD variants of one algorithm become less significant as the number of threads increases.

Finally, we investigate the impact of the employed SIMD extension on the relative speeds on NVRAM compared to DRAM. Figure 9(d-e) provides the results for the compression and decompression, respectively. Regarding the compression, wider vectors tend to cause an even stronger slow-down from DRAM to NVRAM. This can be explained by the fact that, with accelerated computations, the impact of the memory access on the overall performance grows. Furthermore, especially for the cascades on data set D3, an increase of the vector size causes worse compression rates, i.e., more compressed data must be written by the algorithm, which clearly propagates to the worse relative speeds of the AVX2 and AVX-512 variants of the cascades compared to their SSE variants. The only exception here is SIMD-FastPFor512, which incurs a less severe slow-down from DRAM to NVRAM than its SSE and AVX2 counterparts. Regarding the decompression, the differences between the SIMD variants of one algorithm are generally insignificant. Again, we also experimented with different number of threads and found them to further equalize the results of different SIMD variants of one algorithm.

Conclusions. We conclude that, with a single thread, especially the compression of the middleweight cascades can benefit from newer vector extensions. At the same time, the compression of the lightweight algorithms as well as the decompression should best be done with SSE. For a high number of threads, however, the employed SIMD extension does not make a significant difference.

5 DATABASE OPERATIONS EVALUATION

This section is devoted to the more complex operations that in-memory DBMSs are able to execute on compressed data. We focus on two frequently used operations: column scan aggregation and selection.

5.1 Aggregation

As a first example of a database operation, we address a column scan aggregating the compressed data.

Implementation. We implemented the aggregation operators the following way: For the light- and middleweight algorithms, we changed the *SSE variant* of the decompression, such that it adds the decompressed data elements to a running sum, instead of storing them to an output buffer. For the standard variant of LZ4, we decompress the data in blocks of 16 KiB (which fit into the L1 data cache of our experimentation platform) and execute an aggregation on the uncompressed data residing in the L1 cache.

Evaluation. In Figure 10(a), we only consider the basic placements, because only a negligible single data element is actually written to the output buffer. On NVRAM, the aggregation performance can be as low as 36 % of that on DRAM, which shows that the effects observed for (de)compression-only workloads also propagate to database operations.

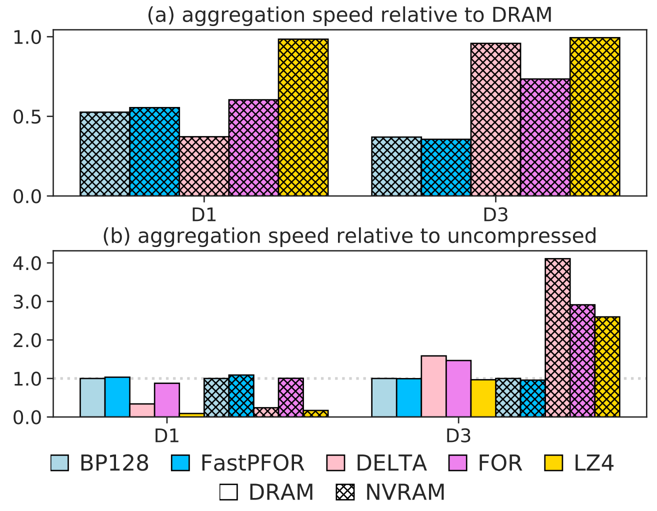


Figure 10: 24-thread aggregation speeds relative to DRAM and relative to aggregating uncompressed data.

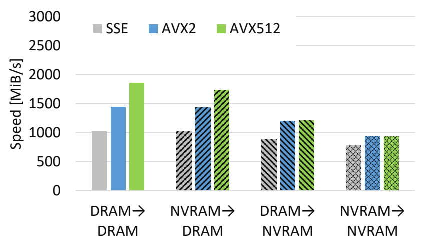
Next, we investigate the speed-up achieved by aggregating compressed data compared to uncompressed data. The results are shown in Figure 10(b). We observe that the speed-ups achieved by a compression scheme suitable for the data set are higher on NVRAM (up to 4.1x for the SSE variant of DELTA + SIMD-BP128 on D3) than on DRAM (only 1.6x for the same algorithm and data set).

Conclusions. We conclude that processing compressed data directly – while already being reasonable on DRAM – is even more important on NVRAM and can, at least partially, compensate this novel medium's lower bandwidth.

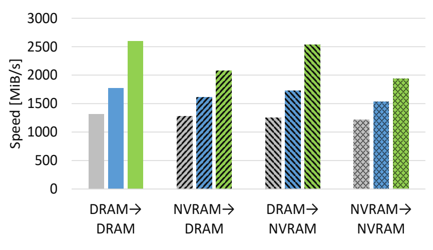
5.2 Selection

As a second example of a database operation, we investigate a selection using a full column scan over compressed data. One of the state-of-the-art approaches for this is called *BitWeaving* [27]. BitWeaving requires an order-preserving mapping of values to fixed-bit-width codes. This mapping can be obtained from a lightweight or middleweight compression algorithm, such as SIMD-BP128 or FOR. The arrangement of these codes in memory follows a specific layout, which facilitates efficient scan operations. For instance, with BitWeaving/H, the codes are packed with the selected fixed bit width, whereby a special delimiter bit is inserted between adjacent codes. In the following, we focus on BitWeaving/H, since it closely resembles the storage layout of vertical bit packing, which is used by all lightweight and middleweight compression algorithms we consider in this paper. The BitWeaving/H column scan algorithm reads a sequence of such codes and writes a bit vector indicating for each logical data element whether it fulfils the selection predicate. All commonly used predicates such as (in)equality and range-comparisons are supported and can be calculated using full-word bitwise and arithmetic instructions. Depending on the predicate, the number of instructions and, thus, the ratio of load/store and computation differs. For instance, evaluating the equality predicate requires 4 instructions per (vector)register, while the greater-than predicate requires only 3 instructions. Furthermore, with smaller

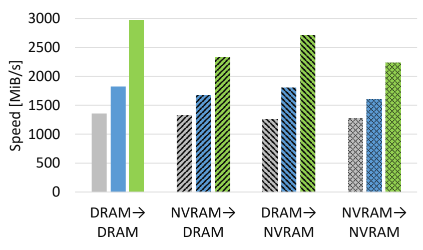
Final edited form was published in "SIGMOD/PODS '19: International Conference on Management of Data. Amsterdam 2019", Art. Nr. 11, ISBN 978-1-4503-6801-8
https://doi.org/10.1145/3329785.3329923



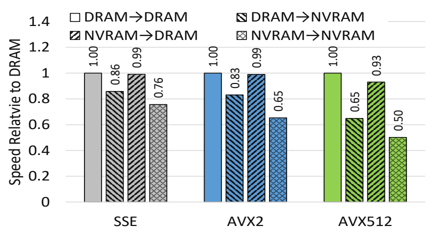
(a) Absolute speed, bit width 1.



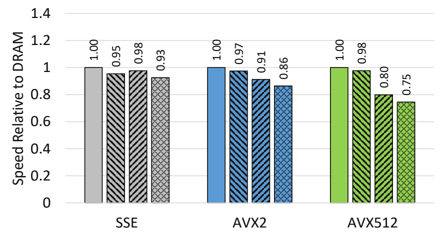
(b) Absolute speed, bit width 15.



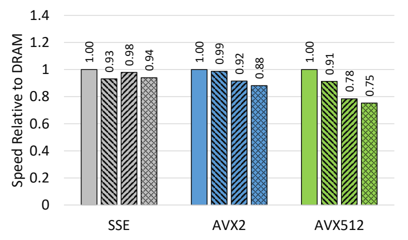
(c) Absolute speed, bit width 31.



(d) Speed relative to DRAM⟶DRAM, bit width 1.

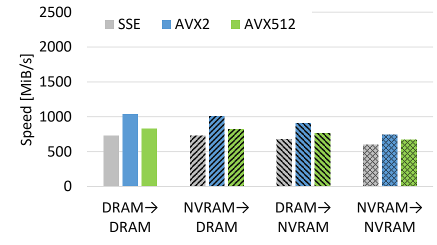


(e) Speed relative to DRAM⟶DRAM, bit width 15.

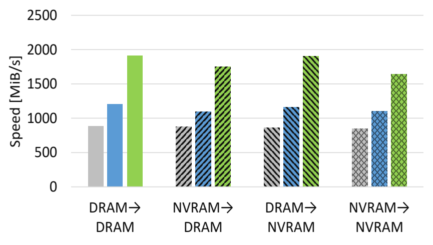


(f) Speed relative to DRAM⟶DRAM, bit width 31.

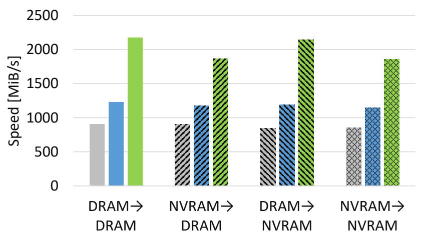
Figure 11: Execution speeds (absolute and relative to DRAM-only) for different SIMD extensions, bit widths and memory settings. Particular exemplification of BitWeaving/H algorithm with the "greater-than" predicate.



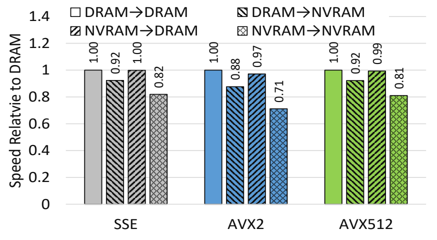
(a) Absolute speed, bit width 1.



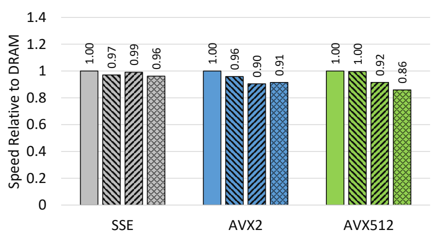
(b) Absolute speed, bit width 15.



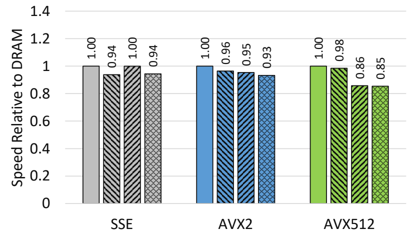
(c) Absolute speed, bit width 31.



(d) Speed relative to DRAM⟶DRAM, bit width 1.



(e) Speed relative to DRAM⟶DRAM, bit width 15.



(f) Speed relative to DRAM⟶DRAM, bit width 31.

Figure 12: Execution speeds (absolute and relative to DRAM-only) for different SIMD extensions, bit widths and memory settings. Particular exemplification of BitWeaving/H algorithm with the "equality" predicate.

bit widths, more logical values can be processed at once, than with larger bit widths. However, at the same time, with smaller bit widths, the ratio between physical input and output size is less extreme, e.g., for a bit width of 1, the output is half as large as the input, while for a bit width of 31, the output has $\frac{1}{32}$ of the input's size.

Implementation. We obtained a *scalar* version of the BitWeaving source code from the original authors [27] and *vectorized* it using Intel's SIMD extensions in a straightforward way. The same general implementation remarks as mentioned in Section 3.2 hold. For all considered vector extensions the code is implemented in C++ and compiled with g++-7.1 with -O3, and we make use of NVRAM as mentioned above.

Evaluation. For the BitWeaving/H algorithm we present two separate sets of single-threaded experiments (Figures 11 and 12) varying the bit width (e.g., 1, 15 and 31), deployed SIMD vector extension (e.g., SSE, AVX2, AVX-512) and input-output buffer allocation (our four scenarios). The BitWeaving/H algorithm is completely agnostic of the data characteristics and depends only on the number of bits used to represent a data element. Thus, instead of re-using the data sets we used so far, we simply execute the BitWeaving/H selection on a 4 GiB input buffer in the BitWeaving/H storage layout containing random data elements of the respective bit width. The reported execution speeds in MiB/s refer to the input size, while the output size differs depending on the bit width.

The first set of charts is devoted to the BitWeaving/H column scan algorithm based on the greater-than selection predicate. As demonstrated in Figures 11(a)-11(c) the absolute execution speed can be significantly increased via applying larger vector registers and actually reaches its peak using the AVX-512 instruction set. This observation holds for all memory combinations and bit widths, apart from experiments with a bit width equal to 1 writing the output to NVRAM. Such configurations feature similar throughput for both AVX2 and AVX-512. Overall, the performance increase is higher for larger bit widths. To better understand the influence of the deployed mediums for input and output buffer placement, we show the speeds relative to DRAM→DRAM in Figures 11(d)-11(f). Similarly to the compression experiments (cf., Section 4), we observe that the performance of NVRAM-only and interplayed schemes is never faster than that of DRAM-only allocation. Furthermore, the experiments revealed that the NVRAM→DRAM configuration is preferable for small bit widths (e.g., 1) as it can reach from 93 % (AVX-512) to 99 % (SSE and AVX2) of the DRAM→DRAM speed, while the other alternatives degrade the performance more significantly (possibly to just 50 % of DRAM-only for AVX-512 executed on the NVRAM-only setup). This result is explained by the fact that for smaller bit widths, more data is being stored to the output, and, therefore, the NVRAM→DRAM setup is able to take advantage of the lower DRAM latency. On the contrary, for larger bit widths, less data is expected to be written and flushed, thus, the performance of read operations start to dominate. Indeed, for most of the larger bit width experiments (Figures 11(e)-11(f)) the DRAM→NVRAM setting provides the best performance (from 91 % to 99 % of the DRAM-only allocation).

The second set of experiments investigates the BitWeaving/H column scan algorithm based on the equality selection predicate (Figure 12). Generally, most of the first set observations still hold for the second group and here we mention only the differences. Comparing to the greater-than implementation, the equality predicate requires 4 instructions instead of 3 for its execution step. Hence, we see that the absolute speed (Figures 12(a)-12(c))) is always lower than the respective one of the previous case. Unlike the bit width 1 experiments of the greater-then predicate (Figure 11(a)), the equality version (Figure 12(a)) reaches the best absolute speed using the AVX2 and not the AVX-512 SIMD extension for all buffer allocation settings. Finally, the performance drop between the DRAM→DRAM setup and other schemes is typically lower compared to the greater-than case, since the increased weight of the compute component decreases the memory accesses and compute ratio.

Conclusions. We conclude that, unlike to most compression experiments of Section 4.4, the use of SIMD extensions for the selection scans is justified and able to deliver significantly better performance compared to the SSE implementation. Moreover, the choice of the most practical SIMD variant as well as memory setting depends on the targeted bit width, as it determines the memory/compute ratio in the resulting code.

6 RELATED WORK

Since in-memory integer compression was already thoroughly addressed in Section 3, we omit the respective discussion here and proceed with NVRAM-related research. For smoothed deployment of this memory class in the wide range of applications a number of specialised programming tool-kits, file systems and allocators [2, 5, 28, 35, 36, 40, 42] were developed. Thus, it is now possible to adapt traditional in-DRAM DBMSs to NVRAM-centric data stores via the development of persistent memory-tuned data structures and algorithms [8, 9, 22, 30, 31, 37, 38, 41, 41]. The existing examples of such hybrid DRAM-NVRAM engines are SAP HANA [3], SOFORT [29, 32, 33], FOEDUS [23] and Peloton [4]. However, to the best of our knowledge, the behavior of data compression algorithms in such hybrid main memory systems has not been researched so far and now, with the help of our investigations, they are able to minimize the memory footprint, accelerate the inter-memory data flows, and increase the hardware utilization.

7 CONCLUSIONS

In this paper, we addressed the topic of integer data compression for NVRAM-centric data stores. Similarly to DRAM-backed in-memory systems, compression is an effective way to reduce the size of primary data and to overcome the memory bandwidth limitations (the latter is especially sensible for NVRAM). However, due to specific features (e.g. asymmetric read-write latencies) of NVRAM, the behavior of the state-of-the-art compression schemes requires an analysis to better understand their applicability and to fully exploit the possible advantages of persistent memory. Thus, we provided a detailed experimental evaluation of selected light- and middleweight (using Intel's vector extensions SSE, AVX2, and AVX-512) as well as heavyweight data compression schemes in this paper. Within our evaluation, we revealed that NVRAM provides worse scalability opportunities than DRAM and prefers middleweight (cascades of lightweight) compression algorithms. We showed that compression is an effective way not only to save space but also to speed up the inter-memory data flows. Moreover, our investigation of two important database operations, aggregation and selection, revealed that processing compressed data directly is even more recommendable on NVRAM than it is on DRAM.

8 ACKNOWLEDGMENT

This work is partly funded by the German Research Foundation (DFG) in the context of (i) the project "Self-Recoverable and Highly Available Data Structures for NVRAM-centric Database Systems" (LE-1416/27-1), (ii) by RTG 1907 (RoSI), and (iii) by DFG-CRC 912 (HAEC).

REFERENCES

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *SIGMOD*. 671–682.

[2] Alfons Kemper Thomas Neumann Takushi Hashida Kazuichi Oe Yoshiyasu Doi Lilian Harada Sato Mitsuru Alexander van Renen, Viktor Leis. 2018. Managing Non-Volatile Memory in Database Systems. In *SIGMOD*. 691–706.

[3] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. 2017. SAP HANA Adoption of Non-volatile Memory. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1754–1765. https://doi.org/10.14778/3137765.3137780

[4] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *PVLDB* 10, 4 (2016), 337–348.

[5] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-Juergen Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *OOPSLA*. 677–694.

[6] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*. 283–296.

[7] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*.

[8] Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2011. Rethinking Database Algorithms for Phase Change Memory. In *CIDR*. 21–31.

[9] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB* 8, 7 (2015), 786–797.

[10] LZ4 Community. [n. d.]. LZ4. https://lz4.github.io/lz4/.

[11] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *EDBT*. 72–83.

[12] Patrick Damme, Dirk Habich, and Wolfgang Lehner. 2015. A Benchmark Framework for Data Compression Techniques. In *TPCTC*. 77–93.

[13] Patrick Damme, Annett Ungethüm, Juliana Hidebrandt, Dirk Habich, and Wolfgang Lehner. 2019. From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms. In *accepted for publication in TODS*.

[14] Patrick Damme et al. 2019. Lightweight compression benchmarking and selection. https://github.com/MorphStore/LC-BaSe

[15] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *ICDE*. 370–379.

[16] Google. 2019. Snappy - A fast compressor/decompressor. https://github.com/google/snappy.

[17] Dirk Habich, Patrick Damme, Annett Ungethüm, Alexander Krause, Juliana Hildebrandt, and Wolfgang Lehner. 2019. MorphStore — In-Memory Query Processing based on Morphing Compressed Intermediates LIVE. In *SIGMOD*. 1–4.

[18] Juliana Hildebrandt, Dirk Habich, Patrick Damme, and Wolfgang Lehner. 2016. Compression-Aware In-Memory Query Processing: Vision, System Design and Beyond. In *ADMS@VLDB*. 40–56.

[19] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers* 40, 9 (September 1952), 1098–1101.

[20] Intel 2019. *Intel Optane DC Persistent Memory Module*. Intel. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[21] Intel 2019. *Intel PQoS Utility*. Intel. https://github.com/intel/intel-cmt-cat/tree/master/pqos

[22] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. 2018. clfB-tree: Cacheline Friendly Persistent B-tree for NVRAM. *ACM Trans. Storage* 14, 1 (Feb. 2018), 5:1–5:17.

[23] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *SIGMOD*. 691–706.

[24] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*. 311–326.

[25] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.* 45, 1 (2015), 1–29.

[26] Daniel Lemire, Leonid Boytsov, Owen Kaser, Maxime Caron, Louis Dionne, Michel Lemay, Erik Kruus, Andrea Bedini, Matthias Petri, Robson Braga Araujo, and Patrick Damme. 2019. The FastPFOR C++ library: Fast integer compression. https://github.com/lemire/FastPFOR

[27] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: fast scans for main memory data processing. In *SIGMOD*. 289–300.

[28] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *TRIOS@SOSP*. 1:1–1:17.

[29] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In *DaMoN*. 8:1–8:7.

[30] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*. 371–386.

[31] Ismail Oukid and Wolfgang Lehner. 2017. Data Structure Engineering For Byte-Addressable Non-Volatile Memory. In *SIGMOD*. 1759–1764.

[32] Ismail Oukid and Wolfgang Lehner. 2017. Towards a Single-Level Database Architecture on Non-Volatile Memory. In *NVMW*.

[33] Ismail Oukid, Wolfgang Lehner, Thomas Kissinger, Thomas Willhalm, and Peter Bumbulis. 2015. Instant Recovery for Main Memory Databases. In *CIDR*.

[34] Mark A. Roth and Scott J. Van Horn. 1993. Database Compression. *SIGMOD Rec.* 22, 3 (1993), 31–39.

[35] Andy Rudoff. 2015. Persistent Memory Programming. *Login: The Usenix Magazine* 42 (2015), 34–40.

[36] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. 2015. nvm malloc: Memory Allocation for NVRAM. In *ADMS@VLDB*.

[37] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *FAST*. 5–5.

[38] Stratis Viglas. 2014. Write-limited sorts and joins for persistent memory. *PVLDB* 7, 5 (2014), 413–424.

[39] Ian H. Witten, Radford M. Neal, and John G. Cleary. 1987. Arithmetic Coding for Data Compression. *Commun. ACM* 30, 6 (June 1987), 520–540.

[40] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[41] Jun Yang, Qingsong Wei, Chundong Wang, Cheng Chen, Khai Leong Yong, and Bingsheng He. 2016. NV-Tree: A Consistent and Workload-Adaptive Tree Structure for Non-Volatile Memory. *IEEE Trans. Computers* 65, 7 (2016), 2169–2183.

[42] Songping Yu, Nong Xiao, Mingzhu Deng, Yuxuan Xing, Fang Liu, Zhiping Cai, and Wei Chen. 2015. WAlloc: An efficient wear-aware allocator for non-volatile main memory. In *IPCCC*. 1–8.

[43] Mikhail Zarubin, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. 2018. Efficient Compute Node-local Replication Mechanisms for NVRAM-centric Data Structures. In *Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18)*. ACM, New York, NY, USA, Article 7, 9 pages. https://doi.org/10.1145/3211922.3211931

[44] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.* 23, 3 (1977), 337–343.