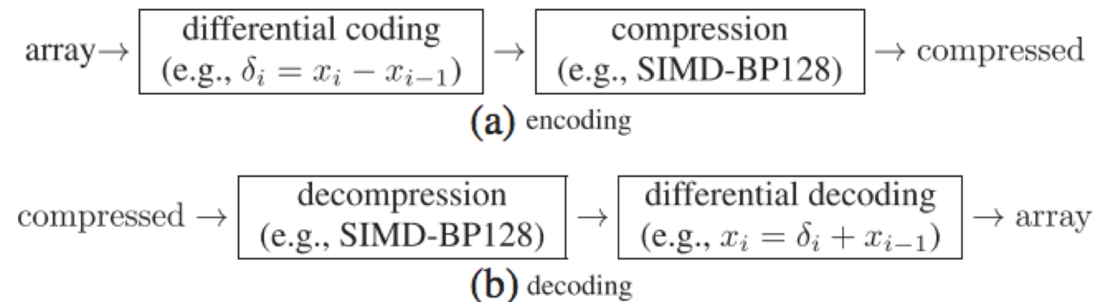
Decoding Billions of Integers Per Second Through Vectorization

D. LEMIRE AND L. BOYTSOV

PRESENTED BY ROSHNI SAHOO, SPRING 2019 6.886

Data Compression

- Data compression can improve query performance by reducing the main-memory bandwidth requirements.
- Data compression helps to load and keep more of the data into a faster storage.
- Applications to relational database systems, text retrieval systems.
- Performance Metrics: Decoding Speed, Encoding Speed, Compression Ratio



(a) encoding

(b) decoding

Differential Coding

- Instead of storing the original array of sorted integers, we keep only the difference between successive elements together with the initial value.
- The deltas are nonnegative integers that are typically much smaller than the original integers.
- Example:

Sorted Array (A)	150	153	154	156	160	161	162
Differential Coding Applied to A (B)	150	3	1	2	4	1	2

- Compute the prefix sums to reconstruct the original array
- Example: A[3] = B[0] + B[1] + B[2] + B[3] + B[4]

Variable Byte Encoding

- Variable byte rarely compresses data optimally, but it is reasonably efficient.
- Variable byte encodes data three times faster than most alternatives.
- Codes the data in units of bytes
  - Lower order 7 bits to store the data
  - 8th bit equal to 1 only for the last byte of a sequence that encodes an integer.

Example:

200 is written as 11001000 in binary notation.

Variable byte encoding would code it using 16 bits 10000001 01001000

Binary Packing

- Arrays are partitioned into blocks.
- The range of values in the blocks is first coded and all values in the block are written in reference to the range of values.
- Example: If the values in a block are in the range [1000, 1127] then they can be stored using 7 bits/int as offsets from the number 1000 in binary notation.
- Assumption that integers are small, so we only need to code a bit width b per block. Successive values are stored using b bits/int using fast bit-packing functions.

Patched coding

- Sometimes, binary packing compresses poorly.

Example:

Compress 1, 4, 255, 4, 3, 12, 4294967295 with binary packing → at least 32 bits/int required

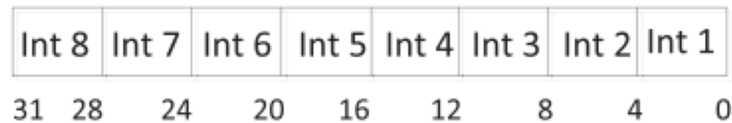
- PFOR: Instead, use a small bit width b for bit packing but store exceptions on a page (values greater than or equal to 2^b) in a separate location.

Replace the exception with an offset that is equal to the difference between the index of the next exception and the index of the current exception minus 1. Store exceptions in a linked list. Then, bit pack all the integers as usual.

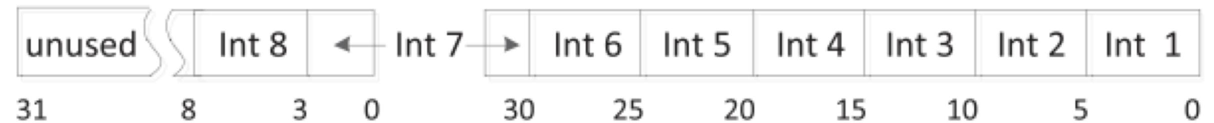
10	10	1	10	100110	10	1	11	10	100000	10	110100
10	10	1	10	100	10	1	11	10	1	10	…

Bit Packing

- Bit packing is a process of encoding small integers in $[0, 2^b)$
- Example: We can store 8 4-bit integers in a 32-bit word. If we want to store 8 5-bit integers, the packed representation uses two words.



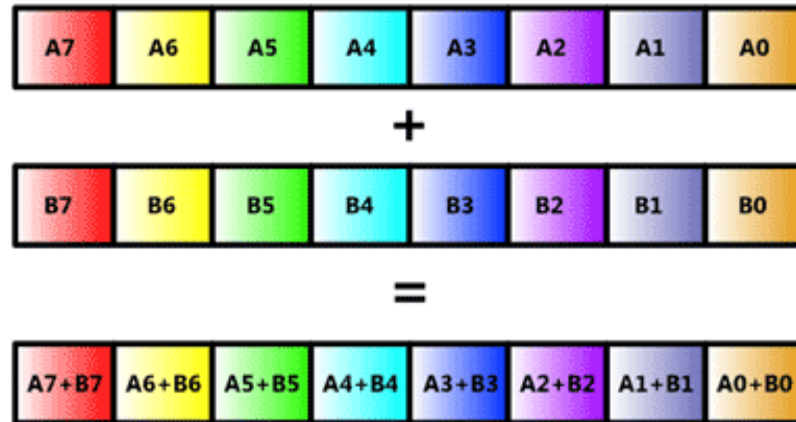
(a) 4-bit integers

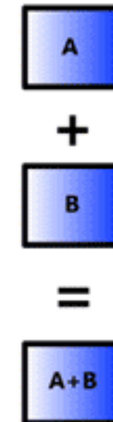


(b) 5-bit integers

- C and C++ support the concept of bit packing but the data may not be fully packed and their routines are not optimally fast.

Vectorization





Modern microprocessors often incorporate vector hardware to process data in a singleinstruction stream, multiple-data stream (SIMD) fashion.

Vector instructions operate in an element-wise fashion.

- All vector lanes operate in lock-step and use the same instruction and control signals.
- The i-th element of one vector register can only take part in operations with the i-th element of other vector registers.
- All lanes perform exactly the same operation on their respective elements of the vector.
- Some architectures support cross-lane operations, such as inserting or extracting subsets of vector elements, permuting (a.k.a., shuffling) the vector, scatter, or gather.

Related Work

- There are many techniques for integer compression
  - Earliest: Golomb coding, Rice coding, Elias gamma and delta coding
  - Variable byte encoding
  - Faster, more recent: Simple family, binary packing, patched coding

- Stepanov et al’s varint-G8IU reported that their SIMD-based varint G*IU algorithm outperformed classic variable byte coding by 300%. In addition, SIMD instructions allows one to improve the performance of decoding algorithms by more than 50%.

Stepanov et al’s varint-G8IU

- Generalizes variable byte into a family of byte-oriented encodings
- Each encoded byte contains bits from only one integer
- Groups of 8 bytes: each may store from 2 to 8 integer. A single byte descriptor indicates whether each byte is a continuation or end of an integer. When a descriptor bit is set to 0, the corresponding byte is the end of an integer.
- Integers packed with varint-G8IU can be efficiently decoded using the SSSE3 shuffle instruction: `pshufb`
- The operation takes two 16 element vectors of bytes. The first vector contains the bytes to be shuffled into an output vector and the second vector is a shuffle mask, which defines a hardcoded sequence of operations that copy bytes from the source to buffer or fill selected bytes with zeroes.

Contributions

- The authors focus on compression techniques for 32-bit integer sequences.
- SIMD-BP128*: a vectorized binary packing coding scheme that is nearly twice as fast as the previously fastest schemes on desktop processors
  - SIMD-BP128* relies on vectorized bit packing.
  - SIMD-BP128* is implemented with vectorized differential coding, as well.
- SIMD-FASTPFOR: a vectorized patched coding scheme that has a compression ratio within 10% of a state-of-the-art scheme while being two times faster during decoding.

Fast Differential Coding and Decoding

- Instead of just storing the initial value of the array, store the first four elements unmodified. From each remaining element with index i, subtract the element with the index $i-4$.
- We can compute the four differences by using a single SIMD operation.
- The original array $(x_1, x_2, x_3 \ \ldots)$ is converted into

$$(x_1, x_2, x_3, x_4, \delta_5 = x_5 - x_1, \delta_6 = x_6 - x_2, \ \delta_7 = x_7 - x_3, \delta_8 = x_8 - x_4 \ \ldots).$$

- To prevent memory bandwidth from becoming a bottleneck, differential coding and decoding is computed in place.
- To improve data locality and reduce cache misses, arrays containing more than 2^{16} integers are broken down into smaller arrays, and each array is decompressed independently.

Operations Required for Bit Packing

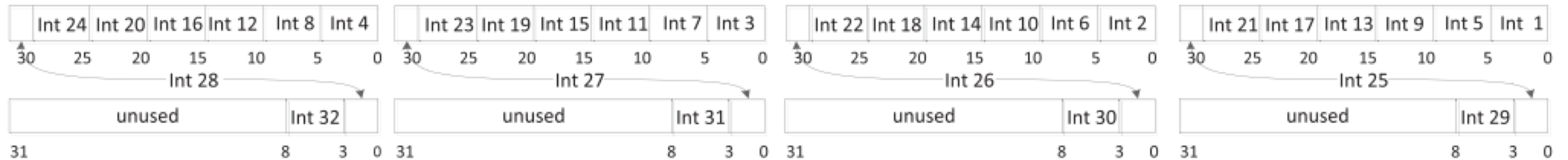
- Recall the examples of bit packing from earlier- decoding 8 4-bit integers from a 32 bit word and decoding 8 5-bit integers from two words.
- Each integer can be extracted four operations (shift, mask, store, and pointer increment).

```
void unpack4_8(const uint32_t* in,
          uint32_t* out) {
  *out++ = ((*in))         & 15;
  *out++ = ((*in) >> 4)    & 15;
  *out++ = ((*in) >> 8)    & 15;
  *out++ = ((*in) >> 12)   & 15;
  *out++ = ((*in) >> 16)   & 15;
  *out++ = ((*in) >> 20)   & 15;
  *out++ = ((*in) >> 24)   & 15;
  *out =   ((*in) >> 28);
}
```

```
void unpack5_8(const uint32_t* in,
          uint32_t* out) {
  *out++ = ((*in))         & 31;
  *out++ = ((*in) >> 5 )   & 31;
  *out++ = ((*in) >> 10)   & 31;
  *out++ = ((*in) >> 15)   & 31;
  *out++ = ((*in) >> 20)   & 31;
  *out++ = ((*in) >> 25)   & 31;
  *out   = ((*in) >> 30);
  ++in;
  *out++ |= ((*in) & 7) << 2;
  *out =    ((*in) >> 3) & 31;
}
```

Vectorized Bit Unpacking

- To improve the performance of the decoding step use vectorization. Suppose `in` and `out` are m-element vectors instead, so a single call to the unpack function decodes m x 8 integers.
- Example: Vectorized unpacking of 5-bit integers



Vectorized Bit Unpacking

```
void unpack5_8(const uint32_t* in,
              uint32_t* out) {
  *out++ = ((*in))        & 31;
  *out++ = ((*in) >> 5 ) & 31;
  *out++ = ((*in) >> 10) & 31;
  *out++ = ((*in) >> 15) & 31;
  *out++ = ((*in) >> 20) & 31;
  *out++ = ((*in) >> 25) & 31;
  *out   = ((*in) >> 30);
  ++in;
  *out++ |= ((*in) & 7) << 2;
  *out =    ((*in) >> 3) & 31;
}
```

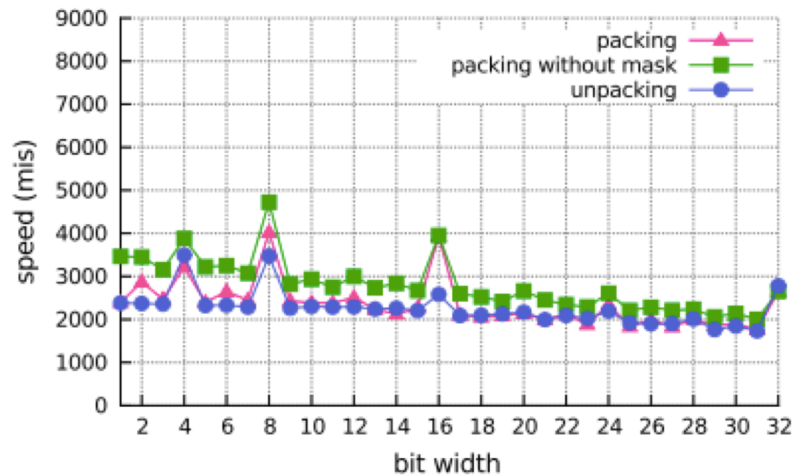
```
const static __m128i m7  =  _mm_set1_epi32(7U);
const static __m128i m31 =  _mm_set1_epi32(31U);

void SIMDunpack5_8(const __m128i* in,  __m128i* out) {
  __m128i i = _mm_load_si128(in);
  _mm_store_si128(out++, _mm_and_si128( i , m31));
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,5)  , m31));
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,10) , m31));
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,15) , m31));
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,20) , m31));
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,25) , m31));
  __m128i  o = _mm_srli_epi32(i,30);
  i = _mm_load_si128(++in);
  o = _mm_or_si128(o, _mm_slli_epi32(_mm_and_si128(i, m7), 2));
  _mm_store_si128(out++, o);
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,3)  , m31));
}
```

Replace scalar operations with respective SIMD instructions.

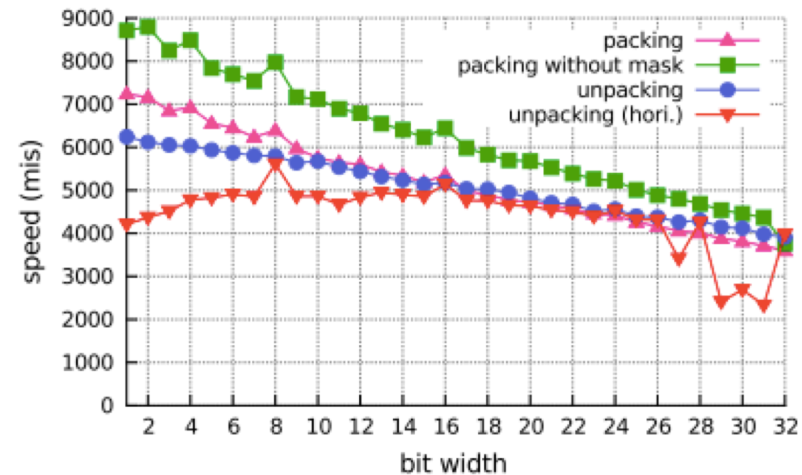
Bit Packing Experiments

Graphs illustrate the speed at which we can pack and unpack integers using blocks of 32 integers.

- Packing Without Mask – given a bit width b, indicates that integers all are in the range $[0, 2^b)$
- Packing – there are integers larger than or equal to 2^b



(a) Optimized but portable C++



(b) Vectorized with SSE2 instructions

SIMD-FASTPFOR – Achieving Fast and Efficient Compression

- Prior patched coding schemes – NewPFD (better compression, slower speed) and PFOR (speed, but worse compression)
- Instead of recording the exceptions per page as in PFOR, record them per block. Instead of bit-packing the exception locations, bit-pack the lower order *b* bits of the exceptions.
- Store each exception in an array corresponding to difference of the maximal bit width and *b*. Bit pack each array.
- Write metadata for each block to a temporary byte array containing the following information
  - The number of bits allocated per truncated integer
  - The maximal bit width
  - The number of exceptions
  - The locations of the exceptions

SIMD-FAST PFOR

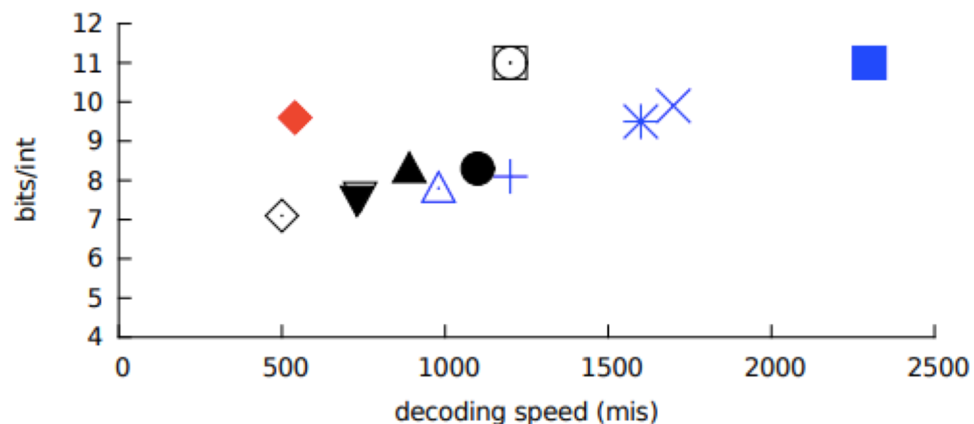
Data to be compressed:	...10, 10, 1, 10, 100110, 10, 1, 11, 10, 100000, 10, 110100, 10, 11, 11, 1...
Truncated data: (16 × 2 = 32 bits)	...10, 10, 01, 10, 10, 10, 01, 11, 10, 00, 10, 00, 10, 11, 11, 01 ...
Byte array: (6 × 8 = 48 bits)	...2, 6, 3, 4, 9, 11 ...
Exception data: (to be compressed)	...1001, 1000, 1101 ...

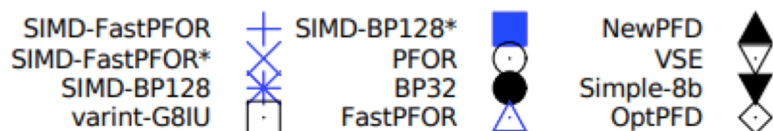
Experiments on Synthetic Datasets

- Datasets
  - ClusterData, Uniform model from which sets of distinct integers can be generated in the range $[0, 2^{29})$
  - Created short and long arrays.

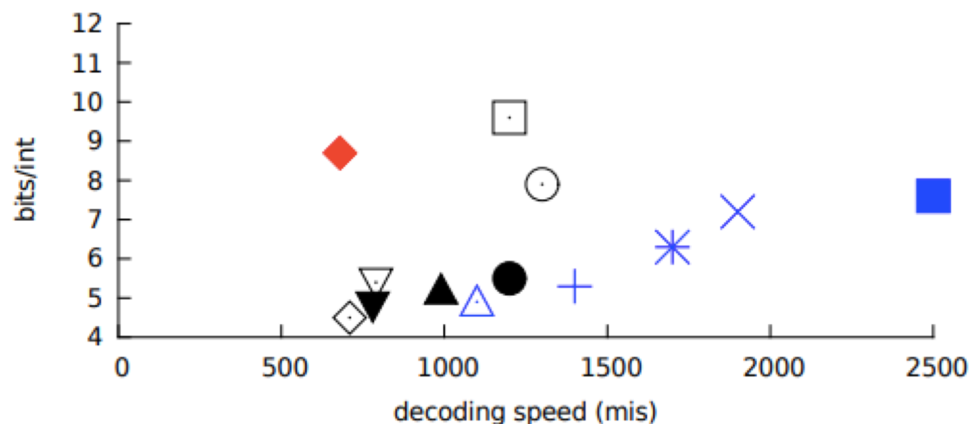
- Long arrays
  - best compression ratios are SIMD-FastPFOR, FastPFOR, SimplePFOR, Simple-8b, OptPFD
  - SIMD-FastPFOR dominates in decoding speed among these.
- Short arrays
  - SIMD-BP128 dominates in coding and decoding speed.
  - Relatively little difference in compression efficiency among the algorithms

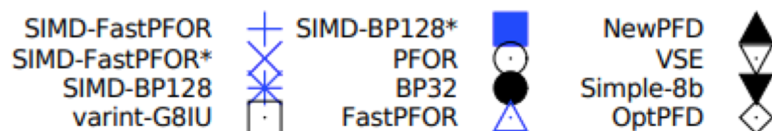
Experiments on Realistic Datasets





(a) ClueWeb09





(b) GOV2

Conclusion

- Vectorized bit-packing schemes are efficient; they encode/decode at speeds of 4000-8500 mis, so the computation of deltas and prefix sums may become a bottleneck.
- The differential coding and prefix sum computation can be made more performant using vectorization at the expense of poorer compression ratios.
- SIMD-BP128 outperforms PFOR on all three metrics- compression ratio, coding, and decoding speed.
- The performance of SIMD-FastPFOR indicates that patching is a fruitful strategy because it is faster than the SIMD-based varint-G8IU by at least 35% and provides a much better compression ratio.

Discussion

- Strengths
  - Each encoding technique was explained very well with examples and diagrams. I included many of diagrams from the paper in this presentation!
  - The paper explains the tradeoffs between different algorithms with both speed and compressive ratio and provides evaluations on all metrics.
- Weaknesses
  - In the paper, the authors first describe fast techniques for differential coding and bit unpacking without motivating that these are used in their main algorithm. Though these techniques are general, defining SIMD-BP128* prior to presenting these sections would have given the paper better structural clarity.