Metamodeling Lightweight Data Compression Algorithms and its Application Scenarios

Juliana Hildebrandt[1], Dirk Habich[1], Thomas Kühn[2], Patrick Damme[1], and Wolfgang Lehner[1]

[1] Technische Universität Dresden, Database Systems Group, Dresden, Germany
[2] Technische Universität Dresden, Software Technology Group, Dresden, Germany
{firstname.lastname}@tu-dresden.de[1] thomas.kuehn3@tu-dresden.de[2]

Abstract. Lossless lightweight data compression is a very important optimization technique in various application domains like database systems, information retrieval or machine learning. Despite this importance, currently, there exists no comprehensive and non-technical abstraction. To overcome this issue, we have developed a systematic approach using metamodeling that focuses on the non-technical concepts of these algorithms. In this paper, we describe COLLATE, the metamodel we developed, and show that each algorithm can be described as a model conforming with COLLATE. Furthermore, we use COLLATE to specify a compression algorithm language COALA, so that lightweight data compression algorithms can be specified and modified in a descriptive and abstract way. Additionally, we present an approach to transform such descriptive algorithms into executable code. As we are going to show, our abstract and non-technical approach offers several advantages.

1 Introduction

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

For the lossless compression of a sequence of integer values, a large corpus of lightweight algorithms has been developed [1, 3, 4, 9–13, ?][3]. In contrast to heavyweight algorithms, like arithmetic coding [14], Huffman [15], or Lempel Ziv [16], lightweight algorithms achieve comparable or even better compression rates [1, 3,

[3] Without claim of completeness.

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.
In: C. Cabanillas, S. España, S. Farshidi (eds.):
Proceedings of the ER Forum 2017 and the ER 2017 Demo track,
Valencia, Spain, November 6th-9th, 2017,
published at http://ceur-ws.org

4, 9–13, ?]. Moreover, the computational effort for (de)compression is lower than for heavyweight algorithms. To achieve these unique properties, each lightweight compression *algorithm* employs one or more basic compression *techniques* such as frame-of-reference [9, 11] or null suppression [1, 13], that allow the appropriate utilization of contextual knowledge like value distribution, sorting, or data locality. Every single algorithm is important and has its field of application [17]. This is true not only for current, but also for future algorithms.

Our Contributions and Outline. Despite this importance, there exists currently no comprehensive and non-technical abstraction for this domain. Therefore, the algorithms are only presented technically. However, this makes both the selection as well as the adaptation of algorithms to the respective context difficult. To overcome that, we have developed an approach using metamodeling that focuses on the non-technical concepts of lightweight data compression algorithms. Thus, our main contributions in this paper are:

1. We present our metamodeling approach with regard to separation of concerns by summarizing (i) the core results of our algorithm analysis and (ii) the derived metamodel called COLLATE in Section 2.
2. We apply COLLATE to develop a compression algorithm language COALA in Section 3. This allows us to specify and modify lightweight data compression algorithms in a descriptive and non-technical way. Additionally, we present an approach to transform such descriptive algorithms into executable C/C++ code.
3. We evaluate our language and execution approach using case studies in Section 4. In particular, we show (i) that we are able to describe existing as well as new algorithms with COALA and (ii) that our transformation approach produces correct executable code.

We close this paper with related work and a conclusion in Sections 5 and 6.

2 Metamodeling Lightweight Compression Algorithms

In this paper, we limit our discussion to lossless lightweight integer compression. But the described approach applies to decompression as well. Before we introduce COLLATE, we summarize the results of our algorithm analysis with regard to common non-functional aspects.

2.1 Properties of Data Compression Algorithms

To compress a finite sequence of integer values losslessly, all available compression algorithms use the following five basic techniques: frame-of-reference (FOR) [9, 11], delta coding (DELTA) [12, 13], dictionary compression (DICT) [1, 9], run-length encoding (RLE) [1, 13], and null suppression (NS) [1, 13]. FOR and DELTA represent each value as the difference to a certain given reference value (FOR) respectively to its predecessor value (DELTA). DICT replaces each value by its unique key given by a dictionary. The objective of these three well-known techniques is to represent the original data as a sequence of small integers,

which is then suited for actual compression using the NS technique. NS is the most well-studied kind of all lightweight compression techniques. Its basic idea is the omission of leading zeros in the bit representation of small integers. Finally, RLE tackles uninterrupted sequences of occurrences of the same value, so called runs. In its compressed format, each run is represented by its value and length. Thus, the compressed data is a sequence of such pairs.

Generally, these five techniques address different *data levels*. While FOR, DELTA, DICT, and RLE consider the *logical* data level, NS addresses the *physical* level of bits or bytes. This explains why lightweight data compression algorithms are always composed of one or more of these techniques. In the following, we denote the techniques from the logical level as preprocessing techniques for the physical compression with NS. These techniques can be further divided into two groups depending on how the input values are mapped to output values (often called codewords). FOR, DELTA, and DICT map each input value to exactly one integer as output value (*1:1 mapping*). The objective is to achieve smaller numbers that can be better compressed on the bit level. In RLE, not every input value is necessarily mapped to an encoded output value, because a successive sub-sequence of equal values is encoded in the output as a pair of run value and run length (*N:1 mapping*). The NS technique is either a *1:1* or an *N:1 mapping* depending on the implementation [18].

The genericity of these basic techniques is the foundation to tailor the algorithms to different data characteristics. The NS technique has been studied most extensively. There is a very large number of specific algorithms showing the diversity of the implementations for a single technique. The pure NS algorithms can be divided into the following classes [18]: (i) bit-aligned, (ii) byte-aligned, and (iii) word-aligned. While bit-aligned NS algorithms try to compress an integer using a minimal number of *bits*, byte-aligned NS algorithms compress an integer with a minimal number of *bytes* (1:1 mapping). The word-aligned NS algorithms encode as many integer values as possible into 32-bit or 64-bit words (N:1 mapping). The logical-level techniques have been usually investigated in connection with the NS technique [17].

Concluding, a lightweight data compression algorithm is always a combination of one or more of these basic techniques. The algorithms differ greatly in how they apply the individual techniques. This great variability results from the multitude of opportunities in which the input sequence of integers can be subdivided. For each subsequence, the algorithms analyze the contained values to optimize the application of the basic techniques.

2.2 COLLATE Metamodel

Thus, subdivision and specific application of the five basic techniques are important aspects of each lightweight data compression algorithm. For the specific application, the algorithms usually determine parameters for each subsequence and with the help of the calculated parameter values, the techniques are specialized for the subsequences.

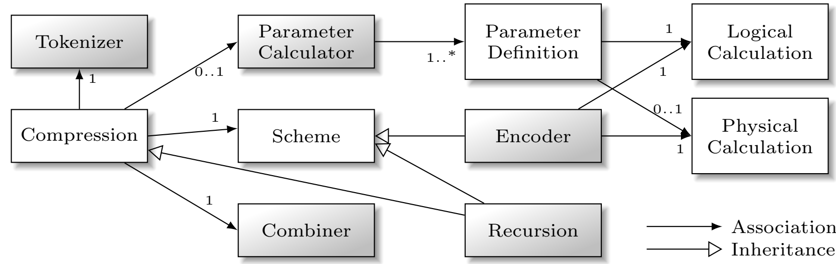


Fig. 1: CoLLATE Metamodel.

CoLLATE Concepts. To describe every single one of these lightweight compression algorithms, we identified five core concepts as highlighted in gray boxes in Fig. 1. Because compression algorithms are often characterized by a hierarchical data subdivision, we used the composite design pattern as foundation. Thus, each algorithm is a `Compression`, which is associated with one `Tokenizer`, zero to one `Parameter Calculator`, one `Encoder` or one further `Compression`, and one `Combiner`. The different concepts fulfill the following tasks:

`Recursion`: This concept is responsible for the hierarchical data subdivision and for applying the included concepts in the `Recursion` on each data subsequence.

`Tokenizer`: This concept is responsible for dividing an input sequence into finite subsequences or single values.

`Parameter Calculator`: The optional concept `Parameter Calculator` determines parameter values for finite subsequences or single values. The specification of the parameter values is done using parameter definitions. Parameters can be calculated on the logical as well as the physical level.

`Encoder`: The fourth concept determines the encoded form (codewords) for integer values to be compressed at bit level. The concrete encoding is specified using functions representing the basic techniques on the logical and physical data level.

`Combiner`: The `Combiner` arranges the encoded values (codewords) and the calculated parameters for the output representation.

Concept Properties. The concepts `Recursion` as well as `Combiner` should be clear enough. For the `Tokenizer` concept, we identified three classifying characteristics. The first one is *data dependency*. A *data independent* `Tokenizer` outputs a special number of values without considering the value itself, while a *data dependent* `Tokenizer` is used in case the decision about how many values to output is led by the knowledge of the concrete values. For example, a data dependent `Tokenizer` is necessary for the technique RLE, where the `Tokenizer` output depends on the values themselves. A second characteristic is *adaptivity*. A `Tokenizer` is adaptive if the calculation rule changes depending on previously read data. The third property is the *necessary input for decisions*. Most `Tokenizers` need only a finite prefix of a data sequence to decide how many values to output. The rest of the sequence is used as further input for the `Tokenizer` and processed in the same manner. Only those `Tokenizers` are able to process

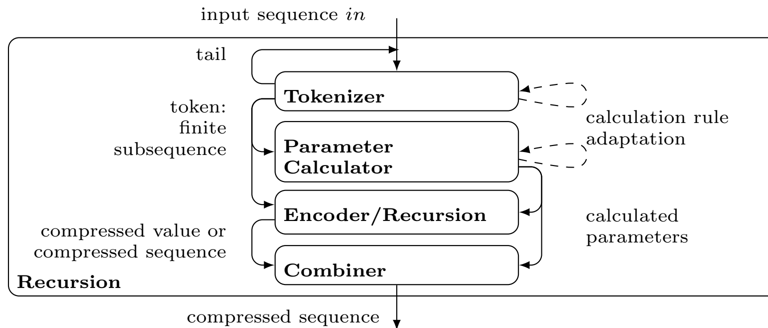


Fig. 2: Interaction and Data Flow of the COLLATE Metamodel.

data streams with potentially infinite data sequences. Moreover, there are `Tokenizers` requiring the whole (finite) input sequence to decide how to subdivide the sequence.

Parameters are often required for the compression. Therefore, we introduce the `Parameter Calculator` concept, which follows special rules (parameter definitions) for the calculation of several parameters. There are different kinds of parameter definitions. We often need single numbers like a common bit width for all values or mapping information for dictionary based encodings. We call a parameter definition *adaptive*, if the knowledge of a calculated parameter for one token (output of the `Tokenizer`) is needed for the calculation of parameters for further tokens at the same hierarchical level. For example, an adaptive parameter definition is necessary for DELTA. Calculated parameters have a logical representation for further calculations including encoding of values and optionally a representation at bit level, because on the one hand they are needed to calculate the encoding of values, on the other hand they mostly have to be stored additionally to allow the decoding.

The `Encoder` processes an atomic input, where the output of the `Parameter Calculator` and other parameters are additional inputs. The input is a token that cannot or should not be subdivided anymore. In practice the `Encoder` mostly obtains a single integer value to be mapped into a binary code (1:1 mapping techniques). An exception is RLE as N:1 mapping technique, where the `Parameter Calculator` maps a sequence of equal values to its run length and the `Encoder` maps the sequence to the special value. Equally to parameter definitions, the `Encoder` calculates a logical representation of its input value and a bit encoding.

2.3 Interaction of COLLATE Concepts

Besides the single concepts, we are also able to specify (i) the interactions of the concepts and (ii) the data flow through the concepts for lightweight data compression algorithms as illustrated in Fig. 2. The dashed lines highlight optional data flows. In general, this arrangement can be used as metaobject protocol for

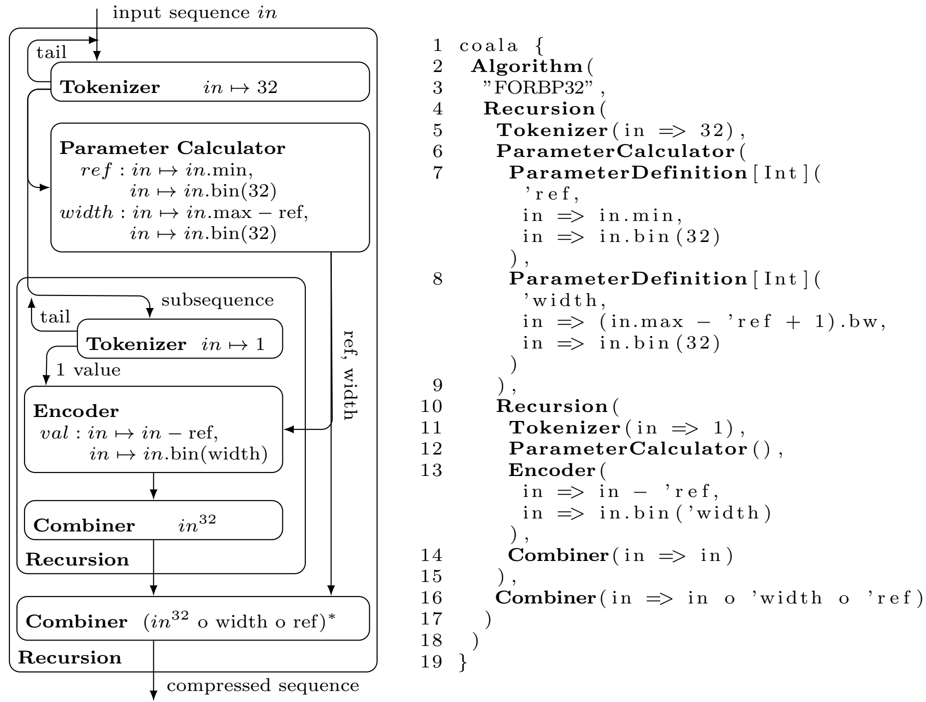


Fig. 3: Graphical and COALA Code for *FORBP32*.

lightweight compression algorithms and each algorithm conforms to this interaction of the specific concepts.

3 Application Scenarios

The goal of this section is to show different application scenarios of COLLATE. In detail, we apply COLLATE (i) to design and describe algorithms in an abstract way as *models*, (ii) to define a compression algorithm language COALA for an unambigious, declarative *non-technical specification* of algorithms, and (iii) to transform descriptive algorithms into *executable code*.

3.1 Algorithms as Models

Generally, each lightweight data compression algorithm can be modeled as an instance of the COLLATE metamodel. To show that, we use a self-constructed algorithm called *FORBP32* as a running example. This algorithm does not occur in recent literature. But it works well in the case when (i) values have a good locality (consecutive values in the same range), (ii) there are no single huge outliers and when (iii) the absolute size of values to compress does not matter. *FORBP32* combines the frame-of-reference (FOR) technique at the logical level to achieve

smaller numbers which are afterwards compressed using binary packing. Binary packing (BP) is a kind of null suppression, where a common bit width is used for the encoding of values within a finite subsequence [12]. Thus, in *FORBP32*, a common reference value and bit width are calculated for each subsequence of 32 integer values and stored with 32 bits each before the 32 encoded values follow.

We can model *FORBP32* by specifying all COLLATE concepts as depicted in Fig. 3. Like any algorithm model, *FORBP32* is a `Recursion`. It consists of a `Tokenizer` outputting 32 integer values independently in each subsequence, which is formalized by the function $(in \mapsto 32)$, a `Parameter Calculator`, where (i) the calculation rule for the reference value as the minimum of each subsequence, expressed by the mapping $(in \mapsto in.\text{min})$ and (ii) its binary encoding with 32 bits $(in \mapsto in.\text{bin}(32))$ are determined by a parameter definition. Also (i) the calculation rule for the bit width on the basis of the range of 32 values, expressed by $(in \mapsto in.\text{max} - \text{ref})$ and (ii) its encoding with 32 bits specify a further parameter definition. At the end, the `Combiner` arranges bit width, the reference value and the corresponding sequence of 32 encoded integer values. This is expressed by $((in^{32} \text{o width o ref})^{*})$, which means that the binary representations of the reference value and the bit width are followed by the encoding of 32 data values for every block of 32 values. Instead of an `Encoder`, we use a further `Recursion`, where the processing of a subsequence containing 32 values is done. This processing consists of a `Tokenizer` outputting single values and an `Encoder`, which (i) calculates the difference of a data value and the reference value and (ii) encodes the result with the number of bits determined before the `Combiner` constructs the sequence of 32 encoded compressed integer values.

At this point, we can draw three conclusions. First, the *FORBP32* algorithm can be expressed by the colluding concepts according to COLLATE. The same can be shown for a wide range of lightweight compression algorithms. Second, the models/algorithms can be easily adjusted. An example is the adjustment of the subdivision in 32, 64 or 128 values, which results in new algorithms. Third, new algorithms can be designed applying user-specified calculation rules for the logical as well the physical data level as shown by *FORBP32*.

3.2 Compression Algorithm Language COALA

To simplify the coding of lightweight data compression algorithms, we designed a domain-specific language called COALA allowing a declarative and non-technical specification. COALA is based on COLLATE and offers the opportunity to express instances of concepts as higher-order functions – concepts are functions taking functions as parameters. We decided to embed COALA into the host language Scala[4], because Scala is a high level programming language, which is object-oriented as well as functional and which treats higher-order functions are first class citizens. The advantage of this embedding is that COALA inherits many of the features and benefits of Scala [19]. Thus, we are able to use the higher-order function behavior of Scala to instantiate our five COLLATE concepts with an

[4] Scala Programming Language - https://www.scala-lang.org

adjustment by functions. This leads to a declarative description of algorithms as shown below.

The COALA code of the *FORBP32* example algorithm is depicted on the right side of Fig. 3, which is more or less a 1:1 mapping of the corresponding model. Each COLLATE concept is implemented using a domain-specific higher-order function of COALA. For example, the higher-order function `tokenizer` represents the COLLATE concept `Tokenizer`. In *FORBP32*, this `tokenizer` contains a mapping function from the input sequence `in` to the number 32 as function parameter. Here, we declare that the input sequence *has* to be subdivided in subsequences, where each subsequence contains 32 values. We do not specify *how* to do it, which is the difference between a declarative and an imperative programming approach. Another example is the definition of the parameter named `'ref`, which determines the minimum of a sequence and encodes it with 32 bits. It can be described by `ParameterDefinition[Int]('ref, in => in.min, in => in.bin(32))`, because the parameter definition is characterized by a name, a mapping from the input sequence to a logical (integer) value, which can be used for further calculations, and a mapping from the logical value to the binary encoding with 32 bits as indicated by the predefined function `in.bin(32)`.

To further support the declarative approach of COALA, the data flow and the interaction of the concepts are implicitly realized using the COLLATE metaobject protocol as illustrated in Fig. 2. Therefore, we omit an explicit data flow coding in COALA. All major and minor concepts shown in Fig. 3 can be fully expressed by higher-order functions. In summary, *FORBP32* is expressed by 10 concepts containing 10 simple lambda functions in a declarative way. The code is expressive and easy to understand, since it complies with the model instance. Furthermore, the code is without any technical detail about the *how*.

Concluding, with COLLATE we have a systematic approach to understand, to model and to adapt compression algorithms. The compression algorithm language COALA allows us to express these algorithms in a descriptive and non-technical way. Adaptations for reasons of optimization like the justification of the data subdivision, calculation of parameters, or the order of the compressed values can be implemented by changing single values without considering side effects. On the contrary, this could occur by changing native executable code.

3.3 Transformation of COALA Code to Executable Code

Our COALA approach allows designers of lightweight data compression algorithms to specify their algorithms in a novel declarative and domain-specific way. To execute such declarative algorithms, we transform COALA code into executable C/C++ code using generative programming [20]. Our generative programming approach for this transformation is based on the COLLATE concepts and is realized using Scala macros [21].

Generally, we defined a fixed transformation for each COLLATE concept depending on the properties as described in Section 2. We illustrate our approach using the running example algorithm *FORBP32*. The resulting C/C++ code is depicted in Fig. 4. Fundamentally, a COALA `Algorithm` is transformed into a

```
1   inline uint32_t min(uint32_t *in, int cntr, int step) {...}
2   inline uint32_t max(uint32_t *in, int cntr, int step) {...}
3   // algorithm
4   size_t forbp32(uint32_t *in, int length, uint32_t *out) {
5     // Initialization
6     uint32_t* initout = out;
7     int bitcounter = 0;
8     // Parameter Declaration Level 0
9     int ref;
10    int width;
11    // Recursion and Tokenizer Level 0
12    int step0 = 32;
13    for (int c0 = 0; c0 < length; c0 += step0) {
14      // Parameter Calculation Level 0
15      ref = min( in, c0, step0);
16      width = bw(max( in, c0, step0) - ref + 1);
17      // Encoding Parameters
18      *out |= ref << bitcounter;
19      out = ...
20      bitcounter = ...
21      ...
22      // Recursion and Tokanizer Level 1
23      int step1 = 1;
24      for (int c1 = c0; c1 < c0 + step0; c1 += step1) {
25        // Encoding Data
26        *out |= ( in[c1] - ref ) << bitcounter;
27        out = ...
28        bitcounter = ...
29      }
30    }
31    return out + ( bitcounter != 0 ) - initout;
32  }
```

Fig. 4: Generated C/C++ Code for *FORBP32*.

C/C++ generic function, which expects (i) an array with 32-bit integer values to be compressed, (ii) its length, and (iii) an output array as function parameters. Thus, we specify that we represent the input sequence of integers as an array on the execution level and the further processing has to be done on this array structure. Furthermore, each algorithm returns the word size of the compressed values. For our *FORBP32*, the generated C/C++ function starts with lines 4-7 and ends with line 31-32 in Fig. 4.

An algorithm is always a `Recursion` iterating over the input. For this reason, the `Recursion` concept is always mapped to a loop iterating over an array in our case. For the exact loop specification, the following `Tokenizer` has to be included. In *FORBP32*, we have two similar cases. Both cases are nested. Therefore, the nested recursions and tokenizers are also mapped to nested loops with fixed step widths of 32 and 1. This is done in lines 12-13 and 23-24 in Fig. 4.

The `Tokenizers` in *FORBP32* are very simple, but it is also possible to calculate the steps in `Tokenizers` as a function of parameters or data dependent. This is necessary for example for the RLE technique. The corresponding C/C++ transformation is illustrated in Fig. 5.

`Parameter Calculator:` For the algorithm *FORBP32*, the reference value and the bit width have to be calculated for each subsequence of 32 values. This is done for the logical value within the outer loop. The generated code is depicted in lines 15-16 in Fig. 4, where each aggregation function respectively other operation

```
1  inline int calc1( uint32_t *in, size_t length) {...}
2  ...
3  int step1 = calc1( in + 0, length - 0 );
4  for(int c0 = 0; c0 < length; c0 += step1) ) {
5    ...
6    step1 = calc1( in + c0, length - c0 );
7  }
```

Fig. 5: Realization of a Data Dependent (Adaptive) `Tokenizer`.

used in COALA is mapped to a predefined C/C++ function respectively a C/C++ operator, shown in lines 1-2.

The code for the `Encoder` is transformed in the same way. In addition to calculations at the logical level, calculations on the physical level are still needed. The physical calculations are always transformed into bit shift operations. The binary encoding of 32 values with the same bit width is done as depicted in lines 26-28, the binary encoding of the parameters in lines 18-21. The transformation of the `Combiner` results in the code position of the physical parameter and data encoding. In our case, first the parameters are written to the output in lines 18-21 and afterwards the data values.

To summarize, the transformation of COALA code to executable C/C++ is mainly driven by the transformation of the single COLLATE concepts. On the one hand, lightweight data compression algorithms iterate over the input integer sequence and subdivide it in subsequences or single values, which is specified by `Recursion` and `Tokenizer`. Thus, both concepts have to be considered jointly for the translation. On the other hand, the other concepts are more or less independent, only the order must be maintained. This is necessary to establish the necessary data flow as depicted in Fig. 2.

4 Case Studies

With the help of two use cases, we want to highlight the applicability and the advantages of our language as well as our execution approach. While the first use case considers an existing algorithm, the second use case evaluates our self-constructed algorithm *FORBP32*.

Existing Null Suppression Algorithm There is a large variety of lightweight data compression algorithms implementing only the null suppression (NS) technique. The pure NS algorithms can be divided into the following classes [18]: (i) bit-aligned, (ii) byte-aligned, and (iii) word-aligned. To show the applicability of our approach for these kinds of algorithms, we decided to use the existing byte-oriented algorithm *varint-SU* [4]. The algorithm suppresses leading zeros for each single integer by determining the smallest number of 7-bit units that are needed for binary representation without losing any information. To support decoding, the 7-bit number is stored as additional descriptor. This is done in a

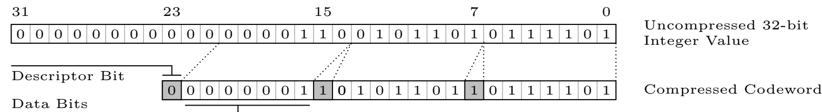
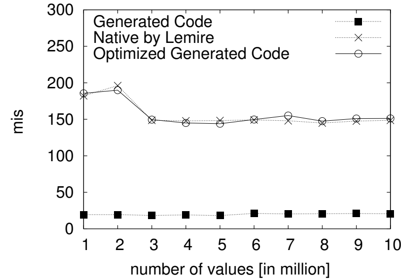


Fig. 6: Compressed Data Format for *varint-SU*.

```
1  coala {
2  Algorithm(
3   "VarintSU",
4   Recursion(
5    Tokenizer( in => 1),
6    ParameterCalculator(
7     ParameterDefinition[Int](
       'units,
       in => in.bw(7),
       in => in.unary011
      )
     ),
8    Encoder(
      in => in,
      in => in.bin( 'units * 7 )),
9    Combiner(
      in => zip(List((1, 'units), (7, in)))
     )
10  )
11 }
```

(a) CoALa Code



(b) Runtime Comparison

Fig. 7: Evaluation of *varint-SU* [12].

unary way by placing a single parameter bit at the high end of a 7-bit unit as depicted in Fig. 6.

The algorithm specification in CoALa is shown in Fig. 7(a). As we can see, the descriptive code consists of eleven code lines. The `Tokenizer` subdivides the input sequence into single integer values (indicated by `in => 1`). For each value, the `Parameter Calculator` determines the number of necessary 7-bit units using an appropriate function. The determined number is used in the subsequent `Encoder` to compress the integer value. The `Combiner` subdivides the bit level representation of the input value into several 7-bit units and arranges the bits of the encoded data and the descriptor.

Table 1: Comparison of the Code Sizes for *varint-SU*.

	Absolute measures			Relative measures	
	CoALa	Generated	Lemire	CoALa/Generated	CoALa/Lemire
Lines of Code	11	36	47	31%	23%
Number of Characters	220	953	908	23%	24%

For this algorithm, a native C/C++ implementation of Lemire exists [12]. Table 1 compares the code size of our descriptive CoALa specification with the generated C/C++ code and the native implementation. While our generated code has 36 lines of code, the native implementation has 47,because the native implementation uses mainly means of case distinctions. In contrast, our CoALa description has eleven lines of code, which is only 23% to 31% of the size of the executable code. The same applies to the number of characters as depicted in Fig. 1. Furthermore, we compared the compression result of our generated C/C++ code with the result of the Lemire implementation for different input sequences using our compression benchmark framework [22]. In all cases, the results are identical which is a clear indicator for the correctness of our generated executable code.

In addition to the comparison of the code sizes, we also compared the runtime behavior of our generated and the native variant. The runtime comparison was done on a standard server with an Intel Core i5 2.9GHz and 8GB main memory. Both compression algorithms are executed single-threaded and compiled with `g++` 4.2.1 using the optimization flag `-O3`. In the experiment, we varied the num-

ber of integer values to be compressed, thereby the compressed representations randomly vary between one and four 7-bit units per integer value. We report compression speeds in *million integers per second (mis)* as it is usually done in that domain [17]. As we can see in Fig. 7(b), our generated code performs poorly compared to the native implementation, whereas our generated code is approximately 7 times slower. The reason is that we are doing bit encoding with a loop, which is avoided in the native implementation by explicitly unrolling the loop. We included that unrolling concept in our transformation concept leading to a comparable compression speed as illustrated in Fig. 7(b). Some other optimization steps are case distinctions as well as avoidance of bit shift calculations, of copying, and of branching. However, these effects must be investigated more closely, which we will do in future work.

This analysis has been carried out for various NS algorithms, all of which have led to similar results. Therefore, we can draw the following conclusions:

1. Our COALA language can be used to specify NS algorithms in a descriptive and non-technical way, whereby those descriptions are much shorter than any imperative and technical implementation.
2. Our transformation approach produces correct executable codes, which are comparable to native implementations on a code size basis. The currently generated code is slower than native code, but we are convinced of being able to achieve comparable performance with automatical optimizations.

Novel and Self-Constructed Algorithms: As shown with *FORBP32*, we are able to specify and modify novel lightweight data compression algorithms with COALA. Furthermore, COALA algorithms can be executed with the help of our transformation approach. The COALA code of *FORBP32* is much smaller than any executable code, because the COALA code includes only non-technical details. This leads to more clearance and understandability.

5 Related Work

To the best of our knowledge, there exists only a high-level modularization concept for data compression in the domain of data transmission [23]. That scheme subdivides compression methods merely in (i) a data model adopting to data already read and (ii) a coder encoding the incoming data by means of the calculated data model. In that work, the area of conventional data compression was considered. Here, a multitude of approaches like arithmetic coding [14] Huffman [15] or Lempel-Ziv [16] exists. These classical algorithms achieve high compression rates, but the computational effort is high. Therefore, these techniques are usually denoted as heavyweight. The modularization concept of [23] is well-suited for this kind of heavyweight algorithms, but it does not adequately reflect different properties of the lightweight algorithms. For example, it does not support a sophisticated segmentation or a multi-level hierarchical data partitioning.

Additionally, a wide range of metamodels and domain-specific languages for various applications domains has been proposed. Hitherto, none of them has considered the field of lightweight data compression. Considering the fact that

lightweight data compression algorithms are an important optimization technique in different application domains, COLLATE and the domain-specific language COALA are more than important.

6 Conclusion

The continuous growth of data volumes is still a major challenge for efficient data processing. To tackle this challenge with regard to in-memory data processing, application domains like databases, information retrieval or machine learning follow a common approach: (i) encode values of each data attribute as sequence of integers using some kind of dictionary encoding and (ii) apply lossless lightweight data compression algorithms to each sequence of integer values. Aside from reducing the amount of data, a large variety of operations can be directly performed on compressed data.

In this paper, we described COLLATE, the metamodel we developed, for the large and evolving corpus of lightweight data compression algorithms and showed that each algorithm can be described as a model conforming to COLLATE. Furthermore, we used our metamodel to specify a novel compression algorithm language COALA, so that lightweight data compression algorithms can be expressed in a descriptive and non-technical way. Additionally, we presented an approach to transform such descriptive algorithms into executable code.

From our perspective, the following two areas of future application are available, where our presented approach can play an important role: *test platform* and *system integration*. In the test platform, we target the design of a specific development and test environment for lightweight data compression algorithms. In addition to the development of specific algorithms, the behavior of the algorithms must also be checked. To systematically test the algorithm behavior, we want to combine our COALA approach with our lightweight data compression benchmark system [22]. This enables us to establish a comprehensive *test platform* for this emerging optimization technology.

In addition, we want to extend our approach with the aim to integrate the large and evolving corpus of lightweight data compression algorithms in corresponding systems. For example, to the best of our knowledge, there is currently no in-memory database system available providing this corpus to a full extent. Therefore, the most challenging task is now to define an appropriate integration approach. With COALA, we have a language approach, but the transformation has to be database specific, which will be feasible.

References

1. D. J. Abadi, S. Madden, and M. Ferreira, “Integrating compression and execution in column-oriented database systems,” in *SIGMOD*, 2006.
2. P. A. Boncz, M. Zukowski, and N. Nes, “Monetdb/x100: Hyper-pipelining query execution,” in *CIDR*, 2005.

3. D. Arroyuelo, S. González, M. Oyarzún, and V. Sepulveda, “Document identifier reassignment and run-length-compressed inverted indexes for improved search performance,” in *SIGIR*, 2013.
4. A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi, “Simd-based decoding of posting lists,” in *CIKM*, 2011.
5. A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, “Compressed linear algebra for large-scale machine learning,” *PVLDB*, vol. 9, no. 12, 2016.
6. P. A. Boncz, M. L. Kersten, and S. Manegold, “Breaking the memory wall in monetdb,” *Commun. ACM*, vol. 51, no. 12, 2008.
7. T. Kissinger, T. Kiefer, B. Schlegel, D. Habich, D. Molka, and W. Lehner, “ERIS: A numa-aware in-memory storage engine for analytical workloads,” in *ADMS*, 2014.
8. C. Binnig, S. Hildenbrand, and F. Färber, “Dictionary-based order-preserving string compression for main memory column stores,” in *SIGMOD*, 2009, pp. 283–296.
9. M. Zukowski, S. Héman, N. Nes, and P. A. Boncz, “Super-scalar RAM-CPU cache compression,” in *ICDE*, 2006.
10. V. N. Anh and A. Moffat, “Index compression using 64-bit words,” *Softw., Pract. Exper.*, vol. 40, no. 2, 2010.
11. J. Goldstein, R. Ramakrishnan, and U. Shaft, “Compressing relations and indexes,” in *ICDE*, 1998.
12. D. Lemire and L. Boytsov, “Decoding billions of integers per second through vectorization,” *Softw., Pract. Exper.*, vol. 45, no. 1, pp. 1–29, 2015.
13. M. A. Roth and S. J. Van Horn, “Database compression,” *SIGMOD Rec.*, vol. 22, no. 3, 1993.
14. F. Silvestri and R. Venturini, “Vsencoding: Efficient coding and fast decoding of integer lists via dynamic programming,” in *CIKM*, 2010.
15. I. H. Witten, R. M. Neal, and J. G. Cleary, “Arithmetic coding for data compression,” *Commun. ACM*, vol. 30, no. 6, 1987.
16. D. A. Huffman, “A method for the construction of minimum-redundancy codes,” *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, 1952.
17. J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Trans. Inf. Theor.*, vol. 23, no. 3, 1977.
18. 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.
19. W. X. Zhao, X. Zhang, D. Lemire, D. Shan, J. Nie, H. Yan, and J. Wen, “A general simd-based approach to accelerating compression algorithms,” *ACM Trans. Inf. Syst.*, vol. 33, no. 3, 2015.
20. L. Tratt, “Domain specific language implementation via compile-time meta-programming,” *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 6, pp. 31:1–31:40, 2008.
21. K. Czarnecki and U. W. Eisenecker, *Generative programming - methods, tools and applications*. Addison-Wesley, 2000.
22. E. Burmako, M. Odersky, C. Vogt, S. Zeiger, and A. Moors, “Scala macros,” *URL http://scalamacros. org*, 2012.
23. P. Damme, D. Habich, and W. Lehner, “A benchmark framework for data compression techniques,” in *TPCTC*, 2015.
24. R. Williams, *Adaptive Data Compression*, ser. Kluwer international series in engineering and computer science: Communications and information theory. Springer US, 1991.