IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 15, 2019, accepted November 3, 2019, date of publication November 5, 2019, date of current version November 15, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2951746

Progressive Multicore RLNC Decoding With Online DAG Scheduling

SIMON WUNDERLICH[1], FRANK H. P. FITZEK[1,2], AND MARTIN REISSLEIN[2,3], (Fellow, IEEE)

[1]Deutsche Telekom Chair of Communication Networks, Technische Universität Dresden, 01062 Dresden, Germany
[2]Centre for Tactile Internet with Human-in-the-Loop (CeTI), Technische Universität Dresden, 01062 Dresden, Germany
[3]School of Electrical, Computer, and Energy Engineering, Arizona State University, Tempe, AZ 85287-5706, USA

Corresponding author: Martin Reisslein (reisslein@asu.edu)

This work was supported in part by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany's Excellence Strategy–EXC 2050/1–Project ID 390696704–Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop" (CeTI) of Technische Universität Dresden, and in part by the Federal Ministry of Education and Research of the Federal Republic of Germany (Förderkennzeichen, 5Gang) under Grant 16KIS0727.

ABSTRACT A complete generation of packets coded with Random Linear Network Coding (RLNC) can be quickly decoded on a multicore system by scheduling the involved matrix block operations in parallel with an offline (pre-recorded) directed acyclic graph (DAG). The waiting for a complete generation of packets can be avoided with progressive RLNC decoding that commences the decoding (and can decode some packets) before all packets in a generation have been received. This article develops and evaluates a novel progressive RLNC decoding strategy based on the principle of DAG scheduling of parallel matrix block operations. The novel strategy involves helper matrices for conducting the Gauss Jordan elimination based on rows of blocks of matrix elements. The matrix block computations are dynamically scheduled by an online DAG which permits branching, e.g., to skip unnecessary matrix block operations. The throughput and delay of the novel progressive RLNC decoding strategy are evaluated with experiments on two heterogeneous multicore processor boards. The novel progressive RLNC decoding achieves throughput levels on par with state-of-the-art non-progressive (full-generation) RLNC decoding and achieves three times higher throughput than the fastest (highest-throughput) known progressive RLNC decoder for small generation sizes and short data packets. Also, our progressive RLNC decoding greatly reduces receiver delays for moderate to large generation sizes; the delay reductions are particularly pronounced when a low-delay RLNC version is employed (e.g., reduction to one tenth of the non-progressive decoding delay for a generation size of 256 packets).

INDEX TERMS Directed acyclic graph (DAG), helper matrix, heterogeneous multicore architecture, online scheduling, parallel computing, random linear network coding (RLNC).

I. INTRODUCTION

Random linear network coding (RLNC) can significantly enhance the communication over unreliable complex networks, such as body area networks [1], caching networks [2]–[4], cellular networks [5], the Internet of Things (IoT) [6]–[8], radio access networks [9], vehicular networks [10], wireless sensor networks [11], and general wireless networks [12]–[16]. One main challenge of RLNC based communication is that the decoding in receiver nodes involves computationally highly demanding matrix multiplication and matrix inversion [17], [18]. Recently, the directed acyclic graph (DAG) scheduling of parallel matrix block operations from the field of high-performance computing [19], [20] has been adapted for high-throughput RLNC encoding and decoding of a complete generation of source symbols (source data packets) [21]. However, the decoding of a complete generation introduces long delays at the receiving nodes since at least a full generation worth of packets needs to be received before the decoding can commence [22].

Progressive RLNC decoding reduces the delays at the receiving nodes by commencing the decoding computations when only a single or a few packets (less than the full generation) have been received [23]. Progressive decoding becomes especially important with the emergence of low-delay RLNC coding schemes that permit the completion of the decoding

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Awais Javed.

of some packets (and their release to the higher protocol layers) before a complete generation worth of packets has been received [24]–[26]. To the best of our knowledge, the highly efficient DAG scheduling of parallel matrix block operations has not yet been studied in the context of progressive RLNC decoding.

In this article, we introduce a progressive RLNC decoding strategy based on the principle of DAG scheduling of parallel matrix block operations. Whereas generation-based (non-progressive) RLNC decoding can invert and multiply matrices, and thus perform the Gauss Jordan elimination, with the lower-upper (LU) factorization, progressive decoding cannot employ LU factorization (as LU factorization requires the full matrix). Thus, we develop a novel helper matrix based technique for performing the Gauss Jordan elimination over rows of blocks of matrix elements. Non-progressive RLNC decoding can proceed according to a statically (offline) configured DAG schedule that can be pre-recorded ahead of the actual computation execution. In contrast, our progressive RLNC decoding requires dynamic (online) scheduling decisions to work efficiently (particularly, efficient backward substitution requires frequent online branching decisions, see Section III-B2).

This article is structured as follows. Section II presents background on network coding and the different types of RLNC decoding, as well as related work on efficient computing strategies for network coding. Section III introduces the novel progressive RLNC decoding strategy with helper matrices for the Gauss Jordan elimination and the online DAG scheduling on multiple parallel processor cores. In addition, Section III-D introduces a stripe optimization and a full rows optimization that jointly process multiple matrix blocks in a given row. Section IV presents the evaluation of the novel progressive RLNC decoding strategy through measurements with two heterogeneous multicore processor boards. Section V concludes this article.

II. BACKGROUND AND RELATED WORK

A. NETWORK CODING BASICS

The sender partitions the source data into source symbols of size m [in units of words, whereby for the typically considered GF(2^8), 1 word = 8 bit = 1 Byte]. Then, g consecutive source symbols form a generation, which can be represented by a data matrix A with g rows and m columns. RLNC encoding combines the source symbols in a generation linearly in a Galois Field GF(2^p) [27], which can be represented as the matrix multiplication of a random coding coefficient matrix C_{sender} [with g rows (or $g + r$ rows when r redundant coded packets are to be generated) and g columns] with the data matrix A to form the coded symbol (data) matrix $D_{\text{sender}} = CA$ (with g rows and m columns). A coded packet consists of a coding coefficient row of matrix C_{sender} and the corresponding row of the coded data matrix D_{sender}.

The receiver accumulates received coding vectors in a coefficient matrix C and the coded symbols in a data matrix D (which we write without subscripts as computations with these two matrices are our focus and we want to avoid notational clutter). The receiver matrices C and D may have different row order than the sender matrices C_{sender} and D_{sender}. Without loss of generality of the RLNC computing methodology, we neglect errors or erasures during the network transmission and linear dependencies of the coding coefficient rows; accordingly, we consider $r = 0$ redundant coded packets. (We briefly outline the processing of linearly dependent coding coefficient rows at the end of Section III-B1.) The receiver decodes the data symbols by computing $A = C^{-1}D$. The receiver can completely decode all g data symbols if it has received g coded packets with linearly independent coding coefficient rows.

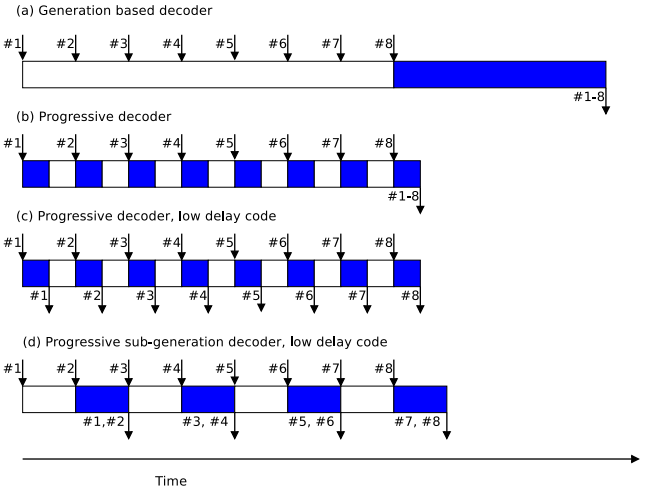


FIGURE 1. Coded packet arrival and decoded packet release for (a) non-progressive generation-based decoder with generation size $g = 8$, (b) progressive decoder (processing individual packets) and generation-based RLNC coding, (c) progressive decoder (processing individual packets) with low-delay RLNC coding, and (d, our focus) progressive sub-generation decoder with low-delay RLNC coding with a block fill level $p = 2$ (and matrix block size $b \geq 2$). Blue sections represent time spent on decoding computations, while white sections represent idle time (or other work).

Following high-performance computing strategies [28]–[30], efficient kernel (base) operations for various common operations, such as multiplication and inversion, on matrix blocks in GF(2^8) have recently been developed [21], [31]–[33]. We employ these kernel operations, which exploit the single instruction multiple data (SIMD) instructions that are commonly available on IoT and wireless node processor boards [34].

B. BACKGROUND: NON-PROGRESSIVE AND PROGRESSIVE RLNC DECODERS

RLNC decoders can be categorized into non-progressive and progressive decoders. A non-progressive decoder expects that all information is fully available before starting to decode. For instance, in generation-based RLNC coding [35], a full generation of coded packets (symbols) is collected before the non-progressive decoding process starts, as illustrated in Fig. 1(a). Having all data available allows the decoder to employ matrix inversion and matrix multiplication algorithms for full matrices. Thus, matrix inversion algorithms other than

Gauss-Jordan elimination, e.g., LU inversion [21], can be employed. Moreover, cache-friendly or parallelization-friendly algorithms can be utilized when the encoding data for the full generation is available.

On the other hand, a progressive decoder does not need to wait for all data to arrive. Rather, a progressive decoder can partially decode the data which has already been received. Individual newly received coded data packets containing coded data and coding coefficients can be fed into the progressive decoder as they arrive. The decoder can then perform operations based on this new information before it stalls to wait for more information, as illustrated in Fig. 1(b). In particular, the illustration in Fig. 1(b) considers a conventional full-vector RLNC code; thus, decoded packets can only be released after the last decoding computation for the generation has been completed. A progressive decoder can exploit low-delay codes, such as sliding window codes [36]–[38] or systematic generation based codes [39] (which may have interspersed redundancy [26]), by releasing any fully decoded information to the upper layers before all coded packets for a generation have been received. Fig. 1(c) illustrates the operation of a progressive decoder with a low-delay code, where decoded packets can be released after each respective computation process. (We assume RLNC coding without losses and redundancy in Fig. 1(c), however, the general principle of progressive decoding applies in those cases as well.) Progressive decoders are useful in delay-sensitive applications, such as live streaming or conference applications.

A hybrid scheme is to perform sub-generation based progressive decoding [22], which processes multiple coded packets at once, e.g., two coded packets at once as illustrated in Fig. 1(d). Progressive sub-generation decoding generally processes more than a single coded packet, but less than the normal generation size so as to combine the strengths of non-progressive and progressive decoders: Using efficient algorithms while decreasing the decoding delay.

C. RELATED WORK ON NETWORK CODING COMPUTATIONS

Some research studies have sought to mitigate the computational complexities of network coding by considering small Galois fields [40]–[43] or novel forms of network coding [44]. We consider the conventional RLNC over large fields, e.g., GF(2^8), which have negligible linear dependencies of the coding coefficient rows. The computation of the GF(2^8) RLNC can be efficiently sped up on servers with large numbers of Graphics Processing Units (GPUs) [22], [45]–[47]. However, many ubiquitous computing nodes, e.g., smartphones and IoT nodes, have only few GPUs [48]; hence, copying to the GPU threads does not amortize [33]. We consider heterogeneous multicore processors [49] that are common on smartphones and IoT nodes [50] and therefore do not specifically optimize for GPU processing. We note for completeness that a custom very large scale integration (VLSI) design for network coding has been studied in [51].

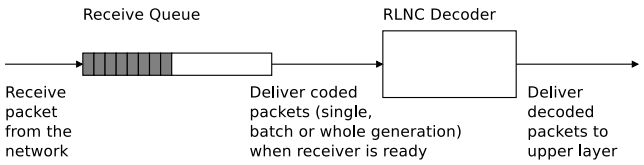


FIGURE 2. Illustration of receive buffer integration with decoder.

TABLE 1. Summary of main notations and parameters.

Notation		Description
Data size	m	represents the symbol size, i.e., amount of source data in a packet [in words, 1 word = 8 bit = 1 byte in GF(2^8)].
Generation size	g	is the maximum number of symbols that are combined to form a coded symbol.
Coefficient matrix	C	contains the coefficients used to code the symbols, has g rows and g columns.
Data matrix	D	contains the coded symbols, has g rows and m columns.
New coeff. matrix	C'	contains the coding coefficients of recently arrived packets, has b rows and g columns.
New data matrix	D'	contains the coded symbols of recently arrived packets, has b rows and m columns.
Threads	t	are the number of parallel processing threads used by the decoder.
Block size	b	size of block matrices ($b \times b$) used for matrix computations, whereby g/b is an integer. Also, b is the default number of packets processed in an iteration by a progressive sub-generation decoder
Block fill level	p	specifies how many packets should be collected before a decoding iteration starts, $1 \leq p \leq b$
Stripe size	s	is the number of columns in the stripe of blocks of data matrix D, s is an integer multiple of b and $s \leq m$.

Computational strategies for GF(2^8) network coding on general-purpose multicore CPUs have mainly focused on judiciously partitioning the coefficient and data matrices to facilitate parallel processing [52]–[55]. These partitioning strategies have greatly sped up the network coding computations (and reduced the energy consumption [56], [57]). Some additional speed up can be achieved by scheduling the matrix block operations according to the dependency structure of the computations in a DAG [21]. A key drawback of the DAG approach in [21] is that it is limited to non-progressive decoding of a full generation of coded packets; whereas, most of the partitioning approaches [52]–[55] are suitable for progressive RLNC decoding. The present study seeks to bring the benefits of DAG scheduling to progressive RLNC decoding.

III. ONLINE DAG RLNC DECODING

A. SETTING AND NOTATION

We assume that incoming coded packets are stored in an intermediate receive buffer (queue) upon reception, as shown in Fig. 2. When the decoder is idle or finished with a previous batch of packets, it can draw packets from the receive buffer and decode them. A classic progressive decoder would draw one coded packet at a time, while a generation-based decoder would draw a full generation worth of coded packets.

Our overall decoding approach follows the principles of progressive sub-generation decoding, as for instance used in the hybrid decoder in [22]: Incoming coded packets, each including coding coefficients and coded data, are stored in an intermediate receive buffer. When p, $p \leq b$, packets have been stored in the intermediate buffer and the decoder is available, then an iteration of the decoding process starts. Since the matrix computations are designed for blocks of size $b \times b$ elements, the remaining ($b - p$ bottom rows of a given block of) coefficients and coded data in the intermediate buffer are padded with zeros before the coefficients and data are released to the decoder. Note that it is generally not advantageous to start progressive decoding with multiple complete rows of blocks (e.g., with $2b$ received packets) since the matrix computations operate on one block row at a time. We also note that under special circumstances e.g., when a decoding timeout has expired, a progressive decoder may start the decoding with fewer than p received packets.

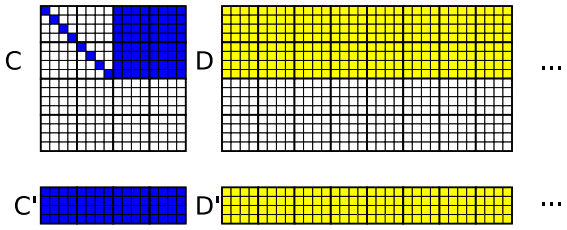
Figure 3(a) illustrates an intermediate state of a $g \times g = 16 \times 16$ coefficient matrix C and the corresponding data matrix D at the decoder, before applying the next batch of $p = 4$ packets. The first (top most) eight packets have been partially decoded: The diagonal elements (1, 1) through (8, 8) as well as the columns 9 to 16 in the coefficient matrix C are set, the white spaces represent zeros. Matrices C' and D' contain the coefficients and coded symbols of the next batch of $p = 4$ packets which have just arrived. After the next iteration of the blocked Gauss Jordan elimination, assuming no linear dependencies in the coefficients, these new packets will be partially decoded, extending the diagonal of ones in matrix C to elements (9, 9) to (12, 12) and filling in rows 9 to 12 of the data matrix D.

One iteration of the blocked Gauss Jordan elimination [58], [59] consists of the usual phases of forward elimination, backward substitution, and swapping rows. Instead of working on single matrix elements (as e.g., in [60]), we apply the operations on square submatrix blocks of size $b \times b$, or multiples of those blocks. In the following section, we describe each phase of the blocked Gauss Jordan elimination in detail. We describe the elimination process as an iterative program; however, the actual implementation will run parallelized by scheduling tasks according to the data dependencies between the operations, as explained in Section III-C.

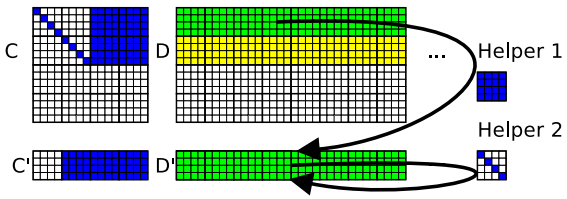
B. BLOCKED GAUSS JORDAN ELIMINATION WITH HELPER MATRICES

1) FORWARD ELIMINATION

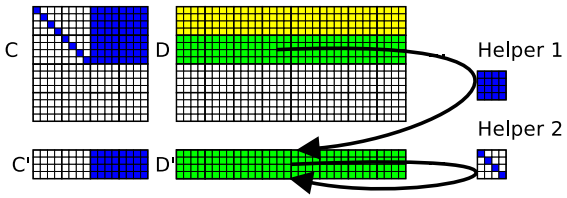
Forward elimination is the first phase of processing the intermediate buffer. Each $b \times b$ block of the new coefficient matrix C' is processed with the respective pivot block of the old coefficient matrix C to eliminate elements at positions where pivots had already been found. These operations are then applied to the rest of the coefficient matrix C' and the data matrix D'. We describe the process next and illustrate the behaviors in Fig. 3.



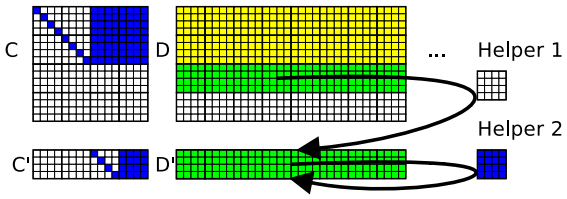
(a) Step 0 (start): Top-most eight rows in C and D are partially decoded; next batch of $p = 4$ arrived packets in C' and D'.



(b) Step 1: Pivots in first block row (top $b = 4$ lines) of C are processed to eliminate left-most (first) block of C'.



(c) Step 2: Pivots in second block row of C are processed to eliminate second block of C'.



(d) Step 3: Gaussian elimination gives new pivots in third block of C' (C did not yet have pivots in columns 9–12).

FIGURE 3. Illustration of steps of the forward elimination phase for newly received coded packets in new coefficient matrix C' and new data matrix D' given the current coefficient matrix C and data matrix D with $b \times b = 4 \times 4$ blocks. Helper matrix 1 records operations on blocks of C and D for replication on C' and D'. Helper matrix 2 records operations on blocks of C' and D' for replication on remaining blocks of C' and D'.

One forward elimination step works similar to the conventional Gauss Jordan algorithm with pivots: Pivots are expected on the main diagonal. The process checks the main diagonal from top left to bottom right. If a pivot element i in position (i, i) of the coefficient matrix C is one, the new coding coefficients in matrix C' are checked row by row for elements in column i. If the element in a row r in column i in matrix C' is non-zero, then the pivot row i of the coefficient matrix C is multiplied by the value of the element in row r in column i in matrix C' and subtracted element by element from row r of C'. However, if the element in the pivot position (i, i) in matrix C is zero, then the new coefficient rows in matrix C' will be checked for a coefficient row with a non-zero element

in column i. If such a row r with a non-zero element in column i is found in matrix C', then row r will be scaled (multiplied with the inverse) of the element of row r in column i, and marked to be swapped afterwards into matrix C. In that way, the elements of the new coefficient matrix C' on existing pivot columns of the coefficient matrix C will be eliminated, and new pivots will be found if they did not previously exist.

Note that all row operations (multiply-subtract as well as scaling and row swapping) can be described as linear operations using matrices. We use this fact to create two helper matrices during the forward elimination. These helper matrices are used to replicate the respective row operations on the remainder of the coefficient matrix C' and the data matrix D'.

Fig. 3 illustrates this behavior: Initially, in step 0 (Fig. 3(a)), the first two block rows (rows 1–8) of C are partially decoded, i.e., pivots have been found in the rows 1–8. A new set of packets has been received, forming matrices C' and D'.

As a first step (Fig. 3(b)), the existing pivots in positions $(1, 1)$ to $(4, 4)$ of C are processed to eliminate the elements in the first (left-most) block of the new coefficient matrix C'. During this first step, the helper matrices record the performed matrix operations; the exact same linear operations must be replicated on the remaining rows. Helper matrix 1 records the linear operations that are performed from the blocks of C and D for replication in C' and D', respectively. Helper matrix 2 records the linear operations that are performed from C' and D' for replication on the remaining blocks of C' and D', respectively. In the first step, only pivot elements from C are used to clear elements in C', as recorded in helper matrix 1 (while helper matrix 2 is an identity matrix that will not change C' and D' when applied).

In step 2 (Fig. 3(c)), the existing pivots in positions $(5, 5)$ to $(8, 8)$ of C are applied to the second block of C'. Again, helper matrix 1 records the operations that are applied to the remainder of C and all blocks of the data matrix D for replication in C' and D', respectively. Helper matrix 2 remains again an identity matrix.

In step 3 (Fig. 3(d)), C does not have any pivots on positions $(9, 9)$ to $(12, 12)$. Therefore, no rows of C are applied on C' and the helper matrix 1 remains a zero matrix. Within the third block of the new coefficients in C', Gaussian elimination is performed, resulting in pivots in the third block of C', the corresponding row operations are recorded in helper matrix 2. Those rows of C' and their target positions in C are also remembered for swapping in a later phase.

After each pivot processing part in a step of the forward elimination phase, simple matrix multiplication is used to apply the row operations on the remaining blocks of the coefficient matrix C' and the data matrix D' using the helper matrices 1 and 2. As we have seen during the example, there are special cases: When the helper matrix 1 is a zero matrix or the helper matrix 2 is the identity matrix, then the matrix multiplication does not change the matrices C' and D'. These cases can therefore be optimized: we include a flag in the matrix data structure to indicate an identity or zero matrix. If the flag is set, then the matrix multiplication is skipped, which can save up to half of the matrix multiplication operations.

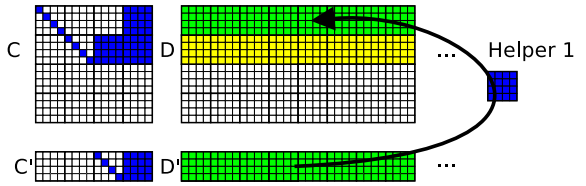
Newly received packets with coding coefficient rows that are not linearly independent of the coding coefficient rows of previously received packets undergo the regular forward elimination processing. However, during the forward elimination phase those packets will be "zeroed out", because those linear dependent coding coefficient rows will not result in pivots. Each such linearly dependent coding coefficient row will result in a row of zeros in the new coding coefficient matrix C'. Hence, the following backward substitution and row swapping will be skipped because they are no longer necessary. This online adaption of the processing of linearly dependent coding coefficient vectors is an advantage compared to the offline DAG approach where linear dependent coding coefficient rows cannot be detected until the entire decoding operation is completed and we notice that the coding coefficient matrix was not transformed into an identity matrix.

2) BACKWARD SUBSTITUTION

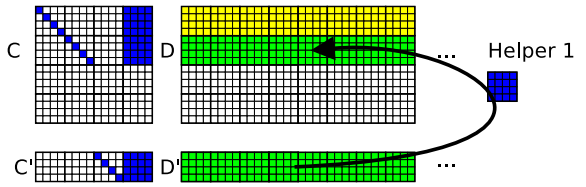
Backward substitution is used to eliminate elements from the blocks above a newly found pivot, so that more positions are filled with zeros. Similar to the conventional Gauss Jordan algorithm, we sweep matrix C' from the last column (column g) to the first column and check for each column whether matrix C' has a new (non-zero) pivot element. The presence of new pivot elements has been marked in the forward elimination phase as we remembered those rows for swapping. For each block in matrix C' with new pivots, we apply the backward substitution into the blocks of matrix C above the pivot block. Fig. 4 illustrates the backward substitution steps of the third block of matrix C' into the first (step 1, Fig. 4(a)) and second (step 2, Fig. 4(b)) block row of matrix C. For each non-zero element in the block row of matrix C, the pivot row from C' will be scaled (multiplied with) the non-zero element and then subtracted element by element in both matrix C and matrix D.

As in forward elimination, the row operations (scaling, subtracting) are recorded in a helper matrix, which is possible due to the linear nature of the operations. The helper matrix is then used to apply the same operations on the remaining blocks of matrices C and D.

We note that the backward substitution involves frequent online branching decisions and thus necessitates an online execution approach for efficient implementation. In particular, we only need to process block rows where a new packet has been swapped in, and can skip the remaining block rows. A computation execution according to a schedule that has been pre-recorded offline (as employed in the offline DAG approach [21]) would have to schedule tasks for the entire matrices since some rows could have been swapped "up". Possibly, a task could be quit early if it notices that nothing



(a) Step 1: Substitute third block of C' into first block row of C to eliminate third block of first block row of C and update right-most block of first block row of C and first block row of D.



(b) Step 2: Substitute third block of C' into second block row of C to eliminate third block of second block row of C and update right-most block of second block row of C and second block row of D.

FIGURE 4. Illustration of steps of backward substitution phase: New pivots found in step 3 of the forward elimination in C' (Fig. 3(d)) are backward substituted into C and D. The helper matrix records the operations on the first applicable block of C [e.g., block with elements (9,1) to (12,4), i.e., the third block from the left in the top-most block row for the first row, i.e., Step 1; Step 2 uses block (9,5 to 12,8)] for replication on the remaining blocks of C and D.

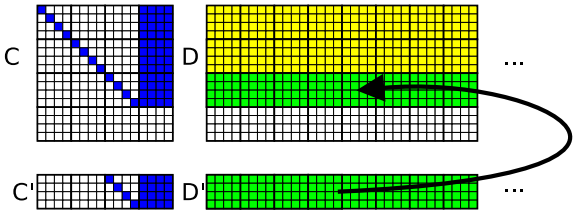


FIGURE 5. Illustration of row swapping phase: The new pivot rows found in step 3 of the forward elimination in C' and D' (Fig. 3(d)) are swapped into the third block row of C and D.

has been swapped in the current block row. Nevertheless, an offline execution would still incur a massive overhead for creating all these useless tasks.

3) ROW SWAPPING

After the backward substitution is complete, the new pivot rows from matrices C' and D' are swapped into their respective positions in matrices C and D. Unlike the conventional Gauss Jordan algorithm, we perform this row swapping phase after completing the backward substitution in order to avoid disturbing the linear application of the helper matrices.

Figure 5 illustrates how the pivot rows of matrices C' and D' are swapped into their final positions into the third block row of matrices C and D. In our example, all pivots have been placed in the same block, but in practice it can happen that pivots are found in non-consecutive positions and can therefore be placed in different blocks.

C. PARALLELIZATION WITH ONLINE DAG OF DATA DEPENDENCIES

1) OVERVIEW OF DATA DEPENDENCIES

The previous section described a blocked variant of the Gauss Jordan algorithm with helper matrices to implement a progressive RLNC decoder. The blocked Gauss Jordan algorithm with the helper matrices exhibits many opportunities for parallelization: In each of the three phases, the block-by-block matrix multiplications according to the helper matrices as well as the swap operations are independent from each other and can therefore be performed in parallel. For example, the backward substitution steps for one column of C' and the related scaling and subtraction operations in C and D can be simultaneously executed for the different block rows of C and D, e.g., the operations illustrated in Fig. 4(a) and (b) can be executed simultaneously. However, there are not only parallelization opportunities among the operations within one phase, but operations from different phases can potentially run in parallel. For example, the matrix multiplication tasks from the forward elimination phase can still be running while the tasks from the following backward substitution phase are already executed. In particular, while the matrix multiplications with helper matrices from the forward elimination phase are still applied on D' (see Fig. 3(d)), the backward substitution steps on C and C' (see Fig. 4(a) and (b)) can already be executed by different computing threads since there are no data dependencies.

Generally, we can allow any operation to be executed in parallel with other operations if there are no data dependencies prohibiting the parallel execution. Data dependencies can be formulated using the following four rules:

1) Read after Write: a block of data must be written before the next process can read the result.
2) Read after Read: Multiple read operations can be performed at the same time from a given data block
3) Write after Read: All readers must be finished before a process can write to a data block
4) Write after Write: A process needs to finish writing to a block of data before the next process writes to the same block of data. Note that writing also includes ''changing'' data, as in reading and writing at the same time or performing in-place operations.

2) REVIEW OF OFFLINE DAG FOR DATA DEPENDENCIES

According to these dependencies, each input/output data block of a block operation task can be annotated whether the block operation reads or writes the data block. This tracking of data blocks, or more generally of ''data objects'', allows to define the data dependencies. Generally, an iterative program can be parallelized for high-performance computing with a Directed Acyclic Graph (DAG) [19]. An offline DAG strategy for RLNC has been described in [21]. The offline DAG strategy involves a ''recording'' step, where the program is executed without performing the actual computations to record the data dependencies of each operation. The result

of this step is a DAG that represents the data dependencies. As a second step, this pre-recorded DAG is executed using a scheduler that executes the recorded operations in parallel when the respective dependencies have been cleared. In summary, in the offline DAG strategy [21], the DAG of the dependencies is created completely offline before the actual RLNC decoding is executed in real-time (online).

The offline DAG approach has two main drawbacks: First, the offline DAG approach in [21] employed the LU inversion to invert the entire coefficient matrix C and to multiply the inverted coefficient matrix C^{-1} with the data matrix D to decode all g symbols in a generation at the same time. The LU matrix inversion requires a complete matrix and is therefore not suitable for progressive decoding. A second drawback is that the schedule creation prior to the execution does not allow for branches in the execution, i.e., it is not possible to react to intermediate outcomes within the algorithm, e.g. skipping some computations if a coefficient is found to be zero. While this does not affect the LU matrix inversion approach, this inflexibility limits the applicability on our blocked Gauss Jordan algorithm where the backward substitution step only needs to be applied on few rows and therefore employs branching (see Section III-B2). Creating tasks for all rows in each iteration would lead to a lot of unnecessary computations.

3) ONLINE DAG CONCEPT FOR RLNC DECODING

Inspired by online data dependency management principles for parallelism in high performance computing [19], [61], we develop an online DAG approach for progressive RLNC decoding. Instead of constructing the entire DAG a priori (offline), we add block operations on the fly: A main thread executes the iterative RLNC program. Whenever there is a block operation to be executed, it will not be executed right away; instead, we add a new task description that characterizes the read and write dependencies in the online DAG. Based on the task operation description, the main thread "delegates" a block operation task to a later time or to another worker thread. At the same time, worker threads check for task descriptions in the DAG which have all dependencies resolved, pick them, and execute them. The worker threads will therefore pick up the delegated operations from the main thread. The independent individual worker threads can pick up and execute the operation tasks completely asynchronously.

Importantly, there is no explicit synchronization needed in this process; the main thread can go ahead and add new task descriptions similarly to the offline DAG approach. However, the main thread can also choose to wait for certain tasks or certain data objects to be finished, and then act upon the intermediate results. This makes branching possible, e.g., skipping certain computations if all coefficients are zero. However, this branching comes at the cost of adding an explicit synchronization, which can lead to idle times if not enough tasks are available to keep all cores busy.

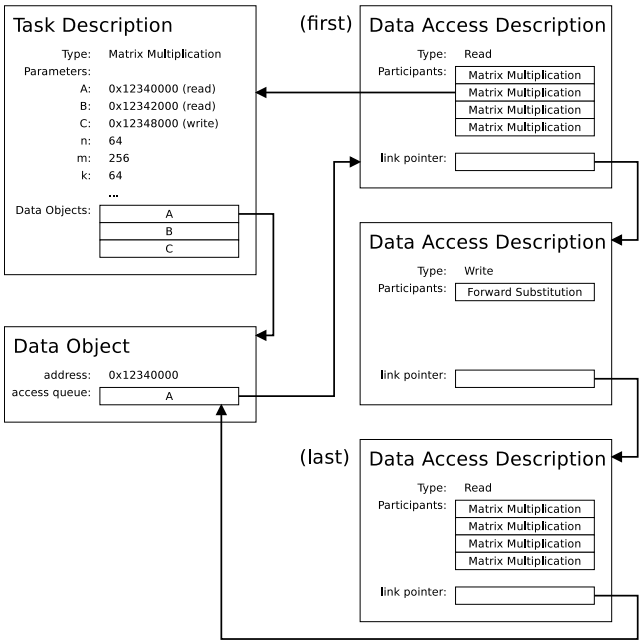


FIGURE 6. Illustration of data structures for online DAG for data dependencies.

4) DELEGATION OF MATRIX OPERATION TASKS

Figure 6 illustrates the data structures for the online DAG. In the depicted example, the main thread has delegated a matrix multiplication. All parameters are copied to the "task description" structure in the upper left of Figure 6 so the block operation can be called later as in conventional programs. The matrix multiplication block operation performs the operation $C = A \cdot B$. The task description therefore includes pointers to memory where the matrix blocks A, B, and C are stored, as well as information about the size of the matrices (n, m, k) and possibly more. For each pointer in the task description, such as the matrix block pointers A, B, and C, a separate "data object" is created which handles the data dependencies. Each data object contains an "access queue" where the data access of each task is registered when it is delegated. By registering the data accesses, the order of the data accesses of the tasks on the data objects is noted. In this way, the data dependencies are registered.

When a new task is delegated, data objects which will be accessed will be looked up from a hash, or created if not yet present. The data access for each data object will be registered. To do so, for each data object, the last data access description in the access queue (the one which was added last) will be investigated—the last data access represents the valid state after already scheduled (preceding) tasks have been executed. If the new task will read from the data object and the last data access is also reading, then the new task just adds itself as a "participant" to the participant list of the read data access. This is possible since multiple tasks can read from the same data. In any other case, a new operation is appended to the access queue and the delegated task is entered as the only participant.

Only the first data access in the data objects access queue is ''ready''. A task which is ready for execution needs to be ''ready'' with all tasks parameters' data objects. In other words, the task needs to be a participant of the first data access description of the access queue in all of the task parameters' data objects. To avoid look ups, we have a dependency counter in the task description which is decremented whenever a data access description becomes the first in the data access queue. When the dependency counter reaches zero, the task description is moved into a ''ready queue'' where worker threads can pick up and execute the task.

Once a task has been executed, the data objects and their entries in the respective data access participant lists are removed. If a participant list is empty, the data access is complete and the next data access in the queue is assigned to become the first of the queue; also, the dependencies of all task descriptions in the new data access participant list are decremented. New tasks may become ready for execution if their dependency counters have now been decremented to zero; those tasks have now their data dependencies met and can be safely executed. In case of multiple ready tasks, the threads select tasks as follows. We define forward elimination and backward substitution tasks as ''priority'' tasks. Slow cores, e.g., the LITTLE ODROID-XU-3 cores (see Section IV-A1) select the first non-priority task in the queue, but never execute a priority task. Fast cores execute the first priority task, or the first non-priority task if no other task is ready.

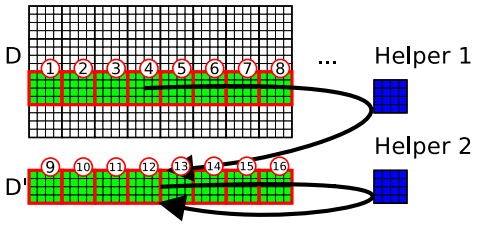
One caveat in this process is that the iterative program must be written in a way that data objects do not overlap in memory; otherwise, the described process cannot correctly detect data dependencies.

D. FURTHER OPTIMIZATIONS

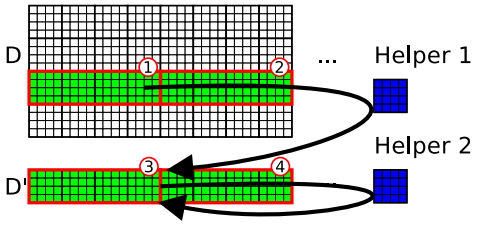
This section describes two optimizations of the principles described in Sections III-B and III-C.

1) STRIPE OPTIMIZATION

We have found that applying helper matrices on the remaining blocks in a block-by-block fashion creates many individual tasks, but also quite considerable overhead due to the dependency management for each task description. Instead of processing individual blocks, it may be preferable to group multiple blocks of a row into a ''stripe'' and to process the entire stripe as one delegated task so that there are fewer matrix multiplication tasks and less overhead. The stripe size should be large enough to avoid many ''short'' tasks, but small enough to create enough tasks descriptions to keep all cores (threads) busy. Figure 7 illustrates the stripe optimization for a forward elimination example. As illustrated in Figure 7(a), without the stripe optimization, the helper matrices are applied block-by-block on the new data matrix D', resulting in sixteen matrix multiplications tasks. Figure 7(b) shows the stripe optimization with a stripe size of $s = 16$, which reduces the number of tasks to four; these four tasks

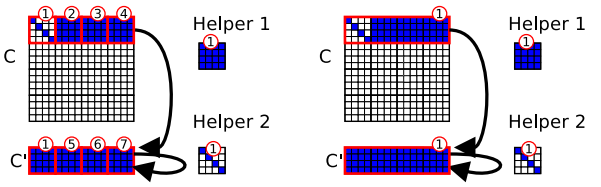


(a) Single block, 16 tasks



(b) Stripe optimization with stripe size $s = 16$, 4 tasks

FIGURE 7. Illustration of operation without and with stripe optimization for block size $b = 4$: Without stripe optimization, single $b \times b$ blocks are processed; with stripe optimization, stripes of b rows and s columns are processed.



(a) Default operation (b) Full rows optimization applied

FIGURE 8. Operation without and with full rows optimization.

involve larger matrices and require therefore longer computation times per task.

2) FULL ROWS OPTIMIZATION

Similarly to the striping of the data matrix, the ''full rows'' optimization reduces the number of block operation tasks in the coefficient matrix when performing forward elimination and backward substitution. The original implementation of the forward elimination and backward substitution computes a helper matrix and applies this helper matrix on the remaining blocks of a row using matrix multiplication block per block. The ''full rows'' optimization instead computes the helper matrix and immediately applies it on the remaining blocks of the same block row within one task. This full rows optimization consolidates multiple tasks into one. The consolidation reduces the task management overhead, but potentially creates a bottleneck if not enough tasks are available to keep all cores busy.

Figure 8 illustrates the full rows optimization for the forwarding elimination step (the backward elimination step works analogously): The default operation (without full rows optimization) computes the two helper matrices using the pivot block in the coefficient matrix C and the respective new coefficient block in C' in the first task. The remaining tasks apply the helper matrices on the remaining blocks in the

coefficient matrix C and new coefficient matrix C', respectively, using matrix multiplication. With full rows optimization, there is only one task performing all computations on the entire pivot block row of the coefficient matrix C and the complete new coefficient matrix C'. This single task takes longer to compute and is only processed by a single thread.

IV. EVALUATION

This section presents decoding throughput and delay measurements for two heterogeneous multicore processor boards that evaluate the proposed online DAG RLNC decoding approach. We conduct a measurement-based evaluation as to the best of our knowledge and understanding, a reasonably accurate analytical characterization of the decoding throughput and delay for heterogeneous multicore processors is currently intractable.

A. EVALUATION SET-UP

1) MULTICORE PROCESSOR BOARDS

The experiments have been conducted with contemporary system on a chip (SoC) boards, namely the ODROID-XU-3 (based on Samsung Exynos 5422 SoC) and ODROID-XU+E (based on Samsung Exynos 5410 SoC), which have been employed in previous RLNC coding evaluations [21], [33]. Both boards have four Cortex-A15 (big) cores (clocked at 2.0 GHz in XU-3 and 1.6 GHz in XU+E) and four Cortex-A7 (LITTLE) cores (clocked at 1.4 GHz in XU-3 and 1.2 GHz in XU+E). Also, on both boards, each core has an L1 cache of 32 KiB for data and 32 KiB for instructions with 64 byte cache lines. On both boards, the A15 cores share a 2 MiB L2 cache and the A7 cores share a 512 KiB L2 cache. Both boards have 2 GiB LPDDR3 random access memory (clocked at 933 MHz in XU-3 and 800 MHz in XU-E) and support NEON (an advanced form of single instruction multiple data (SIMD) instructions). The main difference is that the XU-3 supports heterogeneous multi-processing (HMP) while the XU+E does not. With HMP, the XU-3 can simultaneously utilize all eight cores, whereas the XU+E can only use either the four big cores or the four little cores (but not all eight cores at the same time).

2) TEST MATRICES AND PARAMETER SETTINGS

For each evaluation replication, we consider an independently randomly generated coding coefficient matrix C. Specifically, we consider two coefficient matrix types:

- **Full vector coefficient matrix** has all $g \times g$ elements set to random values. In general, no decoded symbols will be released until the last (gth) packet of the generation is processed by the decoder.
- **Triangle coefficient matrix** is a lower triangular matrix where all elements above the main diagonal are zero. Thus, the first packet (coding coefficient row) has only one non-zero coefficient, the second packet has two non-zero coefficients, and so on (we do not consider reordering of packets). This allows the decoder to decode and release one decoded symbol per incoming coded packet (provided the coding coefficient rows are linearly independent).

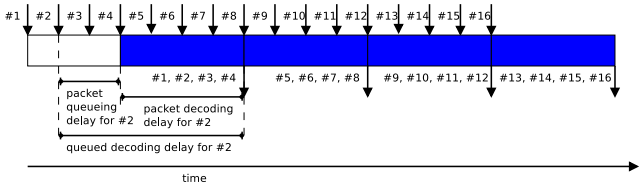


FIGURE 9. Illustration of packet queueing delay, packet decoding delay, and queued decoding delay for a progressive sub-generation decoder for generation size $g = 16$ and block size $b = 4$ (and block fill level $p = b = 4$) for a triangle coefficient matrix.

Throughout, we only consider linearly independent coding coefficient rows.

Data size m (symbol size) was varied between $m = 1024$ representing short data packets, $m = 1536$ which is close to the typical Ethernet Maximum Transfer Unit (MTU) of 1500 bytes but a multiple of 512, as well as $m = 4096$ and $m = 16384$ Bytes which may be used for data center storage. (Unequal packet sizes can be accommodated with padding or other techniques [62].)

Generation size g has been chosen within $16, 32, 64, \ldots, 1024$. We consider zero redundant data packets, i.e., we assume that all g coded packets for a given generation are received (eventually, according to the arrival process defined in Section IV-A3) and have linearly independent coding coefficient rows. Larger generation sizes generally yield lower decoding throughput and higher latency, but have better chances to recover packets lost in an error burst [63].

Threads t are varied between $t = 1, 2, 4$, and 8 threads. Threads are pinned on the physical CPUs to ensure that the operating system scheduler never overbooks cores with multiple threads. The pinning also ensures that for $t = 4$ and fewer threads, only the big cores are used.

Block size b has been chosen within $16, 32, 64, \ldots, g$. The block size $b = 16$ is the minimum block size where the NEON SIMD code can perform at its full potential.

Block fill level p was set to $1, 2, 4, \ldots, b$.

Stripe size s has been chosen from $s = 64, 128, 256$, and 512.

Full rows optimization has been turned on and off.

3) PERFORMANCE METRICS

We define the **throughput** as the size gm [bytes] of the data matrix D divided by the measured decoding computation time from the time instant when the decoder starts processing the first coded packet to the time instant when the decoder emits the last (gth) decoded symbol of a given generation.

We define the **packet decoding delay** as the time interval from the time instant when a coded symbol enters the decoder until the time instant when the decoder emits the corresponding decoded symbol, as illustrated in Fig. 9 for coded packet #2. In our evaluations, we consider $r = 0$

redundant packets; thus, there are exactly as many coded symbols as decoded symbols, so a coded symbol will correspond to a decoded symbol. We define the **queued decoding delay** as the sum of the **packet decoding delay** and the **packet queueing delay**, which is evaluated as follows. In a progressive sub-generation decoder, the decoder only starts computing an iteration when block fill level p packets have been received. However, we can not expect that packets arrive in convenient batches of p packets just in time as the decoder needs them to proceed without stalling. Instead, we assume that packets arrive evenly distributed with a constant inter-packet (time interval) spacing (which is computed based on the measured throughput; note that the throughput is the inverse of the decoding delay of a full generation of g packets). The queue entry times of the packets are set to the latest possible entry times that satisfy a constant inter-packet spacing for all g packets in a generation, while ensuring that p packets have been received when the decoder needs the next batch of p packets to proceed without stalling. (For the offline DAG approach, i.e., the generation-based benchmark, all $p = g$ packets of a generation need to be received by the time the decoder starts decoding the generation.) The queueing delay is defined as the time interval from the time instant when a packet enters the queue to the time instant when the decoder pulls the packet from the intermediate receive queue. (For the offline DAG benchmark, the average packet queueing delay equals half the decoding delay for a generation.) Thus, the queued decoding delay is the time interval from the time instant when a packet enters the queue (for an equal packet spacing arrival pattern) until the time instant when the decoder emits the corresponding decoded symbol, as illustrated for packet #2 in Fig. 9. We average the g individual queued decoding delays for the g symbols in a given generation to obtain one mean queued decoding delay for a given generation (i.e., for a given evaluation replication).

We briefly note that alternatively the delay could be defined as the time interval from the time instant when the last coded symbol of a generation enters the decoder until the time instant when the decoder has emitted all symbols of the generation. However, such a delay definition based on the last symbol does not capture the effects of symbols that are decoded and released early to the upper protocol layers. For instance, in the example illustrated in Fig. 9, symbols #1 through #8 are decoded and released to the upper layers even before the last symbol #16 arrives. Accordingly, in order to accurately evaluate the decoding delay performance of progressive decoding we do not consider this alternate ''last symbol'' delay definition; instead, we consider the queued decoding delay as defined above based on individual symbols.

4) EVALUATION METHODOLOGY

We conducted all evaluations with the Galois Field GF(2^8) using the NEON-enabled SIMD code of the Fifi/Kodo library [64], which employs fast SIMD operations for the Galois Field [32], [65].

We present results for prescribed combinations of coefficient matrix type C, data size m, generation size g, and number of threads t in the figures in the next section. For each considered prescribed combination of C type, m, g, and t, we performed 20 independent replications for each combination of the remaining parameters block size b ($b = 16, 32, 64, \ldots, g$), block fill level p ($p = 1, 2, 4, \ldots, b$), and stripe size s ($s = 64, 128, 256, 512$), as well as full rows optimization (on or off) to determine the combination of b, p, s, and full rows optimization that achieves the maximum throughput as well as the combination of b, p, s, and full rows optimization that achieves the minimum delay. For these maximum-throughput and minimum-delay settings of b, p, s, and the full rows optimization, we then conducted between 18000 and 60000 independent replications of the evaluation for each of the prescribed combinations of C type, m, g, and t. All presented throughput and delay results are averages over the independent replications. The resulting widths of the 95% statistical confidence intervals of the evaluated performance metrics were less than 0.5% of the respective means and are omitted from the plots to avoid visual clutter.

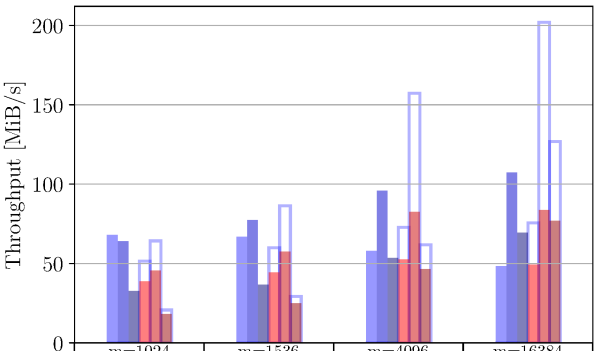
B. RESULTS

1) THROUGHPUT

a: PROGRESSIVE ONLINE DAG VS. NON-PROGRESSIVE OFFLINE DAG

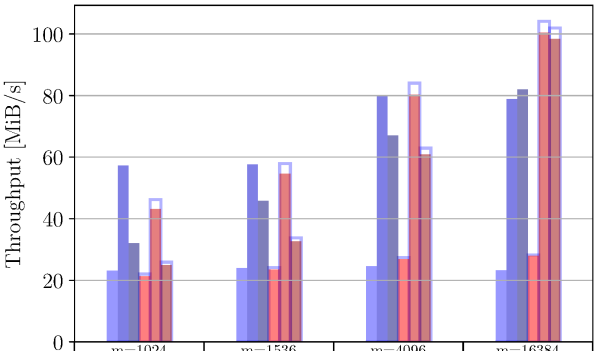
Fig. 10 compares the decoding throughput of the proposed progressive online DAG approach with the non-progressive offline DAG approach [21]. The comparison considers the range of generation sizes g and symbol sizes m for $t = 1, 4$, and 8 threads and was conducted with the ODROID-XU-3 board, which can operate up to eight cores simultaneously. We observe from Fig. 10 that the online DAG approach achieves generally similar decoding throughput levels as the offline DAG approach. Focusing on Fig. 10(a) for the small generation size $g = 16$, we observe that the online DAG approach achieves generally somewhat higher throughput than the SE throughput for scheduling and executing the offline DAG. This is mainly because the online approach includes the stripe optimization (Section III-D1), whereas the offline DAG approach of [21] is limited to block processing and does not form stripes. For small generation sizes g and correspondingly small block sizes b, $b \leq g$, the individual block computing tasks are relatively small. Thus, the overhead of scheduling the computing tasks is relatively high compared to the actual computing task execution. The stripe optimization reduces the scheduling overhead relative to the task execution and thus increases the decoding throughput.

We further observe from Fig. 10(a) that the execute (E) decoding throughputs achieved with a pre-scheduled (pre-recorded) offline DAG (represented by the outlined bars in Fig. 10) significantly exceed the online DAG decoding throughputs for the large $m = 4096$ and 16384 symbol sizes. For large symbol sizes there are numerous block computing tasks in the offline DAG approach [21] offering plentiful



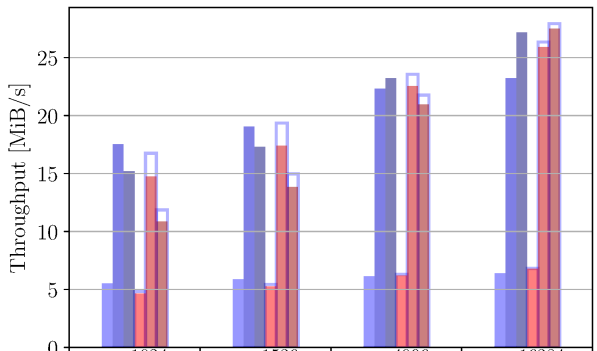
	m=1024	m=1536	m=4096	m=16384
Progr1	68.1	67.02	58.0	48.41
Progr4	64.0	77.43	96.01	107.44
Progr8	32.8	36.7	53.54	69.39
Offl1	38.92 / 51.45	44.54 / 59.83	52.56 / 72.79	49.48 / 75.55
Offl4	45.57 / 64.25	57.52 / 86.41	82.54 / 157.26	83.74 / 201.88
Offl8	18.29 / 20.72	25.04 / 29.26	46.64 / 61.69	76.99 / 126.86

(a) $g = 16$



	m=1024	m=1536	m=4096	m=16384
Progr1	23.22	24.0	24.59	23.27
Progr4	57.35	57.66	80.25	78.87
Progr8	32.04	45.81	67.13	81.98
Offl1	21.30 / 22.00	23.51 / 24.10	26.95 / 27.35	27.95 / 28.23
Offl4	43.20 / 46.21	54.61 / 57.90	80.37 / 84.04	100.43 / 104.09
Offl8	24.96 / 25.87	32.69 / 33.79	60.90 / 62.91	98.45 / 101.91

(b) $g = 64$



	m=1024	m=1536	m=4096	m=16384
Progr1	5.55	5.89	6.16	6.39
Progr4	17.53	19.06	22.32	23.24
Progr8	15.22	17.3	23.24	27.19
Offl1	4.65 / 4.83	5.27 / 5.43	6.22 / 6.29	6.78 / 6.81
Offl4	14.75 / 16.75	17.41 / 19.35	22.54 / 23.56	25.93 / 26.33
Offl8	10.88 / 11.86	13.86 / 14.97	20.96 / 21.76	27.49 / 27.92

(c) $g = 256$

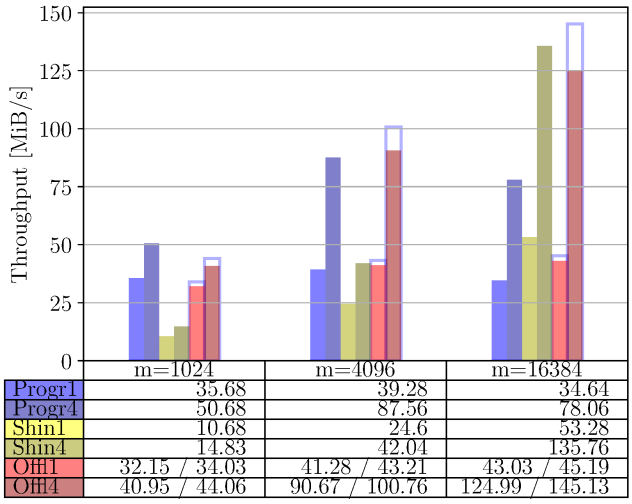
FIGURE 10. Throughput comparison of progressive (online DAG) decoding (blue) with the non-progressive (offline DAG) decoding [21] (red bars, SE as solid bars and E as outlines), for $t = 1, 4$, and 8 threads for various generation sizes g and symbol sizes m on XU-3 for full vector coefficient matrix.

opportunities for parallelization. With pre-recording of the offline DAG schedule, the overhead of scheduling the numerous computing tasks does not directly influence the actual decoding computation. Thus, the benefits from executing the pre-recorded DAG with an increased level of parallelization can outweigh the overheads of handling more computing tasks, and thus increase throughput compared to the online DAG approach that schedules the computing tasks in a "live" manner.

Overall, the decoding throughput results in Fig. 10 demonstrate that the online DAG approach exhibits the same general performance trends as the conventional offline DAG approach: The decoding throughputs generally decrease with increasing generation size g, mainly due to the increasing computational complexity of $O(g^3)$ of the inversion of the coding coefficient matrix C, also each element has to be touched g times for the matrix multiplication. Moreover, the decoding throughputs increase with increasing symbol size m, mainly due to the increasing opportunities for parallelization. Importantly, the online DAG approach can generally take advantage of multiple threads as efficiently as the offline DAG approach. With both approaches, the decoding throughputs typically greatly increase with four parallel computing threads (cores) compared to a single computing thread. With eight threads, the throughputs decrease for small to moderate generation sizes g and symbol sizes m compared to four threads, while eight threads achieve throughput increases for large g in combination with large m. Combinations of large g and m provide many opportunities for parallel execution which outweigh the overhead of operating eight cores and the delays when critical tasks are scheduled on one of the slow (LITTLE) cores.

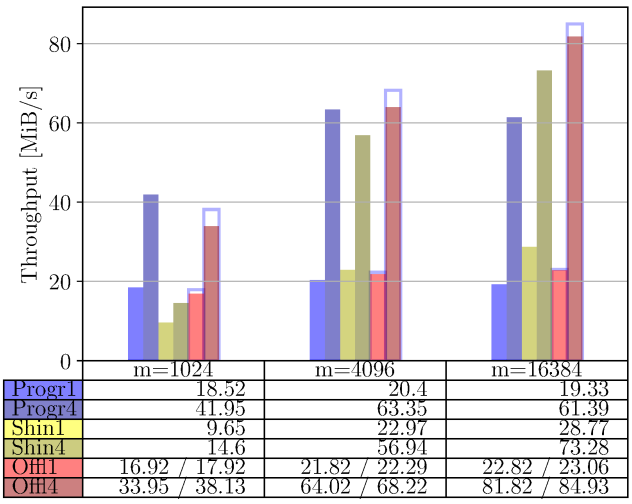
b: PROGRESSIVE ONLINE DAG VS. PROGRESSIVE CD

Fig. 11 compares the decoding throughput of the proposed progressive online DAG approach with the state-of-the-art progressive coefficient matrix duplication (CD) approach [33], for additional context, the decoding throughputs of the offline DAG approach from [21] are also included. These comparisons were conducted with the XU+E board, which was considered in [33] and can operate up to four cores simultaneously. We observe from Fig. 11 that our online DAG approach achieves higher decoding throughput than the CD approach for the small symbol size $m = 1024$, whereby the advantage of the online DAG approach is particularly pronounced for the small generation sizes $g = 32$ and 64. This result is mainly due to the relatively high static overhead of the CD approach, which dominates the computational complexity for small generation sizes [21], [33]. In contrast, we observe from Fig. 11(a) that the proposed online DAG approach has a low static overhead and achieves for $g = 32$, $m = 1024$ more than three times the decoding throughput of the CD approach. Small generation and symbol sizes are especially relevant for low-delay applications, such as media streaming.



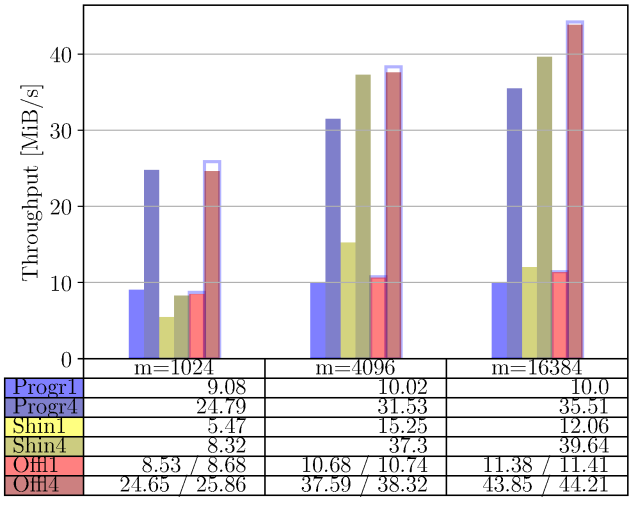
	m=1024	m=4096	m=16384
Progr1	35.68	39.28	34.64
Progr4	50.68	87.56	78.06
Shin1	10.68	24.6	53.28
Shin4	14.83	42.04	135.76
Offl1	32.15 / 34.03	41.28 / 43.21	43.03 / 45.19
Offl4	40.95 / 44.06	90.67 / 100.76	124.99 / 145.13

(a) $g = 32$



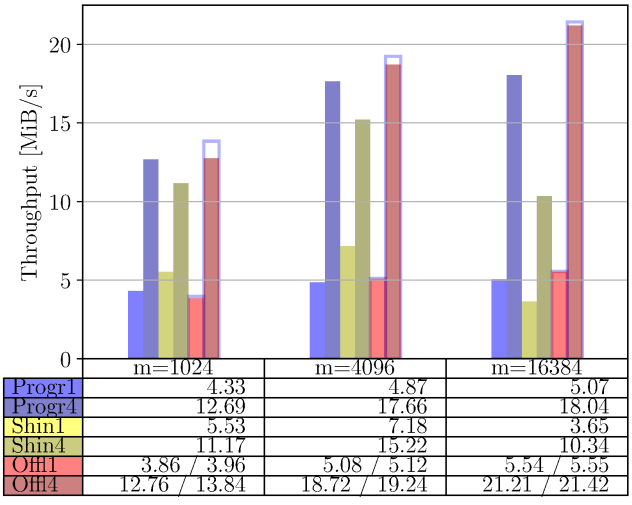
	m=1024	m=4096	m=16384
Progr1	18.52	20.4	19.33
Progr4	41.95	63.35	61.39
Shin1	9.65	22.97	28.77
Shin4	14.6	56.94	73.28
Offl1	16.92 / 17.92	21.82 / 22.29	22.82 / 23.06
Offl4	33.95 / 38.13	64.02 / 68.22	81.82 / 84.93

(b) $g = 64$



	m=1024	m=4096	m=16384
Progr1	9.08	10.02	10.0
Progr4	24.79	31.53	35.51
Shin1	5.47	15.25	12.06
Shin4	8.32	37.3	39.64
Offl1	8.53 / 8.68	10.68 / 10.74	11.38 / 11.41
Offl4	24.65 / 25.86	37.59 / 38.32	43.85 / 44.21

(c) $g = 128$



	m=1024	m=4096	m=16384
Progr1	4.33	4.87	5.07
Progr4	12.69	17.66	18.04
Shin1	5.53	7.18	3.65
Shin4	11.17	15.22	10.34
Offl1	3.86 / 3.96	5.08 / 5.12	5.54 / 5.55
Offl4	12.76 / 13.84	18.72 / 19.24	21.21 / 21.42

(d) $g = 256$

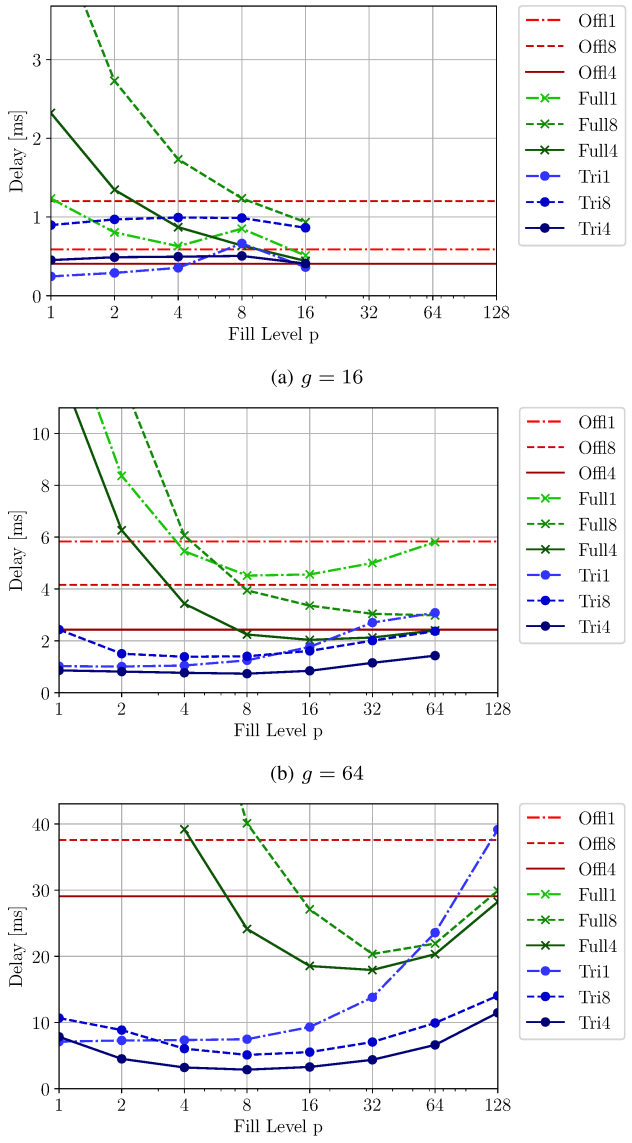
FIGURE 11. Throughput comparison of progressive (online DAG) decoding (blue), CD decoding [33] (yellow), and non-progressive (offline) DAG decoding [21] (red bars, SE as solid bars and E as outlines), for $t = 1$ and 4 threads for various generation sizes g and symbol sizes $m = 1024$, 4096, and 16384 on ODROID-XU+E for full vector coefficient matrix.

On the other hand, we observe from Fig. 11 that the CD approach achieves higher decoding throughput than the online DAG approach for the very large symbol size $m = 16384$ in combination with small to moderately large generation sizes $g = 32$, 64, and 128. This result is mainly due to the more efficient matrix multiplication in the CD approach, where a given computing thread can handle a complete vertical partition of the data matrix D with low overhead as one "computing task". In contrast, our online DAG approach incurs overheads due to managing each matrix block as a distinct computing task.

However, we observe from Fig. 11(d) that for the very large symbol size $m = 16384$ in conjunction with the large generation size $g = 256$, the online DAG approach achieves substantially higher decoding throughputs than the CD approach. The throughput degradation of the CD approach for large g and m is mainly due to caching inefficiencies [33]. In contrast, our block-based online DAG approach remains highly cache efficient for scenarios with large g and m, which are highly relevant for storage applications.

2) DELAY

Fig. 12 presents the mean queued decoding delay as a function of the block fill level p. For the delay evaluation, the methodology from Section IV-A4 was modified to prescribe a combination of C type, data size m, generation size g, number of threads t, and block fill level p to find the combination of the remaining parameters block size b, stripe size s, and full rows optimization that minimizes the mean queued decoding delay. The exhaustive search considered 20 independent replications for each combination of these remaining parameters. We then conducted between 18000 and 60000



(a) $g = 16$

(b) $g = 64$

(c) $g = 256$, Offl1 and Full1 are above plotted range

FIGURE 12. Mean queued decoding delay for online DAG approach for $t = 1$, 4, and 8 threads for full vector (Full) and triangle (Tri) coefficient matrices C, for various generation sizes g with symbol size $m = 1536$ as a function of the block fill level p on ODROID-XU-3. The offline DAG (Offl) [21] considers full vector coefficient matrices.

independent replications of the evaluation for each of the prescribed combinations of C type, m, g, t, and p (for the delay minimizing settings of the remaining parameters), obtaining 95% statistical confidence intervals of less than 0.5% of the respective means.

We observe from Fig. 12(a) that for the small generation size $g = 16$ there are no pronounced differences between the minimum delays achieved by the compared approaches. This is mainly because the small $g = 16$ generation size does not provide sufficient "room" to significantly reduce delays through progressive decoding. Rather, for full vector coefficient matrices, progressive decoding with a block fill level p less than the generation size $g = 16$ increases delays due to the introduced computation inefficiencies for new coefficient C' and data D' matrices with less rows than the minimum block size $b = 16$. Triangular coefficient matrices can offset these computational inefficiencies by releasing the decoded symbols early and achieve minuscule delay reductions for a block fill level of $p = 1$.

We observe from Fig. 12(b) and (c) that progressive decoding with the online DAG approach can achieve substantial delay reductions compared to the generation-based offline DAG approach. The delay reductions become more pronounced with increasing generation size g and with the Tri C type. For instance, we observe from Fig. 12(c) that for $t = 4$ threads and the Full C type, the block fill level $p = 32$ in the online DAG approach reduces the delay down to approximately two thirds of the offline DAG delay. These delay reductions for full vector coefficient matrices are mainly due to the reduction of the decoding computation delay from the time instant of the receipt of the last (gth) packet of a generation until the time instant when the generation has been completely decoded and all g decoded packets are released. In particular, with the online DAG approach, only the last p received packets in the new coefficient matrix C' and the new data matrix D' need to be "worked" via the Gauss Jordan elimination into the coefficient matrix C and data matrix D, which can already contain $g - p$ rows when the decoding process for the last p packets starts.

We observe from Fig. 12(c) that for $t = 4$ threads for the Tri C type, the block fill level $p = 8$ reduces the delay down to nearly one tenth of the offline DAG delay. The delay reductions for the triangular coefficient matrix are due to a combination of the reduced computation time for "working" p received packets into the matrices C and D as well as the early release of the decoded packets immediately after completing the processing of each batch of p packets. Large generation sizes g provide more "room" for these delay reductions compared to offline DAG decoding.

Generally, the delay curves for the online DAG approach in Fig. 12(b) and (c) exhibit a delay decrease for increasing block fill level p up to a delay-optimal p value and then increasing delays for further increases of p. For small block fill levels p below the minimum block size $b = 16$, the block has to be padded, creating inefficiencies and thus delays increases that become more pronounced for very small p. Large block fill levels p require correspondingly large block sizes b because the block fill level must be less than or equal to the block size b, i.e., $p \leq b$. Large blocks are more difficult to cache, reducing the caching efficiency. Additionally, the Tri C type permits the release of the p symbols in a processing batch as soon as the decoding is completed for the batch. A large block fill level p increases the packet queueing delay until the decoding processing can commence, contributing to the delay increases for large p for the Tri C type.

The delay results for $t = 8$ threads corroborate the earlier findings in Section IV-B1.a regarding the additional delays and overheads arising from utilizing the four slow (LITTLE)

TABLE 2. Optimized parameter settings for online DAG approach: For prescribed coefficient matrix C type, data size $m = 1536$, generation size g, and number of threads t on ODROID-XU-3, the tables give the block size b, block fill level p, stripe size s, and full row optimization achieving the highest throughput in MiB/s (Fig. 10) and the shortest queued decoding delay in ms (Fig. 12), respectively. The respective optimized performance metric (throughput or delay) is written in bold.

C typ.	t	b	p	s	fullr.	TH	Del.
Full	1	16	16	512	off	**67.02**	0.51
Full	4	16	16	512	on	**77.43**	0.44
Full	8	16	16	512	on	**36.70**	0.93
Full	1	16	16	512	off	67.02	**0.51**
Full	4	16	16	512	on	77.43	**0.44**
Full	8	16	16	512	on	36.70	**0.93**
Tri	1	16	16	512	on	**93.47**	0.36
Tri	4	16	16	512	on	**83.57**	0.41
Tri	8	16	16	512	off	**39.36**	0.86
Tri	1	16	1	512	off	11.79	**0.25**
Tri	4	16	16	512	on	83.57	**0.41**
Tri	8	16	16	512	off	39.36	**0.86**

(a) $g = 16$

C typ.	t	b	p	s	fullr.	TH	Del.
Full	1	64	64	512	off	**24.00**	5.81
Full	4	64	64	256	off	**57.66**	2.40
Full	8	64	64	128	off	**45.81**	2.98
Full	1	32	8	512	on	17.49	**4.52**
Full	4	64	16	512	on	46.38	**2.04**
Full	8	64	64	128	off	45.81	**2.98**
Tri	1	64	64	512	on	**45.13**	3.08
Tri	4	64	64	256	on	**96.46**	1.43
Tri	8	64	64	128	off	**58.01**	2.37
Tri	1	32	2	512	off	8.01	**1.01**
Tri	4	64	8	512	off	37.55	**0.73**
Tri	8	64	4	512	off	10.25	**1.39**

(b) $g = 64$

C typ.	t	b	p	s	fullr.	TH	Del.
Full	1	64	64	512	on	**5.89**	67.45
Full	4	64	64	256	on	**19.06**	20.32
Full	8	64	64	256	on	**17.30**	21.93
Full	1	32	32	512	on	5.70	**56.93**
Full	4	64	32	512	on	17.92	**17.93**
Full	8	64	32	256	on	14.19	**20.35**
Tri	1	64	64	512	on	**11.16**	23.58
Tri	4	64	64	512	off	**31.53**	6.64
Tri	8	128	128	128	on	**24.78**	14.44
Tri	1	128	1	512	off	0.73	**7.10**
Tri	4	64	8	512	on	11.90	**2.87**
Tri	8	128	8	512	on	6.07	**5.10**

(c) $g = 256$

cores in addition to the four fast (big) cores on the considered SoC boards.

3) PARAMETER SETTINGS FOR THROUGHPUT-DELAY TRADEOFF

The goal of this section is to provide insights into the range of throughput-delay tradeoffs that can be attained with the proposed online DAG approach and the corresponding parameter settings that achieve the various throughput-delay tradeoffs. Table 2 considers both types of coding coefficient matrices (C types), the $m = 1536$ data size, generation sizes $g = 16$, 64, and 256, and $t = 1$, 4, and 8 threads. For each prescribed combination of C type, m, g, and t, Table 2 gives the block size b, block fill level p, stripe size s, and full row optimization (on or off) that achieve the highest throughput and the shortest queued decoding delay, respectively.

We observe from Table 2 that for the Full C type, the block size b and block fill level p are generally 16 for the small generation size $g = 16$; whereas, for the larger $g = 64$ and 256 generation sizes, the block size $b = 64$ corresponds to the 64-byte length of the L1 cache lines while the block fill level is $p = 64$ for the maximum throughput and smaller block fill levels of $p = 8$, 16, and 32 achieve the minimum delays as specified in Table 2. Interestingly, the differences between the b, p, s, and full row optimization for highest throughput and shortest delay are relatively minor for the Full C type. Concomitantly, the differences between the highest throughputs and the throughputs corresponding to the shortest delays are relatively minor; specifically, for $g = 16$, the highest throughput and shortest delay operating points coincide; for $g = 64$ and 256, the maximum throughput operating points have somewhat higher delays (up to about 20% higher for $g = 256$) than the shortest delay operating points; conversely, the shortest delay operating points have somewhat lower throughputs (up to 20% lower for $g = 256$) than the highest throughput operating points. The stripe sizes are $s = 512$ for $g = 16$ as well as for $g = 64$ and 256 for small thread numbers t, whereas large thread numbers t tend to prefer shorter stripe sizes of $s = 256$ and 128.

In contrast, we observe for the Tri C type in Table 2 large differences between the highest throughput and shortest delay parameter settings and achieved throughput-delay operating points. For instance, for $t = 4$ threads for the generation sizes $g = 64$ and 256, the highest throughputs are roughly three times higher than the throughputs for the shortest delay operating points; the delays of the highest throughput operating points are a little over twice the shortest delays, albeit still short compared to the Full C type delays. This wide range of throughput-delay tradeoffs of the operating points for Tri C type decoding is mainly due to the wide range of flexibilities for releasing decoded packets as soon as block fill level p, $p \leq b \leq g$, packets have been received and decoded. Decoding $p < b$ received coded packets can reduce the delays but substantially degrades the throughput as $b - p$ rows need to be zero padded in the new coefficient matrix C' and the new data matrix D' to enable the block based matrix processing.

Generally, for Tri C type decoding, the highest throughput settings have large block sizes b equal to the generation size g for $g = 16$ and 64, and up to twice the cache line length, i.e., up to 128, for $g = 256$. The block fill levels match the block sizes, i.e., $p = b$ for the highest throughput; whereas, the shortest delay requires small block fill levels in the range from 1 to 8. We also observe from Table 2 that the high throughput vs. short delay tradeoff is mainly controlled by the block fill level p (while the other parameters b, s, and full row optimization exhibit generally only minor differences for the different operating points).

Generally, for utilizing the online DAG approach in operational practice, we recommend to first conduct an offline

TABLE 3. Optimized throughput in MiB/s and queued decoding delay in ms for sparse RLNC as a function of density γ/g for random sparsity and pseudo-systematic sparsity for full coefficient matrix C type, data size $m = 1536$, generation size $g = 64$, and $t = 4$ threads on ODROID-XU-3. The respective optimized performance metric (throughput or delay) is written in bold.

Spar. typ.	Opt. typ.	Density γ/g 6/64 TH, Del.	12/64 TH, Del.	25/64 TH, Del.	51/64 TH, Del.
Ran.	TH	**74.01**, 1.65	**64.47**, 1.85	**61.71**, 1.94	**57.69**, 2.40
Ran.	Del.	61.47, **1.48**	55.83, **1.62**	53.69, **1.72**	48.88, **1.93**
Sys.	TH	**89.77**, 1.29	**84.53**, 1.38	**72.61**, 1.66	**58.13**, 2.38
Sys.	Del.	74.17, **1.12**	67.63, **1.28**	58.02, **1.59**	49.26, **1.93**

search for the optimal parameter settings. Specifically, for the employed coding coefficient matrices (C types), data sizes m, generation sizes g, and available number t of processor threads we recommend to conduct simulations for the different combinations of block size b, block fill level p, stripe size s, and full row optimization, similar to our evaluation methodology (see Section IV-A4) so as to determine the throughput-delay tradeoffs. Then, the desired operational objective in terms of the high throughput vs. short delay tradeoff can be attained through appropriate selection of the parameters b, p, s, and full rows optimization.

4) IMPACT OF SPARSE RLNC

Sparse RLNC has recently been proposed to reduce the RLNC computations [66]–[69]. With sparse RLNC, only a prescribed subset of the coding coefficients in the coefficient matrix C are set to values drawn randomly from a given GF(2^p). In particular, only a subset of γ, $\gamma \leq g$, of the source symbols of a given generation of g source symbols are combined to form a coded symbol. We consider a random sparsity approach that uniformly randomly selects γ source symbols among the g source symbols of the generation to form a coded symbol [70]. That is, γ uniformly randomly selected positions (out of the g positions) of the coding coefficient row are set to random GF(2^p) values and the remaining $g - \gamma$ positions are zero. More specifically, in order to avoid dimensionality problems, we selected the position on the diagonal of the coding coefficient matrix (i.e., position i of coding coefficient row i, $i = 1, 2, \ldots, g$) and then $\gamma - 1$ other random positions of the row.

We also consider the pseudo-systematic perpetual sparse coding from [60], which prescribes a particular coding coefficient row pattern for the successive rows r, $r = 1, 2, \ldots, g$, of C: Row r has a one in position r and random GF(2^p) values in the next $\gamma - 1$ positions (with wrap-around, e.g., row $r = g$ has a one in position g and random GF(2^p) values in positions 1 through $\gamma - 1$).

Following the evaluation methodology of Section IV-B3, we examine the highest throughput and the shortest queued decoding delay. In particular, we consider the full coefficient matrix C type for symbol size $m = 1536$, generation size $g = 64$, and $t = 4$ threads. Table 3 presents the throughput and queued decoding delay values for the highest throughput and shortest queued decoding delay operating points. We vary the sparsity, or equivalently the density, of the encoding by varying the ratio γ/g.

We observe from Table 3 in comparison with the entries for the Full C type and $t = 4$ threads in Table 2(b) that, as expected, the throughput and delay values for sparse coding with a high density γ/g approach the corresponding throughput and delay values for conventional encoding (which corresponds to $\gamma/g = 1$). For decreasing density γ/g, i.e., increasing levels of sparsity in the RLNC coding, we observe from Table 3 increasing throughputs and decreasing delays. The throughput increases and delay reductions are more pronounced for the pseudo-systematic sparse coding than for the random sparse coding. These results indicate that the proposed online DAG RLNC decoding approach can extract performance gains from sparse RLNC coding. The performance gains are particularly pronounced for the pseudo-systematic sparsity approach which has more consecutive zeros in the coding coefficient rows, thus allowing for more efficient processing, than for the random sparsity approach.

V. CONCLUSION

We have developed and evaluated a novel progressive decoding methodology for Random Linear Network Coding (RLNC). Our methodology schedules matrix block operations in an online manner in a directed acyclic graph (DAG) so as to facilitate highly efficient parallel computation on multicore processors while allowing for dynamic branching decisions. Our performance evaluations on heterogeneous multiprocessor boards have indicated that our online DAG approach achieves similar decoding throughput as the state-of-the-art non-progressive (offline DAG) RLNC decoding methodology. Importantly, the online DAG approach achieves three times higher decoding throughput than the state-of-the-art progressive coefficient matrix duplication (CD) RLNC decoding for small generations of 1 KiB data symbols and is thus well suited for low-latency packet network applications. Also, the online DAG approach achieves higher decoding throughput than CD decoding for large generations of very large data symbols, which are common in storage applications. Compared to offline decoding, the progressive online DAG approach can reduce the decoding delay by one third for full vector RLNC encoding, while delay reductions by a factor of ten are possible for low-latency RLNC encoding methods. The online DAG approach allows for the control of the high throughput vs. short decoding delay tradeoff through a block fill level parameter that indicates how many received coded packets are processed at a time.

The online DAG approach developed in this study opens up several interesting future research directions. The present study developed the online DAG approach for heterogeneous multicore processor systems without an extensive GPU infrastructure. Future work could extend the online DAG approach to utilize GPUs and investigate how to manage the task assignments to GPUs so as to minimize decoding latencies. Another future research direction is to examine

online DAG based RLNC decoding for RLNC versions that are not based on generations, but rather employ dynamic sliding encoding windows, e.g., [38].

REFERENCES

[1] H. Alshaheen and H. Takruri-Rizk, "Energy saving and reliability for wireless body sensor networks (WBSN)," *IEEE Access*, vol. 6, pp. 16678–16695, 2018.

[2] A. Ahmed, H. Shan, and A. Huang, "Modeling the delivery of coded packets in D2D mobile caching networks," *IEEE Access*, vol. 7, pp. 20091–20105, 2019.

[3] W. He, Y. Su, X. Xu, Z. Luo, L. Huang, and X. Du, "Cooperative content caching for mobile edge computing with network coding," *IEEE Access*, vol. 7, pp. 67695–67707, 2019.

[4] N. Ma and M. Diao, "CoFi: Coding-assisted file distribution over a wireless LAN," *Symmetry*, vol. 11, no. 1, pp. 71:1–71:19, 2019.

[5] R. Torrea-Duran, M. M. Céspedes, J. Plata-Chaves, L. Vandendorpe, and M. Moonen, "Topology-aware space-time network coding in cellular networks," *IEEE Access*, vol. 6, pp. 7565–7578, 2018.

[6] K. Lei, S. Zhong, F. Zhu, K. Xu, and H. Zhang, "An NDN IoT content distribution model with network coding enhanced forwarding strategy for 5G," *IEEE Trans. Ind. Informat.*, vol. 14, no. 6, pp. 2725–2735, Jun. 2017.

[7] T. Li, W. Chen, Y. Tang, and H. Yan, "A homomorphic network coding signature scheme for multiple sources and its application in IoT," *Secur. Commun. Netw.*, vol. 2018, pp. 9641273-1–9641273-6, Jun. 2018.

[8] J. Li, Y. Liu, Z. Zhang, J. Ren, and N. Zhao, "Towards green IoT networking: Performance optimization of network coding based communication and reliable storage," *IEEE Access*, vol. 5, pp. 8780–8791, 2017.

[9] Y. N. Shnaiwer, S. Sorour, T. Y. Al-Naffouri, and S. N. Al-Ghadhban, "Opportunistic network coding-assisted cloud offloading in heterogeneous fog radio access networks," *IEEE Access*, vol. 7, pp. 56147–56162, 2019.

[10] H. Kang, H. Yoo, D. Kim, and Y. S. Chung, "CANCORE: Context-aware network coded repetition for VANETs," *IEEE Access*, vol. 5, pp. 3504–3512, 2017.

[11] X. Shao, C. Wang, C. Zhao, and J. Gao, "Traffic shaped network coding aware routing for wireless sensor networks," *IEEE Access*, vol. 6, pp. 71767–71782, 2018.

[12] J.-W. Kim and J.-S. No, "Code equivalences between network codes with link errors and index codes with side information errors," *IEEE Access*, vol. 7, pp. 54144–54154, 2019.

[13] F. A. Monteiro, A. Burr, I. Chatzigeorgiou, C. Hollanti, I. Krikidis, H. Seferoglu, and V. Skachek, "Special issue on network coding," *EURASIP J. Adv. Signal Process.*, vol. 2017, no. 29, pp. 29-1–29-3, Apr. 2017.

[14] H. V. Nguyen, S. X. Ng, W. Liang, P. Xiao, and L. Hanzo, "A network-coding aided road-map of large-scale near-capacity cooperative communications," *IEEE Access*, vol. 6, pp. 21592–21620, 2018.

[15] Q. Wang, X. Zhang, Q. Wang, P. Liu, and B. Deng, "The network coding algorithm based on rate selection for device-to-device communications," *IEEE Access*, vol. 7, pp. 23396–23406, 2019.

[16] C. Zhang, C. Li, and Y. Chen, "Joint opportunistic routing and intra-flow network coding in multi-hop wireless networks: A survey," *IEEE Netw.*, vol. 33, no. 1, pp. 113–119, Jan. 2019.

[17] J. Kwon and H. Park, "Low complexity algorithms for network coding based on singular value decomposition," in *Proc. IEEE Int. Conf. Ubiquitous Future Netw. (ICUFN)*, Jul. 2016, pp. 641–643.

[18] J. Wang, K. Xu, X. Yang, L. Chen, W. Xie, and J. Xu, "On minimizing decoding complexity for binary linear network codes," in *Proc. Int. Conf. Wireless Satell. Syst.*, 2019, pp. 576–586.

[19] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek, "High performance matrix inversion based on LU factorization for multicore architectures," in *Proc. ACM Int. Workshop Many Task Comput. Grids Supercomput.*, 2011, pp. 33–42.

[20] S. Donfack, J. Dongarra, M. Faverge, M. Gates, J. Kurzak, P. Luszczek, and I. Yamazaki, "A survey of recent developments in parallel implementations of Gaussian elimination," *Concurrency Comput., Pract. Exper.*, vol. 27, no. 5, pp. 1292–1309, Apr. 2015.

[21] S. Wunderlich, J. A. Cabrera, F. H. P. Fitzek, and M. Reisslein, "Network coding in heterogeneous multicore IoT nodes with DAG scheduling of parallel matrix block operations," *IEEE Internet Things J.*, vol. 4, no. 4, pp. 917–933, Aug. 2017.

[22] M. Kim, K. Park, and W. W. Ro, "Benefits of using parallelized non-progressive network coding," *J. Netw. Comput. Appl.*, vol. 36, no. 1, pp. 293–305, Jan. 2013.

[23] H. Shojania and B. Li, "Parallelized progressive network coding with hardware acceleration," in *Proc. IEEE Int. Workshop Qual. Service*, Jun. 2007, pp. 47–55.

[24] A. Garcia-Saavedra, M. Karzand, and D. J. Leith, "Low delay random linear coding and scheduling over multiple interfaces," *IEEE Trans. Mobile Comput.*, vol. 16, no. 11, pp. 3100–3114, Nov. 2017.

[25] M. Karzand, D. J. Leith, J. Cloud, and M. Médard, "Design of FEC for low delay in 5G," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 8, pp. 1783–1793, Aug. 2017.

[26] S. Pandi, F. Gabriel, J. A. Cabrera, S. Wunderlich, M. Reisslein, and F. H. Fitzek, "PACE: Redundancy engineering in RLNC for low-latency communication," *IEEE Access*, vol. 5, pp. 20477–20493, 2017.

[27] T. Ho, M. Médard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong, "A random linear network coding approach to multicast," *IEEE Trans. Inf. Theory*, vol. 52, no. 10, pp. 4413–4430, Oct. 2006.

[28] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, vol. 9. Philadelphia, PA, USA: SIAM, 1999.

[29] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 14, pp. 1–17, Mar. 1988.

[30] P. Luszczek, J. Kurzak, and J. Dongarra, "Looking back at dense linear algebra software," *J. Parallel Distrib. Comput.*, vol. 74, no. 7, pp. 2548–2560, Jul. 2014.

[31] S. M. Gunther, M. Riemensberger, and W. Utschick, "Efficient GF arithmetic for linear network coding using hardware SIMD extensions," in *Proc. IEEE Int. Symp. Netw. Coding (NetCod)*, Jun. 2014, pp. 1–6.

[32] J. S. Plank, K. M. Greenan, and E. L. Miller, "Screaming fast Galois field arithmetic using Intel SIMD instructions," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 299–306.

[33] H. Shin and J.-S. Park, "Optimizing random network coding for multimedia content distribution over smartphones," *Multimedia Tools Appl.*, vol. 76, pp. 19379–19395, Oct. 2017.

[34] A. Raza, A. A. Ikram, A. Amin, and A. J. Ikram, "A review of low cost and power efficient development boards for IoT applications," in *Proc. IEEE Future Technol. Conf. (FTC)*, Dec. 2016, pp. 786–790.

[35] P. A. Chou and Y. Wu, "Network coding for the Internet and wireless networks," *IEEE Signal Process. Mag.*, vol. 24, no. 5, pp. 77–85, Sep. 2007.

[36] S. Wunderlich, F. Gabriel, S. Pandi, F. H. Fitzek, and M. Reisslein, "Caterpillar RLNC (CRLNC): A practical finite sliding window RLNC approach," *IEEE Access*, vol. 5, pp. 20183–20197, 2017.

[37] F. Gabriel, S. Wunderlich, S. Pandi, F. H. Fitzek, and M. Reisslein, "Caterpillar RLNC with feedback (CRLNC-FB): Reducing delay in selective repeat ARQ through coding," *IEEE Access*, vol. 6, pp. 44787–44802, 2018.

[38] P. U. Tournoux, E. Lochin, J. Lacan, A. Bouabdallah, and V. Roca, "On-the-fly erasure coding for real-time video applications," *IEEE Trans. Multimedia*, vol. 13, no. 4, pp. 797–812, Aug. 2011.

[39] D. E. Lucani, M. Médard, and M. Stojanovic, "Systematic network coding for time-division duplexing," in *Proc. IEEE Int. Symp. Inf. Theory*, Jun. 2010, pp. 2403–2407.

[40] A. Douik and S. Sorour, "Data dissemination using instantly decodable binary codes in fog-radio access networks," *IEEE Trans. Commun.*, vol. 66, no. 5, pp. 2052–2064, May 2018.

[41] M. Nistor, D. E. Lucani, T. T. V. Vinhoza, R. A. Costa, and J. Barros, "On the delay distribution of random linear network coding," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 5, pp. 1084–1093, May 2011.

[42] J. Qureshi, C. H. Foh, and J. Cai, "Online XOR packet coding: Efficient single-hop wireless multicasting with low decoding delay," *Comput. Commun.*, vol. 39, pp. 65–77, Feb. 2014.

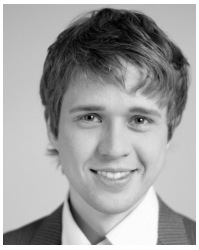
[43] H. Tang, Q. T. Sun, Z. Li, X. Yang, and K. Long, "Circular-shift linear network coding," *IEEE Trans. Inf. Theory*, vol. 65, no. 1, pp. 65–80, Jan. 2019.

[44] J. Heide and D. Lucani, "Composite extension finite fields for low overhead network coding: Telescopic codes," in *Proc. IEEE ICC*, Jun. 2015, pp. 4505–4510.

[45] S. Lee and W. W. Ro, "Accelerated network coding with dynamic stream decomposition on graphics processing unit," *Comput. J.*, vol. 55, no. 1, pp. 21–34, Jan. 2012.

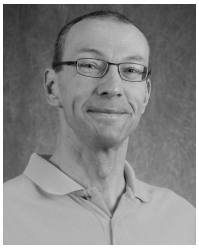
[46] J.-S. Park, S. J. Baek, and K. Lee, "A highly parallelized decoder for random network coding leveraging GPGPU," *Comput. J.*, vol. 57, no. 2, pp. 233–240, Feb. 2014.

[47] H. Shojania, B. Li, and X. Wang, "Nuclei: GPU-accelerated many-core network coding," in *Proc. IEEE INFOCOM*, Apr. 2009, pp. 459–467.
[48] D. Chen, J. Cong, S. Gurumani, W.-M. Hwu, K. Rupnow, and Z. Zhang, "Platform choices and design demands for IoT platforms: Cost, power, and performance tradeoffs," *IET Cyber-Phys. Syst., Theory Appl.*, vol. 1, no. 1, pp. 70–77, Dec. 2016.
[49] S. Mittal, "A survey of techniques for architecting and managing asymmetric multicore processors," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 45:1–45:38, Feb. 2016.
[50] S. J. Johnston, P. J. Basford, C. S. Perkins, H. Herry, F. P. Tso, D. Pezaros, R. D. Mullins, E. Yoneki, S. J. Cox, and J. Singer, "Commodity single board computer clusters and their applications," *Future Gener. Comput. Syst.*, vol. 89, pp. 201–212, Dec. 2018.
[51] G. Angelopoulos, A. Paidimarri, M. Médard, and A. P. Chandrakasan, "A random linear network coding accelerator in a 2.4 GHz transmitter for IoT applications," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 64, no. 9, pp. 2582–2590, Sep. 2017.
[52] S.-M. Choi, K. Lee, and J.-S. Park, "Fast parallel implementation for random network coding on embedded sensor nodes," *Int. J. Distrib. Sensor Netw.*, vol. 10, no. 2, 2014, Art. no. 974836.
[53] M. Kim and W. W. Ro, "Architectural investigation of matrix data layout on multicore processors," *Future Gener. Comput. Syst.*, vol. 37, pp. 64–75, Jul. 2014.
[54] B. Li and D. Niu, "Random network coding in peer-to-peer networks: From theory to practice," *Proc. IEEE*, vol. 99, no. 3, pp. 513–523, Mar. 2011.
[55] K. Park, J.-S. Park, and W. W. Ro, "On improving parallelized network coding with dynamic partitioning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 11, pp. 1547–1560, Nov. 2010.
[56] H. Shin and J.-S. Park, "Energy efficient QoS-aware random network coding on smartphones," *Mobile Netw. Appl.*, vol. 22, no. 5, pp. 880–893, Oct. 2017.
[57] H. Shin and J.-S. Park, "Reducing energy consumption of RNC based media streaming on smartphones via sampling," *Multimedia Tools Appl.*, vol. 78, no. 20, pp. 28461–28475, Oct. 2019.
[58] N. Melab, E.-G. Talbi, and S. Petiton, "A parallel adaptive Gauss-Jordan algorithm," *J. Supercomput.*, vol. 17, no. 2, pp. 167–185, Jan. 2000.
[59] C. Wang, L. Suo, D. Bian, J. Lv, and S. Tian, "Performance analysis and design of LT codes under the Gaussian elimination algorithm in wireless sensor networks," *IEEE Access*, vol. 7, pp. 55797–55806, 2019.
[60] J. Heide, M. V. Pedersen, F. H. P. Fitzek, and M. Medard, "A perpetual code for network coding," in *Proc. IEEE Veh. Technol. Conf. (VTC)*, May 2014, pp. 1–6.
[61] A. YarKhan, J. Kurzak, and J. Dongarra, "QUARK users' guide: Queuing and runtime for kernels," Innov. Comput. Lab., Univ. Tennessee, Knoxville, TN, USA, Tech. Rep. ICL-UT-11-02, Version 1.0, Apr. 2011.
[62] M. Taghouti, D. E. Lucani, J. A. Cabrera, M. Reisslein, M. V. Pedersen, and F. H. P. Fitzek, "Reduction of padding overhead for RLNC media distribution with variable size packets," *IEEE Trans. Broadcast.*, vol. 65, no. 3, pp. 558–576, Sep. 2019.
[63] J. Heide, M. V. Pedersen, F. H. Fitzek, and M. Médard, "On code parameters and coding vector representation for practical RLNC," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2011, pp. 1–5.
[64] M. V. Pedersen, J. Heide, and F. H. P. Fitzek, "Kodo: An open and research oriented network coding library," in *Proc. Netw. Workshops*. Berlin, Germany: Springer, 2011, pp. 145–152.
[65] H. P. Anvin. (2011). *The Mathematics of RAID-6*. Accessed: Jun. 15, 2019. [Online]. Available: https://mirrors.edge.kernel.org/pub/linux/kernel/people/hpa/raid6.pdf
[66] S. Brown, O. Johnson, and A. Tassi, "Reliability of broadcast communications under sparse random linear network coding," *IEEE Trans. Veh. Technol.*, vol. 67, no. 5, pp. 4677–4682, May 2018.
[67] Y. Li, W.-Y. Chan, and S. D. Blostein, "On design and efficient decoding of sparse random linear network codes," *IEEE Access*, vol. 5, pp. 17031–17044, 2017.
[68] Y. Li, J. Zhu, and Z. Bao, "Sparse random linear network coding with precoded band codes," *IEEE Commun. Lett.*, vol. 21, no. 3, pp. 480–483, Mar. 2017.
[69] Y. Li, J. Wang, S. Zhang, Z. Bao, and J. Wang, "Efficient coastal communications with sparse network coding," *IEEE Netw.*, vol. 32, no. 4, pp. 122–128, Jul./Aug. 2018.
[70] V. Nguyen, G. T. Nguyen, F. Gabriel, D. E. Lucani, and F. H. P. Fitzek, "Integrating sparsity into Fulcrum codes: Investigating throughput, complexity and overhead," in *Proc. IEEE Int. Conf. Commun. Workshops (ICC Workshops)*, May 2018, pp. 1–6.



SIMON WUNDERLICH received the Dipl.-Inf. degree in computer science from Chemnitz Technical University, Germany, in 2009. He is currently pursuing the Ph.D. degree in electrical engineering from the Technische Universität Dresden, Germany. He is a coauthor of the Wi-Fi Mesh software B. A. T. M. A. N. Advanced.

FRANK H. P. FITZEK received the Diploma (Dipl.Ing.) degree in electrical engineering from the University of Technology–Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen University, Aachen, Germany, in 1997, and the Ph.D. (Dr.Ing.) degree in electrical engineering from Technical University Berlin, Germany, in 2002. He became an Adjunct Professor with the University of Ferrara, Italy, in 2002. He is currently a Professor and the Head of the Deutsche Telekom Chair of Communication Networks, Technische Universität Dresden, Germany, where he is also coordinating the 5G Lab. He is the Spokesman of the DFG Cluster of Excellence CeTI. In 2003, he joined Aalborg University as an Associate Professor and later became a Professor. His current research interests include in the areas of wireless and 5G communication networks, network coding, cloud computing, compressed sensing, cross layer and energy-efficient protocol design, and cooperative networking. He was a recipient of the NOKIA Champion Award several times, from 2007 to 2011, the YRP Award for the work on MIMO MDC, in 2005, the Nokia Achievement Award for his work on cooperative networks, in 2008, the SAPERE AUDE Research Grant from the Danish Government, in 2011, the Vodafone Innovation Prize, in 2012, and the Young Elite Researcher Award of Denmark. He was also a recipient of the honorary degree Doctor Honoris Causa from the Budapest University of Technology and Economics (BUTE), in 2015.



MARTIN REISSLEIN (A'96–S'97–M'98–SM'03–F'14) received the Ph.D. in systems engineering from the University of Pennsylvania, in 1998. He is currently a Professor with the School of Electrical, Computer, and Energy Engineering, Arizona State University (ASU), Tempe, and an External Associated Investigator with the Centre for Tactile Internet with Human-in-the-Loop (CeTI), Technische Universität Dresden, Germany. He also serves as an Associate Editor for the IEEE TRANSACTIONS ON MOBILE COMPUTING, the IEEE TRANSACTIONS ON EDUCATION, IEEE ACCESS, as well as *Computer Networks*. He is an Associate Editor-in-Chief for the *IEEE Communications Surveys and Tutorials*, a Co-Editor-in-Chief for *Optical Switching and Networking*, and chairs the steering committee of the IEEE TRANSACTIONS ON MULTIMEDIA.