Network Coding in Heterogeneous Multicore IoT Nodes With DAG Scheduling of Parallel Matrix Block Operations

Simon Wunderlich, Juan A. Cabrera, Frank H. P. Fitzek, and Martin Reisslein, *Fellow, IEEE*

***Abstract*—Random linear network coding (RLNC) has the potential to improve the performance of current and future Internet of Things (IoT) communication systems, but is computationally demanding due to matrix multiplications and inversions. Some single-core RLNC implementations achieve already sufficient coding speeds for contemporary multimedia streaming formats. However, advances in multimedia streaming formats and IoT applications will require the exploitation of heterogeneous multicore architectures, which are becoming common for a wide range of IoT nodes, including smartphones. In this paper, we introduce and evaluate efficient RLNC computing strategies for IoT node architectures, including the emerging heterogeneous big.LITTLE multicore architectures with multiple big (fast) cores and multiple LITTLE (slow) cores. In contrast to existing RLNC implementation strategies, we build on and adapt highly optimized dense matrix operations from the high performance computing field to RLNC on heterogeneous multicore IoT nodes. Our approach includes the optimization of RLNC matrix operations through optimized operations on matrix blocks with single instruction multiple data instructions. We schedule block operations on the heterogeneous cores through a directed acyclic graph that avoids artificial synchronization points while ensuring the data dependencies. We examine priority scheduling according to the number of outgoing dependencies of a task and data locality of cached blocks. Our extensive measurements with several heterogeneous big.LITTLE multicore IoT node and smartphone processor boards demonstrate higher RLNC encoding and decoding throughputs than existing approaches. Moreover, our measurements indicate that the utilization of more cores decreases energy consumption, which is an important goal for IoT nodes.**

***Index Terms*—Directed acyclic graph (DAG), heterogeneous multicore architecture, Internet of Things (IoT) node, matrix inversion, matrix multiplication, parallel computing, random linear network coding (RLNC), smartphone.**

Manuscript received January 30, 2017; revised April 24, 2017; accepted May 9, 2017. Date of publication May 11, 2017; date of current version August 9, 2017. This work was supported in part by the German Research Foundation (DFG) in the Collaborative Research Center 912 Highly Adaptive Energy-Efficient Computing (HAEC) and in part by a DRESDEN Senior Fellowship. A preliminary form of the multicore approach appeared in [1]. *(Corresponding author: Frank H. P. Fitzek.)*

S. Wunderlich, J. A. Cabrera, and F. H. P. Fitzek are with the Deutsche Telekom Chair of Communication Networks, Technische Universitat Dresden, 01062 Dresden, Germany (e-mail: simon.wunderlich@mailbox.tu-dresden.de; juan.cabrera@tu-dresden.de; frank.fitzek@tu-dresden.de).

M. Reisslein is with the School of Electrical, Computer, and Energy Engineering, Arizona State University, Tempe, AZ 85287 USA (e-mail: reisslein@asu.edu).

Digital Object Identifier 10.1109/JIOT.2017.2703813

I. Introduction

RANDOM linear network coding (RLNC) [2], [3] is a popular coding approach for efficiently transferring data in complex, chaotic, or lossy networks, such as wireless networks [4], [5], which are the underlying network for many Internet of Things (IoT) settings [6]–[10]. RLNC can also be applied for data storage where data is scattered over different, possibly faulty storage servers [11]–[14], in multicast distribution networks [3], [15], and in peer-to-peer distribution networks [16]–[19]. RLNC divides the original data into symbols, combines the symbols linearly based on random coefficients, and distributes these coded symbols along with the coefficients. A receiver then only needs sufficiently many of these (linearly independent) coded symbols to solve the linear system posed by the random coefficients to decode the original data. This RLNC system relaxes the requirement for specific feedback to the sender or special topologies and structures, and thus can be readily applied in complex IoT structures or lossy IoT distribution channels. However, this simplicity comes at the expense of increased computation complexity to encode and decode the data in the IoT nodes.

Modern PCs, laptops, smartphones, and embedded systems, such as WiFi access points, TVs, and network attached storage systems, increasingly employ multiple heterogenous CPU cores. Smartphones and a wide range of embedded systems and sensors [20], [21] are envisioned to function as IoT communication nodes and to host IoT applications [22]. For instance, video streaming is an important form of IoT communication [23]–[27] and often involves smartphones, e.g., for video crowdsourcing [28]–[30]. Smartphones can host IoT event triggering [31] and cyber-physical IoT applications [32], including health related applications [33], [34]. Taking together the potential benefits of RLNC for IoT communication and the importance of smartphones and similar embedded systems as IoT communication nodes, it is highly important to effectively exploit heterogeneous multicore systems to increase network coding throughput while decreasing power consumption.

In this paper, we develop and evaluate parallelization strategies for speeding up RLNC on heterogeneous multicore architectures [35]–[38], which are well suited for IoT nodes [39]–[41]. We consider homogeneous multicore architectures with multiple identical CPU cores as well as heterogeneous big.LITTLE multicore architectures. The big.LITTLE architectures consist of a set of fast (*big*) CPU cores with relatively high power consumption, and a set of slow and low-power (LITTLE) CPU cores [35]–[38]. Modern smartphones and many other embedded systems feature these big.LITTLE architectures to support a wide range of IoT computing and communication applications [42]. We demonstrate that it is possible to exploit both types of cores at the same time to maximize throughput while minimizing power consumption.

Our general approach is to build on and to adapt multicore parallel computing principles from the field of high performance computing for RLNC on IoT nodes based on heterogeneous multicore architectures. We optimize matrix operations through processing matrices in blocks and optimization of basic (kernel) matrix operations through single instruction multiple data (SIMD) instructions which are commonly available on IoT node and smartphone processors. We schedule the matrix block processing tasks with a directed acyclic graph (DAG). In contrast to existing RLNC computing approaches, our novel DAG approach to RLNC does not require any artificial synchronization points. With DAG scheduling, the synchronization is only due to the data block dependencies. We examine schedule optimizations that consider the data dependencies, i.e., prioritize tasks with the most outgoing dependencies, and data locality, i.e., prioritize tasks with presently cached blocks. We conduct extensive evaluations with heterogeneous octacore systems, including the Samsung Exynos 5422 system on a chip (SoC), which is found in modern smartphones, such as the Samsung Galaxy S5, and is well suited for IoT nodes [22], [43], [44]. The evaluations indicate increased RLNC encoding and decoding throughputs compared to the state-of-the-art coefficient matrix duplication (CD) approach [45], [46].

This paper is structured as follows. Section II reviews RLNC basics. Section III presents related work on efficient RLNC computation. Section IV introduces the optimized matrix block operations for RLNC. Section V introduces the DAG-based approach for scheduling the block operation tasks. Sections VI and VII present the experimental evaluation set-up and the evaluation results, respectively. Section VIII summarizes the conclusions and outlines future work directions.

II. Network Coding Basics

RLNC performs linear operations in the Galois field GF(2^p) [3]. The sender splits the original data into n symbols. Each symbol has length (symbol size) m [in units of words, whereby for the typically considered GF(2^8), 1 word = 8 bit = 1 byte]. Consecutive symbols are further grouped into generations, whereby each generation contains g symbols, i.e., g is the generation size. The original data of one generation can then be described as matrix $\mathbf{M}$ with g rows and m columns, whereby each row represents one original symbol.

For encoding, we let r denote the number of redundant packets that are to be generated for a set of g packets, whereby r is set based on the expected losses. RLNC is a rateless coding mechanism and thus can create an arbitrary number r, $r \geq 0$, of redundant packets. A coefficient matrix $\mathbf{C}$ with $g + r$ rows and g columns with random coefficients is created for the encoding. Encoding is then performed by multiplying the coefficient matrix $\mathbf{C}$ with the original data (symbol) matrix $\mathbf{M}$

$$\mathbf{X} = \mathbf{CM}. \tag{1}$$

Each coded symbol x, which is a row of matrix $\mathbf{X}$, is then distributed along with its corresponding coefficient (row) vector c of matrix $\mathbf{C}$ as a coded packet. A receiver that has acquired at least g linearly independent coded packets forms a new coefficient matrix $\bar{\mathbf{C}}$ from the received coding vectors, and a new coded symbol matrix $\bar{\mathbf{X}}$. The receiver matrices $\bar{\mathbf{C}}$ and $\bar{\mathbf{X}}$ differ from the sender matrices $\mathbf{C}$ and $\mathbf{X}$ in row order and number of rows. To decode and reconstruct the original symbol matrix $\mathbf{M}$, the receiver calculates

$$\mathbf{M} = \bar{\mathbf{C}}^{-1}\bar{\mathbf{X}}. \tag{2}$$

The encoding and decoding operations share the matrix multiplication step. In addition, the decoding operation requires the inversion of the receiver coefficient matrix $\bar{\mathbf{C}}$.

Many applications implement decoding by combining the inversion and matrix multiplication, e.g., by performing the Gauss–Jordan algorithm [47] on matrix $\bar{\mathbf{C}}$, while applying row operations on matrix $\bar{\mathbf{X}}$. These row operations on $\bar{\mathbf{X}}$ can be parallelized for GPUs [48]. However, there is only limited GPU availability on resource-constrained lightweight IoT nodes and smartphones [49]. Also, the speed-up from the combined processing is limited for small symbol sizes ($m \leq 2048$ byte), since working with multiple threads on the same coded symbol requires tight synchronization. Our general strategy is therefore not to combine matrix inversion and multiplication. Instead, we first explicitly invert matrix $\bar{\mathbf{C}}$ with an optimized inversion technique. Then, we reuse the optimized matrix multiplication (from the encoding) to achieve overall very efficient decoding.

III. Related Work

This section gives an overview of the different existing research perspectives on the computationally efficient execution of network coding and contrasts the present study from related work on network coding on IoT nodes and smartphones. One research perspective has been to simplify network coding, e.g., by coding over the small binary GF(2) Galois field [50]–[53] or by developing alternative computationally simpler forms of network coding (see [54], [55]). In this paper, we focus on conventional RLNC over the commonly considered GF(2^8) Galois field.

For fast implementation of GF(2^8) RLNC, parallelization based on general purpose graphics processing units has been extensively studied [48], [56]–[62]. Desktop computers and servers have hundreds or even thousands of GPU threads available that amortize the data copying between CPU and GPU memory. However, for IoT nodes and smartphones with a relatively small number of GPU threads [49], the data copy overhead can typically not be amortized [46]. Therefore, GPU-based techniques are typically not suitable for IoT nodes and smartphones.

However, modern IoT nodes and smartphones feature multicore CPUs that can be exploited for parallelizing the matrix operations [47], [63]. Moreover, modern IoT node and smartphone CPUs feature SIMD instructions that can be exploited for parallel data processing [64]. More specifically, parallelized progressive network coding (PPNC) with SIMD instruction [65] statically partitions the coefficient matrix $\bar{\mathbf{C}}$ and the coded symbol matrix $\bar{\mathbf{X}}$ vertically to threads running in parallel on the multiple cores. For instance, for two parallel threads, the left half of $\bar{\mathbf{C}}$, i.e., the left $g/2$ columns of $\bar{\mathbf{C}}$, and the left half of $\bar{\mathbf{X}}$, i.e., the left $m/2$ columns of $\bar{\mathbf{X}}$, are fixed assigned to thread one, while the right halves of $\bar{\mathbf{C}}$ and $\bar{\mathbf{X}}$ are assigned to thread two. PPNC, as well as its existing refinements combine matrix inversion and multiplication to perform the decoding operation in (2). Matrix multiplication operations have been optimized for the iPhone in [66], while other network coding prototypes for smartphones have been explored in [67] and [68]. The microcast study [69] compared a pure Java implementation of network coding with a native implementation on the phone CPU.

The static partitioning of PPNC may lead to uneven thread loading, which has been addressed in dynamic vertical partitioning (DVP) [70]. DVP adapts the number of columns of $\bar{\mathbf{C}}$ and $\bar{\mathbf{X}}$ that are assigned to threads according to the number of rows that need to be processed. However, the parallel threads in both PPNC and DVP need to synchronously process the matrix partitions [65], [70]. Role division progressive decoding (RDPD) [71] has sought to solve this synchronization issue by defining a supervisor thread that focuses on the Gaussian elimination of $\bar{\mathbf{C}}$ while worker threads focus on the encoded data matrix $\bar{\mathbf{X}}$. RDPD thus still has dependencies, namely between worker threads and the supervisor thread. Recently, the CD approach [45], [46] has sought to circumvent these dependencies by providing each thread with a full copy of the coefficient matrix $\bar{\mathbf{C}}$. The CD approach builds on the previously developed partitioning approaches by assigning a vertical partition of the encoded data matrix $\bar{\mathbf{X}}$ to each core; however, different from prior approaches, the CD approach gives each core the full coefficient matrix $\bar{\mathbf{C}}$. The CD approach thus essentially decodes the coefficient matrix $\bar{\mathbf{C}}$ redundantly in each thread (core), eliminating the need for tight synchronization at the expense of duplicating the matrix inversion effort. The CD approach exploits the advanced SIMD extension of the ARM architecture, which is also known as NEON, and has demonstrated superior performance over previous approaches in the evaluations in [46]. We consider therefore the CD approach [46] as the main performance benchmark in our evaluations.

In contrast to the existing approaches, which combine matrix inversion and multiplication, we compute both steps individually. We employ matrix operation strategies that require minimal synchronization. This allows us to minimize the synchronization overhead while achieving excellent parallelization speed-up, without adding redundant computations.

IV. Optimized Matrix Block Operations for RLNC

A. General Principles

1) Review of Optimized Matrix Operations: Matrix multiplication and inversion are standard operations in many scientific applications and have therefore been extensively researched in the numerical computer science and high performance computing fields. Researchers in these fields have focused mainly on execution on powerful servers or supercomputers and developed optimized dense matrix computation principles [72]. Standard interfaces, such as the basic linear algebra subprograms (BLASs) [73], [74] and LAPACK [75] provide many common vector and matrix operations. Optimized and self-optimizing libraries, such as such as the library by Goto and van de Geijn [76] and ATLAS [77], take cache hierarchies, translation lookaside buffer (TLB), and SIMD instructions into account to maximize performance. However, these libraries operate on floating point or integer numbers, not on Galois fields (i.e., the libraries are not suitable for finite field arithmetic). Some libraries, such as FFLAS/FFPACK [78] and LinBox [79], leverage the highly optimized floating-point BLAS implementations for various finite field variants by converting finite field elements to floating point numbers and back.

2) Optimizing Matrix Operations for RLNC: This conversion approach from finite field elements to floating point numbers and back performs generally well for large problem sizes. However, the conversions add considerable overhead for small problem sizes. Also, native implementations can leverage SIMD instruction for performing many operations for "small" Galois field sizes, such as GF(2) and GF(2^8), simultaneously with one instruction. Our general strategy is therefore to build on the general principles of efficient floating point BLAS operations from the existing numerical computer science literature. Based on these principles, we independently develop computational strategies for small Galois fields to efficiently compute RLNC on IoT nodes and smartphones.

B. Block-Based Operation

1) Matrix Partitioning Into Blocks for Parallel Thread Processing: An important optimization is to let parallel threads operate on square block partitions of matrices rather than to process full rows and columns of a matrix, e.g., when multiplying matrices. We define a square block of a matrix (for brevity referred to as *block*) to consist of $b \times b$ words [bytes with the considered GF(2^8)], whereby typical block sizes are $b = 16$, 32, and 64 bytes. Working on square blocks improves the spatial locality of the data and maximizes the operations per fetched data, at least for $O(b^3)$ algorithms [80]. The optimal block size depends on the IoT node platform architecture [49] and is usually chosen so that all utilized parallel threads are busy and all operands fit into the L1 cache. Also, the block size b should be a multiple of the architecture's SIMD operation size. Further possible optimizations include multiple levels of block formation (blocking) to match L2 and L3 caches or the TLB, recursive blocking, and reordering the input data to adapt to the algorithm's access pattern [81]. For simplicity, we focus on a single level of blocking in this paper. (We introduce an additional level of blocking, referred to as subblocking, to improve cache efficiency in Section IV-C2; subblocking is not introduced to enhance parallel thread processing.) A limitation of our blocking-based approach is that

TABLE I
OVERVIEW OF GF(2^8) (BINARY 8) KERNEL (BASE) MATRIX BLOCK OPERATIONS FOR NETWORK CODING

Kernel	Description
B8_GEMM	Matrix multiplication of two matrices
B8_GETF2	LU factorization
B8_GETF2_TRAIL	Update panel (update part of GETF2)
B8_TRSM	Solve linear system for a triangular matrix
B8_TRMM	Matrix multiplication of a square and triangular matrix
B8_TRTI	Compute inverse of a triangular matrix
B8_LASET	Initialize a matrix with specific diagonal and non-diagonal elements
B8_LAPCY	Copy all or part of a matrix $\mathbf{A}$ to another matrix $\mathbf{B}$

all coded symbols need to be received before starting the decoding process; hence, our approach is not well suited for online decoding.

2) Matrix Inversion: For the matrix inversion, we developed a block-based (blocked) version of the lower upper (LU) factorization, which shares similarities with the LAPACK GETRI routine [75]. The three main stages of the LU factorization are as follows.

1) LU factoring of the input matrix $\mathbf{A}$.
2) Invert upper matrix $\mathbf{U}$.
3) Solve $\mathbf{A}^{-1}\mathbf{L} = \mathbf{U}^{-1}$ for $\mathbf{A}^{-1}$.

Each step involves various operations on matrix blocks, such as matrix–matrix multiplication, matrix-triangle matrix multiplication, and triangle-matrix system solving. We individually optimize these few base matrix operation "kernels," e.g., using SIMD operations.

C. Optimization of Kernel (Base) Operations

We represent each of the block matrix operations, e.g., matrix–matrix multiplication, matrix-triangle-matrix multiplication, and triangle-matrix system solution, as a base (kernel) operation. Each kernel operation can be represented as a task with input blocks and output blocks in memory. We have broken the problem into a small set of eight simple block operations in GF(2^8), see Table I, which are based on corresponding BLAS/LAPACK functions [73]–[75]. This small set of kernel operations makes it easy to examine which block operations are used frequently. The most frequently used block operations can then be optimized.

1) SIMD Operation Optimization for Matrix Multiplication: Matrix multiplication operations can be well optimized using SIMD operations. In particular, a matrix multiplication can be broken into multiple vector-scalar operations. These vector-scalar operations can be readily optimized with SIMD instructions, such as NEON for ARM processors, as has been studied in [46] and [82]. We employ the approach from [46] and [82] to implement the multiplication of a 16 byte wide vector [whereby each vector element has 1 byte in GF(2^8)] with a scalar. The vector elements are processed in parallel using the SIMD instructions `shift right`, `xor`, and `table lookup` (also known as `shuffle`). These SIMD instructions are widely available, e.g., in NEON on ARM and in the streaming SIMD extension 3 (SSSE3) within the x86 instruction set architecture, which is widely employed for IoT applications [83]. The implementation requires only seven SIMD instructions to compute the 16 multiplications for the 16 elements in a vector in parallel with a scalar [46, Sec. 3.3].

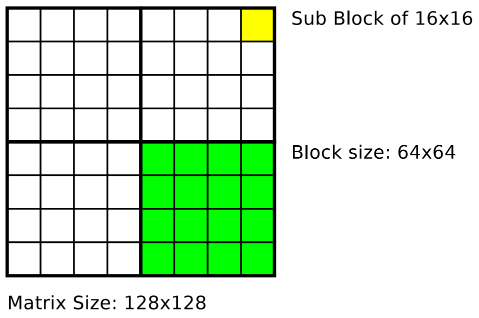


Fig. 1. Illustration of subblocking for a 128×128 byte matrix with block size $b = 64$, i.e., four 64×64 blocks for parallel processing on four threads (see Section IV-B1) and 16×16 subblocks for cache efficiency (see Section IV-C2).

2) Subblocking for Cache Efficiency: We further optimized the matrix multiplication through subblocking in the implementation of the kernel computation (computation kernel). We do not subblock to divide the workload into more threads (which is the purpose of the blocking in Section IV-B1). Instead, we seek to improve the cache efficiency of the computation kernel. That is, we seek to frequently reuse the data in the L1 cache, while meeting the width of the SIMD instructions. Specifically, for matrix multiplication, we divide the variable-sized block into 16×16 [bytes] subblocks that fit into the typical 32 KiB L1 data cache and match the width of the SIMD instructions of IoT nodes [22], [83]. For instance, in the example with 64×64 blocks illustrated in Fig. 1, each 64×64 block is divided into 16 subblocks of dimension 16×16. We perform the matrix kernel operations on the subblocks in order to perform a matrix multiplication on the entire block. This subblocking method enables efficient computations on large blocks, where the input and output blocks would not fit into the L1 cache of low-cost IoT nodes. Large blocks that do not fit into the L1 cache have led to performance degradations in earlier studies, e.g., [46], which have not considered subblocking for cache efficiency.

V. OPTIMIZED RLNC PARALLELIZATION WITH SCHEDULING GRAPHS

A. Directed Acyclic Graph Schedule

1) DAG Scheduling Principle: Each of the matrix block operations can be considered a separate task with inputs from memory and outputs to memory. Dongarra *et al.* [84] have described how to exploit these data dependencies among tasks to parallelize matrix inversion. The basic idea is to first formulate the algorithm conventionally. Data operations on blocks are then recorded, and the resolved data dependencies are formulated in a DAG. DAG nodes represent the individual tasks and DAG edges represent data dependencies among the nodes. To actually invert the matrix, a scheduler distributes tasks while meeting dependencies, until the DAG is completely processed.

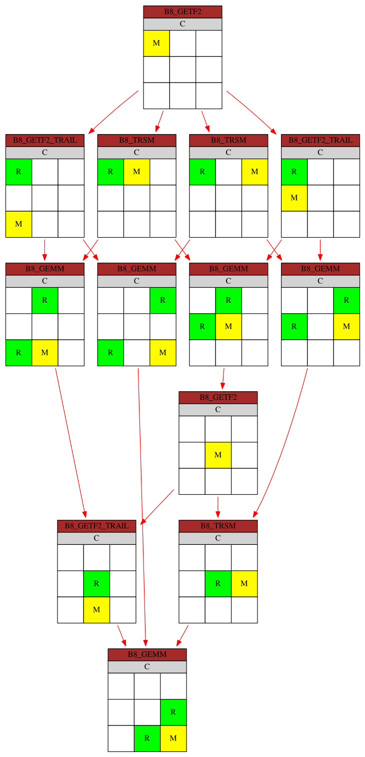


Fig. 2. Illustration of DAG of task dependencies of matrix inversion stage 1, LU factorization of the input matrix, for a 3×3 block matrix $\mathbf{C}$. Red arrows indicate read-after-write dependencies. Data is read from the green R blocks and both read from and written to the yellow B blocks. The headers of the illustrated blocks indicate the GF(2^8) kernel operations (see Table I).

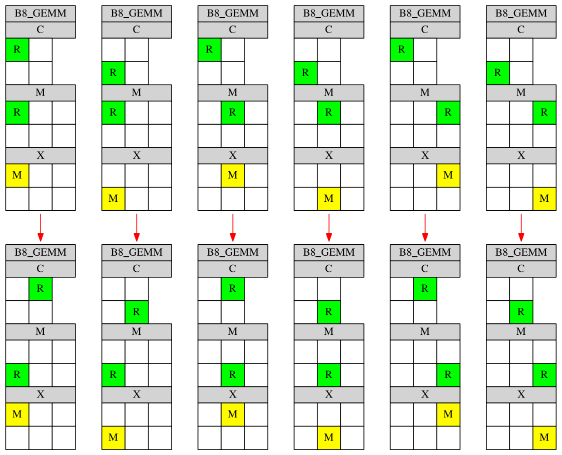


Fig. 3. DAG of task dependencies of multiplication of a square coefficient matrix $\mathbf{C}$ (1024×1204) with a rectangular symbol matrix $\mathbf{M}$ (1536×1024) at block size $b = 512$ to obtain coded symbol matrix $\mathbf{X}$.

2) DAG Schedule Creation (Recording): For our problem setting of RLNC encoding and decoding, we schedule the parallel matrix block multiplication and inversion operations as follows. We first record (create) the DAG schedule by simulating the execution of a single-thread program. The simulation does not execute the block operations (tasks); instead it records the input and output data of each block operation as well as the order of the block operation. Next, we explicitly find the dependencies between the tasks and prepare a DAG representation of the dependencies. We illustrate the DAG created for the first stage of the inversion (see Section IV-B2) of a 3×3 (block) matrix in Fig. 2. Fig. 3 illustrates the DAG for a matrix multiplication. For each node, the involved blocks have been colored: green blocks with letter "R" are blocks from which data is read, red blocks with letter "W" are written to, and both operations are performed on yellow blocks with letter "B." Green arrows indicate write-after-read dependencies, while red arrows indicate read-after-write dependencies. We observe from Fig. 3 that the matrix multiplication DAG provides a very high degree of parallelism to exploit. This is because all blocks of the output matrix are computed independently through the various matrix–matrix multiplications of the respective blocks, as recorded in the DAG.

The final step of the DAG schedule creation places the task(s) from the DAG that do not have any dependencies, i.e., the first task(s) into a "task queue." The task queue is a simple linked list of tasks that are ready, i.e., do not depend on other tasks to be finished first.

3) DAG Schedule Execution: The execution of a prerecorded schedule proceeds as follows. We first start up multiple threads. Then, each thread accesses the task queue to obtain a new task to work on. After a task has been processed by a core, the list of tasks that depend on the just completed task is traversed. For each of those outgoing dependent tasks, a dependency counter is decreased, whereby the dependency counter denotes the number of dependencies that have to be met before the task can be scheduled. When the dependency counter of a task reaches zero, all dependencies have been cleared, and the task is added to the beginning (head) of the task queue. The rationale for adding the task to the head of the queue, i.e., for the last-in-first-out/stack scheduling policy, is to improve the data locality for the first-task scheduler (see Section V-C1), since the same core will immediately try to pick up the next task.

The execution is completed when the task queue is empty and all cores have returned to their idle state. In a practical implementation of an IoT application, the schedule can be prerecorded during an offline computation. The prerecorded schedule can then be executed in an online manner by copying the DAG graph for each invocation of a generation. Accordingly, we define a schedule and execute (SE) throughput measure (for online SE) and an execute throughput measure (for offline scheduling and online execution), see Section VI-C.

4) DAG Schedule Properties: This DAG-based method has important properties: first, the synchronization depends only on data dependencies, *no artificial* synchronization points need to be inserted. More specifically, parallel programs typically require artificial synchronization points, where threads are created and joined after a particular processing step. In contrast, with the DAG approach, the synchronization depends only on data dependencies. No artificial synchronization points are needed, not even between the different steps defined in Section IV-B2. Rather, threads are created on startup of the matrix operation, and can pick a task with satisfied dependencies at any time. This dynamic processing with low

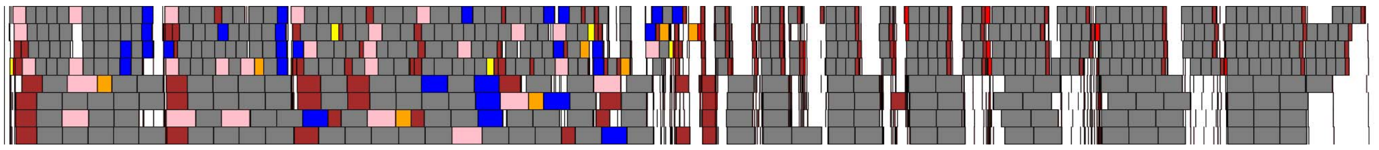


Fig. 4. Recorded example schedule for a matrix inversion on big.LITTLE IoT node architecture with four big (fast) cores on top and four LITTLE (slow) cores on the bottom, for generation size $g = 1024$, block size $b = 128$: white color patches indicate idle times, and red color patches indicate overhead time, e.g., for waiting for a dependency. Through the schedule tuning strategies described in Section V-B, we reduced the white and red color patches.

synchronicity allows threads that run on slow cores or threads that take breaks (e.g., for processing I/O, other system tasks, or other IoT applications) to contribute to the computation without creating idle time on other cores.

Second, the task creation overhead is minimized as the scheduler creates threads based on the number of used cores on startup, whereby a thread is pinned to each core. Keeping threads open is substantially faster than creating new threads, since the operating system overhead can be avoided and threads only need to work with the light-weight data structures of the task queue. Finally, this DAG approach can cope very well with heterogeneous cores running at different speeds, e.g., big.LITTLE heterogeneous multicore IoT node architectures [39]–[41], [49], and systems where some cores may be busy with other tasks, e.g., input/output processing or processing of the received data.

B. Schedule Tuning

For evaluating and tuning our scheduler, we tracked the execution times and created a graphical representation to find hotspots and problems in the overall execution. An example run on the ODROID XU-3 big.LITTLE octacore system (see Section VI-A1) is shown in Fig. 4. The x-axis represents time, and each row shows one core, whereby the four fast cores are on top and the four slow cores on the bottom. White patches represent idle time, bright red patches represent overhead time (e.g., for acquiring a new task or for waiting for a lock), while other colors denote different block operations.

1) Data Chunk Splitting: The graphical representation of the recorded schedule revealed long running tasks that operated on "big data chunks" consisting of multiple blocks, e.g., on a 16×128 data chunk. By splitting the big data chunk into smaller chunks (whereby a smaller chunk could consist of a single block or multiple blocks) we reduced the idle times of the other processors. We note that the data chunk splitting is orthogonal to the kernel optimization in Section IV-C in that more tasks on smaller inputs/output chunks/blocks are created, but the kernel computation is still the same and can be optimized separately.

2) Adapt Block Processing Order to Avoid Cache Line Conflicts: One limitation of the block size selection is the cache line size of the processor architecture (platform). On many contemporary IoT platforms, including the evaluated platforms (see Section VI-A), the L1 cache line size is 64 bytes [49], [83]. Column-major order is the standard format for storing matrix data in FORTRAN, in particular in the BLAS/LAPACK [73]–[75] subprograms. We built on the principles of the mathematical kernels and algorithms

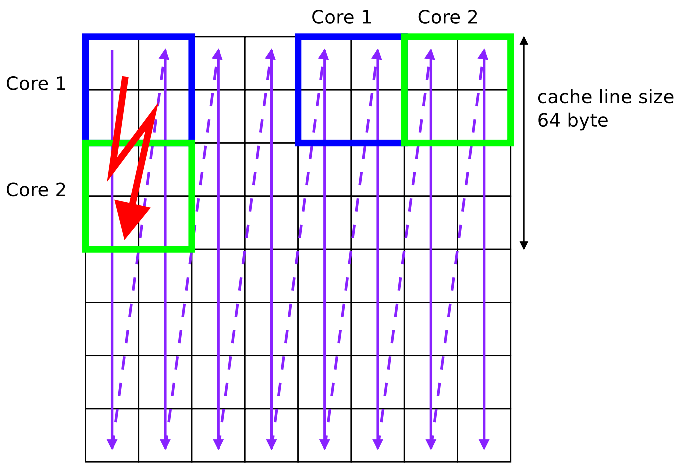


Fig. 5. Illustration of cache line conflicts for two cores working on the same cache line in the column-major data layout and conflict resolution through adapting the block processing order: a black square denotes an SIMD/subblock of 16×16 bytes. The 32×32 block computed by core 1 is indicated by the blue square, while the green square indicates the block computed by core 2. In the upper left, the two cores operate on the same 64-byte cache lines (which hold columns of four SIMD/subblocks), creating conflicts. In the upper right, the two cores operate on blocks that are held in different columns, i.e., different cache lines, avoiding the conflict.

from BLAS/LAPACK and therefore employ the prerequisite column-major order. An alternative approach could employ row-major order; however, the row-major order would require extensive redesign and reprogramming of mathematical kernels and algorithms.

As illustrated in Fig. 5, using block sizes of $b = 16$ or 32 [byte] could result in having multiple cores work on separate blocks that occupy the same cache lines. The two 32×32 blocks in the upper left of Fig. 5 are computed by cores 1 and 2, indicated by blue and green squares, respectively. In the employed column-major format, cores 1 and 2 will need to work on the same (vertical) 64 byte cache lines. This will result in synchronization among the L1 caches of the two cores, increasing the computation latency through synchronization stalls ("cache thrashing") [85], [86]. This cache thrashing can significantly slow down the matrix operations as a cache coherency protocol will need to stall a processor core until the data has been synchronized with the other processor core that is currently "in charge" of the cache line.

We avoided this cache thrashing problem for the matrix multiplication through adapting the processing order of the blocks. Specifically, we first loop over row blocks, as illustrated in the upper right of Fig. 5. That is, the blocks that cores 1 and 2 are computing are arranged horizontally in the upper right of Fig. 5. Cores 1 and 2 work now on blocks in the same row (instead of the same column). In the employed column-major format, the two cores thus work on different cache lines, avoiding cache thrashing and cache coherency stalls. Accordingly, we have constructed the loops to first sweep through the row blocks of the result matrix containing the product of the matrix multiplication, i.e., $\mathbf{X}$ in (1) or $\mathbf{M}$ in (2).

The looping of row blocks implies that for matrix multiplication, the cores typically operate simultaneously on blocks in the same row. However, some instances of cores working on the same (vertical) cache lines can still arise. One example instance is that the cores "wrap around" during the looping over the block rows. For example, suppose four cores are operating on a matrix that is three blocks wide. Then, the fourth core would already work on the next row (wrapped around) and can conflict with the first core operating on the first block in the preceding row. Another instance can arise through "online" conflicts, e.g., when one core has been busy with another task and returns to operate on a block while another core has advanced to operating on a block directly below the block with the returning core.

An alternative approach for avoiding the cache thrashing is to reorder the data into "tiles" so that all elements of one block are in a contiguous buffer [87]. However, this tile-based approach adds reordering overhead before and after each coding operation. This reordering overhead can be quite significant, especially for small generation sizes g. In particular, matrix multiplication touches each element in the result matrix g times; thus for small g with relatively low multiplication effort, the reordering overhead can be quite significant. Another alternative approach could be to add hints for the scheduler to avoid scheduling the same cache lines on different processors.

C. Priority Scheduling: Data Dependencies and Locality

1) First Task Scheduling: An elementary "first task" scheduling policy schedules the first task in the task queue (see Section V-A2) for the next available processor, irrespective of whether it is a fast or slow processor. In addition to this elementary first task scheduling, we examine priority scheduling according to task dependencies and data locality and a combination thereof, as introduced next. Throughout, all scheduling policies give fast cores precedence over slow cores. More specifically, each of the considered scheduling policies instructs slow cores to go to sleep and wake fast cores, when a slow core (that has just finished a task) finds a fast core waiting for new tasks.

2) Task Dependency Scheduling (Priority for Tasks With Most Outgoing Dependencies): We observed from the graphical representation of the execution times, as illustrated in Fig. 4, in conjunction with graphical illustrations of the task dependencies, as illustrated in Figs. 2 and 3 that tasks with many outgoing dependencies have the tendency to hold up (delay) many subsequent tasks. We have addressed this delay due to many outgoing dependencies through prioritizing tasks with the most outgoing dependencies. More specifically, our scheduler parses the list of queued open tasks. For each open task, the scheduler counts the number of outgoing dependencies. The scheduler then schedules the queued task with the most outgoing dependencies on the next available fast core. On the other hand, the queued task with the fewest outgoing dependencies is scheduled for the next available slow core.

3) Data Locality Scheduling (Priority for Tasks Operating on Cached Blocks): A related scheduling optimization is to prioritize the tasks that operate on a block that is presently in the cache so as to exploit data locality. In particular, the scheduler parses the list of queued open tasks for tasks that operate on a presently cached block. More specifically, we remember the last call processed on a given core. For each open task in the queue, we compare each input/output block with the input/output blocks of the task that has just completed its processing on the considered core. We award a score of one for each match. For example, if both tasks work on exactly the same blocks and we have two input blocks and one output block, as is common, we award a score of three. The scheduler then selects the task with the highest score for the next available core (irrespective of whether it is a fast or slow core).

4) Combined Priority for Task Dependency and Data Locality: Our combined priority policy distinguishes between the slow and fast cores. The combined priority score of the slow cores is computed as the score from the data locality scheduling policy (see Section V-C3) minus the dependency count from task dependency scheduling (see Section V-C2). Subtracting the dependency count from the data locality score encourages the slow cores to select less "important," but still local tasks. In contrast, for the fast cores, the combined priority policy adds the dependency count from task dependency scheduling (see Section V-C2) and the locality score from data locality scheduling (see Section V-C3) to obtain a combined priority score. Thus, important tasks with several dependencies, are assigned with priority to the fast cores. Throughout, we select the task with the highest score for the considered (slow or fast) core.

VI. Evaluation Setup

A. IoT Node Boards

1) ODROID XU-3: We have conducted experimental evaluations mainly with the ODROID XU-3 board [88]. The ODROID XU-3 board features the Samsung Exynos 5422 SoC, which is widely considered as basis for IoT nodes [22], [43], [44]. The Samsung Exynos 5422 SoC has a (big) Cortex-A15 quad core CPU clocked at up to 2.0 GHz and a (LITTLE) Cortex-A7 quad core CPU clocked at up to 1.4 GHz. The Samsung Exynos 5422 SoC implements the big.LITTLE architecture with heterogeneous multiprocessing (HMP). The HMP can simultaneously use all eight cores, compared to previous big.LITTLE systems that could only utilize combinations of up to four cores out of a total of eight cores. Each core has a 32 KiB (KiB = 2^{10} byte) L1 data cache and a 32 KiB L1 instruction cache, which is organized as 2-way (Cortex A15 [89]) or 4-way (Cortex A7 [90]) set-associative cache with a fixed cache line length of 64 bytes. The four A-15 cores share a 2 MiB (MiB = 2^{20} byte) L2 cache, while the A7 cores share a 512 KiB L2 cache. Both quad core CPUs are connected to each other and to a 2 GiB LPDDR3 RAM clocked at 933 MHz with an 128-bit AMBA ACE Coherent Bus interface. All cores support the NEON extension with the 128-bit SIMD instruction set, which can greatly speed up the GF(2^8) operations [82], [91].

2) ODROID XU+E: For benchmark comparison with the recent evaluation results for the CD approach [45], [46], we also conducted evaluations with the ODROID XU board. The ODROID XU board is identical to the ODROID XU+E board used for the evaluations in [45] and [46], except for the missing energy measurement components. The ODROID XU+E board is similar to our main ODROID XU-3 evaluation board. However, the ODROID XU+E board has an earlier version of the (big) Cortex-A15 quad core CPU clocked at up to 1.6 GHz. While the ODROID XU+E has also a big.LITTLE architecture with an Cortex-A7 quad core (LITTLE) CPU, it does not yet have HMP; thus, only four cores out of the total of eight cores can be utilized at a time. The ODROID XU+E has also a slightly slower RAM clocked at 800 MHz.

3) Cubieboard 4: The Cubieboard, which is a widely considered IoT node platform [92], [93], is implemented around an AllWinner A80 SoC, which contains a (big) Cortex-A15 quad core CPU clocked at up to 2.0 GHz and a (LITTLE) Cortex-A7 quad core CPU clocked at up to 1.3 GHz. The big.LITTLE architecture supports HMP. The 2 GB DDR3 memory is clocked at 480 MHz, i.e., substantially slower than the ODROID-XU3. The cache configuration is identical to the ODROID-XU3 with separate 32 KiB L1 caches for instructions and data on each core and separate 2 MiB and 512 KiB L2 caches for the A15 and A7 cores, respectively.

4) Raspberry Pi 2: For comparison with our earlier evaluations [1] we conducted measurements on the Raspberry Pi 2 Model B board, which features four ARM Cortex-A7 cores in a Broadcom BCM2836 SoC, clocked at 900 MHz. Each core features a 32 KiB L1 data cache and a 32 KiB L1 instruction cache. All cores share a 512 KiB L2 cache. The Raspberry Pi 2 also features the NEON SIMD instructions, but has only four homogenous cores.

B. Parameter Settings and Test Matrices

We conduct evaluations for a wide range of generation sizes g and symbol sizes m. Specifically, we consider symbol sizes ranging from $m = 1024$ and 1536 bytes, which are multiples of 512 close to the standard Ethernet maximum transfer unit, to $m = 16\,384$ bytes, which would be applicable for data centers and storage usage. We randomly fill the data (symbol) matrix and the coefficient matrix and consider matrices that do not require pivoting. As is common for the evaluation of network coding computations, we set the number r of redundant packets to zero. We performed test runs for all available block sizes b. We report results for the best performing block size, which would be employed in a real IoT implementation. All benchmark results presented in this paper have been performed with NEON-enabled code using optimized GF(2^8) operations which have been adopted from the fifi/kodo library [94]. We initially employ the first task scheduling policy (see Section V-C1). We conducted independent replications for each evaluation so that the widths of the 95% confidence intervals of the performance metrics were less than 2% of the corresponding sample means. The confidence intervals are not plotted so as to avoid visual clutter.

C. Throughput Metrics

As reviewed in Section II, encoding requires a matrix multiplication, while decoding requires one matrix inversion and one matrix multiplication. For a given combination of generation size g, symbol size m, and block size b, we measured the execution (run) time [s] for creating and executing the schedule for the matrix multiplication for encoding, resp. the matrix inversion and multiplication for decoding. We define the respective encoding and decoding throughput metrics [bytes/s] as the measured execution times divided by the size gm [bytes] of the data (symbol) matrix $\mathbf{M}$.

More specifically, for both the encoding throughput and the decoding throughput, we provide two throughput measures: 1) the throughput for creating the schedule and executing the schedule, referred to as the *SE throughput* and 2) the throughput for executing a precomputed schedule, referred to as the *E throughput*. When employing our DAG scheduling-based encoding or decoding approach in practice, the DAG schedule needs to be computed (recorded) only once for a given parameter combination, i.e., a given combination of generation size g, symbol size m, and block size b. Subsequent encoding or decoding operations can utilize the exact same DAG schedule, i.e., only need to execute the DAG schedule. Thus, the first encoding or decoding for a particular combination of generation size g, symbol size m, and block size b needs to compute the DAG schedule and execute the DAG schedule, i.e., achieves the SE throughput. Subsequent encoding or decoding operations for the same parameter combination only need to execute the prerecorded DAG schedule and thus achieve the higher E throughput. We note that execution of a prerecorded DAG schedule incurs overhead for copying and reinitializing the DAG schedule. However, this overhead is typically negligible.

D. Energy Metric

The ODROID XU-3 board features integrated power sensors that individually measure the currents and voltages of the big A15 cores, the little A7 cores, the GPU, and the RAM. For our power evaluations, we have combined all four sensors to form one overall power value [Watts]. We have repeatedly sampled the power sensors, near the beginning and end of the execution and every 5 ms during the execution. We integrated the sampled power values over time to obtain the consumed energy [Joules] for a given encoding or decoding task.

VII. Evaluation Results

Initially, we examine the impact of the block size b on the throughput performance in Section VII-A. We then proceed to investigate the implications of operating multiple threads on heterogeneous multicore systems by contrasting the throughout levels for a single thread (see Section VII-B) with the throughput levels achieved for multiple threads (see Section VII-C) on the ODROID XU3 board. We then

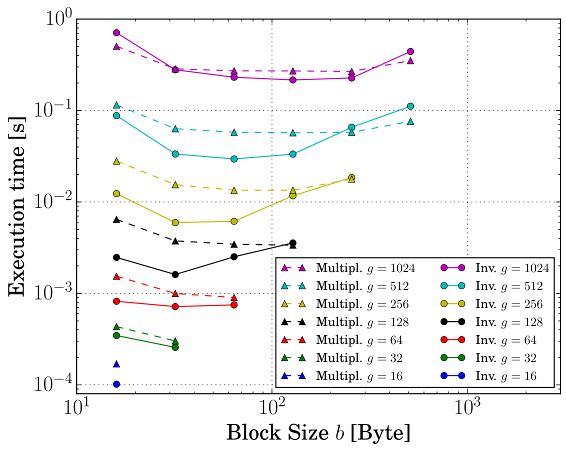


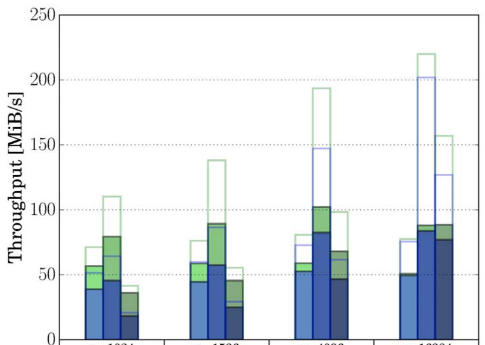
Fig. 6. Average matrix multiplication and inversion execution times [s] as a function of block size b [bytes] for range of generation sizes g. Fixed parameters: ODROID XU3 board with four utilized threads, symbol size $m = 1536$.

compare the decoding throughput of our approach based on the DAG scheduling of parallel matrix block with the CD approach of Shin and Park [46] for the ODROID XU board in Section VII-D. In Section VII-E, we compare the throughput performance of our DAG-based approach for the different boards described in Section VI-A. Consumed energy is reported in Section VII-F for the ODROID XU3 board while the different scheduling policies (see Section V-C) are evaluated in Section VII-G.

A. Block Size b Tradeoffs

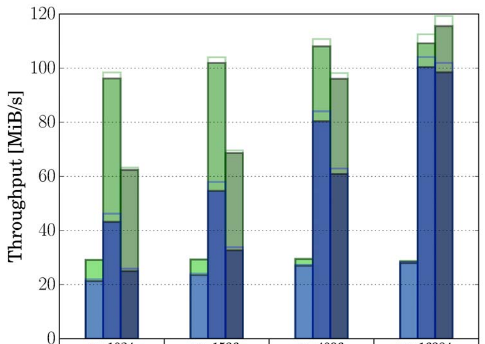
In Fig. 6, we plot the average execution times for the matrix inversion and the matrix multiplication as a function of the block size b for the ODROID XU3 board with four utilized threads for a symbol size of $m = 1536$. The results for the other boards and symbol sizes are very similar and are not included due to space constraints. We focus initially on the multiplication times for a large generation size of $g = 512$ or 1024 in Fig. 6. We observe that the multiplication time initially decreases as the block size b increases. The multiplication time then reaches a minimum for medium block sizes, e.g., for $b = 128$ for $g = 512$. Further increases in the block size b lead to an increase in the multiplication time. This behavior is mainly due to a tradeoff between the level of parallelism and the cache efficiency. Small block sizes lead to many tasks that can be executed in parallel on the four utilized threads. However, loading many small blocks into the cache creates inefficiencies. Thus, the block size b that provides a ratio of generation size g to block size b of four—so as to have a sufficient level of parallelism for the four cores that are typical for IoT nodes [49], [83], [93]—achieves generally the shortest multiplication times. We found in additional evaluations that when utilizing only one thread, larger block sizes generally result in shorter multiplication times. When utilizing only one thread, there is no parallelism to exploit; thus, cache efficiency governs the overall performance.

We note that the block sizes b achieving the shortest multiplication times in Fig. 6 may exceed the L1



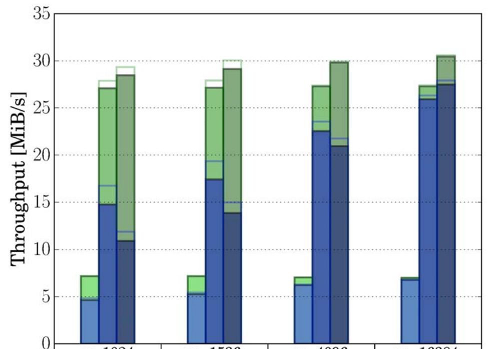
	m=1024	m=1536	m=4096	m=16384
Enc1	56.81 / 71.13	58.63 / 76.16	58.82 / 80.68	50.75 / 77.51
Enc4	79.28 / 110.30	89.32 / 138.11	102.22 / 193.55	88.04 / 219.91
Enc8	36.13 / 41.49	45.57 / 55.35	68.03 / 98.34	88.49 / 156.93
Dec1	38.92 / 51.45	44.54 / 59.83	52.56 / 72.79	49.48 / 75.55
Dec4	45.57 / 64.25	57.42 / 86.41	82.54 / 147.25	83.74 / 201.88
Dec8	18.29 / 20.72	25.04 / 29.26	46.63 / 61.69	76.99 / 126.86

(a)



	m=1024	m=1536	m=4096	m=16384
Enc1	29.06 / 29.26	29.26 / 29.43	29.44 / 29.64	28.58 / 28.80
Enc4	96.15 / 98.43	101.91 / 103.98	108.04 / 110.76	109.16 / 112.50
Enc8	62.38 / 63.21	68.66 / 69.60	96.03 / 98.17	115.53 / 119.26
Dec1	21.30 / 22.00	23.51 / 24.10	26.95 / 27.35	27.95 / 28.23
Dec4	43.20 / 46.21	54.61 / 57.90	80.37 / 84.04	100.43 / 104.09
Dec8	24.96 / 25.87	32.69 / 33.79	60.90 / 62.91	98.45 / 101.91

(b)



	m=1024	m=1536	m=4096	m=16384
Enc1	7.16 / 7.22	7.17 / 7.22	7.04 / 7.05	7.01 / 7.01
Enc4	27.09 / 27.87	27.15 / 27.92	27.29 / 27.40	27.29 / 27.40
Enc8	28.47 / 29.33	29.15 / 30.04	29.82 / 29.94	30.46 / 30.60
Dec1	4.65 / 4.83	5.27 / 5.43	6.22 / 6.29	6.78 / 6.81
Dec4	14.75 / 16.75	17.41 / 19.35	22.54 / 23.56	25.93 / 26.33
Dec8	10.88 / 11.86	13.86 / 14.97	20.96 / 21.76	27.49 / 27.92

(c)

Fig. 7. Encoding (Enc) and decoding (Dec) throughputs [MiB/s = 2^{20} Byte/s] for using 1, 4, and 8 threads of the ODROID XU3 board for a range of combinations of generation size g and symbol size m. The schedule creating and executing (SE) throughput is represented by the solid bars, while the bar outlines indicate the execution (E) throughput. (a) $g = 16$. (b) $g = 64$. (c) $g = 256$.

cache size of IoT nodes. The matrix multiplication uses the B8_GEMM kernel (see Table I), which employs subblocking (see Section IV-C2). The subblocking ensures high cache efficiency for the large block sizes that exceed the L1 cache size.

Turning to the inversion times in Fig. 6, we observe that the shortest inversion time is generally achieved for the block size b that corresponds to half the block size that achieves the shortest multiplication time. For instance, for generation size $g = 512$, the block size $b = 128$ achieves the shortest multiplication time, while $b = 64$ achieves the shortest inversion time. Aside from the B8_GEMM kernel which employs subblocking, the inversion requires other time consuming kernels, such as B8_trsm and B8_trmm (for which we have not yet implemented subblocking; these subblocking implementation are left as future work). Therefore, the inversion incurs some inefficiencies when the block size exceeds the L1 cache size and performs generally better for smaller block sizes b compared to multiplication.

Throughout the remainder of this paper, we employ the respective block sizes achieving the shortest matrix inversion and multiplication times. That is, for a given decoding scenario, we employ the block size achieving the shortest inversion time during the matrix inversion step and then the block size achieving the shortest multiplication time during the matrix multiplication step.

B. Throughput for Single Thread

For a single thread (i.e., the "Enc1" and "Dec1" results), we observe from Fig. 7 that the throughput levels generally decrease with increasing generation size g. This trend continues for generation sizes $g > 256$, which we cannot include due to space constraints. From additional evaluations, we observed that the throughput level for a single thread drops to around 3.5 MiB/s for $g = 512$ and to around 1.5 MiB/s for $g = 1024$. In order to explain the dropping throughput for increasing g, we focus initially on the encoding. The encoding throughput is proportional to the ratio of encoding data size $g \cdot m$ to the matrix multiplication computational complexity of order $O(g^2 \cdot m)$ for multiplying the $g \times g$ matrix $\mathbf{C}$ with the $g \times m$ matrix $\mathbf{M}$ (1), that is,

$$\text{Encoding Throughput} \sim \frac{g \cdot m}{g^2 \cdot m} = \frac{1}{g}. \tag{3}$$

Thus, the encoding throughput is essentially independent of the symbol size m, and inversely proportional to the generation size g. Intuitively, each byte in a symbol (row) of the original data (payload) matrix $\mathbf{M}$ has to be combined g times with the corresponding bytes of the other symbols. We note that smaller generation sizes g require more frequent reinitialization of the DAG; however, the DAG reinitialization overhead is typically negligible compared to the actual matrix multiplication computations. Decoding additionally incurs the computational complexity $O(g^3)$ for the inversion of the coefficient matrix $\bar{\mathbf{C}}$, further reducing the decoding throughput for increasing generation size g. Generally, large generation sizes g are beneficial for some IoT application scenarios, e.g., for compensating "bursty" losses, i.e., when several successive packets are lost. Also, large g decrease the likelihood of linearly dependent packets [95]. On the other hand, small g are preferable for low-latency IoT communication [96]–[98].

We also observe from Fig. 7 that for a given generation size g, the encoding throughput stays nearly constant for increasing symbol size m. This nearly constant encoding throughput is in agreement with (3). The very slight increase of the encoding throughput with larger symbol size m in Fig. 7 is due to the increase of parallelization opportunities, which permit the hiding of overheads. In contrast, we observe from Fig. 7 that the decoding throughput for a single thread (Dec1) for a given generation size g increases for increasing symbol size m. This is because the decoding involves the matrix inversion, which has computational complexity $O(g^3)$ followed by the matrix multiplication with complexity $O(g^2 \cdot m)$. Thus, the decoding throughput is proportional to

$$\text{Decoding Throughput} \sim \frac{g \cdot m}{O\big(g^3\big) + O\big(g^2 \cdot m\big)}. \tag{4}$$

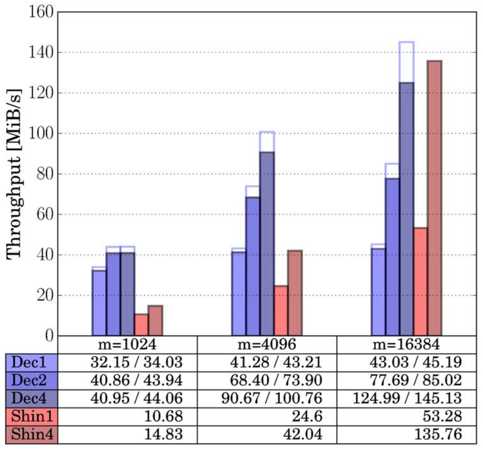
With increasing symbol size m, the impact of the inversion complexity $O(g^3)$ is reduced relative to the multiplication complexity $O(g^2 \cdot m)$, leading to a decoding throughput approaching the encoding throughput [see (3)] for large symbol size m.

Another important observation from Fig. 7 for one thread is that for the small generation size $g = 16$, the E throughput for executing a prerecorded schedule is significantly higher than the SE throughput for creating (recording) and executing the schedule. This throughput increase achieved for executing a prerecorded schedule becomes more pronounced for increasing thread numbers and will be examined in detail in the next section.

C. Throughput Increases for Multiple Threads

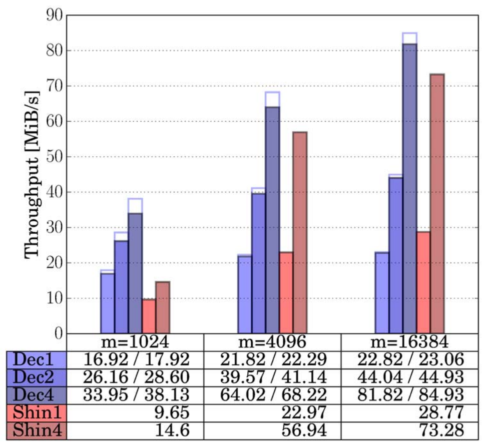
In this section, we examine the throughput increases achieved by utilizing multiple threads (cores) of the examined big.LITTLE IoT node architecture. We first estimate the theoretical peak performance with eight heterogeneous cores, by evaluating the execution times with a single thread on a big core and on a little core, respectively. For this evaluation, we set the generation size to $g = 1024$, the symbol size to $m = 4096$, and the block size to $b = 64$ in order to test a configuration with plentiful parallelization opportunities. We found that the big cores were on average 3.64 times faster than the little cores for the multiplication, and 3.27 times faster for the inversion. Therefore, a peak speed-up of 5.1 to 5.22 can be expected when all four big cores and all four little cores are employed (relative to employing a single big core).

We observe from Fig. 7 that using four threads generally achieves speed-ups over using a single thread; except for decoding for $g = 16$ with $m = 1024$ and 1536, which is slowed by the additional cores. For the small symbol size $m = 1024$ or 1536, the matrix inversion effort is still quite significant [it diminishes for large m, see (4)]. The small generation size $g = 16$ matches the minimum considered block size $b = 16$. Thus, there is only a single block available for processing the inversion of the receiver coefficient matrix $\bar{\mathbf{C}}$, which has dimension 16×16 for the considered $g = 16$ and $r = 0$. Thus, for the small $g = 16$ and small $m = 1024$ or 1536 scenarios, the matrix inversion, which accounts for a significant portion of the overall decoding effort, cannot benefit from the multiple threads. However, the operation of multiple



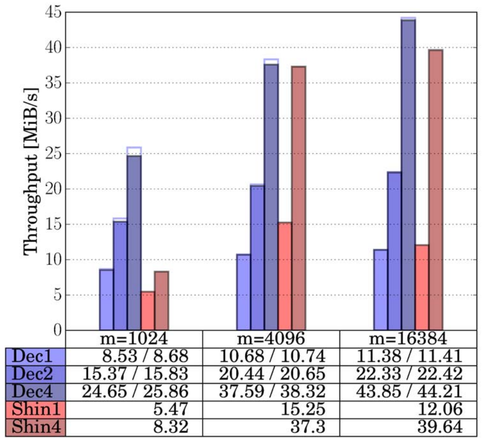
	m=1024	m=4096	m=16384
Dec1	32.15 / 34.03	41.28 / 43.21	43.03 / 45.19
Dec2	40.86 / 43.94	68.40 / 73.90	77.69 / 85.02
Dec4	40.95 / 44.06	90.67 / 100.76	124.99 / 145.13
Shin1	10.68	24.6	53.28
Shin4	14.83	42.04	135.76

(a)



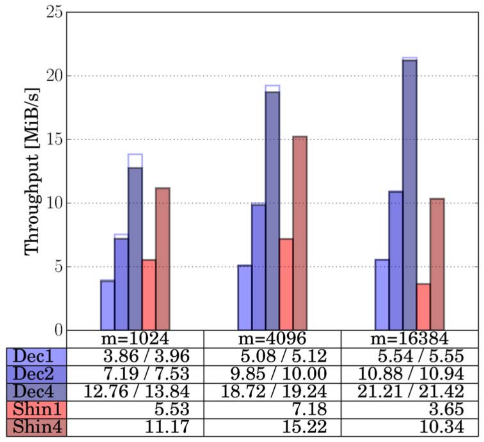
	m=1024	m=4096	m=16384
Dec1	16.92 / 17.92	21.82 / 22.29	22.82 / 23.06
Dec2	26.16 / 28.60	39.57 / 41.14	44.04 / 44.93
Dec4	33.95 / 38.13	64.02 / 68.22	81.82 / 84.93
Shin1	9.65	22.97	28.77
Shin4	14.6	56.94	73.28

(b)



	m=1024	m=4096	m=16384
Dec1	8.53 / 8.68	10.68 / 10.74	11.38 / 11.41
Dec2	15.37 / 15.83	20.44 / 20.65	22.33 / 22.42
Dec4	24.65 / 25.86	37.59 / 38.32	43.85 / 44.21
Shin1	5.47	15.25	12.06
Shin4	8.32	37.3	39.64

(c)



	m=1024	m=4096	m=16384
Dec1	3.86 / 3.96	5.08 / 5.12	5.54 / 5.55
Dec2	7.19 / 7.53	9.85 / 10.00	10.88 / 10.94
Dec4	12.76 / 13.84	18.72 / 19.24	21.21 / 21.42
Shin1	5.53	7.18	3.65
Shin4	11.17	15.22	10.34

(d)

Fig. 8. Comparison of decoding throughputs (SE as solid bars, E as outlines) of our DAG-based approach (Dec) with Shin and Park [46] approach (Shin) for 1 to 4 threads on ODROID XU board for different combinations of generation size g and symbol size m. (a) $g = 32$. (b) $g = 64$. (c) $g = 128$. (d) $g = 256$.

threads incurs some overhead. Thus, the decoding is overall slightly slowed down when employing multiple threads for the small $g = 16$ and small $m = 1024$ or 1536 scenarios.

We observe from Fig. 7 that the speed-ups are particularly pronounced for encoding moderate to large generation sizes g, whereby the speed-up factors are consistently around three or somewhat higher for the entire range of symbol sizes m. In contrast, for decoding, the speed-up factor increases with increasing symbol size m. This is because matrix inversion has generally more data dependencies compared to matrix multiplication, as illustrated by the DAGs for a matrix inversion step in Fig. 2 and a matrix multiplication in Fig. 3. Thus, matrix multiplication is more amenable to parallelization on multiple threads than matrix inversion. With increasing symbol size m, the matrix multiplication accounts for increasingly larger portions of the decoding effort [see (4)]. Thus, our parallelization-based approach achieves increasing decoding throughputs for increasing symbol size m.

We also observe from Fig. 7 that using eight threads achieves speed-ups over using four threads only for moderate to large generation sizes in combination with large symbol sizes m. The throughput reduction when going from four to eight threads for the small symbol sizes and small generation sizes is partly due to cache effects. In particular, each set of cores (big and LITTLE) has its own layer 2 cache, which needs synchronization when altered. Moreover, when employing eight cores, there is chance that some important tasks (with relatively many outgoing dependencies) are scheduled on a slow (LITTLE) core. Then, fast (big) cores may have to wait for the slow core to complete the task. Also, operating the eight cores increases the overall overhead compared to operating only four cores. Large symbol sizes m and generation sizes g offer many parallelization opportunities. The plentiful parallelization opportunities amortize the additional overheads and potential slow downs due to using eight cores. Thus, for large symbol sizes m and generation sizes g, the eight cores can be productively employed to enhance the encoding and decoding throughputs.

Importantly, we observe from Fig. 7 that our DAG-based network coding approach achieves high execution (E) to SE throughput ratios for the small generation size $g = 16$ and moderate to large symbol sizes m. For $m = 16\,384$, the E throughputs for both encoding and decoding are more than twice the corresponding SE throughputs. The computational effort for the schedule creation, which is single threaded, depends mainly on the number of tasks, because the scheduling has to consider the dependencies between the tasks. The number of tasks, in turn depends mainly on the ratio of the generation size g and symbol size m to the block size b. Generally, small blocks imply many tasks, and therefore large scheduling overhead. Thus, for the small blocks which correspond to small generation sizes, see Section VII-A, large E to SE throughput speed-ups are achieved.

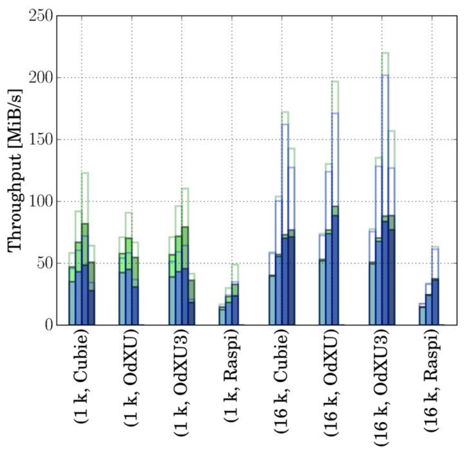
We note that small generation sizes g are generally preferred for low-latency IoT networking scenarios. Thus, our DAG-based approach with the high E throughput for small g appears particularly well suited for low-latency IoT and smartphone communication [96]–[98].

D. Comparison With Shin and Park [46]

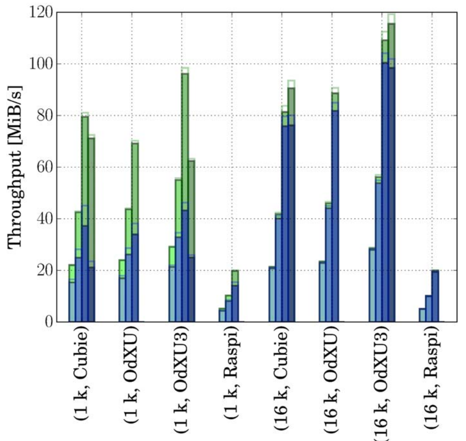
In Fig. 8, we compare the decoding throughput results of the CD approach by Shin and Park [46] (marked by “Shin” in Fig. 8) with our approach. The CD approach has been shown in [46] to achieve the highest throughput levels of all existing network coding approaches suitable for IoT and smartphone settings. We observe from Fig. 8 that our approach achieves higher throughputs than the CD approach for scenarios with small generation sizes g in combination with small to moderately large symbol sizes m as well as for scenarios with large generation sizes g in combination with large symbol sizes m. We also observe that for the multithread decoding with four cores, which is highly relevant in IoT practice, our approach achieves higher throughput in all examined scenarios.

These decoding throughput differences between our DAG-based approach and the CD approach of Shin and Park [46] are due to several differences in the computing dynamics. First, the CD approach appears to have a relatively high static overhead that becomes particularly apparent for small generation sizes g. More specifically, we expect from (3) that doubling the generation size g cuts the throughput (as governed by the matrix multiplication effort) in half. Indeed, our Dec1 results for a single thread in Fig. 8(a) and (b) clearly illustrate this “halving” trend. In contrast, for symbol sizes $m = 1024$ and 4096, the Shin1 decoding throughputs for a single thread remain nearly constant as the generation size is increased from $g = 32$ [Fig. 8(a)] to $g = 64$ [Fig. 8(b)]. This indicates that the Shin1 decoding throughputs for small g appear to be governed by a high static overhead that dominates over the computational complexity of the matrix multiplication (which is inversely proportional to g). Our DAG approach avoids such a static overhead and achieves for small generation sizes $g = 32$ and 64 for the small symbol size $m = 1024$, which is realistic for IoT packet networks, roughly twice the decoding throughput of the CD approach.

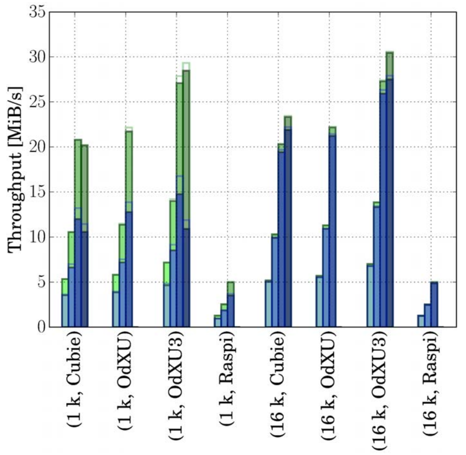
On the other hand, we observe from Fig. 8(a) and (b) that for the large symbol size $m = 16\,384$, the CD throughput with a single thread (Shin1), is higher than our single-thread throughput (Dec1). Similarly, we observe from Fig. 8(c) and (d) that for the moderately large symbol size $m = 4096$ (as well as for $m = 1024$ for $g = 256$), Shin1 exceeds Dec1. This performance advantage of the CD approach for single-thread computation appears to be due to a more efficient implementation of the matrix multiplication which dominates the computing dynamics for scenarios with moderately large combinations of generation and symbol sizes. The CD approach divides the data matrix $\bar{\mathbf{X}}$ into vertical partitions



(a)



(b)



(c)

Fig. 9. IoT board comparison: encoding throughput (green bars) and decoding throughput (blue bars), SE as solid bars and E as outlines, of our DAG-based approach for 1, 2, and 4 (and 8 when available) threads for various generation sizes g and symbol sizes $m = 1024$ and $16\,384$. (a) $g = 16$. (b) $g = 64$. (c) $g = 256$.

(see [46, Fig. 4]). Once a thread has started to work on a partition, it can complete the processing of the partition without any further overhead, which is efficient for multiplication processing of large partitions that still benefit from caching. In contrast, we divide the data into blocks of a prescribed size (that is smaller than the data partitions in the CD approach in the considered scenarios). A given thread then retrieves tasks based on these blocks. The task retrieval introduces some overhead, and this overhead is incurred multiple times on the thread.

However, we observe from Fig. 8(d) that for the large generation size ($g = 256$), the CD throughput drops when increasing the symbol size from $m = 4096$ to $m = 16\,384$. This performance drop is due to missing cache efficiency [46]. In contrast, our DAG-based approach consistently achieves throughput increases for increasing symbol size m. These throughput increases indicate that our optimized block operations with subblocking (see Section IV-C2) are highly cache efficient.

We further observe from Fig. 8 that our speed-ups from one to four threads are generally higher than for the CD approach, especially for large generation sizes g. The CD approach duplicates the full coefficient matrix $\bar{\mathbf{C}}$ on each thread; thus each thread has to work with the full coefficient matrix $\bar{\mathbf{C}}$. In contrast, our DAG-based approach employs optimized block operations throughout. Thus, each core works only on blocks (subsets) of the full coefficient and coded symbol matrices, achieving high degrees of parallelism.

Overall, we conclude from the comparison in Fig. 8 that our DAG-based approach with optimized block operations performs particularly well for small generation sizes g, which are preferable for low-latency IoT communication [96]–[98]. On the other hand, the CD approach has favorable performance for mid-level generation and symbol sizes on a single thread. Also, we note that the CD approach is more readily amenable to online or "on-the-fly" decoding when packets trickle in, whereas our approach requires a complete generation for decoding. For multithread computation (as well as for large generation and symbol sizes on a single thread) our DAG approach gives higher decoding throughout than the CD approach due to the cache efficiency of our optimized block operations and the consistent parallelization of the matrix operations.

E. Comparison of IoT Boards

In Fig. 9, we compare the SE and E encoding and decoding throughputs of our DAG-based approach for different IoT boards [49], [83], [93]. Note that the Cubieboard and the ODROID XU3 board feature HMP and thus can utilize up to eight threads, whereas the ODROID XU and Raspberry Pi2 boards do not feature HMP and thus can utilize at most four threads in parallel. We observe from Fig. 9(a) that our DAG-based approach achieves execution (E) encoding and decoding throughputs for $g = 16$ and $m = 16\,384$ above 150 MiB/s when four cores are utilized on the Cubie, ODROID XU, and ODROID XU3 boards. For generation sizes $g = 64$ and 256, we observe that the small symbol size $m = 1024$ does generally not benefit from the HMP feature. However, for the large symbol size $m = 16\,384$, the HMP boards, i.e., Cubieboard and ODROID XU3, achieve higher throughputs with eight threads than with four. For instance, for $g = 256$ and $m = 16\,384$, the ODROID XU3 achieves decoding throughputs of 27.91 MiB/s with eight threads and 26.32 MiB/s with four threads; the corresponding encoding throughputs are 30.59 MiB/s with eight threads and 27.4 MiB/s with four threads. These throughput increases with eight threads indicate that our DAG-based approach can effectively utilize the heterogeneous big.LITTLE IoT node architectures to achieve significant throughput enhancements.

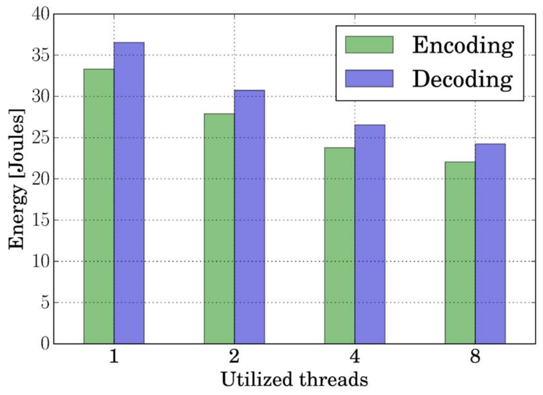


Fig. 10. Consumed energy [Joules] for encoding or decoding of a generation size $g = 1024$ with symbol size $m = 16\,384$ [bytes] utilizing 1, 2, 4, or 8 threads on the ODROID XU3 board.

TABLE II
MATRIX INVERSION EXECUTION TIMES [MS] FOR DIFFERENT PRIORITY SCHEDULING POLICIES FOR GENERATION SIZE $g = 512$ AND BLOCK SIZE $b = 64$ FOR 1, 2, 4, AND 8 UTILIZED THREADS ON ODROID XU3

Utilized Threads	First Task	Task Depend.	Data Local.	Comb. Prior.
1	103.93	101.86	101.58	101.66
2	54.37	52.98	52.47	52.37
4	29.52	28.86	28.06	28.29
8	34.45	32.16	31.90	32.15

F. Energy Consumption

Fig. 10 shows the consumed energy for the scheduling and execution (SE performance) of the encoding or decoding of a generation of size $g = 1024$ with $m = 16\,384$ [byte] symbols. We plot results for utilizing 1, 2, 4, or 8 threads (cores) of the ODROID XU3 board. We observe from Fig. 10 that the consumed energy drops as the number of utilized threads increases. In particular, Fig. 10 indicates that completing the same encoding or decoding task with eight threads instead of a single thread requires about 34% less energy When operating a smartphone board, various static components, such as memory, GPU, and bus systems, require a constant power supply. Thus, using multiple threads so as to complete the task faster, reduces the energy consumed during the task completion. Overall, the energy consumption results in Fig. 10 indicate that our parallelization approach is energy efficient, which is important for many IoT settings [39], [41], [49], [83], [99]. Our parallelization approach is able to reduce the consumed energy when more threads are utilized.

G. Evaluation of Priority Scheduling: Data Dependencies and Locality

In this section, we examine the scheduling policies described in Section V-C for the ODROID XU3 board. Table II presents execution times for the matrix inversion for generation size $g = 512$ for block size $b = 64$ for different numbers of utilized threads. We observe from Table II that for the relatively large generation size of $g = 512$, data locality scheduling achieved generally the shortest execution times. However, the execution time reductions are quite small; for instance, for four and eight threads, data locality scheduling reduced the execution times only by 5% and 7%, respectively, compared to first task scheduling. Also, data locality scheduling gives typically very similar execution times as combined priority scheduling and these two scheduling policies achieve only very minor time reductions compared to task dependency scheduling.

In additional evaluations that are not included due to space constraints we observed that for very small block sizes b, the first task scheduling gives the shortest execution times. For instance, for block size $b = 16$ for generation size $g = 512$, the other policies require roughly 10%–20% more time than first task scheduling. For small block sizes, it does not pay off to be very careful in selecting the next task, as each task is processed quickly, plus scanning through the large number of queued tasks requires significant effort. On the other hand, for large blocks, the processing of a correspondingly large task can take a relatively long time, plus there are only relatively few large tasks to consider.

In summary, we have found that data locality scheduling and combined priority scheduling achieve some performance enhancements for scenarios with moderately large to large block sizes b (and moderately few to few blocks in the task queue). We note that we only examined the greedy, one-sweep heuristic priority scheduling policies introduced in Section V-C. The observed performance degradations with the simple priority scheduling for small block sizes indicate that policies for scheduling matrix block operations on IoT node and smartphone cores must have very low complexity. Thus, more complex scheduling policies, e.g., policies that consider the tasks executing on all cores or consider multiple processing steps into the future, are likely not suited for the relatively small sized blocks that are processed on smartphones (relative to the blocks processed on high performance computers).

VIII. Conclusion

We have investigated efficient RLNC for heterogeneous multicore architectures, which are likely to be widely employed in IoT communications and applications. Building on dense linear algebra computing principles from numerical computer science, we have developed efficient block-based matrix inversion and multiplication strategies. We schedule matrix block processing tasks with a DAG. The DAG scheduling avoids artificial synchronization and only considers the inherent data computing dependencies. Our extensive experimental evaluations with heterogeneous big.LITTLE multicore IoT node systems indicate that our approach yields excellent speed-ups for various generation and symbol sizes; high speed-ups are achieved even for small symbol sizes which have proven problematic in earlier studies, e.g., [56], [65], and [100], and are especially interesting for low-latency IoT network applications [96]–[98]. Our evaluations also indicate that our DAG-based scheduling of matrix block operations on multiple parallel threads (cores) outperforms the CD approach [45], [46], which is the fastest previously known network coding approach for IoT nodes and smartphones. We have further demonstrated that using all cores can significantly decrease the energy consumption, which is important for battery powered IoT nodes and smartphones [39], [41], [49], [83]. Our approach is applicable to heterogeneous multicore computing architectures in smartphones and other IoT nodes.

There are many directions for important future research on efficient network coding computation for heterogeneous multicore architectures. The present study has examined the DAG scheduling of parallel block operations without artificial synchronization, i.e., loose synchronization, on heterogeneous multicore architectures with a shared memory architecture. An interesting future research directions is to investigate the loose synchronization approach for distributed memory architectures. Moreover, the RLNC techniques developed in this paper for IoT nodes and smartphones without usage of GPUs, could be extended in future research for other processors types, e.g., for processors with abundant GPU availability, whereby the GPUs could perform some of the tasks defined in this paper. Finally, the data dependency scheduling approach is not limited to RLNC and matrix problems, but could be investigated in the context of many other problems that involve bulk data processing with minimal branching in the processing flow.

References

[1] S. Wunderlich, J. Cabrera, F. H. P. Fitzek, and M. V. Pedersen, "Network coding parallelization based on matrix operations for multicore architectures," in *Proc. IEEE Int. Conf. Ubiquitous Wireless Broadband (ICUWB)*, Montreal, QC, Canada, Oct. 2015, pp. 1–5.

[2] T. Ho, R. Koetter, M. Médard, D. R. Karger, and M. Effros, "The benefits of coding over routing in a randomized setting," in *Proc. IEEE ISIT*, Yokohama, Japan, 2003, p. 442.

[3] T. Ho *et al.*, "A random linear network coding approach to multicast," *IEEE Trans. Inf. Theory*, vol. 52, no. 10, pp. 4413–4430, Oct. 2006.

[4] P. Pahlevani *et al.*, "Novel concepts for device-to-device communication using network coding," *IEEE Commun. Mag.*, vol. 52, no. 4, pp. 32–39, Apr. 2014.

[5] H. Zhang, K. Sun, Q. Huang, Y. Wen, and D. Wu, "FUN coding: Design and analysis," *IEEE/ACM Trans. Netw.*, vol. 24, no. 6, pp. 3340–3353, Dec. 2016.

[6] L. Bondi, L. Baroffio, M. Cesana, A. Redondi, and M. Tagliasacchi, "EZ-VSN: An open-source and flexible framework for visual sensor networks," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 767–778, Oct. 2016.

[7] X. Huang and N. Ansari, "Content caching and distribution in smart grid enabled wireless networks," *IEEE Internet Things J.*, vol. 4, no. 2, pp. 513–520, Apr. 2017.

[8] B. W. Khoueiry and M. R. Soleymani, "A novel machine-to-machine communication strategy using rateless coding for the Internet of Things," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 937–950, Dec. 2016.

[9] A. Nessa and M. Kadoch, "Joint network channel fountain schemes for machine-type communications over LTE-advanced," *IEEE Internet Things J.*, vol. 3, no. 3, pp. 418–427, Jun. 2016.

[10] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of Things for smart cities," *IEEE Internet Things J.*, vol. 1, no. 1, pp. 22–32, Feb. 2014.

[11] C. Fragouli, J.-Y. Le Boudec, and J. Widmer, "Network coding: An instant primer," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, pp. 63–68, Jan. 2006.

[12] N. Kumar, S. Zeadally, and J. J. P. C. Rodrigues, "QoS-aware hierarchical Web caching scheme for online video streaming applications in Internet-based vehicular ad hoc networks," *IEEE Trans. Ind. Electron.*, vol. 62, no. 12, pp. 7892–7900, Dec. 2015.

[13] T. H. Luan, L. X. Cai, J. Chen, X. S. Shen, and F. Bai, "Engineering a distributed infrastructure for large-scale cost-effective content dissemination over urban vehicular networks," *IEEE Trans. Veh. Technol.*, vol. 63, no. 3, pp. 1419–1435, Mar. 2014.

[14] J. Qiao, Y. He, and X. S. Shen, "Proactive caching for mobile video streaming in millimeter wave 5G networks," *IEEE Trans. Wireless Commun.*, vol. 15, no. 10, pp. 7187–7198, Oct. 2016.

[15] S.-C. Lin and K.-C. Chen, "Statistical QoS control of network coded multipath routing in large cognitive machine-to-machine networks," *IEEE Internet Things J.*, vol. 3, no. 4, pp. 619–627, Aug. 2016.

[16] U. Lee *et al.*, "Efficient peer-to-peer file sharing using network coding in MANET," *J. Commun. Netw.*, vol. 10, no. 4, pp. 422–429, Dec. 2008.

[17] 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.

[18] E. Magli, M. Wang, P. Frossard, and A. Markopoulou, "Network coding meets multimedia: A review," *IEEE Trans. Multimedia*, vol. 15, no. 5, pp. 1195–1212, Aug. 2013.

[19] D. Niu and B. Li, "Analyzing the resilience-complexity tradeoff of network coding in dynamic P2P networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 11, pp. 1842–1850, Nov. 2011.

[20] A. Seema and M. Reisslein, "Towards efficient wireless video sensor networks: A survey of existing node architectures and proposal for a flexi-WVSNP design," *IEEE Commun. Surveys Tuts.*, vol. 13, no. 3, pp. 462–486, 3rd Quart., 2011.

[21] B. Tavli, K. Bicakci, R. Zilan, and J. M. Barcelo-Ordinas, "A survey of visual sensor network platforms," *Multimedia Tools Appl.*, vol. 60, no. 3, pp. 689–726, Oct. 2012.

[22] S. J. Johnston, M. Apetroaie-Cristea, M. Scott, and S. J. Cox, "Applicability of commodity, low cost, single board computers for Internet of Things devices," in *Proc. World Forum Internet Things*, Reston, VA, USA, 2016, pp. 141–146.

[23] I. Bisio, F. Lavagetto, and G. Luzzati, "Cooperative application layer joint video coding in the Internet of Remote Things," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 1418–1426, Dec. 2016.

[24] N. Kumar, J. J. P. C. Rodrigues, and N. Chilamkurti, "Bayesian coalition game as-a-service for content distribution in Internet of Vehicles," *IEEE Internet Things J.*, vol. 1, no. 6, pp. 544–555, Dec. 2014.

[25] N. Kumar, S. Misra, J. J. P. C. Rodrigues, and M. S. Obaidat, "Coalition games for spatio-temporal big data in Internet of Vehicles environment: A comparative analysis," *IEEE Internet Things J.*, vol. 2, no. 4, pp. 310–320, Aug. 2015.

[26] X. Li *et al.*, "Smart community: An Internet of Things application," *IEEE Commun. Mag.*, vol. 49, no. 11, pp. 68–75, Nov. 2011.

[27] K. Sun, H. Zhang, D. Wu, and H. Zhuang, "MPC-based delay-aware fountain codes for real-time video communication," *IEEE Internet Things J.*, to be published.

[28] Q. He, J. Liu, C. Wang, and B. Li, "Coping with heterogeneous video contributors and viewers in crowdsourced live streaming: A cloud-based approach," *IEEE Trans. Multimedia*, vol. 18, no. 5, pp. 916–928, May 2016.

[29] H.-J. Hong, C.-L. Fan, Y.-C. Lin, and C.-H. Hsu, "Optimizing cloud-based video crowdsensing," *IEEE Internet Things J.*, vol. 3, no. 3, pp. 299–313, Jun. 2016.

[30] G. Zhuo, Q. Jia, L. Guo, M. Li, and P. Li, "Privacy-preserving verifiable set operation in big data for cloud-assisted mobile crowdsourcing," *IEEE Internet Things J.*, vol. 4, no. 2, pp. 572–582, Apr. 2017.

[31] P. Kolios, C. Panayiotou, G. Ellinas, and M. Polycarpou, "Data-driven event triggering for IoT applications," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 1146–1158, Dec. 2016.

[32] W. He and P.-H. Ho, "On achieving cyber-physical real-time snapshot acquisition in billboard/signage networks," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 1213–1221, Dec. 2016.

[33] N. Kumar, K. Kaur, A. Jindal, and J. J. P. C. Rodrigues, "Providing healthcare services on-the-fly using multi-player cooperation game theory in Internet of Vehicles (IoV) environment," *Digit. Commun. Netw.*, vol. 1, no. 3, pp. 191–203, Aug. 2015.

[34] B. M. C. Silva, J. J. P. C. Rodrigues, I. de la Torre Díez, M. López-Coronado, and K. Saleem, "Mobile-health: A review of current state in 2015," *J. Biomed. Inf.*, vol. 56, pp. 265–272, Aug. 2015.

[35] T. Hruby, H. Bos, and A. S. Tanenbaum, "When slower is faster: On heterogeneous multicores for reliable systems," in *Proc. USENIX Annu. Techn. Conf.*, San Jose, CA, USA, 2013, pp. 255–266.

[36] Y. Li, W. Shi, C. Jiang, J. Zhang, and J. Wan, "Energy efficiency analysis of heterogeneous platforms: Early experiences," in *Proc. IEEE Int. Green Sustain. Comput. Conf. (IGSC)*, Hangzhou, China, 2016, pp. 1–6.

[37] S. Mittal, "A survey of techniques for architecting and managing asymmetric multicore processors," *ACM Comput. Surveys*, vol. 48, no. 3, pp. 1–38, Feb. 2016.

[38] S. Srinivasan, L. Zhao, R. Illikkal, and R. Iyer, "Efficient interaction between OS and architecture in heterogeneous platforms," *ACM SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 62–72, Jan. 2011.

[39] P. Bogdan, M. Pajic, P. P. Pande, and V. Raghunathan, "Making the Internet-of-Things a reality: From smart models, sensing and actuation to energy-efficient architectures," in *Proc. IEEE ACM IFIP Int. Conf. Hardw. Softw. Codesign Syst. Synth.*, Pittsburgh, PA, USA, 2016, pp. 1–10.

[40] Y. Nikolakopoulos *et al.*, "Highly concurrent stream synchronization in many-core embedded systems," in *Proc. ACM Int. Workshop Many Core Embedded Syst.*, Seoul, South Korea, 2016, pp. 2–9.

[41] C. Tan *et al.*, "LOCUS: Low-power customizable many-core architecture for wearables," in *Proc. ACM Int. Conf. Compilers Archit. Synth. Embedded Syst.*, Pittsburgh, PA, USA, 2016, pp. 1–10.

[42] A. Kamilaris and A. Pitsillides, "Mobile phone computing and the Internet of Things: A survey," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 885–898, Dec. 2016.

[43] Y.-J. Kim, J.-S. Seok, Y. Jung, and O.-K. Ha, "Light-weight and versatile monitor for a self-adaptive software framework for IoT systems," *J. Sensors*, vol. 2016, 2016, Art. no. 8085407.

[44] B. Kim *et al.*, "Heterogeneous distributed shared memory for lightweight Internet of Things devices," *IEEE Micro*, vol. 36, no. 6, pp. 16–24, Nov./Dec. 2016.

[45] H. Shin and J.-S. Park, "On optimizing random network coding implementation," in *Proc. IEEE Int. Conf. Inf. Commun. Techn. Converg. (ICTC)*, Jeju-do, South Korea, 2015, pp. 1365–1367.

[46] H. Shin and J.-S. Park, "Optimizing random network coding for multimedia content distribution over smartphones," *Multimedia Tools Appl.*, to be published,

[47] 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.

[48] H. Shojania, B. Li, and X. Wang, "Nuclei: GPU-accelerated many-core network coding," in *Proc. IEEE Infocom*, Rio de Janeiro, Brazil, 2009, pp. 459–467.

[49] D. Chen *et al.*, "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.

[50] J. Heide, M. V. Pedersen, F. H. P. Fitzek, and T. Larsen, "Network coding for mobile devices-systematic binary random rateless codes," in *Proc. ICC Workshops*, Dresden, Germany, 2009, pp. 1–6.

[51] 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.

[52] J. Qureshi, C. H. Foh, and J. Cai, "Optimal solution for the index coding problem using network coding over GF(2)," in *Proc. IEEE SECON*, Seoul, South Korea, 2012, pp. 209–217.

[53] 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.

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

[55] P. Maymounkov, N. J. Harvey, and D. S. Lun, "Methods for efficient network coding," in *Proc. 44th Annu. Allerton Conf. Commun. Control Comput.*, 2006, pp. 482–491.

[56] S.-M. Choi, K. Lee, and J.-S. Park, "Massive parallelization for random linear network coding," *Appl. Math. Inf. Sci.*, vol. 9, no. 2L, pp. 571–578, 2015.

[57] X. B. Gan, L. Shen, Z. Y. Wang, X. Lai, and Q. Zhu, "Parallelizing network coding using CUDA," *Adv. Mater. Res.*, vol. 186, pp. 484–488, Jan. 2011.

[58] 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.
[59] S. Lee and W. W. Ro, "Accelerated network coding with dynamic stream decomposition on graphics processing unit," *Comput. Soc. Comput. J.*, vol. 55, no. 1, pp. 21–34, Jan. 2012.
[60] J.-S. Park, S. J. Baek, and K. Lee, "A highly parallelized decoder for random network coding leveraging GPGPU," *Comput. Soc. Comput. J.*, vol. 57, no. 2, pp. 233–240, Feb. 2014.
[61] H. Shojania and B. Li, "Pushing the envelope: Extreme network coding on the GPU," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Montreal, QC, Canada, 2009, pp. 490–499.
[62] H. Shojania and B. Li, "Tenor: Making coding practical from servers to smartphones," in *Proc. ACM Int. Conf. Multimedia*, Florence, Italy, 2010, pp. 45–54.
[63] 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.
[64] 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)*, 2014, pp. 1–6.
[65] H. Shojania and B. Li, "Parallelized progressive network coding with hardware acceleration," in *Proc. IEEE Int. Workshop Qual. Service*, Aalborg, Denmark, 2007, pp. 47–55.
[66] H. Shojania and B. Li, "Random network coding on the iPhone: Fact or fiction?" in *Proc. ACM NOSSDAV*, Williamsburg, VA, USA, 2009, pp. 37–42.
[67] M. V. Pedersen, J. Heide, F. H. P. Fitzek, and T. Larsen, "A mobile application prototype using network coding," *Eur. Trans. Telecommun.*, vol. 21, no. 8, pp. 738–749, Dec. 2010.
[68] P. Vingelmann, F. H. P. Fitzek, M. V. Pedersen, J. Heide, and H. Charaf, "Synchronized multimedia streaming on the iPhone platform with network coding," *IEEE Commun. Mag.*, vol. 49, no. 6, pp. 126–132, Jun. 2011.
[69] L. Keller *et al.*, "Microcast: Cooperative video streaming on smartphones," in *Proc. ACM Int. Conf. Mobile Syst. Appl. Services*, 2012, pp. 57–70.
[70] 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.
[71] 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. 2014, 2014, Art. no. 974836.
[72] 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.
[73] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage," *ACM Trans. Math. Softw. (TOMS)*, vol. 5, no. 3, pp. 308–323, Sep. 1979.
[74] 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. (TOMS)*, vol. 14, no. 1, pp. 1–17, Mar. 1988.
[75] E. Anderson *et al.*, *LAPACK Users' Guide*, vol. 9. Philadelphia, PA, USA: SIAM, 1999.
[76] K. Goto and R. van de Geijn, "On reducing TLB misses in matrix multiplication," Dept. Comput. Sci., The Univ. Texas at Austin, Austin, TX, USA, Tech. Rep. TR-2002-55, 2002. [Online]. Available: http://citeseer.ist.psu.edu/goto02reducing.html
[77] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proc. ACM/IEEE Conf. Supercomput.*, San Jose, CA, USA, 1998, pp. 1–27.
[78] J.-G. Dumas, P. Giorgi, and C. Pernet, "Dense linear algebra over word-size prime fields: The FFLAS and FFPACK packages," *ACM Trans. Math. Softw. (TOMS)*, vol. 35, no. 3, pp. 1–42, Oct. 2008.
[79] J.-G. Dumas *et al.*, "LinBox: A generic library for exact linear algebra," in *Proc. Int. Congr. Math. Softw.*, 2002, pp. 40–50.
[80] G. H. Golub and C. F. Van Loan, *Matrix Computations*, vol. 3. Baltimore, MD, USA: JHU Press, 2012.
[81] L. Karlsson, "Computing explicit matrix inverses by recursion," M.S. thesis, Dept. Comput. Sci., Umeå Univ., Umeå, Sweden, 2006.
[82] 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)*, San Jose, CA, USA, 2013, pp. 299–306.
[83] 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)*, 2016, pp. 786–790.
[84] 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.*, Seattle, WA, USA, 2011, pp. 33–42.
[85] A. Kayi, Y. Yao, T. El-Ghazawi, and G. Newby, "Experimental evaluation of emerging multi-core architectures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Long Beach, CA, USA, 2007, pp. 1–6.
[86] J. Meng and K. Skadron, "Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling," in *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, South Lake Tahoe, CA, USA, Oct. 2009, pp. 282–288.
[87] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek, "Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting," *Concurrency Comput. Pract. Exp.*, vol. 26, no. 7, pp. 1408–1431, May 2014.
[88] ODROID. *Odroid XU-3*. Accessed on May 22, 2017. [Online]. Available: http://www.hardkernel.com/
[89] "Cortex-A15 technical reference manual," ARM Holdings, Cambridge, U.K., Tech. Rep. ARM DDI 0438C, 2011.
[90] "Cortex-A7 MPCore technical reference manual," ARM Holdings, Cambridge, U.K., Tech. Rep. ARM DDI 0464D, 2012.
[91] H. P. Anvin. (2011). *The Mathematics of RAID-6*. Accessed on May 22, 2017. [Online]. Available: https://www.kernel.org/pub/linux/kernel/people/hpa/raid6.pdf
[92] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2347–2376, 4th Quart., 2015.
[93] U. Isikdag, "Internet of Things: Single-board computers," in *Enhanced Building Information Models*. New York, NY, USA: Springer, 2015, pp. 43–53.
[94] M. V. Pedersen, J. Heide, and F. H. P. Fitzek, "Kodo: An open and research oriented network coding library," in *Proc. Networking Workshops*, Valencia, Spain, 2011, pp. 145–152.
[95] J. Heide, M. V. Pedersen, F. H. P. Fitzek, and M. Médard, "On code parameters and coding vector representation for practical RLNC," in *Proc. IEEE ICC*, Kyoto, Japan, 2011, pp. 1–5.
[96] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for Internet of Things and analytics," in *Big Data and Internet of Things: A Roadmap for Smart Environments*. N. Bessis and C. Dobre, Eds. Cham, Switzerland: Springer, 2014, pp. 169–186.
[97] R. Deng, R. Lu, C. Lai, T. H. Luan, and H. Liang, "Optimal workload allocation in fog-cloud computing toward balanced delay and power consumption," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 1171–1181, Dec. 2016.
[98] R. Want, B. N. Schilit, and S. Jenson, "Enabling the Internet of Things," *IEEE Comput.*, vol. 48, no. 1, pp. 28–35, Jan. 2015.
[99] H. Shin and J.-S. Park, "Energy efficient QoS-aware random network coding on smartphones," *Mobile Netw. Appl.*, to be published.
[100] X. Chu, K. Zhao, and M. Wang, "Accelerating network coding on many-core GPUs and multi-core CPUs," *J. Commun.*, vol. 4, no. 11, pp. 902–909, Dec. 2009.

Simon Wunderlich received the Dipl.-Inf. degree in computer science from Chemnitz Technical University, Chemnitz, Germany, in 2009. He is currently pursuing the Ph.D. degree in electrical engineering at the Technical University Dresden, Dresden, Germany.

He co-authored the Wi-Fi mesh software B.A.T.M.A.N. Advanced.

Juan A. Cabrera received the B.Sc. degree in electronics engineering from Simón Bolívar University, Caracas, Venezuela, in 2013, the M.Sc. degree in wireless communication systems from Aalborg University, Aalborg, Denmark, in 2015, and is currently pursuing the Ph.D. degree at the Deutsche Telekom Chair of Communication Networks, Technical University Dresden, Dresden, Germany.

His current research interests include network coding, fog computing, distributed storage systems, and mobile edge cloud solutions.

Frank H. P. Fitzek received the Diploma (Dipl.-Ing.) degree in electrical engineering from the University of Technology—Rheinisch-Westfälische Technische Hochschule, Aachen, Germany, in 1997, the Ph.D. (Dr.-Ing.) degree in electrical engineering from the Technical University of Berlin, Berlin, Germany, in 2002, and the Doctor Honoris Causa degree from the Budapest University of Technology and Economy, Budapest, Hungary, in 2015.

He is a Professor and the Head of the Deutsche Telekom Chair of Communication Networks, Technical University Dresden, Dresden, Germany, coordinating the 5G Lab Germany. In 2002, he became an Adjunct Professor with the University of Ferrara, Ferrara, Italy. In 2003, he joined Aalborg University, Aalborg, Denmark, as an Associate Professor and later became a Professor. He co-founded several start-up companies beginning with Acticom GmbH, Berlin, in 1999. His current research interests include wireless and mobile 5G communication networks, mobile phone programming, network coding, cross layer as well as energy efficient protocol design, and cooperative networking.

Dr. Fitzek was a recipient of the NOKIA Champion Award from 2007 to 2011, the Nokia Achievement Award for his research on cooperative networks in 2008, the SAPERE AUDE Research Grant from the Danish Government in 2011, and the Vodafone Innovation Prize in 2012.

Martin Reisslein (A'96–S'97–M'98–SM'03–F'14) received the Ph.D. degree in systems engineering from the University of Pennsylvania, Philadelphia, PA, USA, in 1998.

He is a Professor with the School of Electrical, Computer, and Energy Engineering, Arizona State University, Tempe, AZ, USA.

Dr. Reisslein currently serves as an Associate Editor for the IEEE TRANSACTIONS ON MOBILE COMPUTING, the IEEE TRANSACTIONS ON EDUCATION, IEEE ACCESS, *Computer Networks*, and *Optical Switching and Networking*. He is an Associate Editor-in-Chief for the IEEE COMMUNICATIONS SURVEYS & TUTORIALS and chairs the Steering Committee of the IEEE TRANSACTIONS ON MULTIMEDIA.