Optimum LDPC Decoder: A Memory Architecture Problem

Erick Amador
EURECOM
06904 Sophia Antipolis, France
erick.amador@eurecom.fr

Renaud Pacalet
TELECOM ParisTech
06904 Sophia Antipolis
renaud.pacalet@telecom-paristech.fr

Vincent Rezard
Infineon Technologies France
06560 Sophia Antipolis
vincent.rezard@infineon.com

ABSTRACT

This paper addresses a frequently overlooked problem: designing a memory architecture for an LDPC decoder. We analyze the requirements to support the codes defined in the IEEE 802.11n and 802.16e standards. We show a design methodology for a flexible memory subsystem that reconciles design cost, energy consumption and required latency on a multistandard platform. We show results after exploring the design space on a CMOS technology of $65nm$ and analyze various use cases from the standardized codes. Comparisons among representative work reveal the benefits of our exploration.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles

General Terms

Algorithm, Design

Keywords

LDPC codes, low power architectures, memory optimization

1. INTRODUCTION

Low-density parity-check (LDPC) codes [4] are among the best known error-correcting codes because of their capacity-approaching performance and the explicit parallelism exhibited by their iterative decoding algorithm. These codes have already been adopted by several wireless communication standards like IEEE 802.11n [5] and 802.16e[6] among others. Such standards define several coding rates and codeword lengths in order to provide different levels of link adaptability, consequently an implementation of a decoder should be flexible to support various modes of operation. Furthermore, if the target application entails mobile terminals then ultra-low power operation is mandatory. Standardized LDPC codes embed a particular structure into the parity-check matrix that reduces the interconnection complexity between the processing nodes and enables the realization of semi-parallel architectures. These type of architectures are amenable for flexibility, i.e. supporting different parity-check matrices that arise from different codeword lengths and coding rates. Decoding of LDPC codes comprises a memory intensive kernel, rendering the memory subsystem as the main source of energy consumption. Although this subsystem is an important factor for the overall performance of the decoder, its design is very frequently overlooked. Even though there are numerous published works on LDPC decoders design, to the best of our knowledge the memory subsytem design exploration has been omitted in most cases.

Several works have presented the tradeoff between throughput and area, finding the right compromise between the level of processing parallelism and implementation area. Representative state-of-the-art works like [1] and [9] present an exploration of the design space that spans from processing parallelism, the processing units approximation up to the decoding schedules.

In [3] the memory allocation problem is investigated to avoid memory conflicts, but like the other works lacks an exploration in terms of energy efficiency and area cost. While these works explore the fundamental tradeoff between processing parallelism, operating frequency and implementation area, they usually result in overdimensioned designs. The resulting memory organizations are costly in terms of area per bit, since fewer bigger memories are preferable to several small memories for on-chip SRAM.

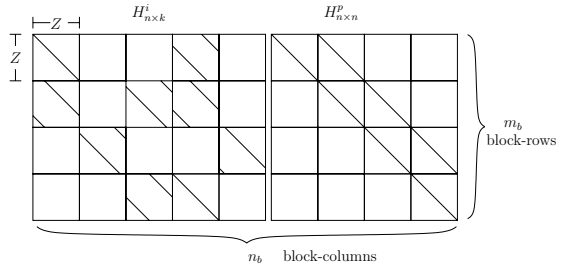
In this work we concentrate on the exploration of the memory architecture to optimally implement an LDPC decoder that supports the codes defined in the IEEE 802.11n and 802.16e standards. The paper is organized as follows: structured LDPC codes and the implemented decoding algorithm are presented in Section 2. Section 3 presents the required storage components of the decoder and section 4 presents the real time requirements. Section 5 presents the methodology we propose and results from the design space exploration, as well as comparisons among similar works. Section 6 concludes the paper.

2. STRUCTURED LDPC CODES

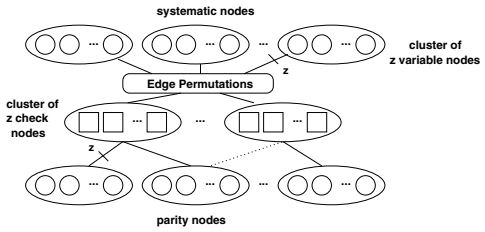
LDPC codes are linear block codes described by a sparse parity-check matrix $H_{M\times N}$ that defines M parity-check constraints over N codeword symbols. The number of non-zero elements in a column and in a row determine the degree of the column and the row respectively. The code can be described by a bipartite graph, where columns of H are mapped to *variable* nodes and rows are mapped to *check* nodes. A non-zero element in H represents an edge be-

DAC 2009, July 26 - 31, 2009, San Francisco, California USA.

tween the corresponding variable and check node. *Structured* LDPC codes are composed of several layers of non-overlapping rows, where these layers are formed by $Z \times Z$ sub-matrices. Z is an *expansion factor* that shows the degree of available parallelism as Z non-overlapping rows can be processed concurrently. Each of these sub-matrices can be either a zero matrix or a circularly right-shifted version of the identity matrix. Furthermore, these codes are constructed such that their encoding process exhibits linear complexity, identifying two sections within H: a random part for the systematic information and a prearranged part for the parity information. Figure 1 shows the H matrix and the bipartite graph of an example structured code, where $H_{M\times N} = [H^i_{n\times k}|H^p_{n\times n}]$, with $M = n$ and $N = n + k$. Indeed, H consists of an array $m_b \times n_b$ of $Z \times Z$ blocks.



(a) Parity-check matrix



(b) Bipartite graph

Figure 1: Structured LDPC code

The graph representation in figure 1(b) contains edges of width Z, grouping Z nodes into clusters. This grouping enables the possibility to instantiate a subset of the processing nodes, representing a clear advantage in terms of flexibility and silicon area as well as reducing the complexity of the interconnection network (edge permutations).

LDPC codes are decoded iteratively using a two-phase message-passing algorithm [4], where check and variable nodes exchange extrinsic reliability values associated with each codeword symbol. Node operations are independent and may be executed in parallel, allowing the possibility to use different scheduling techniques for various purposes. The turbo-decoding message-passing (TDMP) schedule [8] shows important improvements over the two-phase schedule: a faster convergence and reduced memory requirements.

In TDMP the check nodes of the graph are evaluated sequentially, updating and propagating more reliable messages along the graph. In the following we summarize this algorithm from [8]. Initial channel observations per codeword symbol are given as log-likelihood ratios in an initial vector $\boldsymbol{\delta} = [\delta_1, \ldots, \delta_N]$. Each row i in H has a corresponding vector of *extrinsic* messages $\boldsymbol{\lambda}^i = [\lambda^i_1, \ldots, \lambda^i_{c_i}]$, where c_i is the degree of row i. Let I_i denote the set of c_i elements in row i. A vector $\boldsymbol{\gamma} = [\gamma_1, \ldots, \gamma_N]$ contains the sum of all messages generated in the rows for each codeword symbol, thereby encompassing *posterior* messages. Hard decisions are taken by slicing the vector $\boldsymbol{\gamma}$ to obtain the decoded message. The decoding of the ith row involves these steps:

1. Read $\boldsymbol{\lambda}^i$ and the c_i elements of $\boldsymbol{\gamma}$ that participate in the row, denoted as $\boldsymbol{\gamma}(I_i)$.
2. Generate a vector $\boldsymbol{\rho}$ of prior messages given by: $\boldsymbol{\rho} = \boldsymbol{\gamma}(I_i) - \boldsymbol{\lambda}^i$.
3. Process $\boldsymbol{\rho}$ with a soft-input soft-output (SISO) decoding algorithm. Denote the result as $\boldsymbol{\Lambda} = SISO(\boldsymbol{\rho})$. Any SISO algorithm may be implemented with different levels of hardware complexity and error-correcting performance. A practical choice commonly found is the *Min-Sum* algorithm [2] and its variants.
4. Update the previously read vectors according to these rules: $\boldsymbol{\lambda}^i \leftarrow \boldsymbol{\Lambda}$; $\boldsymbol{\gamma}(I_i) \leftarrow \boldsymbol{\rho} + \boldsymbol{\Lambda}$

This method merges check and variable updates in one step reducing the memory requirements when compared to the traditional two-phase method. Another important advantage is the reduction in the number of iterations by up to 50%, indeed saving energy by the same proportion.

3. MEMORY ARCHITECTURE

TDMP decoding requires the following storage elements: posterior messages memory ($\boldsymbol{\gamma}$) and extrinsic messages memory ($\boldsymbol{\lambda}$). It is important to note that as described in [8], with TDMP decoding the vector $\boldsymbol{\gamma}$ is initialized with the channel observations $\boldsymbol{\delta}$, such that $\boldsymbol{\delta}$ does not play a role in the iterative process other than during initialization. Besides these memories, storage is required for the structure of the parity-check matrix, involving the location of the non-zero sub-matrices and their shift value.

3.1 Parity-check matrix storage

The IEEE 802.11n and 802.16e standards define different parity-check matrices for different use cases, where a use case is defined by a codeword length and a coding rate. In 802.11n there are a total of 12 matrices for 12 use cases, and in 802.16e there are 6 matrices for 19 use cases. The structure of each of these matrices follows the structure shown in figure 1(a). As described in the previous section, for decoding the ith row it is necessary to retrieve the set I_i of c_i elements from $\boldsymbol{\gamma}$. $H_{M\times N}$ consists of an array $m_b \times n_b$ of $Z \times Z$ blocks, where for both standards $n_b = 24$ and $m_b \in \{4, 6, 8, 12\}$ which depends upon the use case. Storing H involves saving the shift value of the non-zero blocks and the block-column position. Since for Z rows c_i remains constant, the decoding of Z rows involves retrieving c_i shift values from H. These values can be encoded in a read-only memory by concatenating in one word the shift value and column position. The shift values for 802.11n are taken directly from the specified matrices, whereas the shift values for 802.16e follow a precalculation dependent upon the use case. For all cases the shift value $s \leq Z$, where $Z_{11n} = \{27, 54, 81\}$ and $Z_{16e} = \{24, 28, 32, \ldots, 96\}$. In this way one 12-bit word encodes both the shift value and the column position (7-bits for shift value and 5-bits for the column number). It is evident that only the random part of H, $H^i_{n\times k}$ needs to be stored. For the 18 matrices this translates into a total of 1295 values.

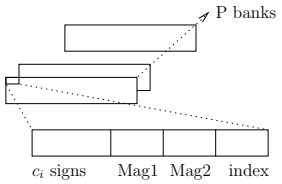


Figure 2: Extrinsic memory

3.2 Posterior messages memory

This memory is initialized with the log-likelihood ratio values $\boldsymbol{\delta}$ taken from the channel observations and updated after each row processing. This memory contains as many values as the codeword length. Hard decisions on the stored vector produce a valid codeword after successful decoding. One property provided by the structure of the codes is that when processing each block-row of H only a subset of the block-columns is used. This property can be exploited when mapping the data to the memory architecture, which we explain in section 5.

3.3 Extrinsic messages memory

This memory stores the messages that are obtained after each decoding round per row, and are used as input prior messages to decode subsequent rows. The dimension of this memory corresponds to the number of non-zero elements in H, or equivalently to the number of edges in the code graph. While this is the largest of the two memories it is easier to design, since the data allocation is straightforward by following the flow in TDMP decoding: the processing of each row uses always the same values from the same locations. The data does not have to be neither shifted nor distributed to different processing units if each row decoding is bound to a specific processing unit. The choice of *Min-Sum* as the SISO algorithm has further consequences on this memory. This algorithm takes c_i inputs and produces c_i outputs, with each having a sign and one of two possible magnitudes: the first or second minimum value out of the c_i inputs. So instead of storing c_i messages it is only necessary to store c_i signs, two magnitudes and the index where the first minimum is to be found. For our case this represents a reduction in memory size of up to 65% when using messages of 8-bits. Figure 2 shows the storage for these values.

Figure 3(a) shows a top level view of the architecture for a TDMP LDPC decoder. Shuffling units π and π^{-1} are used to distribute the data to and from P computation units. These units perform the processing of the rows in H, figure 3(b) shows the dataflow of the processing pipeline.

4. BANDWIDTH REQUIREMENTS

The codes defined in the 802.11n and 802.16e standards exhibit very tight and demanding timing requirements. Table 1 shows the main characteristics defined in the standards for both the most and less demanding cases (4 out of 31 cases shown). The decoding latency corresponds to the maximum permissible time in order to perform the decoding task, such that the baseband processing respects the time allotted for generating negative or positive acknowledgement messages.

From this table we can estimate the sizes of the memories for posterior and extrinsic messages and their respective bandwidth requirements. In terms of memory size the 802.16e case is used as it requires the most number of samples, but for the timing constraints it is clear that 802.11n has tighter deadlines. From this point on we use a maximum number of 8 decoding iterations and 8-bits long messages. Initially we can fix a hypothetical operating frequency to observe the minimum required bandwidth for the memory architecture. This bandwidth is obtained by:

Table 1: Specifications for standardized LDPC codes

Use case	Most demanding		Less demanding	
Standard 802.xx	11n	16e	11n	16e
Codeword length	1944	2304	648	576
H dimensions ($m_b \times n_b$)	12x24	12x24	4x24	4x24
Z value	81	96	27	24
Latency	8μs	0.25ms	8μs	0.25ms
Row degree c_i	7,8	6,7	22	20
Rows in H	972	1152	108	96
Graph edges	6966	7296	2376	1920

Table 2: Required frequency and memory width

Bus width (samples)	12	16	32
Average freq. [MHz]	580.50	435.38	217.69
Peak freq. [MHz]	648.00	486.00	243.00

$$BW = \frac{total\ \ samples \times iterations}{decoding\ \ latency \times frequency} \tag{1}$$

In order to manage realistic and practical widths on the memory organization, it is possible to fix the memory bus width after observing the initial estimations provided by table 1 and obtain the required operating frequency. It is worth noting from table 1 that when decoding different rows the number of required samples may vary, this given by the degree of the rows having different values, e.g. $c_i \in \{7, 8\}$ for the 802.11n most demanding case. From this we define an average and a peak operating frequency, where the average frequency considers the actual cases where different row degrees are used, and where the peak frequency considers the maximum row degree only. By introducing a peak operating frequency, we introduce several dead cycles or equivalently a decrease in the utilization of the memory bandwidth. Table 2 shows the obtained operating frequencies for different bus widths.

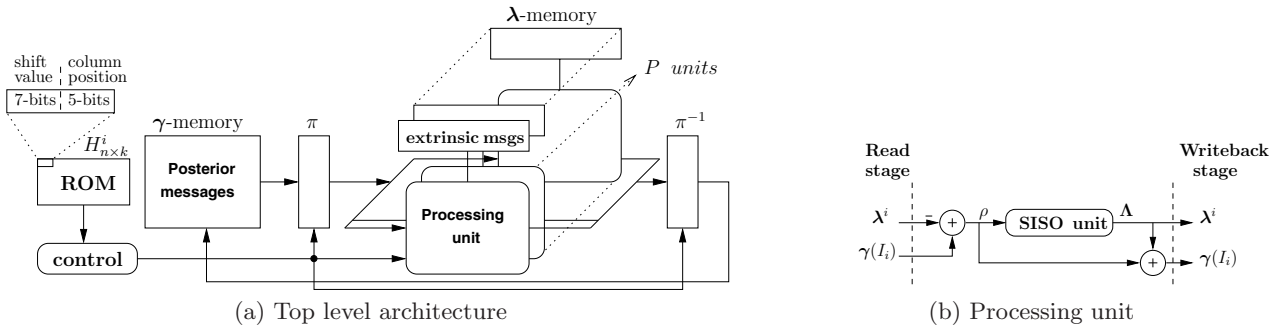
The decoding throughput can be estimated by the number of decoded bits, the operating frequency and the number of computation cycles. For our case we use P serial processing units that decode a row of degree c_i in $2 \times c_i$ cycles (including read and writeback stages). The throughput T is given by:

$$T = \frac{(n_b - m_b) \times Z}{v \times m_b \times 2c_i \times t} \times f_{clock} \tag{2}$$

where t is the number of iterations and $v = Z/P$ shows the level of parallelism used.

5. DESIGN STRATEGY

The memory architecture has to be dimensioned to handle several use cases with differences in workload. We concentrate on a worst case analysis to observe the greatest impacts on energy, so we use the case of a memory bandwidth of 12 samples/cycle. For the 802.11n most demanding case 8



(a) Top level architecture

(b) Processing unit

Figure 3: TDMP LDPC Decoder

samples are required per decoding of a row operating at the peak frequency. If 12 samples are available per read cycle this suggests either the parallel processing of 1.5 rows per cycle, or a combination of decoding a number of rows in a number of cycles: e.g. 3 rows in 2 cycles. In order to avoid cooperative processing between SISO units and take advantage of the structure in H, we assign the decoding of a row to a processing unit. Notice that we are dealing with the most demanding case (biggest H) such that for the less demanding cases there is more time available to fetch the samples. As mentioned in section 3.3 the extrinsic messages memory data allocation is straightforward and it is conveniently distributed among different processing units. In section 5.2 we explore how to further partition this memory to capitalize on energy scalability. Next we elaborate on the particular challenges for designing the posterior messages memory.

5.1 Data mapping and allocation

With a bandwidth requirement and a cycle budget, we design the number of banks and the dimensions of the posterior messages memory by exploiting the structure of the codes. From this we identify clearly two types of data organization and placement:

1. Micro-organization: provided by the shifted identity matrices that make up the block structure of H. This structure allows the processing of non-overlapping rows, so that consecutive samples are distributed to several processing units by means of a shuffling unit. From this it follows that for distributing samples to P units each memory location could store exactly P samples, such that all are accessible per read operation. Figure 4 shows this organization for $P = 3$ and $Z = 81$ for one block-column. The codeword length is in this case $24 \times Z = 1944$, a use case from 802.11n. The micro-organization is shown for the block-column that corresponds to the codeword symbols $\{324, 325, \ldots, 404\}$. Due to the structure of the $Z \times Z$ blocks all Z samples will be used when processing a block-row. In figure 4 we further show how 3 samples that are required are read from two memory locations, and how the remaining samples of the read accesses can be stored as they will be used in subsequent rows' processing.

2. Macro-organization: c_i block-columns are used per block-row processing. For the most demanding case (802.11n) when operating on a given block-row it is required only to have access to either 7 or 8 block-columns. From this we define the macro-organization as the mapping of block-columns to several banks such that the required blocks are all accessible per read operation.

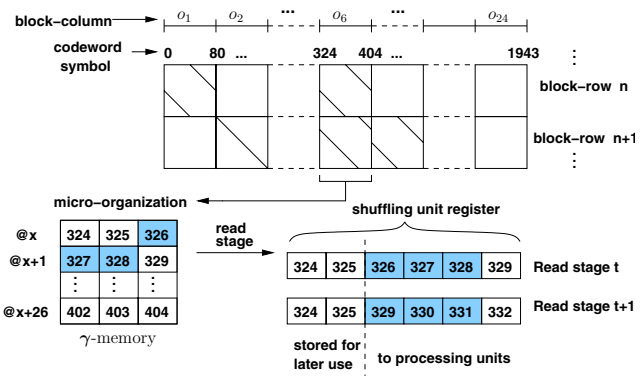


Figure 4: Micro-organization example

This constitutes a problem of allocating static objects (block-columns) to memory banks. As shown in [7] this problem is *NP-complete* and it maps naturally onto the well-known graph colouring problem. This introduces another dimension to the design criteria to organize the memory architecture, since not only does the bandwidth have to be achieved, but the data allocation required has to be possible. This situation is formalized as follows: a set of static objects $O = \{o_1, o_2, \ldots, o_{24}\}$ is to be mapped to a set of memory banks B. Objects in O *conflict* when they need to be accessed at the same time. The memory allocation problem is solved by finding an allocation function $alloc: \ O \rightarrow B$ that avoids as much conflicts as possible. A conflict represents a loss in execution speed, introducing stall cycles and bubbles into a pipelined implementation.

From the most demanding case in table 1, we analyze the mapping of data in the posterior messages memory for the case in 802.11n at rate 1/2 and codeword length of 1944. From [5] we take the LDPC matrix definition and build *conflict graphs* $G = (V, E)$, where V is the set of vertices representing the block-columns used on a given block-row and E is the set of edges between the nodes of V that incur in a conflict. The chromatic number $\chi(G)$ of a conflict graph determines the minimum number of colors needed such that all vertices of the graph are coloured and adjacent vertices do not share the same color. In other words, the chromatic number determines the minimum number of memories needed to provide the required bandwidth. There should be as many conflict graphs as block-rows in H, but we consider now the bandwidth requirement and the degree of the rows in H to observe the cycle budget for reading the required samples. The case of 12 samples/cycle and the concurrent processing of 3 rows every 2 cycles leads to ac-

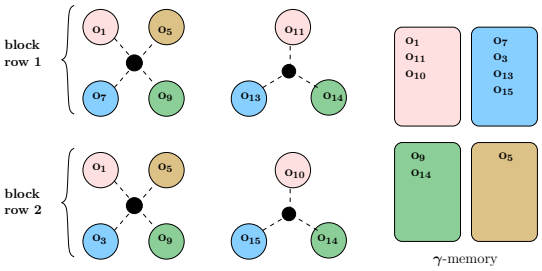


Figure 5: Macro-organization by conflict graphs

Table 3: Energy from H matrix storage

Use case	Energy/iteration [nJ]
$R = 1/2, N = 2304$ (16e)	0.47
$R = 5/6, N = 576$ (16e)	0.64
$R = 5/6, N = 648$ (11n)	0.70

cessing as much as 4 block-columns per read access. This produces 2 conflict graphs per block-row, where the chromatic number is bounded, $\chi(G) \leq 4$. Figure 5 shows the situation for the first 2 block-rows of H for this use case, using $B = 4$ memory banks that provide a bandwidth of 12 samples/cycle to $P = 3$ processing units.

This mapping is performed offline and is solvable by the well known graph colouring techniques. By analyzing the parity-check matrix the conflict graphs can be constructed and coloured such that the block-columns are allocated to the memory banks and the number of conflicts is reduced or even avoided.

5.2 Exploration

One important parameter for the cost of a memory architecture is the area per bit. By reducing the number of banks also interconnection links and control circuitry are reduced. This impacts not only cost but also power consumption, so it is fundamental to perform a study on the memory architecture candidates to achieve an efficient and economical design. From the area/bit point of view there are two extreme cases: using one single large memory or a maximum number of memories that guarantees the required number of parallel accesses. Compared to small memories, large memories consume more energy per access because of the longer word and bit lines. But distributing the data along several small memories leads also to relatively high energy consumption due to the added interconnection lines and decoding circuitry overhead.

Using an in-house memory exploring tool and *Synopsys Primetime*, we investigate candidate memory architectures where the data mapping has been shown to be possible. The target technology is a $65nm$ typical CMOS process with $V_{dd} = 1.32V$.

The exploration for the storage of H is limited since it requires non-volatile storage in principle. We synthesized this memory as a [324x48] low power ROM. In table 3 we show the energy consumed per decoding iteration when reading this control data for the corner use cases, defined by the number of non-zero blocks in each H matrix.

The exploration of the extrinsic messages memory involves inquiring about the possibility of partitioning each of the memories that are bound to each processing unit. This memory stores the compressed row vector shown in figure

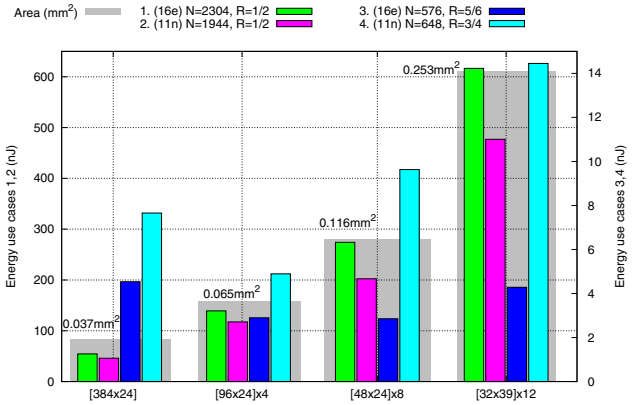


Figure 6: λ-memory exploration

2. Each of the 31 use cases has different requirements. The corner cases are defined by the number of edges in the code graph and the degree of the rows. In figure 6 we explored the area and average energy per iteration consumed in the extrinsic messages memory for the two corner cases (both from 16e) and two intermediate cases with several memory configurations, using low power dual-port SRAMs clocked at 648MHz. For the less demanding cases the unused partitions are powered off. This simple power management strategy allows to observe the energy cost behavior according to the memory partitions, where an optimal configuration exists for a particular use case. Once an optimal choice is taken more elaborated power management techniques (voltage/frequency scaling) can further improve the energy efficiency.

For the case of the posterior messages memory it is compelling to follow suit and partition further this memory while maintaining the required bandwidth and achieving gains on energy by exploiting the flexibility of the architecture. We explored several configurations for bandwidths of 12 and 32 samples/cycle using low power dual-port SRAMs clocked at 648MHz and 243MHz respectively. The corner use cases explored are characterized by the number of memory accesses, which depends upon the degree of the block-rows. For example the case $R = 1/2$ and $N = 2304$ in 802.16e requires 76 accesses per decoding iteration to the 24 blocks of 96 samples each stored in this memory. Figure 7 shows the average energy consumed per iteration in this memory for several partitions.

For both use cases shown we can observe that the configurations for 32 samples/cycle are more energy efficient than the 12 samples/cycle ones, but more costly in terms of area/bit.

With this exploration an appropriate selection of the optimum memory configurations is possible. Figure 8 shows the breakdown of the estimated implementation area [1] and average energy consumed per iteration on different use cases for the following realization, λ-memory: [96x24]x4, γ-memory: [192x48]x2 and $P = 3$. While the use case dictates which of the memories (λ- or γ-memory) dominates the energy consumption, it is clear that at least 85% of the energy is spent on the memory architecture.

Although there are numerous published works on LDPC

[1] Min-Sum used as SISO kernel and Benes networks as shuffling units

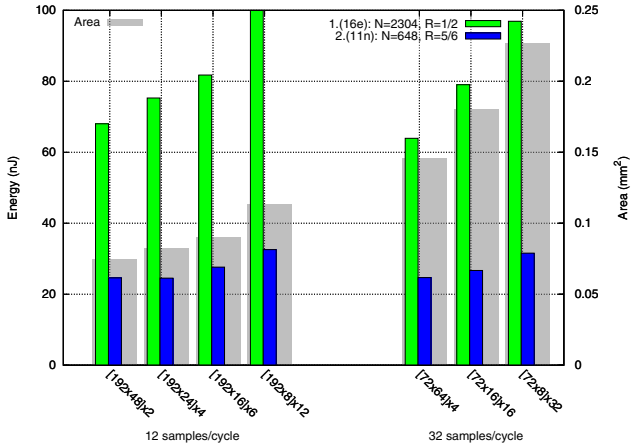


Figure 7: γ-memory exploration

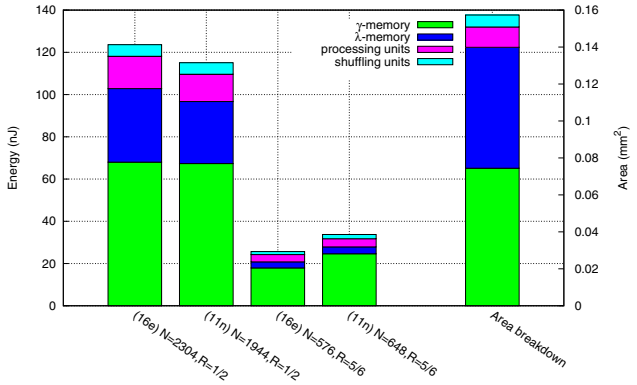


Figure 8: Area and use case energy breakdown

decoder design, it is hard to find a thorough report on the breakdown of implementation complexity and energy on a use case basis when addressing standardized codes. In table 4 we show a comparison with representative state-of-the-art works. All works have been implemented on $65nm$ CMOS technologies. In this table our work corresponds to the realization λ-memory: [96x24]x4 and γ-memory: [72x64]x4. Even though it is difficult to draw comparisons among multi-objective explorations we can observe that our approach avoids the usual overdimensions (in terms of throughput) and focuses instead on meeting the requirements (latency) while choosing an optimal memory architecture. In our case optimality being energy and area costs. Only [9] reports the breakdown of energy for a use case, so we show in table 4 the total energy consumed on λ- and γ-memories. While [9] averages over three different SNRs, ours is a worst case estimation at the maximum number of iterations.

6. CONCLUSIONS

In this work we explored the frequently overlooked problem of finding an optimal memory configuration on a memory-driven application. We have presented how the memory architecture alone represents one of the most challenging parts in the design of an LDPC decoder. Our design space exploration emphasizes on implementation area cost and energy consumption on a use case basis. The design steps followed comprise the analysis of the real-time requirements, an initial estimation on dimensions and bandwidth of the memories, and a further pruning of the search space by considering

Table 4: Memory area and energy comparison

Work	this work	[1]		[9]
Standard 802.xx	11n/16e	11n	16e	11n
Operating point	1.32V 243MHz	N/A V 400MHz		1.2V 240MHz
Messages length	8-bit	6-bit		5 and 8-bit
Memory area $[mm^2]$	0.211	0.467	0.551	0.265
Throughput $[Mbps]$	92-162	108-337	96-399	175-292
Latency $[\mu s]$	3.0-7.5	5.8-6.0	5.7-6.0	5.65-5.98
Memory energy Use case 802.11n: R=1/2,N=648	275.71 nJ	N/A	N/A	590.81 nJ

the cost merits of different realizations in terms of area and energy per use case. We detailed the challenges of designing and partitioning the posterior messages memory by identifying two levels of data organization. Albeit it remains challenging to implement LDPC decoders, our work reveals that a thorough exploration yields more attractive results in terms of cost of the storage elements and energy scalability according to the supported use cases.

7. REFERENCES

[1] T. Brack, M. Alles, T. Lehnigk-Emden, F. Kienle, N. Wehn, N. E. L'Insalata, F. Rossi, M. Rovini, and L. Fanucci. Low Complexity LDPC Code Decoders for Next Generation Standards. In *Proc. of the Conference on Design, Automation and Test in Europe*, pages 331–336, San Jose, USA, 2007.

[2] J. Chen, A. Dholakia, E. Eleftheriou, M. Fossorier, and X. Hu. Reduced-Complexity Decoding of LDPC Codes. *IEEE Trans. Comms.*, 53:1288–1299, August 2005.

[3] J. Dielissen and A. Hekstra. Non-fractional Parallelism in LDPC Decoder Implementations. In *Proc. 2007 Design, Automation and Test in Europe*, Nice, France, 2007.

[4] R. Gallager. Low-Density Parity-Check Codes. *IRE Trans. Inf. Theory*, 7:21–28, January 1962.

[5] IEEE-802.11n. Wireless LAN Medium Access Control and Physical Layer Specifications: Enhancements for Higher Throughput. *P802.11n/D3.07*, March 2008.

[6] IEEE-802.16e. Air Interface for Fixed and Mobile Broadband Wireless Access Systems. *P802.16e-2005*, October 2005.

[7] P. Keyngnaert, B. Demoen, B. De Sutter, B. De Sus, and K. De Bosschere. Conflict Graph Based Allocation of Static Objects to Memory Banks. *Informal proceedings of the First workshop on Semantic, Program Analysis, and Computing Environments*, pages 131–142, September 2001.

[8] M. Mansour. A Turbo-Decoding Message-Passing Algorithm for Sparse Parity-Check Matrix Codes. *IEEE Trans. on Signal Processing*, 54(11):4376–4392, November 2006.

[9] M. Rovini, G. Gentile, F. Rossi, and L. Fanucci. A Scalable Decoder Architecture for IEEE 802.11n LDPC Codes. In *Proc. of IEEE GLOBECOMM*, pages 3270–3274, 2007.