Synthesizing Interconnect-Efficient Low Density Parity Check Codes

Marghoob Mohiyuddin
The University of Texas at Austin

Amit Prakash
The University of Texas at Austin

Adnan Aziz
The University of Texas at Austin

Wayne Wolf
Princeton University

ABSTRACT

Error correcting codes are widely used in communication and storage applications. Codec complexity has usually been measured with a software implementation in mind. A recent hardware implementation of a Low Density Parity Check code (LDPC) indicates that interconnect complexity dominates the VLSI cost. We describe a heuristic interconnect-aware synthesis algorithm which generates LDPC codes that use an order of magnitude less wiring with little or no loss of coding efficiency.

Categories and Subject Descriptors: B.6.3 Automatic Synthesis, E.4 Error Control Codes, G.2.2 Graph Algorithms.

General Terms: Algorithms, Performance, Design.

Keywords: Error correcting codes, low-density parity-check codes, routing congestion.

1. INTRODUCTION

Error correcting codes are widely used in communication and storage applications, e.g., Turbo codes in 802.16a and Reed-Solomon codes in DVDs. Information theorists have studies these codes for decades. However, their measure of codec complexity has been more appropriate for software implementation than hardware implementation, e.g., the number of states in the FSM performing decoding, rather than the gate count, or wire length.

It is becoming clearer that there are significant advantages to using custom hardware for codecs. One benefit naturally is raw performance. More subtle, however, is the benefit in reduced power. This is particularly true for coding schemes which can use the parallelism offered by hardware: by using more devices, we can reduce frequency, and therefore scale voltage, thereby achieving the same throughput for less power [4].

Turbo codes [2] and Low Density Parity Check codes (LDPCs) [5] are two leading families of error correcting codes—both come very close to achieving the bounds predicted by Shannon theory. Of these two families, Turbo codes are hard to parallelize [10], and hence, benefit less from hardware implementation. LDPCs however have recently begun to gain attention because they are parallelizable, and VLSI technology has gotten to the point where LDPCs over fairly large code word sizes can be implemented in hardware.

With the selection of LDPC codes for the satellite digital broadcasting standard (DVB-S2), efficient VLSI implementation of these codes has become all the more important. Existing algorithms for building such codes do not take into account the VLSI cost [9], specially the wiring complexity of the decoder. In a recently published implementation of an LDPC decoder hardware, interconnect completely dominates the layout [1].

2. LDPC CODE IMPLEMENTATION

Low Density Parity Check codes were introduced by Gallager [5]. They are a natural extension of the parity bit commonly used for single bit error detection. Given m message bits, an LDPC encoder constructs a code word of length c ($> m$) bits. The coding scheme is associated with p possibly overlapping subsets of the code bits such that the parity of each subset in any codeword is 0. These subsets are described by a $p \times c$ matrix $H = [h_{ij}]$, such that $h_{ij} = 1$ iff the i-th subset has bit j in it. The H matrix is called the parity check matrix of the code; it is sparse, hence the name 'low density'.

The H matrix defines the set of possible codewords. If all messages are equally likely, then the error correction capability of a code is only dependent on the set of codewords; specifically it is independent of the actual mapping of messages to codewords. Hence the parity check matrix defines the error correcting capability of an LDPC code.

The parity check matrix can be visualized as representing a bipartite graph $G = (V, E)$, referred to as the Tanner graph of the code, with $p + c$ vertices $V = \{1, 2, ..., p, p + 1, ..., p + c\}$. The first p vertices ($\{1, 2, ..., p\}$) correspond to p parity checks and are called check nodes. The remaining c vertices ($\{p + 1, ..., p + c\}$) are called code nodes (they can be thought of as representing the c bits of the codeword that was transmitted). A code node is connected to a check node if the bit corresponding to it participates in the parity check corresponding to the check node. We shall refer to the Tanner graph of an LDPC code simply as an LDPC graph.

Message passing [5] is an iterative decoding algorithm for decoding LDPC codes. Messages are passed between code nodes and check nodes along the edges connecting them. These messages are an estimate of the transmitted code bit and a measure of reliability of these estimates.

Until recently, the major focus of research had been the software implementation of LDPC decoders. With the increasing demand for high throughput decoders, the complexity of hardware implementation of LDPC encoder/decoder has been gaining attention.

In the first published implementation of a practical LDPC decoder hardware, Blanskby *et al.* [1] describe a high throughput LDPC decoder implementation. Each node in the LDPC graph is implemented as a separate functional unit. The interconnects between the units describe the connectivity of the nodes. They report that routing congestion was the determining factor in the area of the chip (50% utilization). Furthermore, they also mention that because of the large number of long nets, most of the power dissipation is due to the switching activity of the wires.

It becomes important to consider the issues related to hardware implementation when designing LDPC codes, to make them practical.

Current approaches to handling hardware related issues have been limited to generating LDPC codes from special families of graphs (e.g., Ramanujam graphs) [6, 7] which imposes constraints on the code parameters. Another approach uses simulated annealing [8] to generate LDPC graphs satisfying certain conditions.

Campeliot *et al.* [3] describe the bit-filling method to generate LDPC graphs given constraints on the degree of the nodes and the girth (length of the smallest cycle in a bipartite graph).

Our main contribution is an algorithm to generate LDPC codes with a bound on the 'height' of the VLSI layout of the decoder for the code. This limits routing congestion with little or no loss in the effectiveness of the code. We compare the performance of the generated codes with different constraints on the 'height' of the decoder and examine the performance-area tradeoffs in the code design. In this context, we also compare the performance of the codes designed using the bit-filling algorithm [3] with the ones generated by our algorithm.

3. ALGORITHM AND ANALYSIS

3.1 Minimizing Cuts

We introduce the notion of a cut for the Tanner graph of an LDPC code.

Consider the example shown in Figure 1(a). The layout in Figure 1(b) represents a routing channel with nets to be routed between the terminals on both sides of the channel. The objective is to minimize the vertical height of the channel (which is the number of horizontal tracks required). A vertical line defines a cut (we shall refer to this as a vertical cut) on the edges of the graph. The maximum number of edges crossing any vertical cut (dashed vertical lines in Figure 1(b)) is a lower bound on the height of the channel, since the different edges crossing any cut have to be routed on different horizontal tracks. For the example shown in Figure 1(b), the maximum vertical cut-size is 9, hence a feasible layout would require at least 9 horizontal tracks. The layout shown uses exactly 9 horizontal tracks.

For the Tanner graph of an arbitrary LDPC code with e edges, the expected size of the maximum cut would be at least $\frac{e}{2}$, since the edge connectivities are not constrained. Thus, the height of the channel and, consequently, the area of the layout is expected to be large. A larger height means longer wires and therefore, greater interconnect capacitance, implying more power consumption. Therefore, congestion becomes an important issue as the block size of the code increases.These are exactly the problems with the VLSI design of Blanksby *et al.* [1].

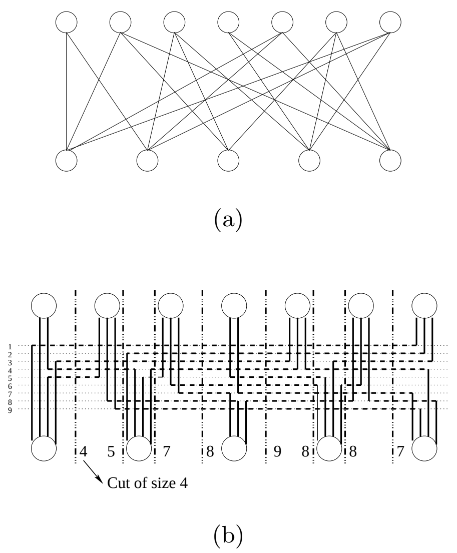

Figure 1: A Tanner graph and its layout.

In summary, VLSI layout issues need to be incorporated in the design of the code. Our algorithm addresses this issue by generating codes with a constraint on their maximum cut-size.

3.2 Our Algorithm: Intuition

It has been shown in [5] that the girth of an LDPC graph affects its error correcting properties. A larger girth is desirable for a better performance. Therefore, it is important to incorporate the girth constraints into the LDPC code generation process.

The bit-filling algorithm [3] assumes a given number of check nodes. Code nodes and the corresponding edges are added to the graph until some (e.g., girth) constraints are violated. All the edges of a code node are added one after another during the algorithm. Our algorithm assumes a given number of code nodes and check nodes. At a high level, it consists of multiple passes. In a single pass, at most a single edge is added to a check node. Intuitively, since we want all the cut-sizes to be constrained the same way, all the nodes should be treated in the same way. Therefore, we proceed in a breadth first manner instead of the depth first manner of bit-filling. Edges are added between a code node and a check node only if the constraints are not violated. Thus, in a single pass, at most p edges would be added (with at most 1 edge added to a code node or check node). A single pass can be thought of generating a 1-1 mapping from the check nodes to the code nodes. The mapping generated in a single pass is constrained by the maximum cut-size. The number of passes is determined by the maximum degree of the check nodes or the average degree of the check nodes. In our implementation, the number of passes is governed by the average degree of the check nodes.

3.3 Our Algorithm Formalized

3.3.1 Objective

Generate an LDPC graph with the following parameters: number of code nodes c, number of (parity) check nodes p, average degree of (parity) check nodes d_{avg}, minimum girth of the graph g, and maximum cut-size of the mapping generated in a single pass is cut_{max}.

3.3.2 Preliminaries

1. $U = \{u_1, u_2, \ldots, u_p\}$ is the set of check nodes. $V = \{v_1, v_2, \ldots, v_c\}$ is the set of code nodes.
2. pos_u denotes the column number of node u.
3. $dist(u, v)$ denotes the number of edges in the shortest path from node u to node v.
4. For $u \in U$, $Checks_u$ is the set of nodes adjacent to u.
5. For $v \in V$, we denote by N_v the set of code nodes which are at distance exactly 2 from v.
6. As in [3], for a check node u,
$$\mathcal{N}_u^i = \begin{cases} Checks_u & (i = 1); \\ \bigcup_{v \in \mathcal{N}_u^{i-1}} N_v & (i > 1). \end{cases}$$
$\mathcal{N}_u^2$ is the set of code nodes at distance 2 from some code node in $Checks_u$. $\mathcal{N}_u^3$ is the set of code nodes at distance 2 from some code node in $\mathcal{N}_u^2$, hence at distance 4 from some code node in $Checks_u$. By a similar argument, $\mathcal{N}_u^i$ is the set of nodes at distance $2i-2$ from some code node in $Checks_u$. Hence $\mathcal{N}_u^i$ is the set of nodes at distance $2i-1$ from check node u.
7. For $u \in U$, I_u is the largest continuous block of column numbers, such that $pos_u \in I_u$ and $cut_i < cut$ (for a given value of cut) for all $i \in I_u$.

3.3.3 Main Algorithm

We recall the ideas in the bit-filling algorithm [3]. If the current graph does not violate the girth constraints, then edge (u, v) can be added, without violating the girth constraint of g, if and only if $dist(u, v) \geq g - 1$. Since $dist(u, v) \geq g - 1$, the edge (u, v) would create a cycle of length at least g (or no cycle if there was no path from u to v previously).

The set $\mathcal{N}_u^i$ contains code nodes at distance $2i - 1$ from check node u. Therefore, the set of code nodes at distance at most $g - 2$ from u would be $\mathcal{N}_u^* = \bigcup_{i=1}^{\frac{g-1}{2}} \mathcal{N}_u^i$.

We maintain $\mathcal{N}_u^*$ for each check node u, and N_v for each code node v.

We have assumed that the layout consists of c columns, with one code node per column (v_i placed on column i). The check nodes are placed uniformly on the c columns (assume u_i placed on column $\lceil \frac{ic}{p} \rceil$).

Consider a single pass of the algorithm. During a single pass, the cut-size for each column is maintained. The cut-sizes during a pass are initialized to 0. The cut-size constraint cut is initialized to $init_{cut}$. Each check node is considered one by one. A check node u of minimum degree is selected randomly (Step 6). If the girth constraint is not considered, then an edge (u, v) can be added if it does

```
1: $\mathcal{N}_u^* \leftarrow \emptyset, u \in U$, $N_v \leftarrow \emptyset, v \in V$,
   $e \leftarrow p \cdot d_{avg}$, $num\_edges \leftarrow 0$
2: while $num\_edges < e$ and $numiter < max\_iter$ do
3:   $A \leftarrow U$, $cut_i \leftarrow 0, 1 \leq i \leq c$
4:   $cut \leftarrow init_{cut}$
5:   while $A \neq \emptyset$ do
6:     Select $u \in A$ {using some heuristic}
7:     $A \leftarrow A - \{u\}$
8:     if $\mathcal{N}_u^* \neq V$ then
9:       Compute $I_u$, $S = \{v \in V \mid pos_v \in I_u\}$
10:      $C = S - \mathcal{N}_u^*$
11:      if $C \neq \emptyset$ then
12:        Select $v \in C$ {using some heuristic}
13:        Update $\mathcal{N}_u^*, Checks_u, N_v$
14:        Update, for all $v' \in V$, $N_{v'}$
15:        $num\_edges \leftarrow num\_edges + 1$
16:        Update $cut_i$ for all $i$
17:      else if $cut < cut_{max}$ then
18:        $cut \leftarrow min(2cut, cut_{max})$
19:        Repeat Steps 9-16
20:      end if
21:    end if
22:  end while
23: end while
```

Algorithm 1: LDPC Code Generation

not increase the cut-size of the columns that the edge traverses. Edge (u, v) would violate the cut-size constraint if $pos_v \notin I_u$ (since I_u is maximum, $cut_j = cut$ for a column j on the boundary of I_u). Therefore, the set S of code nodes which can be made neighbors of u (without considering the girth constraint) is $S = \{v \in V \mid pos_v \in I_u\}$. The set of code nodes that can be added as neighbors of u, without violating the girth and the cut constraints, would then be $C = S - \mathcal{N}_u^*$. Choose a code node v of minimum degree randomly from C (Step 13) and add the edge (u, v). Update the cut-sizes, $\mathcal{N}_u^*$, $N_{v'}$ for all $v' \in Checks_u$, and N_v. If C turns out to be empty because of the cut constraint, then the cut-size constraint cut is increased (until cut_{max}), and S and C are recomputed. If no edges can be added to u in the current pass, due to the maximum cut constraint or the girth constraint, it is not considered further in the current pass.

Updating the set N_v is easy since v now shares check node u with code nodes in $Checks_u$. Hence, $N_v = N_v \bigcup Checks_u$. Similarly, for all v' in $Checks_u$, $N_{v'} = N_{v'} \bigcup \{v\}$.

Update the set $\mathcal{N}_u^*$ by first computing the set $\mathcal{M}^*$ of code nodes at distance at most $g - 2$ from v (set $\mathcal{M}^1$ as $\{v\}$ and then compute $\mathcal{M}^i = \bigcup_{v' \in \mathcal{M}^{i-1}} N_{v'}$, let $\mathcal{M}^* = \bigcup_{i=1}^{\frac{g-1}{2}} \mathcal{M}^i$, this is exactly similar to the definition of $\mathcal{N}_u^i$). Change $\mathcal{N}_u^*$ to $\mathcal{N}_u^* \bigcup \mathcal{M}^*$.

The number of passes is determined by the total number of edges to be added. So, the objective is to generate an LDPC graph with $e = d_{avg} \cdot p$ edges.

For Steps 6 and 13, we select a node of minimum degree randomly. This can be extended to the node selection heuristic in the bit-filling algorithm [3].

3.3.4 Complexity Analysis

Steps 3-4 take $O(c)$ time. Steps 9-10 take $O(c)$ time. For Steps 13-16, $\mathcal{M}^*$ can be computed in $O(c)$ time; other updates also take $O(c)$ time. Steps 17-19 are a repetition of Steps 9-16, they require $O(c)$ time. Inner loop runs p times. Outer loop runs $\leq max\_iter$ times. Time complexity is, therefore, $O(max\_iter \cdot p \cdot c)$. For our implementation,

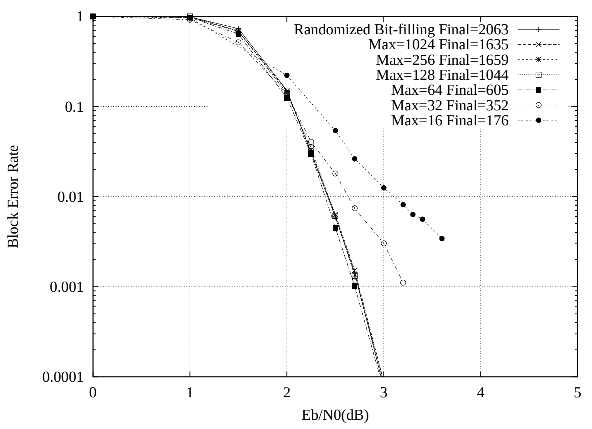

Figure 2: A comparison of LDPC codes with parameters c=1024, p=512, g=6, d_{avg}=8. Max is the maximum cut-size constraint for each pass. $Final$ is the maximum cut-size of the generated graph. In randomized bit-filling, nodes are picked at random from the set of candidate nodes.

$max\_iter = O(d_{avg})$, therefore time taken $= O(d_{avg}pc) = O(ep) = O(p^2)$ (since $e = O(p)$).

4. RESULTS

Message passing decoding was used with 64 iterations of decoding for each block. BPSK mapping was used for transmitting codewords ($0 \rightarrow +1, 1 \rightarrow -1$). LDPC codes being linear codes, we only need to transmit the all zeros codeword and try decoding it for simulation studies. This is because the Gaussian noise added to the transmitted codeword is independent of the transmitted codeword. The best codes out of 5 randomly generated codes, were used for the results.

For comparison with Blanksby *et al.*'s [1] code, we generated rate-1/2 codes with c=1024, p=512, g=6 and d_{avg}=6.5 and 8. Higher values of g have been mentioned in the figures, whenever increasing g resulted in an improvement. The simulations were done for an additive white Gaussian channel with varying signal-to-noise ratio (SNR), where the SNR is defined as the ratio of the signal power (which is normalized to 1) to the variance of the Gaussian noise.

Figure 2 illustrates the change in performance of codes generated as cut_{max} is varied. Observe that except for very small cut_{max} (in this case – 16 and 32) the performance of codes degrades only marginally as cut_{max} is decreased. Therefore, our algorithm is able to find good LDPC codes with small maximum cut-size. Note that the code with final maximum cut-size = 605 (generated by our algorithm) has performance same as that of the code with final maximum cut-size = 2063, about 75% reduction in the maximum cut-size (without loss in performance). The height of the layout as well as the area is directly proportional to the cut-size.

Figure 3 compares the performance of codes generated with different parameters. We used the data points for Blanksby *et al.*'s code [1]. For the same block error rate, the difference in SNR is less than 0.7 db. This means that using our codes, the increase in transmitter power would be less than 17%, while the area of the decoder reduces to a quarter, which is desirable in many applications.

We did interconnect layout for the decoder of the generated codes in 0.18μ technology. For two codes with $c = 1024, p = 512, g = 6, d_{avg} = 8$, and varying cut_{max} we computed the height and width of the channel layout (assuming single bit messages). With $cut_{max} = 256$, the width, height and area are 3.56 mm, 1.31 mm and 4.66 mm^2 respectively. For $cut_{max} = 64$, the width, height and area are 3.56 mm, 0.36 mm and 1.27 mm^2 respectively. This is a 75% reduction in channel layout area with little performance degradation. Furthermore, reducing the wire lengths would decrease the power consumption of the chip.

Blanksby *et al.*'s [1] approach to building codes is based more on extensive simulations for a large number of randomly generated LDPC codes. Our algorithm, in comparison, adopts a constructive approach to generating layout-aware LDPC codes.

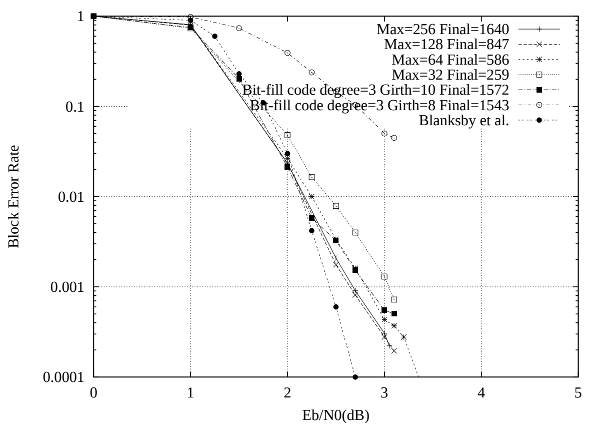

Figure 3: A comparison of LDPC codes with parameters c=1024, p=512, d_{avg}=6.5. Unless stated otherwise, g=6.

5. REFERENCES

[1] A. J. Blanksby and C. J. Howland. A 690-mW 1-Gb/s 1024-b, Rate-1/2 Low-Density Parity-Check Decoder. *IEEE J. Solid-State Circuits*, 37:404–412, Mar. 2002.

[2] M. Bossert. *Channel Coding for Telecommunications.* John Wiley and Sons, Aug. 1999.

[3] J. Campeliot, D. S. Modha, and S. Rajagopalan. Designing LDPC Codes Using Bit-Filling. In *IEEE Int. Conf. Communications*, pages 55–59, June 2001.

[4] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-Power CMOS Digital Design. *IEEE J. Solid-State Circuits*, 27:473–484, Apr. 1992.

[5] R. G. Gallager. *Low-Density Parity-Check Codes.* PhD thesis, MIT, Cambridge, MA, 1962.

[6] M. Mansour and N. Shanbhag. Architecture-Aware Low-Density Parity-Check codes. *IEEE Int. Symp. Circuits and Systems*, May 2003.

[7] M. Mansour and N. Shanbhag. A Novel Design Methodology for High Performance Programmable Decoder Cores for AA-LDPC Codes. *IEEE Workshop on Signal Processing Systems*, Aug. 2003.

[8] J. Thorpe. Design of LDPC Graphs for Hardware Implementation. In *IEEE Int. Symp. Inform. Theory*, page 483, Lausanne, Switzerland, June 2002.

[9] E. Yeo, B. Nikolic, and V. Anantharam. Iterative Decoder Architectures. *IEEE Communications Magazine*, Aug. 2003.

[10] E. Yeo, P. Pakzad, B. Nikolic, and V. Anantharam. VLSI Architectures for Iterative Decoders in Magnetic Recording Channels. *IEEE Trans. Magnetics*, 37:748–755, Jan. 2001.