Eleventh International Workshop on Algebraic and Combinatorial Coding Theory
June 16-22, 2008, Pamporovo, Bulgaria *pp. 161-165*

Hard decision decoding of binary low-density codes based on extended Hamming codes

Alexey Kuznetsov ring0@list.ru
Institute for Information Transmission Problems, Russian Academy of Sciences, Moscow 101447, RUSSIA

Abstract. In this paper we investigate decoding performance of binary low-density codes based on extended Hamming codes, using a hard decision decoding algorithm similar to Gallager's bit-flipping algorithm. The possibilities to implement the decoder in hardware are also discussed.

1 Introduction

Low-density parity-check codes were first introduced by Gallager [1]. In [2] asymptotical properties of low-density codes based on Hamming codes were investigated. The simulation results of the soft decision decoding of these codes were given in [3]. It has been also proven there that the minimum distance of such codes grows linearly with the code length. In this paper we study similar codes based on extended Hamming codes.

2 Code structure

The codes in question are defined by their parity-check matrix.

Let n_0 be a component code length. Let a matrix $\mathbf{H_1}$ be a parity-check matrix for the component code (extended Hamming code in our case). Let r_0 be a number of rows in this matrix. We can now construct a $n_0 m \times r_0 m$ -sized matrix $\mathbf{H_m}$ in the following way: every row $j = ir_0 + k$ ($k < r_0$, $i \leq m - 1$) is filled using k's row of $\mathbf{H_1}$ starting with $i * n0$ column. Other positions are filled with zeros. I. e.,

$$\mathbf{H_m} = \begin{bmatrix} \mathbf{H_1} & \mathbf{0} & \mathbf{0} & ... & \mathbf{0} \\ \mathbf{0} & \mathbf{H_1} & \mathbf{0} & ... & \mathbf{0} \\ ... & & & & \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & ... & \mathbf{H_1} \end{bmatrix}$$

Now, a code's parity-check matrix can be defined as follows:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H_m P_1} \\ \mathbf{H_m P_2} \\ \cdots \\ \mathbf{H_m P_l} \end{bmatrix}$$

Here $\mathbf{P_i}$ are permutation matrixes.

We will investigate a particular code class with the following parameters: $n_0 = 128$, $m = 256$, $r_0 = 8$, $l = 3$. We'll also consider P_1 to be a trivial transform, since it doesn't influence code BER performance.

3 Decoding algorithm

A simple hard decision decoding algorithm exists for these codes.

At first, one can decode every component code. There are three possibilities:

1. The syndrome equals **0**.
2. The syndrome doesn't equal **0** and corresponds to the particular error (a component code has 'detected the error at some position).
3. The syndrome doesn't equal **0** and doesn't correspond to any error. I. e., the decoder fails to decode this codeword.

Each bit in the codeword is contained in three component codes. For every component code word it is contained in, each bit is assigned one of four *states*:

1. This bit wasn't marked as erroneous and the syndrome of the component code word was **0**.
2. This bit was marked as erroneous by the component code decoder.
3. This bit wasn't marked as erroneous and the component code decoder failed to decode the corresponding code word.
4. This bit wasn't marked as erroneous, but the component code decoder marked other bit in the corresponding code word.

After that, for each bit the following decision process is invoked:

1. If the bit was corrected in one component codeword, and no component codeword containing this bit had zero syndrome, this bit is *inverted.*
2. If the bit was corrected in two component codewords, it is also *inverted*, regardless of the other codeword decoding result.

3. Otherwise, the bit is leaved as it is (i. e., *not inverted*).

The decoding process is now repeated until no component code can correct a error (i. e., every component codeword either has zero syndrome or the component code decoder fails to decode it). This condition will be called further a *stop condition.*

4 Simulation

The decoding algorithm described above was simulated for this code class. The permutations $\mathbf{P_2}$ and $\mathbf{P_3}$ were randomly generated. Binary symmetric channel (BSC) was used for simulation.

It was found that the most frequent error set leading to the stop condition consists of two errors, which are located in one codeword for every codeword group. This possibility can be eliminated if we arbitrarily choose $\mathbf{P_2}$ instead of choosing it randomly. This goal can be achieved, for example, in such a way:

$$i^* = 128(i \cdot \mathrm{mod} 128) + \left\lfloor \frac{i}{128} \right\rfloor,$$

for $i < 2^{14}$, and

$$i^* = 2^{14} + 128((i - 2^{14}) \cdot \mathrm{mod} 128) + \left\lfloor \frac{i - 2^{14}}{128} \right\rfloor,$$

for $i \geq 2^{14}$, where i is a column position and i^* is a new column position (counting from 0).

The simulation shows that the code with arbitrarily chosen $\mathbf{P_2}$ tends to have slightly better BER performance in the BSC, at least when the input error probability is considerably high.

The table below shows some simulation results. All probabilities here are bit error probabilities.

p_{in}	p_{out} (random $\mathbf{P_2}$)	p_{out} (arbitrarily chosen $\mathbf{P_2}$)
0.011	$6.9 \cdot 10^{-6}$	$4.6 \cdot 10^{-6}$
0.010	$8.2 \cdot 10^{-7}$	$3.2 \cdot 10^{-7}$
0.009	$4.3 \cdot 10^{-7}$	$9.8 \cdot 10^{-8}$

5 Hardware implementation

The decoding of this code requires a huge amount of data transfers, so parallel computing can hardly be benefited from. Instead, a simple sequential architecture can be proposed.

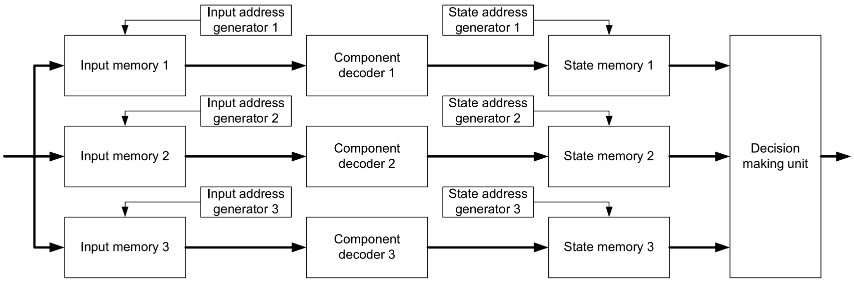

Figure 1: Decoder architecture

This architecture has quite a large memory requirements, but everything else will occupy quite a few logic cells.

The input memories are conventional 2-port static RAM blocks, configured for 1-bit data port width. This component should be available in any library. They are filled with identical data. Three blocks are needed to provide a possibility for simultaneous reading by the component code (extended Hamming) decoders. These three blocks can be, of course, replaced with one quad-port memory, if there's an appropriate component in the library.

Each of the component code decoders processes 1 bit per cycle. Each decoder contains a counter, which is used as a parity check matrix generator, and modulo 2 adder unit for syndrome calculation. The input data should be stored in internal memory until the syndrome is calculated. Then for each bit component code writes a bit value (unchanged) and a 2-bit state to the corresponding state memory.

Decision making unit then applies a decision process to every bit. The output data can be then iterated through the same hardware, at the cost of degraded processing performance. Alternatively, a long pipeline containing enough stages to make a decision in virtually any case can be used. Of course, this will lead to hardware duplication.

The memory requirements for the design can be estimated as follows:

- input memories - 32kbit $\cdot$ 3 = 96kbit.
- input address generators - $(32 \cdot 16) \cdot 3$kbit = 1536kbit (assuming 16-bit address width) in the general case. If the first permutation is trivial, and the second is arbitrarily chosen, the first two generators won't use memory, this will amount to 512kbit.
- component code decoders - negligible, about 128 bits each.
- stage memory address generators - like input address generators.

- stage memory blocks - $(32 \cdot 4) \cdot 3$kbit = 384kbit.

So, the whole design will use no less than 1504kbit of RAM, and 3552kbit in the general case.

The major drawback of the sequential architecture is that it can't process more than 1 bit per cycle. On modern programmable logic devices, such as Xilinx Virtex-5 chips, this amounts to about 500Mbit/s processing performance. Using a chip, designed specially for this problem (ASIC), one can reach about 1Gbit/s.

Getting processing performance considerably above 1Gbit/s is difficult due to the clock frequency limitations on semiconductor devices. To solve the problem, one can use several such decoders in parallel, possibly on different chips and with separate demultiplexer. This will lead to the increased latency, of course.

It should be also noted that in practical implementations the maximum number of iterations per codeword has to be limited. This will also lead to somewhat degraded BER performance.

6 Conclusion

The sequential architecture presented above has limited processing performance. In order to increase the performance further, one need to employ internal parallelism. It would be desirable to have a separate component code decoder for every component codeword. Unfortunately, this is connected with significant difficulties, since the problem requires a number of connections between the processing units, which seems to be beyond any practical limits.

References

[1] R. G. Gallager, *Low-density parity-check codes*, Cambridge: M.I.T. Press, 1963.

[2] S. Stiglmayr, V. V. Zyablov, Assymptotically good low-density codes based on Hamming codes, *Proc. Eleventh Intern. Symp. Probl. Redund. Inform. Contr. Syst.*, St. Petersburg, 2007.

[3] M. Lentmaier, K. S. Zigangirov, On generalized low-density parity-check codes based on Hamming component codes, *IEEE Commun. Lett.* 3, 1999.