バーストエラーに強く計算効率の良い FEC 方式についての提案

PRUG 根本 徳人 / JH1FBM

1.

高速無線データ通信において FEC は重要な技術である。また無線での通信に於ける FEC ではバーストエラー対策が必須と言われている。

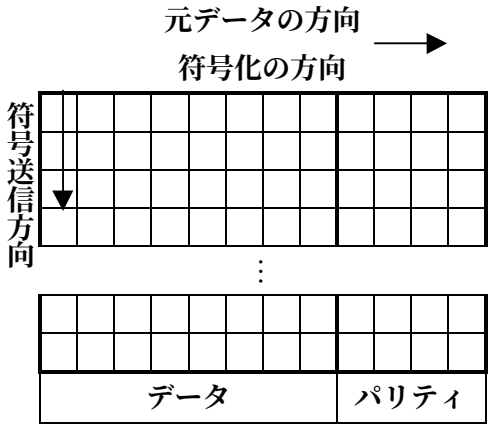
一般的に FEC にブロック符号を導入した場合、元のデータにパリティを付加したブロックをそのまま送信するとバーストエラーに対する耐性が低くなってしまう。そこで、ブロック符号などランダムエラー用の誤り訂正符号のバーストエラー耐性向上対策として交錯法(インターリーブ)がよく使われている。

インターリーブを行なう事によって局所に集中して起こるエラーを多くのブロックに拡散し、ブロック誤り訂正の処理能力内に収め、バーストエラーを検出、訂正出来るようにする効果がある。PRUG96 システムにおいて採用された APFS では 1bit 誤り訂正 8/12 縮退ハミング符号を使い、それをインターリーブしている(図 1)。

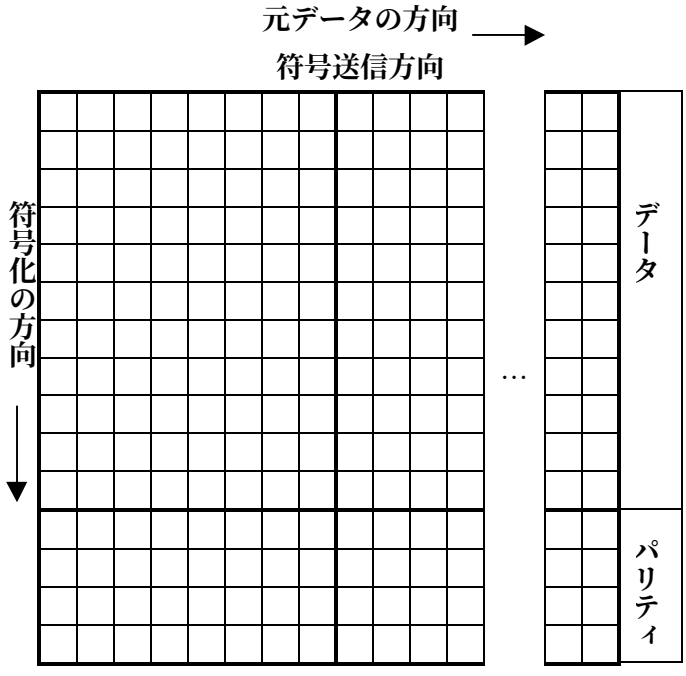
ところが計算機によって交錯処理を行なう場合 bit シフトを繰り返せざるを得ず、計算効率の低下が起こりやすい。また、元データが交錯されることで、デコードしない限りデータとパリティを分離しにくいという欠点をもつ。

ここでバーストエラーをブロック間に分散するために重要なのは、基本的に FEC のブロックの計算方向と符号化後のビット送信方向が交錯していることであり、元データのビット進行方向とは関係ないと言う点に注目した(図 2)。

つまり FEC の符号化方向(パリティの計算方向)が送信方向と交差するように計算すれば符号化後にインターリーブする必要が無くなるのである。


元データの方向
符号化の方向
符号送信方向
データ
パリティ

(図 1)


元データの方向
符号送信方向
符号化の方向
データ
パリティ

(図 2)

2.

1 で述べた様な、ブロックを送信方向と交錯した方向(図 2 での垂直方向)に計算する方法について考えて見る。例として 1bit 誤り訂正 11/15 ハミング符号を使っての実装を考えて見る。

送信側では送るべき情報を D0～DA(D10)とすると、付加するパリティ P0～P3 を以下の様に計算する(+は xor,*は and,~は not である)。

```
P0 = D0 + D1 + D3 + D4 + D6 + D8 + DA
P1 = D0 + D2 + D3 + D5 + D6 + D9 + DA
P2 = D1 + D2 + D3 + D7 + D8 + D9 + DA
P3 = D4 + D5 + D6 + D7 + D8 + D9 + DA
```

また受信側では受信語 D0'～DA',P0'～P3' (伝送路エラー H0～HE が加算されている)とすると、エラービット H0～HA は以下の様に計算できる(元の D0～DA のみ戻せば良いので、ここでは 0～A のみ計算している)。

```
S0 = P0'+ D0'+ D1'+ D3'+ D4'+ D6'+ D8'+ DA'
S1 = P1'+ D0'+ D2'+ D3'+ D5'+ D6'+ D9'+ DA'
S2 = P2'+ D1'+ D2'+ D3'+ D7'+ D8'+ D9'+ DA'
S3 = P3'+ D4'+ D5'+ D6'+ D7'+ D8'+ D9'+ DA'

H0 = S0 * S1 * ~S2 * ~S3
H1 = S0 * ~S1 * S2 * ~S3
H2 = ~S0 * S1 * S2 * ~S3
H3 = S0 * S1 * S2 * ~S3
H4 = S0 * ~S1 * ~S2 * S3
H5 = ~S0 * S1 * ~S2 * S3
H6 = S0 *  S1 * ~S2 * S3
H7 = ~S0 * ~S1 * S2 * S3
H8 = S0 * ~S1 * S2 * S3
H9 = ~S0 * S1 * S2 * S3
HA = S0 * S1 * S2 * S3
```

ここで訂正後の受信語は Dn = Dn' + Hn となる

この 1 ブロックを計算する基本的な方法は、図 2 のデータを横方向にシフトして順次取り出し、縦方向にブロックを計算した後で横方向にシフトするという事が考えられる。これは一回交錯を行なったあと、符号化を行ない再度交錯することになる。原理的にはこの方法が正しいが、これでは単純に交錯法より手間が掛かってしまう。

発想を転換すれば bit 単位で取り出す必要はなく、byte 単位、あるいはそれ以上をまとめて図 2 の縦方向に演算をすればよいことに気付く。これを利用して効率のよい演算をすることが可能である。

3.

付録に実際の C 言語による試験実装例を挙げる。long を用いる事によって 32bit 分をまとめて計算することで、大幅な処理の高速化を図っており、さらに計算量を少なくするための工夫も若干おこなっている。
符号自身は計算方向が違うだけで単純な 11/15 ハミング符号であり、96 バイト x 11 の情報領域に 96 バイト x 4 のパリティを付加する。ランダムエラーに対する特性は当然 11/15 ハミング符号と同じであり、データ、パリティ併せて 15bit に対して 2bit 以上の誤りは訂正できない。訂正不能、もしくは誤訂正の検出は別の機構(例えば CRC チェック)に委ねられることになる。また、この実装では最大 768bit のバーストエラーに対応できる。
なお、現在、上記方式の 7/15 2 重誤訂正 BCH の関数も作成中である。

4.

この FEC 計算方式の利点としては

・各種ブロック符号に広く応用可能
・FEC のために大きな表を用意しなくても代数演算で計算できメモリの少ない IPSM 上などでの実装が容易
・複数 bit をまとめて演算することにより高速化が可能
・組織符号が保たれているので、エラー検出後にパリティ送出をする、さらに強いパリティを送る、などの応用が可能
・情報 bit 11 など各種ブロック符号の符号の限界点で使う事が容易で符号の効率化が図れる

などが挙げられる。

また問題点としては

・計算が複雑で符号化率の低いブロック符号(例 5/15 3 重誤訂正 BCH 符号など)では符号化(表引き)＋インターリーブと比べても計算量の削減効果が薄いか、かえって多くなる可能性が高い
・ハードウエアでの実装する場合、他に比べ複雑になる
・組織符号が保たれている故、同一ビットの連続によって問題が起こる変調方式等によってはスクランブルなどの処理が必要になる

などが考えられる。

5.

当初、この方式の元情報の送出方向が変わらないので非交錯方式として考えていたが、参考文献などによると、符号化の方向と送出方向が交錯させる方法をインターリーブとして扱っていることを鑑み、この方法は、単に実現方法が違うだけで、交錯方式に分類すべきと考えている。

いずれにせよ、この方式では、計算機によってインターリーブを行なう欠点が大きく改善されていると考えられるので、ソフトウエアによって FEC 処理を行う上で有効な選択肢の一つであろう。

参考文献

誤り訂正符号とその応用（江藤 良純・金子 敏信）　オーム社
符号理論 （今井 秀樹）　電子情報通信学会

付録

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define DATALEN 11   /* 11/15 hamming code */
#define PARITY 4
#define CODELEN (DATALEN+PARITY)
typedef unsigned long calc_unit;
/*
#define BLOCKLEN 1440
#define REPCOUNT (BLOCKLEN / CODELEN / sizeof (calc_unit))
*/
#define REPCOUNT 24
#define RAWLEN (REPCOUNT * sizeof (calc_unit))
#define BLOCKLEN (CODELEN * RAWLEN)
#define BUFFLEN BLOCKLEN

void enc1115fec(calc_unit data[][REPCOUNT])
{
      int i;
      calc_unit tmpa, tmpb;

      for (i = 0; i < REPCOUNT; i++) {
            tmpa = data[0][i] ^ data[3][i] ^ data[6][i] ^ data[10][i];
            data[11][i] = tmpa ^ data[1][i] ^ data[4][i] ^ data[8][i];
            data[12][i] = tmpa ^ data[2][i] ^ data[5][i] ^ data[9][i];

            tmpb = data[7][i] ^ data[8][i] ^ data[9][i] ^ data[10][i];
            data[13][i] = tmpb ^ data[1][i] ^ data[2][i] ^ data[3][i];
            data[14][i] = tmpb ^ data[4][i] ^ data[5][i] ^ data[6][i];
      }
      return;
}
```

```
void dec1115fec(calc_unit y[][REPCOUNT])
{
	int i;
	calc_unit tmpa, tmpb;

	for (i = 0; i < REPCOUNT; i++) {
		tmpa = y[0][i] ^ y[3][i] ^ y[6][i] ^ y[10][i];
		y[11][i] ^= tmpa ^ y[1][i] ^ y[4][i] ^ y[8][i];
		y[12][i] ^= tmpa ^ y[2][i] ^ y[5][i] ^ y[9][i];

		tmpb = y[7][i] ^ y[8][i] ^ y[9][i] ^ y[10][i];
		y[13][i] ^= tmpb ^ y[1][i] ^ y[2][i] ^ y[3][i];
		y[14][i] ^= tmpb ^ y[4][i] ^ y[5][i] ^ y[6][i];


		tmpa = y[11][i] & y[12][i];
		y[10][i] ^= tmpa & y[13][i] & y[14][i];
		y[6][i] ^= tmpa & ~y[13][i] & y[14][i];
		y[3][i] ^= tmpa & y[13][i] & ~y[14][i];
		y[0][i] ^= tmpa & ~(y[13][i] | y[14][i]);

		tmpa = y[11][i] & ~y[12][i];
		y[8][i] ^= tmpa & y[13][i] & y[14][i];
		y[4][i] ^= tmpa & ~y[13][i] & y[14][i];
		y[1][i] ^= tmpa & y[13][i] & ~y[14][i];

		tmpa = ~y[11][i] & y[12][i];
		y[5][i] ^= tmpa & ~y[13][i] & y[14][i];
		y[2][i] ^= tmpa & y[13][i] & ~y[14][i];
		y[9][i] ^= tmpa & y[13][i] & y[14][i];

		y[7][i] ^= ~(y[11][i] | y[12][i]) & y[13][i] & y[14][i];
	}
	return;
}
```

```
#define TESTCOUNT 1000

main()
{
	int n,i,j, irand, count, norecover;
	unsigned char orgdata[BUFFLEN], data[BUFFLEN], ebit[BUFFLEN];
	unsigned char erd;

	for (n = 0; n < TESTCOUNT; n++) {
		norecover = count = 0;
		for (i = 0; i < RAWLEN*DATALEN; i++) {
			data[i] = (unsigned char)((double)rand() * 256 / RAND_MAX);
		}

		enc1115fec((void *)data);

		for (i = 0; i < RAWLEN*CODELEN; i++) {
			orgdata[i] = data[i];
		}

		for (i = 0; i < RAWLEN*CODELEN; i++) {
			erd = 0;
			for (j = 0; j < 7; j++, erd<<=1) {
				irand = (unsigned int)((double)rand() * 10000 / RAND_MAX+1);
				if (irand % 2000 == 49) { /* p = 2000/10000 */
					erd |= 1;
					count++;
				}
			}
			data[i] ^= ebit[i] = erd;
		}

		dec1115fec((void *)data);
```

```
            for (i = 0; i < RAWLEN; i++) {
                  for (j = 0; j < DATALEN; j++) {
                        if (data[j*RAWLEN+i] != orgdata[j*RAWLEN+i]) norecover++;
                  }
            }
            printf("no recover %5d in error count %5d \n", norecover, count);
            if (norecover >0) for (i = 0; i < RAWLEN; i++) {
                  for (j = 0; j < CODELEN; j++) {
                        printf("%2.2X ", ebit[j*RAWLEN+i]);
                  }
                  printf("\n");
            }
      }
}
```