NAND Error-correction Code

Introduction

Having looked at the linux mtd/nand Hamming software ECC engine driver I felt there was room for optimisation. I bashed the code for a few hours performing tricks like table lookup removing superfluous code etc. After that the speed was increased by 35-40%. Still I was not too happy as I felt there was additional room for improvement.

Bad! I was hooked. I decided to annotate my steps in this file. Perhaps it is useful to someone or someone learns something from it.

The problem

NAND flash (at least SLC one) typically has sectors of 256 bytes. However NAND flash is not extremely reliable so some error detection (and sometimes correction) is needed.

This is done by means of a Hamming code. I'll try to explain it in laymans terms (and apologies to all the pro's in the field in case I do not use the right terminology, my coding theory class was almost 30 years ago, and I must admit it was not one of my favourites).

As I said before the ecc calculation is performed on sectors of 256 bytes. This is done by calculating several parity bits over the rows and columns. The parity used is even parity which means that the parity bit = 1 if the data over which the parity is calculated is 1 and the parity bit = 0 if the data over which the parity is calculated is 0. So the total number of bits over the data over which the parity is calculated + the parity bit is even. (see wikipedia if you can't follow this). Parity is often calculated by means of an exclusive or operation, sometimes also referred to as xor. In C the operator for xor is ^

Back to ecc. Let's give a small figure:

byte 0:	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	rp0	rp2	rp4	...	rp14
byte 1:	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	rp1	rp2	rp4	...	rp14
byte 2:	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	rp0	rp3	rp4	...	rp14
byte 3:	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	rp1	rp3	rp4	...	rp14
byte 4:	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	rp0	rp2	rp5	...	rp14
...													
byte 254:	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	rp0	rp3	rp5	...	rp15
byte 255:	bit7 cp1 cp3 cp5	bit6 cp0 cp3 cp5	bit5 cp1 cp2 cp5	bit4 cp0 cp2 cp5	bit3 cp1 cp3 cp4	bit2 cp0 cp3 cp4	bit1 cp1 cp2 cp4	bit0 cp0 cp2 cp4	rp1	rp3	rp5	...	rp15

This figure represents a sector of 256 bytes. cp is my abbreviation for column parity, rp for row parity.

Let's start to explain column parity.

- cp0 is the parity that belongs to all bit0, bit2, bit4, bit6.

  so the sum of all bit0, bit2, bit4 and bit6 values + cp0 itself is even.

Similarly cp1 is the sum of all bit1, bit3, bit5 and bit7.

- cp2 is the parity over bit0, bit1, bit4 and bit5
- cp3 is the parity over bit2, bit3, bit6 and bit7.
- cp4 is the parity over bit0, bit1, bit2 and bit3.
- cp5 is the parity over bit4, bit5, bit6 and bit7.

Note that each of cp0 .. cp5 is exactly one bit.

Row parity actually works almost the same.

- rp0 is the parity of all even bytes (0, 2, 4, 6, ... 252, 254)
- rp1 is the parity of all odd bytes (1, 3, 5, 7, ..., 253, 255)
- rp2 is the parity of all bytes 0, 1, 4, 5, 8, 9, ... (so handle two bytes, then skip 2 bytes).
- rp3 is covers the half rp2 does not cover (bytes 2, 3, 6, 7, 10, 11, ...)
- for rp4 the rule is cover 4 bytes, skip 4 bytes, cover 4 bytes, skip 4 etc.

  so rp4 calculates parity over bytes 0, 1, 2, 3, 8, 9, 10, 11, 16, ...)
- and rp5 covers the other half, so bytes 4, 5, 6, 7, 12, 13, 14, 15, 20, ..

The story now becomes quite boring. I guess you get the idea.

- rp6 covers 8 bytes then skips 8 etc
- rp7 skips 8 bytes then covers 8 etc

- rp8 covers 16 bytes then skips 16 etc
- rp9 skips 16 bytes then covers 16 etc
- rp10 covers 32 bytes then skips 32 etc
- rp11 skips 32 bytes then covers 32 etc
- rp12 covers 64 bytes then skips 64 etc
- rp13 skips 64 bytes then covers 64 etc
- rp14 covers 128 bytes then skips 128
- rp15 skips 128 bytes then covers 128

In the end the parity bits are grouped together in three bytes as follows:

ECC	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ECC 0	rp07	rp06	rp05	rp04	rp03	rp02	rp01	rp00
ECC 1	rp15	rp14	rp13	rp12	rp11	rp10	rp09	rp08
ECC 2	cp5	cp4	cp3	cp2	cp1	cp0	1	1

I detected after writing this that ST application note AN1823 (http://www.st.com/stonline/) gives a much nicer picture.(but they use line parity as term where I use row parity) Oh well, I'm graphically challenged, so suffer with me for a moment :-)

And I could not reuse the ST picture anyway for copyright reasons.

Attempt 0

Implementing the parity calculation is pretty simple. In C pseudocode:

```
for (i = 0; i < 256; i++)
{
  if (i & 0x01)
     rp1 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp1;
  else
     rp0 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp0;
  if (i & 0x02)
     rp3 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp3;
  else
     rp2 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp2;
  if (i & 0x04)
    rp5 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp5;
  else
    rp4 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp4;
  if (i & 0x08)
    rp7 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp7;
  else
    rp6 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp6;
  if (i & 0x10)
    rp9 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp9;
  else
    rp8 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp8;
  if (i & 0x20)
    rp11 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp11;
  else
    rp10 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp10;
  if (i & 0x40)
    rp13 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp13;
  else
    rp12 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp12;
  if (i & 0x80)
    rp15 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp15;
  else
    rp14 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp14;
  cp0 = bit6 ^ bit4 ^ bit2 ^ bit0 ^ cp0;
  cp1 = bit7 ^ bit5 ^ bit3 ^ bit1 ^ cp1;
  cp2 = bit5 ^ bit4 ^ bit1 ^ bit0 ^ cp2;
  cp3 = bit7 ^ bit6 ^ bit3 ^ bit2 ^ cp3
  cp4 = bit3 ^ bit2 ^ bit1 ^ bit0 ^ cp4
  cp5 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ cp5
}
```

Analysis 0

C does have bitwise operators but not really operators to do the above efficiently (and most hardware has no such instructions either). Therefore without implementing this it was clear that the code above was not going to bring me a Nobel prize :-)

Fortunately the exclusive or operation is commutative, so we can combine the values in any order. So instead of calculating all the bits individually, let us try to rearrange things. For the column parity this is easy. We can just xor the bytes and in the end filter out the relevant bits. This is pretty nice as it will bring all cp calculation out of the for loop.

Similarly we can first xor the bytes for the various rows. This leads to:

Attempt 1

```
const char parity[256] = {
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0
};

void ecc1(const unsigned char *buf, unsigned char *code)
{
    int i;
    const unsigned char *bp = buf;
    unsigned char cur;
    unsigned char rp0, rp1, rp2, rp3, rp4, rp5, rp6, rp7;
    unsigned char rp8, rp9, rp10, rp11, rp12, rp13, rp14, rp15;
    unsigned char par;

    par = 0;
    rp0 = 0; rp1 = 0; rp2 = 0; rp3 = 0;
    rp4 = 0; rp5 = 0; rp6 = 0; rp7 = 0;
    rp8 = 0; rp9 = 0; rp10 = 0; rp11 = 0;
    rp12 = 0; rp13 = 0; rp14 = 0; rp15 = 0;

    for (i = 0; i < 256; i++)
    {
        cur = *bp++;
        par ^= cur;
        if (i & 0x01) rp1 ^= cur; else rp0 ^= cur;
        if (i & 0x02) rp3 ^= cur; else rp2 ^= cur;
        if (i & 0x04) rp5 ^= cur; else rp4 ^= cur;
        if (i & 0x08) rp7 ^= cur; else rp6 ^= cur;
        if (i & 0x10) rp9 ^= cur; else rp8 ^= cur;
        if (i & 0x20) rp11 ^= cur; else rp10 ^= cur;
        if (i & 0x40) rp13 ^= cur; else rp12 ^= cur;
        if (i & 0x80) rp15 ^= cur; else rp14 ^= cur;
    }
    code[0] =
        (parity[rp7] << 7) |
        (parity[rp6] << 6) |
        (parity[rp5] << 5) |
        (parity[rp4] << 4) |
        (parity[rp3] << 3) |
        (parity[rp2] << 2) |
        (parity[rp1] << 1) |
        (parity[rp0]);
    code[1] =
        (parity[rp15] << 7) |
        (parity[rp14] << 6) |
        (parity[rp13] << 5) |
        (parity[rp12] << 4) |
        (parity[rp11] << 3) |
        (parity[rp10] << 2) |
        (parity[rp9]  << 1) |
        (parity[rp8]);
    code[2] =
        (parity[par & 0xf0] << 7) |
        (parity[par & 0x0f] << 6) |
        (parity[par & 0xcc] << 5) |
        (parity[par & 0x33] << 4) |
        (parity[par & 0xaa] << 3) |
        (parity[par & 0x55] << 2);
    code[0] = ~code[0];
    code[1] = ~code[1];
    code[2] = ~code[2];
}
```

Still pretty straightforward. The last three invert statements are there to give a checksum of 0xff 0xff 0xff for an empty flash. In an

empty flash all data is 0xff, so the checksum then matches.

I also introduced the parity lookup. I expected this to be the fastest way to calculate the parity, but I will investigate alternatives later on.

Analysis 1

The code works, but is not terribly efficient. On my system it took almost 4 times as much time as the linux driver code. But hey, if it was *that* easy this would have been done long before. No pain. no gain.

Fortunately there is plenty of room for improvement.

In step 1 we moved from bit-wise calculation to byte-wise calculation. However in C we can also use the unsigned long data type and virtually every modern microprocessor supports 32 bit operations, so why not try to write our code in such a way that we process data in 32 bit chunks.

Of course this means some modification as the row parity is byte by byte. A quick analysis: for the column parity we use the par variable. When extending to 32 bits we can in the end easily calculate rp0 and rp1 from it. (because par now consists of 4 bytes, contributing to rp1, rp0, rp1, rp0 respectively, from MSB to LSB) also rp2 and rp3 can be easily retrieved from par as rp3 covers the first two MSBs and rp2 covers the last two LSBs.

Note that of course now the loop is executed only 64 times (256/4). And note that care must taken wrt byte ordering. The way bytes are ordered in a long is machine dependent, and might affect us. Anyway, if there is an issue: this code is developed on x86 (to be precise: a DELL PC with a D920 Intel CPU)

And of course the performance might depend on alignment, but I expect that the I/O buffers in the nand driver are aligned properly (and otherwise that should be fixed to get maximum performance).

Let's give it a try...

Attempt 2

```
extern const char parity[256];

void ecc2(const unsigned char *buf, unsigned char *code)
{
    int i;
    const unsigned long *bp = (unsigned long *)buf;
    unsigned long cur;
    unsigned long rp0, rp1, rp2, rp3, rp4, rp5, rp6, rp7;
    unsigned long rp8, rp9, rp10, rp11, rp12, rp13, rp14, rp15;
    unsigned long par;

    par = 0;
    rp0 = 0; rp1 = 0; rp2 = 0; rp3 = 0;
    rp4 = 0; rp5 = 0; rp6 = 0; rp7 = 0;
    rp8 = 0; rp9 = 0; rp10 = 0; rp11 = 0;
    rp12 = 0; rp13 = 0; rp14 = 0; rp15 = 0;

    for (i = 0; i < 64; i++)
    {
        cur = *bp++;
        par ^= cur;
        if (i & 0x01) rp5 ^= cur; else rp4 ^= cur;
        if (i & 0x02) rp7 ^= cur; else rp6 ^= cur;
        if (i & 0x04) rp9 ^= cur; else rp8 ^= cur;
        if (i & 0x08) rp11 ^= cur; else rp10 ^= cur;
        if (i & 0x10) rp13 ^= cur; else rp12 ^= cur;
        if (i & 0x20) rp15 ^= cur; else rp14 ^= cur;
    }
    /*
       we need to adapt the code generation for the fact that rp vars are now
       long; also the column parity calculation needs to be changed.
       we'll bring rp4 to 15 back to single byte entities by shifting and
       xoring
    */
    rp4 ^= (rp4 >> 16); rp4 ^= (rp4 >> 8); rp4 &= 0xff;
    rp5 ^= (rp5 >> 16); rp5 ^= (rp5 >> 8); rp5 &= 0xff;
    rp6 ^= (rp6 >> 16); rp6 ^= (rp6 >> 8); rp6 &= 0xff;
    rp7 ^= (rp7 >> 16); rp7 ^= (rp7 >> 8); rp7 &= 0xff;
    rp8 ^= (rp8 >> 16); rp8 ^= (rp8 >> 8); rp8 &= 0xff;
    rp9 ^= (rp9 >> 16); rp9 ^= (rp9 >> 8); rp9 &= 0xff;
    rp10 ^= (rp10 >> 16); rp10 ^= (rp10 >> 8); rp10 &= 0xff;
    rp11 ^= (rp11 >> 16); rp11 ^= (rp11 >> 8); rp11 &= 0xff;
    rp12 ^= (rp12 >> 16); rp12 ^= (rp12 >> 8); rp12 &= 0xff;
    rp13 ^= (rp13 >> 16); rp13 ^= (rp13 >> 8); rp13 &= 0xff;
    rp14 ^= (rp14 >> 16); rp14 ^= (rp14 >> 8); rp14 &= 0xff;
    rp15 ^= (rp15 >> 16); rp15 ^= (rp15 >> 8); rp15 &= 0xff;
```

```
        rp3 = (par >> 16); rp3 ^= (rp3 >> 8); rp3 &= 0xff;
        rp2 = par & 0xffff; rp2 ^= (rp2 >> 8); rp2 &= 0xff;
        par ^= (par >> 16);
        rp1 = (par >> 8); rp1 &= 0xff;
        rp0 = (par & 0xff);
        par ^= (par >> 8); par &= 0xff;

        code[0] =
            (parity[rp7] << 7) |
            (parity[rp6] << 6) |
            (parity[rp5] << 5) |
            (parity[rp4] << 4) |
            (parity[rp3] << 3) |
            (parity[rp2] << 2) |
            (parity[rp1] << 1) |
            (parity[rp0]);
        code[1] =
            (parity[rp15] << 7) |
            (parity[rp14] << 6) |
            (parity[rp13] << 5) |
            (parity[rp12] << 4) |
            (parity[rp11] << 3) |
            (parity[rp10] << 2) |
            (parity[rp9]  << 1) |
            (parity[rp8]);
        code[2] =
            (parity[par & 0xf0] << 7) |
            (parity[par & 0x0f] << 6) |
            (parity[par & 0xcc] << 5) |
            (parity[par & 0x33] << 4) |
            (parity[par & 0xaa] << 3) |
            (parity[par & 0x55] << 2);
        code[0] = ~code[0];
        code[1] = ~code[1];
        code[2] = ~code[2];
    }
```

The parity array is not shown any more. Note also that for these examples I kinda deviated from my regular programming style by allowing multiple statements on a line, not using { } in then and else blocks with only a single statement and by using operators like ^=

Analysis 2

The code (of course) works, and hurray: we are a little bit faster than the linux driver code (about 15%). But wait, don't cheer too quickly. There is more to be gained. If we look at e.g. rp14 and rp15 we see that we either xor our data with rp14 or with rp15. However we also have par which goes over all data. This means there is no need to calculate rp14 as it can be calculated from rp15 through rp14 = par ^ rp15, because par = rp14 ^ rp15; (or if desired we can avoid calculating rp15 and calculate it from rp14). That is why some places refer to inverse parity. Of course the same thing holds for rp4/5, rp6/7, rp8/9, rp10/11 and rp12/13. Effectively this means we can eliminate the else clause from the if statements. Also we can optimise the calculation in the end a little bit by going from long to byte first. Actually we can even avoid the table lookups

Attempt 3

Odd replaced:

```
    if (i & 0x01) rp5 ^= cur; else rp4 ^= cur;
    if (i & 0x02) rp7 ^= cur; else rp6 ^= cur;
    if (i & 0x04) rp9 ^= cur; else rp8 ^= cur;
    if (i & 0x08) rp11 ^= cur; else rp10 ^= cur;
    if (i & 0x10) rp13 ^= cur; else rp12 ^= cur;
    if (i & 0x20) rp15 ^= cur; else rp14 ^= cur;
```

with:

```
    if (i & 0x01) rp5 ^= cur;
    if (i & 0x02) rp7 ^= cur;
    if (i & 0x04) rp9 ^= cur;
    if (i & 0x08) rp11 ^= cur;
    if (i & 0x10) rp13 ^= cur;
    if (i & 0x20) rp15 ^= cur;
```

and outside the loop added:

```
    rp4  = par ^ rp5;
    rp6  = par ^ rp7;
    rp8  = par ^ rp9;
    rp10  = par ^ rp11;
    rp12  = par ^ rp13;
    rp14  = par ^ rp15;
```

And after that the code takes about 30% more time, although the number of statements is reduced. This is also reflected in the assembly code.

Analysis 3

Very weird. Guess it has to do with caching or instruction parallellism or so. I also tried on an eeePC (Celeron, clocked at 900 Mhz). Interesting observation was that this one is only 30% slower (according to time) executing the code as my 3Ghz D920 processor.

Well, it was expected not to be easy so maybe instead move to a different track: let's move back to the code from attempt2 and do some loop unrolling. This will eliminate a few if statements. I'll try different amounts of unrolling to see what works best.

Attempt 4

Unrolled the loop 1, 2, 3 and 4 times. For 4 the code starts with:

```
for (i = 0; i < 4; i++)
{
    cur = *bp++;
    par ^= cur;
    rp4 ^= cur;
    rp6 ^= cur;
    rp8 ^= cur;
    rp10 ^= cur;
    if (i & 0x1) rp13 ^= cur; else rp12 ^= cur;
    if (i & 0x2) rp15 ^= cur; else rp14 ^= cur;
    cur = *bp++;
    par ^= cur;
    rp5 ^= cur;
    rp6 ^= cur;
    ...
```

Analysis 4

Unrolling once gains about 15%

Unrolling twice keeps the gain at about 15%

Unrolling three times gives a gain of 30% compared to attempt 2.

Unrolling four times gives a marginal improvement compared to unrolling three times.

I decided to proceed with a four time unrolled loop anyway. It was my gut feeling that in the next steps I would obtain additional gain from it.

The next step was triggered by the fact that par contains the xor of all bytes and rp4 and rp5 each contain the xor of half of the bytes. So in effect par = rp4 ^ rp5. But as xor is commutative we can also say that rp5 = par ^ rp4. So no need to keep both rp4 and rp5 around. We can eliminate rp5 (or rp4, but I already foresaw another optimisation). The same holds for rp6/7, rp8/9, rp10/11 rp12/13 and rp14/15.

Attempt 5

Effectively so all odd digit rp assignments in the loop were removed. This included the else clause of the if statements. Of course after the loop we need to correct things by adding code like:

```
rp5 = par ^ rp4;
```

Also the initial assignments (rp5 = 0; etc) could be removed. Along the line I also removed the initialisation of rp0/1/2/3.

Analysis 5

Measurements showed this was a good move. The run-time roughly halved compared with attempt 4 with 4 times unrolled, and we only require 1/3rd of the processor time compared to the current code in the linux kernel.

However, still I thought there was more. I didn't like all the if statements. Why not keep a running parity and only keep the last if statement. Time for yet another version!

Attempt 6

THe code within the for loop was changed to:

```
for (i = 0; i < 4; i++)
{
    cur = *bp++; tmppar  = cur; rp4 ^= cur;
    cur = *bp++; tmppar ^= cur; rp6 ^= tmppar;
```

```
        cur = *bp++; tmppar ^= cur; rp4 ^= cur;
        cur = *bp++; tmppar ^= cur; rp8 ^= tmppar;

        cur = *bp++; tmppar ^= cur; rp4 ^= cur; rp6 ^= cur;
        cur = *bp++; tmppar ^= cur; rp6 ^= cur;
        cur = *bp++; tmppar ^= cur; rp4 ^= cur;
        cur = *bp++; tmppar ^= cur; rp10 ^= tmppar;

        cur = *bp++; tmppar ^= cur; rp4 ^= cur; rp6 ^= cur; rp8 ^= cur;
        cur = *bp++; tmppar ^= cur; rp6 ^= cur; rp8 ^= cur;
        cur = *bp++; tmppar ^= cur; rp4 ^= cur; rp8 ^= cur;
        cur = *bp++; tmppar ^= cur; rp8 ^= cur;

        cur = *bp++; tmppar ^= cur; rp4 ^= cur; rp6 ^= cur;
        cur = *bp++; tmppar ^= cur; rp6 ^= cur;
        cur = *bp++; tmppar ^= cur; rp4 ^= cur;
        cur = *bp++; tmppar ^= cur;

        par ^= tmppar;
        if ((i & 0x1) == 0) rp12 ^= tmppar;
        if ((i & 0x2) == 0) rp14 ^= tmppar;
    }
```

As you can see tmppar is used to accumulate the parity within a for iteration. In the last 3 statements is added to par and, if needed, to rp12 and rp14.

While making the changes I also found that I could exploit that tmppar contains the running parity for this iteration. So instead of having: rp4 ^= cur; rp6 ^= cur; I removed the rp6 ^= cur; statement and did rp6 ^= tmppar; on next statement. A similar change was done for rp8 and rp10

Analysis 6

Measuring this code again showed big gain. When executing the original linux code 1 million times, this took about 1 second on my system. (using time to measure the performance). After this iteration I was back to 0.075 sec. Actually I had to decide to start measuring over 10 million iterations in order not to lose too much accuracy. This one definitely seemed to be the jackpot!

There is a little bit more room for improvement though. There are three places with statements:

```
rp4 ^= cur; rp6 ^= cur;
```

It seems more efficient to also maintain a variable rp4_6 in the while loop; This eliminates 3 statements per loop. Of course after the loop we need to correct by adding:

```
rp4 ^= rp4_6;
rp6 ^= rp4_6
```

Furthermore there are 4 sequential assignments to rp8. This can be encoded slightly more efficiently by saving tmppar before those 4 lines and later do rp8 = rp8 ^ tmppar ^ notrp8; (where notrp8 is the value of rp8 before those 4 lines). Again a use of the commutative property of xor. Time for a new test!

Attempt 7

The new code now looks like:

```
for (i = 0; i < 4; i++)
{
    cur = *bp++; tmppar  = cur; rp4 ^= cur;
    cur = *bp++; tmppar ^= cur; rp6 ^= tmppar;
    cur = *bp++; tmppar ^= cur; rp4 ^= cur;
    cur = *bp++; tmppar ^= cur; rp8 ^= tmppar;

    cur = *bp++; tmppar ^= cur; rp4_6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp4 ^= cur;
    cur = *bp++; tmppar ^= cur; rp10 ^= tmppar;

    notrp8 = tmppar;
    cur = *bp++; tmppar ^= cur; rp4_6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp4 ^= cur;
    cur = *bp++; tmppar ^= cur;
    rp8 = rp8 ^ tmppar ^ notrp8;

    cur = *bp++; tmppar ^= cur; rp4_6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp4 ^= cur;
    cur = *bp++; tmppar ^= cur;
```

```
        par ^= tmppar;
        if ((i & 0x1) == 0) rp12 ^= tmppar;
        if ((i & 0x2) == 0) rp14 ^= tmppar;
    }
    rp4 ^= rp4_6;
    rp6 ^= rp4_6;
```

Not a big change, but every penny counts :-)

Analysis 7

Actually this made things worse. Not very much, but I don't want to move into the wrong direction. Maybe something to investigate later. Could have to do with caching again.

Guess that is what there is to win within the loop. Maybe unrolling one more time will help. I'll keep the optimisations from 7 for now.

Attempt 8

Unrolled the loop one more time.

Analysis 8

This makes things worse. Let's stick with attempt 6 and continue from there. Although it seems that the code within the loop cannot be optimised further there is still room to optimize the generation of the ecc codes. We can simply calculate the total parity. If this is 0 then rp4 = rp5 etc. If the parity is 1, then rp4 = !rp5;

But if rp4 = rp5 we do not need rp5 etc. We can just write the even bits in the result byte and then do something like:

```
    code[0] |= (code[0] << 1);
```

Lets test this.

Attempt 9

Changed the code but again this slightly degrades performance. Tried all kind of other things, like having dedicated parity arrays to avoid the shift after parity[rp7] << 7; No gain. Change the lookup using the parity array by using shift operators (e.g. replace parity[rp7] << 7 with:

```
    rp7 ^= (rp7 << 4);
    rp7 ^= (rp7 << 2);
    rp7 ^= (rp7 << 1);
    rp7 &= 0x80;
```

No gain.

The only marginal change was inverting the parity bits, so we can remove the last three invert statements.

Ah well, pity this does not deliver more. Then again 10 million iterations using the linux driver code takes between 13 and 13.5 seconds, whereas my code now takes about 0.73 seconds for those 10 million iterations. So basically I've improved the performance by a factor 18 on my system. Not that bad. Of course on different hardware you will get different results. No warranties!

But of course there is no such thing as a free lunch. The codesize almost tripled (from 562 bytes to 1434 bytes). Then again, it is not that much.

Correcting errors

For correcting errors I again used the ST application note as a starter, but I also peeked at the existing code.

The algorithm itself is pretty straightforward. Just xor the given and the calculated ecc. If all bytes are 0 there is no problem. If 11 bits are 1 we have one correctable bit error. If there is 1 bit 1, we have an error in the given ecc code.

It proved to be fastest to do some table lookups. Performance gain introduced by this is about a factor 2 on my system when a repair had to be done, and 1% or so if no repair had to be done.

Code size increased from 330 bytes to 686 bytes for this function. (gcc 4.2, -O3)

Conclusion

The gain when calculating the ecc is tremendous. Om my development hardware a speedup of a factor of 18 for ecc calculation was achieved. On a test on an embedded system with a MIPS core a factor 7 was obtained.

On a test with a Linksys NSLU2 (ARMv5TE processor) the speedup was a factor 5 (big endian mode, gcc 4.1.2, -O3)

For correction not much gain could be obtained (as bitflips are rare). Then again there are also much less cycles spent there.

It seems there is not much more gain possible in this, at least when programmed in C. Of course it might be possible to squeeze something more out of it with an assembler program, but due to pipeline behaviour etc this is very tricky (at least for intel hw).

Author: Frans Meulenbroeks