Speeding up arithmetic coding and decoding in the Schroedinger implementation of Dirac

Version 0.4

Issued: December 23, 2008

Thomas Davies, BBC Research

1 Introduction

This document explains the various methods used to design the implementation of arithmetic decoding in Schroedinger.

Arithmetic coding and decoding operations can conceivably be called many times per pixel, especially for intra frame coding, and so it is imperative that these processes are as quick as possible.

Throughout this document, reference is made to version 2.2 of the Dirac specification (http://www.diracvideo.org/specifications). Annex B.2 defines the operations of the arithmetic decoding engine. The reader should study this annex and refer to it whilst reading this document.

In this document we use C-style pseudocode, not the pseudocode as used in the specification. Variables in capital letters are deemed to be global variables.

2 General observations

Decoding a value consists of two basic operations: extracting a bit, i.e. determining a bit value and rescaling the interval; and renormalisation. These operations can be done in any order, since the renormalisation initiated after extracting one bit could be done just before extracting the next. In fact it is more efficient to do renormalisation first in the decoder since then the value extracted does not need to be stored during renormalisation. In other words the penultimate While loop in the table of B.2.4 could be moved to the beginning of that function.

Encoding a bit likewise has two parts, which are rescaling the interval and renormalisation. Both these processes, mathematically at least, mirror those in the decoder in that the bounds of the interval [LOW, HIGH) = [LOW, LOW+RANGE) must be the same in the decoder before decoding a value as they were in the encoder before encoding that same value.

Renormalisation itself (Section B.2.5 of the specification) appears to be very inefficient, since it inputs bits one at a time, and the loop itself is a While which could be mispredicted. Inside the loop, data is input one bit at a time, and there is a branch to account for straddle conditions. The original Schrodinger decoder renormalisation was more efficient in that data was read in a byte at a time but contained more branches as a result.

Encoder renormalisation is still less efficient in that sometimes bits are indeterminate due to straddle conditions (low and high are close, but fall either side of a bit or byte decision boundary). In this case it is usual to renormalise anyway, but keep track of the number of carry bits that may be required later. As we shall see, this makes the branch inside this loop even more complicated.

3 Common encoder and decoder optimisations

3.1 Probability LUT

The probability LUT consists of 256 values by which the context probabilities can be either incremented or decremented. This places the probability modification inside a branch, which is unfortunate:

```
if (value){
    prob0 -= LUT[prob0>>8];
}
else{
    prob0 += LUT[255-(prob0>>8)];
}
```

Although this branch can be combined with rescaling the interval, it is still costly. Also unfortunate is the fact that the requisite probability update values $LUT[prob0 \gg 8]$ and $LUT[255 - (prob0 \gg 8)]$ are at opposite end

of the table, which will be inefficient given a string of values in the same context.

So it is better to form a new LUT by:

```
for (i=0; i<256; ++i){
    LUT2[2*i] = LUT[255-i];
    LUT2[2*i+1] = -LUT[i];
}
```

Then probabilities can be updated by:

```
int lut_index = prob0>>7 & ~1;
prob0 += LUT2[lut_index | value];
```

outside of any branch.

3.2 Counting renormalisation iterations

It's possible also to remove the While loop from renormalisation, and compute from the size of RANGE how many renormalisations will be required, turning this into a fully deterministic for loop. So far this has proven not to be faster, but may do so in assembler versions.

RANGE	(RANGE-1)≫6	number of loops
63-64	0	9
65-128	1	8
129-256	2-3	7
257-512	4-7	6
513-1024	8-15	5
1025-2048	16-31	4
2049-4096	32-63	3
4097-8192	64-127	2
8193-16384	128-255	1
$\geq$ 16385	$\geq$ 256	0

Table 3.1: Number of loop iterations by size of RANGE

These values can be extracted either by using a LUT or by using an intrinsic bitscan operation on a processor to find the MSB of RANGE-1, which marks the threshold between loop iteration counts.

The only issue to be determined is that the loop count is indeed limited to 9, as range cannot get smaller than 63, and in particular cannot be less than on equal to 32.

To see this, observe that the smallest possible value of RANGE before renormalisation must be at least that obtained by taking the smallest possible value of RANGE *after* renormalisation, and applying either the lowest possible value of the 16-bit probability (in case 0 is coded) or the highest possible probability value (otherwise) to rescale RANGE.

The smallest possible value of RANGE after renormalisation is at least 0x4001.

The smallest possible 16-bit probability can be obtained from the probability LUT (section B.2.6 of the specification). This LUT shows decrements to the probability, and shows that if the value of the probability is ¡256, there can be no further decrements. So the smallest possible probability is

$$\min_{x \geq 256} (x - LUT(x \gg 8) = 254)$$

These values together give a rescaled RANGE of value 63.

Likewise, if 1 were coded and the probability were the maximum allowable, RANGE would be rescaled to be 64.

4 Decoder optimisations

4.1 Changing variables in the arithmetic decoder

It is possible to re-write the entire arithmetic decoder engine in terms of just two variables instead of three. Instead of maintaining CODE, LOW and RANGE, we maintain just CODEMINUSLOW and RANGE. It turns out that doing this enormously simplifies renormalisation.

To see that we can do this we must show three things. Firstly that we can extract a bit using just these values. Secondly we must prove that we can keep track of CODEMINUSLOW is such a way that in fact it will always equal CODE-LOW. Thirdly, we must show that this makes renormalisation simpler as claimed.

4.1.1 Extracting a bit

The first point is simple, since if we assume that CODEMINUSLOW=CODE-LOW, then assuming we have a context and probability value all we need to do is:

```
uint32_t rangexprob = (RANGE*prob0)>>16;
value = (CODEMINUSLOW>=RANGE);
if (value){
    CODEMINUSLOW -= rangexprob;
    RANGE -= rangexprob;
}
else{
    RANGE = rangexprob;
}
```

Note that we subtract the value of rangexprob from CODEMINUSLOW since LOW will have increased if we decode a 1.

4.1.2 Keeping track of CODEMINUSLOW through renormalisation

LOW is modified in extracting a bit, and we've seen that we had to modify CODEMINUSLOW in the opposite was there. Here the claim is that the renormalise function simply consists of:

```
RANGE <<= 1;
CODEMINUSLOW <<= 1;
CODEMINUSLOW |= read_bit();
```

which is hugely simpler.

The proof is by induction: if we initialise CODEMINUSLOW to the right value (i.e. the initial value of CODE, as LOW is 0) and we can show that for each renormalisation, if CODEMINUSLOW is equal to CODE-LOW before renormalisation, it still is afterwards, then the two values will always be equal.

To see this we need to break down the renormalise procedure in the specification into different cases. At the beginning, there is a complex IF statement that is true if we're in a straddle condition: that is, LOW is less that 1/2, represented as 0x8000 in the spec, and HIGH=LOW+RANGE-1 is greater than 1/2. In this case, the interval is doubled, expanding from 1/2 rather than from 0. This change in origin affects LOW and CODE by causing the 2nd MSB to be flipped prior to doubling.

Clearly, if the straddle condition is not activated then LOW and CODE are both doubled and a bit read into CODE to make the new difference equal to double the old difference plus the value of read_bit() - just what the new renormalisation does.

Hence if CODEMINUSLOW is equal to CODE-LOW before renormalisation, doubling it will maintain equality afterwards.

If we are in a straddle condition, then we know that LOW=01[bar] and HIGH=10[foo] for some 14-bit sequences [foo] and [bar]. So renormalisation does maps LOW to

2*LOW-0x8000= 0[bar]0

CODE lies somewhere between LOW and HIGH. So CODE could be 01[stg] or it could be 10[stg]. We don't know which, in general. If the former, then renormalisation also changes CODE to

2*CODE-0x8000 + read_bit() = 0[stg][read_bit()]

in which case the new value of CODE-LOW is precisely what has been calculated for CODEMINUSLOW as the 0x8000 values cancel.

The final case is when we're in a straddle but CODE=10[stg]. Here we flip the second bit to get 11[stg], then we double to get 11[stg]0, and finally we read in a bit and throw the top bit away (by &ing with 0xFFFF) to get 1[stg][read_bit()], which is equal to

2*CODE-0x8000+read_bit()

also. So yet again, CODEMINUSLOW tracks CODE-LOW.

4.2 Reading in multiple bits

Renormalisation can be further improved by only reading in a byte at a time, or even two bytes at a time.

In this case, CODEMINUSLOW now will be up to 24 bits wide – or 32 if two bytes are input – and will use the bottom byte or two as a buffer for holding future data in. To do this, a counter value COUNTER, maintaining a bit position modulo is needed, so as to determine when to pull in a whole new byte. Initialisation becomes (in the two-byte case):

```
RANGE=0xFFFF;
CODEMINUSLOW = DATA[0]<<24;
CODEMINUSLOW |= DATA[1]<<16;
CODEMINUSLOW |= DATA[2]<<8;
CODEMINUSLOW |= DATA[3];
COUNTER=16;
```

where DATA is the array of input bytes. y Then in extracting a bit we need the condition

```
uint32 rangexprob = (RANGE*prob0)>>16;
value = (CODEMINUSLOW>>16 >= rangexprob );
```

In rescaling the interval, we need to take account of the fact that CODEMINUSLOW is now sitting 16 bits up from its original position, so we have:

```
if (value){
    CODEMINUSLOW -= rangexprob<<16;
    RANGE -= rangexprob;
}
else{
    RANGE = rangexprob;
}
```

The renormalisation loop becomes:

```
while( RANGE<=0x4000)
{
    RANGE <<= 1;
    CODEMINUSLOW <<= 1;
```

```
    if (!--COUNTER){
        CODEMINUSLOW |= DATA[POS++]<<8;
        CODEMINUSLOW |= DATA[POS++];
        COUNTER = 16;
    }
}
```

4.3 Scaling up RANGE with multiple bit input

A further enhancement over the previous section can be obtained by also shifting range up by 16 bits so as to be commensurate in size with the new CODEMINUSLOW. This eliminates some shift instructions.

Decoder initialisation becomes:

```
RANGE=0xFFFF0000;
CODEMINUSLOW = DATA[0]<<24;
CODEMINUSLOW |= DATA[1]<<16;
CODEMINUSLOW |= DATA[2]<<8;
CODEMINUSLOW |= DATA[3];
COUNTER=16;
```

Extracting a bit must compensate for the size of RANGE, but we can leave rangexprob to be 16 bits larger and mask off the lower bits, giving,

```
uint32 rangexprob = ((RANGE>>16)*prob0)&0xFFFF0000;
value = (CODEMINUSLOW >= rangexprob );
```

Note that one might think that we should have

```
value = ((CODEMINUSLOW>>16) >= (rangexprob>>16) );
```

as this would be the equivalent condition to the original one. But the extra 16 bits at the lower end of CODEMINUSLOW are actual input code bits, and so merely give more informative on where the codeword lies in the interval and so the inequality is equivalent. Rescaling the interval becomes:

```
if (value){
    CODEMINUSLOW -= rangexprob;
    RANGE -= rangexprob;
}
else{
    RANGE = rangexprob;
}
```

and the renormalisation loop is:

```
while( RANGE<=0x40000000)
{
    RANGE <<= 1;
    CODEMINUSLOW <<= 1;

    if (!--COUNTER){
        CODEMINUSLOW |= DATA[POS++]<<8;
        CODEMINUSLOW |= DATA[POS++];
        COUNTER = 16;
    }
}
```

5 Encoder optimisations

The original Schrodinger arithmetic encoder improved on the Dirac research version by renormalising bytewise. In this way LOW accumulated 8 bits before a byte was output. This made straddle conditions rarer, since many would have resolved themselves by the time 8 bits were accumulated. It also made output much faster as there was no need to compute a postion in the current byte.

Before launching into describing improvements it's worth investigating the business of bytewise output and straddle conditions further.

5.1 Bytewise output and straddles

The original renormalisation function for the decoder is given in section B.2.5 of the specification. In the encoder, we have to keep track of "underflows" or carry bits also, where values are not yet resolved and can't be output. So we would get an encoder renormalisation function that looked like:

```
while ( RANGE <= 0x4000 ){
    if ( ( (LOW+RANGE-1)^LOW)>=(1<<15) ){
        LOW  ^= (1<<14);
        CARRIES++;
    }
    else{
        WriteBit( LOW & (1<<15));
        for (; CARRIES > 0; CARRIES-- )
            WriteBit(~LOW & (1<<15));
    }

    LOW  <<= 1;
    RANGE <<= 1;
    LOW   &= 0xFFFF;
}
```

What this says is: if there is no straddle, output the top bit (bit 15) followed by CARRIES number of opposite bits. If there is a straddle, flip the second bit (bit 14) and add to the number of carries. Finally, in both cases double LOW and RANGE and throw away the top bit of LOW. The effect of keeping track of the carries is to extend the dynamic range of the interval until the straddle is resolved. If it's resolved to 1 then we know we did CARRIES rescalings since the last bit output, so we must have CARRIES 0s following, and vice-versa.

To do this byte-wise, the Schrodinger code did:

```
while (RANGE <= 0x4000) {
    LOW <<= 1;
    RANGE <<= 1;

    if (!--COUNTER) {
        if (LOW < (1<<24) && (LOW + RANGE) >= (1<<24)) {
            CARRIES++;
        }
        else {
            if (LOW >= (1<<24)) {
                DATA[POS-1]++;
                while (CARRIES) {
                    DATA[POS] = 0x00;
                    CARRIES--;
                    POS++;
```

```
                }
            }
            else {
                while (CARRIES) {
                    DATA[POS] = 0xFF;
                    CARRIES--;
                    POS++;
                }
            }
            DATA[POS] = LOW >> 16;
            POS++;
        }

        LOW &= 0xFFFF;
        COUNTER = 8;
    }
}
```

First, some explanation. The doubling operations have been moved to the front so that the top bit is now bit 16. Next, we only output data when COUNTER has decremented 8 times down to zero, i.e. we have had 8 doublings. This now puts the top bit at bit 24. Bit 24 will be 1 if LOW is in the top half of the range, i.e. a 1 has just been resolved, and 0 otherwise.

The straddle condition now no longer implements a bit flip. Instead this has been absorbed into the condition where bit 24 is set, which *increments* the previous byte. The relationship with the original renormalisation is obscure and complicated to prove, but effectively it is operating bytewise, one bit in arrears.

5.2 Removing renormalisation branches

The renormalisation loop of the previous section is complicated and has lots of branches even though they are not taken all that often. The branches relate to detecting when carries are required and tiffling previous output bytes when they are resolved. Suppose instead that we always keep track of the relevant bytes of LOW, some of which we know will be wrong because of carries, but then modify them all at the end of encoding to make sure that the output is correct.

The point to note is that the output values of LOW are determined by doing an overlap-and-add of the top two bytes of LOW at each stage. This even works with carries, since when there is a straddle LOW will have the top 2 bytes equal to 0x00FF. If this is resolved to 1 later on, adding 1 will turn these two bytes to 0x0100 correctly.

So instead of recording instances of straddle, we can carry on outputting the OxFF bytes and correct them later by carrying 1s in an enormous overlapped sum.

So the new renormalisation merely stores the top two bytes of LOW, thus:

```
while(range<=0x4000){
    RANGE <<= 1;
    LOW <<= 1;
    if (!--COUNTER){
        LOWLIST[POS++] = (LOW>>16) & 0xFFFF;
        COUNTER=8;
        LOW &= 0xFFFF;
    }
}
```

At the end, we need to calculate the true output from all the stored values of LOW. The flush routine is as

follows:

```
LOW++;
LOW <<= COUNTER;

LOWLIST[POS++] = (LOW >> 16)&0xFFFF;
LOWLIST[POS++] = (LOW >> 8)&0xFFFF;

DATA[POS-1] = LOWLIST[POS-1] & 0xFF;
DATA[POS-2] = LOWLIST[POS-2] & 0xFF;

for (int i=POS-3; i>=0 ; --i ){
    LOWLIST[i] += LOWLIST[i+1]>>8;
    DATA[i] = LOWLIST[i]&0xFF;
}
```

The first instruction ensures that the final code value unambiguously lies inside the [LOW,LOW+RANGE) interval. The next instruction shifts the final value of LOW up to be commensurate with all the other values, which were output when COUNTER had reached zero. Then two more values are added to LOWLIST, to capture the remaining bytes 1 and 2 in LOW. These are the values that would be produced if the arithmetic encoder had carried on.

Finally, the arithmetic encoder implements the equivalent overlap and add that is performed on the fly in the old renormalisation loop.

This arrangement can be improved a little further by recording all of LOW instead of just the top two bytes. Then we lose the shift in the loop:

```
while(range<=0x4000){
    RANGE <<= 1;
    LOW <<= 1;
    if (!--COUNTER){
        LOWLIST[POS++] = LOW;
        COUNTER=8;
LOW &= 0xFFFF;
    }
}
```

and flushing can take this into account:

```
for (int i=POS-3; i>=0 ; --i ){
    LOWLIST[i] >>= 16;
    LOWLIST[i] += LOWLIST[i+1]>>8;
    DATA[i] = LOWLIST[i]&0xFF;
}
```

Furthermore, if we move the bit shift in LOW until after LOW is copied into LOWLIST then we have that at this point LOW occupies bits 0 to 23. If we then do this copy when COUNTER has decremented 16 times, LOW will occupy bits 0 to 31 and will not overflow a 32 bit register. Unfortunately, the final overlap and add will need to shift values by 15 instead of 16 bits so in software this may not be much faster.