

原 x264源代码简单分析：宏块分析（Analysis）部分-帧内宏块（Intra）

2015年05月22日 16:08:02 阅读数：9562

=====

H.264源代码分析文章列表：

【编码 - x264】

[x264源代码简单分析：概述](#)

[x264源代码简单分析：x264命令行工具（x264.exe）](#)

[x264源代码简单分析：编码器主干部分-1](#)

[x264源代码简单分析：编码器主干部分-2](#)

[x264源代码简单分析：x264_slice_write\(\)](#)

[x264源代码简单分析：滤波（Filter）部分](#)

[x264源代码简单分析：宏块分析（Analysis）部分-帧内宏块（Intra）](#)

[x264源代码简单分析：宏块分析（Analysis）部分-帧间宏块（Inter）](#)

[x264源代码简单分析：宏块编码（Encode）部分](#)

[x264源代码简单分析：熵编码（Entropy Encoding）部分](#)

[FFmpeg与libx264接口源代码简单分析](#)

【解码 - libavcodec H.264 解码器】

[FFmpeg的H.264解码器源代码简单分析：概述](#)

[FFmpeg的H.264解码器源代码简单分析：解析器（Parser）部分](#)

[FFmpeg的H.264解码器源代码简单分析：解码器主干部分](#)

[FFmpeg的H.264解码器源代码简单分析：熵解码（EntropyDecoding）部分](#)

[FFmpeg的H.264解码器源代码简单分析：宏块解码（Decode）部分-帧内宏块（Intra）](#)

[FFmpeg的H.264解码器源代码简单分析：宏块解码（Decode）部分-帧间宏块（Inter）](#)

[FFmpeg的H.264解码器源代码简单分析：环路滤波（Loop Filter）部分](#)

=====

本文记录x264的 x264_slice_write()函数中调用的x264_macroblock_analyse()的源代码。x264_macroblock_analyse()对应着x264中的分析模块。分析模块主要完成了下面2个方面的功能：

- （1）对于帧内宏块，分析帧内预测模式
- （2）对于帧间宏块，进行运动估计，分析帧间预测模式

由于分析模块比较复杂，因此分成两篇文章记录其中的源代码：本文记录帧内宏块预测模式的分析，下一篇文章记录帧间宏块预测模式的分析。

函数调用关系图

宏块分析（Analysis）部分的源代码在整个x264中的位置如下图所示。



[单击查看更清晰的图片](#)

宏块分析（Analysis）部分的函数调用关系如下图所示。



[单击查看更清晰的图片](#)

从图中可以看出，分析模块的x264_macroblock_analyse()调用了如下函数（只列举了几个有代表性的函数）：

x264_mb_analyse_init()：Analysis模块初始化。
x264_mb_analyse_intra()：Intra宏块帧内预测模式分析。
x264_macroblock_probe_pskip()：分析是否是skip模式。
x264_mb_analyse_inter_p16x16()：P16x16宏块帧间预测模式分析。
x264_mb_analyse_inter_p8x8()：P8x8宏块帧间预测模式分析。
x264_mb_analyse_inter_p16x8()：P16x8宏块帧间预测模式分析。
x264_mb_analyse_inter_b16x16()：B16x16宏块帧间预测模式分析。
x264_mb_analyse_inter_b8x8()：B8x8宏块帧间预测模式分析。
x264_mb_analyse_inter_b16x8()：B16x8宏块帧间预测模式分析。

本文重点分析其中帧内宏块（Intra宏块）的分析函数x264_mb_analyse_intra()。下一篇文章再对x264_mb_analyse_inter_p16x16()等一系列帧间宏块的分析函数。

x264_slice_write()

x264_slice_write()是x264项目的核心，它完成了编码了一个Slice的工作。有关该函数的分析可以参考文章《[x264源代码简单分析：x264_slice_write\(\)](#)》。本文分析其调用的x264_mb_analyse()函数。

x264_macroblock_analyse()

x264_macroblock_analyse()用于分析宏块的预测模式。该函数的定义位于encoder\analyse.c，如下所示。

```
[cpp]  
1.  /*****
2.  * 分析-帧内预测模式选择、帧间运动估计等
3.  *
4.  * 注释和处理：雷霄骅
5.  * http://blog.csdn.net/leixiaohua1020
6.  * leixiaohua1020@126.com
7.  *****/
8.  void x264_macroblock_analyse( x264_t *h )
9.  {
10.     x264_mb_analysis_t analysis;
11.     int i_cost = COST_MAX;
12.     //通过码率控制方法，获取本宏块QP
13.     h->mb.i_qp = x264_ratecontrol_mb_qp( h );
14.     /* If the QP of this MB is within 1 of the previous MB, code the same QP as the previous MB,
15.      * to lower the bit cost of the qp_delta.  Don't do this if QPRD is enabled. */
16.     if( h->param.rc.i_aq_mode && h->param.analyse.i_subpel_refine < 10 )
17.         h->mb.i_qp = abs(h->mb.i_qp - h->mb.i_last_qp) == 1 ? h->mb.i_last_qp : h->mb.i_qp;
18.
19.     if( h->param.analyse.b_mb_info )
20.         h->fdec->effective_qp[h->mb.i_mb_xy] = h->mb.i_qp; /* Store the real analysis QP. */
21.     //初始化
22.     x264_mb_analyse_init( h, &analysis, h->mb.i_qp );
23.
24.     /*----- Do the analysis -----*/
25.     //I帧：只使用帧内预测，分别计算亮度16x16（4种）和4x4（9种）所有模式的代价值，选出代价最小的模式
26.
27.     //P帧：计算帧内模式和帧间模式（ P Slice允许有Intra宏块和P宏块；同理B帧也支持Intra宏块）。
28.     //对P帧的每一种分割进行帧间预测，得到最佳的运动矢量及最佳匹配块。
29.     //帧间预测过程：选出最佳矢量——>找到最佳的整像素点——>找到最佳的二分之一像素点——>找到最佳的1/4像素点
30.     //然后取代价最小的为最佳MV和分割方式
31.     //最后从帧内模式和帧间模式中选择代价比较小的方式（有可能没有找到很好的匹配块，这时候就直接使用帧内预测而不是帧间预测）。
32.
33.     if( h->sh.i_type == SLICE_TYPE_I )
34.     {
35.         //I slice
36.         //通过一系列帧内预测模式（16x16的4种,4x4的9种）代价的计算得出代价最小的最优模式
37.         intra_analysis:
38.         if( analysis.i_mbrd )
39.             x264_mb_init_fenc_cache( h, analysis.i_mbrd >= 2 );
40.         //帧内预测分析
41.         //从16x16的SAD,4个8x8的SAD和,16个4x4SAD中选出最优方式
42.         x264_mb_analyse_intra( h, &analysis, COST_MAX );
43.         if( analysis.i_mbrd )
44.             x264_intra_rd( h, &analysis, COST_MAX );
45.         //分析结果都存储在analysis结构体中
46.         //开销
47.         i_cost = analysis.i_satd_i16x16;
48.         h->mb.i_type = I_16x16;
49.         //如果I4x4或者I8x8开销更小的话就拷贝
50.         //copy if little
51.         COPY2_IF_LT( i_cost, analysis.i_satd_i4x4, h->mb.i_type, I_4x4 );
52.         COPY2_IF_LT( i_cost, analysis.i_satd_i8x8, h->mb.i_type, I_8x8 );
53.         //画面极其特殊的时候，才有可能用到PCM
54.         if( analysis.i_satd_pcm < i_cost )
55.             h->mb.i_type = I_PCM;
56.
57.         else if( analysis.i_mbrd >= 2 )
58.             x264_intra_rd_refine( h, &analysis );
59.     }
```

```

59.     }
60.     else if( h->sh.i_type == SLICE_TYPE_P )
61.     {
62.         //P slice
63.
64.         int b_skip = 0;
65.
66.         h->mc.prefetch_ref( h->mb.pic.p_fref[0][0][h->mb.i_mb_x&3], h->mb.pic.i_stride[0], 0 );
67.
68.         analysis.b_try_skip = 0;
69.         if( analysis.b_force_intra )
70.         {
71.             if( !h->param.analyse.b_psy )
72.             {
73.                 x264_mb_analyse_init_qp( h, &analysis, X264_MAX( h->mb.i_qp - h->mb.ip_offset, h->param.rc.i_qp_min ) );
74.                 goto intra_analysis;
75.             }
76.         }
77.         else
78.         {
79.             /* Special fast-skip logic using information from mb_info. */
80.             if( h->fdec->mb_info && (h->fdec->mb_info[h->mb.i_mb_xy]&X264_MBINFO_CONSTANT) )
81.             {
82.                 if( !SLICE_MBAFF && (h->fdec->i_frame - h->fref[0][0]->i_frame) == 1 && !h->sh.b_weighted_pred &&
83.                     h->fref[0][0]->effective_qp[h->mb.i_mb_xy] <= h->mb.i_qp )
84.                 {
85.                     h->mb.i_partition = D_16x16;
86.                     /* Use the P-SKIP MV if we can... */
87.                     if( !M32(h->mb.cache.pskip_mv) )
88.                     {
89.                         b_skip = 1;
90.                         h->mb.i_type = P_SKIP;
91.                     }
92.                     /* Otherwise, just force a 16x16 block. */
93.                     else
94.                     {
95.                         h->mb.i_type = P_L0;
96.                         analysis.l0.me16x16.i_ref = 0;
97.                         M32( analysis.l0.me16x16.mv ) = 0;
98.                     }
99.                     goto skip_analysis;
100.                }
101.                /* Reset the information accordingly */
102.                else if( h->param.analyse.b_mb_info_update )
103.                {
104.                    h->fdec->mb_info[h->mb.i_mb_xy] &= ~X264_MBINFO_CONSTANT;
105.                }
106.                int skip_invalid = h->i_thread_frames > 1 && h->mb.cache.pskip_mv[1] > h->mb.mv_maxspel[1];
107.                /* If the current macroblock is off the frame, just skip it. */
108.                if( HAVE_INTERLACED && !MB_INTERLACED && h->mb.i_mb_y * 16 >= h->param.i_height && !skip_invalid )
109.                {
110.                    b_skip = 1;
111.                    /* Fast P-SKIP detection */
112.                    else if( h->param.analyse.b_fast_pskip )
113.                    {
114.                        if( skip_invalid )
115.                            // FIXME don't need to check this if the reference frame is done
116.                            {}
117.                        else if( h->param.analyse.i_subpel_refine >= 3 )
118.                            analysis.b_try_skip = 1;
119.                        else if( h->mb.i_mb_type_left[0] == P_SKIP ||
120.                            h->mb.i_mb_type_top == P_SKIP ||
121.                            h->mb.i_mb_type_topleft == P_SKIP ||
122.                            h->mb.i_mb_type_topright == P_SKIP )
123.                            b_skip = x264_macroblock_probe_pskip( h ); //检查是否是Skip类型
124.                    }
125.                }
126.                h->mc.prefetch_ref( h->mb.pic.p_fref[0][0][h->mb.i_mb_x&3], h->mb.pic.i_stride[0], 1 );
127.
128.                if( b_skip )
129.                {
130.                    h->mb.i_type = P_SKIP;
131.                    h->mb.i_partition = D_16x16;
132.                    assert( h->mb.cache.pskip_mv[1] <= h->mb.mv_maxspel[1] || h->i_thread_frames == 1 );
133.                    skip_analysis:
134.                    /* Set up MVs for future predictors */
135.                    for( int i = 0; i < h->mb.pic.i_fref[0]; i++ )
136.                        M32( h->mb.mvr[0][i][h->mb.i_mb_xy] ) = 0;
137.                }
138.                else
139.                {
140.                    const unsigned int flags = h->param.analyse.inter;
141.                    int i_type;
142.                    int i_partition;
143.                    int i_satd_inter, i_satd_intra;
144.
145.                    x264_mb_analyse_load_costs( h, &analysis );
146.                    /*
147.                     * 16x16 帧间预测宏块分析-P
148.                     *
149.                     * +-----+-----+
150.                     * |

```

```

151.
152.
153.
154.
155.
156.
157.
158.
159.
160.
161.
162.
163.
164.
165.
166.
167.
168.
169.
170.
171.
172.
173.
174.
175.
176.
177.
178.
179.
180.
181.
182.
183.
184.
185.
186.
187.
188.
189.
190.
191.
192.
193.
194.
195.
196.
197.
198.
199.
200.
201.
202.
203.
204.
205.
206.
207.
208.
209.
210.
211.
212.
213.
214.
215.
216.
217.
218.
219.
220.
221.
222.
223.
224.
225.
226.
227.
228.
229.
230.
231.
232.
233.
234.
235.
236.
237.
238.
239.
240.
241.
*/
x264_mb_analyse_inter_p16x16( h, &analysis );

if( h->mb.i_type == P_SKIP )
{
    for( int i = 1; i < h->mb.pic.i_fref[0]; i++ )
        M32( h->mb.mvr[0][i][h->mb.i_mb_xy] ) = 0;
    return;
}

if( flags & X264_ANALYSE_PSUB16x16 )
{
    if( h->param.analyse.b_mixed_references )
        x264_mb_analyse_inter_p8x8_mixed_ref( h, &analysis );
    else{
        /*
        * 8x8帧间预测宏块分析-P
        * +-----+
        * |         |
        * |         |
        * |         |
        * +-----+
        */
        x264_mb_analyse_inter_p8x8( h, &analysis );
    }
}

/* Select best inter mode */
i_type = P_L0;
i_partition = D_16x16;
i_cost = analysis.l0.me16x16.cost;

//如果8x8的代价值小于16x16
//则进行8x8子块分割的处理

//处理的数据源自于l0
if( ( flags & X264_ANALYSE_PSUB16x16 ) && (!analysis.b_early_terminate ||
analysis.l0.i_cost8x8 < analysis.l0.me16x16.cost) )
{
    i_type = P_8x8;
    i_partition = D_8x8;
    i_cost = analysis.l0.i_cost8x8;

    /* Do sub 8x8 */
    if( flags & X264_ANALYSE_PSUB8x8 )
    {
        for( int i = 0; i < 4; i++ )
        {
            //8x8块的子块的分析
            /*
            * 4x4
            * +-----+
            * |         |
            * +-----+
            * |         |
            * +-----+
            * |         |
            * +-----+
            */
            x264_mb_analyse_inter_p4x4( h, &analysis, i );
            int i_thresh8x8 = analysis.l0.me4x4[i][1].cost_mv + analysis.l0.me4x4[i][2].cost_mv;
            //如果4x4小于8x8
            //则再分析8x4, 4x8的代价
            if( !analysis.b_early_terminate || analysis.l0.i_cost4x4[i] < analysis.l0.me8x8[i].cost + i_thresh8x8 )
            {
                int i_cost8x8 = analysis.l0.i_cost4x4[i];
                h->mb.i_sub_partition[i] = D_L0_4x4;
                /*
                * 8x4
                * +-----+
                * |         |
                * +-----+
                * |         |
                * +-----+
                */
                //如果8x4小于8x8
                x264_mb_analyse_inter_p8x4( h, &analysis, i );
                COPY2_IF_LT( i_cost8x8, analysis.l0.i_cost8x4[i],
                    h->mb.i_sub_partition[i], D_L0_8x4 );
            }
            /*
            * 4x8
            * +-----+
            * |         |

```

```

242.         * +   +   +
243.         * |   |   |
244.         * +---+---+
245.         *
246.         */
247.         //如果4x8小于8x8
248.         x264_mb_analyse_inter_p4x8( h, &analysis, i );
249.         COPY2_IF_LT( i_cost8x8, analysis.l0.i_cost4x8[i],
250.                     h->mb.i_sub_partition[i], D_L0_4x8 );
251.
252.         i_cost += i_cost8x8 - analysis.l0.me8x8[i].cost;
253.     }
254.     x264_mb_cache_mv_p8x8( h, &analysis, i );
255. }
256. analysis.l0.i_cost8x8 = i_cost;
257. }
258. }
259.
260. /* Now do 16x8/8x16 */
261. int i_thresh16x8 = analysis.l0.me8x8[1].cost_mv + analysis.l0.me8x8[2].cost_mv;
262.
263. //前提要求8x8的代价值小于16x16
264. if( ( flags & X264_ANALYSE_PSUB16x16 ) && (!analysis.b_early_terminate ||
265.     analysis.l0.i_cost8x8 < analysis.l0.me16x16.cost + i_thresh16x8) )
266. {
267.     int i_avg_mv_ref_cost = (analysis.l0.me8x8[2].cost_mv + analysis.l0.me8x8[2].i_ref_cost
268.                             + analysis.l0.me8x8[3].cost_mv + analysis.l0.me8x8[3].i_ref_cost + 1) >> 1;
269.     analysis.i_cost_est16x8[1] = analysis.i_satd8x8[0][2] + analysis.i_satd8x8[0][3] + i_avg_mv_ref_cost;
270.     /*
271.      * 16x8 宏块划分
272.      *
273.      * +-----+-----+
274.      * |           |           |
275.      * |           |           |
276.      * |           |           |
277.      * +-----+-----+
278.      *
279.      */
280.     x264_mb_analyse_inter_p16x8( h, &analysis, i_cost );
281.     COPY3_IF_LT( i_cost, analysis.l0.i_cost16x8, i_type, P_L0, i_partition, D_16x8 );
282.
283.     i_avg_mv_ref_cost = (analysis.l0.me8x8[1].cost_mv + analysis.l0.me8x8[1].i_ref_cost
284.                         + analysis.l0.me8x8[3].cost_mv + analysis.l0.me8x8[3].i_ref_cost + 1) >> 1;
285.     analysis.i_cost_est8x16[1] = analysis.i_satd8x8[0][1] + analysis.i_satd8x8[0][3] + i_avg_mv_ref_cost;
286.     /*
287.      * 8x16 宏块划分
288.      *
289.      * +-----+
290.      * |           |
291.      * |           |
292.      * |           |
293.      * +-----+
294.      * |           |
295.      * |           |
296.      * |           |
297.      * +-----+
298.      *
299.      */
300.     x264_mb_analyse_inter_p8x16( h, &analysis, i_cost );
301.     COPY3_IF_LT( i_cost, analysis.l0.i_cost8x16, i_type, P_L0, i_partition, D_8x16 );
302. }
303.
304. h->mb.i_partition = i_partition;
305.
306. /* refine qpel */
307. //亚像素精度搜索
308. //FIXME mb_type costs?
309. if( analysis.i_mbrd || !h->mb.i_subpel_refine )
310. {
311.     /* refine later */
312. }
313. else if( i_partition == D_16x16 )
314. {
315.     x264_me_refine_qpel( h, &analysis.l0.me16x16 );
316.     i_cost = analysis.l0.me16x16.cost;
317. }
318. else if( i_partition == D_16x8 )
319. {
320.     x264_me_refine_qpel( h, &analysis.l0.me16x8[0] );
321.     x264_me_refine_qpel( h, &analysis.l0.me16x8[1] );
322.     i_cost = analysis.l0.me16x8[0].cost + analysis.l0.me16x8[1].cost;
323. }
324. else if( i_partition == D_8x16 )
325. {
326.     x264_me_refine_qpel( h, &analysis.l0.me8x16[0] );
327.     x264_me_refine_qpel( h, &analysis.l0.me8x16[1] );
328.     i_cost = analysis.l0.me8x16[0].cost + analysis.l0.me8x16[1].cost;
329. }
330. else if( i_partition == D_8x8 )
331. {
332.     i_cost = 0;

```

```

333.         for( int i8x8 = 0; i8x8 < 4; i8x8++ )
334.         {
335.             switch( h->mb.i_sub_partition[i8x8] )
336.             {
337.                 case D_L0_8x8:
338.                     x264_me_refine_qpel( h, &analysis.l0.me8x8[i8x8] );
339.                     i_cost += analysis.l0.me8x8[i8x8].cost;
340.                     break;
341.                 case D_L0_8x4:
342.                     x264_me_refine_qpel( h, &analysis.l0.me8x4[i8x8][0] );
343.                     x264_me_refine_qpel( h, &analysis.l0.me8x4[i8x8][1] );
344.                     i_cost += analysis.l0.me8x4[i8x8][0].cost +
345.                         analysis.l0.me8x4[i8x8][1].cost;
346.                     break;
347.                 case D_L0_4x8:
348.                     x264_me_refine_qpel( h, &analysis.l0.me4x8[i8x8][0] );
349.                     x264_me_refine_qpel( h, &analysis.l0.me4x8[i8x8][1] );
350.                     i_cost += analysis.l0.me4x8[i8x8][0].cost +
351.                         analysis.l0.me4x8[i8x8][1].cost;
352.                     break;
353.
354.                 case D_L0_4x4:
355.                     x264_me_refine_qpel( h, &analysis.l0.me4x4[i8x8][0] );
356.                     x264_me_refine_qpel( h, &analysis.l0.me4x4[i8x8][1] );
357.                     x264_me_refine_qpel( h, &analysis.l0.me4x4[i8x8][2] );
358.                     x264_me_refine_qpel( h, &analysis.l0.me4x4[i8x8][3] );
359.                     i_cost += analysis.l0.me4x4[i8x8][0].cost +
360.                         analysis.l0.me4x4[i8x8][1].cost +
361.                         analysis.l0.me4x4[i8x8][2].cost +
362.                         analysis.l0.me4x4[i8x8][3].cost;
363.                     break;
364.                 default:
365.                     x264_log( h, X264_LOG_ERROR, "internal error (!8x8 && !4x4)\n" );
366.                     break;
367.             }
368.         }
369.     }
370.
371.     if( h->mb.b_chroma_me )
372.     {
373.         if( CHROMA444 )
374.         {
375.             x264_mb_analyse_intra( h, &analysis, i_cost );
376.             x264_mb_analyse_intra_chroma( h, &analysis );
377.         }
378.         else
379.         {
380.             x264_mb_analyse_intra_chroma( h, &analysis );
381.             x264_mb_analyse_intra( h, &analysis, i_cost - analysis.i_satd_chroma );
382.         }
383.         analysis.i_satd_il6x16 += analysis.i_satd_chroma;
384.         analysis.i_satd_i8x8   += analysis.i_satd_chroma;
385.         analysis.i_satd_i4x4   += analysis.i_satd_chroma;
386.     }
387.     else
388.         x264_mb_analyse_intra( h, &analysis, i_cost ); //P Slice中也允许有Intra宏块, 所以也要进行分析
389.
390.     i_satd_inter = i_cost;
391.     i_satd_intra = X264_MIN3( analysis.i_satd_il6x16,
392.                             analysis.i_satd_i8x8,
393.                             analysis.i_satd_i4x4 );
394.
395.     if( analysis.i_mbrd )
396.     {
397.         x264_mb_analyse_p_rd( h, &analysis, X264_MIN(i_satd_inter, i_satd_intra) );
398.         i_type = P_L0;
399.         i_partition = D_16x16;
400.         i_cost = analysis.l0.i_rd16x16;
401.         COPY2_IF_LT( i_cost, analysis.l0.i_cost16x8, i_partition, D_16x8 );
402.         COPY2_IF_LT( i_cost, analysis.l0.i_cost8x16, i_partition, D_8x16 );
403.         COPY3_IF_LT( i_cost, analysis.l0.i_cost8x8, i_partition, D_8x8, i_type, P_8x8 );
404.         h->mb.i_type = i_type;
405.         h->mb.i_partition = i_partition;
406.         if( i_cost < COST_MAX )
407.             x264_mb_analyse_transform_rd( h, &analysis, &i_satd_inter, &i_cost );
408.         x264_intra_rd( h, &analysis, i_satd_inter * 5/4 + 1 );
409.     }
410.     //获取最小的代价
411.     COPY2_IF_LT( i_cost, analysis.i_satd_il6x16, i_type, I_16x16 );
412.     COPY2_IF_LT( i_cost, analysis.i_satd_i8x8, i_type, I_8x8 );
413.     COPY2_IF_LT( i_cost, analysis.i_satd_i4x4, i_type, I_4x4 );
414.     COPY2_IF_LT( i_cost, analysis.i_satd_pcm, i_type, I_PCM );
415.
416.     h->mb.i_type = i_type;
417.
418.     if( analysis.b_force_intra && !IS_INTRA(i_type) )
419.     {
420.         /* Intra masking: copy fdec to fenc and re-encode the block as intra in order to make it appear as if
421.          * it was an inter block. */
422.         x264_analyse_update_cache( h, &analysis );
423.         x264_macroblock_encode( h );

```

```

424.         for( int p = 0; p < (CHROMA444 ? 3 : 1); p++ )
425.             h->mc.copy[PIXEL_16x16]( h->mb.pic.p_fenc[p], FENC_STRIDE, h->mb.pic.p_fdec[p], FDEC_STRIDE, 16 );
426.         if( !CHROMA444 )
427.         {
428.             int height = 16 >> CHROMA_V_SHIFT;
429.             h->mc.copy[PIXEL_8x8] ( h->mb.pic.p_fenc[1], FENC_STRIDE, h->mb.pic.p_fdec[1], FDEC_STRIDE, height );
430.             h->mc.copy[PIXEL_8x8] ( h->mb.pic.p_fenc[2], FENC_STRIDE, h->mb.pic.p_fdec[2], FDEC_STRIDE, height );
431.         }
432.         x264_mb_analyse_init_qp( h, &analysis, X264_MAX( h->mb.i_qp - h->mb.ip_offset, h->param.rc.i_qp_min ) );
433.         goto intra_analysis;
434.     }
435.
436.     if( analysis.i_mbrd >= 2 && h->mb.i_type != I_PCM )
437.     {
438.         if( IS_INTRA( h->mb.i_type ) )
439.         {
440.             x264_intra_rd_refine( h, &analysis );
441.         }
442.         else if( i_partition == D_16x16 )
443.         {
444.             x264_macroblock_cache_ref( h, 0, 0, 4, 0, analysis.l0.me16x16.i_ref );
445.             analysis.l0.me16x16.cost = i_cost;
446.             x264_me_refine_qpel_rd( h, &analysis.l0.me16x16, analysis.i_lambda2, 0, 0 );
447.         }
448.         else if( i_partition == D_16x8 )
449.         {
450.             h->mb.i_sub_partition[0] = h->mb.i_sub_partition[1] =
451.             h->mb.i_sub_partition[2] = h->mb.i_sub_partition[3] = D_L0_8x8;
452.             x264_macroblock_cache_ref( h, 0, 0, 4, 2, 0, analysis.l0.me16x8[0].i_ref );
453.             x264_macroblock_cache_ref( h, 0, 2, 4, 2, 0, analysis.l0.me16x8[1].i_ref );
454.             x264_me_refine_qpel_rd( h, &analysis.l0.me16x8[0], analysis.i_lambda2, 0, 0 );
455.             x264_me_refine_qpel_rd( h, &analysis.l0.me16x8[1], analysis.i_lambda2, 8, 0 );
456.         }
457.         else if( i_partition == D_8x16 )
458.         {
459.             h->mb.i_sub_partition[0] = h->mb.i_sub_partition[1] =
460.             h->mb.i_sub_partition[2] = h->mb.i_sub_partition[3] = D_L0_8x8;
461.             x264_macroblock_cache_ref( h, 0, 0, 2, 4, 0, analysis.l0.me8x16[0].i_ref );
462.             x264_macroblock_cache_ref( h, 2, 0, 2, 4, 0, analysis.l0.me8x16[1].i_ref );
463.             x264_me_refine_qpel_rd( h, &analysis.l0.me8x16[0], analysis.i_lambda2, 0, 0 );
464.             x264_me_refine_qpel_rd( h, &analysis.l0.me8x16[1], analysis.i_lambda2, 4, 0 );
465.         }
466.         else if( i_partition == D_8x8 )
467.         {
468.             x264_analyse_update_cache( h, &analysis );
469.             for( int i8x8 = 0; i8x8 < 4; i8x8++ )
470.             {
471.                 if( h->mb.i_sub_partition[i8x8] == D_L0_8x8 )
472.                 {
473.                     x264_me_refine_qpel_rd( h, &analysis.l0.me8x8[i8x8], analysis.i_lambda2, i8x8*4, 0 );
474.                 }
475.                 else if( h->mb.i_sub_partition[i8x8] == D_L0_8x4 )
476.                 {
477.                     x264_me_refine_qpel_rd( h, &analysis.l0.me8x4[i8x8][0], analysis.i_lambda2, i8x8*4+0, 0 );
478.                     x264_me_refine_qpel_rd( h, &analysis.l0.me8x4[i8x8][1], analysis.i_lambda2, i8x8*4+2, 0 );
479.                 }
480.                 else if( h->mb.i_sub_partition[i8x8] == D_L0_4x8 )
481.                 {
482.                     x264_me_refine_qpel_rd( h, &analysis.l0.me4x8[i8x8][0], analysis.i_lambda2, i8x8*4+0, 0 );
483.                     x264_me_refine_qpel_rd( h, &analysis.l0.me4x8[i8x8][1], analysis.i_lambda2, i8x8*4+1, 0 );
484.                 }
485.                 else if( h->mb.i_sub_partition[i8x8] == D_L0_4x4 )
486.                 {
487.                     x264_me_refine_qpel_rd( h, &analysis.l0.me4x4[i8x8][0], analysis.i_lambda2, i8x8*4+0, 0 );
488.                     x264_me_refine_qpel_rd( h, &analysis.l0.me4x4[i8x8][1], analysis.i_lambda2, i8x8*4+1, 0 );
489.                     x264_me_refine_qpel_rd( h, &analysis.l0.me4x4[i8x8][2], analysis.i_lambda2, i8x8*4+2, 0 );
490.                     x264_me_refine_qpel_rd( h, &analysis.l0.me4x4[i8x8][3], analysis.i_lambda2, i8x8*4+3, 0 );
491.                 }
492.             }
493.         }
494.     }
495. }
496.
497. else if( h->sh.i_type == SLICE_TYPE_B ) //B Slice的时候
498. {
499.     int i_bskip_cost = COST_MAX;
500.     int b_skip = 0;
501.
502.     if( analysis.i_mbrd )
503.         x264_mb_init_fenc_cache( h, analysis.i_mbrd >= 2 );
504.
505.     h->mb.i_type = B_SKIP;
506.     if( h->mb.b_direct_auto_write )
507.     {
508.         /* direct=auto heuristic: prefer whichever mode allows more Skip macroblocks */
509.         for( int i = 0; i < 2; i++ )
510.         {
511.             int b_changed = 1;
512.             h->sh.b_direct_spatial_mv_pred ^= 1;
513.             analysis.b_direct_available = x264_mb_predict_mv_direct16x16( h, i && analysis.b_direct_available ? &b_changed : NULL

```

```

514.         if( analysis.b_direct_available )
515.         {
516.             if( b_changed )
517.             {
518.                 x264_mb_mc( h );
519.                 b_skip = x264_macroblock_probe_bskip( h );
520.             }
521.             h->stat.frame.i_direct_score[ h->sh.b_direct_spatial_mv_pred ] += b_skip;
522.         }
523.         else
524.             b_skip = 0;
525.     }
526. }
527. else
528.     analysis.b_direct_available = x264_mb_predict_mv_direct16x16( h, NULL );
529.
530. analysis.b_try_skip = 0;
531. if( analysis.b_direct_available )
532. {
533.     if( !h->mb.b_direct_auto_write )
534.         x264_mb_mc( h );
535.     /* If the current macroblock is off the frame, just skip it. */
536.     if( HAVE_INTERLACED && !MB_INTERLACED && h->mb.i_mb_y * 16 >= h->param.i_height )
537.         b_skip = 1;
538.     else if( analysis.i_mbrd )
539.     {
540.         i_bskip_cost = ssd_mb( h );
541.         /* 6 = minimum cavl cost of a non-skipped MB */
542.         b_skip = h->mb.b_skip_mc = i_bskip_cost <= ((6 * analysis.i_lambda2 + 128) >> 8);
543.     }
544.     else if( !h->mb.b_direct_auto_write )
545.     {
546.         /* Conditioning the probe on neighboring block types
547.          * doesn't seem to help speed or quality. */
548.         analysis.b_try_skip = x264_macroblock_probe_bskip( h );
549.         if( h->param.analyse.i_subpel_refine < 3 )
550.             b_skip = analysis.b_try_skip;
551.     }
552.     /* Set up MVs for future predictors */
553.     if( b_skip )
554.     {
555.         for( int i = 0; i < h->mb.pic.i_fref[0]; i++ )
556.             M32( h->mb.mvr[0][i][h->mb.i_mb_xy] ) = 0;
557.         for( int i = 0; i < h->mb.pic.i_fref[1]; i++ )
558.             M32( h->mb.mvr[1][i][h->mb.i_mb_xy] ) = 0;
559.     }
560. }
561.
562. if( !b_skip )
563. {
564.     const unsigned int flags = h->param.analyse.inter;
565.     int i_type;
566.     int i_partition;
567.     int i_satd_inter;
568.     h->mb.b_skip_mc = 0;
569.     h->mb.i_type = B_DIRECT;
570.
571.     x264_mb_analyse_load_costs( h, &analysis );
572.
573.     /* select best inter mode */
574.     /* direct must be first */
575.     if( analysis.b_direct_available )
576.         x264_mb_analyse_inter_direct( h, &analysis );
577.     /*
578.      * 16x16 帧间预测宏块分析-B
579.      *
580.      * +-----+-----+
581.      * |               |
582.      * |               |
583.      * |               |
584.      * |   +       +   |
585.      * |               |
586.      * |               |
587.      * |               |
588.      * +-----+-----+
589.      *
590.      */
591.     x264_mb_analyse_inter_b16x16( h, &analysis );
592.
593.     if( h->mb.i_type == B_SKIP )
594.     {
595.         for( int i = 1; i < h->mb.pic.i_fref[0]; i++ )
596.             M32( h->mb.mvr[0][i][h->mb.i_mb_xy] ) = 0;
597.         for( int i = 1; i < h->mb.pic.i_fref[1]; i++ )
598.             M32( h->mb.mvr[1][i][h->mb.i_mb_xy] ) = 0;
599.         return;
600.     }
601.
602.     i_type = B_L0_L0;
603.     i_partition = D_16x16;
604.     i_cost = analysis.l0.me16x16.cost;
605.     const int i_type = B_L1_L1;

```



```

605.     COPY2_IF_LT( i_cost, analysis.l1.me16x16.cost, i_type, B_L1_L1 );
606.     COPY2_IF_LT( i_cost, analysis.i_cost16x16bi, i_type, B_BI_BI );
607.     COPY2_IF_LT( i_cost, analysis.i_cost16x16direct, i_type, B_DIRECT );
608.
609.     if( analysis.i_mbrd && analysis.b_early_terminate && analysis.i_cost16x16direct <= i_cost * 33/32 )
610.     {
611.         x264_mb_analyse_b_rd( h, &analysis, i_cost );
612.         if( i_bskip_cost < analysis.i_rd16x16direct &&
613.             i_bskip_cost < analysis.i_rd16x16bi &&
614.             i_bskip_cost < analysis.l0.i_rd16x16 &&
615.             i_bskip_cost < analysis.l1.i_rd16x16 )
616.         {
617.             h->mb.i_type = B_SKIP;
618.             x264_analyse_update_cache( h, &analysis );
619.             return;
620.         }
621.     }
622.
623.     if( flags & X264_ANALYSE_BSUB16x16 )
624.     {
625.
626.         /*
627.          * 8x8 帧间预测宏块分析-B
628.          * +-----+
629.          * |         |
630.          * |         |
631.          * |         |
632.          * +-----+
633.          *
634.          */
635.
636.         if( h->param.analyse.b_mixed_references )
637.             x264_mb_analyse_inter_b8x8_mixed_ref( h, &analysis );
638.         else
639.             x264_mb_analyse_inter_b8x8( h, &analysis );
640.
641.         COPY3_IF_LT( i_cost, analysis.i_cost8x8bi, i_type, B_8x8, i_partition, D_8x8 );
642.
643.         /* Try to estimate the cost of b16x8/b8x16 based on the satd scores of the b8x8 modes */
644.         int i_cost_est16x8bi_total = 0, i_cost_est8x16bi_total = 0;
645.         int i_mb_type, i_partition16x8[2], i_partition8x16[2];
646.         for( int i = 0; i < 2; i++ )
647.         {
648.             int avg_l0_mv_ref_cost, avg_l1_mv_ref_cost;
649.             int i_l0_satd, i_l1_satd, i_bi_satd, i_best_cost;
650.             // 16x8
651.             i_best_cost = COST_MAX;
652.             i_l0_satd = analysis.i_satd8x8[0][i*2] + analysis.i_satd8x8[0][i*2+1];
653.             i_l1_satd = analysis.i_satd8x8[1][i*2] + analysis.i_satd8x8[1][i*2+1];
654.             i_bi_satd = analysis.i_satd8x8[2][i*2] + analysis.i_satd8x8[2][i*2+1];
655.             avg_l0_mv_ref_cost = ( analysis.l0.me8x8[i*2].cost_mv + analysis.l0.me8x8[i*2].i_ref_cost
656.                                   + analysis.l0.me8x8[i*2+1].cost_mv + analysis.l0.me8x8[i*2+1].i_ref_cost + 1 ) >> 1;
657.             avg_l1_mv_ref_cost = ( analysis.l1.me8x8[i*2].cost_mv + analysis.l1.me8x8[i*2].i_ref_cost
658.                                   + analysis.l1.me8x8[i*2+1].cost_mv + analysis.l1.me8x8[i*2+1].i_ref_cost + 1 ) >> 1;
659.             COPY2_IF_LT( i_best_cost, i_l0_satd + avg_l0_mv_ref_cost, i_partition16x8[i], D_L0_8x8 );
660.             COPY2_IF_LT( i_best_cost, i_l1_satd + avg_l1_mv_ref_cost, i_partition16x8[i], D_L1_8x8 );
661.             COPY2_IF_LT( i_best_cost, i_bi_satd + avg_l0_mv_ref_cost + avg_l1_mv_ref_cost, i_partition16x8[i], D_BI_8x8 );
662.             analysis.i_cost_est16x8[i] = i_best_cost;
663.
664.             // 8x16
665.             i_best_cost = COST_MAX;
666.             i_l0_satd = analysis.i_satd8x8[0][i] + analysis.i_satd8x8[0][i+2];
667.             i_l1_satd = analysis.i_satd8x8[1][i] + analysis.i_satd8x8[1][i+2];
668.             i_bi_satd = analysis.i_satd8x8[2][i] + analysis.i_satd8x8[2][i+2];
669.             avg_l0_mv_ref_cost = ( analysis.l0.me8x8[i].cost_mv + analysis.l0.me8x8[i].i_ref_cost
670.                                   + analysis.l0.me8x8[i+2].cost_mv + analysis.l0.me8x8[i+2].i_ref_cost + 1 ) >> 1;
671.             avg_l1_mv_ref_cost = ( analysis.l1.me8x8[i].cost_mv + analysis.l1.me8x8[i].i_ref_cost
672.                                   + analysis.l1.me8x8[i+2].cost_mv + analysis.l1.me8x8[i+2].i_ref_cost + 1 ) >> 1;
673.             COPY2_IF_LT( i_best_cost, i_l0_satd + avg_l0_mv_ref_cost, i_partition8x16[i], D_L0_8x8 );
674.             COPY2_IF_LT( i_best_cost, i_l1_satd + avg_l1_mv_ref_cost, i_partition8x16[i], D_L1_8x8 );
675.             COPY2_IF_LT( i_best_cost, i_bi_satd + avg_l0_mv_ref_cost + avg_l1_mv_ref_cost, i_partition8x16[i], D_BI_8x8 );
676.             analysis.i_cost_est8x16[i] = i_best_cost;
677.         }
678.         i_mb_type = B_L0_L0 + (i_partition16x8[0]>>2) * 3 + (i_partition16x8[1]>>2);
679.         analysis.i_cost_est16x8[1] += analysis.i_lambda * i_mb_b16x8_cost_table[i_mb_type];
680.         i_cost_est16x8bi_total = analysis.i_cost_est16x8[0] + analysis.i_cost_est16x8[1];
681.         i_mb_type = B_L0_L0 + (i_partition8x16[0]>>2) * 3 + (i_partition8x16[1]>>2);
682.         analysis.i_cost_est8x16[1] += analysis.i_lambda * i_mb_b16x8_cost_table[i_mb_type];
683.         i_cost_est8x16bi_total = analysis.i_cost_est8x16[0] + analysis.i_cost_est8x16[1];
684.
685.         /* We can gain a little speed by checking the mode with the lowest estimated cost first */
686.         int try_16x8_first = i_cost_est16x8bi_total < i_cost_est8x16bi_total;
687.         if( try_16x8_first && (!analysis.b_early_terminate || i_cost_est16x8bi_total < i_cost) )
688.         {
689.             x264_mb_analyse_inter_b16x8( h, &analysis, i_cost );
690.             COPY3_IF_LT( i_cost, analysis.i_cost16x8bi, i_type, analysis.i_mb_type16x8, i_partition, D_16x8 );
691.         }
692.         if( !analysis.b_early_terminate || i_cost_est8x16bi_total < i_cost )
693.         {
694.             x264_mb_analyse_inter_b8x16( h, &analysis, i_cost );
695.             COPY3_IF_LT( i_cost, analysis.i_cost8x16bi, i_type, analysis.i_mb_type8x16, i_partition, D_8x16 );
696.         }

```

```

697.         if( !try_16x8_first && (!analysis.b_early_terminate || i_cost_est16x8bi_total < i_cost) )
698.         {
699.             x264_mb_analyse_inter_b16x8( h, &analysis, i_cost );
700.             COPY3_IF_LT( i_cost, analysis.i_cost16x8bi, i_type, analysis.i_mb_type16x8, i_partition, D_16x8 );
701.         }
702.     }
703.
704.     if( analysis.i_mbrd || !h->mb.i_subpel_refine )
705.     {
706.         /* refine later */
707.     }
708.     /* refine qpel */
709.     else if( i_partition == D_16x16 )
710.     {
711.         analysis.l0.me16x16.cost -= analysis.i_lambda * i_mb_b_cost_table[B_L0_L0];
712.         analysis.l1.me16x16.cost -= analysis.i_lambda * i_mb_b_cost_table[B_L1_L1];
713.         if( i_type == B_L0_L0 )
714.         {
715.             x264_me_refine_qpel( h, &analysis.l0.me16x16 );
716.             i_cost = analysis.l0.me16x16.cost
717.                 + analysis.i_lambda * i_mb_b_cost_table[B_L0_L0];
718.         }
719.         else if( i_type == B_L1_L1 )
720.         {
721.             x264_me_refine_qpel( h, &analysis.l1.me16x16 );
722.             i_cost = analysis.l1.me16x16.cost
723.                 + analysis.i_lambda * i_mb_b_cost_table[B_L1_L1];
724.         }
725.         else if( i_type == B_BI_BI )
726.         {
727.             x264_me_refine_qpel( h, &analysis.l0.bi16x16 );
728.             x264_me_refine_qpel( h, &analysis.l1.bi16x16 );
729.         }
730.     }
731.     else if( i_partition == D_16x8 )
732.     {
733.         for( int i = 0; i < 2; i++ )
734.         {
735.             if( analysis.i_mb_partition16x8[i] != D_L1_8x8 )
736.                 x264_me_refine_qpel( h, &analysis.l0.me16x8[i] );
737.             if( analysis.i_mb_partition16x8[i] != D_L0_8x8 )
738.                 x264_me_refine_qpel( h, &analysis.l1.me16x8[i] );
739.         }
740.     }
741.     else if( i_partition == D_8x16 )
742.     {
743.         for( int i = 0; i < 2; i++ )
744.         {
745.             if( analysis.i_mb_partition8x16[i] != D_L1_8x8 )
746.                 x264_me_refine_qpel( h, &analysis.l0.me8x16[i] );
747.             if( analysis.i_mb_partition8x16[i] != D_L0_8x8 )
748.                 x264_me_refine_qpel( h, &analysis.l1.me8x16[i] );
749.         }
750.     }
751.     else if( i_partition == D_8x8 )
752.     {
753.         for( int i = 0; i < 4; i++ )
754.         {
755.             x264_me_t *m;
756.             int i_part_cost_old;
757.             int i_type_cost;
758.             int i_part_type = h->mb.i_sub_partition[i];
759.             int b_bidir = (i_part_type == D_BI_8x8);
760.
761.             if( i_part_type == D_DIRECT_8x8 )
762.                 continue;
763.             if( x264_mb_partition_listX_table[0][i_part_type] )
764.             {
765.                 m = &analysis.l0.me8x8[i];
766.                 i_part_cost_old = m->cost;
767.                 i_type_cost = analysis.i_lambda * i_sub_mb_b_cost_table[D_L0_8x8];
768.                 m->cost -= i_type_cost;
769.                 x264_me_refine_qpel( h, m );
770.                 if( !b_bidir )
771.                     analysis.i_cost8x8bi += m->cost + i_type_cost - i_part_cost_old;
772.             }
773.             if( x264_mb_partition_listX_table[1][i_part_type] )
774.             {
775.                 m = &analysis.l1.me8x8[i];
776.                 i_part_cost_old = m->cost;
777.                 i_type_cost = analysis.i_lambda * i_sub_mb_b_cost_table[D_L1_8x8];
778.                 m->cost -= i_type_cost;
779.                 x264_me_refine_qpel( h, m );
780.                 if( !b_bidir )
781.                     analysis.i_cost8x8bi += m->cost + i_type_cost - i_part_cost_old;
782.             }
783.             /* TODO: update mvp? */
784.         }
785.     }
786.
787.     i_satd_inter = i_cost;

```

```

788.
789.     if( analysis.i_mbrd )
790.     {
791.         x264_mb_analyse_b_rd( h, &analysis, i_satd_inter );
792.         i_type = B_SKIP;
793.         i_cost = i_bskip_cost;
794.         i_partition = D_16x16;
795.         COPY2_IF_LT( i_cost, analysis.l0.i_rd16x16, i_type, B_L0_L0 );
796.         COPY2_IF_LT( i_cost, analysis.l1.i_rd16x16, i_type, B_L1_L1 );
797.         COPY2_IF_LT( i_cost, analysis.i_rd16x16bi, i_type, B_BI_BI );
798.         COPY2_IF_LT( i_cost, analysis.i_rd16x16direct, i_type, B_DIRECT );
799.         COPY3_IF_LT( i_cost, analysis.i_rd16x8bi, i_type, analysis.i_mb_type16x8, i_partition, D_16x8 );
800.         COPY3_IF_LT( i_cost, analysis.i_rd8x16bi, i_type, analysis.i_mb_type8x16, i_partition, D_8x16 );
801.         COPY3_IF_LT( i_cost, analysis.i_rd8x8bi, i_type, B_8x8, i_partition, D_8x8 );
802.
803.         h->mb.i_type = i_type;
804.         h->mb.i_partition = i_partition;
805.     }
806.
807.     if( h->mb.b_chroma_me )
808.     {
809.         if( CHROMA444 )
810.         {
811.             x264_mb_analyse_intra( h, &analysis, i_satd_inter );
812.             x264_mb_analyse_intra_chroma( h, &analysis );
813.         }
814.         else
815.         {
816.             x264_mb_analyse_intra_chroma( h, &analysis );
817.             x264_mb_analyse_intra( h, &analysis, i_satd_inter - analysis.i_satd_chroma );
818.         }
819.         analysis.i_satd_i16x16 += analysis.i_satd_chroma;
820.         analysis.i_satd_i8x8   += analysis.i_satd_chroma;
821.         analysis.i_satd_i4x4   += analysis.i_satd_chroma;
822.     }
823.     else
824.     {
825.         x264_mb_analyse_intra( h, &analysis, i_satd_inter );
826.     }
827.     if( analysis.i_mbrd )
828.     {
829.         x264_mb_analyse_transform_rd( h, &analysis, &i_satd_inter, &i_cost );
830.         x264_intra_rd( h, &analysis, i_satd_inter * 17/16 + 1 );
831.     }
832.
833.     COPY2_IF_LT( i_cost, analysis.i_satd_i16x16, i_type, I_16x16 );
834.     COPY2_IF_LT( i_cost, analysis.i_satd_i8x8, i_type, I_8x8 );
835.     COPY2_IF_LT( i_cost, analysis.i_satd_i4x4, i_type, I_4x4 );
836.     COPY2_IF_LT( i_cost, analysis.i_satd_pcm, i_type, I_PCM );
837.
838.     h->mb.i_type = i_type;
839.     h->mb.i_partition = i_partition;
840.
841.     if( analysis.i_mbrd >= 2 && IS_INTRA( i_type ) && i_type != I_PCM )
842.         x264_intra_rd_refine( h, &analysis );
843.     if( h->mb.i_subpel_refine >= 5 )
844.         x264_refine_bidir( h, &analysis );
845.
846.     if( analysis.i_mbrd >= 2 && i_type > B_DIRECT && i_type < B_SKIP )
847.     {
848.         int i_biweight;
849.         x264_analyse_update_cache( h, &analysis );
850.
851.         if( i_partition == D_16x16 )
852.         {
853.             if( i_type == B_L0_L0 )
854.             {
855.                 analysis.l0.me16x16.cost = i_cost;
856.                 x264_me_refine_qpel_rd( h, &analysis.l0.me16x16, analysis.i_lambda2, 0, 0 );
857.             }
858.             else if( i_type == B_L1_L1 )
859.             {
860.                 analysis.l1.me16x16.cost = i_cost;
861.                 x264_me_refine_qpel_rd( h, &analysis.l1.me16x16, analysis.i_lambda2, 0, 1 );
862.             }
863.             else if( i_type == B_BI_BI )
864.             {
865.                 i_biweight = h->mb.bipred_weight[analysis.l0.bi16x16.i_ref][analysis.l1.bi16x16.i_ref];
866.                 x264_me_refine_bidir_rd( h, &analysis.l0.bi16x16, &analysis.l1.bi16x16, i_biweight, 0, analysis.i_lambda2 );
867.             }
868.         }
869.         else if( i_partition == D_16x8 )
870.         {
871.             for( int i = 0; i < 2; i++ )
872.             {
873.                 h->mb.i_sub_partition[i*2] = h->mb.i_sub_partition[i*2+1] = analysis.i_mb_partition16x8[i];
874.                 if( analysis.i_mb_partition16x8[i] == D_L0_8x8 )
875.                     x264_me_refine_qpel_rd( h, &analysis.l0.me16x8[i], analysis.i_lambda2, i*8, 0 );
876.                 else if( analysis.i_mb_partition16x8[i] == D_L1_8x8 )
877.                     x264_me_refine_qpel_rd( h, &analysis.l1.me16x8[i], analysis.i_lambda2, i*8, 1 );
878.                 else if( analysis.i_mb_partition16x8[i] == D_BI_8x8 )
879.                 {

```

```

879.         i_biweight = h->mb.bipred_weight[analysis.l0.me16x8[i].i_ref][analysis.l1.me16x8[i].i_ref];
880.         x264_me_refine_bidir_rd( h, &analysis.l0.me16x8[i], &analysis.l1.me16x8[i], i_biweight, i*2, analysis.i_l
da2 );
881.     }
882. }
883. }
884. else if( i_partition == D_8x16 )
885. {
886.     for( int i = 0; i < 2; i++ )
887.     {
888.         h->mb.i_sub_partition[i] = h->mb.i_sub_partition[i+2] = analysis.i_mb_partition8x16[i];
889.         if( analysis.i_mb_partition8x16[i] == D_L0_8x8 )
890.             x264_me_refine_qpel_rd( h, &analysis.l0.me8x16[i], analysis.i_lambda2, i*4, 0 );
891.         else if( analysis.i_mb_partition8x16[i] == D_L1_8x8 )
892.             x264_me_refine_qpel_rd( h, &analysis.l1.me8x16[i], analysis.i_lambda2, i*4, 1 );
893.         else if( analysis.i_mb_partition8x16[i] == D_BI_8x8 )
894.         {
895.             i_biweight = h->mb.bipred_weight[analysis.l0.me8x16[i].i_ref][analysis.l1.me8x16[i].i_ref];
896.             x264_me_refine_bidir_rd( h, &analysis.l0.me8x16[i], &analysis.l1.me8x16[i], i_biweight, i, analysis.i_lam
2 );
897.         }
898.     }
899. }
900. else if( i_partition == D_8x8 )
901. {
902.     for( int i = 0; i < 4; i++ )
903.     {
904.         if( h->mb.i_sub_partition[i] == D_L0_8x8 )
905.             x264_me_refine_qpel_rd( h, &analysis.l0.me8x8[i], analysis.i_lambda2, i*4, 0 );
906.         else if( h->mb.i_sub_partition[i] == D_L1_8x8 )
907.             x264_me_refine_qpel_rd( h, &analysis.l1.me8x8[i], analysis.i_lambda2, i*4, 1 );
908.         else if( h->mb.i_sub_partition[i] == D_BI_8x8 )
909.         {
910.             i_biweight = h->mb.bipred_weight[analysis.l0.me8x8[i].i_ref][analysis.l1.me8x8[i].i_ref];
911.             x264_me_refine_bidir_rd( h, &analysis.l0.me8x8[i], &analysis.l1.me8x8[i], i_biweight, i, analysis.i_lambda
);
912.         }
913.     }
914. }
915. }
916. }
917. }
918.
919. x264_analyse_update_cache( h, &analysis );
920.
921. /* In rare cases we can end up qpel-RDing our way back to a larger partition size
922.  * without realizing it. Check for this and account for it if necessary. */
923. if( analysis.i_mbrd >= 2 )
924. {
925.     /* Don't bother with bipred or 8x8-and-below, the odds are incredibly low. */
926.     static const uint8_t check_mv_lists[X264_MBTYPE_MAX] = {[P_L0]=1, [B_L0_L0]=1, [B_L1_L1]=2};
927.     int list = check_mv_lists[h->mb.i_type] - 1;
928.     if( list >= 0 && h->mb.i_partition != D_16x16 &&
929.         M32( &h->mb.cache.mv[list][x264_scan8[0]] ) == M32( &h->mb.cache.mv[list][x264_scan8[12]] ) &&
930.         h->mb.cache.ref[list][x264_scan8[0]] == h->mb.cache.ref[list][x264_scan8[12]] )
931.         h->mb.i_partition = D_16x16;
932. }
933.
934. if( !analysis.i_mbrd )
935.     x264_mb_analyse_transform( h );
936.
937. if( analysis.i_mbrd == 3 && !IS_SKIP(h->mb.i_type) )
938.     x264_mb_analyse_qp_rd( h, &analysis );
939.
940. h->mb.b_trellis = h->param.analyse.i_trellis;
941. h->mb.b_noise_reduction = h->mb.b_noise_reduction || (!h->param.analyse.i_noise_reduction && !IS_INTRA( h->mb.i_type ));
942.
943. if( !IS_SKIP(h->mb.i_type) && h->mb.i_psy_trellis && h->param.analyse.i_trellis == 1 )
944.     x264_psy_trellis_init( h, 0 );
945. if( h->mb.b_trellis == 1 || h->mb.b_noise_reduction )
946.     h->mb.i_skip_intra = 0;
947. }

```

尽管x264_macroblock_analyse()的源代码比较长，但是它的逻辑比较清晰，如下所示：

- (1) 如果当前是I Slice，调用x264_mb_analyse_intra()进行Intra宏块的帧内预测模式分析。
- (2) 如果当前是P Slice，则进行下面流程的分析：
 - a)调用x264_macroblock_probe_pskip()分析是否为Skip宏块，如果是的话则不再进行下面分析。
 - b)调用x264_mb_analyse_inter_p16x16()分析P16x16帧间预测的代价。
 - c)调用x264_mb_analyse_inter_p8x8()分析P8x8帧间预测的代价。
 - d)如果P8x8代价值小于P16x16，则依次对4个8x8的子宏块分割进行判断：
 - i.调用x264_mb_analyse_inter_p4x4()分析P4x4帧间预测的代价。
 - ii.如果P4x4代价值小于P8x8，则调用 x264_mb_analyse_inter_p8x4()和x264_mb_analyse_inter_p4x8()分析P8x4和P4x8帧间预测的代价。
 - e)如果P8x8代价值小于P16x16，调用x264_mb_analyse_inter_p16x8()和x264_mb_analyse_inter_p8x16()分析P16x8和P8x16帧间预

测的代价。

f)此外还要调用x264_mb_analyse_intra(), 检查当前宏块作为Intra宏块编码的代价是否小于作为P宏块编码的代价 (P Slice中也允许有Intra宏块)。

(3) 如果当前是B Slice, 则进行和P Slice类似的处理。

本文记录这一流程中Intra宏块的帧内预测模式分析函数x264_mb_analyse_intra()。

x264_mb_analyse_intra()

x264_mb_analyse_intra()用于对Intra宏块进行帧内预测模式的分析。该函数的定义位于encoder\analyse.c, 如下所示。

```
[cpp]
1. //帧内预测分析-从16x16的SAD,4个8x8的SAD和,16个4x4SAD中选出最优方式
2. static void x264_mb_analyse_intra( x264_t *h, x264_mb_analysis_t *a, int i_satd_inter )
3. {
4.     const unsigned int flags = h->sh.i_type == SLICE_TYPE_I ? h->param.analyse.intra : h->param.analyse.inter;
5.     //计算
6.     //p_fenc是编码帧
7.     pixel *p_src = h->mb.pic.p_fenc[0];
8.     //p_fdec是重建帧
9.     pixel *p_dst = h->mb.pic.p_fdec[0];
10.
11.     static const int8_t intra_analysis_shortcut[2][2][2][5] =
12.     {
13.         {{I_PRED_4x4_HU, -1, -1, -1, -1},
14.          {I_PRED_4x4_DDL, I_PRED_4x4_VL, -1, -1, -1}},
15.         {{I_PRED_4x4_DDR, I_PRED_4x4_HD, I_PRED_4x4_HU, -1, -1},
16.          {I_PRED_4x4_DDL, I_PRED_4x4_DDR, I_PRED_4x4_VR, I_PRED_4x4_VL, -1}},
17.         {{I_PRED_4x4_HU, -1, -1, -1, -1},
18.          {-1, -1, -1, -1, -1}},
19.         {{I_PRED_4x4_DDR, I_PRED_4x4_HD, I_PRED_4x4_HU, -1, -1},
20.          {I_PRED_4x4_DDR, I_PRED_4x4_VR, -1, -1, -1}}},
21.     };
22.
23.     int idx;
24.     int lambda = a->i_lambda;
25.
26.     /*----- Try all mode and calculate their score -----*/
27.     /* Disabled i16x16 for AVC-Intra compat */
28.     //帧内16x16
29.     if( !h->param.i_avcintra_class )
30.     {
31.         //获得可用的帧内预测模式-针对帧内16x16
32.         /*
33.          * 16x16块
34.          *
35.          * +-----+-----+
36.          * |               |
37.          * |               |
38.          * |               |
39.          * +       +       +
40.          * |               |
41.          * |               |
42.          * |               |
43.          * +-----+-----+
44.          *
45.          */
46.         //左侧是否有可用数据?上方是否有可用数据?
47.         const int8_t *predict_mode = predict_16x16_mode_available( h->mb.i_neighbour_intra );
48.
49.         /* Not heavily tuned */
50.         static const uint8_t i16x16_thresh_lut[11] = { 2, 2, 2, 3, 3, 4, 4, 4, 4, 4, 4 };
51.         int i16x16_thresh = a->b_fast_intra ? (i16x16_thresh_lut[h->mb.i_subpel_refine]*i_satd_inter)>>1 : COST_MAX;
52.
53.         if( !h->mb.b_lossless && predict_mode[3] >= 0 )
54.         {
55.             h->pixf.intra_mbcmp_x3_16x16( p_src, p_dst, a->i_satd_i16x16_dir );
56.             a->i_satd_i16x16_dir[0] += lambda * bs_size_ue(0);
57.             a->i_satd_i16x16_dir[1] += lambda * bs_size_ue(1);
58.             a->i_satd_i16x16_dir[2] += lambda * bs_size_ue(2);
59.             COPY2_IF_LT( a->i_satd_i16x16, a->i_satd_i16x16_dir[0], a->i_predict16x16, 0 );
60.             COPY2_IF_LT( a->i_satd_i16x16, a->i_satd_i16x16_dir[1], a->i_predict16x16, 1 );
61.             COPY2_IF_LT( a->i_satd_i16x16, a->i_satd_i16x16_dir[2], a->i_predict16x16, 2 );
62.
63.             /* Plane is expensive, so don't check it unless one of the previous modes was useful. */
64.             if( a->i_satd_i16x16 <= i16x16_thresh )
65.             {
66.                 h->predict_16x16[I_PRED_16x16_P]( p_dst );
67.                 a->i_satd_i16x16_dir[I_PRED_16x16_P] = h->pixf.mbcmp[PIXEL_16x16]( p_dst, FDEC_STRIDE, p_src, FENC_STRIDE );
68.                 a->i_satd_i16x16_dir[I_PRED_16x16_P] += lambda * bs_size_ue(3);
69.                 COPY2_IF_LT( a->i_satd_i16x16, a->i_satd_i16x16_dir[I_PRED_16x16_P], a->i_predict16x16, 3 );
70.             }
71.         }
72.         else
73.         {
74.             //遍历所有的可用的Intra16x16帧内预测模式
```

```

75.         //最多4种
76.         for( ; *predict_mode >= 0; predict_mode++ )
77.         {
78.             int i_satd;
79.             int i_mode = *predict_mode;
80.
81.             //帧内预测汇编函数：根据左边和上边的像素计算出预测值
82.             /*
83.              * 帧内预测举例
84.              * Vertical预测方式
85.              *   |X1 X2 ... X16
86.              *   +-+-----
87.              *   |X1 X2 ... X16
88.              *   |X1 X2 ... X16
89.              *   |.. .. ... X16
90.              *   |X1 X2 ... X16
91.              *
92.              * Horizontal预测方式
93.              *   |
94.              *   +-+-----
95.              *   X1| X1 X1 ... X1
96.              *   X2| X2 X2 ... X2
97.              *   ..| .. .. ... ..
98.              *   X16|X16 X16 ... X16
99.              *
100.             * DC预测方式
101.             *   |X1 X2 ... X16
102.             *   +-+-----
103.             *   X17|
104.             *   X18|   Y
105.             *   ..|
106.             *   X32|
107.             *
108.             * Y=(X1+X2+X3+X4+...+X31+X32)/32
109.             *
110.             */
111.             if( h->mb.b_lossless )
112.                 x264_predict_lossless_16x16( h, 0, i_mode );
113.             else
114.                 h->predict_16x16[i_mode]( p_dst );//计算结果存储在p_dst重建帧中
115.
116.             //计算SAD或者是SATD (SATD(transformed)是经过Hadamard变换之后的SAD)
117.             //即编码代价
118.             //数据位于p_dst和p_src
119.             i_satd = h->pixf.mbcmp[PIXEL_16x16]( p_dst, FDEC_STRIDE, p_src, FENC_STRIDE ) +
120.                 lambda * bs_size_ue( x264_mb_pred_model16x16_fix[i_mode] );
121.
122.             //COPY2_IF_LT()函数的意思是“copy if little”。即如果值更小（代价更小），就拷贝。
123.             //宏定义展开后如下所示
124.             //if((i_satd)<(a->i_satd_i16x16))
125.             //{
126.             //    (a->i_satd_i16x16)=(i_satd);
127.             //    (a->i_predict16x16)=(i_mode);
128.             //}
129.             COPY2_IF_LT( a->i_satd_i16x16, i_satd, a->i_predict16x16, i_mode );
130.             //每种模式的代价都会存储
131.             a->i_satd_i16x16_dir[i_mode] = i_satd;
132.         }
133.     }
134.
135.     if( h->sh.i_type == SLICE_TYPE_B )
136.         /* cavlc mb type prefix */
137.         a->i_satd_i16x16 += lambda * i_mb_b_cost_table[I_16x16];
138.
139.     if( a->i_satd_i16x16 > i16x16_thresh )
140.         return;
141. }
142.
143. uint16_t *cost_i4x4_mode = (uint16_t*)ALIGN((intptr_t)x264_cost_i4x4_mode,64) + a->i_qp*32 + 8;
144. /* 8x8 prediction selection */
145. //帧内8x8 (没研究过)
146. if( flags & X264_ANALYSE_I8x8 )
147. {
148.     ALIGNED_ARRAY_32( pixel, edge,[36] );
149.     x264_pixel_cmp_t sa8d = (h->pixf.mbcmp[0] == h->pixf.satd[0]) ? h->pixf.sa8d[PIXEL_8x8] : h->pixf.mbcmp[PIXEL_8x8];
150.     int i_satd_thresh = a->i_mbrd ? COST_MAX : X264_MIN( i_satd_inter, a->i_satd_i16x16 );
151.
152.     // FIXME some bias like in i4x4?
153.     int i_cost = lambda * 4; /* base predmode costs */
154.     h->mb.i_cbp_luma = 0;
155.
156.     if( h->sh.i_type == SLICE_TYPE_B )
157.         i_cost += lambda * i_mb_b_cost_table[I_8x8];
158.
159.     for( idx = 0;; idx++ )
160.     {
161.         int x = idx&1;
162.         int y = idx>>1;
163.         pixel *p_src_by = p_src + 8*x + 8*y*FENC_STRIDE;
164.         pixel *p_dst_by = p_dst + 8*x + 8*y*FDEC_STRIDE;
165.         int i_best = COST_MAX;

```

```

166.         int i_pred_mode = x264_mb_predict_intra4x4_mode( h, 4*idx );
167.
168.         const int8_t *predict_mode = predict_8x8_mode_available( a->b_avoid_topright, h->mb.i_neighbour8[idx], idx );
169.         h->predict_8x8_filter( p_dst_by, edge, h->mb.i_neighbour8[idx], ALL_NEIGHBORS );
170.
171.         if( h->pixf.intra_mbcmp_x9_8x8 && predict_mode[8] >= 0 )
172.         {
173.             /* No shortcuts here. The SSSE3 implementation of intra_mbcmp_x9 is fast enough. */
174.             i_best = h->pixf.intra_mbcmp_x9_8x8( p_src_by, p_dst_by, edge, cost_i4x4_mode-i_pred_mode, a-
->i_satd_i8x8_dir[idx] );
175.             i_cost += i_best & 0xffff;
176.             i_best >>= 16;
177.             a->i_predict8x8[idx] = i_best;
178.             if( idx == 3 || i_cost > i_satd_thresh )
179.                 break;
180.             x264_macroblock_cache_intra8x8_pred( h, 2*x, 2*y, i_best );
181.         }
182.         else
183.         {
184.             if( !h->mb.b_lossless && predict_mode[5] >= 0 )
185.             {
186.                 ALIGNED_ARRAY_16( int32_t, satd,[9] );
187.                 h->pixf.intra_mbcmp_x3_8x8( p_src_by, edge, satd );
188.                 int favor_vertical = satd[I_PRED_4x4_H] > satd[I_PRED_4x4_V];
189.                 satd[i_pred_mode] -= 3 * lambda;
190.                 for( int i = 2; i >= 0; i-- )
191.                 {
192.                     int cost = satd[i];
193.                     a->i_satd_i8x8_dir[idx][i] = cost + 4 * lambda;
194.                     COPY2_IF_LT( i_best, cost, a->i_predict8x8[idx], i );
195.                 }
196.
197.                 /* Take analysis shortcuts: don't analyse modes that are too
198.                  * far away direction-wise from the favored mode. */
199.                 if( a->i_mbrd < 1 + a->b_fast_intra )
200.                     predict_mode = intra_analysis_shortcut[a->b_avoid_topright][predict_mode[8] >= 0][favor_vertical];
201.                 else
202.                     predict_mode += 3;
203.             }
204.
205.             for( ; *predict_mode >= 0 && (i_best >= 0 || a->i_mbrd >= 2); predict_mode++ )
206.             {
207.                 int i_satd;
208.                 int i_mode = *predict_mode;
209.
210.                 if( h->mb.b_lossless )
211.                     x264_predict_lossless_8x8( h, p_dst_by, 0, idx, i_mode, edge );
212.                 else
213.                     h->predict_8x8[i_mode]( p_dst_by, edge );
214.
215.                 i_satd = sa8d( p_dst_by, FDEC_STRIDE, p_src_by, FENC_STRIDE );
216.                 if( i_pred_mode == x264_mb_pred_mode4x4_fix(i_mode) )
217.                     i_satd -= 3 * lambda;
218.
219.                 COPY2_IF_LT( i_best, i_satd, a->i_predict8x8[idx], i_mode );
220.                 a->i_satd_i8x8_dir[idx][i_mode] = i_satd + 4 * lambda;
221.             }
222.             i_cost += i_best + 3*lambda;
223.
224.             if( idx == 3 || i_cost > i_satd_thresh )
225.                 break;
226.             if( h->mb.b_lossless )
227.                 x264_predict_lossless_8x8( h, p_dst_by, 0, idx, a->i_predict8x8[idx], edge );
228.             else
229.                 h->predict_8x8[a->i_predict8x8[idx]]( p_dst_by, edge );
230.             x264_macroblock_cache_intra8x8_pred( h, 2*x, 2*y, a->i_predict8x8[idx] );
231.         }
232.         /* we need to encode this block now (for next ones) */
233.         x264_mb_encode_i8x8( h, 0, idx, a->i_qp, a->i_predict8x8[idx], edge, 0 );
234.     }
235.
236.     if( idx == 3 )
237.     {
238.         a->i_satd_i8x8 = i_cost;
239.         if( h->mb.i_skip_intra )
240.         {
241.             h->mc.copy[PIXEL_16x16]( h->mb.pic.i8x8_fdec_buf, 16, p_dst, FDEC_STRIDE, 16 );
242.             h->mb.pic.i8x8_nnz_buf[0] = M32( &h->mb.cache.non_zero_count[x264_scan8[ 0]] );
243.             h->mb.pic.i8x8_nnz_buf[1] = M32( &h->mb.cache.non_zero_count[x264_scan8[ 2]] );
244.             h->mb.pic.i8x8_nnz_buf[2] = M32( &h->mb.cache.non_zero_count[x264_scan8[ 8]] );
245.             h->mb.pic.i8x8_nnz_buf[3] = M32( &h->mb.cache.non_zero_count[x264_scan8[10]] );
246.             h->mb.pic.i8x8_cbp = h->mb.i_cbp_luma;
247.             if( h->mb.i_skip_intra == 2 )
248.                 h->mc.memcpy_aligned( h->mb.pic.i8x8_dct_buf, h->dct.luma8x8, sizeof(h->mb.pic.i8x8_dct_buf) );
249.         }
250.     }
251.     else
252.     {
253.         static const uint16_t cost_div_fix8[3] = {1024,512,341};
254.         a->i_satd_i8x8 = COST_MAX;
255.         i_cost = (i_cost * cost_div_fix8[idx]) >> 8;

```

```

256.     }
257.     /* Not heavily tuned */
258.     static const uint8_t i8x8_thresh[11] = { 4, 4, 4, 5, 5, 5, 6, 6, 6, 6, 6 };
259.     if( a->b_early_terminate && X264_MIN( i_cost, a->i_satd_i16x16 ) > ( i_satd_inter*i8x8_thresh[h->mb.i_subpel_refine])>>2 )
260.         return;
261. }
262.
263. /* 4x4 prediction selection */
264. //帧内4x4
265. if( flags & X264_ANALYSE_I4x4 )
266. {
267.     /*
268.      * 16x16 宏块被划分为16个4x4子块
269.      *
270.      * +---+---+---+---+
271.      * |   |   |   |   |
272.      * +---+---+---+---+
273.      * |   |   |   |   |
274.      * +---+---+---+---+
275.      * |   |   |   |   |
276.      * +---+---+---+---+
277.      * |   |   |   |   |
278.      * +---+---+---+---+
279.      */
280.     /*
281.      int i_cost = lambda * (24+16); /* 24 from JVT (SATD0), 16 from base predmode costs */
282.      int i_satd_thresh = a->b_early_terminate ? X264_MIN3( i_satd_inter, a->i_satd_i16x16, a->i_satd_i8x8 ) : COST_MAX;
283.      h->mb.i_cbp_luma = 0;
284.
285.      if( a->b_early_terminate && a->i_mbrd )
286.          i_satd_thresh = i_satd_thresh * (10-a->b_fast_intra)/8;
287.
288.      if( h->sh.i_type == SLICE_TYPE_B )
289.          i_cost += lambda * i_mb_b_cost_table[I_4x4];
290.      //循环所有的4x4块
291.      for( idx = 0;; idx++ )
292.      {
293.          //编码帧中的像素
294.          //block_idx_xy_fenc[]记录了4x4小块在p_fenc中的偏移地址
295.          pixel *p_src_by = p_src + block_idx_xy_fenc[idx];
296.          //重建帧中的像素
297.          //block_idx_xy_fdec[]记录了4x4小块在p_fdec中的偏移地址
298.          pixel *p_dst_by = p_dst + block_idx_xy_fdec[idx];
299.
300.          int i_best = COST_MAX;
301.          int i_pred_mode = x264_mb_predict_intra4x4_mode( h, idx );
302.          //获得可用的帧内预测模式-针对帧内4x4
303.          //左侧是否有可用数据?上方是否有可用数据?
304.          const int8_t *predict_mode = predict_4x4_mode_available( a->b_avoid_topright, h->mb.i_neighbour4[idx], idx );
305.
306.          if( (h->mb.i_neighbour4[idx] & (MB_TOPRIGHT|MB_TOP)) == MB_TOP )
307.              /* emulate missing topright samples */
308.              MPIXEL_X4( &p_dst_by[4 - FDEC_STRIDE] ) = PIXEL_SPLAT_X4( p_dst_by[3 - FDEC_STRIDE] );
309.
310.          if( h->pixf.intra_mbcmp_x9_4x4 && predict_mode[8] >= 0 )
311.          {
312.              /* No shortcuts here. The SSSE3 implementation of intra_mbcmp_x9 is fast enough. */
313.              i_best = h->pixf.intra_mbcmp_x9_4x4( p_src_by, p_dst_by, cost_i4x4_mode-i_pred_mode );
314.              i_cost += i_best & 0xffff;
315.              i_best >= 16;
316.              a->i_predict4x4[idx] = i_best;
317.              if( i_cost > i_satd_thresh || idx == 15 )
318.                  break;
319.              h->mb.cache.intra4x4_pred_mode[x264_scan8[idx]] = i_best;
320.          }
321.          else
322.          {
323.              if( !h->mb.b_lossless && predict_mode[5] >= 0 )
324.              {
325.                  ALIGNED_ARRAY_16( int32_t, satd,[9] );
326.
327.                  h->pixf.intra_mbcmp_x3_4x4( p_src_by, p_dst_by, satd );
328.                  int favor_vertical = satd[I_PRED_4x4_H] > satd[I_PRED_4x4_V];
329.                  satd[i_pred_mode] -= 3 * lambda;
330.                  i_best = satd[I_PRED_4x4_DC]; a->i_predict4x4[idx] = I_PRED_4x4_DC;
331.                  COPY2_IF_LT( i_best, satd[I_PRED_4x4_H], a->i_predict4x4[idx], I_PRED_4x4_H );
332.                  COPY2_IF_LT( i_best, satd[I_PRED_4x4_V], a->i_predict4x4[idx], I_PRED_4x4_V );
333.
334.                  /* Take analysis shortcuts: don't analyse modes that are too
335.                   * far away direction-wise from the favored mode. */
336.                  if( a->i_mbrd < 1 + a->b_fast_intra )
337.                      predict_mode = intra_analysis_shortcut[a->b_avoid_topright][predict_mode[8] >= 0][favor_vertical];
338.                  else
339.                      predict_mode += 3;
340.              }
341.
342.              if( i_best > 0 )
343.              {
344.                  //遍历所有Intra4x4帧内模式,最多9种
345.                  for( ; *predict_mode >= 0; predict_mode++ )
346.                  {

```



```

347.         int i_satd;
348.         int i_mode = *predict_mode;
349.         /*
350.          * 4x4帧内预测举例
351.          *
352.          * Vertical预测方式
353.          * |X1 X2 X3 X4
354.          * ---+-----
355.          * |X1 X2 X3 X4
356.          * |X1 X2 X3 X4
357.          * |X1 X2 X3 X4
358.          * |X1 X2 X3 X4
359.          *
360.          * Horizontal预测方式
361.          * |
362.          * ---+-----
363.          * X5|X5 X5 X5 X5
364.          * X6|X6 X6 X6 X6
365.          * X7|X7 X7 X7 X7
366.          * X8|X8 X8 X8 X8
367.          *
368.          * DC预测方式
369.          * |X1 X2 X3 X4
370.          * ---+-----
371.          * X5|
372.          * X6|      Y
373.          * X7|
374.          * X8|
375.          *
376.          * Y=(X1+X2+X3+X4+X5+X6+X7+X8)/8
377.          *
378.          */
379.         if( h->mb.b_lossless )
380.             x264_predict_lossless_4x4( h, p_dst_by, 0, idx, i_mode );
381.         else
382.             h->predict_4x4[i_mode]( p_dst_by );//帧内预测汇编函数-存储在重建帧中
383.
384.         //计算SAD或者是SATD (SATD (Transformed) 是经过Hadamard变换之后的SAD)
385.         //即编码代价
386.         //p_src_by编码帧, p_dst_by重建帧
387.         i_satd = h->pixf.mbcmp[PIXEL_4x4]( p_dst_by, FDEC_STRIDE, p_src_by, FENC_STRIDE );
388.         if( i_pred_mode == x264_mb_pred_mode4x4_fix(i_mode) )
389.         {
390.             i_satd -= lambda * 3;
391.             if( i_satd <= 0 )
392.             {
393.                 i_best = i_satd;
394.                 a->i_predict4x4[idx] = i_mode;
395.                 break;
396.             }
397.         }
398.         //COPY2_IF_LT()函数的意思是“copy if little”。即如果值更小 (代价更小) , 就拷贝。
399.         //宏定义展开后如下所示
400.         //if((i_satd)<(i_best))
401.         //{
402.         //    (i_best)=(i_satd);
403.         //    (a->i_predict4x4[idx])=(i_mode);
404.         //}
405.
406.         //看看代价是否更小
407.         //i_best中存储了最小的代价值
408.         //i_predict4x4[idx]中存储了代价最小的预测模式 (idx为4x4小块的序号)
409.         COPY2_IF_LT( i_best, i_satd, a->i_predict4x4[idx], i_mode );
410.     }
411. }
412. //累加各个4x4块的代价 (累加每个块的最小代价)
413. i_cost += i_best + 3 * lambda;
414. if( i_cost > i_satd_thresh || idx == 15 )
415.     break;
416. if( h->mb.b_lossless )
417.     x264_predict_lossless_4x4( h, p_dst_by, 0, idx, a->i_predict4x4[idx] );
418. else
419.     h->predict_4x4[a->i_predict4x4[idx]]( p_dst_by );
420.
421. /*
422.  * 将mode填充至intra4x4_pred_mode_cache
423.  *
424.  * 用简单图形表示intra4x4_pred_mode_cache如下。数字代表填充顺序 (一共填充16次)
425.  * |
426.  * ---+-----
427.  * | 0 0 0 0 0 0 0 0
428.  * | 0 0 0 0 1 2 5 6
429.  * | 0 0 0 0 3 4 7 8
430.  * | 0 0 0 0 9 10 13 14
431.  * | 0 0 0 0 11 12 15 16
432.  *
433.  */
434.     h->mb.cache.intra4x4_pred_mode[x264_scan8[idx]] = a->i_predict4x4[idx];
435. }
436. /* we need to encode this block now (for next ones) */
437. x264_mb_encode_i4x4( h, 0, idx, a->i_qp, a->i_predict4x4[idx], 0 );

```

```
439.         if( idx == 15 )//处理最后一个4x4小块 （一共16个块）
440.         {
441.             //开销（累加完的）
442.             a->i_satd_i4x4 = i_cost;
443.             if( h->mb.i_skip_intra )
444.             {
445.                 h->mc.copy[PIXEL_16x16]( h->mb.pic.i4x4_fdec_buf, 16, p_dst, FDEC_STRIDE, 16 );
446.                 h->mb.pic.i4x4_nnz_buf[0] = M32( &h->mb.cache.non_zero_count[x264_scan8[ 0]] );
447.                 h->mb.pic.i4x4_nnz_buf[1] = M32( &h->mb.cache.non_zero_count[x264_scan8[ 2]] );
448.                 h->mb.pic.i4x4_nnz_buf[2] = M32( &h->mb.cache.non_zero_count[x264_scan8[ 8]] );
449.                 h->mb.pic.i4x4_nnz_buf[3] = M32( &h->mb.cache.non_zero_count[x264_scan8[10]] );
450.                 h->mb.pic.i4x4_cbp = h->mb.i_cbp_luma;
451.                 if( h->mb.i_skip_intra == 2 )
452.                     h->mc.memcpy_aligned( h->mb.pic.i4x4_dct_buf, h->dct.luma4x4, sizeof(h->mb.pic.i4x4_dct_buf) );
453.             }
454.         }
455.         else
456.             a->i_satd_i4x4 = COST_MAX;
457.     }
458. }
```

总体说来x264_mb_analyse_intra()通过计算Intra16x16，Intra8x8（暂时没有研究），Intra4x4这3中帧内预测模式的代价，比较后得到最佳的帧内预测模式。该函数的等流程大致如下：

- (1) 进行Intra16X16模式的预测
 - a)调用predict_16x16_mode_available()根据周围宏块的情况判断其可用的预测模式（主要检查左边和上边的块是否可用）。
 - b)循环计算4种Intra16x16帧内预测模式：
 - i.调用predict_16x16[]()汇编函数进行Intra16x16帧内预测
 - ii.调用x264_pixel_function_t中的mbcmp[]()计算编码代价（mbcmp[]()指向SAD或者SATD汇编函数）。
 - c)获取最小代价的Intra16x16模式。
- (2) 进行Intra8x8模式的预测（未研究，流程应该类似）
- (3) 进行Intra4X4块模式的预测
 - a)循环处理16个4x4的块：
 - i.调用x264_mb_predict_intra4x4_mode()根据周围宏块情况判断该块可用的预测模式。
 - ii.循环计算9种Intra4x4的帧内预测模式：
 - 1)调用predict_4x4 []()汇编函数进行Intra4x4帧内预测
 - 2)调用x264_pixel_function_t中的mbcmp[]()计算编码代价（mbcmp[]()指向SAD或者SATD汇编函数）。
 - iii.获取最小代价的Intra4x4模式。
 - b)将16个4X4块的最小代价相加，得到总代价。
- (4) 将上述3中模式的代价进行对比，取最小者为当前宏块的帧内预测模式。

后文将会对其中涉及到的几种汇编函数进行分析。在看源代码之前，简单记录一下相关的知识。

帧内预测知识

简单记录一下帧内预测的方法。帧内预测根据宏块左边和上边的边界像素值推算宏块内部的像素值，帧内预测的效果如下图所示。其中左边的图为图像原始画面，右边的图为经过帧内预测后没有叠加残差的画面。



H.264中有两种帧内预测模式：16x16亮度帧内预测模式和4x4亮度帧内预测模式。其中16x16帧内预测模式一共有4种，如下图所示。



这4种模式列表如下。

模式	描述
Vertical	由上边像素推出相应像素值
Horizontal	由左边像素推出相应像素值
DC	由上边和左边像素平均值推出相应像素值
Plane	由上边和左边像素推出相应像素值

4x4帧内预测模式一共有9种，如下图所示。



可以看出，Intra4x4帧内预测模式中前4种和Intra16x16是一样的。后面多增加了几种预测箭头不是45度角的方式——前面的箭头位于“口”中，而后面的箭头位于“日”中。

像素比较知识

帧内预测代价计算的过程中涉及到SAD和SATD像素计算，简单记录几个相关的概念。有关SAD、SATD、SSD的定义如下：

SAD（Sum of Absolute Difference）也可以称为SAE（Sum of Absolute Error），即绝对误差和。它的计算方法就是求出两个像素块对应像素点的差值，将这些差值分别求绝对值之后再进行累加。

SATD（Sum of Absolute Transformed Difference）即Hadamard变换后再绝对值求和。它和SAD的区别在于多了一个“变换”。

SSD（Sum of Squared Difference）也可以称为SSE（Sum of Squared Error），即差值的平方和。它和SAD的区别在于多了一个“平方”。

H.264中使用SAD和SATD进行宏块预测模式的判断。早期的编码器使用SAD进行计算，近期的编码器多使用SATD进行计算。为什么使用SATD而不使用SAD呢？关键原因在于编码之后码流的大小是和图像块DCT变换后频域信息紧密相关的，而和变换前的时域信息关联性小一些。SAD只能反应时域信息；SATD却可以反映频域信息，而且计算复杂度也低于DCT变换，因此是比较合适的模式选择的依据。

使用SAD进行模式选择的示例如下所示。下面这张图代表了一个普通的Intra16x16的宏块的像素。它的下方包含了使用Vertical，Horizontal，DC和Plane四种帧内预测模式预测的像素。通过计算可以得到这几种预测像素和原始像素之间的SAD（SAE）分别为3985，5097，4991，2539。由于Plane模式的SAD取值最小，由此可以断定Plane模式对于这个宏块来说是最好的帧内预测模式。



下面按照Intra16x16预测，Intra4x4预测，像素计算的顺序记录依次记录各个模块的汇编函数源代码。

Intra16x16帧内预测源代码

Intra16x16帧内预测模块的初始化函数是x264_predict_16x16_init()。该函数对x264_predict_t结构体中的函数指针进行了赋值。X264运行的过程中只要调用x264_predict_t的函数指针就可以完成相应的功能。

x264_predict_16x16_init()



x264_predict_16x16_init()用于初始化Intra16x16帧内预测汇编函数。该函数的定义位于x264\common\predict.c，如下所示。

```
[cpp]
1. //Intra16x16帧内预测汇编函数初始化
2. void x264_predict_16x16_init( int cpu, x264_predict_t pf[7] )
3. {
4.     //C语言版本
5.     //=====
6.     //垂直 Vertical
7.     pf[I_PRED_16x16_V ]    = x264_predict_16x16_v_c;
8.     //水平 Horizontal
9.     pf[I_PRED_16x16_H ]    = x264_predict_16x16_h_c;
10.    //DC
11.    pf[I_PRED_16x16_DC]    = x264_predict_16x16_dc_c;
12.    //Plane
13.    pf[I_PRED_16x16_P ]    = x264_predict_16x16_p_c;
14.    //这几种是啥？
15.    pf[I_PRED_16x16_DC_LEFT]= x264_predict_16x16_dc_left_c;
16.    pf[I_PRED_16x16_DC_TOP ]= x264_predict_16x16_dc_top_c;
17.    pf[I_PRED_16x16_DC_128 ]= x264_predict_16x16_dc_128_c;
18.    //=====
19.    //MMX版本
20.    #if HAVE_MMX
21.        x264_predict_16x16_init_mmx( cpu, pf );
22.    #endif
23.    //ALTIVEC版本
24.    #if HAVE_ALTIVEC
25.        if( cpu & X264_CPU_ALTIVEC )
26.            x264_predict_16x16_init_altivec( pf );
27.    #endif
28.    //ARMV6版本
29.    #if HAVE_ARMV6
30.        x264_predict_16x16_init_arm( cpu, pf );
31.    #endif
32.    //AARCH64版本
33.    #if ARCH_AARCH64
34.        x264_predict_16x16_init_aarch64( cpu, pf );
35.    #endif
36. }
```

从源代码可看出，x264_predict_16x16_init()首先对帧内预测函数指针数组x264_predict_t[]中的元素赋值了C语言版本的函数x264_predict_16x16_v_c()，x264_predict_16x16_h_c()，x264_predict_16x16_dc_c()，x264_predict_16x16_p_c()；然后会判断系统平台的特性，如果平台支持的话，会调用x264_predict_16x16_init_mmx()，x264_predict_16x16_init_arm()等给x264_predict_t[]中的元素赋值经过汇编优化的函数。下文首先看一下Intra16x16中的4种帧内预测模式的C语言版本，作为对比再看一下Intra16x16中Vertical模式的X86汇编版本和NEON汇编版本。

x264_predict_16x16_v_c()

x264_predict_16x16_v_c()是Intra16x16帧内预测Vertical模式的C语言版本函数。该函数的定义位于common\predict.c，如下所示。

```
[cpp]  
1. //16x16帧内预测
2. //垂直预测 (Vertical)
3. void x264_predict_16x16_v_c( pixel *src )
4. {
5.     /*
6.      * Vertical预测方式
7.      * |X1 X2 X3 X4
8.      * --+-----
9.      * |X1 X2 X3 X4
10.     * |X1 X2 X3 X4
11.     * |X1 X2 X3 X4
12.     * |X1 X2 X3 X4
13.     *
14.     */
15.     /*
16.      * 【展开宏定义】
17.      * uint32_t v0 = ((x264_union32_t*)(&src[ 0-FDEC_STRIDE]))->i;
18.      * uint32_t v1 = ((x264_union32_t*)(&src[ 4-FDEC_STRIDE]))->i;
19.      * uint32_t v2 = ((x264_union32_t*)(&src[ 8-FDEC_STRIDE]))->i;
20.      * uint32_t v3 = ((x264_union32_t*)(&src[12-FDEC_STRIDE]))->i;
21.      * 在这里，上述代码实际上相当于：
22.      * uint32_t v0 = *((uint32_t*)(&src[ 0-FDEC_STRIDE]));
23.      * uint32_t v1 = *((uint32_t*)(&src[ 4-FDEC_STRIDE]));
24.      * uint32_t v2 = *((uint32_t*)(&src[ 8-FDEC_STRIDE]));
25.      * uint32_t v3 = *((uint32_t*)(&src[12-FDEC_STRIDE]));
26.      * 即分成4次，每次取出4个像素（一共16个像素），分别赋值给v0，v1，v2，v3
27.      * 取出的值源自于16x16块上面的一行像素
28.      *   0|         4         8         12        16
29.      *   ||      v0      |   v1   |   v2   |   v3   |
30.      *   ---+-----+-----+-----+-----+
31.      *   ||
32.      *   ||
33.      *   ||
34.      *   ||
35.      *   ||
36.      *   ||
37.      *
38.      */
39.     //pixel4实际上是uint32_t（占用32bit），存储4个像素的值（每个像素占用8bit）
40.
41.     pixel4 v0 = MPIXEL_X4( &src[ 0-FDEC_STRIDE] );
42.     pixel4 v1 = MPIXEL_X4( &src[ 4-FDEC_STRIDE] );
43.     pixel4 v2 = MPIXEL_X4( &src[ 8-FDEC_STRIDE] );
44.     pixel4 v3 = MPIXEL_X4( &src[12-FDEC_STRIDE] );
45.
46.     //循环赋值16行
47.     for( int i = 0; i < 16; i++ )
48.     {
49.         // 【展开宏定义】
50.         //(((x264_union32_t*)(src+ 0))>i) = v0;
51.         //(((x264_union32_t*)(src+ 4))>i) = v1;
52.         //(((x264_union32_t*)(src+ 8))>i) = v2;
53.         //(((x264_union32_t*)(src+12))>i) = v3;
54.         //即分成4次，每次赋值4个像素
55.         //
56.         MPIXEL_X4( src+ 0 ) = v0;
57.         MPIXEL_X4( src+ 4 ) = v1;
58.         MPIXEL_X4( src+ 8 ) = v2;
59.         MPIXEL_X4( src+12 ) = v3;
60.         //下一行
61.         //FDEC_STRIDE=32,是重建宏块缓存fdec_buf一行的数据量
62.         src += FDEC_STRIDE;
63.     }
64. }
```

从源代码可以看出，x264_predict_16x16_v_c()首先取出16x16块上面一行像素值，依次存储在v0、v1、v2、v3，然后循环16次赋值给块中的16行像素。

x264_predict_16x16_h_c()

x264_predict_16x16_h_c()是Intra16x16帧内预测Horizontal模式的C语言版本函数。该函数的定义位于common\predict.c，如下所示。

```

1. //16x16帧内预测
2. //水平预测 (Horizontal)
3. void x264_predict_16x16_h_c( pixel *src )
4. {
5.     /*
6.      * Horizontal预测方式
7.      * |
8.      * --+-----
9.      * X5|X5 X5 X5 X5
10.     * X6|X6 X6 X6 X6
11.     * X7|X7 X7 X7 X7
12.     * X8|X8 X8 X8 X8
13.     *
14.     */
15.     /*
16.     * const pixel4 v = PIXEL_SPLAT_X4( src[-1] );
17.     * 宏定义展开后
18.     * const uint32_t v = (src[-1])*0x01010101U;
19.     *
20.     * PIXEL_SPLAT_X4()的作用应该是把最后一个像素（最后8位）拷贝给前面3个像素（前24位）
21.     * 即把0x0100009F变成0x9F9F9F9F
22.     * 推导：
23.     * 前提是x占8bit（对应1个像素）
24.     * y=x*0x01010101
25.     * =x*(0x00000001+0x00000100+0x00010000+0x01000000)
26.     * =x<<0+x<<8+x<<16+x<<24
27.     *
28.     * const uint32_t v = (src[-1])*0x01010101U含义：
29.     * 每行把src[-1]中像素值例如0x02赋值给v.v取值为0x02020202
30.     * src[-1]即16x16块左侧的值
31.     */
32.     //循环赋值16行
33.     for( int i = 0; i < 16; i++ )
34.     {
35.         const pixel4 v = PIXEL_SPLAT_X4( src[-1] );
36.         //宏定义展开后：
37.         //((x264_union32_t*)(src+ 0))->i=v;
38.         //((x264_union32_t*)(src+ 4))->i=v;
39.         //((x264_union32_t*)(src+ 8))->i=v;
40.         //((x264_union32_t*)(src+12))->i=v;
41.         //即分4次，每次赋值4个像素（一行一共16个像素，取值是一样的）
42.         //
43.         //  0|          4          8          12          16
44.         //  ||          |          |          |          |
45.         //  --+-----+-----+-----+-----+
46.         //  ||
47.         //  v ||      v   |      v   |      v   |      v   |
48.         //  ||
49.         //  ||
50.         //  ||
51.         //
52.         MPIXEL_X4( src+ 0 ) = v;
53.         MPIXEL_X4( src+ 4 ) = v;
54.         MPIXEL_X4( src+ 8 ) = v;
55.         MPIXEL_X4( src+12 ) = v;
56.         //下一行
57.         //FDEC_STRIDE=32,是重建宏块缓存fdec_buf一行的数据量
58.         src += FDEC_STRIDE;
59.     }
60. }

```

从源代码可以看出，x264_predict_16x16_h_c()首先取出16x16块每行左边的1个像素，复制4份后存储在v中，然后分成4次将v赋值给这一行像素。其中“PIXEL_SPLAT_X4()”的功能是取出变量低8位的数值复制4份到高24位，相关的推导功能已经记录在源代码中，不再重复叙述。

x264_predict_16x16_dc_c()

x264_predict_16x16_dc_c()是Intra16x16帧内预测DC模式的C语言版本函数。该函数的定义位于common\predict.c，如下所示。

```

1.  #define PREDICT_16x16_DC(v)\
2.      for( int i = 0; i < 16; i++ )\
3.      {\
4.          MPIXEL_X4( src+ 0 ) = v;\
5.          MPIXEL_X4( src+ 4 ) = v;\
6.          MPIXEL_X4( src+ 8 ) = v;\
7.          MPIXEL_X4( src+12 ) = v;\
8.          src += FDEC_STRIDE;\
9.      }
10.
11. void x264_predict_16x16_dc_c( pixel *src )
12. {
13.     /*
14.      * DC预测方式
15.      * |X1 X2 X3 X4
16.      * --+-----
17.      * X5|
18.      * X6|      Y
19.      * X7|
20.      * X8|
21.      *
22.      * Y=(X1+X2+X3+X4+X5+X6+X7+X8)/8
23.      */
24.
25.     int dc = 0;
26.     //把16x16块中所有像素的值加起来，存储在dc中
27.     for( int i = 0; i < 16; i++ )
28.     {
29.         //左侧的值
30.         dc += src[-1 + i * FDEC_STRIDE];
31.         //上方的值
32.         dc += src[i - FDEC_STRIDE];
33.     }
34.     //加起来的值除以32（一共16+16个点）
35.     //“+16”是为了四舍五入？
36.     //PIXEL_SPLAT_X4()的作用应该是把最后一个像素（最后8位）拷贝给前面3个像素（前24位）
37.     //即把0x0100009F变成0x9F9F9F9F
38.     pixel4 dcsplat = PIXEL_SPLAT_X4( ( dc + 16 ) >> 5 );
39.     //赋值到16x16块中的每个像素
40.     /*
41.      * 宏展开之后结果
42.      * for( int i = 0; i < 16; i++ )
43.      * {
44.      *     (((x264_union32_t*)(src+ 0))->i) = dcsplat;
45.      *     (((x264_union32_t*)(src+ 4))->i) = dcsplat;
46.      *     (((x264_union32_t*)(src+ 8))->i) = dcsplat;
47.      *     (((x264_union32_t*)(src+12))->i) = dcsplat;
48.      *     src += 32;
49.      * }
50.      */
51.     PREDICT_16x16_DC( dcsplat );
52. }

```

从源代码可以看出，x264_predict_16x16_dc_c()求出16x16块上面一行像素和左边一列像素的平均值，然后赋值给16x16块中的每一个像素。

X86以及ARM平台汇编函数

除了C语言版本的帧内预测函数之外，还包含了很多汇编语言版本的函数。下面以Intra16x16帧内预测Vertical模式为例，看一下该函数的X86平台汇编版本以及ARM平台汇编版本。

x264_predict_16x16_init_mmx()

x264_predict_16x16_init_mmx()用于初始化经过x86汇编优化过的Intra16x16的帧内预测函数。该函数的定义位于common\x86\predict-c.c（在“x86”子文件夹下），如下所示。

```

1. //Intra16x16帧内预测汇编函数-MMX版本
2. void x264_predict_16x16_init_mmx( int cpu, x264_predict_t pf[7] )
3. {
4.     if( !(cpu&X264_CPU_MMX2) )
5.         return;
6.     pf[I_PRED_16x16_DC]      = x264_predict_16x16_dc_mmx2;
7.     pf[I_PRED_16x16_DC_TOP]  = x264_predict_16x16_dc_top_mmx2;
8.     pf[I_PRED_16x16_DC_LEFT] = x264_predict_16x16_dc_left_mmx2;
9.     pf[I_PRED_16x16_V]       = x264_predict_16x16_v_mmx2;
10.    pf[I_PRED_16x16_H]        = x264_predict_16x16_h_mmx2;
11. #if HIGH_BIT_DEPTH
12.    if( !(cpu&X264_CPU_SSE) )
13.        return;
14.    pf[I_PRED_16x16_V]        = x264_predict_16x16_v_sse;
15.    if( !(cpu&X264_CPU_SSE2) )
16.        return;
17.    pf[I_PRED_16x16_DC]      = x264_predict_16x16_dc_sse2;
18.    pf[I_PRED_16x16_DC_TOP]  = x264_predict_16x16_dc_top_sse2;
19.    pf[I_PRED_16x16_DC_LEFT] = x264_predict_16x16_dc_left_sse2;
20.    pf[I_PRED_16x16_H]       = x264_predict_16x16_h_sse2;
21.    pf[I_PRED_16x16_P]       = x264_predict_16x16_p_sse2;
22.    if( !(cpu&X264_CPU_AVX) )
23.        return;
24.    pf[I_PRED_16x16_V]       = x264_predict_16x16_v_avx;
25.    if( !(cpu&X264_CPU_AVX2) )
26.        return;
27.    pf[I_PRED_16x16_H]       = x264_predict_16x16_h_avx2;
28. #else
29. #if !ARCH_X86_64
30.     pf[I_PRED_16x16_P]      = x264_predict_16x16_p_mmx2;
31. #endif
32.     if( !(cpu&X264_CPU_SSE) )
33.         return;
34.     pf[I_PRED_16x16_V]      = x264_predict_16x16_v_sse;
35.     if( !(cpu&X264_CPU_SSE2) )
36.         return;
37.     pf[I_PRED_16x16_DC]     = x264_predict_16x16_dc_sse2;
38.     if( cpu&X264_CPU_SSE2_IS_SLOW )
39.         return;
40.     pf[I_PRED_16x16_DC_TOP] = x264_predict_16x16_dc_top_sse2;
41.     pf[I_PRED_16x16_DC_LEFT] = x264_predict_16x16_dc_left_sse2;
42.     pf[I_PRED_16x16_P]      = x264_predict_16x16_p_sse2;
43.     if( !(cpu&X264_CPU_SSSE3) )
44.         return;
45.     if( !(cpu&X264_CPU_SLOW_PSHUFB) )
46.         pf[I_PRED_16x16_H]   = x264_predict_16x16_h_ssse3;
47. #if HAVE_X86_INLINE_ASM
48.     pf[I_PRED_16x16_P]       = x264_predict_16x16_p_ssse3;
49. #endif
50.     if( !(cpu&X264_CPU_AVX) )
51.         return;
52.     pf[I_PRED_16x16_P]       = x264_predict_16x16_p_avx;
53. #endif // HIGH_BIT_DEPTH
54.
55.     if( cpu&X264_CPU_AVX2 )
56.     {
57.         pf[I_PRED_16x16_P]     = x264_predict_16x16_p_avx2;
58.         pf[I_PRED_16x16_DC]    = x264_predict_16x16_dc_avx2;
59.         pf[I_PRED_16x16_DC_TOP] = x264_predict_16x16_dc_top_avx2;
60.         pf[I_PRED_16x16_DC_LEFT] = x264_predict_16x16_dc_left_avx2;
61.     }
62. }

```

可以看出，针对Intra16x16的Vertical帧内预测模式，x264_predict_16x16_init_mmx()会根据系统的特型初始化2个函数：如果系统仅支持MMX指令集，就会初始化x264_predict_16x16_v_mmx2()；如果系统还支持SSE指令集，就会初始化x264_predict_16x16_v_sse()。下面看一下这2个函数的代码。

x264_predict_16x16_v_mmx2()

x264_predict_16x16_v_sse()

在x264中，x264_predict_16x16_v_mmx2()和x264_predict_16x16_v_sse()这两个函数的定义是写到一起的。它们的定义位于common\x86\predict-a.asm，如下所示。

```

1. ;-----
2. ; void predict_16x16_v( pixel *src )
3. ; Intra16x16帧内预测Vertical模式
4. ;-----
5. ;SIZEOF_PIXEL取值为1
6. ;FDEC_STRIDE为重建宏块缓存fdec_buf一行像素的大小，取值为32
7. ;
8. ;平台相关的信息位于x86inc.asm
9. ;INIT_MMX中
10. ; mmsize为8
11. ; mova为movq
12. ;INIT_XMM中：
13. ; mmsize为16
14. ; mova为movdqa
15. ;
16. ;STORE16的定义在前面，用于循环16行存储数据
17.
18. %macro PREDICT_16x16_V 0
19. cglobal predict_16x16_v, 1,2
20. %assign %i 0
21. %rep 16*SIZEOF_PIXEL/mmsize ;rep循环执行，拷贝16x16块上方的1行像素数据至m0,m1...
22. ;mmsize为指令1次处理比特数
23.     mova m %+, %i, [r0-FDEC_STRIDE+%i*mmsize] ;移入m0,m1...
24. %assign %i %i+1
25. %endrep
26. %if 16*SIZEOF_PIXEL/mmsize == 4 ;1行需要处理4次
27.     STORE16 m0, m1, m2, m3 ;循环存储16行，每次存储4个寄存器
28. %elif 16*SIZEOF_PIXEL/mmsize == 2 ;1行需要处理2次
29.     STORE16 m0, m1 ;循环存储16行，每次存储2个寄存器
30. %else ;1行需要处理1次
31.     STORE16 m0 ;循环存储16行，每次存储1个寄存器
32. %endif
33.     RET
34. %endmacro
35.
36. INIT_MMX mmx2
37. PREDICT_16x16_V
38. INIT_XMM sse
39. PREDICT_16x16_V

```

从汇编代码可以看出，x264_predict_16x16_v_mmx2()和x264_predict_16x16_v_sse()的逻辑是一模一样的。它们之间的不同主要在于一条指令处理的数据量：MMX指令的MOVA对应的是MOVQ，一次处理8Byte（8个像素）；SSE指令的MOVA对应的是MOVDQA，一次处理16Byte（16个像素，正好是16x16块中的一行像素）。作为对比，我们可以看一下ARM平台下汇编优化过的Intra16x16的帧内预测函数。这些汇编函数的初始化函数是x264_predict_16x16_init_arm()。

x264_predict_16x16_init_arm()

x264_predict_16x16_init_arm()用于初始化ARM平台下汇编优化过的Intra16x16的帧内预测函数。该函数的定义位于common\arm\predict-c.c（“arm”文件夹下），如下所示。

```

1. void x264_predict_16x16_init_arm( int cpu, x264_predict_t pf[7] )
2. {
3.     if (!(cpu & X264_CPU_NEON))
4.         return;
5.
6. #if !HIGH_BIT_DEPTH
7.     pf[I_PRED_16x16_DC ] = x264_predict_16x16_dc_neon;
8.     pf[I_PRED_16x16_DC_TOP] = x264_predict_16x16_dc_top_neon;
9.     pf[I_PRED_16x16_DC_LEFT]= x264_predict_16x16_dc_left_neon;
10.    pf[I_PRED_16x16_H ] = x264_predict_16x16_h_neon;
11.    pf[I_PRED_16x16_V ] = x264_predict_16x16_v_neon;
12.    pf[I_PRED_16x16_P ] = x264_predict_16x16_p_neon;
13. #endif // !HIGH_BIT_DEPTH
14. }

```

从源代码可以看出，针对Vertical预测模式，x264_predict_16x16_init_arm()初始化了经过NEON指令集优化的函数x264_predict_16x16_v_neon()。

x264_predict_16x16_v_neon()

x264_predict_16x16_v_neon()的定义位于common\arm\predict-a.S，如下所示。


```

1.  /*
2.   * Intra16x16帧内预测Vertical模式-NEON
3.   *
4.   */
5.   /* FDEC_STRIDE=32Bytes, 为重建宏块一行像素的大小 */
6.   /* R0存储16x16像素块地址 */
7.   function x264_predict_16x16_v_neon
8.   sub     r0, r0, #FDEC_STRIDE /* r0=r0-FDEC_STRIDE */
9.   mov     ip, #FDEC_STRIDE    /* ip=32 */
10.                                     /* VLD向量加载: 内存->NEON寄存器 */
11.                                     /* d0,d1为64bit双字寄存器, 共16Byte, 在这里存储16x16块上方一行像素 */
12.   vld1.64 {d0-d1}, [r0,:128], ip /* 将R0指向的数据从内存加载到d0和d1寄存器 (64bit) */
13.                                     /* r0=r0+ip */
14.   .rept 16                          /* 循环16次, 一次处理1行 */
15.                                     /* VST向量存储: NEON寄存器->内存 */
16.   vst1.64 {d0-d1}, [r0,:128], ip /* 将d0和d1寄存器中的数据传递给R0指向的内存 */
17.                                     /* r0=r0+ip */
18.   .endr
19.   bx      lr                       /* 子程序返回 */
20. endfunc

```

可以看出, x264_predict_16x16_v_neon()使用vld1.64指令载入16x16块上方的一行像素, 然后在一个16次的循环中, 使用vst1.64指令将该行像素值赋值给16x16块的每一行。

至此有关Intra16x16的Vertical帧内预测方式的源代码就分析完了。

Intra4x4帧内预测源代码

Intra4x4帧内预测模块的初始化函数是x264_predict_4x4_init()。该函数对x264_predict_结构体中的函数指针进行了赋值。X264运行的过程中只要调用x264_predict_t的函数指针就可以完成相应的功能。

x264_predict_4x4_init()

x264_predict_4x4_init()用于初始化Intra4x4帧内预测汇编函数。该函数的定义位于common\predict.c, 如下所示。

```

1.  //Intra4x4帧内预测汇编函数初始化
2.  void x264_predict_4x4_init( int cpu, x264_predict_t pf[12] )
3.  {
4.      //9种Intra4x4预测方式
5.      pf[I_PRED_4x4_V]   = x264_predict_4x4_v_c;
6.      pf[I_PRED_4x4_H]   = x264_predict_4x4_h_c;
7.      pf[I_PRED_4x4_DC]  = x264_predict_4x4_dc_c;
8.      pf[I_PRED_4x4_DDL] = x264_predict_4x4_ddl_c;
9.      pf[I_PRED_4x4_DDR] = x264_predict_4x4_ddr_c;
10.     pf[I_PRED_4x4_VR]  = x264_predict_4x4_vr_c;
11.     pf[I_PRED_4x4_HD]  = x264_predict_4x4_hd_c;
12.     pf[I_PRED_4x4_VL]  = x264_predict_4x4_vl_c;
13.     pf[I_PRED_4x4_HU]  = x264_predict_4x4_hu_c;
14.     //这些是?
15.     pf[I_PRED_4x4_DC_LEFT] = x264_predict_4x4_dc_left_c;
16.     pf[I_PRED_4x4_DC_TOP]  = x264_predict_4x4_dc_top_c;
17.     pf[I_PRED_4x4_DC_128] = x264_predict_4x4_dc_128_c;
18.
19.     #if HAVE_MMX
20.         x264_predict_4x4_init_mmx( cpu, pf );
21.     #endif
22.
23.     #if HAVE_ARMV6
24.         x264_predict_4x4_init_arm( cpu, pf );
25.     #endif
26.
27.     #if ARCH_AARCH64
28.         x264_predict_4x4_init_aarch64( cpu, pf );
29.     #endif
30. }

```

从源代码可看出, x264_predict_4x4_init()首先对帧内预测函数指针数组x264_predict_t中的元素赋值了C语言版本的函数x264_predict_4x4_v_c(), x264_predict_4x4_h_c(), x264_predict_4x4_dc_c(), x264_predict_4x4_p_c()等一系列函数 (Intra4x4有9种, 后面那几种是怎么回事?); 然后会判断系统平台的特性, 如果平台支持的话, 会调用x264_predict_4x4_init_mmx(), x264_predict_4x4_init_arm()等给x264_predict_t中的元素赋值经过汇编优化的函数。下面看一下Intra4x4帧内预测中Vertical、Horizontal、DC模式的C语言版本函数。

x264_predict_4x4_v_c()

x264_predict_4x4_v_c()实现了Intra4x4帧内预测Vertical模式。该函数的定义位于common\predict.c, 如下所示。

```

1. void x264_predict_4x4_v_c( pixel *src )
2. {
3.     /*
4.      * Vertical预测方式
5.      * |X1 X2 X3 X4
6.      * --+-----
7.      * |X1 X2 X3 X4
8.      * |X1 X2 X3 X4
9.      * |X1 X2 X3 X4
10.     * |X1 X2 X3 X4
11.     */
12.
13.
14.     /*
15.      * 宏展开后的结果如下所示
16.      * 注：重建宏块缓存fdec_buf一行的数据量为32Byte
17.      *
18.      * (((x264_union32_t*)&src[(0)+(0)*32]))->i =
19.      * (((x264_union32_t*)&src[(0)+(1)*32]))->i =
20.      * (((x264_union32_t*)&src[(0)+(2)*32]))->i =
21.      * (((x264_union32_t*)&src[(0)+(3)*32]))->i = (((x264_union32_t*)&src[(0)+(-1)*32]))->i);
22.     */
23.     PREDICT_4x4_DC(SRC_X4(0,-1));
24. }

```

x264_predict_4x4_v_c()函数的函数体极其简单，只有一个宏定义“PREDICT_4x4_DC(SRC_X4(0,-1));”。如果把该宏展开后，可以看出它取了4x4块上面一行4个像素的值，然后分别赋值给4x4块的4行像素。

x264_predict_4x4_h_c()

x264_predict_4x4_h_c()实现了Intra4x4帧内预测Horizontal模式。该函数的定义位于common\predict.c，如下所示。

```

1. void x264_predict_4x4_h_c( pixel *src )
2. {
3.     /*
4.      * Horizontal预测方式
5.      * |
6.      * --+-----
7.      * X5|X5 X5 X5 X5
8.      * X6|X6 X6 X6 X6
9.      * X7|X7 X7 X7 X7
10.     * X8|X8 X8 X8 X8
11.     */
12.
13.
14.     /*
15.      * 宏展开后的结果如下所示
16.      * 注：重建宏块缓存fdec_buf一行的数据量为32Byte
17.      *
18.      * 该代码就是把每行左边的值赋值给该行像素，一次赋值一行
19.      *
20.      * (((x264_union32_t*)&src[(0)+(0)*32]))->i=((src[(-1)+(0)*32])*0x01010101U);
21.      * (((x264_union32_t*)&src[(0)+(1)*32]))->i=((src[(-1)+(1)*32])*0x01010101U);
22.      * (((x264_union32_t*)&src[(0)+(2)*32]))->i=((src[(-1)+(2)*32])*0x01010101U);
23.      * (((x264_union32_t*)&src[(0)+(3)*32]))->i=((src[(-1)+(3)*32])*0x01010101U);
24.      *
25.      * PIXEL_SPLAT_X4()的作用应该是把最后一个像素（最后8位）拷贝给前面3个像素（前24位）
26.      * 即把0x0100009F变成0x9F9F9F9F
27.      * 推导：
28.      * 前提是x占8bit（对应1个像素）
29.      * y=x*0x01010101
30.      * =x*(0x00000001+0x00000100+0x00010000+0x01000000)
31.      * =x<<0+x<<8+x<<16+x<<24
32.      *
33.      * const uint32_t v = (src[-1])*0x01010101U含义：
34.      * 每行把src[-1]中像素值例如0x02赋值给v.v取值为0x02020202
35.      * src[-1]即16x16块左侧的值
36.      *
37.     */
38.
39.     SRC_X4(0,0) = PIXEL_SPLAT_X4( SRC(-1,0) );
40.     SRC_X4(0,1) = PIXEL_SPLAT_X4( SRC(-1,1) );
41.     SRC_X4(0,2) = PIXEL_SPLAT_X4( SRC(-1,2) );
42.     SRC_X4(0,3) = PIXEL_SPLAT_X4( SRC(-1,3) );
43. }

```

从源代码可以看出，x264_predict_4x4_h_c()首先取出4x4块每行左边的1个像素，复制4份后赋值给这一行像素。其中“PIXEL_SPLAT_X4()”的功能是取出变量低8位的数值复制4份到高24位。

x264_predict_4x4_dc_c()

x264_predict_4x4_dc_c()实现了Intra4x4帧内预测DC模式。该函数的定义位于common\predict.c，如下所示。

```
[cpp]
1. void x264_predict_4x4_dc_c( pixel *src )
2. {
3.     /*
4.      * DC预测方式
5.      * |X1 X2 X3 X4
6.      * --+-----
7.      * X5|
8.      * X6|     Y
9.      * X7|
10.     * X8|
11.     *
12.     * Y=(X1+X2+X3+X4+X5+X6+X7+X8)/8
13.     */
14.
15.     /*
16.      * 宏展开后的结果如下所示
17.      * 注：重建宏块缓存fdec_buf一行的数据量为32Byte
18.      * 注2：“+4”是为了四舍五入
19.      *
20.      * uint32_t dc=((src[(-1)+(0)*32] + src[(-1)+(1)*32] + src[(-1)+(2)*32] + src[(-1)+(3)*32] +
21.      *      src[(0)+(-1)*32] + src[(1)+(-1)*32] + src[(2)+(-1)*32] + src[(3)+(-1)*32] + 4) >> 3)*0x01010101U)
22.      *
23.      * 一次赋值一行
24.      * (((x264_union32_t*)(&src[(0)+(0)*32]))->i) =
25.      * (((x264_union32_t*)(&src[(0)+(1)*32]))->i) =
26.      * (((x264_union32_t*)(&src[(0)+(2)*32]))->i) =
27.      * (((x264_union32_t*)(&src[(0)+(3)*32]))->i) = dc;
28.      *
29.      */
30.     pixel4 dc = PIXEL_SPLAT_X4( (SRC(-1,0) + SRC(-1,1) + SRC(-1,2) + SRC(-1,3) +
31.      *      SRC(0,-1) + SRC(1,-1) + SRC(2,-1) + SRC(3,-1) + 4) >> 3 );
32.     PREDICT_4x4_DC( dc );
33. }
```

从源代码可以看出，x264_predict_4x4_dc_c()取出了4x4块左边和上边的8个像素，将它们的平均值赋值给4x4块中的每个像素。

像素计算源代码

像素计算模块的初始化函数是x264_pixel_init()。该函数对x264_pixel_function_t结构体中的函数指针进行了赋值。X264运行的过程中只要调用x264_pixel_function_t的函数指针就可以完成相应的功能。

x264_pixel_init()

x264_pixel_init()初始化像素值计算相关的汇编函数（包括SAD、SATD、SSD等）。该函数的定义位于common\pixel.c，如下所示。

```
[cpp]
1. /*****
2.  * x264_pixel_init:
3.  *****/
4. //SAD等和像素计算有关的函数
5. void x264_pixel_init( int cpu, x264_pixel_function_t *pixf )
6. {
7.     memset( pixf, 0, sizeof(*pixf) );
8.
9.     //初始化2个函数-16x16,16x8
10.    #define INIT2_NAME( name1, name2, cpu ) \
11.        pixf->name1[PIXEL_16x16] = x264_pixel_ ##name2##_16x16##cpu;\
12.        pixf->name1[PIXEL_16x8]   = x264_pixel_ ##name2##_16x8##cpu;
13.    //初始化4个函数-(16x16,16x8),8x16,8x8
14.    #define INIT4_NAME( name1, name2, cpu ) \
15.        INIT2_NAME( name1, name2, cpu ) \
16.        pixf->name1[PIXEL_8x16]   = x264_pixel_ ##name2##_8x16##cpu;\
17.        pixf->name1[PIXEL_8x8]    = x264_pixel_ ##name2##_8x8##cpu;
18.    //初始化5个函数-(16x16,16x8,8x16,8x8),8x4
19.    #define INIT5_NAME( name1, name2, cpu ) \
20.        INIT4_NAME( name1, name2, cpu ) \
21.        pixf->name1[PIXEL_8x4]    = x264_pixel_ ##name2##_8x4##cpu;
22.    //初始化6个函数-(16x16,16x8,8x16,8x8,8x4),4x8
23.    #define INIT6_NAME( name1, name2, cpu ) \
24.        INIT5_NAME( name1, name2, cpu ) \
25.        pixf->name1[PIXEL_4x8]    = x264_pixel_ ##name2##_4x8##cpu;
26.    //初始化7个函数-(16x16,16x8,8x16,8x8,8x4,4x8),4x4
27.    #define INIT7_NAME( name1, name2, cpu ) \
28.        INIT6_NAME( name1, name2, cpu ) \
29.        pixf->name1[PIXEL_4x4]    = x264_pixel_ ##name2##_4x4##cpu;
30.    #define INIT8_NAME( name1, name2, cpu ) \
31.        INIT7_NAME( name1, name2, cpu ) \
32.        pixf->name1[PIXEL_4x16]   = x264_pixel_ ##name2##_4x16##cpu;
33. }
```

```

34. //重新起个名字
35. #define INIT2( name, cpu ) INIT2_NAME( name, name, cpu )
36. #define INIT4( name, cpu ) INIT4_NAME( name, name, cpu )
37. #define INIT5( name, cpu ) INIT5_NAME( name, name, cpu )
38. #define INIT6( name, cpu ) INIT6_NAME( name, name, cpu )
39. #define INIT7( name, cpu ) INIT7_NAME( name, name, cpu )
40. #define INIT8( name, cpu ) INIT8_NAME( name, name, cpu )
41.
42. #define INIT_ADS( cpu ) \
43.     pixf->ads[PIXEL_16x16] = x264_pixel_ads4##cpu;\
44.     pixf->ads[PIXEL_16x8] = x264_pixel_ads2##cpu;\
45.     pixf->ads[PIXEL_8x8] = x264_pixel_ads1##cpu;
46. //8个sad函数
47. INIT8( sad, );
48. INIT8_NAME( sad_aligned, sad, );
49. //7个sad函数-一次性计算3次
50. INIT7( sad_x3, );
51. //7个sad函数-一次性计算4次
52. INIT7( sad_x4, );
53. //8个ssd函数
54. //ssd可以用来计算PSNR
55. INIT8( ssd, );
56. //8个satd函数
57. //satd计算的是经过Hadamard变换后的值
58. INIT8( satd, );
59. //8个satd函数-一次性计算3次
60. INIT7( satd_x3, );
61. //8个satd函数-一次性计算4次
62. INIT7( satd_x4, );
63. INIT4( hadamard_ac, );
64. INIT_ADS( );
65.
66. pixf->sa8d[PIXEL_16x16] = x264_pixel_sa8d_16x16;
67. pixf->sa8d[PIXEL_8x8] = x264_pixel_sa8d_8x8;
68. pixf->var[PIXEL_16x16] = x264_pixel_var_16x16;
69. pixf->var[PIXEL_8x16] = x264_pixel_var_8x16;
70. pixf->var[PIXEL_8x8] = x264_pixel_var_8x8;
71. pixf->var2[PIXEL_8x16] = x264_pixel_var2_8x16;
72. pixf->var2[PIXEL_8x8] = x264_pixel_var2_8x8;
73. //计算UV的
74. pixf->ssd_nv12_core = pixel_ssd_nv12_core;
75. //计算SSIM
76. pixf->ssim_4x4x2_core = ssim_4x4x2_core;
77. pixf->ssim_end4 = ssim_end4;
78. pixf->vsad = pixel_vsad;
79. pixf->asd8 = pixel_asd8;
80.
81. pixf->intra_sad_x3_4x4 = x264_intra_sad_x3_4x4;
82. pixf->intra_satd_x3_4x4 = x264_intra_satd_x3_4x4;
83. pixf->intra_sad_x3_8x8 = x264_intra_sad_x3_8x8;
84. pixf->intra_sa8d_x3_8x8 = x264_intra_sa8d_x3_8x8;
85. pixf->intra_sad_x3_8x8c = x264_intra_sad_x3_8x8c;
86. pixf->intra_satd_x3_8x8c = x264_intra_satd_x3_8x8c;
87. pixf->intra_sad_x3_8x16c = x264_intra_sad_x3_8x16c;
88. pixf->intra_satd_x3_8x16c = x264_intra_satd_x3_8x16c;
89. pixf->intra_sad_x3_16x16 = x264_intra_sad_x3_16x16;
90. pixf->intra_satd_x3_16x16 = x264_intra_satd_x3_16x16;
91.
92. //后面的初始化基本上都是汇编优化过的函数
93.
94. #if HIGH_BIT_DEPTH
95. #if HAVE_MMX
96.     if( cpu&X264_CPU_MMX2 )
97.     {
98.         INIT7( sad, _mmx2 );
99.         INIT7_NAME( sad_aligned, sad, _mmx2 );
100.         INIT7( sad_x3, _mmx2 );
101.         INIT7( sad_x4, _mmx2 );
102.         INIT8( satd, _mmx2 );
103.         INIT7( satd_x3, _mmx2 );
104.         INIT7( satd_x4, _mmx2 );
105.         INIT4( hadamard_ac, _mmx2 );
106.         INIT8( ssd, _mmx2 );
107.         INIT_ADS( _mmx2 );
108.
109.         pixf->ssd_nv12_core = x264_pixel_ssd_nv12_core_mmx2;
110.         pixf->var[PIXEL_16x16] = x264_pixel_var_16x16_mmx2;
111.         pixf->var[PIXEL_8x8] = x264_pixel_var_8x8_mmx2;
112.     #if ARCH_X86
113.         pixf->var2[PIXEL_8x8] = x264_pixel_var2_8x8_mmx2;
114.         pixf->var2[PIXEL_8x16] = x264_pixel_var2_8x16_mmx2;
115.     #endif
116.
117.         pixf->intra_sad_x3_4x4 = x264_intra_sad_x3_4x4_mmx2;
118.         pixf->intra_satd_x3_4x4 = x264_intra_satd_x3_4x4_mmx2;
119.         pixf->intra_sad_x3_8x8 = x264_intra_sad_x3_8x8_mmx2;
120.         pixf->intra_sad_x3_8x8c = x264_intra_sad_x3_8x8c_mmx2;
121.         pixf->intra_satd_x3_8x8c = x264_intra_satd_x3_8x8c_mmx2;
122.         pixf->intra_sad_x3_8x16c = x264_intra_sad_x3_8x16c_mmx2;
123.         pixf->intra_satd_x3_8x16c = x264_intra_satd_x3_8x16c_mmx2;
124.         pixf->intra_sad_x3_16x16 = x264_intra_sad_x3_16x16_mmx2;

```

```

125.     pixf->intra_satd_x3_16x16 = x264_intra_satd_x3_16x16_mmx2;
126. }
127. if( cpu&X264_CPU_SSE2 )
128. {
129.     INIT4_NAME( sad_aligned, sad, _sse2_aligned );
130.     INIT5( ssd, _sse2 );
131.     INIT6( satd, _sse2 );
132.     pixf->satd[PIXEL_4x16] = x264_pixel_satd_4x16_sse2;
133.
134.     pixf->sa8d[PIXEL_16x16] = x264_pixel_sa8d_16x16_sse2;
135.     pixf->sa8d[PIXEL_8x8] = x264_pixel_sa8d_8x8_sse2;
136. #if ARCH_X86_64
137.     pixf->intra_sa8d_x3_8x8 = x264_intra_sa8d_x3_8x8_sse2;
138.     pixf->sa8d_satd[PIXEL_16x16] = x264_pixel_sa8d_satd_16x16_sse2;
139. #endif
140.     pixf->intra_sad_x3_4x4 = x264_intra_sad_x3_4x4_sse2;
141.     pixf->ssd_nv12_core = x264_pixel_ssd_nv12_core_sse2;
142.     pixf->ssim_4x4x2_core = x264_pixel_ssim_4x4x2_core_sse2;
143.     pixf->ssim_end4 = x264_pixel_ssim_end4_sse2;
144.     pixf->var[PIXEL_16x16] = x264_pixel_var_16x16_sse2;
145.     pixf->var[PIXEL_8x8] = x264_pixel_var_8x8_sse2;
146.     pixf->var2[PIXEL_8x8] = x264_pixel_var2_8x8_sse2;
147.     pixf->var2[PIXEL_8x16] = x264_pixel_var2_8x16_sse2;
148.     pixf->intra_sad_x3_8x8 = x264_intra_sad_x3_8x8_sse2;
149. }
150. //此处省略大量的X86、ARM等平台的汇编函数初始化代码
151. }

```

x264_pixel_init()的源代码非常的长，主要原因在于它把C语言版本的函数以及各种平台的汇编函数都写到一块了（不知道现在最新的版本是不是还是这样）。x264_pixel_init()包含了大量和像素计算有关的函数，包括SAD、SATD、SSD、SSIM等等。它的输入参数x264_pixel_function_t是一个结构体，其中包含了各种像素计算的函数接口。x264_pixel_function_t的定义如下所示。

```

1. typedef struct
2. {
3.     x264_pixel_cmp_t sad[8];
4.     x264_pixel_cmp_t ssd[8];
5.     x264_pixel_cmp_t satd[8];
6.     x264_pixel_cmp_t ssim[7];
7.     x264_pixel_cmp_t sa8d[4];
8.     x264_pixel_cmp_t mbcmp[8]; /* either satd or sad for subpel refine and mode decision */
9.     x264_pixel_cmp_t mbcmp_unaligned[8]; /* unaligned mbcmp for subpel */
10.    x264_pixel_cmp_t fpelcmp[8]; /* either satd or sad for fullpel motion search */
11.    x264_pixel_cmp_x3_t fpelcmp_x3[7];
12.    x264_pixel_cmp_x4_t fpelcmp_x4[7];
13.    x264_pixel_cmp_t sad_aligned[8]; /* Aligned SAD for mbcmp */
14.    int (*vsad)( pixel *, intptr_t, int );
15.    int (*asd8)( pixel *pix1, intptr_t stride1, pixel *pix2, intptr_t stride2, int height );
16.    uint64_t (*sa8d_satd[1])( pixel *pix1, intptr_t stride1, pixel *pix2, intptr_t stride2 );
17.
18.    uint64_t (*var[4])( pixel *pix, intptr_t stride );
19.    int (*var2[4])( pixel *pix1, intptr_t stride1,
20.                    pixel *pix2, intptr_t stride2, int *ssd );
21.    uint64_t (*hadamard_ac[4])( pixel *pix, intptr_t stride );
22.
23.    void (*ssd_nv12_core)( pixel *pixuv1, intptr_t stride1,
24.                           pixel *pixuv2, intptr_t stride2, int width, int height,
25.                           uint64_t *ssd_u, uint64_t *ssd_v );
26.    void (*ssim_4x4x2_core)( const pixel *pix1, intptr_t stride1,
27.                             const pixel *pix2, intptr_t stride2, int sums[2][4] );
28.    float (*ssim_end4)( int sum0[5][4], int sum1[5][4], int width );
29.
30.    /* multiple parallel calls to cmp. */
31.    x264_pixel_cmp_x3_t sad_x3[7];
32.    x264_pixel_cmp_x4_t sad_x4[7];
33.    x264_pixel_cmp_x3_t satd_x3[7];
34.    x264_pixel_cmp_x4_t satd_x4[7];
35.
36.    /* abs-diff-sum for successive elimination.
37.     * may round width up to a multiple of 16. */
38.    int (*ads[7])( int enc_dc[4], uint16_t *sums, int delta,
39.                  uint16_t *cost_mv, int16_t *mvs, int width, int thresh );
40.
41.    /* calculate satd or sad of V, H, and DC modes. */
42.    void (*intra_mbcmp_x3_16x16)( pixel *fenc, pixel *fdec, int res[3] );
43.    void (*intra_satd_x3_16x16)( pixel *fenc, pixel *fdec, int res[3] );
44.    void (*intra_sad_x3_16x16)( pixel *fenc, pixel *fdec, int res[3] );
45.    void (*intra_mbcmp_x3_4x4)( pixel *fenc, pixel *fdec, int res[3] );
46.    void (*intra_satd_x3_4x4)( pixel *fenc, pixel *fdec, int res[3] );
47.    void (*intra_sad_x3_4x4)( pixel *fenc, pixel *fdec, int res[3] );
48.    void (*intra_mbcmp_x3_chroma)( pixel *fenc, pixel *fdec, int res[3] );
49.    void (*intra_satd_x3_chroma)( pixel *fenc, pixel *fdec, int res[3] );
50.    void (*intra_sad_x3_chroma)( pixel *fenc, pixel *fdec, int res[3] );
51.    void (*intra_mbcmp_x3_8x16c)( pixel *fenc, pixel *fdec, int res[3] );
52.    void (*intra_satd_x3_8x16c)( pixel *fenc, pixel *fdec, int res[3] );
53.    void (*intra_sad_x3_8x16c)( pixel *fenc, pixel *fdec, int res[3] );
54.    void (*intra_mbcmp_x3_8x8c)( pixel *fenc, pixel *fdec, int res[3] );
55.    void (*intra_satd_x3_8x8c)( pixel *fenc, pixel *fdec, int res[3] );
56.    void (*intra_sad_x3_8x8c)( pixel *fenc, pixel *fdec, int res[3] );
57.    void (*intra_mbcmp_x3_8x8)( pixel *fenc, pixel edge[36], int res[3] );
58.    void (*intra_sa8d_x3_8x8)( pixel *fenc, pixel edge[36], int res[3] );
59.    void (*intra_sad_x3_8x8)( pixel *fenc, pixel edge[36], int res[3] );
60.    /* find minimum satd or sad of all modes, and set fdec.
61.     * may be NULL, in which case just use pred+satd instead. */
62.    int (*intra_mbcmp_x9_4x4)( pixel *fenc, pixel *fdec, uint16_t *bitcosts );
63.    int (*intra_satd_x9_4x4)( pixel *fenc, pixel *fdec, uint16_t *bitcosts );
64.    int (*intra_sad_x9_4x4)( pixel *fenc, pixel *fdec, uint16_t *bitcosts );
65.    int (*intra_mbcmp_x9_8x8)( pixel *fenc, pixel *fdec, pixel edge[36], uint16_t *bitcosts, uint16_t *satds );
66.    int (*intra_sa8d_x9_8x8)( pixel *fenc, pixel *fdec, pixel edge[36], uint16_t *bitcosts, uint16_t *satds );
67.    int (*intra_sad_x9_8x8)( pixel *fenc, pixel *fdec, pixel edge[36], uint16_t *bitcosts, uint16_t *satds );
68. } x264_pixel_function_t;

```

在x264_pixel_init()中定义了好几个宏，用于给x264_pixel_function_t结构体中的函数接口赋值。例如"INIT8(sad,)"用于给x264_pixel_function_t中的sad[8]赋值。该宏展开后的代码如下。

```

1. pixf->sad[PIXEL_16x16] = x264_pixel_sad_16x16;
2. pixf->sad[PIXEL_16x8]  = x264_pixel_sad_16x8;
3. pixf->sad[PIXEL_8x16]  = x264_pixel_sad_8x16;
4. pixf->sad[PIXEL_8x8]   = x264_pixel_sad_8x8;
5. pixf->sad[PIXEL_8x4]   = x264_pixel_sad_8x4;
6. pixf->sad[PIXEL_4x8]   = x264_pixel_sad_4x8;
7. pixf->sad[PIXEL_4x4]   = x264_pixel_sad_4x4;
8. pixf->sad[PIXEL_4x16]  = x264_pixel_sad_4x16;

```

"INIT8(ssd,)" 用于给x264_pixel_function_t中的ssd[8]赋值。该宏展开后的代码如下。

```
[cpp]
1.  pixf->ssd[PIXEL_16x16] = x264_pixel_ssd_16x16;
2.  pixf->ssd[PIXEL_16x8]  = x264_pixel_ssd_16x8;
3.  pixf->ssd[PIXEL_8x16]  = x264_pixel_ssd_8x16;
4.  pixf->ssd[PIXEL_8x8]   = x264_pixel_ssd_8x8;
5.  pixf->ssd[PIXEL_8x4]   = x264_pixel_ssd_8x4;
6.  pixf->ssd[PIXEL_4x8]   = x264_pixel_ssd_4x8;
7.  pixf->ssd[PIXEL_4x4]   = x264_pixel_ssd_4x4;
8.  pixf->ssd[PIXEL_4x16]  = x264_pixel_ssd_4x16;
```

"INIT8(satd,)" 用于给x264_pixel_function_t中的satd[8]赋值。该宏展开后的代码如下。

```
[cpp]
1.  pixf->satd[PIXEL_16x16] = x264_pixel_satd_16x16;
2.  pixf->satd[PIXEL_16x8]  = x264_pixel_satd_16x8;
3.  pixf->satd[PIXEL_8x16]  = x264_pixel_satd_8x16;
4.  pixf->satd[PIXEL_8x8]   = x264_pixel_satd_8x8;
5.  pixf->satd[PIXEL_8x4]   = x264_pixel_satd_8x4;
6.  pixf->satd[PIXEL_4x8]   = x264_pixel_satd_4x8;
7.  pixf->satd[PIXEL_4x4]   = x264_pixel_satd_4x4;
8.  pixf->satd[PIXEL_4x16]  = x264_pixel_satd_4x16;
```

下文打算分别记录SAD、SSD和SATD计算的函数x264_pixel_sad_4x4(), x264_pixel_ssd_4x4(), 和x264_pixel_satd_4x4()。此外再记录一个一次性“批量”计算4个点的函数x264_pixel_sad_x4_4x4()。

x264_pixel_sad_4x4()

x264_pixel_sad_4x4()用于计算4x4块的SAD。该函数的定义位于common\pixel.c，如下所示。

```
[cpp]
1.  static int x264_pixel_sad_4x4( pixel *pix1, intptr_t i_stride_pix1,
2.                                pixel *pix2, intptr_t i_stride_pix2 )
3.  {
4.      int i_sum = 0;
5.      for( int y = 0; y < 4; y++ ) //4个像素
6.      {
7.          for( int x = 0; x < 4; x++ ) //4个像素
8.          {
9.              i_sum += abs( pix1[x] - pix2[x] );//相减之后求绝对值，然后累加
10.         }
11.         pix1 += i_stride_pix1;
12.         pix2 += i_stride_pix2;
13.     }
14.     return i_sum;
15. }
```

可以看出x264_pixel_sad_4x4()将两个4x4图像块对应点相减之后，调用abs()求出绝对值，然后累加到i_sum变量上。

x264_pixel_sad_x4_4x4()

x264_pixel_sad_x4_4x4()用于计算4个4x4块的SAD。该函数的定义位于common\pixel.c，如下所示。

```
[cpp]
1.  static void x264_pixel_sad_x4_4x4( pixel *fenc, pixel *pix0, pixel *pix1, pixel *pix2, pixel *pix3,
2.                                      intptr_t i_stride, int scores[4] )
3.  {
4.      scores[0] = x264_pixel_sad_4x4( fenc, 16, pix0, i_stride );
5.      scores[1] = x264_pixel_sad_4x4( fenc, 16, pix1, i_stride );
6.      scores[2] = x264_pixel_sad_4x4( fenc, 16, pix2, i_stride );
7.      scores[3] = x264_pixel_sad_4x4( fenc, 16, pix3, i_stride );
8.  }
```

可以看出，x264_pixel_sad_x4_4x4()计算了起始点在pix0, pix1, pix2, pix3四个4x4的图像块和fenc之间的SAD，并将结果存储于scores[4]数组中。

x264_pixel_ssd_4x4()

x264_pixel_ssd_4x4()用于计算4x4块的SSD。该函数的定义位于common\pixel.c，如下所示。

```

1. static int x264_pixel_ssd_4x4( pixel *pix1, intptr_t i_stride_pix1,
2. pixel *pix2, intptr_t i_stride_pix2 )
3. {
4.     int i_sum = 0;
5.     for( int y = 0; y < 4; y++ ) //4个像素
6.     {
7.         for( int x = 0; x < 4; x++ ) //4个像素
8.         {
9.             int d = pix1[x] - pix2[x]; //相减
10.            i_sum += d*d;           //平方之后,累加
11.        }
12.        pix1 += i_stride_pix1;
13.        pix2 += i_stride_pix2;
14.    }
15.    return i_sum;
16. }

```

可以看出x264_pixel_ssd_4x4()将两个4x4图像块对应点相减之后，取了平方值，然后累加到i_sum变量上。

x264_pixel_satd_4x4()

x264_pixel_satd_4x4()用于计算4x4块的SATD。该函数的定义位于common\pixel.c，如下所示。

```

1. //SAD (Sum of Absolute Difference) =SAE (Sum of Absolute Error)即绝对误差和
2. //SATD (Sum of Absolute Transformed Difference) 即hadamard变换后再绝对值求和
3. //
4. //为什么帧内模式选择要用SATD?
5. //SAD即绝对误差和，仅反映残差时域差异，影响PSNR值，不能有效反映码流的大小。
6. //SATD即将残差经哈德曼变换的4x4块的预测残差绝对值总和，可以将其看作简单的时频变换，其值在一定程度上可以反映生成码流的大小。
7. //4x4的SATD
8. static NOINLINE int x264_pixel_satd_4x4( pixel *pix1, intptr_t i_pix1, pixel *pix2, intptr_t i_pix2 )
9. {
10.     sum2_t tmp[4][2];
11.     sum2_t a0, a1, a2, a3, b0, b1;
12.     sum2_t sum = 0;
13.
14.     for( int i = 0; i < 4; i++, pix1 += i_pix1, pix2 += i_pix2 )
15.     {
16.         a0 = pix1[0] - pix2[0];
17.         a1 = pix1[1] - pix2[1];
18.         b0 = (a0+a1) + ((a0-a1)<<BITS_PER_SUM);
19.         a2 = pix1[2] - pix2[2];
20.         a3 = pix1[3] - pix2[3];
21.         b1 = (a2+a3) + ((a2-a3)<<BITS_PER_SUM);
22.         tmp[i][0] = b0 + b1;
23.         tmp[i][1] = b0 - b1;
24.     }
25.     for( int i = 0; i < 2; i++ )
26.     {
27.         HADAMARD4( a0, a1, a2, a3, tmp[0][i], tmp[1][i], tmp[2][i], tmp[3][i] );
28.         a0 = abs2(a0) + abs2(a1) + abs2(a2) + abs2(a3);
29.         sum += ((sum_t)a0) + (a0>>BITS_PER_SUM);
30.     }
31.     return sum >> 1;
32. }

```

可以看出x264_pixel_satd_4x4()调用了宏HADAMARD4()用于Hadamard变换的计算，并最终将两个像素块Hadamard变换后对应元素求差的绝对值之后，累加到sum变量上。

mbcmp_init()

Intra宏块帧内预测模式的分析函数x264_mb_analyse_intra()中并没有直接调用x264_pixel_function_t中sad[]/satd[]的函数，而是调用了x264_pixel_function_t的mbcmp[]中的函数。mbcmp[]中实际上就是存储的sad[]/satd[]中的函数。mbcmp_init()函数通过参数决定了mbcmp[]使用sad[]还是satd[]。该函数的定义位于encoder\encoder.c，如下所示。


```

1. //决定了像素比较的时候用SAD还是SATD
2. static void mbcmp_init( x264_t *h )
3. {
4.     //b_lossless一般为0
5.     //主要看i_subpel_refine, 大于1的话就使用SATD
6.     int satd = !h->mb.b_lossless && h->param.analyse.i_subpel_refine > 1;
7.
8.     //sad或者satd赋值给mbcmp
9.     memcpy( h->pixf.mbcmp, satd ? h->pixf.satd : h->pixf.sad_aligned, sizeof(h->pixf.mbcmp) );
10.    memcpy( h->pixf.mbcmp_unaligned, satd ? h->pixf.satd : h->pixf.sad, sizeof(h->pixf.mbcmp_unaligned) );
11.    h->pixf.intra_mbcmp_x3_16x16 = satd ? h->pixf.intra_satd_x3_16x16 : h->pixf.intra_sad_x3_16x16;
12.    h->pixf.intra_mbcmp_x3_8x16c = satd ? h->pixf.intra_satd_x3_8x16c : h->pixf.intra_sad_x3_8x16c;
13.    h->pixf.intra_mbcmp_x3_8x8c = satd ? h->pixf.intra_satd_x3_8x8c : h->pixf.intra_sad_x3_8x8c;
14.    h->pixf.intra_mbcmp_x3_8x8 = satd ? h->pixf.intra_sa8d_x3_8x8 : h->pixf.intra_sad_x3_8x8;
15.    h->pixf.intra_mbcmp_x3_4x4 = satd ? h->pixf.intra_satd_x3_4x4 : h->pixf.intra_sad_x3_4x4;
16.    h->pixf.intra_mbcmp_x9_4x4 = h->param.b_cpu_independent || h->mb.b_lossless ? NULL
17.        : satd ? h->pixf.intra_satd_x9_4x4 : h->pixf.intra_sad_x9_4x4;
18.    h->pixf.intra_mbcmp_x9_8x8 = h->param.b_cpu_independent || h->mb.b_lossless ? NULL
19.        : satd ? h->pixf.intra_sa8d_x9_8x8 : h->pixf.intra_sad_x9_8x8;
20.    satd &= h->param.analyse.i_me_method == X264_ME_TESA;
21.    memcpy( h->pixf.fpelcmp, satd ? h->pixf.satd : h->pixf.sad, sizeof(h->pixf.fpelcmp) );
22.    memcpy( h->pixf.fpelcmp_x3, satd ? h->pixf.satd_x3 : h->pixf.sad_x3, sizeof(h->pixf.fpelcmp_x3) );
23.    memcpy( h->pixf.fpelcmp_x4, satd ? h->pixf.satd_x4 : h->pixf.sad_x4, sizeof(h->pixf.fpelcmp_x4) );
24. }

```

从mbcmp_init()的源代码可以看出, 当i_subpel_refine取值大于1的时候, satd变量为1, 此时后续代码中赋值给mbcmp[]相关的一系列函数指针的函数就是SATD函数; 当i_subpel_refine取值小于等于1的时候, satd变量为0, 此时后续代码中赋值给mbcmp[]相关的一系列函数指针的函数就是SAD函数。

至此有关x264中的Intra宏块分析模块的源代码就分析完毕了。

雷霄骅

leixiaohua1020@126.com

<http://blog.csdn.net/leixiaohua1020>

版权声明：本文为博主原创文章，未经博主允许不得转载。 <https://blog.csdn.net/leixiaohua1020/article/details/45917757>

文章标签： x264 帧内编码 Intra SAD H.264

个人分类： x264

所属专栏： [开源多媒体项目源代码分析](#)

此PDF由spygg生成, 请尊重原作者版权!!!

我的邮箱: liushidc@163.com