```
/*!
************************************************************************
*********************
 * \file mv_search.c
 *
 * \brief
 *   Motion Vector Search, unified for B and P Pictures
 *
 * \author
 *   Main contributors (see contributors.h for copyright, address and
affiliation details)
 *     - Stephan Wenger                <stewe@cs.tu-berlin.de>
 *     - Inge Lille-Langoy
<inge.lille-langoy@telenor.com>
 *     - Rickard Sjoberg
<rickard.sjoberg@era.ericsson.se>
 *     - Stephan Wenger                <stewe@cs.tu-berlin.de>
 *     - Jani Lainema                 <jani.lainema@nokia.com>
 *     - Detlev Marpe
 *     - Thomas Wedi                  <wedi@tnt.uni-hannover.de>
 *     - Heiko Schwarz
 *     - Alexis Michael Tourapis        <alexismt@ieee.org>
 *
************************************************************************
*********************
*/

#include "contributors.h"

#include <math.h>
#include <limits.h>
#include <time.h>

#include "global.h"

#include "image.h"
#include "mv_search.h"
#include "refbuf.h"
#include "memalloc.h"
#include "mb_access.h"
#include "macroblock.h"
#include "mc_prediction.h"
#include "conformance.h"
#include "mode_decision.h"

// Motion estimation distortion header file
#include "me_distortion.h"
```

```
#include "me_distortion_otf.h"

// Motion estimation search algorithms
#include "me_epzs.h"
#include "me_epzs_int.h"
#include "me_fullfast.h"
#include "me_fullfast_otf.h"
#include "me_fullsearch.h"
#include "me_umhex.h"
#include "me_umhexsmp.h"
#include "rdoq.h"


static const short bx0[5][4] = {{0,0,0,0}, {0,0,0,0}, {0,0,0,0},
{0,2,0,0}, {0,2,0,2}};
static const short by0[5][4] = {{0,0,0,0}, {0,0,0,0}, {0,2,0,0},
{0,0,0,0}, {0,0,2,2}};


static distblk GetSkipCostMB        (Macroblock *currMB, int
lambda);
static distblk BiPredBlockMotionSearch(Macroblock *currMB, MEBlock
*, MotionVector*, int, int , int*);

/*****************************************************************
*******/


/*****************************************************************
*******/


/*!

*****************************************************************
*******
 * \brief
 *    Set search range. This needs to be changed to provide 2D support

*****************************************************************
*******
 */
void get_search_range(MEBlock *mv_block, InputParameters *p_Inp,
short ref, int blocktype)
{
  SearchWindow *searchRange = &mv_block->searchRange;

  *searchRange = mv_block->p_Vid->searchRange;
  //----- set search range ---
  if (p_Inp->full_search == 1)
```

```
  {
    int scale = (imin(ref, 1) + 1);
    searchRange->min_x /= scale;
    searchRange->max_x /= scale;
    searchRange->min_y /= scale;
    searchRange->max_y /= scale;
  }
  else if  (p_Inp->full_search != 2)
  {
    int scale = ((imin(ref, 1) + 1) * imin(2, blocktype));
    searchRange->min_x /= scale;
    searchRange->max_x /= scale;
    searchRange->min_y /= scale;
    searchRange->max_y /= scale;
  }
}

/*!

************************************************************************
*******
 * \brief
 *    Set search range. This needs to be changed to provide 2D support

************************************************************************
*******
 */
static inline void set_me_parameters( PicMotionParams **motion,
const MotionVector *all_mv, int list, char ref, int step_h, int step_v,
int pic_block_y, int pic_block_x)
{
  int i, j;

  // Set first line
  for (j = pic_block_y; j < pic_block_y + step_v; j++)
  {
    for (i=pic_block_x; i<pic_block_x + step_h; i++)
    {
      motion[j][i].mv[list] = *all_mv;
      motion[j][i].ref_idx[list] = ref;
    }
  }
}

/*!

************************************************************************
*******
 * \brief
 *    Set ME access method
```

```
************************************************************************
*******
 */
void set_access_method(int *access_method, MotionVector *blk, int
min_x, int min_y, int max_x, int max_y)
{
  if ( (blk->mv_x > min_x) && (blk->mv_x < max_x) && (blk->mv_y > min_y)
&& (blk->mv_y < max_y))
  {
    *access_method = FAST_ACCESS;
  }
  else
  {
    *access_method = UMV_ACCESS;
  }
}

/*!

************************************************************************
*******
 * \brief
 *    Initialize ME engine

************************************************************************
*******
 */
void init_ME_engine(Macroblock *currMB)
{
  InputParameters *p_Inp = currMB->p_Inp;
  VideoParameters *p_Vid = currMB->p_Vid;
  switch (p_Inp->SearchMode[p_Vid->view_id])
  {
   case EPZS:
     EPZS_setup_engine(currMB, p_Inp);
     break;
   case UM_HEX:
     currMB->IntPelME      = UMHEXIntegerPelBlockMotionSearch;
     currMB->BiPredME      =
UMHEXBipredIntegerPelBlockMotionSearch;
     currMB->SubPelBiPredME = sub_pel_bipred_motion_estimation;
     currMB->SubPelME      = UMHEXSubPelBlockME;
     break;
   case UM_HEX_SIMPLE:
     currMB->IntPelME      = smpUMHEXIntegerPelBlockMotionSearch;
     currMB->BiPredME      =
smpUMHEXBipredIntegerPelBlockMotionSearch;
     currMB->SubPelBiPredME = sub_pel_bipred_motion_estimation;
     currMB->SubPelME      = smpUMHEXSubPelBlockME;
```

```
    break;
  case FULL_SEARCH:
    currMB->IntPelME      = full_search_motion_estimation;
    currMB->BiPredME      = full_search_bipred_motion_estimation;
    currMB->SubPelBiPredME = sub_pel_bipred_motion_estimation;
    currMB->SubPelME      = sub_pel_motion_estimation;
    break;
  case FAST_FULL_SEARCH:
  default:
    currMB->IntPelME      = fast_full_search_motion_estimation;
    currMB->BiPredME      = full_search_bipred_motion_estimation;
    currMB->SubPelBiPredME = sub_pel_bipred_motion_estimation;
    currMB->SubPelME      = sub_pel_motion_estimation;
    currMB->p_SetupFastFullPelSearch = (p_Inp->OnTheFlyFractMCP) ?
(SetupFastFullPelSearch_otf):(setup_fast_full_search);
    break;
  }
}

/*!

************************************************************************
*******
 * \brief
 *    Prepare Motion Estimation parameters for single list ME

************************************************************************
*******
 */
void PrepareMEParams(Slice *currSlice, MEBlock *mv_block, int
ChromaMEEnable, int list, int ref)
{
  if (mv_block->apply_weights)
  {
    mv_block->weight_luma = currSlice->wp_weight[list][ref][0];
    mv_block->offset_luma = currSlice->wp_offset[list][ref][0];

    if ( ChromaMEEnable)
    {
      mv_block->weight_cr[0] = currSlice->wp_weight[list][ref][1];
      mv_block->weight_cr[1] = currSlice->wp_weight[list][ref][2];
      mv_block->offset_cr[0] = currSlice->wp_offset[list][ref][1];
      mv_block->offset_cr[1] = currSlice->wp_offset[list][ref][2];
    }
  }
}

/*!
```

```
************************************************************************
*******
 * \brief
 *    Prepare Motion Estimation parameters for bipred list ME

************************************************************************
*******
 */
void PrepareBiPredMEParams(Slice *currSlice, MEBlock *mv_block, int
ChromaMEEnable, int list, int list_offset, int ref)
{
  if (mv_block->apply_weights)
  {
    if (list == LIST_0)
    {
      mv_block->weight1  =
currSlice->wbp_weight[list_offset        ][ref][0][0];
     mv_block->weight2  = currSlice->wbp_weight[list_offset +
LIST_1][ref][0][0];
     mv_block->offsetBi =
(currSlice->wp_offset[list_offset        ][ref][0] +
currSlice->wp_offset[list_offset + LIST_1][ref][0] + 1)>>1;

     if ( ChromaMEEnable)
     {
       mv_block->weight1_cr[0] =
currSlice->wbp_weight[list_offset        ][ref][0][1];
       mv_block->weight1_cr[1] =
currSlice->wbp_weight[list_offset        ][ref][0][2];
       mv_block->weight2_cr[0] = currSlice->wbp_weight[list_offset
+ LIST_1][ref][0][1];
       mv_block->weight2_cr[1] = currSlice->wbp_weight[list_offset
+ LIST_1][ref][0][2];

       mv_block->offsetBi_cr[0] =
(currSlice->wp_offset[list_offset       ][ref][1] +
currSlice->wp_offset[list_offset + LIST_1][ref][1] + 1) >> 1;
       mv_block->offsetBi_cr[1] =
(currSlice->wp_offset[list_offset       ][ref][2] +
currSlice->wp_offset[list_offset + LIST_1][ref][2] + 1) >> 1;
     }
    }
    else
    {
     mv_block->weight1  = currSlice->wbp_weight[list_offset +
LIST_1][0  ][ref][0];
     mv_block->weight2  =
currSlice->wbp_weight[list_offset        ][0  ][ref][0];
```

```
      mv_block->offsetBi = (currSlice->wp_offset[list_offset +
LIST_1][0][0] + currSlice->wp_offset[list_offset][0][0] + 1)>>1;

      if ( ChromaMEEnable)
      {
        mv_block->weight1_cr[0] = currSlice->wbp_weight[list_offset
+ LIST_1][0  ][ref][1];
        mv_block->weight1_cr[1] = currSlice->wbp_weight[list_offset
+ LIST_1][0  ][ref][2];
        mv_block->weight2_cr[0] =
currSlice->wbp_weight[list_offset       ][0  ][ref][1];
        mv_block->weight2_cr[1] =
currSlice->wbp_weight[list_offset       ][0  ][ref][2];

        mv_block->offsetBi_cr[0] =
(currSlice->wp_offset[list_offset + LIST_1][0  ][1] +
currSlice->wp_offset[list_offset       ][0  ][1] + 1) >> 1;
        mv_block->offsetBi_cr[1] =
(currSlice->wp_offset[list_offset + LIST_1][0  ][2] +
currSlice->wp_offset[list_offset       ][0  ][2] + 1) >> 1;
      }
    }
  }
  else
  {
    mv_block->weight1 = (short) (1 <<
currSlice->luma_log_weight_denom);
    mv_block->weight2 = (short) (1 <<
currSlice->luma_log_weight_denom);
    mv_block->offsetBi = 0;
    if ( ChromaMEEnable)
    {
      mv_block->weight1_cr[0] =
1<<currSlice->chroma_log_weight_denom;
      mv_block->weight1_cr[1] =
1<<currSlice->chroma_log_weight_denom;
      mv_block->weight2_cr[0] =
1<<currSlice->chroma_log_weight_denom;
      mv_block->weight2_cr[1] =
1<<currSlice->chroma_log_weight_denom;
      mv_block->offsetBi_cr[0] = 0;
      mv_block->offsetBi_cr[1] = 0;
    }
  }
}

/*!

************************************************************************
*******
```

```
 * \brief
 *    Get current block spatial neighbors

************************************************************************
*******
 */
void get_neighbors(Macroblock *currMB,      // <--  current
Macroblock
                   PixelPos   *block,       // <--> neighbor blocks
                   int        mb_x,        // <--  block x position
                   int        mb_y,        // <--  block y position
                   int        blockshape_x  // <--  block width
                   )
{
  VideoParameters *p_Vid = currMB->p_Vid;
  int *mb_size = p_Vid->mb_size[IS_LUMA];

  get4x4Neighbour(currMB, mb_x - 1,          mb_y    , mb_size,
&block[0]);
  get4x4Neighbour(currMB, mb_x,             mb_y - 1, mb_size,
&block[1]);
  get4x4Neighbour(currMB, mb_x + blockshape_x, mb_y - 1, mb_size,
&block[2]);
  get4x4Neighbour(currMB, mb_x - 1,          mb_y - 1, mb_size,
&block[3]);

  if (mb_y > 0)
  {
    if (mb_x < 8)  // first column of 8x8 blocks
    {
      if (mb_y == 8 )
      {
        if (blockshape_x == MB_BLOCK_SIZE)
          block[2].available  = 0;
      }
      else if (mb_x + blockshape_x == 8)
      {
        block[2].available = 0;
      }
    }
    else if (mb_x + blockshape_x == MB_BLOCK_SIZE)
    {
      block[2].available = 0;
    }
  }

  if (!block[2].available)
  {
    block[2] = block[3];
  }
```

```
}

/*!
************************************************************************
*******
* \brief
*   Initialize the motion search
************************************************************************
*******
*/
void init_motion_search_module (VideoParameters *p_Vid,
InputParameters *p_Inp)
{
  int bits;
  int i_min, i_max,k;
  int i, l;
  int search_range_orig        = p_Inp->SepViewInterSearch ?
imax( p_Inp->search_range[0], p_Inp->search_range[1] ) :
p_Inp->search_range[0];
  int search_range             = search_range_orig;
  int max_search_points        = imax(9, (2 * search_range + 1) * (2
* search_range + 1));
  int max_ref_bits             = 1 + 2 * (int)floor(log(imax(16,
p_Vid->max_num_references + 1)) / log(2) + 1e-10);
  int max_ref                  = (1<<((max_ref_bits>>1)+1))-1;
  int number_of_subpel_positions = 4 * (2*search_range+3);
  int max_mv_bits              = 3 + 2 * (int)ceil
(log(number_of_subpel_positions + 1) / log(2) + 1e-10);
  int max_mvd                  = p_Inp->UseMVLimits?
imax(4*imax(p_Inp->SetMVXLimit, p_Inp->SetMVYLimit),
((1<<( max_mv_bits >>1) ) - 1)): ((1<<( max_mv_bits >>1)) - 1);

  p_Vid->max_mvd = max_mvd;
  p_Vid->imgpel_abs_range        =
(imax(p_Vid->max_pel_value_comp[0],p_Vid->max_pel_value_comp[1])
+ 1) * 64;

  //=====   CREATE ARRAYS   =====
  //----------------------------
  if ((p_Vid->spiral_search =
(MotionVector*)calloc(max_search_points, sizeof(MotionVector))) ==
NULL)
    no_mem_exit("init_motion_search_module:
p_Vid->spiral_search");
  if ((p_Vid->spiral_hpel_search =
(MotionVector*)calloc(max_search_points, sizeof(MotionVector))) ==
NULL)
    no_mem_exit("init_motion_search_module:
p_Vid->spiral_hpel_search");
```

```
  if ((p_Vid->spiral_qpel_search =
(MotionVector*)calloc(max_search_points, sizeof(MotionVector))) ==
NULL)
    no_mem_exit("init_motion_search_module:
p_Vid->spiral_qpel_search");


  if ((p_Vid->mvbits = (int*)calloc(2 * max_mvd + 1, sizeof(int)))
== NULL)
    no_mem_exit("init_motion_search_module: p_Vid->mvbits");

  if ((p_Vid->refbits = (int*)calloc(max_ref, sizeof(int))) == NULL)
    no_mem_exit("init_motion_search_module: p_Vid->refbits");

#if (JM_MEM_DISTORTION)
  if ((p_Vid->imgpel_abs = (int*)calloc(p_Vid->imgpel_abs_range,
sizeof(int))) == NULL)
    no_mem_exit("init_motion_search_module: p_Vid->imgpel_abs");
  if ((p_Vid->imgpel_quad = (int*)calloc(p_Vid->imgpel_abs_range,
sizeof(int))) == NULL)
    no_mem_exit("init_motion_search_module: p_Vid->imgpel_quad");
  p_Vid->imgpel_abs  += p_Vid->imgpel_abs_range / 2;
  p_Vid->imgpel_quad += p_Vid->imgpel_abs_range / 2;
#endif

  if (p_Vid->max_num_references)
    get_mem4Ddistblk (&p_Vid->motion_cost, 8, 2,
p_Vid->max_num_references, 4);

  //--- set array offsets ---
  p_Vid->mvbits     += max_mvd;

  //=====   INIT ARRAYS   =====
  //---------------------------
  //--- init array: motion vector bits ---
  p_Vid->mvbits[0] = 1;
  for (bits = 3; bits <= max_mv_bits; bits += 2)
  {
    i_max = (short) (1 << (bits >> 1));
    i_min = i_max >> 1;

    for (i = i_min; i < i_max; i++)
      p_Vid->mvbits[-i] = p_Vid->mvbits[i] = bits;
  }

  //--- init array: reference frame bits ---
  p_Vid->refbits[0] = 1;
  for (bits=3; bits<=max_ref_bits; bits+=2)
  {
    i_max = (short) (1 << ((bits >> 1) + 1)) - 1;
```

```
    i_min = i_max >> 1;

    for (i = i_min; i < i_max; i++)
      p_Vid->refbits[i] = bits;
  }

#if (JM_MEM_DISTORTION)
  //--- init array: absolute value ---
  p_Vid->imgpel_abs[0] = 0;

  for (i=1; i<p_Vid->imgpel_abs_range / 2; i++)
  {
    p_Vid->imgpel_abs[i] = p_Vid->imgpel_abs[-i] = i;
  }

  //--- init array: square value ---
  p_Vid->imgpel_quad[0] = 0;

  for (i=1; i<p_Vid->imgpel_abs_range / 2; i++)
  {
    p_Vid->imgpel_quad[i] = p_Vid->imgpel_quad[-i] = i * i;
  }
#endif

  //--- init array: search pattern ---
  p_Vid->spiral_search[0].mv_x = p_Vid->spiral_search[0].mv_y = 0;
  p_Vid->spiral_hpel_search[0].mv_x =
p_Vid->spiral_hpel_search[0].mv_y = 0;
  p_Vid->spiral_qpel_search[0].mv_x =
p_Vid->spiral_qpel_search[0].mv_y = 0;

  for (k=1, l=1; l <= imax(1,search_range); l++)
  {
    for (i=-l+1; i< l; i++)
    {
      p_Vid->spiral_search[k].mv_x =    (short)  i;
      p_Vid->spiral_search[k].mv_y =    (short) -l;
      p_Vid->spiral_hpel_search[k].mv_x =  (short) (i<<1);
      p_Vid->spiral_hpel_search[k].mv_y =  (short) -(l<<1);
      p_Vid->spiral_qpel_search[k].mv_x =  (short)  (i<<2);
      p_Vid->spiral_qpel_search[k++].mv_y = (short) -(l<<2);
      p_Vid->spiral_search[k].mv_x =    (short)  i;
      p_Vid->spiral_search[k].mv_y =    (short)  l;
      p_Vid->spiral_hpel_search[k].mv_x =  (short) (i<<1);
      p_Vid->spiral_hpel_search[k].mv_y =  (short) (l<<1);
      p_Vid->spiral_qpel_search[k].mv_x =  (short) (i<<2);
      p_Vid->spiral_qpel_search[k++].mv_y = (short) (l<<2);
    }
    for (i=-l;   i<=l; i++)
    {
```

```
      p_Vid->spiral_search[k].mv_x =    (short) -l;
      p_Vid->spiral_search[k].mv_y =    (short)  i;
      p_Vid->spiral_hpel_search[k].mv_x =  (short) -(l<<1);
      p_Vid->spiral_hpel_search[k].mv_y =  (short)  (i<<1);
      p_Vid->spiral_qpel_search[k].mv_x =  (short) -(l<<2);
      p_Vid->spiral_qpel_search[k++].mv_y = (short)  (i<<2);
      p_Vid->spiral_search[k].mv_x =    (short)  l;
      p_Vid->spiral_search[k].mv_y =    (short)  i;
      p_Vid->spiral_hpel_search[k].mv_x =  (short) (l<<1);
      p_Vid->spiral_hpel_search[k].mv_y =  (short) (i<<1);
      p_Vid->spiral_qpel_search[k].mv_x =  (short) (l<<2);
      p_Vid->spiral_qpel_search[k++].mv_y = (short) (i<<2);
    }
  }

  // set global variable prior to ME
  p_Vid->start_me_refinement_hp = (p_Inp->ChromaMEEnable == 1 ||
p_Inp->MEErrorMetric[F_PEL] != p_Inp->MEErrorMetric[H_PEL] ) ? 0 :
1;
  p_Vid->start_me_refinement_qp = (p_Inp->ChromaMEEnable == 1 ||
p_Inp->MEErrorMetric[H_PEL] != p_Inp->MEErrorMetric[Q_PEL] ) ? 0 :
1;

  select_distortion(p_Vid, p_Inp);

  // Setup Distortion Metrics depending on refinement level
  if( p_Inp->OnTheFlyFractMCP )
  {
    for (i=0; i<3; i++)
    {
      switch( p_Inp->MEErrorMetric[i])
      {
      case ERROR_SAD:
        p_Vid->computeUniPred[i]     = computeSAD_otf ;
        p_Vid->computeUniPred[i + 3]  = computeSADWP_otf ;
        p_Vid->computeBiPred1[i]     = computeBiPredSAD1_otf ;
        p_Vid->computeBiPred2[i]     = computeBiPredSAD2_otf ;
        break;
      case ERROR_SSE:
        p_Vid->computeUniPred[i]    = computeSSE_otf;
        p_Vid->computeUniPred[i + 3] = computeSSEWP_otf;
        p_Vid->computeBiPred1[i]    = computeBiPredSSE1_otf;
        p_Vid->computeBiPred2[i]    = computeBiPredSSE2_otf;
        break;
      case ERROR_SATD :
      default:
        p_Vid->computeUniPred[i]    = computeSATD_otf ;
        p_Vid->computeUniPred[i + 3] = computeSATDWP_otf ;
        p_Vid->computeBiPred1[i]    = computeBiPredSATD1_otf ;
        p_Vid->computeBiPred2[i]    = computeBiPredSATD2_otf ;
```

```
      break;
    }
  }
 }
 else
 {
  for (i=0; i<3; i++)
  {
    switch(p_Inp->MEErrorMetric[i])
    {
    case ERROR_SAD:
      p_Vid->computeUniPred[i] = computeSAD;
      p_Vid->computeUniPred[i + 3] = computeSADWP;
      p_Vid->computeBiPred1[i] = computeBiPredSAD1;
      p_Vid->computeBiPred2[i] = computeBiPredSAD2;
      break;
    case ERROR_SSE:
      p_Vid->computeUniPred[i] = computeSSE;
      p_Vid->computeUniPred[i + 3] = computeSSEWP;
      p_Vid->computeBiPred1[i] = computeBiPredSSE1;
      p_Vid->computeBiPred2[i] = computeBiPredSSE2;
      break;
    case ERROR_SATD :
    default:
      p_Vid->computeUniPred[i] = computeSATD;
      p_Vid->computeUniPred[i + 3] = computeSATDWP;
      p_Vid->computeBiPred1[i] = computeBiPredSATD1;
      p_Vid->computeBiPred2[i] = computeBiPredSATD2;
      break;
    }
  }
 }
 if (!p_Inp->IntraProfile)
 {
   if(p_Inp->SearchMode[0] == FAST_FULL_SEARCH ||
p_Inp->SearchMode[1] == FAST_FULL_SEARCH)
    initialize_fast_full_search (p_Vid, p_Inp);

   if (p_Inp->SearchMode[0] == UM_HEX || p_Inp->SearchMode[1] ==
UM_HEX)
    UMHEX_DefineThreshold(p_Vid);
 }
}

/*!

************************************************************************
*******
 * \brief
 *    Free memory used by motion search
```

```
************************************************************************
*******
 */
void
clear_motion_search_module (VideoParameters *p_Vid,
InputParameters *p_Inp)
{
  //int search_range            = p_Inp->search_range;
  //int number_of_subpel_positions = 4 * (2*search_range+3);
  //int max_mv_bits             = 3 + 2 * (int)ceil
(log(number_of_subpel_positions + 1) / log(2) + 1e-10);
  int max_mvd                 = p_Vid->max_mvd;
//(1<<( max_mv_bits >>1)   ) - 1;


  //--- correct array offset ---
  p_Vid->mvbits     -= max_mvd;
#if (JM_MEM_DISTORTION)
  p_Vid->imgpel_abs  -= p_Vid->imgpel_abs_range / 2;
  p_Vid->imgpel_quad -= p_Vid->imgpel_abs_range / 2;
#endif

  //--- delete arrays ---
  free (p_Vid->spiral_search);
  free (p_Vid->spiral_hpel_search);
  free (p_Vid->spiral_qpel_search);
  free (p_Vid->mvbits);
  free (p_Vid->refbits);

#if (JM_MEM_DISTORTION)
  free (p_Vid->imgpel_abs);
  free (p_Vid->imgpel_quad);
#endif


  if (p_Vid->motion_cost)
    free_mem4Ddistblk (p_Vid->motion_cost);

  if ((p_Inp->SearchMode[0] == FAST_FULL_SEARCH ||
p_Inp->SearchMode[1] == FAST_FULL_SEARCH) &&
(!p_Inp->IntraProfile) )
    clear_fast_full_search (p_Vid);
}

static inline int mv_bit_cost(Macroblock *currMB, MotionVector
**all_mv, int cur_list, short cur_ref, int by, int bx, int step_v0,
int step_v, int step_h0, int step_h, int mvd_bits)
{
  int v, h;
```

```
  MotionVector predMV;
  PixelPos block[4];  // neighbor blocks
  VideoParameters *p_Vid = currMB->p_Vid;

  for (v=by; v<by + step_v0; v+=step_v)
  {
    for (h=bx; h<bx + step_h0; h+=step_h)
    {

      get_neighbors(currMB, block, h, v, step_h);
      // Lets recompute MV predictor. This should avoid any problems
with alterations of the motion vectors after ME
      currMB->GetMVPredictor (currMB, block, &predMV, cur_ref,
p_Vid->enc_picture->mv_info, cur_list, h, v, step_h, step_v);

      mvd_bits += p_Vid->mvbits[ all_mv[v>>2][h>>2].mv_x -
predMV.mv_x ];
      mvd_bits += p_Vid->mvbits[ all_mv[v>>2][h>>2].mv_y -
predMV.mv_y ];
    }
  }

  return mvd_bits;
}

/*!

************************************************************************
******
 * \brief
 *    Motion Cost for Bidirectional modes

************************************************************************
******
 */
distblk BPredPartitionCost (Macroblock *currMB,
                    int   blocktype,
                    int   block8x8,
                    short ref_l0,
                    short ref_l1,
                    int   lambda_factor,
                    int   list)
{
  VideoParameters *p_Vid = currMB->p_Vid;
  Slice *currSlice = currMB->p_Slice;
  DecodedPictureBuffer *p_Dpb =
p_Vid->p_Dpb_layer[p_Vid->dpb_layer_id];
  imgpel **cur_img = p_Vid->pCurImg;

  short pic_pix_x, pic_pix_y;
```

```
  short  v, h;
  distblk mcost;

  int   mvd_bits  = 0;

  short parttype  = (short) (blocktype < 4 ? blocktype : 4);
  short step_h0   = block_size[ parttype][0];
  short step_v0   = block_size[ parttype][1];
  short step_h    = block_size[blocktype][0];
  short step_v    = block_size[blocktype][1];
  short by0_part  = by0[parttype][block8x8] << 2;
  short bx0_part  = bx0[parttype][block8x8] << 2;
  short block_size_x = block_size[blocktype][0];
  short block_size_y = block_size[blocktype][1];

  MotionVector  **all_mv_l0 =
currSlice->bipred_mv[list][LIST_0][ref_l0][blocktype];
  MotionVector  **all_mv_l1 =
currSlice->bipred_mv[list][LIST_1][ref_l1][blocktype];
  imgpel  **mb_pred   = currSlice->mb_pred[0];

  // List0
  mvd_bits = mv_bit_cost(currMB, all_mv_l0, LIST_0, ref_l0, by0_part,
bx0_part, step_v0, step_v, step_h0, step_h, mvd_bits);
  // List1
  mvd_bits = mv_bit_cost(currMB, all_mv_l1, LIST_1, ref_l1, by0_part,
bx0_part, step_v0, step_v, step_h0, step_h, mvd_bits);

  mcost = weighted_cost (lambda_factor, mvd_bits);

  // Get prediction
  for (v = by0_part; v < by0_part + step_v0; v = (short) (v +
block_size_y))
  {
    for (h = bx0_part; h < bx0_part + step_h0; h = (short) (h +
block_size_x))
    {
      p_Dpb->pf_luma_prediction_bi (currMB, h, v, block_size_x,
block_size_y, blocktype, blocktype, ref_l0, ref_l1, list);
    }
  }

  //----- cost of residual signal -----
  if ((!currSlice->p_Inp->Transform8x8Mode) || (blocktype>4))
  {
    short diff16[16];
    short *diff;

    pic_pix_y = currMB->opix_y;
    pic_pix_x = currMB->pix_x;
```

```
    for (v = by0_part; v < by0_part + step_v0; v += 4)
    {
      for (h = bx0_part; h < bx0_part + step_h0; h += 4)
      {
        diff = diff16;
        calcDifference(cur_img, pic_pix_x+h, pic_pix_y+v, mb_pred, h,
v, 4, 4, diff);
        mcost += p_Vid->distortion4x4 (diff16, DISTBLK_MAX);
      }
    }
  }
  else
  {
  short diff64[64];
  short *diff;

    pic_pix_y = currMB->opix_y;
    pic_pix_x = currMB->pix_x;
    for (v = by0_part; v < by0_part + step_v0; v += 8)
    {
      for (h = bx0_part; h < bx0_part + step_h0; h += 8)
      {
        diff = diff64;
        calcDifference(cur_img, pic_pix_x+h, pic_pix_y+v, mb_pred, h,
v, 8, 8, diff);
        mcost += p_Vid->distortion8x8(diff64, DISTBLK_MAX);
      }
    }
  }

  return mcost;
}

void update_mv_block(Macroblock *currMB, MEBlock *mv_block, int h,
int v)
{
  mv_block->block_x     = (short) h;
  mv_block->block_y     = (short) v;
  mv_block->pos_x       = (short) (currMB->pix_x  + (h << 2));
  mv_block->pos_y       = (short) (currMB->opix_y + (v << 2));
  mv_block->pos_x2      = (short) (mv_block->pos_x >> 2);
  mv_block->pos_y2      = (short) (mv_block->pos_y >> 2);
  mv_block->pos_x_padded = (short) (mv_block->pos_x << 2);
  mv_block->pos_y_padded = (short) (mv_block->pos_y << 2);

  mv_block->pos_cr_x    = (short) (mv_block->pos_x >>
currMB->p_Vid->shift_cr_x);
  mv_block->pos_cr_y    = (short) (mv_block->pos_y >>
currMB->p_Vid->shift_cr_y);
}
```

```
/*!

************************************************************************
******
 * \brief
 *    Init motion vector block

************************************************************************
******
 */
void init_mv_block(Macroblock *currMB, MEBlock *mv_block, short
blocktype, int list, char ref_idx, short mb_x, short mb_y)
{
  InputParameters *p_Inp = currMB->p_Inp;
  VideoParameters *p_Vid = currMB->p_Vid;
  Slice *currSlice = currMB->p_Slice;
  mv_block->blocktype       = blocktype;
  mv_block->blocksize_x     = block_size[blocktype][0];  //
horizontal block size
  mv_block->blocksize_y     = block_size[blocktype][1];  //
vertical block size
  // update position info
  update_mv_block(currMB, mv_block, mb_x, mb_y);

  mv_block->list           = (char) list;
  mv_block->ref_idx        = ref_idx;

  mv_block->mv[LIST_0].mv_x   = 0;
  mv_block->mv[LIST_0].mv_y   = 0;
  mv_block->mv[LIST_1].mv_x   = 0;
  mv_block->mv[LIST_1].mv_y   = 0;
  // Init WP parameters
  mv_block->p_Vid          = p_Vid;
  mv_block->p_Slice        = currSlice;
  mv_block->cost           = INT_MAX;
  mv_block->search_pos2     = 9;
  mv_block->search_pos4     = 9;

  if (p_Inp->ChromaMEEnable)
    get_mem2Dpel(&mv_block->orig_pic, 3, mv_block->blocksize_x *
mv_block->blocksize_y);
  else
    get_mem2Dpel(&mv_block->orig_pic, 1, mv_block->blocksize_x *
mv_block->blocksize_y);

  mv_block->ChromaMEEnable = p_Inp->ChromaMEEnable;
```

```
  mv_block->apply_bi_weights = p_Inp->UseWeightedReferenceME &&
((currSlice->slice_type == B_SLICE) &&
p_Vid->active_pps->weighted_bipred_idc != 0);
  mv_block->apply_weights    = p_Inp->UseWeightedReferenceME &&
( currSlice->weighted_prediction != 0 );

  if (p_Inp->ChromaMEEnable)
  {
    mv_block->blocksize_cr_x = (short) (mv_block->blocksize_x >>
p_Vid->shift_cr_x);
    mv_block->blocksize_cr_y = (short) (mv_block->blocksize_y >>
p_Vid->shift_cr_y);

    mv_block->ChromaMEWeight = p_Inp->ChromaMEWeight;
  }

  if (mv_block->apply_weights)
  {
    // If implicit WP, single list weights are always non weighted
    if ((currSlice->slice_type == B_SLICE) &&
(p_Vid->active_pps->weighted_bipred_idc == 2))
    {
      mv_block->computePredFPel   = p_Vid->computeUniPred[F_PEL];
      mv_block->computePredHPel   = p_Vid->computeUniPred[H_PEL];
      mv_block->computePredQPel   = p_Vid->computeUniPred[Q_PEL];
    }
    else
    {
      mv_block->computePredFPel   = p_Vid->computeUniPred[F_PEL +
3];
      mv_block->computePredHPel   = p_Vid->computeUniPred[H_PEL +
3];
      mv_block->computePredQPel   = p_Vid->computeUniPred[Q_PEL +
3];
    }
    mv_block->computeBiPredFPel = p_Vid->computeBiPred2[F_PEL];
    mv_block->computeBiPredHPel = p_Vid->computeBiPred2[H_PEL];
    mv_block->computeBiPredQPel = p_Vid->computeBiPred2[Q_PEL];
  }
  else
  {
    mv_block->computePredFPel   = p_Vid->computeUniPred[F_PEL];
    mv_block->computePredHPel   = p_Vid->computeUniPred[H_PEL];
    mv_block->computePredQPel   = p_Vid->computeUniPred[Q_PEL];
    mv_block->computeBiPredFPel = p_Vid->computeBiPred1[F_PEL];
    mv_block->computeBiPredHPel = p_Vid->computeBiPred1[H_PEL];
    mv_block->computeBiPredQPel = p_Vid->computeBiPred1[Q_PEL];
  }
}
```

```
/*!

************************************************************************
******
 * \brief
 *    free motion vector block

************************************************************************
******
 */
void free_mv_block(MEBlock *mv_block)
{
  if (mv_block->orig_pic)
  {
    free_mem2Dpel(mv_block->orig_pic);
  }
}


void get_original_block(VideoParameters *p_Vid, MEBlock *mv_block)
{
  //==================================
  //=====   GET ORIGINAL BLOCK   =====
  //==================================
  imgpel *orig_pic_tmp = mv_block->orig_pic[0];
  int   bsx      = mv_block->blocksize_x;
  int   pic_pix_x = mv_block->pos_x;
  int   i, j;
  imgpel **cur_img = &p_Vid->pCurImg[mv_block->pos_y];

  for (j = 0; j < mv_block->blocksize_y; j++)
  {
    memcpy(orig_pic_tmp,&cur_img[j][pic_pix_x], bsx *
sizeof(imgpel));
    orig_pic_tmp += bsx;
  }

  if ( p_Vid->p_Inp->ChromaMEEnable )
  {
    bsx      = mv_block->blocksize_cr_x;
    pic_pix_x = mv_block->pos_cr_x;

    // copy the original cmp1 and cmp2 data to the orig_pic matrix
    for ( i = 1; i<=2; i++)
    {
      cur_img = &p_Vid->pImgOrg[i][mv_block->pos_cr_y];
      orig_pic_tmp = mv_block->orig_pic[i];
      for (j = 0; j < mv_block->blocksize_cr_y; j++)
      {
```

```
      memcpy(orig_pic_tmp, &(cur_img[j][pic_pix_x]), bsx *
sizeof(imgpel));
      orig_pic_tmp += bsx;
    }
  }
 }
}

void CheckSearchRange(VideoParameters *p_Vid, MotionVector *pPredMV,
MotionVector *pSWC, MEBlock *mv_block)
{
 int iMaxMVD = p_Vid->max_mvd - 2;
 SearchWindow *searchRange = &mv_block->searchRange;
 int left  = pSWC->mv_x + searchRange->min_x;
 int right = pSWC->mv_x + searchRange->max_x;
 int top   = pSWC->mv_y + searchRange->min_y;
 int down  = pSWC->mv_y + searchRange->max_y;

 left  = iClip3(pPredMV->mv_x - iMaxMVD, pPredMV->mv_x + iMaxMVD,
left);
 right = iClip3(pPredMV->mv_x - iMaxMVD, pPredMV->mv_x + iMaxMVD,
right);
 top   = iClip3(pPredMV->mv_y - iMaxMVD, pPredMV->mv_y + iMaxMVD,
top);
 down  = iClip3(pPredMV->mv_y - iMaxMVD, pPredMV->mv_y + iMaxMVD,
down);

 if(left<right && top<down)
 {
  pSWC->mv_x = (short) ((left + right)>>1);
  pSWC->mv_y = (short) ((top + down)>>1);
  searchRange->min_x = left - pSWC->mv_x;
  searchRange->max_x = imin(pSWC->mv_x-left, right-pSWC->mv_x);
  searchRange->min_y = top - pSWC->mv_y;
  searchRange->max_y = imin(pSWC->mv_y-top, down-pSWC->mv_y);
 }
 else
 {
  *pSWC = *pPredMV;
 }
}

/*!

************************************************************************
******
 * \brief
 *   Block motion search
```

```
************************************************************************
******
 */
distblk                                     //!< minimum motion cost
after search
BlockMotionSearch (Macroblock *currMB,      //!< Current Macroblock
                   MEBlock   *mv_block,    //!< Motion estimation
information block
                   int      mb_x,         //!< x-coordinate inside
macroblock
                   int      mb_y,         //!< y-coordinate inside
macroblock
                   int*     lambda_factor) //!< lagrangian parameter
for determining motion cost
{


/************************************************************************
*******/
//    FILE * motion_vector_fp;
//    motion_vector_fp =
fopen("/Users/liangsiyang/Documents/USC-learning/EE-669/HW3/motio
n_vector.dat", "a");
//    if (motion_vector_fp == NULL) {
//       printf("ERROR in creating motion vector storage file!\n");
//    }
/************************************************************************
*******/


  // each 48-pel line stores the 16 luma pels (at 0) followed by 8
or 16 crcb[0] (at 16) and crcb[1] (at 32) pels
  // depending on the type of chroma subsampling used: YUV 4:4:4, 4:2:2,
and 4:2:0
  Slice *currSlice = currMB->p_Slice;
  VideoParameters *p_Vid = currMB->p_Vid;
  InputParameters *p_Inp = currMB->p_Inp;

  int   i, j;
  distblk   max_value = DISTBLK_MAX;
  distblk   min_mcost = max_value;
  int   block_x   = (mb_x>>2);
  int   block_y   = (mb_y>>2);

  int   bsx      = mv_block->blocksize_x;
  int   bsy      = mv_block->blocksize_y;

  short pic_pix_x = (short) (currMB->pix_x + mb_x);
```

```
  int  blocktype = mv_block->blocktype;
  int  list = mv_block->list;
  short ref = mv_block->ref_idx;
  MotionVector *mv = &mv_block->mv[list], pred;

  MotionVector **all_mv =
&currSlice->all_mv[list][ref][blocktype][block_y];

  distblk *prevSad = (p_Inp->SearchMode[p_Vid->view_id] == EPZS)?
currSlice->p_EPZS->distortion[list +
currMB->list_offset][blocktype - 1]: NULL;

  get_neighbors(currMB, mv_block->block, mb_x, mb_y, bsx);

  PrepareMEParams(currSlice, mv_block, p_Inp->ChromaMEEnable, list
+ currMB->list_offset, ref);

  //===================================
  //=====   GET ORIGINAL BLOCK   =====
  //===================================
  if (blocktype > 4)
    get_original_block(p_Vid, mv_block);

  //===========================================
  //=====   GET MOTION VECTOR PREDICTOR   =====
  //===========================================
  if (p_Inp->SearchMode[p_Vid->view_id] == UM_HEX)
  {
    p_Vid->p_UMHex->UMHEX_blocktype = blocktype;
    p_Vid->p_UMHex->bipred_flag = 0;
    UMHEXSetMotionVectorPredictor(currMB, &pred,
p_Vid->enc_picture->mv_info, ref, list, mb_x, mb_y, bsx, bsy,
mv_block);
  }
  else if (p_Inp->SearchMode[p_Vid->view_id] == UM_HEX_SIMPLE)
  {
    smpUMHEX_setup(currMB, ref, list, block_y, block_x, blocktype,
currSlice->all_mv );
    currMB->GetMVPredictor (currMB, mv_block->block, &pred, ref,
p_Vid->enc_picture->mv_info, list, mb_x, mb_y, bsx, bsy);
  }
  else
  {
    currMB->GetMVPredictor (currMB, mv_block->block, &pred, ref,
p_Vid->enc_picture->mv_info, list, mb_x, mb_y, bsx, bsy);
  }



  //===================================
```

```
  //=====   INTEGER-PEL SEARCH   =====
  //==================================
  if (p_Inp->EPZSSubPelGrid)
  {
    *mv = pred;
  }
  else
  {
#if (JM_INT_DIVIDE)
    mv->mv_x = (short) (((pred.mv_x  + 2) >> 2) * 4);
    mv->mv_y = (short) (((pred.mv_y  + 2) >> 2) * 4);
#else
    mv->mv_x = (short) ((pred.mv_x / 4) * 4);
    mv->mv_y = (short) ((pred.mv_y / 4) * 4);
#endif
  }

  if (p_Inp->DisableMEPrediction == TRUE)
  {
    mv->mv_x = 0;
    mv->mv_y = 0;
  }

  if (!p_Inp->rdopt)
  {
    MotionVector center = *mv;
    //--- adjust search center so that the (0,0)-vector is inside ---
    mv->mv_x = (short) iClip3 (mv_block->searchRange.min_x,
mv_block->searchRange.max_x, mv->mv_x);
    mv->mv_y = (short) iClip3 (mv_block->searchRange.min_y,
mv_block->searchRange.max_y, mv->mv_y);
    //mvbits overflow checking;
    if((mv->mv_x != center.mv_x) || (mv->mv_y != center.mv_y))
      CheckSearchRange(p_Vid, &center, mv, mv_block);
  }

  // valid search range limits could be precomputed once during the
initialization process
  clip_mv_range(p_Vid, 0, mv, Q_PEL);

  //--- perform motion search ---
  min_mcost = currMB->IntPelME (currMB, &pred, mv_block, min_mcost,
lambda_factor[F_PEL]);

    /*********************************************** changes
*******************************************************/

//    fwrite(&mv->mv_x, sizeof(short), 1, motion_vector_fp);
//
//    fwrite(&mv->mv_y, sizeof(short), 1, motion_vector_fp);
```

```
//
//    fclose(motion_vector_fp);
//    //printf("\nInteger-pel motion vector x=%d, y=%d\n", mv->mv_x,
mv->mv_y);
//
    /********************************************** end changes
**f**********************************************/

  //==============================
  //=====   SUB-PEL SEARCH   =====
  //==============================
  mv_block->ChromaMEEnable = (p_Inp->ChromaMEEnable ==
ME_YUV_FP_SP ) ? TRUE : FALSE; // set it externally

  if (!p_Inp->DisableSubpelME[p_Vid->view_id])
  {
    if (p_Inp->SearchMode[p_Vid->view_id] != EPZS || (ref == 0 ||
currSlice->structure != FRAME || (ref > 0 && min_mcost < 3.5 *
prevSad[pic_pix_x >> 2])))
    {
      if ( !p_Vid->start_me_refinement_hp )
      {
        min_mcost = max_value;
      }
      min_mcost =  currMB->SubPelME (currMB, &pred, mv_block,
min_mcost, lambda_factor);
    }
  }

  // clip mvs after me is performed (is not exactly the best)
  // better solution is to modify search window appropriately
  clip_mv_range(p_Vid, 0, mv, Q_PEL);

  if (!p_Inp->rdopt)
  {
    // Get the skip mode cost
    if (blocktype == 1 && (currSlice->slice_type == P_SLICE||
(currSlice->slice_type == SP_SLICE) ))
    {
      distblk cost;
      FindSkipModeMotionVector (currMB);

      cost  = GetSkipCostMB (currMB, lambda_factor[Q_PEL]);
      if (cost < min_mcost)
      {
        min_mcost = cost;
        *mv = currSlice->all_mv [0][0][0][0][0];
      }
    }
  }
```

```
  //===========================================
  //=====   SET MV'S AND RETURN MOTION COST   =====
  //===========================================

  // Set first line
  for (i=block_x; i < block_x + (bsx>>2); i++)
  {
    all_mv[0][i] = *mv;
  }

  // set all other lines
  for (j=1; j < (bsy>>2); j++)
  {
    memcpy(&all_mv[j][block_x], &all_mv[0][block_x], (bsx>>2) *
sizeof(MotionVector));
  }


  // Bipred ME consideration: returns minimum bipred cost
  if (is_bipred_enabled(p_Vid, blocktype) && (ref == 0))
  {
    BiPredBlockMotionSearch(currMB, mv_block, &pred, mb_x, mb_y,
lambda_factor);
  }

  return min_mcost;
}


/*!

************************************************************************
******
 * \brief
 *    Bi-predictive motion search

************************************************************************
******
 */
static distblk BiPredBlockMotionSearch(Macroblock   *currMB,
//!< Current Macroblock
                                       MEBlock     *mv_block,
                                       MotionVector *pred_mv,    //!<
current list motion vector predictor
                                       int          mb_x,          //!<
x-coordinate inside macroblock
                                       int          mb_y,          //!<
y-coordinate inside macroblock
```

```
                                    int*        lambda_factor)   //!<
lagrangian parameter for determining motion cost
{
  VideoParameters *p_Vid    = currMB->p_Vid;
  InputParameters *p_Inp    = currMB->p_Inp;
  Slice         *currSlice = currMB->p_Slice;
  int        list = mv_block->list;
  int        i, j;
  short      bipred_type = list ? 0 : 1;
  MotionVector ***** bipred_mv = currSlice->bipred_mv[bipred_type];
  distblk    min_mcostbi = DISTBLK_MAX;
  MotionVector *mv = &mv_block->mv[list];
  MotionVector bimv, tempmv;
  MotionVector pred_mv1, pred_mv2, pred_bi;
  MotionVector *bi_mv1 = NULL, *bi_mv2 = NULL;
  short      iterlist = (short) list;
  int        block_x   = (mb_x>>2);
  int        block_y   = (mb_y>>2);
  int        blocktype = mv_block->blocktype;
  int        bsx      = mv_block->blocksize_x;
  int        bsy      = mv_block->blocksize_y;
  //PixelPos   block[4];  // neighbor blocks

  //get_neighbors(currMB, mv_block->block, mb_x, mb_y, bsx);

  if (p_Inp->SearchMode[p_Vid->view_id] == UM_HEX)
  {
    p_Vid->p_UMHex->bipred_flag = 1;
    UMHEXSetMotionVectorPredictor(currMB, &pred_bi,
p_Vid->enc_picture->mv_info, 0, list ^ 1, mb_x, mb_y, bsx, bsy,
mv_block);
  }
  else
    currMB->GetMVPredictor (currMB, mv_block->block, &pred_bi, 0,
p_Vid->enc_picture->mv_info, list ^ 1, mb_x, mb_y, bsx, bsy);

  if ((p_Inp->SearchMode[p_Vid->view_id] != EPZS) ||
(p_Inp->EPZSSubPelGrid == 0))
  {
    mv->mv_x = ((mv->mv_x  + 2) >> 2) * 4;
    mv->mv_y = ((mv->mv_y  + 2) >> 2) * 4;
    bimv.mv_x = ((pred_bi.mv_x  + 2) >> 2) * 4;
    bimv.mv_y = ((pred_bi.mv_y  + 2) >> 2) * 4;
  }
  else
  {
    bimv = pred_bi;
  }

  //Bi-predictive motion Refinements
```

```
  for (mv_block->iteration_no = 0; mv_block->iteration_no <=
p_Inp->BiPredMERefinements; mv_block->iteration_no++)
  {
    if (mv_block->iteration_no & 0x01)
    {
      pred_mv1  = *pred_mv;
      pred_mv2  = pred_bi;
      bi_mv1    = mv;
      bi_mv2    = &bimv;
      iterlist  = (short) list;
    }
    else
    {
      pred_mv1  = pred_bi;
      pred_mv2  = *pred_mv;
      bi_mv1    = &bimv;
      bi_mv2    = mv;
      iterlist = (short) (list ^ 1);
    }

    tempmv = *bi_mv1;

    PrepareBiPredMEParams(currSlice, mv_block,
mv_block->ChromaMEEnable, iterlist, currMB->list_offset,
mv_block->ref_idx);
    // Get bipred mvs for list iterlist given previously computed mvs
from other list
    min_mcostbi = currMB->BiPredME (currMB, iterlist,
      &pred_mv1, &pred_mv2, bi_mv1, bi_mv2, mv_block,
      (p_Inp->BiPredMESearchRange[p_Vid->view_id]
<<2)>>mv_block->iteration_no, min_mcostbi, lambda_factor[F_PEL]);

    if (mv_block->iteration_no > 0 && (tempmv.mv_x == bi_mv1->mv_x)
&& (tempmv.mv_y == bi_mv1->mv_y))
    {
      break;
    }
  }

  if (!p_Inp->DisableSubpelME[p_Vid->view_id])
  {
    if (p_Inp->BiPredMESubPel)
    {
      if ( !p_Vid->start_me_refinement_hp )
        min_mcostbi = DISTBLK_MAX;
      PrepareBiPredMEParams(currSlice, mv_block,
mv_block->ChromaMEEnable, iterlist, currMB->list_offset,
mv_block->ref_idx);
```

```
    min_mcostbi =  currMB->SubPelBiPredME (currMB, mv_block,
iterlist, &pred_mv1, &pred_mv2, bi_mv1, bi_mv2, min_mcostbi,
lambda_factor);
   }

   if (p_Inp->BiPredMESubPel==2)
   {
     if ( !p_Vid->start_me_refinement_qp )
       min_mcostbi = DISTBLK_MAX;
     PrepareBiPredMEParams(currSlice, mv_block,
mv_block->ChromaMEEnable, iterlist ^ 1, currMB->list_offset,
mv_block->ref_idx);

     min_mcostbi =  currMB->SubPelBiPredME (currMB, mv_block,
iterlist ^ 1, &pred_mv2, &pred_mv1, bi_mv2, bi_mv1, min_mcostbi,
lambda_factor);
   }
  }

  clip_mv_range(p_Vid, 0, bi_mv1, Q_PEL);
  clip_mv_range(p_Vid, 0, bi_mv2, Q_PEL);

  for (j=block_y; j < block_y + (bsy>>2); j++)
  {
    for (i=block_x ; i < block_x + (bsx>>2); i++)
    {
      bipred_mv[iterlist   ][(short)
mv_block->ref_idx][blocktype][j][i] = *bi_mv1;
     bipred_mv[iterlist ^ 1][(short)
mv_block->ref_idx][blocktype][j][i] = *bi_mv2;
   }
  }


  return min_mcostbi;
}

/*!

************************************************************************
******
 * \brief
 *   Motion Cost for Bidirectional modes

************************************************************************
******
 */
distblk BIDPartitionCost (Macroblock *currMB,
                  int   blocktype,
                  int   block8x8,
```

```
                  char  cur_ref[2],
                  int   lambda_factor)
{
  VideoParameters *p_Vid = currMB->p_Vid;
  Slice *currSlice = currMB->p_Slice;
  DecodedPictureBuffer *p_Dpb =
p_Vid->p_Dpb_layer[p_Vid->dpb_layer_id];
  imgpel **cur_img = p_Vid->pCurImg;

  short pic_pix_x, pic_pix_y;
  int   v, h;
  distblk mcost;

  int   mvd_bits  = 0;

  int   parttype  = (blocktype < 4 ? blocktype : 4);
  int   step_h0   = block_size[ parttype][0];
  int   step_v0   = block_size[ parttype][1];
  int   step_h    = block_size[blocktype][0];
  int   step_v    = block_size[blocktype][1];
  int   bx       = bx0[parttype][block8x8] << 2;
  int   by       = by0[parttype][block8x8] << 2;
  short block_size_x = block_size[blocktype][0]; // this is the same
as step_h and could be removed
  short block_size_y = block_size[blocktype][1]; // this is the same
as step_v and could be removed

  MotionVector **all_mv_l0 = currSlice->all_mv [LIST_0][(int)
cur_ref[LIST_0]][blocktype];
  MotionVector **all_mv_l1 = currSlice->all_mv [LIST_1][(int)
cur_ref[LIST_1]][blocktype];
  short bipred_me =  0; //no bipred for this case
  imgpel  **mb_pred = currSlice->mb_pred[0];
  int   list_mode[2];
  list_mode[0] = blocktype;
  list_mode[1] = blocktype;

  //----- cost for motion vector bits -----
  // Should write a separate, small function to do this processing
  // List0
  mvd_bits = mv_bit_cost(currMB, all_mv_l0, LIST_0, cur_ref[LIST_0],
by, bx, step_v0, step_v, step_h0, step_h, mvd_bits);
  // List1
  mvd_bits = mv_bit_cost(currMB, all_mv_l1, LIST_1, cur_ref[LIST_1],
by, bx, step_v0, step_v, step_h0, step_h, mvd_bits);

  mcost = weighted_cost (lambda_factor, mvd_bits);

  // Get prediction
  for (v = by; v < by + step_v0; v += block_size_y)
```

```
  {
    for (h = bx; h < bx + step_h0; h += block_size_x)
    {
      p_Dpb->pf_luma_prediction (currMB, h, v, block_size_x,
block_size_y, 2, list_mode, cur_ref, bipred_me);
    }
  }

  //----- cost of residual signal -----
  if ((!currSlice->p_Inp->Transform8x8Mode) || (blocktype>4))
  {
    short diff16[16];
    short *diff;

    pic_pix_y = (short) currMB->opix_y;
    pic_pix_x = (short) currMB->pix_x;
    for (v= by; v < by + step_v0; v += BLOCK_SIZE)
    {
      for (h = bx; h < bx + step_h0; h += BLOCK_SIZE)
      {
       diff = diff16;
       calcDifference(cur_img, pic_pix_x+h, pic_pix_y+v, mb_pred,
h,v, 4, 4, diff);
       mcost += p_Vid->distortion4x4 (diff16, DISTBLK_MAX);
      }
    }
  }
  else
  {
    short diff64[64];
    short *diff;

    pic_pix_y = (short) currMB->opix_y;
    pic_pix_x = (short) currMB->pix_x;
      for (v= by; v < by + step_v0; v += BLOCK_SIZE_8x8)
      {
        for (h = bx; h < bx + step_h0; h += BLOCK_SIZE_8x8)
        {
          diff = diff64;
          calcDifference(cur_img, pic_pix_x+h, pic_pix_y+v, mb_pred,
h, v, 8, 8, diff);
          mcost += p_Vid->distortion8x8(diff64, DISTBLK_MAX);
        }
      }
  }

  return mcost;
}

/*!
```

```
************************************************************************
*******
 * \brief
 *    Get cost for skip mode for an macroblock

************************************************************************
*******
 */
static distblk GetSkipCostMB (Macroblock *currMB, int lambda)
{
  Slice *currSlice = currMB->p_Slice;
  VideoParameters *p_Vid = currMB->p_Vid;
  InputParameters *p_Inp = currMB->p_Inp;
  DecodedPictureBuffer *p_Dpb =
p_Vid->p_Dpb_layer[p_Vid->dpb_layer_id];
  distblk cost = 0;

  int block;
  imgpel  **mb_pred = currSlice->mb_pred[0];
  char  cur_ref[2] = {0, 0};
  int   list_mode[2] = {0, 0};

  //===== prediction of 16x16 skip block =====
  p_Dpb->pf_luma_prediction (currMB, 0, 0, MB_BLOCK_SIZE,
MB_BLOCK_SIZE, 0, list_mode, cur_ref, 0);

  if (p_Inp->Transform8x8Mode == 0)
  {
    short diff16[16];
    short *diff;
    int block_y, block_x;//, i, j;
    int mb_x, mb_y;

    int pic_pix_y = currMB->opix_y;
    int pic_pix_x = currMB->pix_x;

    for(block = 0;block < 4; block++)
    {
      mb_y = (block >>  1)<<3;
      mb_x = (block & 0x01)<<3;
      for (block_y = mb_y; block_y < mb_y + 8; block_y += 4)
      {
        for (block_x = mb_x; block_x < mb_x + 8; block_x += 4)
        {
          diff = diff16;
          //===== get displaced frame difference ======
          calcDifference(p_Vid->pCurImg, pic_pix_x+block_x,
pic_pix_y+block_y, mb_pred, block_x, block_y, 4, 4, diff);
          cost += p_Vid->distortion4x4 (diff16, DISTBLK_MAX);
```

```
      }
    }
  }
  else
  {
    short diff64[64];
    short *diff;
    //int i, j;
    int mb_x, mb_y;

    int pic_pix_y = currMB->opix_y;
    int pic_pix_x = currMB->pix_x;

    for(block = 0;block < 4;block++)
    {
      mb_y = (block >>  1)<<3;
      mb_x = (block & 0x01)<<3;

      //===== get displaced frame difference ======
      diff = diff64;
      calcDifference(p_Vid->pCurImg, pic_pix_x+mb_x, pic_pix_y+mb_y,
mb_pred, mb_x, mb_y, 8, 8, diff);
      cost += p_Vid->distortion8x8 (diff64, DISTBLK_MAX);
    }
  }

  //cost -= ((lambda_factor[Q_PEL] + 4096) >> 13);
  cost -= weight_cost(lambda, 8);

  return cost;
}

/*!

************************************************************************
*******
 * \brief
 *    Find motion vector for the Skip mode

************************************************************************
*******
 */
void FindSkipModeMotionVector (Macroblock *currMB)
{
  Slice *currSlice = currMB->p_Slice;
  VideoParameters *p_Vid = currMB->p_Vid;
  PicMotionParams **motion = p_Vid->enc_picture->mv_info;
  int   bx, by;
  MotionVector **all_mv = currSlice->all_mv[0][0][0];
```

```
  MotionVector pmv;

  int zeroMotionAbove;
  int zeroMotionLeft;
  PixelPos mb[4];
  int     a_mv_y = 0;
  int     a_ref_idx = 0;
  int     b_mv_y = 0;
  int     b_ref_idx = 0;

  get_neighbors(currMB, mb, 0, 0, 16);

  if (mb[0].available)
  {
    a_mv_y   = motion[mb[0].pos_y][mb[0].pos_x].mv[LIST_0].mv_y;
    a_ref_idx = motion[mb[0].pos_y][mb[0].pos_x].ref_idx[LIST_0];

    if (currMB->mb_field
&& !p_Vid->mb_data[mb[0].mb_addr].mb_field)
    {
      a_mv_y   /=2;
      a_ref_idx *=2;
    }
    if (!currMB->mb_field &&
p_Vid->mb_data[mb[0].mb_addr].mb_field)
    {
      a_mv_y   *= 2;
      a_ref_idx >>=1;
    }
  }

  if (mb[1].available)
  {
    b_mv_y   = motion[mb[1].pos_y][mb[1].pos_x].mv[LIST_0].mv_y;
    b_ref_idx = motion[mb[1].pos_y][mb[1].pos_x].ref_idx[LIST_0];

    if (currMB->mb_field
&& !p_Vid->mb_data[mb[1].mb_addr].mb_field)
    {
      b_mv_y   /=2;
      b_ref_idx *=2;
    }
    if (!currMB->mb_field &&
p_Vid->mb_data[mb[1].mb_addr].mb_field)
    {
      b_mv_y   *=2;
      b_ref_idx >>=1;
    }
  }
```

```
  zeroMotionLeft  = !mb[0].available ? 1 : a_ref_idx==0 &&
motion[mb[0].pos_y][mb[0].pos_x].mv[LIST_0].mv_x ==0 && a_mv_y==0 ?
1 : 0;
  zeroMotionAbove = !mb[1].available ? 1 : b_ref_idx==0 &&
motion[mb[1].pos_y][mb[1].pos_x].mv[LIST_0].mv_x ==0 && b_mv_y==0 ?
1 : 0;

  if (zeroMotionAbove || zeroMotionLeft)
  {
    memset(&all_mv [0][0], 0, 16 * sizeof(MotionVector)); // 4 * 4
  }
  else
  {
    currMB->GetMVPredictor (currMB, mb, &pmv, 0, motion, LIST_0, 0,
0, 16, 16);

    for (by = 0;by < 4;by++)
    for (bx = 0;bx < 4;bx++)
    {
      all_mv [by][bx] = pmv;
    }
  }
}

/*!

************************************************************************
*******
 * \brief
 *    Get cost for direct mode for an 8x8 block

************************************************************************
*******
 */
distblk GetDirectCost8x8 (Macroblock *currMB, int block, distblk
*cost8x8)
{
  Slice *currSlice = currMB->p_Slice;
  VideoParameters *p_Vid = currMB->p_Vid;
  InputParameters *p_Inp = currMB->p_Inp;
  DecodedPictureBuffer *p_Dpb =
p_Vid->p_Dpb_layer[p_Vid->dpb_layer_id];
  int pic_pix_y, pic_pix_x, i, j;

  distblk cost  = 0;
  int mb_y  = (block >> 1)<<3;
  int mb_x  = (block & 0x01)<<3;
  imgpel  **mb_pred = currSlice->mb_pred[0];
  int   list_mode[2] = {0, 0};
```

```
  // Check if valid
  for (j=(currMB->opix_y + mb_y) >> 2; j < (currMB->opix_y + mb_y +
8) >> 2; j++)
  {
    for (i=(currMB->pix_x + mb_x) >> 2; i < (currMB->pix_x + mb_x +
8) >> 2; i++)
    {
      if (currSlice->direct_pdir[j][i] < 0)
      {
        *cost8x8 = DISTBLK_MAX;
        return DISTBLK_MAX; //mode not allowed
      }
    }
  }

  //===== Generate direct prediction =====
  for (j = mb_y; j < mb_y + 8; j += 4)
  {
    pic_pix_y = (currMB->opix_y + j) >> 2;
    for (i = mb_x; i < mb_x + 8; i += 4)
    {
      pic_pix_x = (currMB->pix_x + i) >> 2;
      p_Dpb->pf_luma_prediction (currMB, i, j, 4, 4,
currSlice->direct_pdir[pic_pix_y][pic_pix_x],
        list_mode, currSlice->direct_ref_idx[pic_pix_y][pic_pix_x],
0);
    }
  }

  if(p_Inp->Transform8x8Mode)
  {
    short diff16[4][16];
    short diff64[64];
    short *tmp64 = diff64;
    short *tmp16[4]; //{diff16[0], diff16[1], diff16[2], diff16[3]};
    int index;

    tmp16[0] = diff16[0];
    tmp16[1] = diff16[1];
    tmp16[2] = diff16[2];
    tmp16[3] = diff16[3];

    pic_pix_y = currMB->opix_y;
    pic_pix_x = currMB->pix_x;
    //===== get displaced frame difference ======
    //p_Dpb->pf_calcDifference(p_Vid->pCurImg, pic_pix_x+mb_x,
pic_pix_y+mb_y, mb_pred, mb_x, mb_y, 8, 8, tmp64);
    for (j = mb_y; j < 8 + mb_y; j++)
    {
```

```
      for (i = mb_x; i < 8 + mb_x; i++)
      {
        index = 2 * ((j - mb_y)> 3) + ((i - mb_x)> 3);
        *tmp64++ = *(tmp16[index])++ = (short)
(p_Vid->pCurImg[pic_pix_y + j][pic_pix_x + i] - mb_pred[j][i]);
      }
    }

    cost += p_Vid->distortion4x4 (diff16[0], DISTBLK_MAX);
    cost += p_Vid->distortion4x4 (diff16[1], DISTBLK_MAX);
    cost += p_Vid->distortion4x4 (diff16[2], DISTBLK_MAX);
    cost += p_Vid->distortion4x4 (diff16[3], DISTBLK_MAX);
    *cost8x8 += p_Vid->distortion8x8 (diff64, DISTBLK_MAX);
  }
  else
  {
    int block_y, block_x;
    short diff16[16];
    short *diff;

    for (block_y=mb_y; block_y < mb_y + 8; block_y += 4)
    {
      pic_pix_y = currMB->opix_y + block_y;

      for (block_x=mb_x; block_x<mb_x+8; block_x+=4)
      {
        pic_pix_x = currMB->pix_x + block_x;
        diff = diff16;

        //===== get displaced frame difference ======
        calcDifference(p_Vid->pCurImg, pic_pix_x, pic_pix_y, mb_pred,
block_x, block_y, 4, 4, diff);
        cost += p_Vid->distortion4x4 (diff16, DISTBLK_MAX);
      }
    }
  }

  return cost;
}



/*!

************************************************************************
*******
 * \brief
 *    Get cost for direct mode for an macroblock
```

```
************************************************************************
*******
 */
distblk GetDirectCostMB (Macroblock *currMB)
{
  Slice *currSlice = currMB->p_Slice;
  InputParameters *p_Inp = currSlice->p_Inp;
  int i;
  distblk cost = 0;
  distblk cost8x8 = 0;
  int bslice = currSlice->slice_type == B_SLICE;
#if (MVC_EXTENSION_ENABLE)
  int *InterSearch =
p_Inp->InterSearch[(currSlice->p_Vid->num_of_layers > 1) ?
currSlice->view_id : 0][bslice];
#else
  int *InterSearch = p_Inp->InterSearch[0][bslice];
#endif

  for (i=0; i<4; i++)
  {
    cost += GetDirectCost8x8 (currMB, i, &cost8x8);
    if (cost8x8 == DISTBLK_MAX) return DISTBLK_MAX;
  }

  switch(p_Inp->Transform8x8Mode)
  {
  case 1: // Mixture of 8x8 & 4x4 transform
    if((cost8x8 < cost)||
      !(InterSearch[5] &&
      InterSearch[6] &&
      InterSearch[7])
      )
    {
      cost = cost8x8; //return 8x8 cost
    }
    break;
  case 2: // 8x8 Transform only
    cost = cost8x8;
    break;
  default: // 4x4 Transform only
    break;
  }

  return cost;
}

/*!
```

```
************************************************************************
*******
 * \brief
 *    Motion search for a macroblock partition

************************************************************************
*******
 */
void PartitionMotionSearch (Macroblock *currMB,
                            int    blocktype,
                            int    block8x8,
                            int    *lambda_factor)
{
  VideoParameters *p_Vid = currMB->p_Vid;
  Slice *currSlice = currMB->p_Slice;

#if GET_METIME
  TIME_T me_time_start;
  TIME_T me_time_end;
  int64 me_tmp_time;
  gettime( &me_time_start );    // start time ms
#endif

  if (currSlice->rdoq_motion_copy == 1)
  {
    PicMotionParams **motion = p_Vid->enc_picture->mv_info;
    short by = by0[blocktype][block8x8];
    short bx = bx0[blocktype][block8x8];
    short step_h = (part_size[blocktype][0]);
    short step_v = (part_size[blocktype][1]);

    short pic_block_y = currMB->block_y + by;
    short pic_block_x = currMB->block_x + bx;
    int   list_offset = currMB->list_offset;
    int   numlists    = (currSlice->slice_type == B_SLICE) ? 2 : 1;
    distblk *m_cost;

    short list = LIST_0;
    short ref = 0;

    //===== LOOP OVER REFERENCE FRAMES =====
    for (list = 0; list < numlists; list++)
    {
      for (ref=0; ref < currSlice->listXsize[list+list_offset];
ref++)
      {
        m_cost =
&p_Vid->motion_cost[blocktype][list][ref][block8x8];
```

```
      //===== LOOP OVER SUB MACRO BLOCK partitions
      updateMV_mp(currMB, m_cost, ref, list, bx, by, blocktype,
block8x8);
      set_me_parameters(motion,
&currSlice->all_mv[list][ref][blocktype][by][bx], list, (char) ref,
step_h, step_v, pic_block_y, pic_block_x);
    }
   }
  }
  else
  {
   InputParameters *p_Inp = currMB->p_Inp;
   short by = by0[blocktype][block8x8];
   short bx = bx0[blocktype][block8x8];
   short step_h   = (part_size[blocktype][0]);
   short step_v   = (part_size[blocktype][1]);

   short pic_block_y = currMB->block_y + by;
   short pic_block_x = currMB->block_x + bx;
   int   list_offset = currMB->list_offset;
   int   numlists  = (currSlice->slice_type == B_SLICE) ? 2 : 1;
   short list = LIST_0;
   short ref = 0;
   MEBlock  mv_block;
   distblk *m_cost;


   PicMotionParams **motion = p_Vid->enc_picture->mv_info;

   // Set flag for 8x8 Hadamard consideration for SATD (only used
when 8x8 integer transform is used for encoding)
   mv_block.test8x8 = p_Inp->Transform8x8Mode;

   init_mv_block(currMB, &mv_block, (short) blocktype, list, (char)
ref, bx, by);

   if (p_Inp->SearchMode[p_Vid->view_id] == EPZS)
   {
    if (p_Inp->EPZSSubPelGrid)
     currMB->IntPelME = EPZS_integer_motion_estimation;
    else
     currMB->IntPelME = EPZS_motion_estimation;
   }

   get_original_block(p_Vid, &mv_block);

   //--- motion search for block ---
   {
    //===== LOOP OVER REFERENCE FRAMES =====
    for (list = 0; list < numlists; list++)
```

```
      {
        //----- set arrays -----
        mv_block.list = (char) list;
        for (ref=0; ref < currSlice->listXsize[list+list_offset];
ref++)
        {
            mv_block.ref_idx = (char) ref;
            m_cost =
&p_Vid->motion_cost[blocktype][list][ref][block8x8];

            {
              //----- set search range ---
              get_search_range(&mv_block, p_Inp, ref, blocktype);

              //===== LOOP OVER MACROBLOCK partitions
              *m_cost = BlockMotionSearch (currMB, &mv_block, bx<<2,
by<<2, lambda_factor);
            }
            //--- set motion vectors and reference frame ---
            set_me_parameters(motion,
&currSlice->all_mv[list][ref][blocktype][by][bx], list, (char) ref,
step_h, step_v, pic_block_y, pic_block_x);
        }
      }
    }

    free_mv_block(&mv_block);
  }

#if GET_METIME
  gettime(&me_time_end);   // end time ms
  me_tmp_time = timediff (&me_time_start, &me_time_end);
  p_Vid->me_tot_time += me_tmp_time;
  p_Vid->me_time += me_tmp_time;
#endif
}

/*!

************************************************************************
*******
 * \brief
 *    Motion search for a submacroblock partition

************************************************************************
*******
 */
void SubPartitionMotionSearch (Macroblock *currMB,
                               int   blocktype,
                               int   block8x8,
```

```
                          int    *lambda_factor)
{
  Slice *currSlice = currMB->p_Slice;
  VideoParameters *p_Vid = currMB->p_Vid;

#if GET_METIME
  TIME_T me_time_start;
  TIME_T me_time_end;
  int64 me_tmp_time;
  gettime( &me_time_start );    // start time ms
#endif

  if (currSlice->rdoq_motion_copy == 1)
  {
    int   parttype  = 4;
    PicMotionParams **motion = p_Vid->enc_picture->mv_info;
    short by = by0[parttype][block8x8];
    short bx = bx0[parttype][block8x8];
    short step_h    = (part_size[blocktype][0]);
    short step_v    = (part_size[blocktype][1]);
    int   list_offset = currMB->list_offset;
    int   numlists  = (currSlice->slice_type == B_SLICE) ? 2 : 1;
    distblk *m_cost;
    MotionVector *all_mv;
    short list = LIST_0;
    short ref = 0;

    short step_h0   = (part_size[ parttype][0]);
    short step_v0   = (part_size[ parttype][1]);

    int   v, h;
    int   pic_block_y;

    //===== LOOP OVER REFERENCE FRAMES =====
    for (list = 0; list < numlists; list++)
    {
      for (ref=0; ref < currSlice->listXsize[list+list_offset];
ref++)
      {
        m_cost =
&p_Vid->motion_cost[blocktype][list][ref][block8x8];

        //===== LOOP OVER SUB MACRO BLOCK partitions
        for (v=by; v<by + step_v0; v += step_v)
        {
          pic_block_y = currMB->block_y + v;
          for (h=bx; h<bx+step_h0; h+=step_h)
          {
            all_mv = &currSlice->all_mv[list][ref][blocktype][v][h];
```

```
          updateMV_mp(currMB, m_cost, ref, list, h, v, blocktype,
block8x8);

          //--- set motion vectors and reference frame (for motion
vector prediction) ---
          set_me_parameters(motion, all_mv, list, (char) ref, step_h,
step_v, pic_block_y, currMB->block_x + h);
        } // h
      } // v
    }
  }
 }
 else
 {
  InputParameters *p_Inp = currMB->p_Inp;
  PicMotionParams **motion = p_Vid->enc_picture->mv_info;
  int   parttype  = 4;
  short by = by0[parttype][block8x8];
  short bx = bx0[parttype][block8x8];
  short step_h0   = (part_size[ parttype][0]);
  short step_v0   = (part_size[ parttype][1]);
  short step_h    = (part_size[blocktype][0]);
  short step_v    = (part_size[blocktype][1]);
  int   list_offset = currMB->list_offset;
  int   numlists  = (currSlice->slice_type == B_SLICE) ? 2 : 1;
  MotionVector *all_mv;
  short list = LIST_0;
  short ref = 0;
  MEBlock  mv_block;
  distblk *m_cost;
  distblk  mcost;
  int   v, h;
  int   pic_block_y;

  // Set if 8x8 transform will be used if SATD is used
  mv_block.test8x8 = p_Inp->Transform8x8Mode && blocktype == 4;

  if (p_Inp->SearchMode[p_Vid->view_id] == EPZS)
  {
    if (p_Inp->EPZSSubPelGrid)
    {
      if (blocktype > 4)
        currMB->IntPelME = EPZS_integer_subMB_motion_estimation;
      else
        currMB->IntPelME = EPZS_integer_motion_estimation;
    }
    else
    {
      if (blocktype > 4)
```

```
        currMB->IntPelME = EPZS_subMB_motion_estimation;
      else
        currMB->IntPelME = EPZS_motion_estimation;
    }
  }

  init_mv_block(currMB, &mv_block, (short) blocktype, list, (char)
ref, bx, by);

  if (blocktype == 4)
    get_original_block(p_Vid, &mv_block);

  //===== LOOP OVER REFERENCE FRAMES =====
  for (list=0; list<numlists;list++)
  {
    mv_block.list = (char) list;
    for (ref=0; ref < currSlice->listXsize[list+list_offset];
ref++)
    {
      mv_block.ref_idx = (char) ref;
      m_cost =
&p_Vid->motion_cost[blocktype][list][ref][block8x8];
      //----- set search range ---
      get_search_range(&mv_block, p_Inp, ref, blocktype);

      //----- init motion cost -----
      *m_cost = 0;

      //===== LOOP OVER SUB MACRO BLOCK partitions
      for (v=by; v<by + step_v0; v += step_v)
      {
        pic_block_y = currMB->block_y + v;

        for (h=bx; h<bx+step_h0; h+=step_h)
        {
          all_mv =
&currSlice->all_mv[list][ref][blocktype][v][h];

          //--- motion search for block ---
          update_mv_block(currMB, &mv_block, h, v);
          {
            //----- set search range ---
            get_search_range(&mv_block, p_Inp, ref, blocktype);

            mcost = BlockMotionSearch (currMB, &mv_block, h<<2,
v<<2, lambda_factor);

            *m_cost += mcost;
          }
```

```
            //--- set motion vectors and reference frame (for motion
vector prediction) ---
            set_me_parameters(motion, all_mv, list, (char) ref,
step_h, step_v, pic_block_y, currMB->block_x + h);
          }
        }

        if ((p_Inp->Transform8x8Mode == 1) && p_Inp->RDOQ_CP_MV &&
(blocktype == 4))
        {
          if (currMB->luma_transform_size_8x8_flag)
          {
            currSlice->tmp_mv8[list][ref][by][bx] =
currSlice->all_mv[list][ref][blocktype][by][bx];
            currSlice->motion_cost8[list][ref][block8x8] =
*m_cost;
          }
          else
          {
            currSlice->tmp_mv4[list][ref][by][bx] =
currSlice->all_mv[list][ref][blocktype][by][bx];
            currSlice->motion_cost4[list][ref][block8x8] =
*m_cost;
          }
        }
     }

   }

   free_mv_block(&mv_block);
  }

#if GET_METIME
  gettime(&me_time_end);   // end time ms
  me_tmp_time = timediff (&me_time_start, &me_time_end);
  p_Vid->me_tot_time += me_tmp_time;
  p_Vid->me_time += me_tmp_time;
#endif
}



/*!

************************************************************************
**********
 * \file md_high.c
 *
 * \brief
 *   Main macroblock mode decision functions and helpers
```

```
 *
*************************************************************************
*********
 */

#include <math.h>
#include <limits.h>
#include <float.h>

#include "global.h"
#include "rdopt_coding_state.h"
#include "intrarefresh.h"
#include "image.h"
#include "ratectl.h"
#include "mode_decision.h"
#include "mode_decision_p8x8.h"
#include "fmo.h"
#include "me_umhex.h"
#include "me_umhexsmp.h"
#include "macroblock.h"
#include "md_common.h"
#include "conformance.h"
#include "vlc.h"
#include "rdopt.h"
#include "mv_search.h"

/*************************************************************************
*************************************************************/
/*************************************************************************
*************************************************************/
#include "me_fullfast.h"
extern int block_total_number = 0;
extern long long frame_total_SAD = 0;
/*************************************************************************
*************************************************************/
/*************************************************************************
*************************************************************/


/*!
*************************************************************************
*********************
* \brief
*   Mode Decision for a macroblock
*************************************************************************
*********************
*/
void encode_one_macroblock_high (Macroblock *currMB)
{
```

```
  Slice *currSlice = currMB->p_Slice;
  VideoParameters *p_Vid = currMB->p_Vid;
  InputParameters *p_Inp = currMB->p_Inp;
  PicMotionParams **motion = p_Vid->enc_picture->mv_info;
  RDOPTStructure  *p_RDO = currSlice->p_RDO;

  int        max_index = 9;
  int        block, index, mode, i, j;
  RD_PARAMS   enc_mb;
  distblk    bmcost[5] = {DISTBLK_MAX};
  distblk    cost=0;
  distblk    min_cost = DISTBLK_MAX;
  int        intra1 = 0;
  int        mb_available[3];

  short      bslice     = (short) (currSlice->slice_type ==
B_SLICE);
  short      pslice     = (short) ((currSlice->slice_type == P_SLICE)
|| (currSlice->slice_type == SP_SLICE));
  short      intra      = (short) ((currSlice->slice_type == I_SLICE)
|| (currSlice->slice_type == SI_SLICE) || (pslice && currMB->mb_y ==
p_Vid->mb_y_upd && p_Vid->mb_y_upd != p_Vid->mb_y_intra));
  int        lambda_mf[3];

  imgpel   **mb_pred  = currSlice->mb_pred[0];
  Block8x8Info *b8x8info = p_Vid->b8x8info;

  char       chroma_pred_mode_range[2];
  short      inter_skip = 0;
  BestMode    md_best;
  Info8x8     best;


/*****************************************************************
*********************************************/

/*****************************************************************
*********************************************/
    int current_block_SAD = 0;
    int skip_all_intra_mode = 0; // if  it = 1, skip all intra modes,
else do intra modes
    int total_block_number = p_Vid->p_Inp->source.width[0]/16 *
p_Vid->p_Inp->source.height[0]/16;
    int me_choice = 0; // if me_choice = 1, do ME only for 16x16, 16x8,
8x16 blocks, else do ME for all blocks
*****************************************************************
*********************************************/

/*****************************************************************
*********************************************/
```

```
  init_md_best(&md_best);

  // Init best (need to create simple function)
  best.pdir = 0;
  best.bipred = 0;
  best.ref[LIST_0] = 0;
  best.ref[LIST_1] = -1;

  intra |= RandomIntra (p_Vid, currMB->mbAddrX);   // Forced
Pseudo-Random Intra

  //===== Setup Macroblock encoding parameters =====
  init_enc_mb_params(currMB, &enc_mb, intra);
  if (p_Inp->AdaptiveRounding)
  {
    reset_adaptive_rounding(p_Vid);
  }

  if (currSlice->mb_aff_frame_flag)
  {
    reset_mb_nz_coeff(p_Vid, currMB->mbAddrX);
  }

  //=====   S T O R E   C O D I N G   S T A T E   =====
  //---------------------------------------------------
  currSlice->store_coding_state (currMB, currSlice->p_RDO->cs_cm);

  if (!intra)
  {
    //===== set skip/direct motion vectors =====
    if (enc_mb.valid[0])
    {
      if (bslice)
        currSlice->Get_Direct_Motion_Vectors (currMB);
      else
        FindSkipModeMotionVector (currMB);
    }
    if (p_Inp->CtxAdptLagrangeMult == 1)
    {
      get_initial_mb16x16_cost(currMB);
    }


/*************************************************************************
*************************************************/

/*************************************************************************
*************************************************/
```

```
      current_block_SAD =
p_Vid->p_ffast_me->BlockSAD[0][0][1][0][0];
      // compute w value for different QP
      double w_value;
      switch (p_Vid->p_Inp->qp[0]) {
         case 28:
            w_value = 1.2;
            break;
         case 32:
            w_value = 1.0;
            break;
         case 36:
            w_value = 0.8;
            break;
         case 40:
            w_value = 0.6;
            break;

         default:
            w_value = 1.0;
            break;
      }
//      printf("num of macroblocks in a silce
= %d\n",currSlice->num_mb);
//      printf("SAD of current Mb = %d\n",current_block_SAD);
//      printf("block_total_num = %d\n",block_total_number);

      frame_total_SAD += current_block_SAD;

//      printf("frame_total SAD = %lld\n",frame_total_SAD);

      if (block_total_number != 0) {
         if (frame_total_SAD/block_total_number >= w_value*
current_block_SAD) {
            me_choice = 1;
         }else{
            me_choice = 0;
         }
      }

/*************************************************************************
*****************************************/

/*************************************************************************
*****************************************/

    //===== MOTION ESTIMATION FOR 16x16, 16x8, 8x16 BLOCKS =====
    for (mode = 1; mode < 4; mode++)
    {
      best.mode = (char) mode;
```

```
      best.bipred = 0;
      b8x8info->best[mode][0].bipred = 0;

      if (enc_mb.valid[mode])
      {
        for (cost=0, block=0; block<(mode==1?1:2); block++)
        {
          update_lambda_costs(currMB, &enc_mb, lambda_mf);
          PartitionMotionSearch (currMB, mode, block, lambda_mf);

          //--- set 4x4 block indices (for getting MV) ---
          j = (block==1 && mode==2 ? 2 : 0);
          i = (block==1 && mode==3 ? 2 : 0);

          //--- get cost and reference frame for List 0 prediction ---
          bmcost[LIST_0] = DISTBLK_MAX;
          list_prediction_cost(currMB, LIST_0, block, mode, &enc_mb,
bmcost, best.ref);

          if (bslice)
          {
            //--- get cost and reference frame for List 1 prediction
---
            bmcost[LIST_1] = DISTBLK_MAX;
            list_prediction_cost(currMB, LIST_1, block, mode, &enc_mb,
bmcost, best.ref);

            // Compute bipredictive cost between best list 0 and best
list 1 references
            list_prediction_cost(currMB, BI_PRED, block, mode,
&enc_mb, bmcost, best.ref);

            // currently Bi predictive ME is only supported for modes
1, 2, 3 and ref 0
            if (is_bipred_enabled(p_Vid, mode))
            {
              get_bipred_cost(currMB, mode, block, i, j, &best, &enc_mb,
bmcost);
            }
            else
            {
              bmcost[BI_PRED_L0] = DISTBLK_MAX;
              bmcost[BI_PRED_L1] = DISTBLK_MAX;
            }

            // Determine prediction list based on mode cost
            determine_prediction_list(bmcost, &best, &cost);
          }
          else // if (bslice)
          {
```

```
          best.pdir = 0;
          cost     += bmcost[LIST_0];
        }

        assign_enc_picture_params(currMB, mode, &best, 2 * block);

        //----- set reference frame and direction parameters -----
        set_block8x8_info(b8x8info, mode, block, &best);

        //--- set reference frames and motion vectors ---
        if (mode>1 && block == 0)
          currSlice->set_ref_and_motion_vectors (currMB, motion,
&best, block);
      } // for (block=0; block<(mode==1?1:2); block++)
      if (cost < min_cost)
      {
        md_best.mode = (byte) mode;
        md_best.cost = cost;
        currMB->best_mode = (short) mode;
        min_cost  = cost;
        if (p_Inp->CtxAdptLagrangeMult == 1)
        {
          adjust_mb16x16_cost(currMB, cost);
        }
      }
    } // if (enc_mb.valid[mode])
  } // for (mode=1; mode<4; mode++)


/*************************************************************************
*************************************/

/*************************************************************************
*************************************/
  if (me_choice == 0)
  {

/*************************************************************************
*************************************/

/*************************************************************************
*************************************/
      if (enc_mb.valid[P8x8])
      {
        currMB->valid_8x8 = FALSE;

        if (p_Inp->Transform8x8Mode)
        {
          ResetRD8x8Data(p_Vid, p_RDO->tr8x8);
```

```
          currMB->luma_transform_size_8x8_flag = TRUE; //switch to
8x8 transform size

//=============================================================
          // Check 8x8 partition with transform size 8x8

//=============================================================
          //=====  LOOP OVER 8x8 SUB-PARTITIONS  (Motion Estimation
& Mode Decision) =====
          for (block = 0; block < 4; block++)
          {
            currSlice->submacroblock_mode_decision(currMB, &enc_mb,
p_RDO->tr8x8, p_RDO->cofAC8x8ts[block], block, &cost);
            if(!currMB->valid_8x8)
              break;
            set_subblock8x8_info(b8x8info, P8x8, block,
p_RDO->tr8x8);
          }

        }// if (p_Inp->Transform8x8Mode)

        currMB->valid_4x4 = FALSE;
        if (p_Inp->Transform8x8Mode != 2)
        {
          currMB->luma_transform_size_8x8_flag = FALSE; //switch to
8x8 transform size
          ResetRD8x8Data(p_Vid, p_RDO->tr4x4);

//=============================================================
==
          // Check 8x8, 8x4, 4x8 and 4x4 partitions with transform
size 4x4

//=============================================================
==
          //=====  LOOP OVER 8x8 SUB-PARTITIONS  (Motion Estimation
& Mode Decision) =====
          for (block = 0; block < 4; block++)
          {
            currSlice->submacroblock_mode_decision(currMB, &enc_mb,
p_RDO->tr4x4, p_RDO->coefAC8x8[block], block, &cost);
            if(!currMB->valid_4x4)
              break;
            set_subblock8x8_info(b8x8info, P8x8, block,
p_RDO->tr4x4);
          }
        }// if (p_Inp->Transform8x8Mode != 2)

        if (p_Inp->RCEnable)
```

```
          rc_store_diff(currSlice->diffy,
&p_Vid->pCurImg[currMB->opix_y], currMB->pix_x, mb_pred);

        p_Vid->giRDOpt_B8OnlyFlag = FALSE;
      }

/*************************************************************************
*************************************/

/*************************************************************************
*************************************/
    }

/*************************************************************************
*************************************/

/*************************************************************************
*************************************/
  }
  else // if (!intra)
  {
    min_cost = DISTBLK_MAX;
  }

  // Set Chroma mode
  set_chroma_pred_mode(currMB, enc_mb, mb_available,
chroma_pred_mode_range);

  //========= C H O O S E   B E S T   M A C R O B L O C K   M O D E =========

//-----------------------------------------------------------------
----------

//    printf("curSAD
= %d\n",p_Vid->p_ffast_me->BlockSAD[0][0][1][0][0]);
  for (currMB->c_ipred_mode = chroma_pred_mode_range[0];
currMB->c_ipred_mode<=chroma_pred_mode_range[1];
currMB->c_ipred_mode++)
  {
    // bypass if c_ipred_mode is not allowed
    if ( (p_Vid->yuv_format != YUV400) &&
      (  ((!intra || !p_Inp->IntraDisableInterOnly) &&
p_Inp->ChromaIntraDisable == 1 && currMB->c_ipred_mode!=DC_PRED_8)
     || (currMB->c_ipred_mode == VERT_PRED_8 && !mb_available[0])
     || (currMB->c_ipred_mode == HOR_PRED_8  && !mb_available[1])
     || (currMB->c_ipred_mode == PLANE_8    && (!mb_available[1]
|| !mb_available[0] || !mb_available[2]))))
      continue;

    //===== GET BEST MACROBLOCK MODE =====
```

```
    for (index=0; index < max_index; index++)
    {
      mode = mb_mode_table[index];
      //printf("mode %d rdcost = %7.3f\n", mode, (double)
currMB->min_rdcost);
      if (enc_mb.valid[mode])
      {
        //printf(" mode %d is valid\n", mode);
        if (p_Vid->yuv_format != YUV400)
        {
          currMB->i16mode = 0;
        }

        // Skip intra modes in inter slices if best mode is inter <P8x8
with cbp equal to 0


/************************************************************************
*****************************************/

/************************************************************************
*****************************************/
          if (!intra & (current_block_SAD < currMB->min_rdcost)) {
             skip_all_intra_mode = 1;
          }else{
             skip_all_intra_mode = 0;
          }

/************************************************************************
*****************************************/

/************************************************************************
*****************************************/
        if (currSlice->P444_joined)
        {
          if (p_Inp->SkipIntraInInterSlices && !intra && mode >= I16MB
            && currMB->best_mode <=3 && currMB->best_cbp == 0 &&
currSlice->cmp_cbp[1] == 0 && currSlice->cmp_cbp[2] == 0 &&
(currMB->min_rdcost < weighted_cost(enc_mb.lambda_mdfp,5)))
            continue;
        }
        else
        {
//          if (p_Inp->SkipIntraInInterSlices | skip_all_intra_mode)
            if (p_Inp->SkipIntraInInterSlices)
          {
            if (!intra && mode >= I4MB)
            {
              if (currMB->best_mode <=3 && currMB->best_cbp == 0 &&
(currMB->min_rdcost < weighted_cost(enc_mb.lambda_mdfp, 5)))
```

```
            {
              continue;
            }
            else if (currMB->best_mode == 0 && (currMB->min_rdcost
< weighted_cost(enc_mb.lambda_mdfp,6)))
            {
              continue;
            }
          }
        }
      }

      compute_mode_RD_cost(currMB, &enc_mb, (short) mode,
&inter_skip);

    }
    //printf(" best %d %7.2f\n", currMB->best_mode, (double)
currMB->min_rdcost);
   }// for (index=0; index<max_index; index++)
 }// for (currMB->c_ipred_mode=DC_PRED_8;
currMB->c_ipred_mode<=chroma_pred_mode_range[1];
currMB->c_ipred_mode++)

 restore_nz_coeff(currMB);

 intra1 = is_intra(currMB);

 //=====  S E T   F I N A L   M A C R O B L O C K   P A R A M E T E
R S ======

//-------------------------------------------------------------------
------------
 update_qp_cbp_tmp(currMB, p_RDO->cbp);
 currSlice->set_stored_mb_parameters (currMB);

 // Rate control
 if(p_Inp->RCEnable && p_Inp->RCUpdateMode <= MAX_RC_MODE)
   rc_store_mad(currMB);


 //===== Decide if this MB will restrict the reference frames =====
 if (p_Inp->RestrictRef)
   update_refresh_map(currMB, intra, intra1);


/************************************************************************
*****************************************/

/************************************************************************
*****************************************/
```

```
    block_total_number ++;
    if (block_total_number == total_block_number) {
        block_total_number = 0;
        frame_total_SAD = 0;
    }

/*************************************************************************
*****************************************/

/*************************************************************************
*****************************************/

}
```