ATLAS DAQ Note 94-27
EAST Note 94-37
December 14, 1994

Algorithms in second-level triggers for ATLAS and benchmark results

R.Hauser, I. Legrand

1 Introduction

For the purposes of implementations and modelling, various assumptions about ATLAS detectors, data formats and trigger algorithms had to be made in the course of EAST work over the last years. They are restricted to a few, not all, detectors, and reflect the respective state of overall knowledge, not the very latest detailed definitions, which in turn will not be exactly the final realisations. They are as close as possible to the actual physics and detector assumptions, so that it can be said with confidence that they are representative of what will eventually have to be implemented.

The algorithms are subdivided into the phases region-of-interest (RoI) collection, feature extraction, and global decision, which in turn has an RoI and an event task. Only the event task is truly global; all other tasks can be naturally parallelised, and this is foreseen in the implementation discussion of ATLAS. We define the three phases in more detail:

Phase 1: The full raw detector data for level-1 triggered events are collected in local non-overlapping memory modules (structured into chips, boards, crates), in a detector-dependent modularity. The raw data pertaining to RoI-s have to be selected by some mechanism, which we term *RoI collection*. This operation is guided by a device realized outside the L2 data stream, which indicates the whereabouts of RoI-s. We assume RoI collection to proceed independently and in parallel for different subdetectors and for different RoI-s.

Phase 2: Algorithms in the concept of RoI have the initial task to convert a limited amount of local raw data from a single subdetector into *features*, variables containing the relevant physics information, like cluster or track parameters that can be used to corroborate or disprove the different physics hypotheses. This phase is called *feature extraction*. Feature extraction algorithms are local and thus can exploit the natural double parallelism of RoI-s and subdetectors.

Phase 3: Physics features have to be collected from all subdetectors and from all RoI-s, for forming an overall decision on the entire event. The natural and efficient order of processing is to combine first all subdetectors that relate to the same physics *object*, by an algorithm which is limited to an RoI. This is then followed by an algorithm combining information from all RoIs into an event decision. Both steps together are termed *global decision*.

All algorithms are defined as C programs. For benchmarking and definition purposes, this is desirable, although implementations will not necessarily be on high-level processors or in the particular form of C programs. In a data-driven model, analogous algorithms have already been demonstrated [1,2].

2 RoI Collection Algorithms

The algorithms in this class are representative of the processing that has to be applied to the raw data in the format as they come from the frontend modules, and prepare the data for the feature extraction algorithm both by extracting only the relevant subset relating to an RoI and optimizing the data format. These have been defined for two detectors, the TRT and the calorimeter (details can be found in [4]). For the raw data, we refer to [3] for definitions.

In the TRT, the raw data come uncompressed and for three time slices. The algorithm performs the transformation to a zero-suppressed format. All points in the buffer belonging to the RoI are considered and for each hit a new data word is generated combining the three time slices in a single one, e.g. by a logical OR for the two possible pulse heights. This is done using a lookup table indexed by the full data word including drift time and pulse height. The ϕ and η offsets are computed for a new coordinate system local to the region of interest. The list of these words represents the output of one receiving unit which must be combined in the next stage to provide the full RoI window data.

The calorimeter receives a single time-extracted sample in uncompressed format. The preprocessing task depends on the RoI type identification of the first-level trigger: for *electromagnetic* RoIs only the channels belonging to the RoI have to be identified, while for *jet-like* RoIs a granularity reduction of the elementary cells has to be performed. Depending on the requirements of the feature extraction processor the full RoI window data may have to be combined into a single image.

3 Feature Extraction Algorithms

The algorithms in this class are representative of the conversion of raw detector data in a limited region of a single subdetector to *primitive* physics information, i.e. a detector part is analysed for the existence of a phenomenon defined by the level-1 trigger: a suspected high-p_T track in the tracker (SCT), possibly also with electron properties (TRT), or cluster parameters in the calorimeter compatible with electron absorption or isolation criteria, or parameters giving improved jet energies. Three algorithms were defined in this category; in the inner detector, a high-precision track finder in the SCT and an electron-biased tracking algorithm in the endcap TRT, plus a calorimeter algorithm considering the electromagnetic and hadronic parts as single layers. The extrapolation from these to algorithms in other detector parts is largely understood.

1. SCT: It has been shown in [5] that by reformulating the combinatorial search through all possible and meaningful hit combinations in the 4 r-ϕ projections of high precision as a sufficiently fine histogramming task, a major gain in execution time at high occupancy can be achieved without loss in the result's precision. It is this *histogramming* or *Hough transform* algorithm for zero-suppressed data input that is given for benchmarking. The benchmark algorithm assumed 4 layers with 1000 bins each. The execution time of this algorithm depends linearly on the occupancy while the combinatorial search shows a $O(N^4)$ dependency, where N is the number of hits in the RoI.

2. The TRT algorithm used is one of *variable slope histogramming* or *Hough transform*, much like for the SCT, with the need of building two Hough transforms [9] which get bin by bin combined via a weighting function, in order to extract the *electronness* along with the existence of a high-p_T track. The image to be analyzed (RoI) for the endcap TRT is in the ϕ/z projection and has the dimension 96 planes (of constant z) by 16 straws (along ϕ). Tracks

in this projection appear as straight lines, with the slope $d\phi/dz$ directly related to p_T, the position along ϕ indicating azimuth, and start and end point crudely indicative for z. The algorithm recognizes patterns of digitizings that correspond to high-momentum tracks, taking into account the pulse height distribution of digitizings for identification of electrons. The times measured for the TRT include only the computing part, assuming that the zero-suppression has been already done in the RoI collection algorithm.

3. The calorimeter algorithm analyses a RoI raster (*image*) of 20×20 electromagnetic cells (with a basic tower size of $.025 \times .025$ in $\Delta\eta \times \Delta\phi$) and 5×5 cells of 0.1×0.1 for the hadronic layer. Three electromagnetic and four hadronic layers are assumed. Electromagnetic cells and hadronic cells, properly weighted, are added, the summed *image* is analyzed for its transverse profile[7]. Inside the RoI, a near-circular region is defined in which the peak energy deposition is fully contained (cluster area). The cluster center is found as the center of gravity of the combined image. The cluster area is defined by all pixels at a radius of 6 pixels or less from the cluster center (all coordinates rounded to pixel centers). Features like hadronic energy fraction F_{had}, and the second moment of the cluster radius in two dimensions for each *em* layer are calculated over the cluster area.

Algorithm:	Calorimeter	TRT	SCT (2.5 %)	SCT (1.0 %)
Min (μs)	700	707	3390	1840
Max (μs)	1450	940	6530	3000

Table 1: Feature Extraction Benchmark Results

4 Global Decision

The two algorithms that make up the feature combination, ending with a final Y/N decision on the entire event, are the RoI task and the event decision.

1. The RoI task converts a set of features from different subdetectors into variables that have been maximised to represent a *probability* for different hypotheses for the physics object that has given rise to the RoI. As an artifical neural network has been used for the optimisation, the algorithm is a particularly fast one to implement. A feed- forward neural net with twelve input variables, a 6-node intermediate layer, and 4 output nodes (probabilities) is implemented. The algorithm simply multiplies all features with six ('synaptic') weights each, forms the sums (i.e. a 12×6 matrix multiplication), followed by a ('sigmoid') conversion by lookup table, and followed by the same procedure (6×4 matrix), to obtain the final result, four probabilities (for 'electron', 'muon', 'hadron', 'jet'). The execution time of this algorithm is data-independent and only determined by the size of the neural network.

2. The event decision goes through the calculation of effective masses, using momentum vectors from all RoI-s. Based on the RoI probabilities and these masses, a sequence of 'IF-THEN-ELSE' statements applies cuts to decide about six different physics hypotheses (multiple 'YES' decisions are possible). The algorithm spends most of its execution time in the computation of the invariant masses. This time is proportional to the N_{roi}^2, where N_{roi} is the

number of RoIs in the event. The benchmarks assumed a mean value of $N_{roi} = 5$.

Algorithm:	RoI Task	Event Task
Min (μs)	5	13
Max (μs)	9	31

Table 2: Global Decision Benchmark Results

5 Benchmarks

All algorithms were benchmarked on different processors in isolation; benchmarking with communication is an ambitious goal for the future, although first results are becoming available [8,9,10,11]. Comparisons between different processors reveal the need for local tuning, but typically the modern RISCs of about 100 MIPS range very close: PowerPC, Alpha, Sparc10. The tables give the fastest and slowest result achived on these processors for each algorithm; the relative achievements of processors are different for the different algorithms and are ignored here.

References

[1] R.K. Bock et al., A commercial image processing system considered for triggering in future LHC experiments. EAST note 94-26.

[2] D. Belosloudtsev et al., Programmable ACtive Memories in real-time tasks: implementing data driven triggers for LHC experiments. EAST note 94-27.

[3] R.K.Bock, P. LeDu, Readout data specifications for modelling a level-2 trigger using regions of interest. ATLAS DAQ-NOTE-25.

[4] I. Legrand, Data collection and preprocessing for the ATLAS second-level trigger. EAST note 94-30(RoI collection).

[5] W. Krischer, L. Moll, Implementation of a pattern recognition algorithm for the Si tracker on DecPeRle-1. EAST note 94-21.

[6] L. Lundheim et al., TRT on DecPeRLe-1 : Algorithm, Implementation, Test and Future. EAST note 94-20.

[7] G. Klyuchnikov et al., A second-level trigger, based on calorimetry only. ATLAS DAQ note 07/EAST note 92-23.

[8] R. Hauser, I. Legrand, Global Decision on the CS-2. EAST note 94-28.

[9] F. Chantemargue, Communication benchmarks with the Ancor Fibre Channel Fabric. EAST note 94-22.

[10] F. Chantemargue, Communication benchmarks with IBMs and HPs. EAST note 94-24.

[11] P.E. Clarke et al., DSP beam test results, EAST note 94-34.

A Algorithm Definitions

The trigger algorithms are available on the World Wide Web under
`http://www.cern.ch/RD11/benchmarks/benchmarks.html`.

Some of the header files and the `Makefiles` are missing in this note. To compile the programs get the original source code from the address above.

A.1 Calorimeter

The calorimeter algorithm assumes a RoI size of 20×20 cells for the electromagnetic layers. This parameter and the ratio of hadronic cell size to electromagnetic cell size can be changed to adapt it to the desired configuration. Furthermore the number of electromagnetic and hadronic layers can be varied.

```
/*
 * $Id: calor.c,v 1.1 1994/05/03 14:42:14 hauser Exp hauser $
 */


#include <stdlib.h>
#include <stdio.h>
#include "benchmark.h"

#define abs(x) (x) < 0 ? -(x) : (x)

#define EM_HA_RATIO 4

/* em and ha cell size */
#define EM_ROI_SIZE 20
#define HA_ROI_SIZE (EM_ROI_SIZE/EM_HA_RATIO)

/* cluster radius */
#define RADIUS 6

/* number of em and ha layers */
#define NO_EM_LAYERS 3
#define NO_HA_LAYERS 4

typedef short Em_Layer[EM_ROI_SIZE][EM_ROI_SIZE];
typedef short Ha_Layer[HA_ROI_SIZE][HA_ROI_SIZE];

typedef Em_Layer Em_Data[NO_EM_LAYERS];
typedef Ha_Layer Ha_Data[NO_HA_LAYERS];

/* results returned by calorimeter algorithm */
typedef struct {
  int xcm,ycm;                                    /* cluster center */
  double em_m2[NO_EM_LAYERS];                  /* 2nd moment for em layer N */
  double em_ha_ratio;                           /* fraction of hadronic energy */
} Result;

/*
 * calorimeter algorithm
 */
void calorimeter(Em_Data em_data,Ha_Data ha_data,Result *res)

{
  int x,y,i;                                                /* index variables */
  int x_orig,y_orig;                                 /* 'lower left' corner of cluster */

  double sum_of_energy,total_energy;
  double em_energy,ha_energy;
  int xcm,ycm;
```

```
  /*
   * sum up layers, compute center of gravity
   */
  xcm = ycm = 0;
  total_energy = 0.0;
  for(x = 0; x < EM_ROI_SIZE; x++)
    for(y = 0; y < EM_ROI_SIZE; y++) {
      sum_of_energy = 0.0;

      /* sum over em layers */
      for(i = 0; i < NO_EM_LAYERS; i++)
        sum_of_energy += em_data[i][x][y];

      /* sum over ha layers */
      for(i = 0; i < NO_HA_LAYERS; i++)
        sum_of_energy +=
ha_data[i][x/EM_HA_RATIO][y/EM_HA_RATIO]/(EM_HA_RATIO*EM_HA_RATIO);

      xcm += x * sum_of_energy;
      ycm += y * sum_of_energy;
      total_energy += sum_of_energy;
    }

  /* compute center of gravity */
  xcm /= total_energy;
  ycm /= total_energy;

  res->xcm = xcm;
  res->ycm = ycm;

  /*
   * select 'circle' with radius R
   */
  x_orig = xcm - RADIUS;
  y_orig = ycm - RADIUS;

  /* compute 2nd moment for em layers */
  for(i = 0; i < NO_EM_LAYERS; i++) {
    double m2 = 0.0;

    em_energy = 0.0;
    for(x = x_orig; x < x_orig + RADIUS*2; x++)
      for(y = y_orig; y < y_orig + RADIUS*2; y++) {
        if (!((x < 0) || (x > EM_ROI_SIZE) ||
            (y < 0) || (y > EM_ROI_SIZE))) {
          int dx = abs(x - xcm),
              dy = abs(y - ycm);
          /* second moment */
          m2 += em_data[i][x][y] * (dx * dx + dy * dy);
          em_energy += em_data[i][x][y];
        }
      }
    res->em_m2[i] = m2/em_energy;
  }

  /* compute ha energy and ha fraction */
  for(i = 0; i < NO_HA_LAYERS; i++) {
    ha_energy = 0.0;
    for(x = x_orig; x < x_orig + RADIUS*2; x ++)
      for(y = y_orig; y < y_orig + RADIUS*2; y ++) {
        if (!((x < 0) || (x > EM_ROI_SIZE) ||
            (y < 0) || (y > EM_ROI_SIZE))) {
          ha_energy +=
ha_data[i][x/EM_HA_RATIO][y/EM_HA_RATIO]/(EM_HA_RATIO*EM_HA_RATIO);
        }
      }
    res->em_ha_ratio = em_energy/ha_energy;
```

```
  }

}

/*
 * initialize data with random values
 */
static void init_data(Em_Data em,Ha_Data ha)

{
  int x,y,i;

  for(x = 0; x < EM_ROI_SIZE; x++)
    for(y = 0; y < EM_ROI_SIZE; y++)
      for(i = 0; i < NO_EM_LAYERS; i++)
        em[i][x][y] = random() % 12000;

  for(x = 0; x < EM_ROI_SIZE/EM_HA_RATIO; x++)
    for(y = 0; y < EM_ROI_SIZE/EM_HA_RATIO; y++)
      for(i = 0; i < NO_HA_LAYERS; i++)
        ha[i][x][y] = random() % 12000;
}

/*
 * main routine
 * parse command line arguments
 * time loop which calls calorimeter algorithm
 */
int main(int argc,char *argv[])

{
  int count = 1;

  int i;
  Em_Data em;
  Ha_Data ha;
  Result res;

  CLOCK_T t0,t1,diff;

  if (argc > 1)
    count = strtol(argv[1],0,0);

  init_data(em,ha);

  START_CLOCK(t0);
  START_LOOP(count)
    calorimeter(em,ha,&res);
  END_LOOP
  READ_CLOCK(t1);
  CLOCK_DIFF(diff,t1,t0);

  printf("center of grav.:  x = %d, y = %d\n",res.xcm,res.ycm);
  for(i = 0; i < NO_EM_LAYERS; i++)
    printf("em layer = %d, M2 = %f\n",i,res.em_m2[i]);
  printf("em/ha = %f\n",res.em_ha_ratio);
  printf("total time = "); PRINT_CLOCK(stdout,diff); printf(" usec\n");

  exit(0);
}

/* Local Variables: */
/* compile-command: "gcc -g -O2 -funroll-loops -DBENCHMARK -DSUNOS calor.c -
o calor" */
/* End: */
```

A.2 TRT

The TRT algorithm assumes an input of 400 points to analyze and checks for seven different slopes. These parameters are defined at the beginning of the program and can be adjusted accordingly.

```
/*
 * This is an example implementation for the TRD algorithm.
 * Any implementation on a given architecture should optimize
 * the data representation and the algorithm itself.
 */

#include <stdio.h>
#include <malloc.h>
#include "benchmark.h"

#define NR_POINTS 400                                   /* # of point to be analyzed */
#define NR_SLOPES 7                                    /* # of slopes to be considerd */
#define NR_BINS 32                                                          /* */

static float SLOPE[NR_SLOPES] = {-0.3, -0.2, -0.1, 0.0, 0.1, 0.2, 0.3};

/* LuT  PARAMETERS                    */
#define LUT_SIZE 65536                                                  /* 2^16   */
#define LUT_TYPE unsigned short
#define LUT_OFF1 8
#define LUT_OFF2 12
#define LMASK1 0xff
#define LMASK2 0xf00

/* DETECTOR GEOMETRY PARAMETERS       */
#define FACT1 0.25
#define FACT2 0.25
#define PI 3.141529

/* GLOBAL                             */
static LUT_TYPE *LUT[NR_SLOPES];

void data_decode(unsigned int data, float *z, float *f, float *dt, int *ig)

{
  /* this function uncompresses the data */
  /* this version is s simplified one    */
  /* and does not take into account all  */
  /* details of the detector geometry    */

  unsigned short in1, in2, in3, in4;
  unsigned short mask1 = 0xfe00;                          /* mask for plane #  */
  unsigned short mask2 = 0x01e0;                          /* mask for straw #  */
  unsigned short mask3 = 0x0008;                         /* mask for grey value */
  unsigned short mask4 = 0x0007;                         /* mask for drft time  */

  in1 = (data & mask1) >> 9;
  in2 = (data & mask2) >> 5;
  in3 = (data & mask3) >> 3;
  in4 = data & mask4;

  *z = (float) in1;                                       /* z or r index */
  *f = (float) (in2 - 1);                                 /* phi          */
  *dt = (float) in4;                                       /* drift time   */
  *ig = in3;                                              /* gray value   */
}

void iniLUTs()
/* initialize the LUT for each slope    */
```

```
{
  int i, j, ig, id, val;
  unsigned int k;
  LUT_TYPE *p;
  float z, f, dt;

  for (i = 0; i < NR_SLOPES; i++) {
    p = malloc(LUT_SIZE * sizeof(LUT_TYPE));                    /* allocate memory */
    LUT[i] = p;
  }

  for (j = 0; j < LUT_SIZE; j++) {
    k = j;
    data_decode(k, &z, &f, &dt, &ig);
    for (i = 0; i < NR_SLOPES; i++) {
      p = LUT[i];
      val = (int) ((f - SLOPE[i] * z) * FACT1);
      id = (int) (dt * FACT2);
      if (((val - id) < 0) ||
          ((val - id) ≥ NR_BINS) ||
          ((val - id) < 0) ||
          ((val - id) ≥ NR_BINS))
        p[j] = -1;
      else
        p[j] = (ig ≪ (LUT_OFF2)) + (id ≪ LUT_OFF1) + val;
    }
  }

}

void list_trd(unsigned short *data, int *slope_nr, int *phi_pos, int *t_hit, int *ratio)

{
  int is, i, iad, ig, val, cea, id;
  int histo[NR_BINS];
  int lf_pos, lt_hit;
  short unsigned *po;
  short unsigned j;

  /* reset the features */
  *slope_nr = -1;
  *phi_pos = -1;
  *t_hit = 0;
  *ratio = 0.0;

  for (is = 0; is < NR_SLOPES; is++) {

    for (i = 0; i < NR_BINS; i++)
      histo[i] = 0;

    po = data;
    for (i = 0; i < NR_POINTS; i++) {
      j = *(po++);
      iad = *(LUT[is] + j);

      if (iad > 0) {
        ig = iad ≫ (LUT_OFF2);
        val = iad & LMASK1;
        id = (iad & LMASK2) ≫ LUT_OFF1;
        if (ig == 0)
          cea = 256;
        else
          cea = 257;
        histo[val + id] += cea;
        histo[val - id] += cea;
      }
    }
```

```
    lf_pos = 0;
    lt_hit = histo[0];
    for (i = 1; i < NR_BINS; i++) {
      if (histo[i] > lt_hit) {
        lt_hit = histo[i];
        lf_pos = i;
      }
    }

    if (lt_hit > *t_hit) {
      *slope_nr = is;
      *t_hit = lt_hit;
      *phi_pos = lf_pos;
    }
  }

  *ratio = (*t_hit & 0xff) * 256 / (*t_hit >> 8);
  *t_hit = *t_hit >> 8;

}

int main(int argc, char *argv[])

{

  unsigned short date[NR_POINTS];
  int i;
  int slope_nr, phi_pos, t_hit, ratio;

  int count = 1;
  CLOCK_T t0, t1, diff;

  if (argc > 1)
    count = strtol(argv[1], 0, 0);

  iniLUTs();

  printf("data ini\n");

  /* receive the data     */
  for (i = 0; i < NR_POINTS; i++)
    date[i] = rand();

  /* start time measurement        */
  START_CLOCK(t0);
  START_LOOP(count)
    list_trd(date, &slope_nr, &phi_pos, &t_hit, &ratio);
  END_LOOP;
  /* stop time measurement        */
  READ_CLOCK(t1);
  CLOCK_DIFF(diff, t1, t0);

  printf(" slope_nr=%d phi_pos=%d t_hit=%d ratio=%d \n", slope_nr, phi_pos, t_hit, ratio);
  printf("time = ");
  PRINT_CLOCK(stdout, diff);
  printf(" usec\n");

  exit(0);

}
```

A.3 SCT

The occupancy of the SCT can be defined at the beginning of the program as a parameter.

```
#include <stdio.h>
#include "benchmark.h"


#define NR_POINTS 100                                   /* # of point to be analyzed     */
                                /* used just for testing          */
#define MAX_NR_POINTS 500                           /* maximum # of points           */

/*         detector & track parameters */
#define LGHIST 64
#define NANG 36

int IC;

struct track {
int im;
int ih;
};

int indhist ( int im, int ir, int v )
{

 const float dam = 0.25;
 const float ano = -50.0;
 const float rs[4] = { 70.0 , 80.0, 90.0, 100.0 };
 const float bw = 2.5;
 int cuc;

 cuc = (int) (( (float)v - (float)im * dam*rs[ir] - ano)/bw );
 if ( cuc < 0 ) cuc = LGHIST;
 if ( cuc > LGHIST ) cuc = LGHIST;
 return cuc;
}

void sit( short unsigned *date , int *nr_tracks, struct track *track )
/*                                                  */
/*        SIT       feature extraction              */
/*                                                  */
/*             Werner Krischer's algorithm          */
/*                                                  */

{
 int i,j,ic,ih;
 int ind,isum;
 short unsigned d;
 int val[MAX_NR_POINTS],lay[MAX_NR_POINTS];
 int ihists[LGHIST+1][NANG];
 const int isums[16] = { 0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4 };
 const int ircodes[4] = { 8 , 4, 2 , 1 };

 for( j=0;j<NR_POINTS;j++){
   d = date[j];
   lay[j] = d >> 10;
   val[j] = d & 0x3ff;
 }

 IC =0;

 for ( i=0;i<NANG;i++ ) {
   for (j=0;j<LGHIST+1;j++ )
     ihists[j][i]=0;

   for( j=0;j<NR_POINTS;j++){
     ind = indhist ( i, lay[j], val[j] );
     ihists[ind][i] = ihists[ind][i] | ircodes[lay[j]];
   }
```

```
 }

 *nr_tracks = 0;

 /* ic =0 */
 for ( i =0 ; i < NANG; i++ ) {
   for ( ih =0 ; ih < LGHIST ; ih ++ ) {
     if ( ihists[ih][i] == 15 ) {
       track[*nr_tracks].im = i ;
       track[*nr_tracks].ih = ih;
       (*nr_tracks)++;
     }
   }
 }

 IC =1;
 if ( *nr_tracks > 1000) return;

 for (i =0; i< NANG; i++ ) {
   for (ih=0; ih <LGHIST-1; ih++ )
     ihists[ih][i] = ihists[ih][i] | ihists[ih+1][i];
 }

 /*  ic =1 */
 for ( i =0 ; i < NANG; i++ ) {
   for ( ih =0 ; ih < LGHIST ; ih ++ ) {
     if ( ihists[ih][i] == 15 ) {
       track[*nr_tracks].im = i ;
       track[*nr_tracks].ih = ih;
       (*nr_tracks)++;
     }
   }
 }
 IC =2;
 if ( *nr_tracks > 0) return;

 /*ic = 2 */
 isum =3;

 for ( i =0 ; i < NANG; i++ ) {
   for ( ih =0 ; ih < LGHIST ; ih ++ ) {
     if ( isums[ihists[ih][i]] == isum ) {
       track[*nr_tracks].im = i ;
       track[*nr_tracks].ih = ih;
       (*nr_tracks)++;
     }
   }
 }

 IC =3;
}

main(int argc,char *argv[])

{
  short unsigned date[NR_POINTS];
  struct track track[200];

  int i;
  int nr_track;

  CLOCK_T t1,t2,diff;
  int count = 1;

  if (argc > 1)
    count = strtol(argv[1],0,0);
```

```
/*      receive the data     */
 i =100;

 date[0] = (0 << 10 ) + i;
 date[1] = (1 << 10 ) + i+ 5;
 date[2] = (2 << 10 ) + i+ 10;
 date[3] = (3 << 10 ) + i+ 15;

 for (i=4;i<NR_POINTS;i++)
    date[i]= ((i%4)<<10)+(rand()%1023);
/*  generate random data -> 10bit in z and 4 planes */

   START_CLOCK(t1);
   START_LOOP(count);
   sit( date, &nr_track, track);
   END_LOOP
   READ_CLOCK(t2);
   CLOCK_DIFF(diff,t2,t1);

   printf( " # of tracks = %d & ic =%d \n",nr_track,IC);
   printf("time = "); PRINT_CLOCK(stdout,diff); printf(" usec\n");

}
```

A.4 Region of Interest Task

```
/*
   $Id: neural2.c,v 1.3 1994/01/20 10:39:51 rhauser Exp $

   simulation of the Region of Interest Task (troi) : third
   implementation in C language, with optimization : the inner
   loop has been removed and replaced by simple affectations :
   the sigmoid function has been replaced by a LUT
*/

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

/* int NN_layer0 = 12, */                          /* number of input neurons  */
/*     NN_layer1 = 6,    */                       /* number of hidden neurons */
/*     NN_layer2 = 4;      */                     /* number of output neurons */

/* assume that the network topology is fixed, so that
   we can use #define instead of variables
   => the index range of the loop variable is known by the
   compiler and it can unroll the loop
*/

#define NN_layer0 12
#define NN_layer1 6
#define NN_layer2 4

float w1[200],                                  /* weights input-layer -> hidden-layer */
      w2[200],                                 /* weights hidden-layer -> output-layer */
      t1[20],                                                    /* threshold  */
      t2[20];                                                    /* threshold  */

float LUT[64000];

/*
 * sigmoid function
 */
float sigfx(float x)
{
```

```
  float y;

  y = 1/(1+exp(-2*x));
  return(y);
}

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

main()

{
  float in[12],                                          /* NN inputs */
        out[4];                                          /* NN outputs */

  float a, b,                                            /* intermediate results */
        o[20];                                           /* state of the hidden neurons */

  float c,bb;

  int i, j,jj,j1;                                        /* index variables */

  clock_t clock1, clock2;
  float time;

  int k;                                                 /* index */
  int ia;


  /* initialize inputs,weights and lookup table
   * with random values
   */
  for (j1=0;j1<12;j1++)
    in[j1]= (float) drand48();

  for (j1=0;j1<20;j1++) {
    t1[j1]= (float)drand48();
    t2[j1]= (float)drand48();
  }

  for(j1=0; j1<200; j1++)
    w1[j1]=(float)drand48();

  for (j1=0;j1<100;j1++)
    w2[j1]=(float)drand48();

  for (j1=0; j1<64000; j1++)
    LUT[j1]= (float)drand48();

  bb =0.0;

  /* ********************************** */
  clock1 = clock();

  for ( jj=0;jj<100000;jj++) {                           /* outer loop for timing */

    c=0;
    for (i=0; i<NN_layer1; i++) {                        /* compute state of hidden neurons */

      a=t1[i];                                                    /* threshold */
      k = i*NN_layer0;                                   /* position of weights in array */
                                  /* unrolled loop: */
      a += in[0]*w1[k];
      a += in[1]*w1[k+1];
      a += in[2]*w1[k+2];
      a += in[3]*w1[k+3];
      a += in[4]*w1[k+4];
      a += in[5]*w1[k+5];
```

```
        a += in[6]*w1[k+6];
        a += in[7]*w1[k+7];
        a += in[8]*w1[k+8];
        a += in[9]*w1[k+9];
        a += in[10]*w1[k+10];
        a += in[11]*w1[k+11];

/*        for(j1 = 0; j1 < NN_layer0; j1++) */
/*        a += in[j1] * w1[k+j1]; */

        /*
          o[i] = sigfx(a);
        */

        o[i] = LUT[(int)(a*102)];                          /* lookup value in table */

      }

      for (i=0; i<NN_layer2; i++) {                  /* compute state of output neurons */

        a=t2[i];                                                        /* threshold */
        k = i*NN_layer1;                                  /* position of weights in array */
                                    /* unrolled loop */
        a += o[0]*w2[k];
        a += o[1]*w2[k+1];
        a += o[2]*w2[k+2];
        a += o[3]*w2[k+3];
        a += o[4]*w2[k+4];
        a += o[5]*w2[k+5];

        /* for (j1 = 0; j1 < NN_layer1; j1++) */
        /* a += o[j1] * w2[k+j1]; */

        /*
         * out[i] = sigfx(a);
         */

        out[i]=LUT[(int)(a*102)];
        c += out[i];
      }

      bb += c;
    }

    clock2 = clock();
    time = (clock2-clock1)*(1000000/100000)/(CLOCKS_PER_SEC);
    printf( " bb = %f \n",bb);
    printf("execution time [us]= :  %f \n", time);

}
```

A.5 Event Task

```
/* Trigonometric functions of the invariant mass function have been replaced
   by Look Up Tables
   this file is not a simulation file : luts contain true values
   Computation time measurements of one invariant mass task

*/

/*****************************************************************
*                      EVENT   TASK
*
*                                              version 0
*                                               cil/04/93
*
*  **     This C code is a functional example of this task  representing
```

```
*     the expected complexity of the L2 global  processing.
*     An efficient implementation of this program is architecture
*     dependent; it is anticipated that each specific implementation will
*     make optimizing improvements where possible or necessary, whilst
*     maintaining the functionality of the algorithm.
*
*  **     The " physics  tasks " used in this version must be
*     interpreted as an example. We use only six channels for
*     this exercise but for the real trigger more such "physics
*     tasks " will be necessary. The scalability concerning this
*     part has to be considered and also the flexibility to change
*     these pieces of code. Each channel analysis is suplied as a
*     separate function in order to keep the modularity of the
*     interesting physics processing.
*
*  **     This version is done using floating point arithmetic.
*     An integer arithmetic version requires proper data scaling.
*
*  **     The input data is the output from all the RoI tasks for
*     each event. In this test program the input data are read from file.
*     The data input problem is also architecture dependent. In this
*     program the input data is loaded in a vector and predefined
*     pointers select different RoI. In this data file the MC true physics
*     flag is written for testing purposes.
*
*  **     The thresholds used in this program are also read from a file.
*
*  **     A testing part is incorporated in this program. This is not
*    part of global second level trigger task.
*
*
**********************************************************************/


/*          include files       */

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

/*                                */
int NNROI=16;                                /* maximum number of RoI per event  */
int NWR=8;                                    /* Words per RoI                    */

/*        Thresholds used in the global decision       */
/*          -> read from an input file                  */

float C11,C12,C13,C14;
float C21,C22,C23,C24;
float C31,C32,C33,C34,C35,C36;
float C41,C42,C43,C44,C45,C46;
float C51,C52,C53,C54;
float C61,C62,C63,C64;


/*        Global variables */
int nroi,npair;
int phy_type;
float *proi[16];
float roi_data[256];
float mas[200],pe[200],pj[200],pm[200];
float pte,ptj,ptm;

#define max 1000000

#define etamin -3
#define etamax 3
```

```
#define lim 32768
#define lim_lut 16384
#define angle_min 0
#define angle_max (2*M_PI)
#define diff_eta (etamax-etamin)
#define interval (angle_max-angle_min)
#define lim1 (lim/2)
#define coefficient1 (lim_lut/diff_eta)
#define coefficient2 (lim1/interval)


float eta;                                                    /* from -3 to +3 */

float LUT[lim_lut+1];
float LUT1[lim+1];
float *pLUT1;

/*          functions                */

int read_data( int );                                        /*   open and read input data   */
int read_thre();                                             /*   open and read thresholds   */
float invmas ( int, int );                                   /*   pair invariant mass        */
int t4e();                                                   /*   trigger H->ZZ->4e          */
int t4m();                                                   /*   trigger H->ZZ->4m          */
int t2e2m();                                                 /*   trigger H->ZZ->2e2m        */
int t2e2j();                                                 /*   trigger H->ZZ->2e2j        */
int te3j();                                                  /*   trigger H->tt->e3j         */
int tm3j();                                                  /*   trigger H->tt->m3j         */

main()

{
  int i,j,is,k;
  int gltrg;
  int nevents;
  int bgs,gs,gs1,gs2,gs3,gs4,gs5,gs6;                        /* variables for testing part */
  int dummy, ii, il;
  clock_t clock0, clock2;
  float time0;
  FILE *pfile;

  /*  initialize pointers for data       */
  proi[0]= roi_data;
  for (i=1;i<NNROI;i++)
    proi[i]= proi[i-1]+NWR;

  /*  read thresholds from file          */
  if ( read_thre()!=1 ) {
    printf ( " *** error read thresholds file *** \n");
    exit(0);
  }

  /*     open data file                  */
  if ( read_data(1)!=1) {
    printf ( " *** error open data file *** \n");
    exit(0);
  }

  /*pfile = fopen("inv_mass.data", "w");*/
  pfile = fopen("pipo", "w");

  nevents =0;

  /*        initialize the test variables        */
  bgs=0;
  gs=0;
  gs1=0;
```

```
  gs2=0;
  gs3=0;
  gs4=0;
  gs5=0;
  gs6=0;

  /* fill luts */
  fill();                                                    /* for eta --> theta */
  fill1();                                                 /* for theta -> cos(theta) */
  pLUT1 = &LUT1[lim/2];

  /*        start  EVENT TASK   */
  for ( ; ; ) {                                                  /* events loop    */
    is = read_data(2);                                           /* read one event */
    if ( is == -1) {                                              /* end of file   */
      printf( " EOF %d events analyzed \n",nevents);
      fclose(pfile);

      /*             print the  results from the testing part          */
      printf( "background signal accepted=%d total signal=%d s1=%d s2=%d s3=%d
s4=%d s5=%d s6=%d\n",
              bgs,gs,gs1,gs2,gs3,gs4,gs5,gs6);
      exit(0);
    }
    if (is≠1) {                                                /* error reading data */
      printf( "*** error read event data ***\n");
      exit(0);
    }
    /*scanf("%d", &dummy);*/
    nevents++;
    npair=nroi*(nroi-1)/2;

    k=0;

    /* pair invariant mass & pair particles " probability " calculation */
    for (i=0;i<nroi-1;i++ ) {
      for (j=i+1;j<nroi;j++ ){

        clock0 = clock();
        for (ii=0; ii<max;ii++) {
          for (il=0;il<NNROI;il++) {
            *(proi[il]) += 0.00000001;
            *(proi[il]+1) += 0.00000001;
            *(proi[il]+2) += 0.00000001;
            *(proi[il]+4) += 0.00000001;
            *(proi[il]+6) += 0.00000001;
            *(proi[il]+7) += 0.00000001;
          }
          mas[k] = invmas(i, j);
        }
        clock2 = clock();
        time0 = (float)((clock2 - clock0)/max) ;
        fprintf(pfile,"%d\t %d\t %d\t %f\n",nevents-1,phy_type,nroi,time0);
        fflush(pfile);

        pe[k] = *(proi[i]+1) + *(proi[j]+1);
        pj[k] = *(proi[i]) + *(proi[j]);
        pm[k] = *(proi[i]+2) + *(proi[j]+2);
        k++;
      }
    }

    /* total "particles probability" for this event  */
    pte=0;
    ptj=0;
    ptm=0;
```

```
    for (i=0;i<nroi;i++) {
      pte += *(proi[i]+1);                          /* "electron probability" */
      ptj += *(proi[i]);                            /* " jet probability "    */
      ptm += *(proi[i]+2);                          /* " muon probability  "  */
    }

    /*          create the  global trigger word              */
    gltrg=0;
    gltrg += t4e();
    gltrg += (t4m()<<1);
    gltrg += (t2e2m()<<2);
    gltrg += (t2e2j()<<3);
    gltrg += (te3j()<<4);
    gltrg += (tm3j()<<5);

    /*           testing part ( this is not part of Event task ) */
    if ( (phy_type==1) && ( gltrg!=0) )
      bgs++;
    if ( (phy_type!=1) && ( gltrg!=0) )
      gs++;
    if (( phy_type==2) && (gltrg!=0) )
      gs1++;
    if (( phy_type==3) && (gltrg!=0) )
      gs2++;
    if (( phy_type==4) && (gltrg!=0) )
      gs3++;
    if (( phy_type==5) && (gltrg!=0) )
      gs4++;
    if (( phy_type==6) && (gltrg!=0) )
      gs5++;
    if (( phy_type==7) && (gltrg!=0) )
      gs6++;
  }
}

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

fill()
{
  int l;

  for(l=0; l<=lim_lut; l++) {
    eta = etamin + (diff_eta*l)/lim_lut;
    LUT[l] = atan((1/sinh(eta)));
    if (LUT[l]<0) {
      LUT[l] += M_PI;                                  /* other solution */
    }
  }
}


fill1()

{

  int i, l;
  float angle;

  for(l=0; l<=lim1; l++) {
    angle = angle_min + (interval*l)/lim1;
    LUT1[lim1+l] = cos(angle);
    LUT1[lim1-l] = LUT1[lim1+l];
  }
}


int read_data ( int ci)
```

```
{
  /* read data function
     if ci = 1 open the data file
     if ci = 2 read one event and put the data
     into the input stream
   */
  static FILE *f;
  int i,is;
  float tr,parid;

  if ( ci==1) {
    if ( (f = fopen("sample.data","r")) == NULL )
      return 0;
    return 1;
  }

  if ( ci==2 ) {
    is = fscanf( f," %d %d ",&phy_type,&nroi);                    /* read event header  */
    if ( is==EOF )
      return -1;

    /* read all the RoI data and put them in the input vector              */

    for ( i=0;i<nroi;i++) {
      is = fscanf (f,"%f %f %f %f %f %f %f %f %f %f",
                   &tr,&parid, proi[i],
                   (proi[i]+1),(proi[i]+2),(proi[i]+3),
                   (proi[i]+4),(proi[i]+5),(proi[i]+6),(proi[i]+7));
      if ( is≠10)
        return -3;                                  /* incomplete record for RoI       */
    }
    return 1;                                                          /* O.K. */
  }
  else {
    return -2;                                              /* undefined input flag     */
  }
}

int read_thre()
{
  /*                  Read thresholds used in different type
                      of physics channels which are triggered.
                      This values depends on the RoI algorithm
                      and also may be luminosity dependent.
                      In this program these values are read
                      from a file
 */
  FILE *g;
  int is;

  if ( (g = fopen("thresh.d","r")) == NULL )
    return -2;

  is= fscanf (g,"%f %f %f %f", &C11,&C12,&C13,&C14);
  is= fscanf (g,"%f %f %f %f", &C21,&C22,&C23,&C24);
  is= fscanf (g,"%f %f %f %f %f %f", &C31,&C32,&C33,&C34,&C35,&C36);
  is= fscanf (g,"%f %f %f %f %f %f", &C41,&C42,&C43,&C44,&C45,&C46);
  is= fscanf (g,"%f %f %f %f", &C51,&C52,&C53,&C54);
  is= fscanf (g,"%f %f %f %f", &C61,&C62,&C63,&C64);
  fclose(g);
  return 1;
}


float invmas ( int i, int j)
{
```

```
  float cpd, ctp, ctd;
  int ax1, ax2, ax3;
  float theta1, theta2;
  int l;

        /*theta1 = LUT[(int)( *(proi[i]+6) - etamin)*coefficient1)];
        theta2 = LUT[(int)( *(proi[j]+6) - etamin)*coefficient1)];*/

        /* in the future, we will have theta1 and theta2 instead of eta1 and eta2 */

  theta1 = *(proi[i]+7);
  theta2 = *(proi[j]+7);
  cpd = *(pLUT1 + (int)(( *(proi[i]+7) - *(proi[j]+7) )*coefficient2));
  ctp = *(pLUT1 + (int)((theta1+theta2)*coefficient2));
  ctd = *(pLUT1 + (int)((theta1-theta2)*coefficient2));

  /*return (sqrt((*(proi[i]+4))*(*(proi[j]+4))*(2-ctp*(1-cpd)+ctd*(1+cpd))));*/
  /* it is useless to return the square root of the value */

  return ( (*(proi[i]+4))*(*(proi[j]+4))*(2-ctp*(1-cpd)+ctd*(1+cpd)) );

}

int t4e()
{

  /*          H -> ZZ ->  4e       channel         */

  int imz,imze,i;
  if ( nroi<4)
    return 0;
  if ( pte<C11)                    /* a threshold condition in the total e " probability " */
    return 0;
  imz=0;
  imze=0;
  for ( i=0;i<npair;i++) {
    if (( mas[i]>C12 ) && ( mas[i]<C13) ) {          /* count the pairs mij in mZ region*/
      imz++;
      if ( pe[i]>C14 )
        imze++;                          /* the pair is most probably an electron pair   */
    }
  }
  if ( imz < 2 )
    return 0;
  if ( imze <2 )
    return 0;
  return 1;
}

int t4m()
{

  /*            H  -> ZZ  -> 4m     channel          */
  int imz,imzm,i;

  if ( ptm<C21)              /* a threshold condition in the total muon " probability " */
    return 0;
  return 1;
}

int t2e2m()
{
  /*           H  -> ZZ  -> 2e2m       channel   */
  int imz,imze,imzm,i;
  if ( nroi<4)
    return 0;
  if ( ptm<C31)              /* a threshold condition in the total muon " probability " */
```

```
    return 0;
  if ( pte<C32)                    /* a threshold condition in the total e " probability " */
    return 0;
  imz=0;
  imze=0;
  imzm=0;

  for ( i=0;i<npair;i++) {
    if ( (mas[i]>C33 ) && ( mas[i]<C34) ) {        /* invariant mass condition counting  */

      imz++;
      if ( pm[i]>C35 )                          /* the pair is most probably a muon pair   */
        imzm++;
      if ( pe[i]>C36 )                            /* the pair is most probably an e pair   */
        imze++;
    }
  }
  if ( imz < 2 )
    return 0;
  if ( imzm <1 )
    return 0;
  if ( imze <1 )
    return 0;
  return 1;
}


int t2e2j()
{

  /*           H  -> ZZ  -> 2e2j       channel   */
  int imz,imze,imzj,i;
  if ( nroi<4)
    return 0;
  if ( pte<C41)              /* a threshold condition in the total electron " probability " */
    return 0;
  if ( ptj<C42)                   /* a threshold condition in the total jet " probability " */
    return 0;
  imz=0;
  imze=0;
  imzj=0;
  for ( i=0;i<npair;i++) {
    if (( mas[i]>C43 ) && ( mas[i]<C44) ) {        /* invariant mass condition counting  */
      imz++;
      if ( pj[i]>C45 )                             /* the pair is most probably a jet pair   */
        imzj++;
      if ( pe[i]>C46 )                       /* the pair is most probably an electron pair   */
        imze++;
    }
  }
  if ( imz < 2 )
    return 0;
  if ( imzj <1 )
    return 0;
  if ( imze <1 )
    return 0;
  return 1;
}

int te3j()
{
  /*           H  -> tt  -> e3j       channel   */
  int imze,imzj,i;

  if ( ptm>C51)
    return 0;
```

```
  imzj=0;
  imze=0;
  for ( i=0;i<nroi;i++) {
    if ( *(proi[i])>C52 )
      imzj++;
    if ( *(proi[i]+1)>C53 )
      imze++;
  }
  if ( imzj <2 )
    return 0;
  if ( imze <1 )
    return 0;
  return 1;
}

int tm3j()
{
  /*            H  -> tt  -> m3j       channel   */
  int imzm,imzj,i,k;
  float pm;

  if ( pte>C61)
    return 0;
  pm =0;
  imzm=0;
  imzj=0;
  for ( i=0;i<nroi;i++) {
    if ( *(proi[i])>C62 )
      imzj++;
    if ( *(proi[i]+2)>pm )
      pm = *(proi[i]+2);
  }
  if ( imzj <2 )
    return 0;
  if ( pm<C63 )
    return 0;
  return 1;
}
```

A.6 TRT Preprocessing

The following shows only the actual algorithmic part for the TRT RoI collection algorithm, omitting e.g. the code for histogramming and graphics output.

```
#ifndef _TRDE_
#define _TRDE_

#include "detector.h"
#include "T1.h"

// front_end data structure
typedef struct trde_fe_data {
  int ev_nr;                                          // event number
  int data[TRDE_BUF];                                 // raw data
};

// list of results
typedef struct trde_data_list{
  int ev_nr;                                          // event number
  int len;                                  // number of items in list
  int buff[TRDE_BUF];                                 // items
};

// RoI collection class
class roi_collect {
  int my_id;
```

```
  int nrpac;
  // int point;
  int xnp;
  int cl;
  // int iev;
  // int i,j,k;
  // int lbuff[TRDE_BUF];

  public:

  roi_collect( int id);
  void add_data( struct trde_data_list * buf);
  void print_stat();
};

// TRD receiver (buffer) class
class trde_receiver{
  int my_id;
  int eta1,eta2,phy1,phy2, leta1,leta2;
  int NN1,NN2,NN3;
  int *LuTe, *LuTp ;
  int iLuTe[TRDE_NPL+1], iLuTp[TRDE_NSTM+1];
  struct trde_fe_data *myd;
  trde_data_list *list;
  roi_collect *DEST[maxRoIs];

  int dest;

  public:
  trde_receiver(int i, int le1, int le2, int ieta, int iphy,
                int deta, int dphi, roi_collect *D[]);
  virtual ~trde_receiver();
  void new_ev (int ev);
  int get_T1 ( struct T1_info & iT1 );

};

#endif

#include "trde.h"
#include <ACG.h>
#include <Normal.h>
#include <Uniform.h>
#include <math.h>
#include <stdio.h>
#include <time.h>
#include "histox.h"

extern ACG gen, gen1;
extern int LuTdt[];
extern int NCY;
extern histox *hh[];

// roi_collection creation
roi_collect::roi_collect( int id)
{
  my_id = id;
  // point = 0;
  nrpac = 0;
  // iev = 0;
  xnp = 0;
  cl = 0;
}


void roi_collect::add_data ( trde_data_list * buf)
{
```

```
  //  int i = buf->len;
  //  int j = (buf->ev_nr & 65535);

  //  printf ( " RC id = %d ev_id =%d  l=%d \n",my_id,j,i);

  cl += buf→len;
  xnp++;
  nrpac++;
}


void roi_collect::print_stat()
{
  cl = cl/NCY;
  xnp = xnp/NCY;

  // printf ( "put in H cl =%d xnp = %d from Col_id =%d \n",cl,xnp,my_id);

  if ( xnp > 0 ) {
    hh[5]→add ((float)cl);
    hh[6]→add ((float)xnp);
  }

  cl = 0;
  xnp =0;
}


trde_receiver::trde_receiver(int i,int le1,int le2,int ieta,int iphy,
                             int deta, int dphy, roi_collect *D[])
{
  double p;
  int j;

  my_id = i;
  leta1 = le1;
  leta2 = le2;
  eta1 = ieta;
  eta2 = eta1+deta;
  phy1 = iphy;
  phy2 = phy1+dphy;

  myd = new trde_fe_data;
  list = new trde_data_list;

  for ( j = 0; j < maxRoIs; j++)
    DEST[j] = D[j];

  // generate LuT for transformation from local eta/phy offsets
  // in local indexes

  LuTe = new int[deta+1];
  LuTp = new int[dphy+1];

  // fill the LuTs assuming uniform detector
  p = (double)(TRDE_NPL)/(double)(deta-1);

  for(j = 0; j < deta; j++){
    LuTe[j] = (nint) (p*j);
  }

  // the inverse transformation
  p = (double) deta/double(TRDE_NPL-1);

  for (j = 0; j < TRDE_NPL; j++)
    iLuTe[j] = (nint) ( j*p);
```

```
  p = (double)(TRDE_NSTM)/(double)(dphy-1);

  for( j = 0; j < dphy; j++){
    LuTp[j] = (nint)(p*j);
  }

  p = (double) dphy/(double)(TRDE_NSTM-1);
  for (j = 0; j < TRDE_NSTM; j++ )
    iLuTp[j] = (nint) ( j*p);

  NN1 = 12;
  NN2 = 6;
  NN3 = 0;

}


trde_receiver::~trde_receiver()
{
  delete myd;
  // delete list; ???
}

void trde_receiver::new_ev ( int ev)
{
  //       generate a new event

  int i,m;
  double j,k;
  Normal oc( TRDE_OCC*10.0, TRDE_OCCW*10.0, &gen);
  Uniform t( 0.0, 1000.0, &gen1);

  myd->ev_nr = ev;
  m = 0;

  j = oc();

  // generate front-end data
  for(i = 0;i < TRDE_BUF; i++) {
    k = t();
    if (k < j) {
      myd->data[i] = i % 10 + 2;
      m++;
    } else {
      myd->data[i] = 0;
    }
  }

  hh[2]->add( (float)(m*100/TRDE_BUF));
}

inline int stex ( int &y1, int &y2, int &ay1, int &ay2)

{
  //    test RoI in phy
  //

  if ( ay1 ≥ y2) return 0;
  if ( ay2 < y1 ) return 0;

  if ( ay1 ≥ y1 && ay2 < y2 ) {
    ay1 = ay1-y1;
    ay2 = ay2-y1;
    return 1;
  } else if ( ay1 > y1 && ay2 ≥y2 ) {
    ay1 =ay1-y1;
    ay2 =y2 -y1;
```

```
    return 1;
  } else if ( ay1 < y1 && ay2 < y2) {
    ay1 =0;
    ay2 = ay2-y1;
    return 1;
  } else if ( ay1 ≤ y1 && ay2 ≥y2 ) {
    ay1=0;
    ay2 =y2-y1;
    return 1;
  }

  return -3;
}

inline int sqtex ( int &y1, int &y2, int &ay1, int &ay2)
{
  int stex( int&, int&, int&, int&);
  int ay1s,ay2s;

  //     test RoI in phy
  //

  if ( ay1 ≥0 && ay2 ≤ API )
    return stex ( y1,y2,ay1,ay2);

  ay1s = ay1;
  ay2s = ay2;

  if ( ay1 < 0 ) {
    ay1 = 0;
    if ( stex ( y1,y2,ay1,ay2) == 1 )
      return 1;
    else {
      ay1 = API+ay1s;
      ay2 = API;
      return stex ( y1,y2,ay1,ay2);
    }
  }

  if ( ay2 > API ) {
    ay2 = API;
    if ( stex ( y1,y2,ay1,ay2) == 1 )
      return 1;
    else {
      ay1 = 0;
      ay2 = ay2s-API;
      return stex ( y1,y2,ay1,ay2);
    }
  }

}


int trde_receiver::get_T1 ( struct T1_info & iT1)

{
  int l1,l2,l3,l4;
  //  int reta1,reta2;
  //  int reta1s,reta2s
  int ie1,ie2,ip1,ip2;
  int sqtex( int&, int&, int&, int&);
  int dw,v,d,q,s;

  int k = 0;
  long t1 = clock();

  for (int icycle = 0;icycle < NCY; icycle++) {
```

```
    for (int i = 0; i < iT1.n_ROI; i++) {                    // for all RoIs
      int ll = 0;
      //  test if this buffer contains part of the i th  RoI in eta
      if ((iT1.r_eta[i] > leta1 ) && (iT1.r_eta[i] < leta2) ) {

        //  define the RoI in phy

        int rphy1 = iT1.r_phi[i]-trde_deltap_e;
        int rphy2 = iT1.r_phi[i]+trde_deltap_e;
        int rphy1s = rphy1;
        int rphy2s = rphy2;


        if ( sqtex(phy1,phy2,rphy1,rphy2) == 1 ){
          k++;
          // make the list !

          ip1 = LuTp[rphy1];
          ip2 = LuTp[rphy2-1];

          for (int l1 = ip1; l1 < ip2; l1++) {
            int l2 = l1*TRDE_NPL;
            for ( l3=0;l3<TRDE_NPL;l3++) {
              l4 = l2+l3;
              d = myd->data[l4];
              v = LuTdt[d];
              if ( v ≠ 0 ) {
                int yphy = phy1-rphy1s+iLuTp[l1];
                int yeta = eta1-leta1+iLuTe[l3];
                dw = ( yeta<<NN2) +( yphy<<NN1) + v +icycle;
                list->buff[ll++] = dw;
              }
            }
          }
          ll =ll-1;
          if ( ll≥0) {
            list->len = ll;
            list->ev_nr =iT1.ev_nr+(icycle<<16);
            dest = i;
            DEST[dest]->add_data( list);
          }
        }
      }  // end if
    }  // end RoI loop

  }  // end icycle

  long t2 = clock();
  long dt = (t2-t1)/NCY;
  if ( k >0 && dt > 5)
    hh[3]->add(dt);
  else
    hh[4]->add(dt+0.2);
  //    printf ( " end ev =%d \n",iT1.ev_nr);
  return k;
}
```

A.7 Calorimeter Preprocessing

```
//          CALORIMETER - BARREL


#include "detector.h"                          // constants for the detector geometry
#include "T1.h"                                // First Level trigger class definition

#ifndef _CALB_
```

```
#define _CALB_

// front_end data structure
  // data are sent uncompressed
typedef struct calb_fe_data {
  int ev_nr;
  int presh[BUFSZ_PS];
  int ele1[BUFSZ_EL1];
  int ele2[BUFSZ_EL2];
  int had1[BUFSZ_HA];
  int had2[BUFSZ_HA];
  int had3[BUFSZ_HA];
};

// RoI data collecting unit
class roi_collect {
  int my_id;
  int nrpac;
  int cl;
  int iev;
  int roi_id;

  public:

  roi_collect( int id);
  void add_data( int *buf);
  void print_stat();
};

// Calorimeter receiving unit
class calb_receiver{
  int my_id;
  int eta1,eta2,phy1,phy2;
  int NN1,NN2,NN3;
  int *LuTe, *LuTp ;
  struct calb_fe_data *myd;
  roi_collect *DEST[maxRoIs];
  int *buff;

  int dest;

  public:
  calb_receiver(int i,int ieta, int iphy,
                int deta, int dphi, roi_collect *D[]);
  virtual ~calb_receiver();
  void new_ev (int ev);
  int get_T1 ( struct T1_info & iT1 );
};

#endif

#include "cv_calb.h"
#include "funct.h"
#include <ACG.h>
#include <Normal.h>
#include <Uniform.h>
#include <math.h>
#include <stdio.h>
#include <time.h>
#include "histox.h"

extern ACG gen, gen1;
extern int NCY;
extern histox *hh[];

roi_collect::roi_collect( int id)

{
```

```
  my_id = id;
  roi_id = -1;
  nrpac =0;
  iev = -1;
  cl =0;
}


void roi_collect::add_data ( int *buf)

{
  int l;
  l = buf[1];                                                    // buffer size
  cl += l*2;                                                  // # Bytes received

  if (nrpac == 0) {
    iev = buf[2];                                                      // Event id
    nrpac++;                                                 // number of packeges per RoI
    roi_id = buf[3];
  } else {
    if (buf[2] ≠ iev)
      printf( " *** Error -> event_id \n");
    if (buf[3] ≠ roi_id)
      printf( " *** Error -> Roi_id \n");
    nrpac++;
  }

  // If an "image" format it is used for the FE
  // the data received in buf have to be copied.

  // FE may start here for algorithms which do
  // not need the data in an image format
}


void roi_collect::print_stat()

{
  cl = cl/NCY;
  nrpac = nrpac/NCY;

  hh[5]→add ((float)cl);
  hh[6]→add ((float)nrpac);

  // reset the unit for the next event

  cl = 0;
  nrpac=0;
  iev =-1;
  roi_id =-1;
}


calb_receiver::calb_receiver( int i,int ieta,int iphy,
                              int deta, int dphy, roi_collect *D[])
{
  double p;
  int j,k,k1;
  my_id = i;
  eta1 = ieta;
  eta2 = eta1 + deta;
  phy1 = iphy;
  phy2 = phy1 + dphy;
  buff = new int[1024];

  myd = new calb_fe_data;
```

```
  for (j = 0;j < maxRoIs; j++)                    // copy the pointers for the destination units
    DEST[j] = D[j];

  // generate LuT for transformation from local eta/phy offsets
  // in local indexes
  LuTe = new int[deta + 1];
  LuTp = new int[dphy + 1];

  // fill the LuTs assuming uniform detector
  p = (double)(28.0)/(double)(deta);

  for(j=0;j≤deta;j++){
    k= (nint) (p*j);
    k1 = k/2;
    k=k1*2;
    LuTe[j] = k;
  }

  p = (double)(4.0)/(double)(dphy);

  for(j=0;j≤dphy;j++){
    k = (nint)(p*j);
    k1 =k/2;
    k =k1*2;
    LuTp[j]=k;
  }
}


calb_receiver::~calb_receiver()
{
  delete myd;
}

void calb_receiver::new_ev ( int ev)
{
  //         generate a new event
  //  fill the structure with random values
  //  Loading simulated data will be done later

  int i,m;
  double j,k;

  Uniform DD ( 0.0, 16000.0 ,&gen);

  for (i = 0;i < BUFSZ_PS;i++)
    myd→presh[i] = (int) DD();

  for (i = 0; i < BUFSZ_EL1;i++)
    myd→ele1[i]= (int) DD();

  for (i = 0;i < BUFSZ_EL2;i++)
    myd→ele2[i] = (int) DD();

  for (i = 0;i < BUFSZ_HA;i++){
    myd→had1[i] = (int) DD();
    myd→had2[i] = (int) DD();
    myd→had3[i] = (int) DD();
  }
}

inline int stex (int &y1, int &y2, int &ay1, int &ay2)

{
  //     test RoI
  //
```

```
  if (ay1 ≥ y2) return 0;
  if (ay2 < y1) return 0;

  if (ay1 ≥ y1 && ay2 < y2) {
    ay1 = ay1 - y1;
    ay2 = ay2 - y1;
    return 1;
  } else if (ay1 > y1 && ay2 ≥y2) {
    ay1 = ay1 - y1;
    ay2 = y2 - y1;
    return 1;
  } else if (ay1 < y1 && ay2 < y2) {
    ay1 = 0;
    ay2 = ay2-y1;
    return 1;
  } else if (ay1 ≤ y1 && ay2 ≥ y2) {
    ay1 = 0;
    ay2 = y2 - y1;
    return 1;
  }

  return -3;
}

inline int sqtex ( int &y1, int &y2, int &ay1, int &ay2)
{
  int stex(int&, int&, int&, int&);
  int ay1s,ay2s;

  //     test RoI in phy
  //

  if ( ay1 ≥ 0 && ay2 ≤ API )
    return stex (y1,y2,ay1,ay2);

  ay1s = ay1;
  ay2s = ay2;

  if ay1 < 0 {
    ay1 = 0;
    if (stex (y1,y2,ay1,ay2) == 1)
      return 1;
    else {
      ay1 = API + ay1s;
      ay2 = API;
      return stex(y1,y2,ay1,ay2);
    }
  }

  if (ay2 > API) {
    ay2 = API;
    if (stex ( y1,y2,ay1,ay2) == 1)
      return 1;
    else {
      ay1 = 0;
      ay2 = ay2s - API;
      return stex(y1,y2,ay1,ay2);
    }
  }

}



int calb_receiver::get_T1 ( struct T1_info & iT1)
{
  // Receives the L1 data strcucture and test
```

```
// if this unit it is concerned. For each
// RoI overlapping this buffer region a list
// containing all the raw data necessary for
// the feature extraction units it is generated
// and sent it to the RoI collectors or FE units.

int i,k,lx,ll,ip,ie,ik,je1,je2,il,is;
int reta1,reta2,rphy1,rphy2;
int reta1s,rphy1s,reta2s,rphy2s,yphy,yeta;
int ie1,ie2,ip1,ip2;
int sqtex( int&, int&, int&, int&);
int dw,v,d,q,s;
long t1,t2,dt;
int icycle;

k=0;

t1 = clock();

buff[0]=0;
buff[1]=0;

for ( icycle=0;icycle<NCY;icycle++) {                    // time measurement
  for ( i=0; i<iT1.n_ROI; i++) {                              // for all RoIs

    if ( iT1.r_type[i] == 0 ) {
      reta1 = iT1.r_eta[i] - cal_deltae_e;
      reta2 = iT1.r_eta[i] + cal_deltae_e;
      rphy1 = iT1.r_phi[i] - cal_deltap_e;
      rphy2 = iT1.r_phi[i] + cal_deltap_e;
    } else {
      reta1 = iT1.r_eta[i] - cal_deltae_j;
      reta2 = iT1.r_eta[i] + cal_deltae_j;
      rphy1 = iT1.r_phi[i] - cal_deltap_j;
      rphy2 = iT1.r_phi[i] + cal_deltap_j;
    }

    rphy1s = rphy1;
    rphy2s = rphy2;
    reta1s = reta1;
    reta2s = reta2;
    lx=0;
    ll=0;

    if ((sqtex(phy1,phy2,rphy1,rphy2) == 1) &&
        (stex(eta1,eta2,reta1,reta2) == 1)) {
      k++;
      lx = 1;


      ip1 = LuTp[rphy1];
      ip2 = LuTp[rphy2];
      ie1 = LuTe[reta1];
      ie2 = LuTe[reta2];

      buff[2] = iT1.ev_nr;
      buff[3] = iT1.n_ROI;
      buff[4] = i;


      if (iT1.r_type[i] == 0) {                                  // leptons

        buff[5] = ie1;
        buff[6] = ie2;
        buff[7] = ip1;
        buff[8] = ip2;
        ll = 9;
```

```
    for (ie = ie1*8;ie ≤ ie2*8; ie++) {
      buff[ll++] = myd→presh[ie];
    }

    for (ip = ip1;ip ≤ ip2; ip++) {
      for (ie = ie1;ie ≤ ie2; ie++) {
        ik = ip * 28 + ie;
        buff[ll++]= myd→ele1[ik];
      }
    }

    for (ip = ip1/2;ip ≤ ip2/2; ip++) {
      for (ie = ie1;ie ≤ ie2; ie++) {
        ik = ip * 28 + ie;
        buff[ll++] = myd→ele2[ik];
      }
    }

    for (ie = ie1/4; ie ≤ ie2/4; ie++) {
      buff[ll++] = myd→had1[ie];
      buff[ll++] = myd→had2[ie];
      buff[ll++] = myd→had3[ie];
    }
  }  // end leptons

  else {                                                    // jets
    je1 = ie1/4;
    je2 = ie2/4;
    buff[5] = je1;
    buff[6] = je2;
    buff[7] = ip1;
    buff[8] = ip2;
    ll =9;

    for ( ie = je1; ie ≤ je2; ie ++ ) {
      s = 0;
      is = ie*32;
      for ( ip = 0; ip < 32; ip++)
        s+= myd→presh[is++];

      for (ip = 0; ip < 4; ip++) {
        for (il = 0; il < 4; il++) {
          ik = ie*4+il+ip*28;
          s += myd→ele1[ik];
        }
      }

      buff[ll++]=s;

      s = 0;
      for (ip = 0; ip < 2; ip++) {
        for (il = 0;il < 4; il++) {
          ik = ie*4+il+ip*28;
          s += myd→ele2[ik];
        }
      }

      buff[ll++] = s;
      buff[ll++] = myd→had1[ie];
      buff[ll++] = myd→had2[ie];
      buff[ll++] = myd→had3[ie];
    }
  }  // end  jets

  if ( lx ==1 ) {
```

```
            buff[1] = ll;
            DEST[i]→add_data( buff);
            hh[2]→add( (double)(2*ll));

          }
        }  // end if
      }  // end RoI loop
    }  // end icycle

    t2 = clock();
    dt = (t2-t1)/NCY;
    if (k ==0){
      hh[4]→add(dt);
    } else {
      hh[3]→add(dt);
    }
    return k;
}
```