Reconstructing the Trigger Delay from the TTC Values for FCal Physics Data

Adam Hincks
University of Toronto

29 August 2003

Abstract

Algorithms for determining the delay between the event trigger and the TDC clock is presented.

1 The Problem

During runs a TDC clock produces a square wave with a period of 25 ns. Event triggers are measured relative to this clock. Two values are calculated, a *direct TTC*, measured from the rising edge of the wave, and a *delayed TTC*, measured from the falling edge of the wave (see fig. 1). The units of the TTCs is 50 ps, so their range is from 0 to 500, or 0 ps to 25000 ps = 25 ns.

Figure 2 schematically illustrates the problem of reconstructing the true time from the two TTC values. (Note that the "true" time is always a time measured relative to the TDC clock, so it has some constant phase.) The TDC times are plotted against the true time as dotted lines. They are out of phase with one another. Since their dynamic range is between 0 and 500 counts, or 0 and 25 ns, they wrap around. It is impossible to construct a true time that does not wrap from one TTC value alone. Thus, the problem consists of determining what the true time is by a correct combination of the two TTC values.

2 The Algorithm

2.1 Basics

In fig. 2 a red line indicates the goal of the algorithm, a reconstructed "true" time, which does not wrap. The x-axis has been divided into three regions (I, II, III) whose boundaries are determined by where the TTC values wrap.

In region II, the delayed TTC is always greater than the direct. Here we take an average of the values to get the reconstructed time.

In regions I and III, the direct TTC is greater. To distinguish between them, we compare the delayed TTC value to the *breakpoint*. If it is greater, then we are in region I; if less, we are in region III. The breakpoint is determined by analysing real data, and

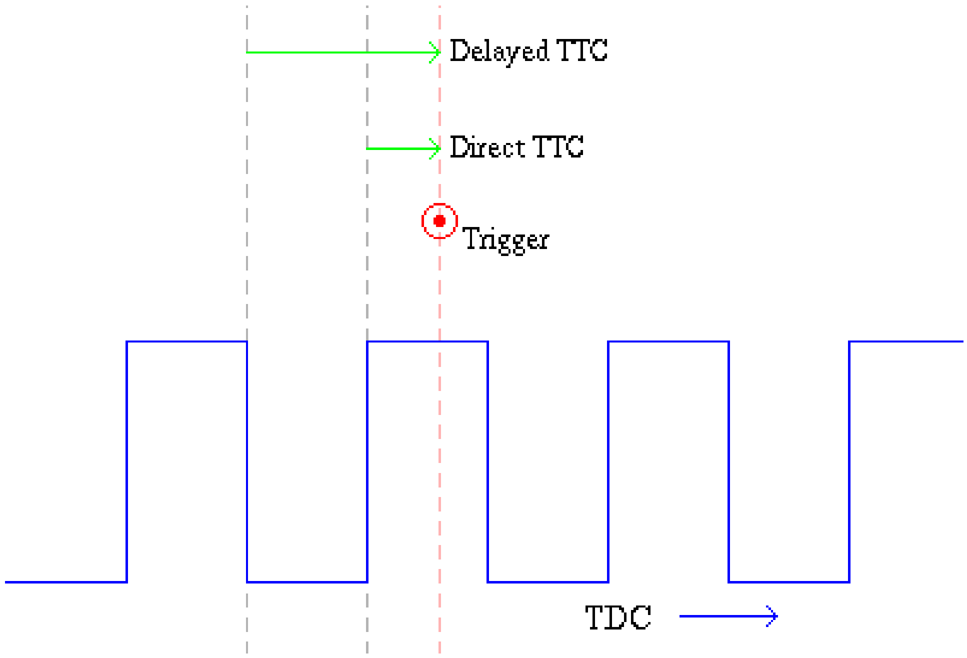

Figure 1: The relationship between the TDC and the TTCs.

will be discussed below. In region I, we take the average and subtract 12.5 ns, or half of the TTC range. In region III, we add 12.5 ns to the average.

2.2 Complications

Figure 3 shows real TTC data. It is essentially the same plot as fig. 2 using TTC values from one event of a physics run. The "real" time is determined by drawing a parabola through the highest three points of the pulse, and then taking the time at which its maximum occurs. (Only events with pulses amplitudes above a reasonable threshold are included.) The gross inaccuracy of this determination of the timing is evident by the width of the lines, but is sufficient to get accurate timing.

Also note that the minimum TTC values are not 0, but that each has its own pedestal, which has to be subtracted off before the time can be determined.

In the meantime, there is another problem with this plot, which is that the break-point is not clearly defined. (See 3.2 and fig. 5. This is quite evident if the plot is zoomed in at around 350 on the y-axis. Some points from the bottom of the right hand arm (region III) of the delayed TTC values are lower than the highest points of the left arm (region I). Thus, points near the boundaries of region I with region II and region II with region III need to be carefully considered. Most points above the breakpoint will be in region III, but some may actually be in region I. If this is not taken account of, some events will appear to have delays which are shifted by 12.5 ns.

My solution was to place the breakpoint where the wrapping occured on average, and to define a zone 0.25 ns above and below it. The TTC values are uncertain to less than 0.25 ns, so if the delayed TTC value is outside of this zone, we can use the simple placement algorithm of 2.1. Inside this zone, we draw a parabola through the highest

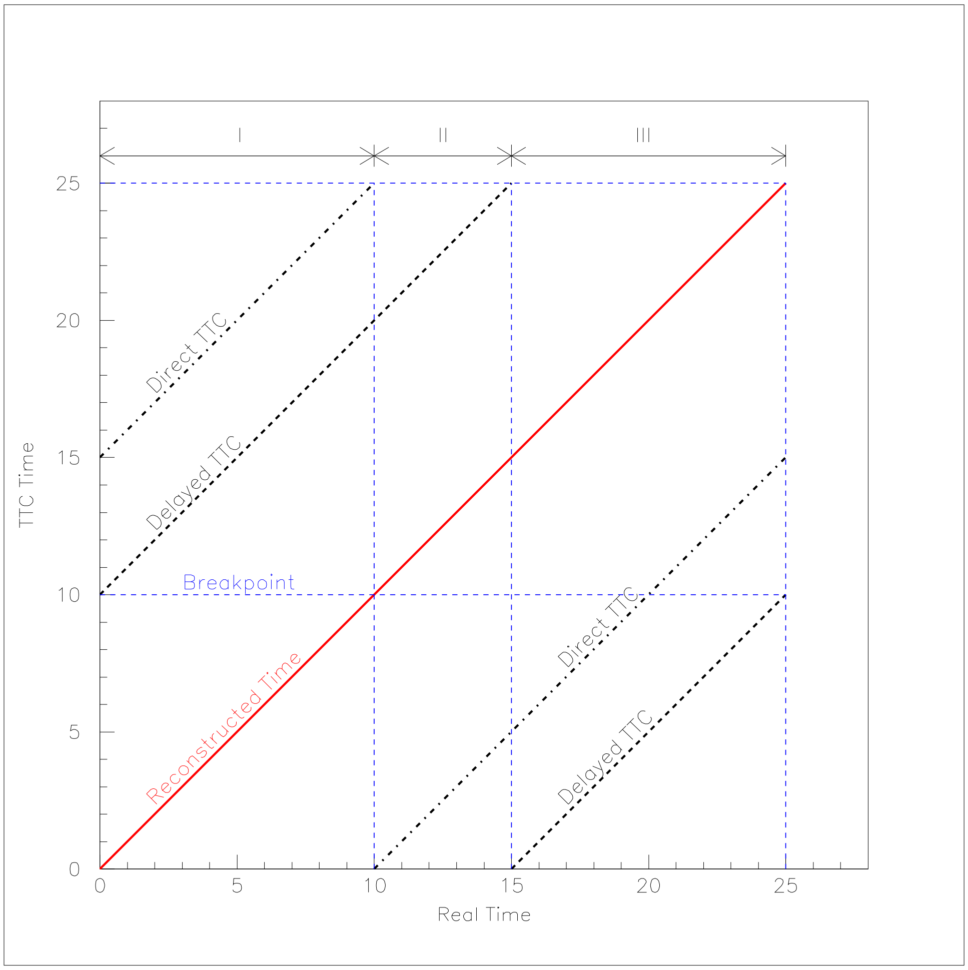

Figure 2: The two TTCs and the reconstructed time

three samples of the pulse and look at the time where the centre falls. This parabola time is used to determine whether we are in region I or II.

In the case where the delayed TTC value is greater than the direct TTC value, the solution is the same as the algorithm of 2.1. Near the borders between regions I and II and regions II and III there may be ambiguities, but if the delayed TTC value is greater, the average of the two will always give the correct reconstructed time.

2.3 C++ Code

The following code is used by my process class[1] to calculate the timing. It has been modified (more comments) to be included here and irrelevant parts removed. It uses the algorithm described above.

```
/* Some values defined elsewhere:
     double ttcdelay = 0.05 -- the conversion between TTC values and nanoseconds
```

[1]See my documentation, reference.html, for more information on this class and how it can be used.

```
     double dirttcmin -- the pedestal for the direct TTC
     double delttcmin -- the pedestal for the delayed TTC
     double calbreakpoint -- the breakpoint
*/

#define TIMING_MARGIN           0.25
#define TIMING_PARABOLAGOODNESS 0.1
#define TIMING_MINPULSE         30

/* This function returns the delay, determined from the TTC values, in
 * conjuction with a parabolic interpolation, in necessary. */

double timing::delay(runfile *rf) {
  double dirttc, delttc, truetime, partime;

  /* The dynamic range of the TTCs is around 506 (not 500 as they would be
   * ideally.  Multiply by 0.05 to convert to nanoseconds. */
  calrange = 506 * 0.05;

  /* Read the TTC values from the run file, subtract pedestals, and convert to
   * nanoseconds. */
  dirttc = (double)(rf->dirttc() - dirttcmin) * ttcdelay;
  delttc = (double)(rf->delttc() - delttcmin) * ttcdelay;

  /* Find the true time as a combination of direct & delayed times. */
  if (delttc > dirttc) {    /* Region II -- simple! */
    truetime = (delttc + dirttc) / 2.0;
  else {
    /* Check to see if the delayed TTC is in the 0.25 ns danger zone about the
     * breakpoint. */
    if (delttc > calbreakpoint - TIMING_MARGIN &&
        delttc < calbreakpoint + TIMING_MARGIN) {

      /* Revert to using a parabola to get the time.  The getparabolatime
       * function is included and explained below. */
      partime = getparabolatime(rf);
      if (partime > 1 - TIMING_PARABOLAGOODNESS)
        truetime = (delttc + dirttc - calrange) / 2.0;  /* Region III. */
      else if (partime < -1 + TIMING_PARABOLAGOODNESS)
        truetime = (delttc + dirttc + calrange) / 2.0;  /* Region I. */
      else
        truetime = NO_TIMING;   /* Still too ambiguous to get a timing. */
    }
    if (delttc < calbreakpoint)
      truetime = (delttc + dirttc + calrange) / 2.0;  /* Region I. */
    else
      truetime = (delttc + dirttc - calrange) / 2.0;  /* Region III. */
```

```
  }

  return truetime;
}

/* This function is called by timing::delay in cases of timing ambiguity.  It
 * uses a parabola drawn through the highest three samples to determine the
 * timing.

double timing::getparabolatime(runfile *rf) {
  double a, b, c;
  double samples[] = {-25, 0, 25, 50};
  double val[rf->numsamples()];
  int i, j, k, sum, num, feb, chan, gain;

  for (i = 0, num = 0, sum = 0; i < hc->numhit(); i++) {
    /* The hc object contains a list of channels that are receiving energy for
     * the current run.  We only want to use the parabola method for channels
     * where there is a substantial pulse.  A parabola drawn through the top
     * three points of a channel with no signal makes no sense. */
    feb = hc->list(i).feb;
    chan = hc->list(i).chan;
    gain = hc->list(i).gain;

    /* Get the pulse shape. */
    for (j = 0; j < rf->numsamples(); j++)
      val[j] = (double)rf->adc(feb, chan, gain, j, true);

    /* The maxvalueindex returns the index for the highest element in the
     * array. */
    k = maxvalueindex(val, rf->numsamples());

    /* Only use pulses where the maximum sample is the fourth/fifth and
     * there is actually a real pulse. */
    if (val[k] < TIMING_MINPULSE || k < 3 || k > 4)
      continue;

    num++;
    /* Draw a parabola through the samples. */
    fitparabola(samples + k - 3, val + k - 1, &a, &b, &c);

    /* Look at whether the parabola falls in region I (< 0) or region III
     * (> 0). */
    if (parabolacentre(a, b, c) <= 0)
      sum--;
    else
      sum++;
```

```
  }

  /* The return value indicates how well the hit channels agree on whether the
   * delay was long or short.  If the (sum / num) == -1 then it means all the
   * parabola centres fell to the left of the fourth sample; a number like 0.97
   * would mean that there were perhaps one or two pulses with weird parabolas;
   * but 0.2 would mean that we really have no significant answer and we cannot
   * determine the timing for the given event.
   *
   * When the value is returned, timing::delay insists that the value be no more
   * than TIMING_PARABOLAGOODNESS away from -1 or 1. */
  if (num) {
    return (double)(sum / num);
  }
  else
    return 0;
}
```

3 Determining the Breakpoint and Pedestals

The breakpoint and the two pedestals (offsets of the TTC values) are tedious to deal with because they all vary from run to run. The pedestals move two or three counts, while the breakpoint varies by more than 10 counts (see fig. 4). For this reason it is necessary to calculate these values for each run file.

3.1 Pedestals

The pedestals are calculated simply by finding the minimum value of the TTC values from all the events of the run file. The direct TTC pedestal is around 300 and the delayed pedestal is around 277.

3.2 Breakpoint

Finding the breakpoint requires a more complicated algorithm. Figure 5 is a plot of the delayed TTC values against the time determined from a parabola. The plot is zoomed in the y-axis where the TTCs wrap. It is clear that there is overlap – sometimes the wrap occurs higher, sometimes lower.

The goal is to define a breakpoint where the delayed TTC wraps on average. Using parabola timing in a buffer zone, as described in 2.2, can then be used to deal with the ambiguous points near the breakpoint.

To find the breakpoint, we seek two values, labelled *breakhigh* and *breaklow* in fig. 5, which define where the delayed TTC appears to wrap in regions I and III, respectively. The breakpoint we take to be the average of these two lines.

Note that the lines leave out a small number of values. For example, the breakhigh line has two points above it, one at 349 counts and another at 353 counts. It is necessary to leave out individual points like this because it happens in every run file that there are a few spurious points far from the true breakpoint. These could be due to a problem with the parabolic timing, or to the fact that the timing of some events really is off for some reason, shifting a point from region III to I or vice-versa.

Ideally, one would determine breakhigh and breaklow by eye for each run file, as it is somewhat subjective. My process class, however, has an algorithm for doing so (see 3.3. It uses a combination of two criteria. First, it insists that the breakhigh and breaklow lines each run through a minimum number of points – in fig. 5, this minimum is 4. Second, it requires that the breakhigh and breaklow values not be separated by too great a distance.

Let us illustrate, using fig. 3.2. Suppose we require that breakhigh and breaklow not be separated by more than 5 counts, and begin with the criterion that they only need to run through one point. Then our breakhigh is at 353 and our breaklow is at 345. $345 - 353 = 12 > 5$, so now we require that the lines run through at least two points. Our breaklow moves up to 347 which certainly has more than two points. Our breakhigh moves down to 349, but this only has one point, so it moves down again to 348, which has 4 points. Now we have $348 - 347 = 1 < 5$, which satisfies both criteria.

3.3 C++ Code

Below is copied the function from my C++ process class[2], with more comments added, which calculates the pedestals and breakpoint.

```
#define TIMING_MINTTCEVENTS     5
#define TIMING_MAXTTCSEPARATION 5

void timing::getttcdata(char *filename) {
  int dirlowest = 99999, dirhighest = 0;
  int dellowest = 99999, delhighest = 0;
  int lowttcvals[1000], highttcvals[1000];
  int breakhigh, breaklow, minttcevents;
  double a, b, c;
  int i, j, k, l, feb, chan, gain;
  runfile *rf;
  double *val;
  double time[] = {-500, 0, 500, 1000};

  printf("Getting TTC information . . .\n");

  /* Access the run file. */
  rf = new runfile(filename);
  val = new double[rf->numsamples()];

  /* Initialise arrays. */
  for (i = 0; i < 1000; i++) {
```

[2]See my documentation, reference.html, for more information on this class and how it can be used.

```
  lowttcvals[i] = 0;
  highttcvals[i] = 0;
}

while (rf->readnext() >= 0) {
  if (rf->type() == EVENT_PHYS) {
    rf->unpack();

    if (rf->delttc() && rf->dirttc()) {

      /* Do a fail-safe check -- the TTC values should always be between 0
       * and 900 with lots of room to spare, so this is perhaps
       * unnecessary. */
      if (rf->delttc() > 900 || rf->delttc() < 200 || rf->dirttc() > 900 ||
          rf->dirttc() < 200)
        continue;

      /* Find the lowest and highest values.  The lowest value for each TTC
       * will be the its pedestal. */
      if (rf->delttc() > delhighest)
        delhighest = rf->delttc();
      if (rf->dirttc() > dirhighest)
        dirhighest = rf->dirttc();
      if (rf->delttc() < dellowest)
        dellowest = rf->delttc();
      if (rf->dirttc() < dirlowest)
        dirlowest = rf->dirttc();
    }

    /* We will fill two histograms of the TTC values.  One will be for
     * values in region I and the other for values in region III.  The region
     * is determined using a parabola. */
    if (rf->delttc() < rf->dirttc()) {
      for (i = 0; i < hc->numhit(); i++) {

        /* The hc object contains a list of channels that receive energy in
         * this run.  We only want to fit a parabola to pulses that have some
         * amplitude. */
        feb = hc->list(i).feb;
        chan = hc->list(i).chan;
        gain = hc->list(i).gain;
        for (j = 0; j < rf->numsamples(); j++)
          val[j] = (double)rf->adc(feb, chan, gain, j, true);

        /* Only use pulses where the maximum sample is the fourth/fifth and
         * there is actually a real pulse. */
        k = maxvalueindex(val, rf->numsamples());
```

```
        if (val[k] < TIMING_MINPULSE || k < 3 || k > 4)
          continue;

        /* Draw a parabola through the top three points. */
        fitparabola(time + k - 3, val + k - 1, &a, &b, &c);

        /* Fill a histogram of TTC values. */
        if (parabolacentre(a, b, c) < 0)
          highttcvals[rf->delttc()]++;  /* Region I. */
        else
          lowttcvals[rf->delttc()]++;   /* Region II. */
      }
    }
  }
}

if (dirlowest == 99999) {
  printf("Warning: run file does not have any TTC information.\n");
  dirttcmin = 0;
  delttcmin = 0;
  range = 0;
  breakpoint = 0;
  return;
}

/* Now we use the histograms to find breakhigh and breaklow, using the
 * algorithm explained in the last section.
 *
 * minttcevents is the minimum number of points through which breakhigh and
 * breaklow must pass; TIMING_MAXTTCSEPERATION is the maximum distance apart
 * breakhigh and breaklow can be */

breaklow = 1000;
breakhigh = 0;
minttcevents = TIMING_MINTTCEVENTS - 1;
while (abs(breakhigh - breaklow) > TIMING_MAXTTCSEPARATION) {
  minttcevents++;

  for (i = 0; i < 1000; i++) {
    if (lowttcvals[i] >= minttcevents) {
      breaklow = i;
      break;
    }
  }

  for (i = 999; i >= 0; i--) {
    if (highttcvals[i] >= minttcevents) {
```

```
        breakhigh = i;
        break;
      }
    }

    /* It is conceivable that this algorithm will fail and go out of control.
     * In this case we use a default value, but this is undesireable.
     * It rarely happens if minttcevents and TIMING_MAXTTCSEPARATION are chosen
     * properly. */
    if (minttcevents > 50) {
      printf("Warning: was not able to find the delayed TTC wrap-around "
          "point. Using the default value of %d instead.\n",
          defaultbreakpoint);
      breakhigh = defaultbreakpoint;
      breaklow = defaultbreakpoint;
      break;
    }
  }

  /* Breakpoint is the average of breakhigh and breaklow. */
  breakpoint = (double)(breakhigh + breaklow) / 2.0;
  dirttcmin = dirlowest;
  delttcmin = dellowest;

  /* Range is the dynamic range of the TTCs -- not too important and usually
   * around 506 counts. */
  range = dirhighest - dirlowest;
  if (delhighest - dellowest > range)
    range = delhighest - dellowest;

  return;
}
```

4 Evaluation & Summary

From my experience, these algorithms work well. On first glance at fig. 3, it would appear that using a parabola to judge ambiguous events would be inadequate, given the large spread of the TTC lines. However, the only precision ever required of a parabola is that it identifies whether a pulse is in region I or region III.

Fig. 6 gives an indication of how well the algorithms work. It is data taken from an active channel over 10 run files (run1555.dat - run1564.dat). Only pulses with an amplitude above 1500 ADC counts are plotted, with the delay given by the process class. There are 2709 events in this plot. 11 events had to be discarded because the timing was ambiguous, or about 1 per run file. One of the pulses is delayed by about 350 ns, obviously due to a delayed trigger, not a problem with the algorithms. Three pulses come 25 ns early. I do not think it is possible to tell whether these are due to an early trigger or an inaccuracy of the algorithm.

The algorithms presented in this paper are rather long and involved, but are the most efficient I could come up with for getting accurate timing. Without special treatment of the buffer zone of 0.25 ns about the breakpoint (see 2.2, it is not uncommon for points to be shifted by an artificial 25 ns. Figure 7 is the same plot as fig. 6, but without using the buffer zone. This time, there are more points off of the main pulse.

The number of inaccuracies is also reduced by finding the breakpoint and pedestals for each file. For the runs used for figs. 6 and 7, the breakpoint varied by 0.1 ns; fig. 4 shows a variation of more than 0.5 ns. With a buffer of 0.25 ns above and below the breakpoint, this is significant. It may be possible to increase the size of the buffer zone and use the same breakpoint for every run. This would increase computing time as more parabolae would have to be fit.

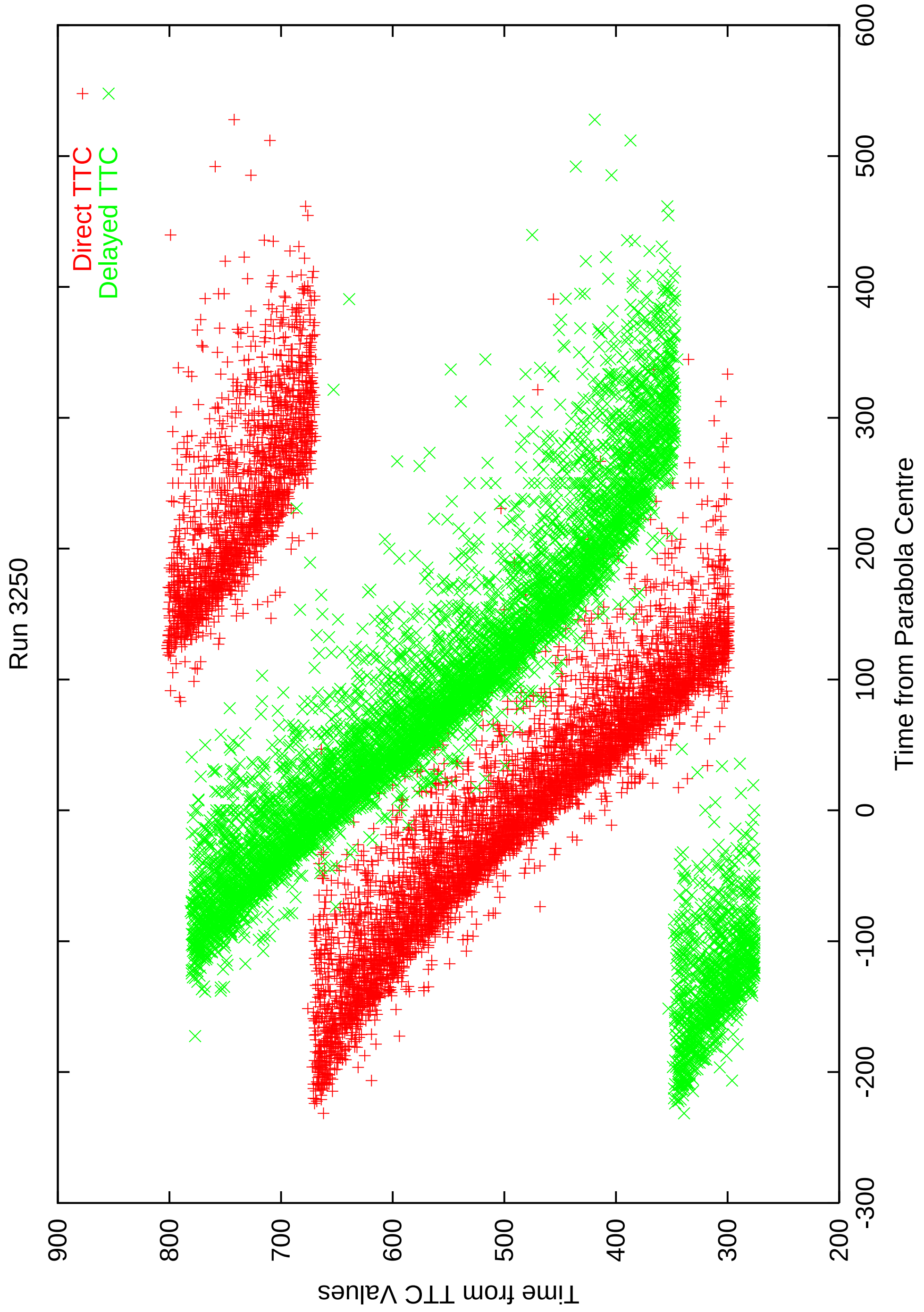

Figure 3: The TTC values plotted against a time determined from a parabola drawn through the highest three samples of the pulse.

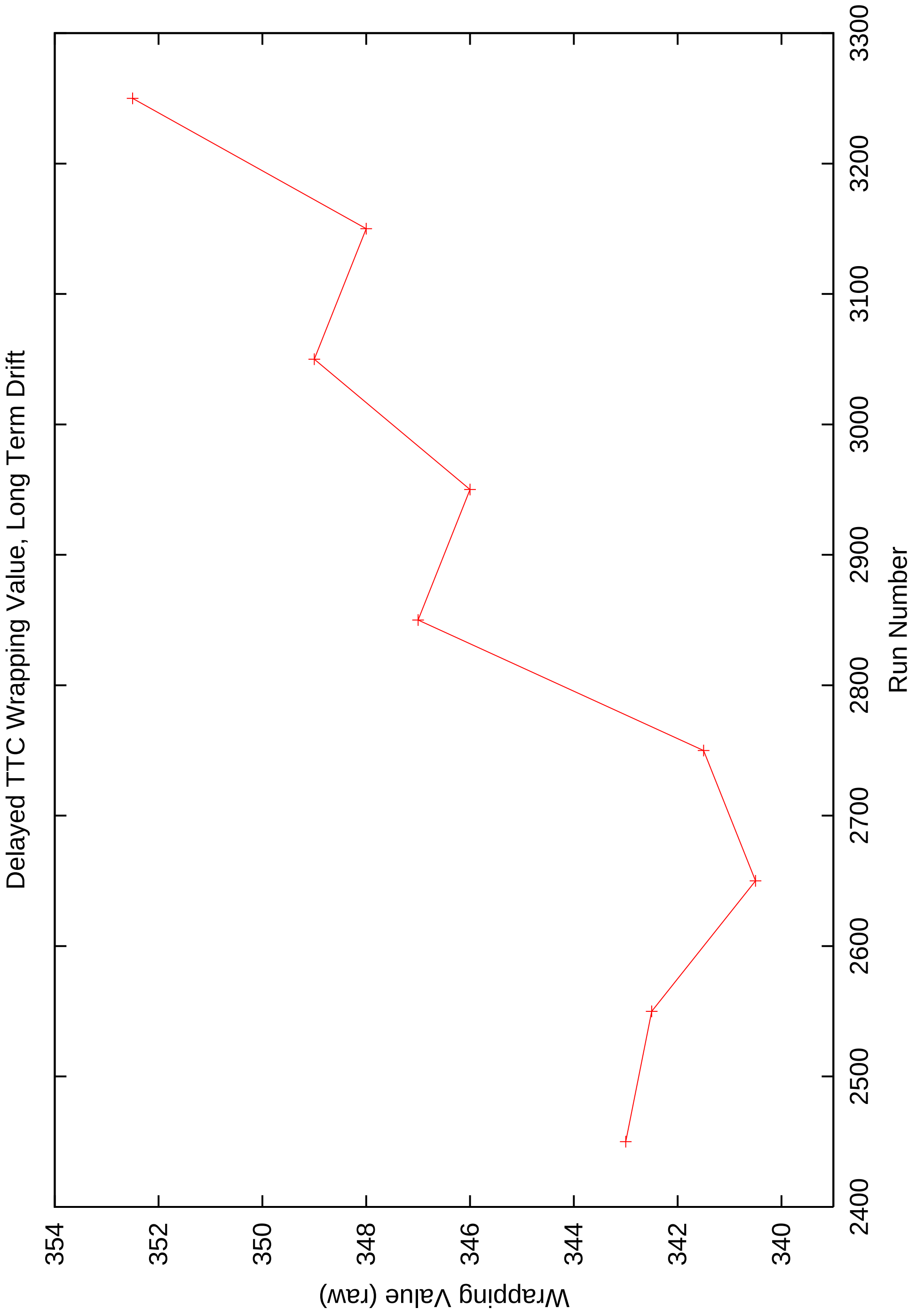

Figure 4: This plot shows how the breakpoint varies significantly from run to run.

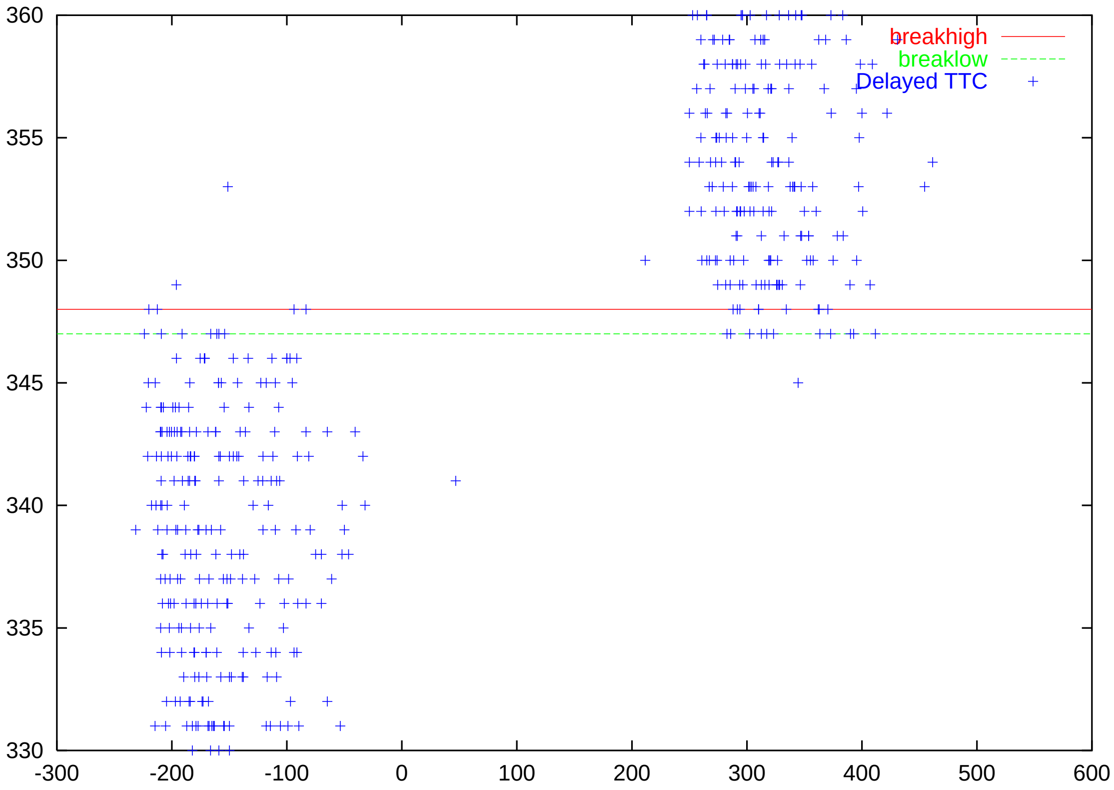

Figure 5: A close-up of the region where the delayed TTC wraps.

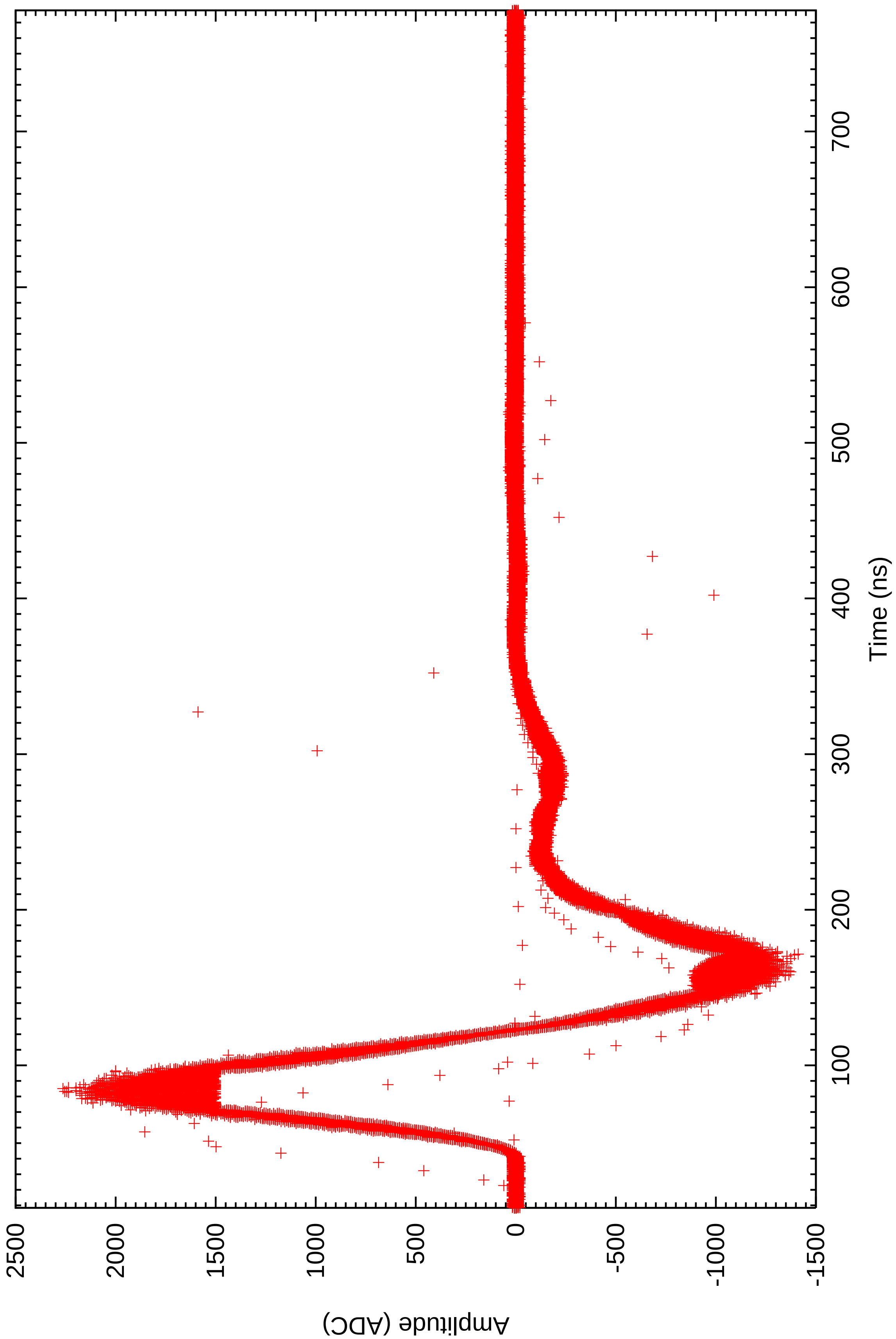

Figure 6: Over-layed pulses, plotted using the algorithms described in this paper..

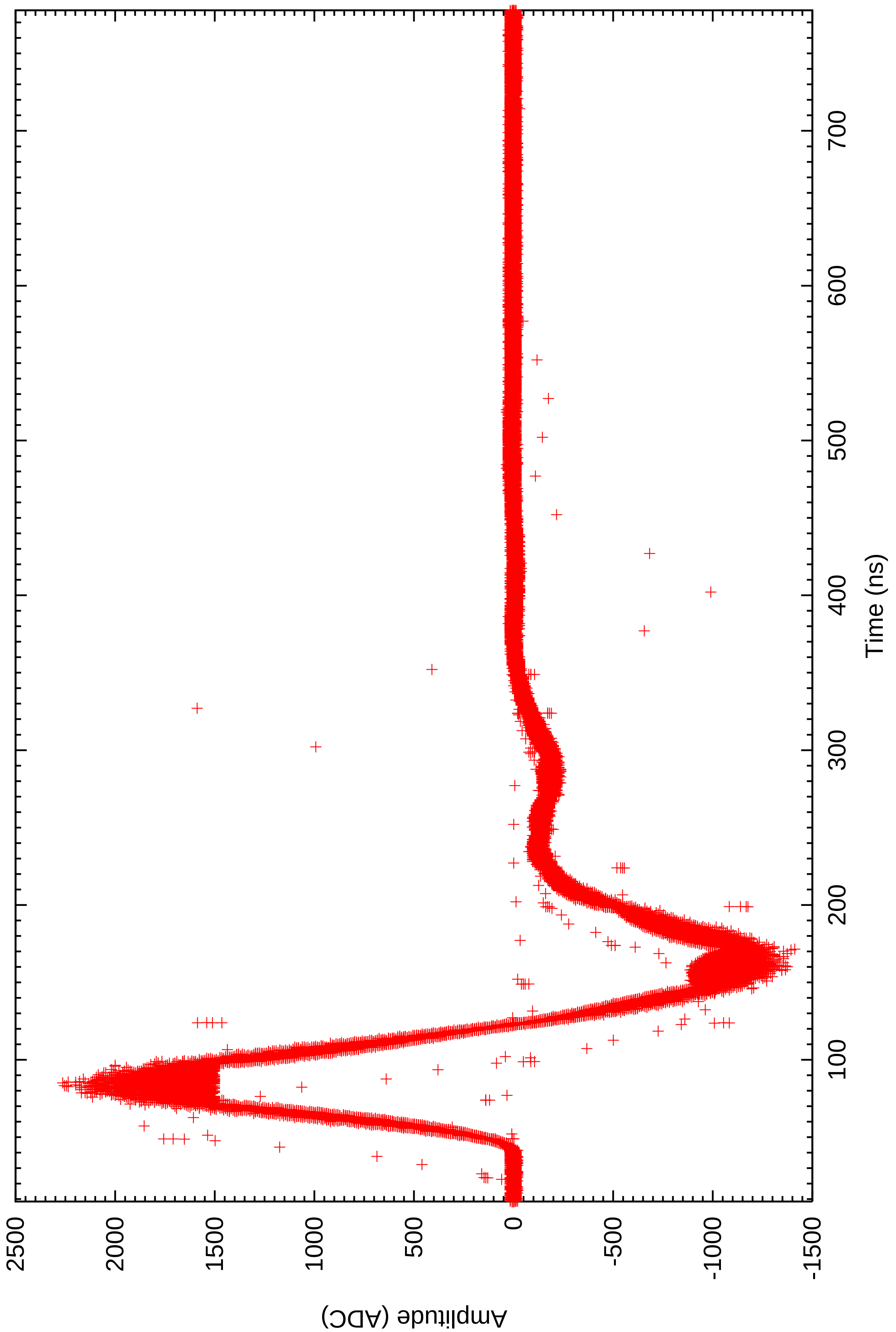

Figure 7: The same plot as in the previous figure, but without considering the buffer-zone around the breakpoint.