Approximating linear time with finite count clocks

Pedro Fonseca

Abstract – **In computer systems, actions are triggered, not only by external events, but also by the passing of time. In the case of real-time systems, these actions must be executed under strict time constraints. Deciding on what actions to take depends on the correct result of operations like comparing time instants. These are complicated by the fact that clock counters have a limited life span. If care is not taken, their wrap-around may present problems and limit the operating life of a system. We present a simple method to circumvent this problem, based on a counter that has a life span of twice the larger interval of interest. This method is easily and efficiently implemented by using the type cast facilities of languages such as C.**

I. Time and clocks in computer systems

Computer systems interact with their environment. In many cases, the start and stop of computer actions are triggered, not by any user interaction, but rather by the passing of time. To execute an action constrained by time, computer systems must contain a device to perceive time: an internal clock. This clock is used by the computer system to take decisions to start and stop the execution of tasks, in order to respect its time constraints. We find here two notions of time. On one side, there is the time that controls the evolution of events in the computer environment. This time is external to the computer system and corresponds to the physical quantity: we call it *physical time*. On the other side ("at the same time", one could say) there is time as perceived by the computer system. This is the time that commands the start and stop of the computer internal actions. We call this *clock time*.

Physical time is (at least, for all practical purposes...) a linear time. We can represent it by a straight line (fig. 1), and events are points in that line. The line is infinite. In our perception of the Universe, time has always existed and it will continue to exist. The notion of an instant (or a time...) after which there would be no more time makes no sense to our minds. Every point in the time line is reached once and only once. Before we have reached that instant, it belongs to the future; for one instant, it is present; after we have reached it, it will be past and it will be no longer reachable (an idea with tremendous implications in our life and psychology).

Dep. de Electrónica, Universidade de Aveiro, P3810-193 AVEIRO, Portugal, E-mail: pf@det.ua.pt, Tel. +351 234 370984, Fax: +351 234 381128, URL: http://sweet.ua.pt/~pf

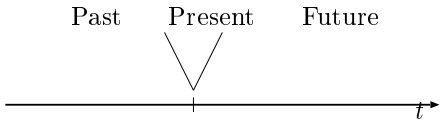

Figure 1 - Linear time

We generally consider time to progress from left to right in a horizontal line. Taking some point to be the present, the past is on the left side, the future is on the right (no political consequences should be inferred from the preceding statement...). We use notions like *happened before* or *happened after* since tender age. We can assign a time mark to every event, and these can be used to establish a sequence of events. We define an ordering of events using only their time marks: the sequence of events corresponds to the sequence of time marks.

All clocks in a computer system are based in the same principle: an oscillator produces periodic events, the clock *ticks*, and a counter is incremented at each clock tick. The value stored in the clock is clock time. Because it is based on a counter, clock time differs from physical time in two fundamental aspects. First, clock time is discrete, whereas physical time is continuous. In mathematical terms, measuring time with a clock is an application C, defined by $C : \mathbb{R} \to \mathcal{V}$, where $\mathcal{V}$ is a discrete set. We represent physical time by the letter t, and clock time by the letter c. $c = C(t)$ is the clock value that corresponds to physical time t. For sake of simplicity, and w.l.o.g., we will consider that physical time and clock time are measured using the same units and that clocks are perfect (there is just a quantization error and no change of scale from physical time to clock time).

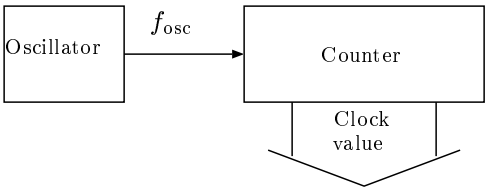

Figure 2 - Model of physical clock

The second difference is that, whereas physical time is infinite and unbounded (at least, to our perception), possible values for clock time are finite and bounded. In mathematical terms, $\mathcal{V}$ is a finite set. We call $M = \#\mathcal{V}$ the clock counter modulo. In many cases $\mathcal{V} = \{0, 1, 2, \ldots, M-1\}$. The count starts at 0 and,

when it reaches $M-1$, the clock value returns to the initial value at the next count, restarting the cycle. This means that clock time is a *circular* time (fig. 3): any point in a circular time is reached repeatedly, over and over (given enough time for the counter to wrap around). Unless the oscillator is stopped, this will go on forever and ever (once again, our notion of never ending time...). This corresponds to a notion of a time represented, not by a straight line, but by a circle. A circular time presents a point of singularity: the clock's value increases as time elapses except in the point when the counter overflows. At this point, a sudden decrease of size $M-1$ occurs.

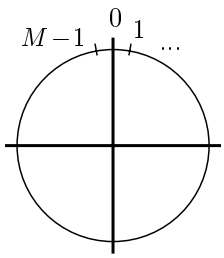

Figure 3 - Circular time

Common uses of clocks in a computer system are to measure the time elapsed between two instants, to determine whether some time instant belongs to the past or to the future or to know how much time we have left before a given instant. These can be generally referred to as *comparing* time instants.

The fact that $\mathcal{V}$ is a finite set (and that, by this reason, clock time is a circular time) introduces some significant differences between clock time and physical time. First, all computations in clock time are performed in modulo M. Secondly, whilst in physical time, the order of events can be unambiguously derived from the instants of their occurrence, the same is no longer true in clock time. A smaller clock value can correspond to a later instant. Moreover, given two clock values $c_1 = C(t_1)$ and $c_2 = C(t_2)$, even if we know their order of occurrence, there is an infinite number of possible values for the difference between them. We want to measure the amount of physical time elapsed between two instants, given the respective clock time. Every value in the set

$$\mathcal{S} = \{x : x = c_2 - c_1 + kM, k \in \mathbb{Z}\} \qquad (1)$$

is a possible solution for the time that elapsed from c_1 to c_2. Obviously, only one of them is correct. But, without further information, we have no means to identify the correct solution.

II. The assumptions

The previous section presented some differences between physical time and clock time and how these cause some ambiguities in the comparison of clock time instants. Namely, we have seen that when trying to compute the time elapsed between two instants, we find an infinite number of solutions and that, without some further information, we are unable to identify the correct solution. In this section, we will present the assumptions that will remove these ambiguities.

In order to do this, we introduce the concept of *interval of interest*. This means an upper bound on the difference between two (physical) time instants that we want to compare, using the corresponding clock time values.

This is used in the assumption:

Assumption 1: The length of any interval of interest is smaller than half the modulo of the counter.

This guarantees that our clock will not complete a full turn during any interval we want to measure. Moreover, it allows us to choose, between the set of possible solutions, the one that corresponds to the smallest arc in the circle of time. In numerical terms, it corresponds to the solution with the smallest absolute value.

Lemma 1: The correct solution for the difference between two clock time values, amongst all numerically possible solutions, is the one with the smaller absolute value.

Any other solution would span over an interval that is larger that half the counter modulo, thus contradicting assumption 1. This means also that values of k in eq. (1) are restricted to the set $\{-1, 0, 1\}$. Note, by the way, that, although c_1 and c_2 belong to $\mathcal{V}$, the result does not necessarily belong to $\mathcal{V}$.

III. Sign conventions

Many programming languages, such as C, allow us to define the value stored in a register (which can be a counter) as signed or unsigned. This distinction goes down to the microcode level: most microprocessors distinguish between signed and unsigned operations in their instruction set. In a 8-bit register (a char in C language), the modulo M is $M = 2^8 = 256$ and every value is in the range $\{0, 1, ..., 255\}$. This is the convention for unsigned char values. For signed quantities, the representation is in 2's-complement. The values from 80_h[1] to FF_h represent negative values: 80_h represents -128, FF_h represent -1. If we consider a clock counter with 8 bits, an unsigned char counter would display values from 0 to 255 and then 0 again, and so on. A signed one would display values from -128 to 127, then -128 again, and so on. But, although the significance we assign to the stored values is different in the signed and unsigned case, the actual bit values in the counter are identical. The difference between signed and unsigned is a mere convention.

The problems with comparing clock time values occur when the singularity point lies in the interval between the two values (*i.e.*, when the interval of interest contains the singularity point). Where does this singularity happen? If the counter value is unsigned, it will happen when the count reaches 255 (in a 8-bit

[1] We will use hexadecimal notation to represent the bit pattern stored in the counter register, and decimal notation to represent the value as we read it

counter): the next value will be 0. But, to a signed value counter, the same transition poses no problem. It corresponds to the counter going from -1 to 0. The same applies to the counting in the antipodes. A signed counter will wrap around in the transition from 127 to -128. For the unsigned counter, this is just going from 127 to 128. All clear, all subtractions work fine. Note that the singularity points for each value convention are half the counter modulo apart.

As we have seen before, what's physically in the counter is independent of whether a counter is signed or unsigned. That is just related how we convene to use the count to represent a value. Second, the same increment can be awkward in one representation (the count decreases 255 when it should increment 1) and totally normal in the other.

So, to avoid the problems caused by counter wrap-around, we just need to choose the representation that eliminates the singularity in the count. Having restricted the larger interval to less than half the maximum count, we know that it can cross the singularity point for one, and only one, of the value conventions.

IV. The algorithm

To implement the algorithm, we consider the full range of the counter divided into four quadrants, 1 to 4 (corresponding to an unsigned count) (fig. 4). The singularity points are in the frontier of quadrants 4 and 1 (for unsigned counting) and of 2 and 3 (for signed counting).

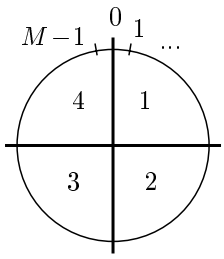

Figure 4 - 4 quadrants

Lemma 2: Comparisons between values in the same quadrant pose no problem.

Lemma 3: Comparisons between values in quadrants 1 and 2 or 3 and 4 pose no problem.

We are left with the problem of comparing values in quadrants (4,1) and (2,3). It is now obvious that:

Lemma 4: Comparisons between values in quadrant 4 with values in quadrant 1 (or *vice-versa*) yield correct values if signed values are used.

Lemma 5: Comparisons between values in quadrant 2 with values in quadrant 3 (or *vice-versa*) yield correct values if unsigned values are used.

To implement the algorithm, we arbitrarily choose a default behaviour (signed or unsigned). The algorithms just needs to check if the values are in the range that contains the singularity point. If this is the case, the alternative behaviour is selected. Otherwise, the computations are performed in the default behaviour. Figure 5 presents the algorithm. In this case, the default behaviour is unsigned. The singularity is in the frontier of the 1st and 4th quadrant; if the values fall into this range, the signed behaviour is selected.

{Default behaviour is unsigned}.
{y is a signed value}
if $x_1 \in Q_1 \cup Q_4$ and $x_2 \in Q_1 \cup Q_4$ **then** {compute using signed values}
  $y = (\text{signed})x_1 - (\text{signed})x_2$
else {compute using unsigned value}
  $y = (\text{unsigned})x_1 - (\text{unsigned})x_2$
end if

Figure 5 - Algorithm for comparing clock values

An implementation is presented in fig. 6, using a char type counter as example. The default for char values is signed. The extension for counter sizes larger than the char type should be adapted to the target processor, depending on the word size in the processor, memory and data bus. Note that, for large sized counters, like 32-bit wide and more, we do not need to use the whole counter for comparisons. Starting with the most significant byte (or word, or ...), a bound-and-branch approach will significantly reduce the comparison time.

```
/*
  compare(x1,x2)

  Calculates the difference between x1 and x2
*/
char compare(char x1,char x2)
{
  char y;

  /* Test if x1 and x2 are in the upper quadrants */
  if( (x1> (signed char)0xC0 || x1 < 0x40) && \
      (x2> (signed char)0xC0 || x2 < 0x40)){
    /* Calculate using signed values */
    y  = x1 - x2;
  }
  else{
    /* Default behaviour */
    y = (signed)((unsigned)x1-(unsigned)x2);
  }

  return y;
}
```

Figure 6 - Example in C code

V. Conclusions

Comparing time instants is a frequent operation in real-time systems, which is some times made difficult by the fact that clock counters have a finite life span. We have presented a method to perform comparisons of time instants using clock values that circumvents these difficulties. The method is based in providing the system with a clock that has a life span larger that twice the maximum interval of interest. The algorithm and an implementation in C language were presented.