Measurement and visualisation of software timing properties

Nick Merriam[1] and Andrew Howes[2]

[1] Gliwa GmbH, 35 Windmill Lane, York
nick@gliwa.com
[2] Manchester Business School, University of Manchester
Andrew.Howes@mbs.ac.uk

Abstract. We observe that programmers in the hard, real-time domain experience severe difficulties when confronted with software timing problems. A number of different visualisations of timing behaviour appear to have large benefits but the specific effects and their significance have not yet been studied with any rigour. Rather than a report of proven results, this is a call for suggestions and contributions to potential, future investigation of the phenomena involved.

1 Introduction

In the study of programming, relatively little attention has been given to the timing of software. This is easily justified, since timing is generally ignored in software development and the focus is entirely on functional correctness: computing the right result. However, there are domains where software timing is critical and there are some special difficulties for developers who have to solve timing problems. In this text, we look at how those difficulties arise and how a number of solutions are evolving to try to address them. Such solutions have, to date, been largely unguided by explicit usability analysis and present an interesting opportunity for HCI.

In particular, we focus on the software timing analysis and visualisation tool offered by Gliwa GmbH, called ***T1***. ***T1*** is used in the development of hard, real-time embedded software, primarily in the automotive sector. Software whose usefulness depends not only on functional correctness but also on guaranteed timely performance is described as hard, real-time software[1]. The term embedded refers to the fact that the computers involved have little or no user interface for the end-user and are embedded within larger devices, such as cars or washing machines.

A distinctive characteristic of ***T1*** is that it is developed by its own, heaviest users, giving advantages and disadvantages over more conventional development. On one hand, it eliminates the gap between user and developer. On the other hand, the users with the greatest input into the design are users with a lot of inside knowledge. The result is that more typical users are often unaware of features for which the affordances are weak or non-existent.

As such, this is not a report of results but rather the demarcation for a potential new area of research. Naturally, there exist analyses that can be related to the study of software timing; this is not an entirely new domain. However, the purpose of this paper is to invite contributions and insights to an open problem rather than to disseminate a solution.

2 Understanding software timing

2.1 Ignoring timing

One of the achievements of high level languages, compilers and operating systems is that they have abstracted away from software timing. Instead, they are designed to make it easier to concentrate on functional correctness. The complex nature of software means that it has historically been plagued with functional defects and the ever advancing nature of hardware means that software timing probably does not matter.

Consider an example of adding two integers:

[1] The qualifer *hard* has been added to the term real-time to distinguish such systems from other (soft) real-time applications such as video decoding, where there is an intention to perform in real-time but no attempt is made to guarantee performance

```
a := a + b;
```

The compiler might translate this high level language statement into a single machine instruction to add two registers together. Or it might generate additional instructions to check the result for an overflow. It might very well generate code to fetch the values from memory before the addition and to write the result back to memory afterwards. One of those fetches might require memory to be paged in from disk and cause a task switch while waiting for the disk to deliver the page. For most kinds of software, the programmer needs to concentrate on *what* is being computed and should not be giving too much thought to the immense potential variation of *how* the compiler and OS achieve the result, as long as it is the correct result.

If the programmer does want to understand how long their software takes to execute, their case seems to be an almost hopeless one. In the addition example, we considered only one statement but even very simple applications consist of thousands of such statements. And even for the simplest case with the addition statement, if we suppose the compiler does generate a single machine instruction to add two registers, it is still very difficult to know how long that one instruction will take to execute. With older, simple processors, you used to be able to look up in a big table to see how many cycles each instruction would take. With modern processors, they are far too complex for such an approach. One of the registers to be added might still be waiting to be loaded with an integer conversion from a floating point value, where the fast but currently overloaded floating point unit has a queue of operations to perform before it can do the conversion to integer, leading to a long delay. Or the instruction might be scheduled in a pipeline "bubble" so that it seems to take no time at all to execute. In order to predict the execution time for the simplest of programs, a detailed model of the processor and memory interfaces is required together with computer software to compute the timing within that model.

On the one hand, most software can be developed perfectly well with only the loosest grasp of its timing properties. On the other hand, hard, real-time software has to react within a short time and with high reliability if it is to be of any use at all. Consider the engine controller of a modern car. Unless the amount of fuel to inject, the time at which to inject it and the time to fire the spark are all calculated both correctly and on time, the performance of the car will be worse than with a carburettor. With increasing sophistication of electronic devices from household goods to aircraft, real-time functions that used to be entirely performed with hardware now involve more and more embedded software. The demand for embedded, hard, real-time software is currently exploding.

Developers of hard, real-time software are faced with an acute problem. They can only be effective by reusing the high level languages and processor core designs that have evolved for mainstream computing, where timing is very hard to predict and has been intentionally hidden from the programmer. However, hard, real-time developers need to be able to provide high assurances about the timing properties of the software that they develop.

The pragmatic solution widely adopted has been to mostly ignore the timing issue. Experience has shown that worrying about timing at the wrong stages of software development will likely damage the all-important functional correctness. The approach is highly remeniscent of the infinitely fast machine myth described in [1]. The thorough testing already performed in the name of functional correctness ought to show up any timing problems that do arise. And given a few careful guess-timates at the start of a project, the specified hardware is probably fast enough to run the software in the available time.

2.2 Being forced to consider timing

Life for the developer of embedded, hard, real-time software therefore becomes rather exciting when testing shows that there are, after all, timing problems. Embedded timing problems generally manifest themselves as some kind of failure to respond within a deadline. Either some watchdog or sensor outside the processor notices the lack of responsiveness or a control algorithm fails because of intolerable timing variations. Having built a complex system whilst

carefully ignoring timing, the programmer is now required to try to understand a lot of new information very quickly and without much framework or support.

To see just how unfortunate the situation is, consider the problem solving process.

Goals The goals are poorly articulated and typically only expressed in two forms:
  - The software should be "fast enough".
  - Almost invariably, periodic functions have to complete before they become due again.[2]

Operators Because hard, real-time software is still a niche area of programming and because many projects do find that the software is fast enough, the operators for repairing timing problems are neither widely-known nor well-practised. Happily, profiling and optimisation are techniques that apply also to general computing, not just the hard, real-time domain, so these two techniques can be attempted. Unfortunately, standard profiling tools do not work in embedded environments because they have excessive run-time overheads and generate data too quickly to be transmitted from the embedded device to an external file-store. Also, some general optimisation techniques turn out to be counter-productive for hard, real-time performance.

Methods Decomposing the already vague goals is virtually impossible. If we know that a sequence of two functions should execute in at most one millisecond, should we allocate 500 microseconds to each? That runs the risk of wasting half the processing power, if it turns out that one function takes almost no time at all. So, at least we would like to get some idea of how long each function takes. But the functions can be preempted, so how much of the one millisecond should be allocated to them and how much to preemptions? Academic research into scheduling has provided a range of techniques for decomposing timing goals but, for a number of reasons, they have achieved almost no widespread adoption.

Selection Given the problems with goals, operators and methods, the developer may well find they can only select one available operator, which is optimisation. They can experiment with compiler optimisation, which may or may not help. They can also try to examine the code for inefficiencies and try to improve those parts. In any case, the optimisation will not be accurately focussed. It is very difficult to achieve good results without being able to decompose the problem and to choose from a wider range of techniques.

2.3 Glossary

To emphasise the complexity involved, and to facilitate comparison with other texts on software timing, we present a short glossary of measured timing terms. Each of these kinds of timing information apply to each task, where by task we mean each independently scheduled, sequential software component. They are shown pictorially in Figure 1 and defined in Table 1. Of course, not all these values are independent but there are still more timing properties that are not listed here because they are not measurable. For example, a deadline is not a measurable property but rather an upper limit on the response time. Even though the deadline is often equal to the period, they do not have to be the same and the deadline may be either smaller or greather than the period.

3 Measuring software timing

Given the many difficulties described above, the programmer's first approach to debugging software timing problems is to try to measure what is actually happening. Given that the real behaviour differs from the programmer's expectation, she wants to find out how actual behaviour diverges from the intended behaviour. A good start is to monitor context switches, where the processor stops executing one task and either switches to another task, or handles

[2] Deadline is equal to period.

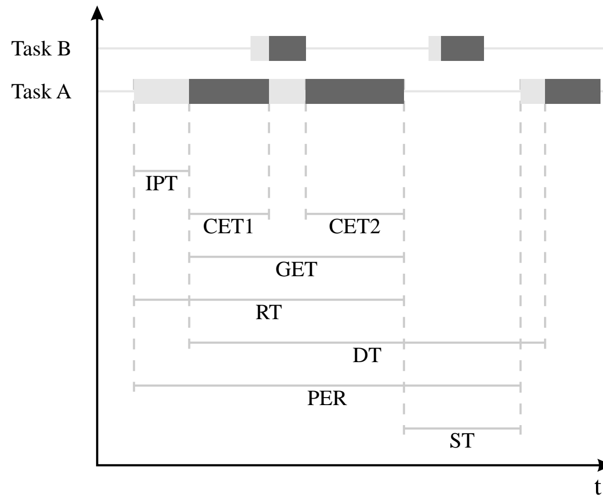

Fig. 1. Kinds of timing information for each task.

Table 1. Kinds of timing information for each task.

IPT	Initial Pending Time is the time between the task becoming due and actually being started.
CET	Core Execution Time is the time required to compute a task on a processor core. It is independent of scheduling as it does not include time for preemptions. In scheduling literature, it is denoted C.
GET	Gross Execution Time is the real time required to execute a task. It is the CET plus time for preemptions.
RT	Response Time is the real time between an event and the software response demanded by that event. Response time includes GET plus IPT.
DT	Delta Time is the real time from one start of the task to the next. It is not the same as the period because of variation in IPT.
PER	PERiod is the regular interval between times the task becomes due. In scheduling literature, it is denoted T. It is normally an input parameter rather than a measured value but observing the period can be useful, for example when a configuration problem causes a task to have an unexpected period. Aperiodic tasks, also called sporadic tasks, often have a non-zero minimum period, which can be a useful property to verify through measurement.
ST	Slack Time is the time between a task finishing and becoming due again.

an interrupt. Only by recording every context switch can we determine which task or tasks are consuming which time. This can then be extended by creating what we call a “stopwatch”, a user-defined pair of start and stop events that measure arbitrary time intervals. For example, where a task consists of a number of functions, we might create a stopwatch to measure one of those functions that is suspected of taking a long or highly variable time to execute.

One approach would be to carefully attribute each segment of time to each task or interrupt handler and to thus determine the CET for each task, interrupt handler and user-defined stopwatch. By measuring the periods and delta times, we can determine the rate at which the tasks run. This can be used to construct a model of the system, using which simulation or schedulability analysis can detect certain kinds of timing defect. However, we have experienced a number of difficulties with such number-crunching approaches:

- Observed timing values vary continuously as they are repeatedly measured. For example, the same task might vary between 0.5ms and 1.5ms on a number of executions. This requires models that either omit significant information or become complicated and potentially difficult to use.
- When a particularly long or short CET value is measured, the programmer invariably asks what else was going on in the system at that time. This question cannot be answered from the digested numbers.
- The numerical data is just not very appealing. Although numerical analysis elegantly distills large amounts of measured data into a small number of values, those values do not seem to be particularly good triggers for diagnosing faults.

4 Visualising software timing

An alternative approach is to attempt to draw a diagram of sofware events, progressing along a time line from left to right similar to that shown in Figure 1. This approach has been adopted very widely and is based on the well-established oscilloscope display. Actual ***T1*** screenshots are shown in Figure 2 and Figure 3.

Anecdotal experience shows that visualising actual software timing behaviour with pictures provides huge benefits. Commonly, the first such picture uncovers a number of surprises for the programmers involved. Not all of these are necessarily timing defects, in the sense that they may be within tolerances of the system design. Even these show that the programmer’s assumptions about the real-time system had been too restrictive and lead to a better understanding. When timing bugs do arise, they almost always show up as unusual changes of shape in the picture, to which the human visual processing system is highly tuned.

There are a number of significant challenges with visualising software timing. One is the sheer volume of data. If we monitor only the start and end of five tasks, each of which has a 10ms period, then one second of system execution generates a million events. Let us leave aside for a moment, the problems of storing this much data on a small, embedded device and then transferring it to a computer with a suitable, graphical display. A picture consisting of a million distinct events cannot be rendered on a normal screen and, even if it could be, it would be incomprehensible. Figure 2 contains just 11 events and Figure 3 contains around 600 events. It is hard to imagine effectively viewing more than 10,000 events at one time.

If the timing defect manifests itself continually, then we can just capture any sequence of 10,000 events and, with some careful zooming and scrolling, find interesting information about the nature of the problem. However, if the defect appears once every second, we have to try to catch the right 10,000 events out of one million, which would be painfully time-consuming if we were to randomly sample until successful. Fortunately, we can often identify a symptom of the timing problem which can be used as a trigger to capture trace data exactly when “something interesting” happens. Even if this is not entirely accurate, such triggers massively increase the value of the data that we choose to visualise.

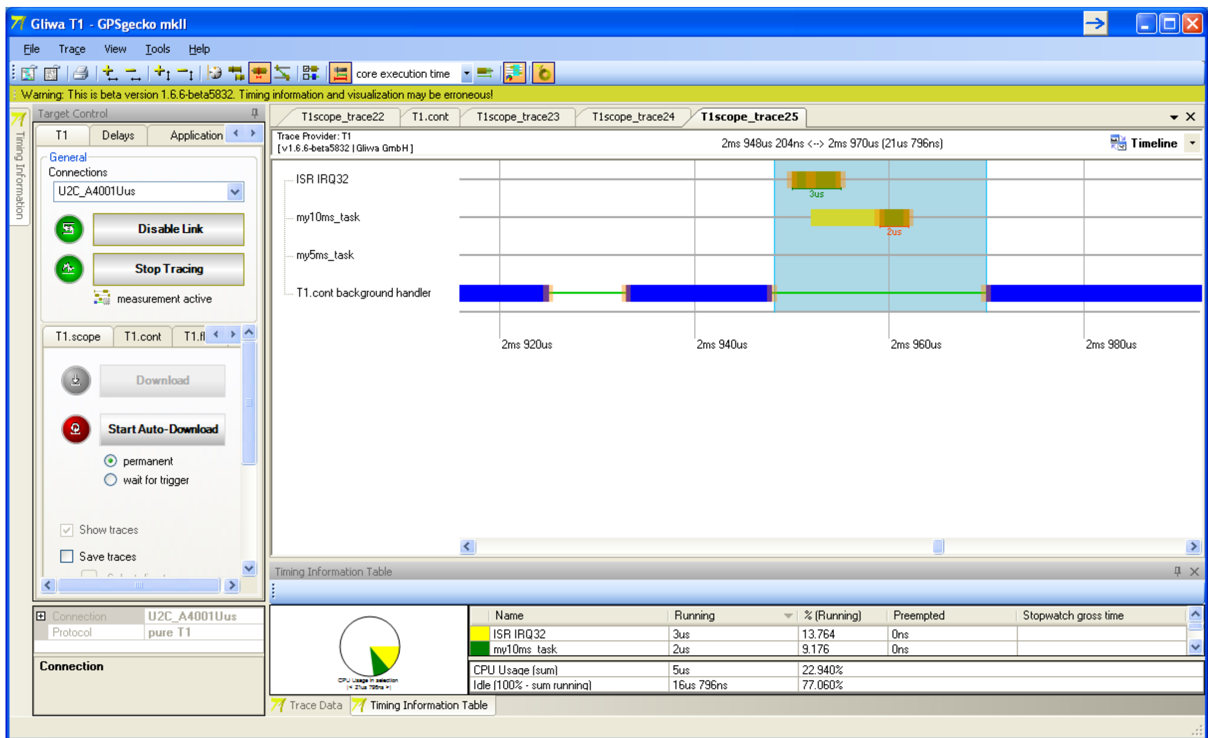

Fig. 2. Oscilloscope style display at high zoom with few events.

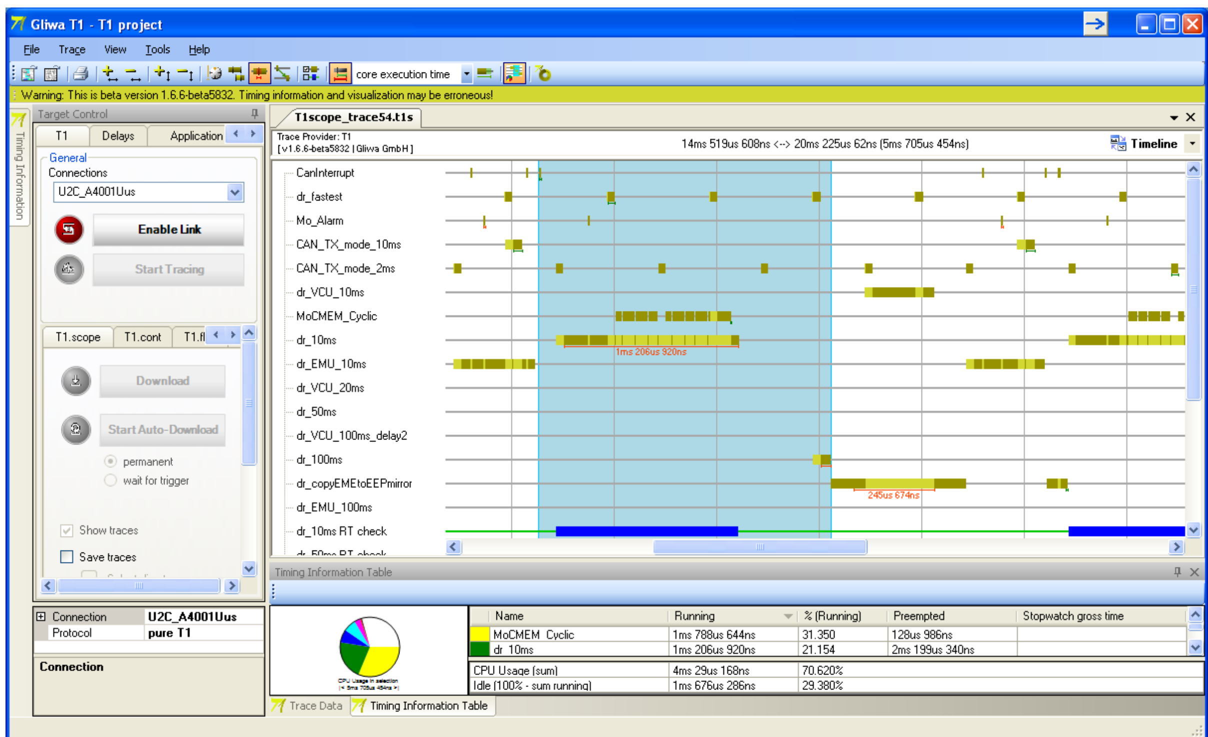

Fig. 3. Oscilloscope style display at low zoom with many events.

Having captured a relatively small number of traces likely to contain clues to timing problems, the challenge is now to find and interpret those clues. It seems that this is very much easier to do in the visual domain, since longer than usual times or changes of sequence show up as unexpected shapes in the picture, to which the eye is drawn.

Visualisation by no means replaces numerical analysis. In fact we can combine triggering with numerical analysis, triggering on analysed timing properties as they arise, increasing the value of triggers still further.

Furthermore, as shown in Figure 4, we can also visualise the distilled numerical analysis. In this display, the length of the horizontal bars corresponds to the magnitude of the calculated number and the darkness of the background behind the numerical display shows the recency of the result. Consider the example of a calculated maximum CET value. A new maximum, indicating a timing behaviour never seen before, causes the horizontal bar to grow to the right and the numerical background to suddenly darken. This quickly attracts attention to this new value. If we then want to see that trace of behaviour that gave rise to this value, we set a trigger to capture a trace the next time that CET is calculated. Then we search the picture for interesting events and patterns of events leading up to the large CET value.

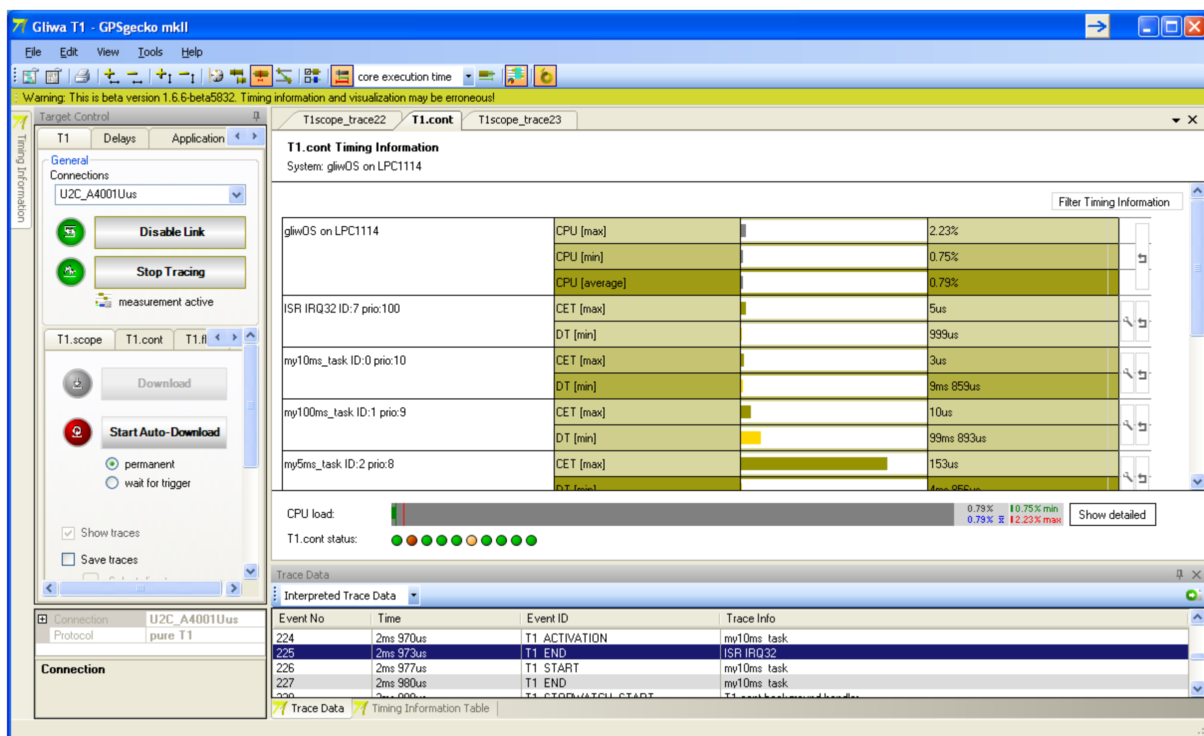

Fig. 4. Numerical analysis display.

5 Solving timing problems

The process of solving a timing problem with ***T1*** might be as follows. Suppose a task does not complete by the next time it becomes due, at which time the operating system raises an error. This is a fairly common first indicator of a timing problem. The first step would be to trigger on detection of the OS error and to view the trace of events leading up to the error.

It might be that the task is being delayed by other processing and not scheduled with sufficient urgency. This kind of defect is relatively easy to detect in a timing picture. The task

priorities can be adjusted to better meet the real-time requirements of the application before testing the system again to see if the OS error and ***T1*** trigger ever occur.

Alternatively, we may see that there is no reshuffling of priorities that can help the task to complete before its deadline. Idle time appears as gaps on the time line when no task is ready or running and so the timing picture makes it clear when there is simply not enough idle time. Also, the numerical analysis will show that the CPU load is high, with a very high transient CPU load near the errors. In this case, the only solution is to do less computational work. We might be able to optimise the code so that the same functionality is performed with less computation, using techniques such as those described in [4]. If that does not yield sufficient improvement, we might have to accept slightly less functionality. For example, a common solution to the challenge of controlling a fast-spinning combustion engine is to compute control parameters only once every two revolutions at the highest speeds.

In selecting the functions to optimise, or reduce, ***T1*** offers user-defined stopwatches that can be used both for visualisation and for numerical analyses. This allows timing goals to be decomposed: if a task has too high a CET, or has too great a variation in its CET, we can try to pinpoint the chief causes by using stopwatches to monitor one set of functions at time. If the top level function calls 8 helper functions, we can easily surround each helper function with a stopwatch. We might discover that 6 use very little time at all and we can then focus our attention on the remaining 2, possibly quadrupling the benefits from a given amount of analysis effort.

6 Benefits of visualisation

Informally, we believe that using a timing visualisation tool like ***T1*** greatly increases the productivity of a programmer required to debug timing problems. However, there is no systematic evidence for this. And apart from intuitive observations that visualisation makes unusual timing behaviour appear as eye-catching changes of shape, we have yet to do any real study of the effects at work and their significance.

There have certainly been previous studies of visualisation of important and complex timing, such as [3], which provide strong indications of how ***T1*** might be so beneficial. The uncertainty in medical treatment times handled by the visualisations in [3] corresponds closely with the variations in software timing behaviour observed in ***T1***. However, we are not aware of studies of visualisation of timing where programming is involved and where the timescales are milliseconds or microseconds and well outside the realm of human perception.

We also note the visual data does some of the work of remembering the system configuration and behaviour to allow the programmer to focus in on just one aspect which the tool remembers the larger picture. The benefits of such phenomena have certainly been studied, see [2].

7 Summary and future developments

The software tracing approach that we have described is, in essence, extremely simplistic and amounts to logging context switches plus user-defined events and drawing them as a picture. This approach is augmented with numerical analysis and simple triggering to make sure that rare behaviours are detected and can be captured within a window of visualised data that amounts to a tiny fraction of the whole data stream. Indeed, the approach has to be rather simple because anything more ambitious would disrupt the very timing behaviour we are trying to observe by adding excessive, on-target overheads.

Even such a limited approach seems to have large benefits, also a conclusion of [3]. There is an opportunity to investigate this effect and to make a significant contribution. The small, reactive nature of Gliwa GmbH means that ***T1*** can evolve rapidly, not only to incorporate proven improvements but also to prototype experimental features for research purposes.

The rapid spread of multicore processors into hard, real-time domains means that the timing is becoming significantly more complex. With a dual core processor, we not only have twice the rate of events but also entirely new timing effects that arise when the cores synchronise with each other. It seems that the relevance and value of tools like ***T1*** is going to increase in the near future.

In summary, the visualistion of software timing is a relatively new technology that offers significant benefits but has not yet been systematically investigated. There are opportunities to understand the effects and, very likely, to significantly improve this technology at a time when the recognised value of that technology is rapidly increasing. We invite observations, suggestions and potential collaboration at the contact addresses provided above.

References

1. A. J. Dix. The myth of the infinitely fast machine. In D. Diaper & R. Winder, editor, *People and Computers III — Proceedings of HCI '87*, pages 215–228. Cambridge University Press, 1987.
2. Edwin Hutchins. How a cockpit remembers its speeds. *Cognitive Science*, 19:265–288, 1995.
3. Robert Kosara and Silvia Miksch. Visualizing complex notions of time. In *PROCEEDINGS OF THE CONFERENCE ON MEDICAL INFORMATICS (MEDINFO 2001)*, pages 211–215. IOS Press, 2001.
4. Class Embedded Systems and Jakob Engblom. Getting the least out of your C compiler, 2001.