State of the Art and Open Research Topics in Embedded Hard Real-Time Systems Design

Wolfgang A. Halang

FernUniversität, Faculty of Electrical and Computer Engineering, Germany

The most important and necessary properties of embedded real-time systems, and the ways to achieve them, are explored. The basic and most prominent domains of real-time systems design are discussed, starting with processor and system hardware architectures, ranging over operating systems, tasking and scheduling, high level real-time programming languages, worst case execution time and schedulability analysis, to application design engineering and safety related applications. Common misconceptions, which are still strongly present in control systems design, are identified. Guidelines for consistent implementation are proposed, and sample solutions in certain areas presented. Finally, directions for future research are indicated.

1. Introduction

More and more computers can be found controlling common everyday utilities, such as microwave ovens, washing machines, video recorders, all kinds of industrial processes and, increasingly, car subsystems. They are often called embedded computer control systems. By definition, they operate in the hard real-time domain, i.e., they must be permanently ready to respond to requests from their environments within pre-determined time frames; hence their other names, responsive or reactive systems. Failing to meet the temporal requirements has the same consequences as if their functional behaviour would be wrong.

With its first applications dating back to the 1940s [24], some thirty years ago the discipline of real-time systems began to gain wider interest. All pertaining topics received considerable attention from computer scientists all over the world as well as from most major domains of computer science. The fundamental guidelines for their design have changed from fast to fast enough: it is of utmost importance that systems meet their pre-set deadlines, which is not guaranteed by mere speed of control systems themselves. To be able to achieve that, determinism and predictability of the temporal behaviour of program executions is necessary: these properties essentially imply other requirements.

Unfortunately, although undisputed among scientists, the state of the art in implementing practical applications is still far away from meeting — or even proving — these requirements: in general, program execution and system response times are usually unpredictable and not consistently analysed. Hence, the state of affairs in performance verification of control systems is

Hope and Pray!

The main reason for this sad situation is that the (commercial off the shelf) hardware and software commonly employed in control applications is optimised towards high average performance, whereas it is necessary to consider the worst case behaviour if the meeting of deadlines is to be guaranteed. In execution time analysis, necessarily a certain amount of pessimism is involved, diminishing estimated average performance considerably.

In order not to employ more powerful hardware platforms due to such pessimistic estimations, designers tend to develop and test control applications in ways not much different from the ones employed for non-real-time systems, resulting in the risk that deadlines will be missed. The consequences can be severe in many cases, like massive material losses or even endangerment of human safety. This is a situation that must be overcome. The gap between academic research results and practical implementations must be removed in order to provide for much more consistent and safer computer control systems.

This paper is organised as follows. First, the basic properties of real-time systems will be summarised. Then, some domains explicitly involved in the design of embedded systems will be considered: hardware architectures, operating system issues, languages, compilers, and worst case execution time analysis. A special section is dedicated to safety related systems, bearing in mind that most control systems are safety critical to a certain extent. Although interesting and relevant to some more sophisticated computer control systems, higher level issues like more advanced software engineering topics, real-time data bases or artificial intelligence, will not be dealt with here.

A word on the references cited in this paper: deliberately not the latest, but the original references on certain topics are quoted. This way, we intend to show when these topics were addressed first and where the general solutions can be found (cp., e.g., papers on scheduling [16, 11]). The development of research is thus revealed, and it becomes obvious that the wheel was re-invented many times.

2. Basic Properties of Embedded Real-Time Systems

Real-time operation is an operating mode of a computer system in which programs for processing data arriving from outside are permanently ready, so that their results are available within pre-determined periods of time. The arrival times of the data can be randomly distributed or be already a priori determined depending on different applications [5]. Although functionally correct, results produced beyond the pre-determined time frames are wrong.

In computer control systems, there is a distinction between hard and soft real-time requirements. Whereas in soft real-time cases the penalty of missing deadlines increases with time, in hard real-time cases overdue results are useless with any consequences that may bring. The general optimisation criterion in soft real-time matters is average performance, while in hard real-time environments it is necessary to consider the worst case behaviour. Most control systems, however, are hybrid: hard and soft real-time tasks usually co-exist in applications with, as a rule, hard real-time tasks executing with higher priority, and soft real-time tasks running within the performance reserve of the former.

There are two different comprehensions of predictability: layer-by-layer (microscopic) and top-layer (macroscopic) predictability [20].

Layer-by-layer predictability requires each of the layers considered in real-time application design to behave deterministically and predictably, thus providing the necessary basis for the next layer. This approach is suitable for embedded control systems with moderate complexity. It has been shown that their temporal behaviour can be guaranteed a priori [14, 8].

Top-layer predictability only considers the behaviour on the highest (application) layer, and provides handlers to deal with the cases when deadlines are missed. This concept is suitable for sophisticated and complex applications, where it is impossible to deal with time in detail. However, in embedded systems, especially those with high integrity requirements for safety critical applications, it is necessary to strive for microscopic predictability.

Beside timeliness and predictability, an important property of control systems is dependability. The hardware part of these systems has already reached a high degree of dependability; methods and techniques for their verification have been well established for a long time already.

In this sense, some extremely dependable hardware platforms were developed some time ago. An example thereof is the VIPER, a simple microprocessor for safety critical applications [13]. Its design has been formally specified and mathematically rigorously proven correct (despite rumours on an error in its model.)

Dependability of software, however, is still lagging far behind. The reason for this phenomenon is its inherent complexity. Unfortunately, software engineering has not made much progress in this specific field, yet. For the time being, the most important principle to be followed, when designing a system with severe dependability requirements, is *simplicity*.

Simple solutions, however, are the most difficult ones to attain. They require high innovation and complete intellectual penetration of the issues involved:

Progress is the road from the primitive via the complicated to the simple [1].

Easy understandability is the most important precondition to prove the correctness of a system.

3. Hardware Architectures

In common state of the art control computers, either programmable logic controllers (PLCs) or generic conventional microprocessors are used. The former are cyclically repeating control algorithms. Since the iterations must terminate within certain periods, temporal predictability is intrinsically guaranteed. However, static state observation-based operation is less flexible, and does not match the inherently dynamic nature of most environments controlled. Often, the paradigm of dynamic event observation must be implemented.

Commercial of-the-shelf microprocessors are optimised Commercial of-the-shelf microprocessors are optimised for best utilisation in the average case. This, however, contradicts the hard real-time optimisation criterion, viz., the worst case performance.

Processor utilisation itself is a thinking category of the early days, now being obsolete due to the achievements in solid state technology and, consequently, the wide availability and low prices of hardware components. Sub-optimal utilisation is a cheap price to be paid for simplicity and dependability.

3.1. Undesirable Properties of Conventional Architectures

Undesirable properties of conventional architectures are those which make a system's temporal behaviour (1) unpredictable or/and (2) difficult to estimate. A common counter-productive characteristic of their design is excessive complexity, which renders their verification difficult or unfeasible with justifiable effort.

The increase in microprocessor performance through the years was mainly influenced by two factors, viz., technological advances and new ideas employed in architectural design. Such new ideas were, e.g., the RISC philosophy, parallel processing, pipelining and caching. While RISC ideas are also very suitable for architectures employed in hard real-time systems because of their simplicity and the implied verifiability, pipelining and caching (although always present, especially in RISC processors) present a hazard to determinism and predictability of temporal behaviour.

Independent parallel operation of internal components. In order to fully exploit inherent parallelism of operations, the internal units of processors are becoming more and more autonomous resulting in highly asynchronous operation. By this development the average performance of processors is increased. However, it is a complex task to analyse corresponding sequences of machine instructions in order to determine their execution times. For the time being, verification of such complex processors and their program execution behaviour is practically impossible.

The most common way to improve the throughput of a processor by exploiting instruction level parallelism is pipelining. Various techniques are used to cope with the well-known pipeline breaking problem due to the dependence of instruction flow control on the results generated by previous instructions. They are, however, all based on the optimisation of average performance.

To predict execution times, complex machine code analysers would be needed, requiring detailed information about instruction execution, which is often proprietary and, thus, inaccessible. Owing to its complexity, a processor with pipelined operation is also very difficult to verify.

Caching. Fetching data from off-processor sources represents a traditional bottle-neck, especially in the von Neumann processor architecture. To avoid it, cache memories were introduced. Since instruction fetching from memory consumes a considerable amount of execution time, the latter thus highly depends on whether an instruction is found in cache or not. To predict this, complex analysers would be necessary, emulating the caching operation. Even the most sophisticated ones would fail in the case of multiprogramming, where caches are filled with new contents on every context switch.

The worst case consideration is, thus, to always count on cash misses and to design applications in a way that deadlines are met anyway — if this is possible, however, caches are not needed!

Similar to processor architecture considerations, in classical system architecture design some features were implemented to improve average performance. Again, these features may lead to undesirable consequences which make process execution time prediction difficult or even impossible. Some of them are listed in the sequel.

Direct memory access. Since general processors are ineffective in transferring blocks of data, direct memory access (DMA) techniques were devised. With respect to the delays caused by DMA transfers, there are two general modes of DMA operation, viz., cycle stealing and burst mode. A DMA controller operating in the cycle stealing mode is literally stealing bus cycles from the processor, while in the other mode the processor is stopped until the DMA transfer is completed. Although the processor has no control over its system and context switching is disabled during data transfer, this mode is more appropriate when predictability is the main issue. If block length and data transfer initiation instant are known at compile time, the delay can be calculated and considered in the program execution time estimation. Furthermore, block transfer is faster, because bus arbitration is required only once.

Data transfer protocols. Data transfer via microcomputer busses also deserves some consideration. Synchronous data transfer protocols by definition ensure predictable data transfer times. Asynchronous ones are more flexible, however, their behaviour is very difficult to control, especially in shared-bus systems.

In the case of multi-master busses, access conflicts and, consequently, arbitration is another issue that could jeopardise the temporal predictability of program execution. It has to be performed so that even in a case of heavy traffic it is assured that the bus is granted to each requester within a certain time frame.

Similar conclusions hold for local area networks as well. For distributed control systems appropriate network protocols with deterministic and predictable behaviour must be chosen. The widely used CSMA/CD (Ethernet) is — like many others — inappropriate because of non-deterministic resolution of collisions. A specific implementation problem is also the complexity of standard protocols: for instance, the standard document DIN 19245 on the field bus Profibus comprises some 750 pages.

Dealing with hardware interrupts. The problems discussed in the previous paragraphs could either be prevented, or the measures potentially causing non-deterministic or unpredictable behaviour could be renounced. However, in process control computers operating in the dynamic mode, i.e., immediately responding to events in their environments, interrupts are unavoidable. Since these events occur asynchronously to program executions, the latter are necessarily delayed. This problem cannot be solved in classical single processor architectures.

A possible solution is to employ a dedicated asymmetrical multiprocessor architecture, in which one processor is handling such events as signals from the environment, time events, changes of synchronisers etc., and performing functions of the operating system kernel [7].

When such an event occurs, an associated task is put into the ready state and all ready tasks are re-scheduled. All this takes place in parallel to the execution, on another processor, of the running task without interrupting or pre-empting it.

Knowing the (residual) execution times of application programs, the run-time schedulability analyser checks whether there exists a feasible schedule of the new set of tasks. If so, the tasks are re-scheduled accordingly.

Apart from provision for predictability, also the inherent parallelism of task execution and operating system administration routines is exploited by this concept, thus considerably improving systems performance. Also, the counterproductive overhead of context switching is minimised.

This solution is known for 15 years now, and was implemented in several academic prototype platforms (e.g., [2, 19, 3]), while no reports were found on industrial implementations.

4. Operating Systems, Tasking and Scheduling

The domain of real-time operating systems and related issues was well covered by both academic research and industrial implementation of commercial systems. Hence, in this section only a coarse overview of the domain is given.

According to nature, scale, purpose, implementation etc., in embedded real-time control systems different design alternatives are pursued.

In smaller applications real-time executives are sufficient, providing basic functions such as task and time management, interrupt and error handling. Often, these functions are custom designed and incorporated into the application program code. In very small applications, there is even no need for that: the program itself handles control algorithms, input/output data exchange and event administration.

In larger scale and complex control applications, usually complete operating systems are employed: they can be either Unix-like (e.g., QNX, Real-Time Unix, Real-Time Linux) or other commercial systems (e.g., VRTX). For the former, standardisation has been achieved by the POSIX standard.

Common functions of real-time operating systems for embedded applications are [22]:

- interrupt handling: lowest level routines, direct hardware support;
- process management: multiprogramming is supported by the concept of processes and tasks, which are supervised by process and task management;
- inter-process communication and synchronisation: means for interaction between asynchronously operating processes and tasks;
- time management: administration of time, actions on processes and tasks at specified instants of time;
- input/output drivers: support for communication with peripheral devices;
- man-machine communication: support for the interaction between system and operator;
- file management: not obligatory in embedded systems;
- memory management: storage administration supporting robustness and safety of control systems (As virtual memory is jeopardising predictability by varying access times to data residing in memory or on mass storage media, it is obsolete due to the very large semiconductor memories available today.);
- error handling: necessary for robust fault tolerant systems; its implementation must allow for the consideration of delays in error situations;
- exception handling: one of the most severe obstacles to the predictability of execution times of tasks; a good solution is to handle exceptions off-line, in parallel on a dedicated processor (see above).

Application programs controlling the processes in embedding environments are organised in form of tasks. Tasks embody application programs whose worst case execution times are analysed and thus known. An important parameter of each task is its deadline, usually expressed as an interval from the task's invocation until the instant when it needs to be concluded. If not, the deadline is violated with consequences depending on whether the task is a hard or a soft one. In the former case the task has failed, the results are meaningless, and the system is in a fault condition.

Computer control systems can be of either static or dynamic nature. The operation of static systems is controlled synchronously by programs, i.e., it can be prepared in advance. There are no interrupts or other events. Tasks are scheduled at design time, allowing to analyse, test and verify such systems a priori. Hence, they are much safer and more robust than dynamic systems.

However, in most cases it is not possible to assume the behaviour of a system in advance. Then, it is necessary to consider dynamic events at run time. Verification of such systems is much more demanding.

In dynamic multi-tasking systems, scheduling algorithms capable of generating appropriate schedules must be implemented. The ones which guarantee to fulfill the requirement that all tasks meet their deadlines (provided the system's performance is sufficient) are referred to as feasible. In the literature, several such algorithms were reported. Actually, scheduling of real-time tasks is a very popular research area, although the most adequate basic policies are known since the early 1970s. The issues considered in the research on scheduling policies include resource contention, precedence relations, synchronisation, maximum system utilisation, prevention of deadlocks, relationship between soft and hard real-time tasks, task priority issues, overload handling, fault tolerance and, last but not least, a priori or run-time schedulability analysis.

The usual means to introduce asynchronous dynamic events into microprocessor-based control systems are interrupts. Handling them by generic microprocessors is, as a rule, priority-based, and this policy can usually be found in real-time executives, too. However, it is not possible to guarantee schedulability [21] of a set of ready tasks by assigning static priorities to them. Although this technique is temptingly simple and broadly used, it is highly inappropriate for hard real-time applications.

In dynamic systems, tasks can be activated periodically or sporadically. A suitable scheduling policy for periodic tasks is rate-monotonic scheduling [16]: the task with the highest frequency of invocations is assigned the highest dynamic priority.

The situation generally prevailing in industrial real-time data processing, however, is that at any point in time there is a number of runnable tasks competing for the assignment of a processor, regardless whether the tasks are aperiodic in nature or the cyclic incarnations of periodic tasks. This task state is entered by explicit activation, continuation, or after releasing synchronizers. For scheduling such ready task sets, the most suitable policy is the earliest-deadline-first scheduling algorithm: the task with the closest deadline is executed first (a modified version is described in [2]). It was shown to be feasible and optimal on single processor systems [11]. With the so-called throw-forward extension it is also feasible on homogeneous multiprocessor systems. However, this extension leads to more pre-emptions, is more complex and, thus, less practical. It is characteristic for the state of the art to note that the two major scheduling methods, viz., the first-come-first-served and the static priority-based disciplines, which are widely supported by contemporary high level languages and implemented in commercially available real-time operating systems, are both not feasible.

When the due dates and execution times of tasks are a priori available, using simple necessary and sufficient conditions [11], one is able to detect whether a ready task set given at a certain instant can be executed meeting the specified deadlines. Thus, the problem of optimum single processor scheduling for real-time applications has been already fully solved by the earliest-deadline-first algorithm. This scheme is characterized by an impressive list of advantageous properties [8], which make it extremely well-suited for industrial practice. This scheduling scheme can be extended to not fully preemptable tasks retaining all the advantageous properties just at the insignificant price of reduced maximum processor utilisation [9].

The practical relevance of scheduling tasks on homogeneous multiprocessor systems is rather low, as in process control applications the process interfaces are usually physically hard wired to sensors and actuators establishing the contact to the environment. Thus, it is natural to implement either single processor systems or dedicated multiprocessors.

5. Programming Languages and Worst Case Execution Time Analysis

Real-time applications on generic computer systems can be programmed using one of the following programming languages:

- Programmable logic controllers are programmed using a variety of vendor specific programming means; there is a good and well established standard IEC 61131-3.
- Assembly languages: unfortunately, a considerable part of applications based on generic microprocessors is still programmed in assembly language resulting in much more errorprone programs and less productive application program development.
- Implementation languages: the general purpose languages C or C++, FORTRAN, Basic, Modula (already with support for tasking) are commonly known and compilers are readily available; however, they lack support for most important real-time features, like scheduling, process hardware access or multitasking.
- Genuine real-time languages:
  – Universal: Ada, LTR, PEARL
  – Proprietary: Atlas

5.1. High Level Real-Time Programming Languages

A high level programming language to be useful in a real-time environment should fulfill the following properties:

- It should support the predictability of temporal behaviour of program executions. Most of the commercially available and broadly used languages do not fully meet this requirement. It is thus left to the programmer to use for the sensitive parts of an application only those constructs which do not jeopardise predictability, to analyse and, at run time, supervise the temporal behaviour of the program.
- It should be secure to allow for reliable programs. This can be achieved by strong type checking and provisions for exception handling.
- Support for multiprogramming should include tasking and good synchronisation concepts.
- As peripheral device drivers are very common and important parts of real-time applications, easy access to non-standard hardware is required.
- Programming in the large is to be supported by modularity and the possibility for separate compilation.
- As real-time software has a long life expectancy, its maintainability is to be fostered by easy readability, understandability, and modifiability.

High level programming languages for programming generic computer systems used in control applications can be divided into two categories: implementation languages and real-time languages.

Implementation languages do not provide much support for real-time programming. Application programmers have to call certain library or operating system kernel routines, which provide the real-time capabilities to a certain extent. By far the most common among implementation languages is C or C++. Its advantages are good availability, connectivity with, and support for, program development tools; most programmers are familiar with it. The disadvantages are poor readability and maintainability, and the lack of almost any of the above mentioned features required in real-time programming.

For programming real-time applications, proper real-time programming languages ought to be used. There are not many such languages commercially available. Many of them were either academic prototypes or proprietary languages of certain large companies or institutions.

Two well established and broadly used real-time languages are mentioned here, Ada and PEARL.

Ada is probably the most widely used genuine real-time language. It has a number of good properties including most of the ones mentioned above. One major disadvantage, however, is that it is very large and complex. Including almost any feature of all other modern languages, it is, thus, hard to learn and use, and tends to produce inefficient code.

PEARL [6] was designed and standardised by a group of German scientists in the early 1970s and was re-defined in 1998 (PEARL90). It also fulfills most of the above requirements: it is well understandable, and much simpler and closer to the control engineer than Ada. Unfortunately, it is not broadly used outside of Germany and Europe, and does not have much support by commercial program development tools.

In recent years, object orientation is also entering the real-time programming domain. The classical programming languages are being adapted to this paradigm, and new ones are emerging. The one to be mentioned in this context is Real-Time Java. Although the common Java is originating from the process control domain, there are still some obstacles to its consistent applicability, the most serious one being the temporal impact of garbage collection. Moreover, the current draft of the real-time extensions to Java reveals that the design committee lacks knowledge on other real-time languages.

5.2. Industrial Controller Programming

Vendors of programmable logic controllers usually provide their own programming tools. Most of their properties were incorporated into the standard IEC 61131-3 [12] with the goal to unify program design in this application area. They may be grouped into five categories:

- Instruction List: a low level assembly-like language;
- Ladder Diagram: logic functions expressed in a form of relay logic;
- Function Block Diagram: standard and commonly used routines are represented by blocks which are then "wired" together;
- Structured Text: a structured high level textual language;
- Sequential Function Chart: graphic representation of programs structured in form of Petri nets, including parallel and alternative execution.

While the first two are merely meant for compatibility with some older systems, the latter three represent powerful tools for application design. They could even be used for programming generic control computers: standard procedures (e.g., some 70 as defined for the control of chemical processes in the guideline VDI/VDE 3696 [23]) are programmed, verified, and placed in a library. Application programmers only need to invoke them and to "wire" them together for specific applications. This way, the complexity of designs could be greatly reduced, and the verification of programs composed out of standard program blocks would turn extremely simple.

5.3. Worst Case Execution Time Analysis

For schedulability analysis, execution times of programs must be known in advance. This is only possible if the language provides for predictability of the programs' temporal execution behaviour. Unfortunately, no existing programming language guarantees that. It is, therefore, necessary to introduce certain restrictions:

- GOTOs are not allowed: their use can result in unstructured code being difficult to manage. Instead, EXIT and LOOP statements should be introduced. The former is used to leave innermost structures, and the latter is used in loops for immediate initiation of the next loop iteration. As a consequence, labels, except for procedure and task declarations, are obsolete and therefore renounced.

- Each loop block must be tightly bounded: lower and upper counts of iterations must be present and defined by compile-time-constant expressions, so that the longest execution time of a loop can be estimated. The "while" or "repeat" conditions should be replaced by explicit IF-EXIT statements within loops.

- Pointers and recursion are not allowed: their use can result in severe memory management problems. They can produce temporally non-deterministic actions, and cannot be considered in a priori timing analysis.

- Each synchronisation construct must be temporally bounded: synchronisation constructs (for example critical regions or semaphores) may take arbitrarily long times for their execution. This must be bounded in real-time systems. Each such command must be temporally guarded, and an explicitly defined action must be given for the case that a time-out occurs.

- Explicitly asserted execution times: in some cases, estimation may yield very pessimistic execution times. To resolve this problem, additional execution information must be given by the programmer. To this end, new constructs (pragmas) may be inserted into the program code, as proposed in [17, 18]. However, this method requires complex analysis, and is not feasible in all situations. To overcome this problem, the explicit execution time of a part of the code can be asserted by the system developer, overriding the estimated results with the asserted ones. However, for safety reasons, such a block must be guarded by time-out control, and a time-out action must be present.

- Task scheduler support: the scheduling algorithm in the operating system kernel processor relies on the residual execution time of tasks. This parameter is computed from the maximum execution time of a task minus the accumulated running time. However, the actual execution time is expected to be shorter than the estimated one. To achieve a more reliable and realistic estimation of the residual time, it can be updated explicitly, whenever possible, by asserting the information about the residual execution time which is gradually more precise as the task approaches its end.

To estimate the execution times of programs written in high level programming languages, the following three steps are necessary:

1. The HLL source code is parsed and analysed for constructs such as straight line code blocks, alternatives, condition evaluations, loops etc.

2. The execution time of each straight line portion of assembly language (or machine) code is calculated. In further analysis, the straight line blocks are not considered any more.

3. The nested structures in the compiled source code are searched recursively until the innermost ones are located. These are then reduced by their worst case execution times: alternatives are reduced to the execution time of the longest one; execution times of loop bodies are multiplied by the maximum number of iterations; subroutine calls are reduced to the corresponding execution times. Then, the same procedure is performed on the next enclosing nested structure. Finally, the recursive application up to the outermost structures yields the execution time of the program.

6. Safety Critical Applications

It appears to be easier to assure dependable hardware architectures than dependable software. The reason is that verification methods and techniques for hardware designs are well established for quite a long time. Hardware can be safety licensed for critical applications by licensing authorities (like TÜV in Germany) using these procedures.

Software is much more complex, taking into account the whole life cycle, from specification, design, coding and compilation into machine code which is actually the final result. There is a number of hazards to its integrity, from the common faults and errors in specifications and in all levels of design, to implementation problems like arithmetic precision (cp. the Ariane 5 disaster), to mention just a few.

Licensing of software-based systems is extremely difficult. Although considerable research efforts have been invested into the area of formal specification and verification of programs, the techniques are generally applicable to relatively simple cases, only. Also, they are not formally accepted by licensing authorities, since they themselves rely on complex computer tools which are inherently unsafe.

The only method officially recognised by German licensing authorities is diverse back-translation [15]. It consists of re-gaining a requirements specification for a software under investigation by several, independently working licensors or groups. To eliminate possible effects of error-prone compilers, this process must be based on the machine code read out of the target system. It is obvious that this method is only feasible for very limited applications like, e.g., emergency shut-down systems. It is not usable for most industrial applications.

For that reason, in extremely critical applications and systems, like safety backup systems in nuclear power plants or avionics, where safety is crucial and formal licenses are necessary, usually only hardwired or trivially simple programmed systems can be licensed for the time being. However, for economic reasons, it is necessary to provide methods and techniques to safety license also program-based systems of reasonable complexity. Therefore, some alternative approaches to control system design were elaborated [10], allowing to rigorously prove correct systems with realistic complexity involving only reasonable effort.

7. Main Topics of Future Research

In this section, we want to identify and shortly discuss a number of topics, which are or will be, in our estimation, the major areas of research activities on real-time systems.

Conceptual foundations of real-time computing. Academic real-time systems research has to be based on a solid, realistic model derived from the application domain incorporating the thinking categories and optimality criteria as outlined above. Computer science has no well developed concept of time. As a matter of fact, time even appears to be systematically suppressed. Therefore, it is necessary to reflect on the rôle the time is playing. A clear understanding of the reasons why and the manner in which time is involved in the design of real-time systems is needed as a prerequisite for a sound methodology. We expect that the systematic exploration of common sense notions about time and of analogies with everyday life solutions for time related problems will yield principles for the design of real-time systems.

Utilisation of domain specific knowledge. In general, real-time systems research must be much more application-oriented than other areas of computer science. The employed methods are generally process-specific, because the process is part of the control loop closed by and in the computer. This holds, e.g., for overload handling and error recovery procedures, which can be designed by exploiting the processes' inertia and their corresponding typical time constants. A real-time system is subjected to variable time conditions in dependence on the process speed. There may be the possibility for their relaxation in the overload case, e.g., by reducing the speed of a robot arm. Adaptive, self-correcting systems with carefully designed graceful performance degradation behaviour in response to error occurrences can only be constructed by full utilisation of the process characteristics. The comparison of two different designs or systems is, analogously, only possible on the basis of application specific benchmarks — mere MIPS figures do not say anything.

Predictability and techniques for schedulability analysis. A time metric must be introduced to realise the predictability requirement. To achieve temporal predictability and full determinism of system behaviour will be a major effort, in the course of which many features of existing programming languages, compilers, operating systems, and hardware architectures will have to be questioned. To this end, real-time systems must be designed in all aspects as simple as possible, for simplicity fosters understandability and enhances dependability and operational safety. As far as possible, parallelism is to be implemented physically in order to prevent problems.

Requirements engineering and design tools. Over the past decade, there has been a proliferation of formal specification methods that incorporate some notion of time. But while these methods may have some use in verifying qualitative timing properties, they are of little value in reducing complexity. (Electrical) engineers designing real-time systems do not yet have requirements engineering and design tools at their disposal, which are oriented at their way of thinking and which allow them to precisely express all timing constraints they encounter. Absolute timing or temporal supervision of activities, expressed in a system independent language, are still not possible. Graphical methods are best suited to express concurrency, cooperation, and temporal behaviour of real-time systems in a fully predictable way, because they take advantage of the inherent capability of pictures to effectively convey complex information. Moreover, they allow for straightforward formalisation, being a prerequisite of their use for program specification and verification.

Reliability and safety engineering with special emphasis on the quality assurance of real-time software. When developing real-time programs, not only the software correctness, in the sense of mathematical mappings as in sequential processing environments, has to be proven; also, their intended behaviour in the time dimension, and the interaction of concurrently active processes, need verification. Although not fully developed yet, there is already a host of methods available to carry out the former task. Working in close co-operation with the requirements engineering and design tools, analytical methods are required which perform an a priori check if the specified time conditions can be met ("schedulability analysis" [21]). Such new verification procedures must be quantitative and time-oriented, and should utilise the partial synchronisation of tasks implied by the timing constraints.

High level languages and their concepts of parallelism, synchronisation, communication, and time control. The guiding principle for the development of the next generation of high level real-time programming languages should be the support of predictable system behaviour and of inherent software safety without impairing understandability. Language constructs for the formulation of absolute time conditions and for controlling the operating system's resource scheduling algorithms must be provided. The latter feature will enable the compiler to already perform, to a large extent, checks for feasible executability of task sets. New languages should further support the re-usability of modules, distributed software, various time dependent fault tolerance mechanisms, and the programming of PLCs. New, user-oriented, synchronisation methods must be devised and provided, which employ time as an easily conceivable and natural control mechanism. The next generation languages should try to combine the advantages of PEARL90 [6] and Real-Time Euclid [14], and should, for safety purposes, incorporate as many ideas from NewSpeak [4] as practically feasible.

Real-time operating systems. It will be expected of future real-time operating systems that they guarantee the deadlines and precedence relations holding between tasks under observation of fault tolerance measures on the basis of an integrated resource scheduling. The common deadlines of several co-operating tasks have to be met in distributed systems taking the transmission overhead into account. Frequent temporal supervision measures must be taken during program execution to guarantee timely system behaviour or to initiate a graceful degradation of performance. The arrival of tasks ready for execution and requesting resources can no longer be considered as a random process. For the sake of predictability, a more deterministic procedure must be applied, which utilises the information about the future instants when tasks will enter the ready state being available in real-time systems. Thus, future resource conflicts can be detected and possibly resolved at a very early stage. It is expected of a real-time operating system, that it can predict at any point in time, if all active tasks will meet their deadlines.

Distributed, fault tolerant, language and/or operating system-oriented innovative computer architectures. A uniform theory of correctness, timeliness, and dependability is urgently needed as foundation for the design of large distributed and fault tolerant systems. The research into fault tolerance should yield effective, time-bounded methods for error handling and administration of redundancy. The effect of system load on the fault susceptibility of real-time systems has not been investigated, yet. When developing new architectures for real-time computers, the main objective must be the support of programming languages, operating systems and scheduling algorithms, of fault tolerance and time management, as well as of error handling and time-bounded communication. This contributes to increased speed and to narrowing the semantic gap between hardware and software.

Hardware and software of process interfacing. Favourable interconnection topologies and specialised components with inherently low internal data transmission requirements are needed for distributed architectures. For predictability reasons and to meet the applications demands, new process peripherals with accurately user-timed behaviour must be developed in connection with the realisation of time-based synchronisation primitives.

Communication systems. With respect to real-time communication systems, future research should mainly emphasise predictably timely network behaviour, integrated scheduling algorithms for communication channels and other resources, as well as dynamic routing for message transmissions with guaranteed deadlines.

Distributed data bases with guaranteed access times. In order to meet the high speed requirements of distributed real-time data base systems, a maximum degree of parallelism has to be realised for the transaction processing. A theory of the integrated control of this parallelism and of the corresponding resource scheduling is needed, which, at the same time, aims to maximise parallelism and to minimise the worst case transaction processing time under observation of the boundary conditions data consistency, transaction correctness, and meeting the transaction deadlines.

Artificial intelligence with special emphasis on real-time expert and planning systems. In real-time systems, artificial intelligence methods based on heuristic knowledge are mainly applied for the control and scheduling of time-bounded processes. The best possible solution of a problem is to be found within dynamically given time limits. It is an open research problem to develop symbol processing methods observing such time limits in a predictable way. Among others, new storage management techniques, different from garbage collection, need to be devised to this end.

Global cost minimisation. Scheduling has an economical raison d'être: the utilisation of resources is to be optimised in order to minimise costs as the ultimate objective. Therefore, research on real-time scheduling ought to address problems significant to industry and to society at large. The optimum to be achieved is utmost simplicity, which is a precondition to construct highly dependable systems with easily predictable behaviour. Thus, new scheduling algorithms should support the search for low complexity task execution schedules featuring inherent prevention of deadlocks and, minimisation of the need for explicit resource access synchronisation, minimisation of context switches (also to meet the characteristics of RISC processors with their large register banks and caches), and incorporation of inter-task and network communication. Such new algorithms should be accompanied by calculation procedures allowing to determine the processor capacity required to feasibly, i.e., timely, execute a given task set.

When it comes to cost minimisation, the "bottom-line approach" should be taken. This means that only the overall costs of an automation project or a computerised technical process are economically relevant leaving room for many trade-off decisions allowing to reduce the cost of one component if a more expensive other one is employed. Following this approach, one will learn that a processor running idle once in a while leads to great benefits with respect to the costs of other system elements. Here scheduling research can draw on the methods of operations research, i.e., that area of science it emanated from some decades ago.

A new instrumentarium is required to address the most pressing question in the optimisation of real-time computing systems, which constitutes, at the same time, the largest potential for cost reductions: to minimise complexity and cost of software.

8. Conclusion

To achieve the properties which are necessary in the real-time systems domain, the methods and techniques common in all areas of embedded control systems design need to be reconsidered. This consolidation effort must be centred around the quest of fulfilling the timeliness, predictability, and dependability requirements, because they have not satisfactorily been met, yet. The price to be paid for better consistency and, consequently, better safety, usability, availability etc., is just worse processor utilisation. This is easily tolerable in the light of the state of the art in solid state technology, and the availability and prices of hardware components.

Even in extremely safety critical systems, economical reasons require to replace relay logic and hardware-based systems by programmed ones. In their development, however, it is necessary to follow the generally known, but usually neglected, principles of hard real-time systems design. One has begun to realise the inherent safety problems associated with software. Since the utilisation of software in automation systems is growing at a rapid pace, the problem of software dependability will exacerbate severely. Future embedded systems will only be able to meet the demands of society if they will be safety licensable.

The principle guideline for the design of embedded control systems must be simplicity: considering the state of the art in all areas of design, and especially in verification, this is the only way to guarantee consistency of functional and temporal behaviour of embedded control systems.

Real-time systems research must, much more than other areas of computer engineering, be based on and oriented at a profound understanding of the application domain. The employed methods are generally process specific, because the process is part of the control loop closed by and in the computer.

9. Acknowledgements

I wish to thank Professor M. Colnarič for many of his ideas which are included in this paper, and for his continuous co-operation. The work of his colleagues in the Real-Time Systems Laboratory of the University of Maribor, Dr. Domen Verber, Dr. Roman Gumzej and Dipl.-Ing. Stanislav Moraus, for the implementation of some of the ideas outlined in this paper, is highly appreciated.

References

[1] K. BIEDENKOPF, “Komplexität und Kompliziertheit”, *Informatik Spektrum* 17, 82–86, 1994.

[2] M. COLNARIČ, W. A. HALANG AND R. M. TOL, “Hardware Supported Hard Real-Time Operating System Kernel”, *Microprocessors and Microsystems* 18, 10, 579–591, 1994.

[3] J. COOLING, “Task Scheduler for Hard Real-Time Embedded Systems”, Proc. *IEE Intl. Workshop on Systems Engineering for Real-Time Applications,* pp. 196–201, London: IEE 1993.

[4] I. CURRIE, “NewSpeak”, In *High Integrity Software,* C. T. Sennett (Ed.), pp. 122–158, London: Pitman 1989.

[5] DIN 44 300 A2: *Informationsverarbeitung,* Berlin-Cologne: Beuth Verlag 1972.

[6] DIN 66 253-2: *Programmiersprache PEARL 90,* Berlin-Cologne: Beuth Verlag 1998.

[7] W.A. HALANG, *Definition of an Auxiliary Processor Dedicated to Real-Time Operating System Kernels,* University of Illinois at Urbana-Champaign, Report UILU-ENG-88-2228 CSG-87, 1988.

[8] W. A. HALANG AND A. D. STOYENKO, *Constructing Predictable Real-Time Systems,* Boston-Dordrecht-London: Kluwer Academic Publishers 1991.

[9] W. A. HALANG, “Load Adaptive Dynamic Scheduling of Tasks with Hard Deadlines Useful for Industrial Applications”, *Computing* 47, 199–213, 1992.

[10] W. A. HALANG AND M. COLNARIČ, “Outsourcing the Development of High Integrity Software”, in *Software Quality — The Way to Excellence,* W. Wintersteiger (Ed.), pp. 495–501. Vienna: Arbeitsgemeinschaft für Datenverarbeitung 1999.

[11] R. HENN, *Deterministische Modelle für die Prozessorzuteilung in einer harten Realzeit-Umgebung,* PhD Thesis, Technical University of Munich, 1975.

[12] IEC 61131-3: *Programmable Controllers, Part 3: Programming Languages,* Geneva: International Electrotechnical Commission 1992.

[13] J. KERSHAW, *The VIPER Microprocessor,* Royal Signal And Radar Establishment, Malvern, Worcs., Report 87014, London: Her Majesty’s Stationery Office 1987.

[14] E. KLIGERMAN AND A. D. STOYENKO, “Real-Time Euclid: A Language for Reliable Real-Time Systems”, *IEEE Transactions on Software Engineering* 12, 9, 941–949, 1986.

[15] H. KREBS AND U. HASPEL, “Ein Verfahren zur Software-Verifikation”, *Regelungstechnische Praxis rtp* 26, 73–78, 1984.

[16] C. L. LIU AND J. W. LAYLAND, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment”, *Journal of the ACM* 20, 1, 46 –61, 1973.

[17] C.-Y. PARK, “Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths”, *Real-Time Systems* 5, 1, 31–62, 1993.

[18] P. PUSCHNER AND CHR. KOZA, “Calculating the Maximum Execution Time of Real-Time Programs”, *Real-Time Systems* 1, 2, 159–176, 1989.

[19] K. RAMAMRITHAM AND J. A. STANKOVIC, “Overview of the SPRING Project”, *Real-Time Systems Newsletter* 5, 1, 79–87, 1989.

[20] J. A. STANKOVIC AND K. RAMAMRITHAM, *Editorial: What is Predictability for Real-Time Systems. Real-Time Systems* 2, 4, 246–254, 1990.

[21] A. D. STOYENKO, *A Real-Time Language With A Schedulability Analyzer,* PhD Thesis, University of Toronto, 1987.

[22] VDI/VDE 3554: *Funktionelle Beschreibung von Prozeßrechner-Betriebssystemen,* Berlin-Cologne: Beuth Verlag 1982.

[23] VDI/VDE 3696: *Manufacturer Independent Configuration of Digital Control Systems,* Berlin-Cologne: Beuth Verlag 1995.

[24] K. ZUSE, Foreword to the book *Constructing Predictable Real-Time Systems* by W. A. Halang and A. D. Stoyenko, Boston-Dordrecht-London: Kluwer Academic Publishers 1991.

Received: April, 2000
Revised: June, 2000
Accepted: July, 2000

Contact address:
Wolfgang A. Halang
FernUniversität,
Faculty of Electrical and Computer Engineering
D-58084, Germany
e-mail: `wolfgang.halang@fernuni-hagen.de`

WOLFGANG A. HALANG born in 1951 in Essen, Germany, received a doctorate in mathematics from Ruhr-Universität Bochum in 1976, and a second one in computer science from Universität Dortmund in 1980. He had worked both in industry (Coca-Cola GmbH and Bayer AG) and academia (University of Petroleum & Minerals, Saudi Arabia and University of Illinois at Urbana-Champaign), before he was appointed to the Chair of Applications-Oriented Computing Science and became head of the Department of Computing Science at the University of Groningen in the Netherlands. Since 1992 he holds the Chair of Computer Engineering at the Faculty of Electrical Engineering, at FernUniversität Hagen in Germany.

His research interests comprise all major areas of hard real-time systems with special emphasis on safety licensing. He is the founder and European editor-in-chief of *Real-Time Systems*, member of editorial boards of 4 other journals, co-director of the 1992 NATO Advanced Study Institute on Real-Time Computing, has authored 7 books and 1 CD-ROM, edited 11 books, written some 240 refereed book chapters, journal publications and conference contributions, given some 60 guest lectures, and is active in various professional organisations and technical committees as well as involved in the programme committees of some 100 conferences.