Design for Time–Predictability [*]

Lothar Thiele
ETH Zürich
Switzerland

Reinhard Wilhelm
Universität des Saarlandes
Saarbrücken, Germany

April 26, 2004

Abstract

A large part of safety-critical embedded systems has to satisfy hard real-time constraints. These need sound methods and tools to derive reliable run-time guarantees. The guaranteed run times should not only be reliable, but also precise. The achievable precision highly depends on characteristics of the target architecture and the implementation methods and system layers of the software. Trends in hardware and software design run contrary to predictability. This article describes threats to time-predictability of systems and proposes design principles that support time predictability. The ultimate goal is to design performant systems with sharp upper and lower bounds on execution times.

1 Acknowledgements

The participants of the Dagstuhl Perspectives Workshop on *Design of Systems with Predictable Behavior* have greatly contributed to this article. They were Christoph Berg, Uwe Brinkschulte, Jakob Engblom, Wolfgang A. Halang, Reinhold Heckmann, Vesa Hirvisalo, Jozef Hooman, Jan Madsen, Peter Marwedel, Alexander Metzner, Erik Norden, Peter Puschner, Christine Rochange, Martin Skambraks, Jörgen Steensgaard-Madsen, Theo Ungerer, Ernesto Wandeler, and Wang Yi. We owe them our thanks for the fruitful discussions during the workshop.

2 Introduction

Embedded systems can be distinguished from general purpose computing by several characteristics such as the diversity of the the application domains, the limited available system resources, and the heterogeneity of the requirements, constraints, specification and implementation.

For example, embedded systems can be found as tiny nodes within a distributed sensor network for environmental monitoring. In this case, the limited resources and constraints mainly concern size, power consumption, computing, communication and cost. Nevertheless, the whole system involves all layers of abstraction, from computer

[*]Results from Dagstuhl Perspectives Workshop on ”Design of Systems with Predictable Behavior”, L. Thiele, R. Wilhelm, November 2003. Work reported herein is supported by ARTIST, the European Network of Excellence in Embedded Systems, and the International Conference and Research Center in Schloss Dagstuhl.

architecture to distributed operation. Usually, these systems do not impose hard real-time constraints on the overall system behavior.

On the other hand, there are application domains such as automotive, air transportation, mechatronics, and multimedia processing where there are less constraints with respect to power consumption and size, but we are faced with high requirements in terms of time predictability. Not only the correctness of the computations, the availability and safety of the whole embedded system are of major concern but also the timeliness of the results. Missing deadlines of events may cause a catastrophic or at least highly undesirable system failure. For example, in case of automotive, space and mechatronics, the embedded system interacts with a physical environment that dictates the necessary speed of executions. In the case of multimedia and contents production, missing audio or video samples needs to be avoided under all circumstances. At the same time, the overall embedded system usually contains heterogeneous computing resources, memories, bus systems, operating systems and involves distributed computing via global communication systems.

Because of cost constraints, there is a tendency to use components that are tailored to the general purpose computing domain for the design of embedded systems, even if there are high demands related to time predictability. Unfortunately, this does not only concern hardware components such as microprocessors, bus systems and communication networks but also software components such as operating systems and middle-ware concepts. To make things even worse, even design tools and methods are increasingly taken from the general purpose domain such as compilers, validation and simulation tools, and design methods such as UML. At the same time, it appears that approaches to improve the average case behavior of systems are often disastrous to time predictability. Well known examples in the case of computer architectures are various forms of caches and advanced speculation techniques to improve instruction level parallelism.

It is the purpose of the paper to analyse the threats to time predictability, to analyse the state of the art and to propose design principles that support time predictability. In this sense, the paper serves as a tutorial and intends to initiate a research discipline that looks at predictability in a synergistic manner and that involves all levels of abstraction in embedded system design. Despite of the fact that the paper does not present specific research results in the usual way, we think that a careful discussion of the notion of time predictable systems and pointing out major deficiencies in the current state of the art is of major importance to the embedded systems community.

3 Definitions

The purpose of this section is to agree on the major terms that will be used throughout the paper. We well deal with time predictability only. Nevertheless one should keep in mind that a similar investigation could be made with other criteria in mind for which predictability is of major importance.

In addition, we will abstract an embedded system by using conventional discrete event notations. In particular, the system under consideration receives events and emits events. To each event there is associated the time when it occurs and some (unspecified) object to model data communication. If we would be interested in functional behavior, we would specify the desired relation between the input and output data objects. In case of time behavior the desired relation between the timing of input and output events is given. For example, in a simple case one may require that the time difference between a specified pair of input and output events may not be larger than a specified deadline. What is considered to be an event and object very much depends on the respective level of abstraction. For example, we may talk about the starting of a task on a single processor and its finishing time, we may talk about starting a distributed algorithm and when it delivers its results, or we may talk about the time for a memory access.

3.1 Execution Time

In terms of predictability, it will be very useful to define a set of parameters that describe essential properties. To this end, we will restrict ourselves on a very simple form of required timing behavior, namely the time interval between a specified pair of events. These events will be denoted as *timing events* and the time interval will be denoted as *execution time*. Note that not all pairs of events are necessarily critical, i.e. there exist deadline requirements. The relation between the major quantities are represented in Fig 1

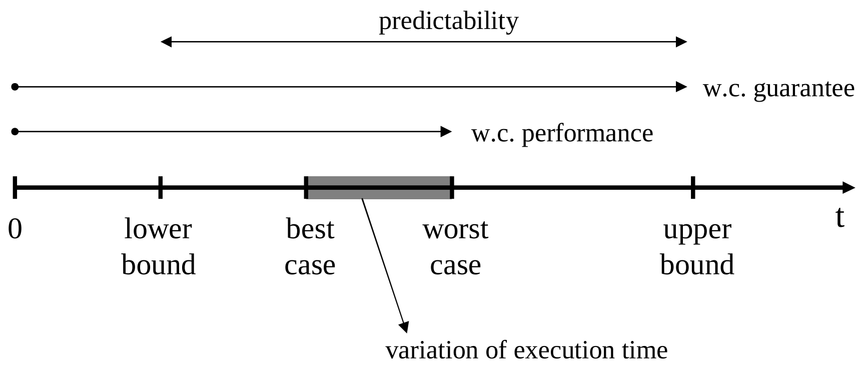

Figure 1: Representation of the relations between measures related to the predictability of system architectures.

- *Worst case and best case*: The worst case and best case execution time is the maximal and minimal time interval between the timing events under all admissible system and environment states. In other words, we look at the final system behavior in terms of the timing between the two events and consider all possible initial system and environment states and all possible environment and execution paths. It should be clear that the execution time may vary largely due to different input data, and interference between concurrent system activities.

- *Upper and lower bounds*: Upper and lower bounds are quantities that bound the worst case and best case behavior. These quantities are usually computed off-line, i.e. not during the run-time of the system. Several methods do exist such as analysis, simulation, emulation and implementation, see also the later discussion. Upper and lower bounds are used in order to verify statically, whether the system meets its timing requirements, e.g. deadlines.

- *Statistical measures*: Instead of computing bounds on the worst case and best case behavior, one may also determine a statistical characterization of the run-time behavior of the system, e.g. expected values, variances and quantiles. As we are interested in time predictability for real-time systems, we will not consider these models and methods in the paper.

The difference between the upper and lower bounds of the execution time is a measure for the time predictability of the whole system. In order to classify different causes for a low (or high) predictability, we need to be more precise on the reasons for varying time intervals between events. As a precondition, we suppose that the whole system under consideration is deterministic, i.e. two executions using the same behavior of the environment and the same initial states will lead to the same timing behavior.

- *Interference*: If there is a dependency between the execution time and some non-observed external behavior, then we will say that the time interval is non-deterministic with respect to the available information. Therefore, there will be a difference between the worst case and the best case behavior. If the upper and lower bounds are computed (or measured) using the same *limited knowledge* about the whole system, then we clearly can not achieve a smaller interval between the upper and lower bounds. An example may be that the execution time of a task may depend on its input data. Even if there is a simple relation between input data and run-time, a large variance in computation time may result if we are blind. Another example is the communication of data packets on a bus in case of an unknown interference. If looking at the latter example, it becomes clear that an embedded system can be constructed in a way that is time-insensitive to interference and reduces this kind of uncertainty, for example by using bus protocols like TDMA. As a result, a low predictability may be caused by limited knowledge, i.e. the system implementation is sensitive to relevant information that is not known or is not easily available at design time.

- *Limited analyzability*: If there is complete knowledge about the whole system, then the behavior of the system is determined. Nevertheless, it may be that because of the system complexity, there is no feasible way of determining close upper and lower bounds on the execution time. For example, if a microprocessor architecture implements techniques like speculation, out-of-order execution, branch prediction and complex cache replacement strategies, there is currently no analysis method available that yields close bounds on the execution time. Again, one could construct computer architectures containing concepts that can be analyzed more easily.

In summary, limited knowledge about the system environment and the system itself may yield a large variance in execution time, thereby decreasing predictability. In addition, the limited analyzability of system architectures leads to worse bounds on the execution time. As a consequence, there are two orthogonal but related ways to improve the time predictability of embedded systems in general: (a) *Reducing the sensitivity* to interference from non-available information and (b) *matching implementation concepts with analysis techniques* to improve analyzability.

There are several methods to determine bounds of the execution time as defined above. Besides analytic methods based on formal models one may also consider simulation, emulation or even implementation. All the latter possibilities should be used with care as only a finite set of initial states, environment behavior and execution traces can be considered. As is well known, the corner cases that lead to a worst case or best case execution time are usually not known and incorrect results may be obtained. The huge state space of realistic system architectures makes it highly improbable that the critical instances of the execution can be determined without the help of analytical methods.

3.2 Predictability, Performance and Guarantees

Despite of the fact that the present paper is devoted to the design of time-predictable systems, one should keep in mind that the performance in terms of timing is of major importance also. Therefore, a system implementation that is highly predictable in its timing but is very slow in computing for its cost can not be considered to be viable.

Our notion of performance is closely related to the execution time. Therefore, we will distinguish between worst case, best case and average case performance. A system that is designed to deliver a high average case performance is characterized by low average case execution times. But as will be seen in the later sections of the paper, this does not necessarily lead to a high worst case performance or low worst case execution time. In the contrary, there are many examples where increasing the average case performance leads to a reduced worst case performance. On the other hand, systems very often not only need to deliver a high worst case performance for real-time guarantees but also a high average case performance for other parts of the application.

In case of hard real-time systems, there is the need of giving either run-time or design-time guarantees on the worst case execution time. Therefore, we define the notion of worst case and best case guarantee as the upper and lower bounds of the execution time. In summary, we have defined and interpreted the following performance and predictability measures:

- *Predictability*: The time predictability of a system is related to the difference between upper and lower bound on execution time. A low predictability (large difference between upper and lower bound) is caused by interference and limited analyzability.
- *Performance*: The worst case and best case performance are related to the worst case and best case execution time, respectively. The average case performance measures the average execution time. Threats to a high worst case performance are usually caused by interference from unavailable or unknown information about the system or its environment.
- *Guarantee*: The worst case and best case guarantee are linked to the upper and lower bound on the execution time. Small worst case guarantees may be caused by interference and limited analyzability.

If we look at major system design tradeoffs such as static techniques vs. dynamic ones, domain specific vs. general purpose designs or run-time vs. design-time techniques, it is not clear at all what the individual impact is on predictability, performance and guarantees. On the other hand, a detailed knowledge about the dependencies would be of major importance in terms of designing predictable embedded systems with high performance.

3.3 System Layers

As will be seen in later sections, these questions arise at all system layers. For the rest of the paper we will distinguish between them as follows:

- *Hardware Architecture*: The hardware architecture layer contains all aspects below the instruction set. For example it contains the microprocessor architecture, I/O systems, bus and memory structures. Predictability refers to the execution times of instructions.
- *Compiler*: The compiler layer relates software programs in a high level language to the hardware-software interface. We include also the analysis and optimization tools conventionally associated to compilers. Timing events are related to the execution times of single tasks without interference by others.
- *Task Level*: We suppose that the whole application is partitioned into tasks and threads. Therefore, the task level refers to operating system issues like scheduling, memory management and arbitration of shared resources. The major additional influence with respect to the execution of tasks or whole applications consisting of several tasks is the interference via task scheduling and shared resources.
- *Distributed Operation*: Finally, we are faced with applications that run on distributed resources. The corresponding layer contains methods of distributed scheduling and networking. On this level of abstraction we interested in end-to-end deadlines.

One of the observations if looking at threats to predictability that cross-layer issues play an increasingly role. Therefore, an approach that concentrates on intra-layer effects needs to be complemented by synergies between layers. This fact is well know in the arena of optimizing the average case behavior, i.e. the partitioning of scheduling and allocation to the compiler (static methods) on the one hand and the hardware (dynamic methods) on the other.

4 Threats to Predictability

As we have seen in the previous section, several orthogonal properties of systems and system components reduce predictability, in particular when they appear in combinations. Whereas we have been giving a very course classification in terms of analyzability and interference, we will refine such properties in the following and classify the presented threats according to these categories. At the end of this section, we will present some experience with systems offering combinations of given and not given sets of such properties.

Non-deterministic behavior of systems An embedded system works in and controls a physical system mostly called the *plant*. The plant may or may be available for analysis. Non-deterministic influence of the plant on the embedded system, e.g. through sporadic interrupts make predictions of the overall system's behavior difficult. Schedulability-analysis methods require at least the knowledge of lower bounds on the frequency of interrupts. Low precision of these lower bounds and large variation in the distribution of interrupts contribute heavily to low quality of prediction and resulting low resource utilization. A major cause of the low predictability in case of non-deterministic behavior is the sensitivity of a design with respect to interfering non-deterministic behavior.

High variability of execution times Modern processor features such as caches, pipelines, and speculation cause a high variability of execution times for individual instructions, for individual task activations, for context-switch times, and for whole programs. This high variability, in particular in combination with other undesirable system properties carries through to overall low predictability.

Non-analyzability of system components Certain system-construction principles and means make the determination of run-time guarantees impossible because of the undecidability of the halting problem. However, even for systems with guaranteed termination the employed analysis methods will work well on systems with restricted means and fail on unrestricted designs. The question is which methods we consider for the purpose of the determination of run-time guarantees. There should be no religious preoccupation for one and against another method. The only criteria are precision of the results and computational effort needed. Exhaustive simulation of all possible executions of a system combined with an analysis of their performance is in general unrealistic. On the other hand, simulation of a restricted subset of potential executions can in general not deliver *safe* results. Safety, precision, and tolerable effort need to be achieved by the chosen methods.

Complexity of designs The methods employed need computational efforts that depends on certain systems parameters. System design should aim at keeping the parameters at favorable values.

4.1 Architectural Features

Computer architects traditionally optimize their design towards average-case performance. The processor-memory bottleneck has led to architectures with caches, pipelines, and control, data, and thread speculation. Experience with several processor architectures has shown that a number of architectural features are responsible for its degree of time-predictability [HLTW03].

Local Non-Determinism Processors behave deterministically, i.e. given a certain execution state, the successor state is uniquely determined by this state and any influence from the processor's environment, e.g. interrupts, inputs etc. Also the timing behavior for the execution of the instruction in the given state and for this external influence is uniquely determined. However, local non-determinism of the timing behavior is introduced by caches, pipelines, speculation. It means that the timing behavior of an individual instruction can not be determined locally, but depends on the execution history. This influential history can be arbitrarily long, as in the case of caches, or rather short, as in the case of pipelines. The mentioned processor components introduce an increasingly high variability of the execution times of instructions. The range for this variation stretches from a few machine cycles to several hundred cycles. This variation of instructions' execution times may carry over to the execution times of tasks and to context-switch times.

Interferences Between Processor Components Interference between architecture components is at the heart of unpredictability at the processor level. It means that one component's activity has an effect on another component's state mostly through modifications of a shared resource. We will list some examples.

Branch prediction prefetches instructions along a control path before it actually knows that this path will be taken. It loads these instructions into the instruction cache if they are not already contained.

Unified data and instruction caches is one particular example of a shared resource. The unified cache is used both by the fetch stage and the execution stage of the pipeline. These interact on the cache; instruction fetch may evict data from the cache, and loading data may evict instructions from the cache. In the case of a superscalar pipeline, the order of memory accesses and cache replacements may not even be clear jeopardizing precision of cache-behavior prediction.

Register overlays ???

Interferences between processor components are responsible for so-called *Timing Anomalies* [LS99]. These are contra-intuitive influences of the (local) execution time of one instruction on the (global) execution time of the whole program. A locally faster execution of an instruction can lead to a globally longer execution time of the whole program or speed up the program by more than this instruction's speed up. The first case is critical for the determination of worst-case execution times, because it does not allow the analysis to continue with a locally favorable assumption. A locally slower execution may lead to a globally shorter execution time. This analogously is critical for the determination of best-case execution times. The general case is the following. An example is the following. The assumption that an instruction is in the instruction cache, may result in an overall shorter execution time of the program, e.g., if it prevents a costly branch misprediction.

Timing anomalies necessitate conservative, i.e., upper approximations to the damages potentially caused by all cases or forces the analysis to follow all possible scenarios.

Unfortunately, as [LS99, Lun02] have observed, the worst case penalties imposed by a timing anomaly need not be bounded by an architecture-dependent, but program-independent constant, but may depend on the program size. This is the so-called *Domino Effect*. This domino effect was shown to exist for the Motorola PowerPC 755 in [Sch03].

Non-predictable Variability We have described above how variability is caused by architectural features. This variability does not harm as long as it can be well bounded for a concrete system. However, it may for principle reasons be hard to control. One example for this are systems with virtual memory and translation-lookaside buffers. TLB misses implemented by linear search or by hashing need not have constant penalties.

Concurrency in Combination with Updating Shared Resources As stated above, shared resources decrease the chance to predict the behavior. This problem is aggraveted by several forms of concurrency, e.g., super-scalarity, out-of-order execution, and dynamically scheduled multi-threading.

DMA may completely ruin predictability. DRAM refresh can sometimes be amortized over the interval between two refreshs, since it happens independently of program actions. However, analysis methods have to careful at program points where control-flow paths are merged, because partially amortized DRAM refreshs may suggest to chose a wrong worst-case path [AP01].

Implicit Actions Analyses are sensitive against implicit side effects as they are often created through aliases. A part of the execution state is changed using one name, and this change is observed through another name. One example for this are memory mapped registers, i.e., registers that are identified with memory cells.

Stochastic Protocols in Networking Only stochastic guarantees can in general be given, if stochastic protocols are used in a system.

4.2 Software

Software design and implementation influence system predictability in several ways. Software may be automatically synthesized from formal specifications. In this case, the code-synthesis method is responsible for the predictability quality. Software may be handwritten in an appropriate or inappropriate programming language. The use of particular features of the language may have a strong influence on system predictability. The chosen software architecture, e.g., as a static of dynamic set of tasks or structured around a broker of some middleware will also influence predictability.

Pointers Programs with pointers including pointers to functions are hard to analyze statically. Data structures built in the heap from dynamically allocated memory cells in general allow only the prediction of asymptotic execution times. These are of little use for hard real-time systems.

An indirect call through an unresolvable function pointer, i.e., a function pointer whose target is not clear to the analysis, has an unknown and therefore maximally desastrous effect on the execution state. It can decrease precision enormously.

Dynamic Task Creation Dynamically growing sets of tasks do not allow for offline scheduling and thus for static run-time guarantees. For these reasons, they are usually forbidden in hard real-time contexts.

Dynamic Method Binding The object-oriented programming style, although attractive as a software development methodology, introduces dynamics into the execution time by the dynamic binding of methods to calls. Although in general the class hierarchy is static and thus the search for the right method is statically bounded, dynamic dispatch enlarges the variability of execution times.

Garbage Collection (GC) Garbage-collected languages offer strong support for memory cleanness, e.g., the absence of dangling references and of freeing aliased memory. However, standard garbage collection methods may block program execution for quite some time undermining real-time performance. Specific real-time garbage collection methods have been developed [BCR03], which attempt to spread the GC effort over the execution time and thus guarantee a bounded response.

Middleware Object-oriented design is often combined with using middleware. A required service may reside on the same processor as its client or may have to be loaded from a remote server. The variance of service times will be very large due to the alternatives of local or global access, network latency, contention etc.

4.3 Task Level

There is a long history in real-time task scheduling. Well developed, and even supported by (forgotten) programming languages. Examples are as follows:

- Resource sharing, priority inversion
- Variation in execution times and context-switch times due to caches
- Dynamic non-real-time scheduling
- Dynamic task creation
- Interrupts

On the other hand, these efforts have not been sufficiently linked to the lower system layers such as task level analysis and compilers. In addition, the results on distributed real-time systems are not directly applicable to embedded systems and therefore, also links to upper system layers are still open.

4.4 Distributed Operation

- Distributed real-time systems
- Communication-centric designs
- Different models of computation and communication in one application
- Need for heterogeneous implementations
- High-level design space exploration needs reliable estimation
- Composability required from application domains
- Resource sharing

4.5 Between layers

Scheduling on several levels There may be several instances of scheduling in a multi-layered system. Some of it may happen offline if guarantees are to be derived or if the hardware requires it. Other scheduling will be executed online either by hardware or by scheduler tasks. Viewing layers bottom up, an EPIC or VLIW architecture requires *instruction scheduling* to be done by a compiler. In contrast, in a superscalar architecture, scheduling is done by the hardware. A multi-threaded processor architecture will again perform dynamic scheduling since the threads are not completely independent, but share resources. The software threads mapped to the processor threads will also be subject to some scheduler realized in software. Uncoordinated scheduling on these two levels probably offers the biggest surprises.

- Compiler vs. Hardware Architecture
- Local vs. distributed task scheduling
- Communication and computation

	Compiler responsible	Processor responsible
Architectural concepts	EPIC/VLIW	Superscalar
Memory	Scratchpad memory	Caches
		Speculation
Properties:		
Focus	large	small
Available information	only static	also dynamic
Complexity	in algorithms	in hardware
	heuristics required	high energy costs
Adaptability	low	high
Heap	difficult to analyze	
Predictability	high	low

Figure 2: The static vs. dynamic issue between architecture and compiler

Inter-Level Dependencies Several layers can interfer on shared resources. For example, task scheduling may experience varying context-swith costs even for the same combination of preempting and preempted tasks, because a dynamically scheduled processor may have different cache states at preemption points and therefore different cache reloading costs.

5 Increasing Predictability

Upon first thought, one would expect to find here the negation of all the properties listed in the introduction of Section 4. However, such a simplistic approach would lead to constraints on system design non-acceptable for their purposes. A simple micro-controller without caches and pipelines would have none of the properties discussed in Subsection 4, but would most likely not deliver the necessary performance. An embedded system not admitting interrupts would often not fit into the physical context it was designed for.

Successful Research Research, particularly in Europe, has been successful in several relevant fields. The problem of *WCET determination* for single tasks and quite complex processors has been solved [WETW04, FHL+01]. Commercial tools are available. Industrial experience regarding precision, analysis times, and ease of use is positive. *Schedulability Analysis for Non-distributed Targets* is very well understood [But97, TBW94]. *Modular performance analysis* for distributed implementations of real-time systems has been developed and implemented [**?**, **?**]. The principle of Time-Triggered Architecture (TTA) enforcing determinsim in the communication behavior of distributed systems has been brought to industrial maturity. *Synchronous languages*, such as Esterel, Lustre, and Signal for the specification of reactive systems are in routine use in highly safety-critical domains.

5.1 Architecture

LRU caches have not only been shown to have the best behavior in theory, they also have the best known predictability properties of all cache associative architectures.

Higher degrees of predictability can be achieved if static decisions instead of dynamic decisions are being used. A number of techniques have been developed. *Compiler-Directed Memory Management* using scratchpad memory [SWLM02] originally developed to decrease energy consumption also increases time predictability. Predictable behavior can be expected of multi-threaded architectures only if they are statically scheduled [URi03]. Parallelism instead of speculation is used in EPIC architectures.

5.2 Software

Model-Based Design and Code Synthesis Model-based design is frequently used in embedded systems development. The code synthesized from formal specifications is often very cleanly structured. This supports its analysis for WCETs. High precision can be achieved [TSH$^+$03].

Coding Guidelines Coding guidelines restricting the implementation language to a disciplined subset are an alternative. A recent survey undertaken by the ARTIST working group on Timing Analysis [WETW03] has shown that developers were willing to adopt coding guidelines.

Specification Formalisms Statecharts, synchronous languages.

5.3 Task Level

Enforced determinism

- Models of computation
  - Ghiotto
  - Process networks
- Static scheduling

5.4 Distributed Operation

- Time Division Multiple Access (TDMA)
- Time Triggered Architecture (TTA)

5.5 Combining Layers

The design principle *globally asynchronous, locally synchronous* (GALS) seems to support the disciplined designs of synchronous languages on the lower level and the independent development of a systems' components on the higher level. It has the potential to offer a homogeneous semantic foundation for systems development [GM02].

6 What's Missing?

A new discipline *Design for Predictability* should be developed. Safety-critical embedded systems should not exhibit surprising behavior. Our main concern in this paper is *Time Predictability*. However, predictability in *energy consumption* is a related and highly desirable property, in particular for mobile devices. Predictability in *space consumption* excludes memory-overrun problems on the one side and allows to reduce system costs, which is of high interest for mass products.

The design rules to be developed concern the design of all systems components, of the system architecture, and of the coordination between components. They also concern the methods used in implementing systems and the tools used in the development process.

6.1 Estimation

Performance analysis including computation and computation.

Rough estimates on the task level The current and probably also the future system development processes [con04] require that rough estimates of the timing behavior of single tasks be determined in order to choose a provisional task distribution, which may be corrected later, when all system components are available and the full hardware specification is known.

6.2 Integration of WCET

Currently, WCET tools work more less in a stand-alone fashion. Little integration is available with preceding passes such as code synthesis and compilation and with subsequent passes such as schedulability or performance analyses. This may lead to a considerable loss in precision.

With performance analysis

Schedulability analysis requires knowledge of the WCETs of the tasks to be scheduled. On modern processors with caches and pipelines the high variability of execution times carries over to the context-switch times. These may vary depending on the amount of *useful state* when a task is preempted. More precisely, task T_2 preempting task T_1 may destroy more of the execution state of T_1 than task T_3 preempting T_1. This damage to the execution state, e.g. cache contents, has to be restored when T_1 resumes. This will take time, which has to be accounted for. Computing the upper bound to all the potential damages is possible, but is too imprecise if the variance is high. Computing the damages for all combinations of preemptor and preempted task for all program points will suffer from combinatorial explosion. Limiting preemption to certain program points, preferably with little useful state, has been proposed and experimentally tested [LLM+98, Sch03].

Many safety-critical embedded systems are developed in a model-based design process. The actual implementation is obtained by code synthesis. Not all information available at the specification level is being transferred into the code and thus made accessible to compiler analyses and optimization. Similarly, compiler and timing analysis offer some synergy so far unexploited. The compiler usually has information that is hard to extract from the target code.

6.3 Multi-Layered Systems

The trend in the design of complex embedded systems both in the aeronautics and in the automotive domains goes towards multi-layered designs. A real-time operating systems (RTOS) is running on the target hardware scheduling the periodic tasks and handling interrupts for sporadic tasks. The tasks of the system are constructed using real-time middleware. Distributed tasks communicate via messages.

Coordination mechanisms have to be developed to ensure a predictable behavior. Consider the scheduling done on the architectural, the single-node operating system, and the distributed system level. Lack of coordination will lead to unpredictability.

The analysis machinery for multi-layered systems does not exist. It is hard to imagine, how halfways precise timing predictions can be obtained with unconstrained multi-layered systems.

Cross-Layer Timing Interfaces

6.4 Supporting an Incremental Development Process

Currently, all high-precision WCET tools require the availability of fully linked executables with all allocation details. Component-based development of real-time systems should be supported by an incremental development process, where components are analyzed as they are produced or imported and the timing behavior of the whole system is conservatively composed of the predictions derived for its components.

6.5 Computation and communication

Better understand the interdependencies between resource sharing on computation and communication. Performance estimation of distributed embedded systems.

References

[AP01] Pavel Atanassov and Peter Puschner. Impact of DRAM refresh on the execution time of real-time tasks. In *Workshop on Application of Reliable Computing and Communication*, December 2001.

[BCR03] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. *SIGPLAN Not.*, 38(1):285–298, 2003.

[But97] Giorgio Butazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer, 1997.

[con04] The ARTIST consortium. Roadmap on hard real-time development environments, 2004. to be published in the Springer Lecture Notes Series.

[FHL$^+$01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. WCET Determination for a Real-Life Processor. In T.A. Henzinger and C. Kirsch, editors, *Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469 – 485. Springer, 2001.

[GM02] A. Girault and C. Ménier. Automatic production of globally asynchronous locally synchronous systems. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *2nd International Workshop on Embedded Software, EMSOFT'02*, volume 2491 of *LNCS*, pages 266–281, Grenoble, France, October 2002. Springer-Verlag.

[HLTW03] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture an the design and the results of WCET tools. *IEEE Proceedings on Real-Time Systems*, 91(7):1038–1054, 2003.

[LLM$^+$98] S. Lee, C.-G. Lee, Lee M., S. L. Min, and C. S. Kim. Limited Preemptible Scheduling to Embrace Cache Memory in Real-Time Systems. In *Proceedings of the ACM SIGPLAN LCTES'98 Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 51–64, June 1998.

[LS99] T. Lundquist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *20th IEEE Real-Time Systems Symposium*, 1999.

[Lun02] Thomas Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 2002.

[Sch03] Joern Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, 2003.

[SWLM02] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE Conference 2002*, 2002.

[TBW94] K. Tindell, A. Burns, and A. Wellings. An Extensible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. *The Journal of Real-Time Systems*, 6(2):133–151, March 1994.

[TSH$^+$03] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software systems. In *Proceedings of the Performance and Dependability Symposium, San Francisco, CA*, June 2003.

[URi03] Theo Ungerer, Borut Robi&#269;, and Jurij &#352;ilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, 2003.

[WETW03] Reinhard Wilhelm, Jakob Engblom, Stephan Thesing, and David B. Whalley. Industrial requirements for wcet tools - answers to the artist questionnaire. In *WCET 2003*, pages 25–29, 2003.

[WETW04] Reinhard Wilhelm, Jakob Engblom, Stephan Thesing, and David Whalley. The determination of worst-case execution times —introduction and survey of available tools—. submitted, 2004.