Real-Time Systems, 28, 157–177, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

Design for Timing Predictability*

LOTHAR THIELE thiele@tik.ee.ethz.ch
Department of Information Technology and Electrical Engineering, ETH Zürich, Switzerland

REINHARD WILHELM wilhelm@cs.uni-sb.de
Informatik, Universität des Saarlandes, Saarbrücken, Germany

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

1. Introduction

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

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 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, avionics, mechatronics, and multimedia processing where there are less constraints with respect to power consumption and size, but with high requirements in terms of timing 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 the case of automotive, avionics, 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 need to be avoided under all circumstances. At the same time, the overall embedded system

* 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 for Computer Science in Schloss Dagstuhl.

usually contains heterogeneous computing resources, memories, bus systems, operating systems and involves distributed computing via global communication systems.

To quote Halang (2004),

> Sources of unpredictable delays and unbounded contention in contemporary computers are the use of synchronisation methods without a notion of time, interrupt inhibition, operating system overhead, direct memory access, caching, pipelining, dynamic storage allocation, virtual storage, garbage collection, multi-tasking, static priority scheduling, probabilistic arbitration and communication protocols etc., i.e., practically all dynamic and ''virtual'' features aiming to enhance the *average* performance of non-real-time systems which are, therefore, *considered harmful.*

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 timing 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 middleware concepts. To make things 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 timing 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 analyze the threats to time predictability, to describe the state of the art, and to propose design principles that support timing 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, we think that a careful discussion of the notion of timing predictable systems and of major deficiencies in the current state of the art is of major importance to the embedded systems community.

2. Definitions

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

2.1. 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 to a very simple form of required timing behavior, namely the time interval between a specified pair of events. For example, the issuing of an instruction or the instantiation of a task could be a start event. 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, that is, have deadline requirements. The relation between the major quantities are represented in Figure 1.

- *Worst case and best case*: The worst case and best case execution time define 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, that is, 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, for example, deadlines.

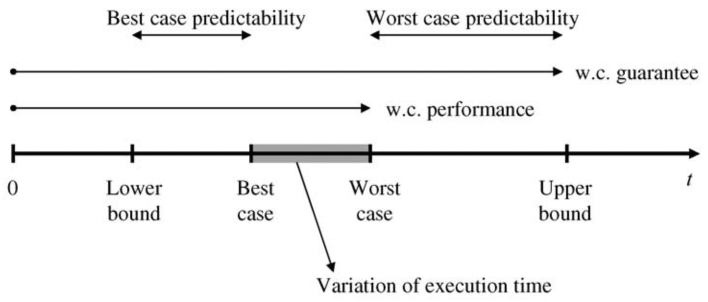

Figure 1. Representation of the relations between measures related to the predictability of system architectures.

- *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, for example, expected values, variances and quantiles. As we are interested in timing predictability for real-time systems, we will not consider these models and methods in this paper.

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

- *Unknown external 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, that is, 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.

As a consequence, there are two orthogonal but related ways to improve the timing 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 spaces of realistic system architectures make it highly improbable that the critical instances of the execution can be determined without the help of analytical methods.

2.2. Predictability, Performance and Guarantees

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

But as will be seen in the later sections of the paper, a high average case performance does not necessarily lead to a high worst case performance or short worst case execution time. In the contrary, there are many examples where increasing the average case performance leads to a reduced worst case performance.

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

- *Performance*: The worst case and best case performance are reciprocically related to the worst case and best case execution times, 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.

- *Predictability*: The timing predictability of a system is related to the differences between best case and lower bound on the one hand and upper bound and worst case on the other. The former we call best case predictability, the latter worst case predictability. This allows us to treat systems which are predictable on the one side and badly predictable on the other, see Section 3.1. Bad predictability is caused by interference and limited analyzability of the behavior.

- *Guarantee*: The worst case and best case guarantees are linked to the upper and lower bound on the execution time. Low-quality guarantees may also 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.

2.3. 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 concerns all aspects below the instruction set. For example, it concerns the microprocessor architecture, I/O systems, bus and memory structures. Predictability here refers to the variability of the execution times of instructions.

- *Software development for single tasks*: This layer concerns all ingredients of the software development process for individual tasks, possibly code synthesis from model-based design, and the compiler, including all analysis and optimization tools. Timing events are related to the execution times of single tasks without interference from 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 are interested in end-to-end deadlines.

One of the observations concerning threats to predictability is that cross-layer issues play an increasing role. Therefore, an approach that concentrates on intra-layer effects needs to be complemented by synergies between layers. This fact is well known in the arena of average-case optimization, that is, the assignment of scheduling and allocation to the compiler (static methods) on the one hand and the hardware (dynamic methods) on the other.

3. 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 combination. We have given a very coarse classification in terms of analyzability and interference, will now refine such properties, and classify the presented threats according to these categories. At the end of this section, we will present some experience with systems possessing or not possessing combinations of such properties.

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.

3.1. Architectural Features

Computer architects traditionally optimize their design towards average-case performance. Computer architectures have been supplied with caches, pipelines, as well as with control, data, and thread speculation to circumvent the memory-processor bottleneck. Experience with several processor architectures has shown that a number of architectural features are responsible for its degree of time-predictability (Heckmann et al., 2003).

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.

Local non-determinism. Processors behave deterministically, that is, given a certain execution state, the successor state is uniquely determined by this state and any influence from the processor's environment, for example, 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.

Caches provide an example of an architecture causing different best case and worst case predictabilities. Ferdinand et al. (2001) describe a tool for the determination of upper and lower bounds on execution times, exemplified on the Motorola ColdFire processor.

Its cache replacement strategy provides for a quite acceptable worst case (Thesing et al., 2003) and very bad best-case predictability. It should be noted, however, that the good worst-case predictability resulted from the very disciplined nature of the analyzed code.

Interferences between processor components. Interference between architecture compo-nents 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. Hence, the same branch once executed speculatively, once not, will have different effects on the instruction cache.

- A 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.

Interferences between processor components are responsible for so-called Timing Anomalies (Lundquist and Stenström, 1999). 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. 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, if it prevents a branch misprediction, that is more costly than the penalty for missing the instruction-cache.

To deal with timing anomalies one could follow two alternative approaches. If an upper bound to the costs that may result from the local best-case assumption exists and is known, this cost could be conservatively added. Otherwise, one would let the analysis follow all possible scenarios starting with all local assumptions and compare the resulting times at appropriate synchronization points such as the end of basic blocks.

Unfortunately, as Lundquist and Stenström (1999) and Lundqvist (2002) 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 (Schneider, 2003).

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 aggravated by several forms of concurrency, for example, 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 be careful at program points where control-flow paths are merged, because partially amortized DRAM refreshs may suggest to chose a wrong worst-case path (Atanassov and Puschner, 2001).

Implicit actions. Analyzes 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, that is, registers that are identified with memory cells. Another example are register overlays.

3.2. 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, for example, 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 generally allow the prediction of asymptotic execution times, only. These are of little use for hard real-time systems.

An indirect call through an unresolvable function pointer, that is, 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 despatch enlarges the variability of execution times.

Garbage collection (GC). Garbage-collected languages offer strong support for memory cleanness, for example, 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 (Bacon et al., 2003), 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 middleware. The use of object brokers allows for the dynamic loading of components. 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. Thus, the use of middleware endangers predictability in general. On the other hand, the dynamic loading of components seems to be superfluous for systems with required predictable behavior. Their evolution consists in sequences of upgrades, each one with an accompanied determination of the necessary guarantees.

3.3. Task Level

There is a long history of research, development and implementation in real-time operating systems. The purpose of this effort is to provide the application programmer with a programming interface that hides most of the underlying implementation details and provides useful services, such as scheduling, handling of shared resources, memory hierarchy, synchronization and communication, see, for example, Buttazzo (1997). We also have seen efforts in integrating some of these concepts into programming languages, such as Ada. Only a few threats to predictability will be explicitly mentioned here, as most results are well known.

Task interference. Major efforts are undertaken to isolate the execution times of different tasks or streams of tasks from each other. This way, one could prevent that non-deterministic changes in arrival times or execution times of one stream of tasks influences the timing of others. However, there are only finite resources available in terms of memory, computing, communication and other sorts of shared resources. Most of the classical results of the real-time systems community try to minimize or eliminate these interferences and to provide hard on-line or off-line time bounds.

Scheduling. The basic scheduling algorithms, like proportional share, fixed or dynamic priority scheduling, try to eliminate the interference caused by the shared computation resource. They require that the arrival patterns of tasks belong to some task model such as periodic, periodic with jitter or sporadic, in order to avoid oversubscription. Nevertheless, even in this restricted domain, there are threats to predictability caused by the inability to accurately determine upper and lower bounds on task execution times and by the variability of task switching times.

Interrupts. Embedded systems are usually sensitive to external events that cause an interrupt of the currently running task. In addition, the scheduling methods mentioned above usually also use preemption. Form a tasking point of view, all of these interrupts occur at non-deterministic times and sometimes not even bounds on the number of interrupts can be given. The associated timing overhead of interrupts can not be determined accurately on the task level. This decreases the predictability of task execution times.

Resource sharing. Tasks not only interfere on the computing resource. They may require the exclusive use of common external resources such as data structures in memory, input/output components or communication media. As soon as the use of such a resource is not atomic, one task may block another one, which increases the interference. Many different protocols such as priority inheritance, priority ceiling and stack resource have been devised to limit interference and to increase analysability by computing bounds on the associated blocking times.

On the other hand, these efforts have not been sufficiently linked to the lower system layers such as task level timing analysis and compilers. For example, the timing consequences of interrupts occurring at unknown program states can hardly be estimated with high accuracy. In addition, many results on distributed real-time systems are not directly applicable to embedded systems and therefore also links to upper system layers are still open.

3.4. Distributed Operation

Embedded systems are getting increasingly distributed. We not only find closely coupled multiprocessors, but also distributed subsystems that are communicating via networks, for example, in automotive applications. Because of the increasing integration density, we see networking concepts emerging on a single chip, thereby moving from computation-centric to communication-centric design methodologies. An example of this development are the advances in Network-on-Chips where computational modules communicate with each other via a uniform interface.

This development creates new threats to predictability in three respects. First, communicating tasks interfere on the common communication resource. Second, we have to deal with dependent tasks and task chains; therefore, we are interested in end-to-end delays of events or packets. Finally, the use of common resources such as I/O systems and memory becomes more critical because of true concurrency. Some more detailed examples are given below.

Synchronization. Hardware and software systems exist, that take care of the access to shared resources in case of contention. Exponential back-offs are generated to prevent tasks from reattempting the acquisition of a shared resource too quickly. Such devices, for example, the store-conditional/load-linked mechanism as described in Hennessy and Patterson (2003, p. 704) introduce task latencies, where in general no sharp upper bounds can be determined. One could argue that the designer of a hard real-time system has to statically exclude unbounded contention in the first place.

Unpredictable communication. One major source of decreased timing predictability is the close interaction between computation and communication in distributed embedded systems. In particular, the response time of a process now depends on the message delay across the network, that is, the time between sending and receiving a message. This interference between different communicating task structures is caused by the variable network performance under varying traffic conditions.

Additional unpredictability is caused by the interfaces between scheduling policies for the computation resource on the one hand and arbitration schemes for the communication system (TDMA, CDMA, fixed priorities, FCFS, collision detection) on the other. For example, we may be faced with the well known ‘‘incompatibility’’ between time triggered and event triggered paradigms.

Analysis. The heterogeneous architecture—the different architectural components are designed assuming different input event models and use different arbitration and resource sharing strategies—makes any kind of compositional analysis difficult. Secondly, embedded system applications very often exploit a high degree of concurrency. Therefore, there are multiple control threads, which additionally complicate timing analysis.

Currently, the analysis of such heterogeneous systems is mainly based on simulation (using SystemC or tools like VCC (VCC,)) or trace based simulation (see for example, Lahiri et al., 2001). However, for guaranteeing correctness, simulation-based approaches do not reveal worst-case bounds on essential properties like delay of events in an event-stream, throughput and memory.

Analytical performance models for distributed embedded systems and embedded processors have been proposed, but the computation, communication, and memory resources of a processor are described using simple algebraic equations that do not take into account the dynamics of the application, that is, variations in resource loads and shared resources. These ‘‘back-of-the-envelope’’ analytic approaches suffer from a lack in accuracy. Therefore the estimation results show large deviations from the properties of the final implementation.

A methodology for performance analysis is missing that integrates seamlessly with the current component-based approach to system synthesis. Whereas it is possible to combine components of a functional system specification, for example, using heterogeneous models of computation such as in Ptolemy (Lee, 2001), SystemC, or even UML component models, this is not possible with analytical models for performance analysis. As a result, today performance analysis is largely detached from the hierarchical design flow.

3.5. Between Layers

The multi-layer architecture of systems creates several problems and correspondingly requires design decisions or asks for supporting methods. First, design concerns several layers, for example, the mapping of functions to a network. Second, the same type of activity, for example, scheduling, may be executed on several layers, see below. Third, the activity on one layer influences the timing of actions on a different layer, see the example of varying context-switch times below.

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. The distribution of responsibilities between the compiler and the hardware architecture recurs on several levels. Design decisions involve considerations not only of predictability, but also of energy consumption by the processor and complexity of the compiler algorithms. Several instances of the static vs. dynamic design decision are shown in Figure 2.

Inter-level dependencies. Several layers can interfere on shared resources. For example, task scheduling may experience varying context-switch 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.

	Compiler responsible	Processor responsible
Architectural concepts	EPIC/VLIW	Superscalar
Memory	Scratchpad memory	Caches
		Speculation
Focus of analysis	Large	Small
Available information	Only static	Also dynamic
Complexity	In algorithms ⇒	In hardware ⇒
	Heuristics required	High energy costs
Adaptability	Low	High
Predictability	High	Low

Figure 2. The static vs. dynamic distribution of responsibilities between architecture and compiler.

4. Increasing Predictability

Upon first thought, one would expect to find here the negation of all the properties listed in the introduction of Section 3. However, such a simplistic approach would lead to constraints on system design non-acceptable for their purposes. A simple microcontroller without caches and pipelines would have none of the properties discussed in Section 3, 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. However, it should be clear that many of the threats listed above should be avoided.

Research has been successful in several relevant fields. This section summarizes insights gained into what increases predictability. The following section then concentrates on what is missing of a discipline one could call design for timing-predictability.

4.1. Architecture

The problem of WCET determination for single tasks and quite complex processors has been solved to a large extent (see Wilhelm et al., 2004; Ferdinand et al., 2001). Commercial tools are available. Industrial experience regarding precision, analysis times, and ease of use is positive. This work has shown that predictability critically hinges on some architectural features. LRU caches have not only been shown to have the best behavior in theory, they also have the best known predictability properties of all set associative architectures.

Higher degrees of predictability can be achieved if static decisions instead of dynamic decisions are being used. A number of techniques has been developed. Compiler-directed memory management using scratchpad memory (Steinke et al., 2002) originally developed to decrease energy consumption also increases timing predictability, as shown in, for example, Wehmeyer and Marwedel (2004).

Predictable behavior can be expected of multi-threaded architectures only if they are statically scheduled (Ungerer et al., 2003). Parallelism instead of speculation is used in EPIC and VLIW architectures nicely exposing the scheduled code to careful analyses.

4.2. Software

An example for removing non-determinism due to input dependence is the single-path approach (Puschner and Burns, 2002). In this conservative method, predictability is achieved at the cost of performance.

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 (Thesing et al., 2003).

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 (Wilhelm et al., 2003) has shown that developers were willing to adopt coding guidelines.

4.3. Task Level

Recently, model-based approaches to the design of complex embedded systems have become popular. The objective is to determine formal models for all relevant elements such as application, communication and synchronization, scheduling, hardware architecture, and the associated mapping. These models need to be consistent allowing for an increased predictability of the overall system behavior. Conservative design aims at avoiding non-functional dependencies between tasks. On the other hand, it is well known that the model-based approach impedes restrictions on the system specification that are not well accepted by designers. Some examples of successful models are summarized below.

Advanced analysis techniques. There are numerous techniques that use advanced mathematical tools to analyze the real-time behavior of systems on the task level. They are very often based on the existence of invariants and use formal verification techniques, for example, symbolic model checking (Larsen et al., 1997; Henzinger et al., 1994), controller synthesis (Maler et al., 1995) or even Petri nets (Bucci and Vicario, 1995). Other approaches to analyze different scheduling policies in model-based approaches are based on process algebras extended with a notion of priority. Examples are extensions of CCS with real-time semantics, RTSL (Real-Time Specification Language) (Lee et al., 1994) or ACSR (Algebra of Communicating Shared Resources) (Bremond and Lee, 1997).

Models of computation. A prominent example of a model of computation that increases the predictability is the time-triggered paradigm. Activities are initiated at predetermined points in time. The time-triggered paradigm can be seen as an extreme solution that combines well known techniques such as the use of preemption points and cooperative multitasking in order to increase the predictability of interrupts and task switching.

A recent example of a methodology based on a restricted model is Giotto (Henzinger et al., 2001). It allows for the consideration of harmonic periods, involves mode changes during the execution, and strictly separates computation and control. The approach is closely related to the synchronous paradigm (Benveniste and Berry, 1991) and corresponding languages, for example, Esterel (Berry and Gonthier, 1992).

In Ptolemy (Lee, 2001; Lee and Messerschmitt, 1987), several models of computation are integrated for the purpose of simulation, analysis and implementation. In particular, many results are available concerning the predictability of restricted data flow models such as synchronous data flow (SDF) graphs. They are characterized by a deterministic partial ordering of tasks which favors timing predictability.

There are recent approaches to extend the purely control-dominated and purely dataflow-oriented models described above towards a unified model that still allows for worst-case analysis. Examples in this direction are the recurring task model (Baruah, 2003), the SPI and the FunState model (Strehl et al., 2001; Ziegenbein et al., 2002).

4.4. Distributed Operation

One major possibility to improve predictability in distributed embedded systems is conservative design. The corresponding policy is guaranteed by following a model-based approach, using compatible models of application, architecture, and mapping. This way, adapted analysis methods can be used, and non-determinism and interference at interfaces between non-compatible models can be avoided.

Reducing interference. In order to predict communication delays, several protocols have been proposed that schedule the messages statically, for example, the time division multiple access (TDMA), the time-triggered protocol and the corresponding time triggered architecture (TTA) (Kopetz, 1997). Here, activities are initiated at pre-specified time instances. In terms of task scheduling, usually statically determined cyclic schedules are used. In case of a distributed system, this mode of operation assumes that all participating nodes are perfectly synchronized.

In a similar way, one can restrict the underlying model of computation, communication and implementation to a purely static dataflow paradigm. In this case, predictable timing behavior can be obtained as well (see, for example, Sriram and Bhattacharyya, 2000).

However, it is well known, that this restriction to a pure static operation comes with the disadvantage of higher cost, higher power consumption, reduced flexibility and smaller utilization factors. Therefore, advanced analysis techniques have been investigated to provide close bounds on the timing behavior even in case of event-triggered or mixed models.

Advanced analysis. Recent approaches to the timing analysis of distributed embedded systems handle messages and communication resources in a similar way as tasks and computation resources (see, for example, Tindell and Clark, 1994; Pop et al., 2003; Richter et al., 2002). They all start form a restricted event model, for example, periodic, sporadic, or periodic with jitter. There have been even analysis results where the interference between event triggered and time-triggered computation and/or communication paradigms can be bounded (see, for example, Pop et al., 2003; Tindell and Clark, 1994).

In Thiele et al. (2001), a unifying approach to performance analysis was proposed that is based on real-time calculus (Thiele et al., 2000). It is based on a generalized event model that allows the modeling of hierarchical and heterogeneous scheduling and arbitration, and can be used to model both computation and communication resources (Chakraborty et al., 2003). The approach can be considered to be a step towards a compositional worst-case analysis of distributed embedded systems.

The design principle globally asynchronous, locally synchronous (GALS) supports the design of large-scale systems consisting of concurrently executing subsystems, These subsystems are designed using synchronous languages or Statecharts. It has the potential to offer a homogeneous semantic foundation for systems development (Girault and Ménier, 2002).

5. What is Missing?

A new discipline design for predictability needs to be developed. Safety-critical embedded systems should not exhibit surprising behavior. Our main concern in this paper is timing 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.

Looking at the increasing complexity and performance demands for embedded systems, it appears that a pure conservative design strategy is not a solution for cost, performance, and power consumption reasons.

5.1. Needs Due to the Design Process

One path to follow is the development of advanced timing analysis methods across all relevant layers. These methods can be used in cases where conservative design techniques are not appropriate for reasons of cost, performance or power consumption. But in this case, they must span a large design space including distributed systems, heterogeneous scheduling, and arbitration policies.

Currently, WCET tools work more or 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.

For example, timing analysis of tasks usually assumes independent processes running on a single processor. Close bounds on the timing effect of non-deterministic interrupts or preemption can only be determined if we assume cooperative scheduling. In case of distributed embedded systems, we have to consider additional data and control dependencies, mutual exclusion, influences of I/O circuits, and the communication protocol.

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 cache contents when a task is preempted.

Model-based design is one possibility to ensure predictability across all layers by reducing non-functional dependencies between subsystems. Therefore, many safety-critical embedded systems are developed in a model-based process. The actual implementation is often obtained by code synthesis.

But 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. This lack of coordination between the different layers makes the main disadvantage of conservative design even worse: Oversubscription of resources that comes at a significant price in other non-functional objectives.

5.2. Needs Due to Systems Architecture

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. Systems constructed this way can only be analyzed, if very rigid rules are obeyed on all and between all layers.

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. For example, there is still a lack in understanding of the interdependencies between resource sharing in computation and communication.

An 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.

5.3. Needs Due to 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. Whereas a component-based methodology exists for the functional behavior of embedded systems, a compatible approach to modular performance analysis and design for predictability is missing. Only first steps are available (Rakib et al., 2004). A framework for performance analysis is missing which enables composition on several layers:

- *Method Composition*: Different parts of the system can be modeled using different performance analysis methods.
- *Process Composition*: If a stream of events needs to be processed by several consecutive application processes, the associated performance components follows the same structural composition.

- *Scheduling Composition*: Within one implementation, different scheduling methods can be combined, even within one computing resource (hierarchical scheduling); the same property holds for the scheduling and arbitration of communication resources.

- *Resource Composition*: One implementation can consist of different heterogeneous computing and communication resources. They can be composed in a similar way as processes and scheduling methods.

Acknowledgments

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. Peter Marwedel, Wolfgang Halang, Peter Puschner, and Stephan Thesing provided valuable discussions during the writeup of this paper.

References

Atanassov, P., and Puschner, P. 2001. Impact of DRAM refresh on the execution time of real-time tasks. In *Workshop on Application of Reliable Computing and Communication*.

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

Baruah, S. 2003. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems* 24(1): 93–128.

Benveniste, A., and Berry, G. 1991. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE* 79(9): 1270–1282.

Berry, G., and Gonthier, G. 1992. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming* 19(2): 87–152.

Bremond, P., and Lee, I. 1997. A process algebra of communicating shared resources with dense time and priorities. *Theoretical Computer Science* 189: 179–219.

Bucci, G., and Vicario, E. 1995. Compositional verification of time-critical systems using communicating time Petri nets. *IEEE Transactions on Software Engineering* 21(12): 969–992.

Buttazzo, G., 1997. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Boston: Kluwer Academic Publishers.

Chakraborty, S., Künzli, S., and Thiele, L. 2003. A general framework for analysing system properties in platform-based embedded system designs. In *Proceedings of the 6th Design, Automation and Test in Europe (DATE)*. Munich, Germany.

Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., and Wilhelm, R. 2001. Reliable and precise WCET determination for a real-life processor. In T. Henzinger and C. Kirsch (eds). *Embedded Software. Lecture Notes in Computer Science*, Vol. 2211. Springer, pp. 469–485.

Girault, A., and Ménier, C. 2002. Automatic production of globally asynchronous locally synchronous systems. In A. Sangiovanni-Vincentelli and J. Sifakis (eds), *2nd International Workshop on Embedded Software, EMSOFT'02. LNCS*, Vol. 2491. Grenoble, France: Springer-Verlag, pp. 266–281.

Halang, W. A. 2004. Simplicity considered fundamental to design for predictability. Dagstuhl Workshop Design for Predictability, http://www.dagstuhl.de/03471/Talks/.

Heckmann, R., Langenbach, M., Thesing, S., and Wilhelm, R. 2003. The influence of processor architecture on the design and the results of WCET Tools. *IEEE Proceedings on Real-Time Systems* 91(7): 1038–1054.

Hennessy, J., and Patterson, D. 2003. *Computer Architecture: A quantitative approach.* Morgan Kauffmann.

Henzinger, T., Horowitz, B., and Kirsch, C. 2001. Embedded control systems development with Giotto. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems LCTES*.

Henzinger, T., Nicollin, X., Sifakis, J., and Yovine, S. 1994. Symbolic model checking for real-time systems. *Information and Computation* 111: 193–244.

Kopetz, H. 1997. *Real-Time Systems Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers.

Lahiri, K., Raghunathan, A., and Dey, S. 2001. Evaluation of the traffic performance characteristics of system-on-chip architectures. In *Proceedings of the International Conference VLSI Design*. pp. 29–35.

Larsen, K., Petterson, P., and Yi, W. 1997. UPAAL in a nutshell. *Journal on Software Tools for Technology Transfer* 1: 134–152.

Lee, E. 2001. Overview of the Ptolemy Project. Technical Report UCB/ERL M01/11, University of California Berkeley.

Lee, E., and Messerschmitt, D. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers* C-36(1): 24–35.

Lee, I., Bremond, P., and Gerber, R. 1994. A process algebraic approach to the specification and analysis of resource-bound real-time systems. In *Proceedings of the IEEE*.

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

Lundqvist, T. 2002. A WCET analysis method for pipelined microprocessors with cache memories. Ph.D. thesis, Chalmers University of Technology, Göteborg, Sweden.

Maler, O., Pnueli, A., and Sifakis, J. 1995. On the synthesis of discrete controllers for timed systems. In E. Mayr and C. Puech (eds), *STACS95, Vol. 900 of Springer LNCS*. Springer Verlag, pp. 292–242.

Pop, P., Eles, P., and Zeng, P. 2003. Schedulability analysis and optimization for the synthesis of multi-cluster distributed embedded systems. In *Proceedings of the Design Automation and Test in Europe Conference*. pp. 184–189.

Puschner, P., and Burns, A. 2002. Writing temporally predictable code. In *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. San Diego, CA, U.S.A., pp. 85–91.

Rakib, A., Parshin, O., Thesing, S., and Wilhelm, R. 2004. Component-wise instruction-cache-behavior prediction. In *Proceedings of 2nd International Symposium on Automated Technology for Verification and Analysis*, Taiwan

Richter, K., Jersak, M., and Ernst, R. 2002. A formal approach to MpSoC performance verification. *IEEE Computer*.

Schneider, J. 2003. Combined schedulability and WCET analysis for real-time operating systems. Ph.D. thesis, Saarland University.

Sriram, S., and Bhattacharyya, S. S. 2000. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker Inc.

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

Strehl, K., Thiele, L., Gries, M., Ziegenbein, D., Ernst, R., and Teich, J. 2001. Funstate—An internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9(4): 524–544.

Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, F., Langenbach, M., Wilhelm, R., and Ferdinand, C. 2003. An abstract interpretation-based timing validation of hard real-time avionics software systems. In *Proceedings of the Performance and Dependability Symposium, San Francisco, CA*.

Thiele, L., Chakraborty, S., Gries, M., Maxiaguine, A., and Greutert, J. 2001. Embedded software in network processors—models and algorithms. In *Proceedings of the 1st Workshop on Embedded Software (EMSOFT)*. LNCS 221. Lake Tahoe, CA, U.S.A., Springer Verlag, pp. 416–434.

Thiele, L., Chakraborty, S., and Naedele, M. 2000. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*. Vol. 4. pp. 101–104.

Tindell, K., and Clark, J. 1994. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming—Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)* 40: 117–134.

Ungerer, T., Robic, B., and Silc, J. 2003. A survey of processors with explicit multithreading. *ACM Computer Survey* 35(1), 29–63.

VCC, The Cadence Virtual Component Co-design (VCC). http://www.cadence.com/products/vcc.html.

Wehmeyer, L., and Marwedel, P. 2004. Influence on onchip scratchpad memories on WCET prediction. In *Euromicro Workshop WCET 2004*. Catania, Sicily.

Wilhelm, R., Engblom, J., Thesing, S., and Whalley, D. 2004. The Determination of Worst-Case Execution Times—Introduction and Survey of Available Tools (submitted).

Wilhelm, R., Engblom, J., Thesing, S., and Whalley, D. B. 2003. Industrial requirements for WCET tools—answers to the ARTIST questionnaire. In *WCET 2003*. pp. 25–29.

Ziegenbein, D., Richter, K., Ernst, R., Thiele, L., and Teich, J. 2002. SPI—a system model for heterogeneously specified embedded systems In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*.