# FROM TIME-TRIGGERED TO TIME-DETERMINISTIC REAL-TIME SYSTEMS

Peter Puschner and Raimund Kirner
*Vienna University of Technology, A-1040 Vienna, Austria*
{peter, raimund}@vmars.tuwien.ac.at

**Abstract** With the increased use of powerful, performance-optimized hardware components in embedded systems, timing prediction is getting more and more complex. Thus while the execution speed of software is generally increasing, it is getting more and more difficult (if not infeasible) to perform an accurate and safe timing analysis of software that runs on those high-end embedded computer systems.

This paper presents a very rigid software execution model for building distributed hard real-time subsystems that are time predictable. The software model is based on the time-triggered communication model. It uses a purely time-triggered input-output interface and relies on single-path code (code that is free from input-data dependent control flow) in both the operating system and application software. Tasks are only preempted at pre-planned task preemption points and a simple clock synchronization keeps the operations of the hard real-time subsystem in synchrony with the real-time environment.

The proposed execution model yields software that is time-predictable by construction. Verifying temporal correctness and tracing the timing behavior of this software is trivial.

**Keywords:** Real-time systems, time-triggered architecture, determinism, time predictability.

## 1. INTRODUCTION

The computer architectures used in embedded systems are becoming increasingly complex. Modern microcontrollers for embedded systems are built around powerful superscalar CPU cores that use a number speedup features including instruction pipelines, caches and branch prediction, whose actual impact on the speedup of a particular block of code strongly depends on the processor's execution history, i.e., the stream of instructions the processor executed before the block. As the number of different execution histories at each point of a program can be enormous, an analysis of all possible behaviors of a piece of software running on such a high-end computer is generally extremely difficult. Consequently, worst-case execution time (WCET) analysis (i.e., assessing the timing of single tasks assuming non-preempted execution) is dif-

ficult for such systems, and analyzing the timing of the whole system, i.e., including preemptions and their effects on overall timing, requires unmanageably high efforts.

While the mentioned complexity problem might seem intuitive for highly dynamic systems with event-triggered task activation and scheduling, it even applies to very simple systems, e.g., time-triggered systems that use a very simple table-driven task activation. It thus seems to become more and more difficult to argue about timing guarantees and the system safety of embedded systems. To avoid the risk of missing a critical deadline, the use of very defensive estimates about resource needs and an over-dimensioned resource planning are becoming a necessity, unless simpler execution models are found.

In the light of this unsatisfactory situation we started to search for a real-time systems architecture which would have time-predictability as its number one property, i.e., temporal correctness (the absence of timing faults) should be easy to validate. All other properties, including performance in the classical sense should come second. This paper presents a software/hardware architecture for safety-critical hard real-time systems that came out of this work.

The only interface of the proposed architecture is a time-triggered state message interface that blocks all asynchronous external control signals that would otherwise disrupt the deterministic timing of the subsystem (Section 2). The software of the proposed architecture builds on a simple task model and table-driven static scheduling (Section 3.1) as well as on a deterministic code execution scheme (single-path code) in applications and in the operating-system code (Section 4). The architecture further uses deterministic task preemption, i.e., the number of instructions executed between each pair of preemptions points is planned offline and therefore exactly known (Section 5). A simple clock synchronization keeps the operations of the hard real-time subsystem synchronized with the clock of the environment (Section 6).

## 2. THE SAFETY-CRITICAL SUBSYSTEM INTERFACE

In this section we describe the interface of the proposed architecture. As a prerequisite for building a time-predictable (sub)system, the interface of this system has to be predictable as well.

The following description uses the model and terminology of the DECOS integrated architecture (6) for which our work was originally conceived. This does, however, not mean that our work is only useful in the context of DECOS. On the contrary, the architecture model can be adopted to any architecture that provides a time-triggered state message interface.

## 2.1 THE CONNECTOR UNIT

A DECOS component consists of two separated subsystems, the safety-critical subsystem for executing all safety-critical tasks and the non safety-critical subsystem for performing all other, non-critical services. Both types of subsystems are connected to the rest of the distributed computer system via so-called connector units. The connector units realize the architectural services of the distributed architecture, comprising the predictable transportation of messages, the fault-tolerant clock synchronization, fault isolation, and the consistent diagnosis of node failures (6).

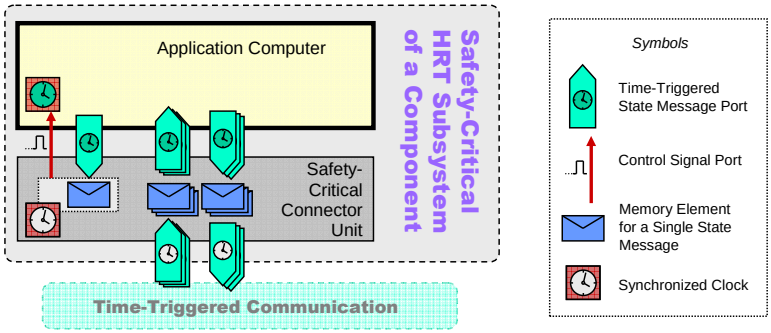


*Figure 1.* Interfacing between the Safety-Critical Hard Real-Time Subsystem of a Component and the Time-Triggered Communication Channel.

Within this paper our focus is on the safety-critical subsystem of a component (see Figure 1). This is where time predictability is needed. The application computer of this subsystem communicates with its environment solely via the safety-critical connector unit. The connector unit provides the following services in support of the time-predictable software architecture of the application computer.

- The connector unit implements a temporal firewall interface (5) for all data elements exchanged between the application computer and the communication subsystem. The read and write operations of the communication subsystem access the memory elements of the temporal firewalls only at predefined times, according to the a-priory known time-triggered communication schedule (in Figure 1 envelopes represent the memory elements and the arrows marked with light clocks show the accesses of the communication subsystem to the firewalls).

- The time windows during which the communication system accesses the memory elements of the connector unit are known for each of temporal firewall.

- The communication system provides a time-signal service to the application computer. A dedicated memory element in the connector unit can be written to set the timer (Figure 1, left). When the global system time reaches the timer value, the connector unit sends an interrupt signal over the signal port to the application processor.

# 3. A TIME-PREDICTABLE APPLICATION COMPUTER

The timing of the actions performed by a computer system depends on both, the software running on the computer and the properties of the hardware executing the software (8). We therefore list the hardware and software features that in combination allow us to make an application computer time-predictable.

## 3.1 HARDWARE ARCHITECTURE

A central idea of our approach is to obtain time predictability by using a software architecture that has an invariable control flow (see below). As a consequence of using this restrictive software model, we can allow for the use of hardware features that are otherwise considered as being "unpredictable" (e.g., instruction caches) and yet build systems whose timing is invariable. So the idea is to keep hardware restrictions and modifications within limits (e.g., we restrict caches to direct-mapped caches but do not demand special hardware modifications as, for example, needed for the SMART cache (2)). To support our execution model, the following hardware properties have to be fulfilled:

- The execution times of instructions do not depend on the values of the operands.

- The CPU supports a conditional move instruction or a set of predicated instructions that have invariable execution times.

- Instruction caches are either direct mapped or set-associative with LRU replacement strategy.

- Memory access times for data are invariable for all data items. (In our view, this is the strongest limitation at the moment. We will try to relax this in future work).

- CPU has a counter that counts the number of instructions executed. The counter can be reset and used to generate an interrupt when a given number of instructions has been completed.

## 3.2 THE SOFTWARE ARCHITECTURE

To construct a time-predictable computer system while being not more restrictive about the hardware than explained above, we need to be very strict about the software structure. In fact, the proposed software architecture does not allow for any decisions in the control flow whose outcome has not already been determined before the start of the system. This property is true for both the application tasks and the operating system. Even task preemptions are implemented in a way that does not allow for any timing variation between different task invocations.

**Task Model.** The structure of all tasks follows the simple-task model found in (4). Tasks never have to wait for the completion of an input/output operation and do never block. There are no statements for explicit input/output or synchronization within a task. It is assumed that the static schedule of application tasks and kernel routines ensures that all inputs for a task are available when the task starts and that outputs are ready in the output variables when the task completes. The actual data transfers for input and output are under control of the operating system and are scheduled before respectively after the task execution.

An important and unique property of our task model is that all tasks have only a single possible execution path. By translating the code of all real-time tasks into single-path code we ensure that all tasks follow the only possible, pre-determined control flow during execution and have invariable timing. For more details about the single-path translation see Section 4.

**Operating System Structure.** If not properly designed, the activities of the operating system can create a lot of indeterminism in the timing of a computer system. We have therefore been very restrictive in the design of the operating system and its mechanisms.

Predictability in the code execution of the operating system is achieved by two mechanisms. First, single-path coding is used wherever possible. Second, all data that are relevant for run-time decisions of the operating system are computed at compile time. These data include the pre-determined times for I/O, task communication, task activation, and task switching. They are stored in static decision tables that the operating system interprets at runtime.

Task communcation and I/O is implemented by simple read and write operations to specific memory locations. As these memory accesses are prescheduled together with the application tasks, no synchronization and no waiting is necessary at run time.

The two greatest challenges in building a fully predictable operating system were in maintaining time-predictability in case of task preemptions and keep-

ing the activities of the application computer in synchrony with its environment (the rest of the system).

- To maintain the deterministic timing in the presence of preemptions it was necessary to introduce a mechanism that allows for a precise preemption when a given number of instructions have finished execution, i.e., planning preemptions at specific times of the CPU clock turned out to be insufficient (see Section 5).
- The programmable time interrupt provided by the communication system is used to synchronize the operation of the application computer with the global time base (see Section 6).

### 3.3 TOOL SUPPORT

The software structure of our architecture is very specialized. Code generation for an application therefore needs to be supported by a number of tools:

- To generate single-path code, either a special compiler or a code conversion tool that converts branching code into single-path code is needed.
- A tool for worst-case execution-time analysis (either a static analyzer or a measurement tool) returns the execution times of the tasks and the operating system routines.
- An off-line scheduler generates the tables that control all operations of the application computer. The scheduler has to resolve all precedence and mutual exclusion constraints between task pairs as well as tasks and the communication system. It further has to plan all preemptions, thereby taking into account the effects of the preemptions on the system timing.

## 4. DETERMINISTIC SINGLE-PATH TASK EXECUTION

As all branches in the control flow of a task may potentially cause variable timing, we translate the code of all tasks into so-called single-path code (9). The code resulting from the single-path translation has only a single execution trace, hence the name single-path translation.

The strategy of the single-path translation is to remove input-data dependencies in the control flow. To achieve this, the single-path translation replaces all input-data dependent branching operations in the code by predicated code. It serializes the input-dependent alternatives of the code and uses predicates (instead of branches) and, if necessary, speculative execution to select the right code to be executed at runtime.

For pieces of code with an if-then-else semantics, a similar transformation, called if-conversion, has been used before to avoid pipeline stalls in processors with deep pipelines (1). In addition to code with if-then-else semantics the single-path translation transforms loops with input-data dependent control conditions. This transformation yields loops with constant iteration counts, again with a single execution path (7).

As a prerequisite for the single-path translation of a piece of code, the upper bounds for the number of iterations of all loops have to be available. These numbers can either be computed by a semantic analysis of the code or can be provided by the programmer in the form of annotations, in case an automated analysis is not possible or available.

## 5. PREDICTABLE TASK PREEMPTION

The idea of predictable task preemption is to preempt each task that needs to be preempted at the same points in time in each execution cycle of the static schedule. By doing so, the overall timing of all repetitive executions of the cyclic schedule would also be invariable.

Our original plan was to implement the predictable task preemption by using the CPU clock for task preemptions, i.e., preempt tasks always when the CPU clock assumed one of the values given in the preemption-time tables of the operating system. It turned out, however, that on hardware with instruction cache this simple preemption strategy does not guarantee temporal predictability. It may lead to oscillating task execution times, see (3).

Figure 2 shows the execution of two tasks, $T_1$ and $T_2$. $T_1$ preempts $T_2$ at the pre-scheduled time marked by the dashed line on the left. $T_2$ resumes after $T_1$ has completed its execution. Let us assume that $T_1$ and $T_2$ execute instructions that map to the same cache line of a direct mapped cache ($T_2$ executes these instructions twice, e.g., in a loop). The execution of these conflicting instructions is marked by the dark boxes; upon a cache miss, the execution time of the instruction increases, which is shown by the striped boxes.

Let us assume the very first activation of our schedule leads to the execution shown in Scenario A. The first access of $T_2$ to the conflicting cache line leads to a cache miss, and so do the other accesses by $T_1$ respectively $T_2$ (The latter misses are due to the order in which the tasks access memory).

When the schedule is repeated, $T_2$ has a cache hit on the first memory access. So $T_2$ makes faster progress and the second access to the conflicting address occurs before $T_2$ is preempted, thus resulting in a hit, too. $T_1$ then executes with a cache miss, and as $T_2$ has already completed its two critical memory accesses, the instruction of $T_1$ remains in cache (see Scenario B). On the next execution of the schedule, $T_2$ has a cache miss when accessing the conflicting

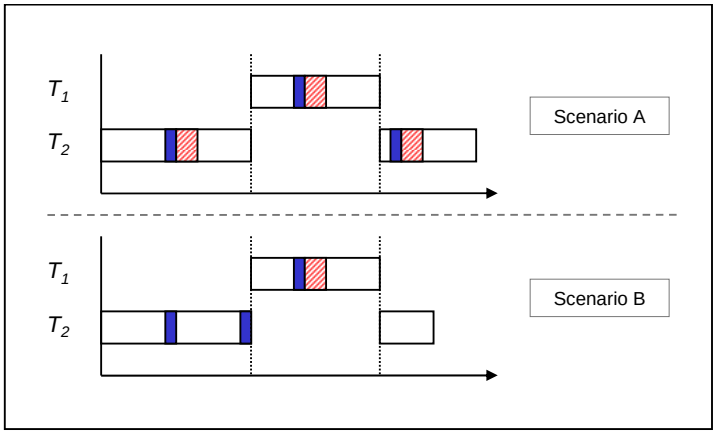


*Figure 2.* Task preemption by clock interrupt.

address, and so Scenario A is repeated. Following Scenario A, Scenario B happens again, and so on. The timing does not stabilize.

We found that preempting tasks based on the number of instructions executed yields the desired time-predictable behavior. So instead of using a clock we count the number of instructions completed. Preemptions happen when the value of this counter matches an entry in the scheduling table.

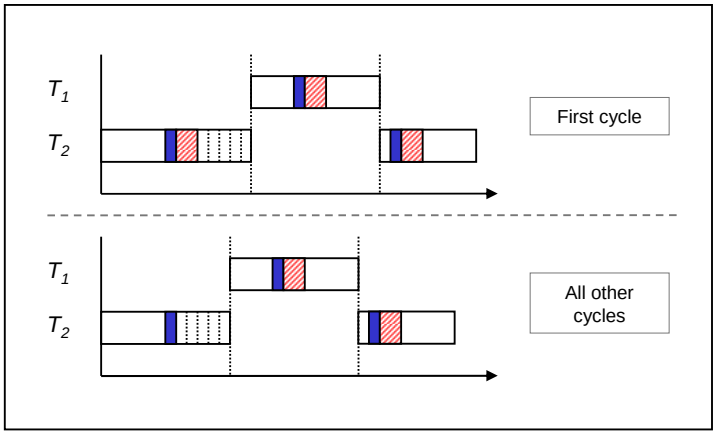


*Figure 3.* Task preemption by instruction counter.

Figure 3 shows the schedule for our example, using an instruction counter. Still, the timing of the second execution of the schedule differs from the first, in which we have an initial cache miss. From the second execution on, how-

ever, the execution always starts from the same cache state and has a constant execution time.

## 6. MAINTAINING SYNCHRONY WITH THE GLOBAL CLOCK

To keep in phase with the rest of the system, the execution of all actions of the application computer has to be synchronized to the global time reference that is provided by the time-triggered communication system. Deviations of the local clock from the global time are corrected. This clock synchronization uses the clock signal from the communication subsystem. Whenever the programmed timer expires the clock of the application computer is reset and a new round of the execution cycle is started.

In order to maintain the time predictability of the software, the clock synchronization must not interfere with the software execution on the application computer (i.e., the clock signal, that it is not in synchrony with the CPU clock, must not preempt any software running). As a consequence, the application computer has to be idle when a clock signal arrives. This is achieved by using schedules that consist of alternative intervals of task activity and inactivity, where the latter are used as synchronization windows. Clock synchronization interrupts have to be configured such that the clock signals arrive inside the synchronization windows even in case of the worst-case deviation of the local time from the global time.

## 7. SUMMARY AND CONCLUSION

In this paper we described a software architecture for safety critical hard real-time systems. This software architecture relies on time-triggered communication and uses the static task-activation scheme of a cyclic executive. Further, the operating system design and the single-path translation of code, together with a task preemption mechanism that triggers preemptions based on the number of instructions executed and the simple master clock synchronization make it possible to build fully time-deterministic computer systems on powerful, state-of-the-art hardware. These computer systems are easy to analyze for their timing and their timing properties and correctness can easily be traced. So they can safely be used in time-critical systems for which temporally correct behavior has to be guaranteed.

## ACKNOWLEDGMENTS

This work has been supported in part by the European IST project ARTIST2 under project No. IST-004527 and the European IST project DECOS under project No. IST-511764.

## REFERENCES

[1] Allen, J., Kennedy, K., Porterfield, C., and Warren, J. (1983). Conversion of Control Dependence to Data Dependence. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, pages 177–189.

[2] Kirk, D. B. (1989). Smart (strategic memory allocation for real-time) cache design. In *Proc. 10th Real-Time Systems Symposium*, pages 229–237, Santa Monica, CA, USA.

[3] Kirner, Raimund and Puschner, Peter (2006). Time-Predictable Task Preemption in Real-Time Systems with Instruction Cache. Research Report 27/2006, Technische Universitat Wien, Institut fur Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria.

[4] Kopetz, H. (1997). *Real-Time Systems*. Kluwer Academic Publishers.

[5] Kopetz, Hermann and Nossal, Roman (1997). Temporal Firewalls in Large Distributed Real-Time Systems. In *Proc. 6th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 310–315.

[6] Obermaisser, Roman, Peti, Philipp, and Kopetz, Hermann (2005). Virtual Networks in an Integrated Time-Triggered Architecture. In *Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 241–253.

[7] Puschner, Peter (2002). Transforming execution-time boundable code into temporally predictable code. In Kleinjohann, Bernd, Kim, K.H. (Kane), Kleinjohann, Lisa, and Rettberg, Achim, editors, *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Kluwer Academic Publishers. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).

[8] Puschner, Peter and Burns, Alan (2000). A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128.

[9] Puschner, Peter and Burns, Alan (2002). Writing temporally predictable code. In *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91.