Samarjit Chakraborty
Jörg Eberspächer *Editors*

Advances in Real-Time Systems

Springer

Advances in Real-Time Systems

Samarjit Chakraborty • Jörg Eberspächer
Editors

Advances in Real-Time Systems

Springer

Editors
Samarjit Chakraborty
TU München
LS für RealzeitComputersysteme
Arcisstr. 21
80290 München
Germany
Samarjit.Chakraborty@rcs.ei.tum.de

Prof. Dr. Jörg Eberspächer
TU München
LS Kommunikationsnetze
Arcisstr. 21
80290 München
Germany
joerg.eberspaecher@tum.de

ISBN 978-3-642-24348-6 e-ISBN 978-3-642-24349-3
DOI 10.1007/978-3-642-24349-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011942997

Mathematics Subject Classification (2000): 01-01, 04-01, 11Axx, 26-01

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To Georg Färber
on the occasion of his appointment as
Professor Emeritus at TU München after 34
illustrious years as the Head of the Lehrstuhl
für Realzeit-Computersysteme

Preface

This book is a tribute to Georg Färber on the occasion of his appointment as Professor Emeritus at TU München after 34 years of service as the Head of the Lehrstuhl für Realzeit-Computersysteme.

Georg Färber was born in 1940 and obtained his PhD in 1967 from TU München. In an illustrious career, spanning over 40 years, Prof. Färber contributed significantly to the area of real-time and embedded systems, in terms of both education and research. His early research dealt with specification and design of embedded systems and mapping of such specifications onto appropriate hardware and software architectures. In particular, he was active in the area of computer-aided process control, with a focus on distributed and fault-tolerant systems. Later, his interests broadened into real-time capturing and interpretation of visual information, especially in the context of robot vision. More recently, he has worked in the area of cognitive automobiles, e.g., driver assistance systems and autonomous cars. Since a number of years, Prof. Färber also took a keen interest in topics at the intersection of engineering and medicine, in particular in those related to "e-health".

In parallel to his research and teaching activities, Prof. Färber was a highly successful entrepreneur. In 1969, along with his brother Eberhard Färber, he founded PCS-Computersysteme GmbH. This company was once considered to be among the most innovative IT companies in Munich and led to more than 20 other spin-offs. Among other products, PCS developed the first UNIX Workstations in Germany (named CADMUS), which were the only European alternatives to US-based products for a long time. In 1986, Mannesmann/Kienzle became the majority stakeholder of PCS, and Georg Färber provided the technical leadership during 1988-89, while on leave from TU München.

In addition to authoring one of the earliest books on real-time systems, Georg Färber served as the editor of the journal "Information Technology" and is a member of the Board of Trustees of the Fraunhofer-Institute for Information and Data Processing (Fraunhofer IITB). He has also served in various – often advisory – capacities at the DFG, the Max Planck Society, and in several other scientific and industrial councils and German government agencies.

Given Georg Färber's remarkable achievements and his reputation, we invited a number of well-known researchers to contribute a collection of chapters reflecting the state of the art in the area of real-time systems. These chapters cover a variety of topics spanning over automotive software and electronics, software timing analysis, models for real-time systems, compilation of real-time programs, real-time microkernels, and cyber-physical systems. We believe that this collection can serve as a reference book for graduate-level courses. It will also be helpful to both researchers in the academia and practitioners from the industry.

Munich, Germany

Samarjit Chakraborty
Jörg Eberspächer

Contents

Part I
Theoretical Foundations

Chapter 1
System Behaviour Models with Discrete and Dense Time

Manfred Broy

1.1 Introduction and Motivation

The notion of *system* is present in many scientific disciplines. Biology speaks of *biological system*. There are terms like *economic system*, *ecological system*, *logical system*. The whole world can be understood as a *dynamical system*. Describing systems, their structure and their dynamics by appropriate models is a major goal of scientific disciplines. However, the different disciplines use quite different notions, concepts, and models of systems. Mathematics, has developed differential and integral theory over the centuries as one way of modelling and studying systems in terms of mathematical models. Relevant system aspects are captured by real valued variables that change dynamically and continuously depending on the parameter of time. This way the system dynamics and the dependencies between the system variables can be described by differential and integral equations. The engineering discipline of modelling and designing systems applying these mathematical modelling concepts is control theory.

Another way to capture and specify systems in terms of discrete events is logic. Logic was developed originally as a branch of philosophy addressing the art and science of reasoning. As a discipline, logic dates back to Aristotle, who established its fundamental place in philosophy. Historically, logic was intended as a discipline of capturing ways of thinking of human beings aiming at crisp lines of arguments. Over the centuries, logic was further developed mainly as a basis for carrying out proofs by formal logical deduction, leading to mathematical logic with its foundations *propositional logic* and *predicate logic* in its many variations. With the arrival of digital systems, logic became more and more also a technical discipline

M. Broy (✉)
Institut für Informatik, Technische Universität München, 80290 München, Germany
e-mail: broy@in.tum.de

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_1,

for designing logical circuits and electronic devices. With software becoming more significant in all kinds of information processing applications, more general forms of logic were invented as the conceptual basis for engineering information processing systems. For information processing systems, including embedded systems, many different aspects are captured by various forms of logic both at the technical level and the application domain level.

The logic of the behaviour of complex discrete event systems can be captured by families of discrete events that are causally related with logically specified properties.

In contrast to real-valued function based models of systems such as applied in control theory, by logics we can capture better ways of arguing about systems. When capturing requirements about systems such as in requirements engineering in terms of natural language, a logical way of making requirements precise is more appropriate than modelling behaviours by real time parameterized continuous functions. On the other hand, when solving problems in control theory we finally aim at mathematical descriptions of the system dynamics by differential and integral equations.

In this paper we aim at a step integrating logical system views with system views based on differential and integration calculus such as used in control theory. In contrast to well-known approaches, where the step from a description of systems by real valued functions into digital systems is performed using techniques of discretization such as in numerical analysis, we are rather interested in the step from a logical description of requirements to modelling of system behaviours by continuous real valued functions and in a formal relationship between the logical description of requirements and the real valued functions describing systems dynamics.

We are aiming at systems that interact with their environment. Streams of data for exchanging input and output capture this interaction. We consider both discrete and continuous streams.

One way to describe interactive systems is state machines with input and output also known as Mealy or Moore machines. These are machines, where in given states input triggers transitions generating new states and output. I/O state machines define computations, being infinite runs by their transitions. Given an initial state and an infinite stream of input messages I/O machines produce infinite streams of states and infinite streams of output messages. In so-called interface abstractions we forget about the chain of states of computations and just keep the relation between the input and output streams. This yields what we call an *interface abstraction*. Talking only about interface properties we can formulate so-called *interface assertions*, which describe logical relationships between the input and the output streams. They specify the interface behaviour of I/O state machines.

In the following, we aim at techniques for modelling interfaces of discrete as well as dense and continuous interactive systems. We define models of such systems and discuss concepts how to specify, compose and relate them.

1.2 Hybrid Interactive Behaviours

The essential difference between a *non-interactive* and an *interactive computation* lies in the way in which input is provided to the computing device before or during the computation and how output is provided by the computing device to its environment during or after the computation.

1.2.1 Streams, Channels, and Histories

In this section we briefly introduce the notions of stream, channel, and history.

Throughout this paper, we model interaction by message exchange over sequential communication media called *channels*. These message streams can be based on discrete or dense time and may be discrete or – in the case of dense time – continuous. In general, in an interactive computation, several communication channels may participate. In our setting, a channel is simply an identifier for a communication line. In the following, we distinguish input from output channels. Throughout the paper, let I be a set of input channels, O be a set of output channels and $\mathtt{M}$ be a set of messages.

A stream may be discrete or continuous. A stream has a data type that determines the type of messages it carries. A stream can be finite or infinite.

1.2.1.1 Discrete Finite and Infinite Streams

Let $\mathtt{M}$ be a set of elements, called *messages*. We use the following notation (where set $\mathbb{N}_+$ is specified by $\mathbb{N}_+ = \mathbb{N}\backslash\{0\}$):

$\mathtt{M}^*$ denotes the set of finite sequences, with elements of the set $\mathtt{M}$, including the *empty* sequence $\langle\rangle$,
$\mathtt{M}^\infty$ denotes the set of infinite sequences with elements of the set $\mathtt{M}$ (that are represented by the total mappings $\mathbb{N}_+ \to \mathtt{M}$).

By

$$\mathtt{M}^\omega = \mathtt{M}^* \cup \mathtt{M}^\infty$$

we denote the set of discrete untimed streams of elements of set $\mathtt{M}$. Streams of elements of set $\mathtt{M}$ are finite or infinite sequences of elements of set $\mathtt{M}$.

The set of streams has a rich algebraic and topological structure. We make use of only parts of this structure. We introduce concatenation $\hat{}$ as an operator on streams written in infix notation:

$$\_\hat{}\_ : \mathtt{M}^\omega \times \mathtt{M}^\omega \to \mathtt{M}^\omega$$

On finite streams concatenation is defined as usual on finite sequences. For infinite streams r, s: $\mathbb{N}_+ \to \mathtt{M}$ and finite stream $x \in \mathtt{M}^*$ we define the result of concatenation for infinite streams as follows:

$$s\hat{}x = s$$

$$s\hat{}r = s$$

$$\langle x_1 \ldots x_n\rangle\hat{}\langle s_1 \ldots\rangle = \langle x_1 \ldots x_n s_1 \ldots\rangle$$

We may represent finite streams by total functions $\{1, \ldots, t\} \to M$ or also by partial functions $\mathbb{N}_+ \to M$ and infinite streams by total functions $\mathbb{N}_+ \to M$.

Streams are used to represent the flow of messages sent over a communication channel during the lifetime of a system. Of course, in concrete physical systems this communication takes place in a specific time frame. Hence, it is often convenient or even essential to be able to refer to time. Moreover, with an explicit notion of time the theory of feedback loops in networks of communicating components gets even simpler (see [1]). Therefore we prefer to work with *timed streams*.

Streams represent histories of communications of data messages transmitted within a time frame. Given a message set M of type T a *uniformly discretely timed stream* is a function

$$s : \mathbb{N}_+ \to M^* \quad \text{i.e. } s \in (M^*)^\infty$$

Actually, given stream $s \in (M^*)^\infty$ for every time interval $t \in \mathbb{N}_+$ the sequence s(t) denotes the sequence of messages communicated in the time interval t in the stream s. Let $\mathbb{R}_+ = \{t \in \mathbb{R} : t \geq 0\}$ be the set of positive real numbers. The basic idea here is that s(t) represents the sequence of messages communicated in the real time interval $[(t-1)\delta : t\delta[$ where $\delta \in \mathbb{R}_+$ is called the *time granularity* of the stream s. A *partial stream* is given for $t \in \mathbb{N}$ by a mapping

$$s : \{1, \ldots, t\} \to M^* \quad \text{i.e. } s \in (M^*)^*$$

It represent a communication history till time step t. Throughout this paper we work with a number of simple basic operators and notations for streams and timed streams respectively that are briefly summarized below:

$\langle\rangle$	empty sequence or empty stream,
$\langle m\rangle$	one-element sequence containing m as its only element,
s.t	t-th element of the stream s (which is a message in case s is an untimed stream and a finite sequence of messages in case s is a timed stream),
#s	length of a stream
$s{\downarrow}t$	prefix of length t of the stream s (which is a sequence of messages of length t, provided $\#s \geq t$, in the case of an untimed stream and a sequence of t sequences in case s is a discretely timed stream),
$s \uparrow t$	the stream derived from s by deleting its first t elements (without the first t sequences of s in the case of a discretely timed streams)

In a uniformly discretely timed stream $s \in (M^*)^\infty$ it is specified in which time intervals which sequences of messages are transmitted. The timing of the messages within a time interval is not specified, however, only their order is observable.

1.2.1.2 Dense and Discrete Time Domains and Timed Streams

Every subset TD $\subseteq \mathbb{R}_+$ is called a *time domain*. A time domain TD is called *discrete,* if for every number $t \in \mathbb{R}_+$ the set

$$TD_t = \{x \in TD : x < t\}$$

is finite. Obviously, discrete time domains contain minimal elements.

A set S with a linear order $\leq$ is called *dense*, if the following formula holds

$$\forall x,\ y \in S : x < y \Rightarrow \exists z \in S : x < z < y$$

On $\mathbb{R}$ we choose the classical linear order $\leq$. If a nontrivial time domain TD is discrete, it is certainly not dense. Vice versa, however, if a time domain is not dense, it is not necessarily discrete.

Let M be a set of messages and TD be a time domain. A timed stream over time domain TD is a total mapping

$$s : TD \rightarrow M$$

TD is called the (time) domain of the stream s. We write dom(s) = TD. If TD is discrete, then the stream s is called *discrete*, too. If TD is dense, then s is called *dense*, too.

Stream s is called *continuous*, if TD is an interval in $\mathbb{R}_+$ and M is a metric space with distance function d such that s is a continuous function on the set TD in Cauchy's sense. More precisely s is continuous in $t \in \mathbb{R}_+$ if the following formula holds:

$$\forall \varepsilon \in \mathbb{R}, \varepsilon > 0 : \exists \delta \in \mathbb{R}, \delta > 0 : \forall x' \in TD : |x - x'| < \delta \Rightarrow d(s(x), s(x')) < \varepsilon$$

An interval based timed stream s is given by an interval [t: t'[with $t, t' \in \mathbb{R}_+$, $t \leq t'$, and by the time domain TD $\subseteq$ [t: t'[where

$$s : [t : t'[\rightarrow M$$

is a partial function and TD is its domain. We write then interval(s) = [t: t'[and dom(s) = TD. Note that for the partial functions we denote by interval(s) the set of potential arguments for function s while dom(s) $\subseteq$ [t: t'[defines the set of arguments for which function s is defined.

A stream s is called *permanent*, if interval(s) = dom(s). Then s is a total function on its interval(s).

A special case of a permanent stream s is one who is piecewise *constant* with only a discrete set of "discontinuities". Then we expect a discrete set of time points (with $t_0 = 0$):

$$\{t_i : i \in \mathbb{N}\}$$

such that in each interval $[t_i : t_{i+1}[$ the stream s is constant, i.e. $s(t) = d$ with some $d \in T$ where T is the type of the stream s. A typical example is $T = \mathbb{B}$ where stream s(t) signals whether a certain condition holds or does not at time t.

1.2.1.3 Operations on Timed Streams

For timed streams we define a couple of simple basic operators and notations that are summarized below:

$\langle\rangle$	empty sequence or empty stream with $dom(TD) = \emptyset$ and $interval(\langle\rangle) = [0 : 0[= \emptyset$,
$\langle m@t\rangle$	one-message stream containing m as its only message at time t with $dom(\langle m@t\rangle) = \{t\}$,
#s	number of messages in a stream (which is given by the cardinality $\lvert dom(s)\rvert$)
s.j	j-th element in the discrete stream s (which is uniquely determined provided $\#s \geq j$ holds),
$s{\downarrow}t$	prefix until time t of the stream s (which is also denoted by $s\lvert dom(s) \cap [0: t[$ where by $f\lvert M$ denotes the restriction of a function $f: \mathtt{D} \to \mathtt{R}$ to the set $M \subseteq \mathtt{D}$ of arguments),
$s{\uparrow}t$	the stream derived from stream s by deleting its messages until time t (which is $s\lvert dom(s) \setminus (dom(s) \cap [0: t[)$) with domain $dom(s) \setminus (dom(s) \cap [0: t[$ and the interval $interval(s) \setminus interval(s{\downarrow}t)$

Let $interval(s) = [0: t[$; by $s{\downarrow}t'$ we get a stream for the interval $[0: t'[$ and with $dom(s{\downarrow}t') = dom(s) \cap [0: t'[$. By PTS we denote the set of partial timed streams. Given time $t \in \mathbb{R}_+$, we denote by PTS(t) the set of partial streams that are either discrete with $dom(s) \subseteq [0: t[$ or that are permanent with domain $dom(s) = [0: t[$.

Given times $t, t' \in \mathbb{R}_+$ by $PTS[t: t'[$ we denote the set of partial streams with $interval(s) = [t: t'[$ that are either discrete with $dom(s) \subseteq [t: t'[$ or that are permanent on their domain $dom(s) = [t: t'[$.

This way we get the universe of streams over a given universe of messages types.

A *time shift* of a timed stream s by the time $u \in \mathbb{R}_+$ yields stream $s^{TM}u$ defined by the equations

$$interval(s^{TM}u) = [t + u : t' + u[\Leftarrow interval(s) = [t : t'[$$
$$dom(s^{TM}u) = \{t + u : t \in dom(s)\}$$

and for $t \in dom(s)$ defined by the equation

$$(s^{TM}u)(t + u) = s(t)$$

Given a stream s with interval $[t: t'[$ where $t' < \infty$ (otherwise $s\hat{}s' = s$) we define *concatenation* of stream s with a stream s' by the equation

$$\text{interval}(s\hat{}s') = [t : t''[\Leftarrow \text{interval}(s'^{TM}t') = [t''', t''[\wedge \text{interval}(s) = [t : t'[$$
$$\text{dom}(s\hat{}s') = \text{dom}(s) \cup \text{dom}(s'^{TM}t')$$

and for $t'' \in \text{dom}(s\hat{}s')$

$$(s\hat{}s')(t'') = s(t'') \Leftarrow t'' \in \text{dom}(s)$$
$$(s\hat{}s')(t'') = (s'^{TM}t')(t'') \Leftarrow t'' \in \text{dom}(s'^{TM}t')$$

This generalizes the operations on discrete streams to operations on timed streams.

1.2.1.4 Time Deltas and Delta Transactions

For timed streams their time granularity is of major interest. A discrete stream s has a guaranteed message distance δ if in each time interval of length δ at most one message or event occurs. Formally

$$\forall t \in \mathbb{R}_+ : \#(s|[t : t + \delta[) \leq 1$$

Here s|[t: t + δ[denotes the stream which is the result of restricting stream s (seen as a mapping) to the set [t: t + δ[.

In a time interval of length δ a communication stream is given by a finite sequence of messages, by a continuous function, by a discrete real time sequence, or a mixture thereof.

1.2.2 Channels

Generally, several communication streams may appear in a system. To distinguish and identify these streams we use channels. A channel is a named sequential communication medium. In logical formulas about systems, a channel is simply an identifier in a system that evaluates to a stream in every execution of the system.

Definition 1.1. Channel snapshot and channel history

Let C be a set of channels; given times t, t' $\in \mathbb{R}_+ \cup \{\infty\}$ with t < t' a channel snapshot is a mapping

$$x : C \to \text{PTS}[t : t'[$$

such that x(c) $\in$ PTS[t: t'[is a partial or total stream for each channel c $\in$ C. A snapshot is called *finite* if t' $< \infty$. By $\widehat{C}$[t: t'[the set of finite channel snapshots for channel set C for times t, t' is denoted. By $\widehat{C}$ the set of all channel snapshots for channel set C is denoted.

A complete channel history is a mapping

$$x : C \to PTS[0 : \infty[$$

such that x(c) is a timed stream for each channel $c \in C$. $\vec{\vec{C}}$ the set of complete channel histories for channel set C. $\vec{C}$ denotes the set of channel histories where all channels carry uniformly discretely timed streams $s \in (M^*)^\infty$. □

All operations and notations introduced for streams generalize in a straightforward way to channel histories applying them to the streams in the histories elementwise.

For instance given a channel history

$$x : C \to PTS[0 : \infty[$$

by x|[t: t‘[we denote a snapshot

$$x|[t : t‘[: C \to PTS[t : t‘[$$

where each stream x(c) for channels $c \in C$ is restricted to the interval [t: t‘[as follows:

$$(x|[t : t‘[)(c) = x(c)|[t : t‘[$$

The remaining operators generalize in analogy from streams to channel histories.

1.2.3 I/O-Behaviours: Interface Behaviours of Hybrid Systems

Let I be a set of typed input channels and O be a set of typed output channels. Figure 1.1 gives an illustration of the system where the channels are represented by arrows annotated with their names and their message types. In addition to the message types that show which elements are communicated via the channels we indicate which type of stream is associated with the channel – a discrete or a dense one. We use the prefix *Dsc* to indicate that a stream is discrete and *Prm* to indicate that it is permanent. For instance *Dsc* Bool is the type of a discrete stream of Boolean values while *Prm* Bool is the type of a permanent stream of Boolean values. If no prefix is used than nothing specific is assumed about the stream.

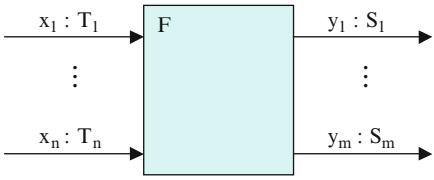


Fig. 1.1 Hybrid system with its channels

We represent hybrid system behaviours by functions:

$$F : \overrightarrow{I} \to \wp(\overrightarrow{O})$$

that model input and output of interactive nondeterministic computations. F maps every input history onto its set of output histories.

Definition 1.2. Causality

An I/O-behaviour F fulfils the property of *causality* if there exists some time distance $\delta \in \mathbb{R}$ with $\delta \geq 0$ such that the following formula holds for all histories $x, z \in \overrightarrow{I}$, $y \in \overrightarrow{O}$, $t \in \mathbb{R}_+$:

$$x{\downarrow}t = z{\downarrow}t \Rightarrow \{y{\downarrow}t + \delta : y \in F(x)\} = \{y{\downarrow}t + \delta : y \in F(z)\}$$

If the formula holds for $\delta = 0$ then F is called *causal* and if it holds for some delay $\delta > 0$ then F is called *strongly causal* and also *strongly δ causal.* □

We assume for I/O-behaviours that they fulfil the law of *strong δ causality* for some $\delta > 0$. Strong causality characterizes proper time flow and the fact that computation takes time. It captures the principle that a reaction to input can happen only after the input has been received. Since the output at time t is produced while the input in step t is provided, the output in step t must depend at most on input provided before time t.

A behaviour F is called *deterministic* if F(x) is a one element set for each input history x. Such a behaviour is equivalent to a function

$$f : \overrightarrow{I} \to \overrightarrow{O} \quad \text{where } F(x) = \{f(x)\}$$

f represents a deterministic I/O-behaviour, provided that for some $\delta \geq 0$ the δ causality property holds. Then the following property is valid:

$$x{\downarrow}t = z{\downarrow}t \Rightarrow f(x){\downarrow}t + \delta = f(z){\downarrow}t + \delta$$

As for nondeterministic behaviours this causality property models proper time flow.

Definition 1.3. Realizability

An I/O-behaviour F is called (strongly) *realizable*, if there exists a (strongly) causal total function

$$f : \overrightarrow{I} \to \overrightarrow{O}$$

such that we have:

$$\forall x \in \overrightarrow{I} : f(x) \in F(x).$$

f is called a *realization* of F. By [F] we denote the set of all realizations of F. An output history $y \in F(x)$ is called *realizable* for an I/O-behaviour F with input x, if there exists a realization $f \in [F]$ with $y = f(x)$. □

A δ causal function $f: \overset{\Rightarrow}{I} \to \overset{\Rightarrow}{O}$ with $\delta > 0$ provides a deterministic *strategy* to calculate for every input history x a particular output history $y = f(x)$. The strategy is called *correct for input x and output y* with respect to an I/O-behaviour F if $y = f(x) \in F(x)$. According to strong δ causality the output y can be computed inductively in an interactive computation. Only input $x{\downarrow}t$ received till time t determines the output till time $t+\delta$ and, in particular, the output at time $t+\delta$. In fact, f essentially defines a deterministic "abstract" automaton with input and output which is, in case of strong δ causality, actually a Moore machine. Strong δ causality guarantees that for each time t the output produced after time t till time $t+\delta$ does only depend on input received before time t.

Theorem 1.1. *Full Realizability*

Strongly δ causal functions $f: \overset{\Rightarrow}{C} \to \overset{\Rightarrow}{C}$ always have unique fixpoints $y = f(y)$.

Proof. This is easily proved by an inductive construction of the fixpoint $y = f(y)$ as follows. Since f is strongly causal output $f(x){\downarrow}\delta$ does not depend on x at all. So we define

$$y{\downarrow}\delta = f(x){\downarrow}\delta$$

for arbitrarily chosen input $x \in \overset{\Rightarrow}{C}$. Then history y is constructed inductively as follows: given

$$y{\downarrow}i\delta$$

we define

$$y{\downarrow}(i+1)\delta = f(x){\downarrow}(i+1)\delta$$

with arbitrary chosen input history x such that

$$x{\downarrow}i\delta = y{\downarrow}i\delta$$

Note again that then $f(x){\downarrow}(i+1)\delta$ does not depend on the choice of the history $x{\downarrow}(i+1)\delta$ due to strong δ causality of f. The construction yields history y such that the fixpoint equation $y = f(y)$ holds. Moreover, since the construction yields a unique result the fixpoint is unique. □

The construction indicates the existence of a computation strategy for the fixpoint y of f.

Definition 1.4. Full Realizability

An I/O-behaviour F is called *fully realizable*, if it is realizable and if for all input histories $x \in \overset{\Rightarrow}{I}$

$$F(x) = \{f(x) : f \in [F]\}$$

holds. Then also every output is realizable. □

Full realizability of a behaviour F guarantees that for all output histories $y \in F(x)$ for some input x there is a strategy that computes this output history. In other words, for each input history x each output history $y \in F(x)$ is realizable.

1.2.4 Hybrid State Machines with Input and Output

In this section we introduce the concept of a hybrid state machine with input and output via channels.

A hybrid state machine (Δ, Λ) with input and output communicated over a set I of input channels and a set O of output channels is given by a state space Σ, which represents a set of states, a set $\Lambda \subseteq \Sigma$ of initial states as well as a state transition function

$$\Delta : (\Sigma \times \widehat{I}) \to \wp(\Sigma \times \widehat{O})$$

For each state $\sigma \in \Sigma$ and each valuation $\alpha \in \widehat{\mathrm{I}}$ of the input channels in I by sequences a snapshot we obtain by every pair $(\sigma', \beta) \in \Delta(\sigma, \alpha)$ a successor state σ' and a valuation $\beta \in \widehat{\mathrm{O}}$ of the output channels consisting of the snapshot of messages produced on the output channels by the state transition. Such state machines are a generalization of *Mealy machines* (more precisely Mealy machines generalized to infinite state spaces and infinite input/output alphabets).

A state machine (Δ, Λ) is called:

- *Deterministic*, if, for all states $\sigma \in \Sigma$ and inputs α, both $\Delta(\sigma, \alpha)$ and Λ are sets with at most one element.
- *Total*, if for all states $\sigma \in \Sigma$ and all inputs α the sets $\Delta(\sigma, \alpha)$ and Λ are not empty; otherwise the machine (Δ, Λ) is called *partial*,
- A (generalized) *Moore machine*, if the output of Δ always depends only on the state and not on the current input of the machine. A Mealy machine is a Moore machine iff the following equation holds for all input sequences α, α' and output sequences β, and all states σ:

$$(\exists \sigma' \in \Sigma : (\sigma', \beta) \in \Delta(\sigma, \alpha)) \Leftrightarrow (\exists \sigma' \in \Sigma : (\sigma', \beta) \in \Delta(\sigma, \alpha'))$$

- *Time based* if the states $\sigma \in \Sigma$ in the state space contain a time attribute denoted by $\mathrm{time}(\sigma) \in \mathbb{R}$ such that for all $(\sigma', \beta) \in \Delta(\sigma, \alpha)$ we have $\mathrm{time}(\sigma) < \mathrm{time}(\sigma')$
- A δ *step timed state machine* for $\delta \in \mathbb{R}$ with $\delta > 0$ if (Δ, Λ) is a time based machine and if for all $(\sigma', \beta) \in \Delta(\sigma, \alpha)$ where

$$\mathrm{time}(\sigma) = \mathrm{j}\delta \text{ and } \mathrm{interval}(\alpha) = [\mathrm{j}\delta{:}(\mathrm{j}{+}1)\delta[$$

we get

$$\mathrm{time}(\sigma') = \mathrm{time}(\sigma) + \delta \text{ and } \mathrm{interval}(\beta) = [\mathrm{j}\delta{:}(\mathrm{j}{+}1)\delta[.$$

Hybrid state machines are a straightforward generalisation of Mealy machines.

1.2.5 Computations of State Machines

In this section we introduce the idea of computations for δ step timed state machines with input and output.

Figure 1.2 shows a computation of a δ step timed state machine with input and output. Actually a computation comprises three infinite streams:

- The infinite streams x of inputs: $x_1, x_2, \ldots \in \widehat{I}$
- The infinite streams y of outputs: $y_1, y_2, \ldots \in \widehat{O}$
- The infinite streams s of states: $\sigma_0, \sigma_1, \ldots \in \Sigma$

Note that every computation can be inductively generated given the input stream x_1, $x_2, x_3, \ldots$ and the initial state $\sigma_0 \in \Lambda$ by choosing step by step state σ_{i+1} and output y_{i+1} by the formula

$$(\sigma_{i+1}, y_{i+1}) \in \Delta(\sigma_i, x_{i+1}).$$

If the state machine is deterministic, then the computation is fully determined by the initial state σ_0 and the input stream x.

Each input history $x \in \overset{\Rightarrow}{I}$ specifies a stream $x_1, x_2, \ldots \in \widehat{I}$ of input snapshots by (for all $j \in \mathbb{N}$)

$$x_{j+1} = x|[j\delta : (j+1)\delta[$$

Given history x a computation of a state machine (Δ, Λ) generates a sequence of states

$$\{\sigma_j : j \in \mathbb{N}\}$$

and a stream $y_1, y_2, \ldots \in \widehat{O}$ of output snapshots where for all times $j \in \mathbb{N}$ we have:

$$(\sigma_{j+1}, y_{j+1}) \in \Delta(\sigma_j, x_{j+1}) \quad \text{and} \quad \sigma_0 \in \Lambda$$

This way every computation specifies an output history $y \in \overset{\Rightarrow}{O}$ that is uniquely specified by

$$y|[j\delta : (j+1)\delta[= y_{j+1}$$

The history y is then called an *output* of the computation of the state machine (Δ, Λ) for input x and initial state σ_0. We also say that the machine computes the output history y for the input history x and the initial state σ_0. This way we can associate an interface behaviour

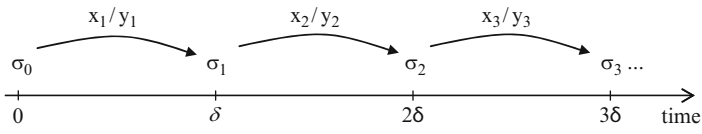


Fig. 1.2 Computation of a δ step timed I/O-machine

$$F_{(\Delta,\Lambda)} : \vec{\vec{I}} \rightarrow \wp(\vec{\vec{O}})$$

with state machine (Δ, Λ) defining $F_{(\Delta,\Lambda)}(x)$ as the set of all histories that are outputs of computations of machine (Δ, Λ) for input history x. System behaviour $F_{(\Delta,\Lambda)}$ is called the *interface abstraction* of hybrid state machine (Δ, Λ).

1.3 Logical Properties of the Interface Behaviour of Hybrid Systems and State Machines

Traditionally temporal logic is used to formulate properties about state transition systems that are represented by state machines. Usually in temporal logic state machines without input and output are considered such that the formulas of temporal logic specify properties for the infinite streams of states generated as computations by these state machines. There are several variations of temporal logic including so-called *linear time temporal logic*, which talks about the state traces of a state machine and *branching-time temporal logic*, which considers trees of computations defined by a state machine.

Since we are not mainly interested in states but rather in interface behaviour in terms of input and output streams of computations, classical temporal logic seems not the right choice for us. Moreover, temporal logic is limited in its expressive power. Although we could introduce a version of temporal logic that talks about input and output of computations, we prefer to talk about the interface behaviour in terms of more general and more expressive interface assertions, given by predicates that contain the channel identifiers of the syntactic interface of a system as identifiers for streams. An interface assertion is a formula, which refers to the input and output channels of the systems as variables for timed streams. This way we write logical formulas that express properties of the input and output streams of hybrid systems. These formulas are written in classical predicate logic using, in addition, a number of basic operators for streams.

1.3.1 Events in Continuous Streams

A permanent continuous stream s is represented by a continuous function. The values of the function define the valuation of an attribute of the system at any chosen point in time. This defines for each time a kind of interface state. An *event* then can be defined as “a significant change in state”.

With a continuous stream s we associate a certain (logical) event at time t if s fulfils a particular property at time t. Simple examples would be that the continuous stream s has reached a particular value at time t, or assumes a maximum or a minimum at time t, or that its values in an interval around t lie in a certain range.

In full generality, an event e is a predicate

$$e : PTS \times \mathbb{R}_+ \to \mathbb{B}$$

We say that an event occurs at time t in the hybrid stream s if e(s, t) holds. Events provide a logical view onto hybrid streams. By definition there are lots of events. Which events are of relevance for a system has to be determined depending on the logics of the application. Typical examples would be "target speed reached", "temperature too high", "speed too high" or "signal available".

Actually an infinite number of events may occur in a given stream. Given a stream s, a set of events E and a time $t \in \mathbb{R}_+$ the set

$$\{t' \in [t : t+\varepsilon[: \forall e \in E : e(s, t')\}$$

is called the (t, ε)-*footprint* for an event set E on stream s. A (t, ε)-footprint may be *discrete*, *dense* or *durable*. Accordingly, we call an event *durable*, if it holds for all time points in some interval, which means that its footprint is identical to the set [t: t + ε[.

An event e is called *flickering at time* t in stream s, if one of the following formulas is valid:

(a) $\forall\, \varepsilon \in \mathbb{R}_+, \varepsilon > 0 : \exists\, t', t'' \in]t, t + \varepsilon[: e(s, t') \wedge \neg e(s, t'')$
(b) $\forall\, \varepsilon \in \mathbb{R}_+, \varepsilon > 0 : \exists\, t', t'' \in]t\text{-}\varepsilon, t[\cap \mathbb{R}_+ : e(s, t') \wedge \neg e(s, t'')$

In case (a) event e is called *flickering after* t, in case (b) *flickering before* t.

We say that an event is not Zeno, if it is never flickering. We say that for a stream a an event e is

- *Switched on* at time t, if $e(s, t) \wedge \forall \varepsilon \in \mathbb{R}_+, \varepsilon > 0 : \exists\, t' \in [t - \varepsilon, t[: \neg e(s, t')$
- *Switched off* at time t, if $e(s, t) \wedge \forall \varepsilon \in \mathbb{R}_+, \varepsilon > 0 : \exists\, t' \in [t, t + \varepsilon[: \neg e(s, t')$

and e is not flickering in s at time t.

Given event e, by ¬e we denote the complement event of e.

We call a set of events E ε-discrete for a stream s if for a real number $\varepsilon \in \mathbb{R}_+$ all (t, ε)-footprints for s and E contain at most one element. Then there is at most one event from the event set E in every time interval of length ε. In this case we can associate a discrete stream of events with the continuous stream s.

We are interested in associating a timed discrete stream of events with each hybrid stream to capture the event logics of histories. Given some time granularity $\delta \in \mathbb{R}_+$ we relate a discrete stream $r \in (E^*)^\infty$ with each timed stream $s \in PTS$ by

- Defining a set E of events
- Mapping the hybrid stream s to a discrete stream r = dis(s, E) by a function

$$dis : PTS \times E \to (E^*)^\infty$$

The set E is called the set of *logical observations*. To define the set of events E we assume a set of given events E_0. Since the set E_0 may not have discrete footprints and may contain durable events for streams $s \in PTS$ we replace durable events e by two events e_α and e_ω, where durable event e is starting at time t, characterizing the beginning and the end of the phase in which the event is durable by choosing and specifying:

$$e_\alpha(s,t) =_{\text{def}} \exists \varepsilon \in \mathbb{R}_+ \backslash \{0\} : \forall t' \in \mathbb{R}_+ \cap [t-\varepsilon : t[\ : \neg e(s,t')$$
$$\wedge \forall t' \in \mathbb{R}_+ \cap]t : t+\varepsilon[\ : e(s,t')$$
$$e_\omega(s,t) =_{\text{def}} \exists \varepsilon \in \mathbb{R}_+ \backslash \{0\} : \forall t' \in \mathbb{R}_+ \cap [t-\varepsilon : t[\ : e(s,t')$$
$$\wedge \forall t' \in \mathbb{R}_+ \cap]t : t+\varepsilon[\ : \neg e(s,t')$$

This gives us a set of events E' derived from E by replacing all durable events by the shown two events. This approach does work only, however, if the stream s avoids flickering and Zeno's paradox.

To avoid Zeno's paradox and generally flickering events we use a finite time granularity $\delta > 0$ and define the set of events E from E' as follows. We replace each event $e \in E'$ by an event e_E that is specified as follows. We make that sure we debounce events such that every event in E' can occur only once in each interval of length δ. To do this we first assume a linear strict order $<_{\text{prio}}$ defining priorities on the set of E' specifying the importance of events. Based on these priorities we define the event e_{prio} for each event e

$$e_{\text{prio}}(s,t) = (e(s,t) \wedge \neg \exists d \in E',\ t' \in \mathbb{R}_+ \cap \left[t - \frac{1}{2}\delta, t + \frac{1}{2}\delta\right] : d(s,t') \wedge e <_{\text{prio}} d)$$

i.e., e_{prio} (s, t) is an event with highest priority in the interval $\left[t - \frac{1}{2}\delta, t + \frac{1}{2}\delta\right]$. Now we define the set of events

$$E = \{e_{\text{prio}} : e \in E'\}$$

Given the set of events E we define a function

$$\text{dis}_E : PTS \to (E^*)^\infty$$

as follows:

$$\text{dis}_E(s)(i) = \langle\rangle \Leftarrow \neg \exists e \in E,\ t \in [i\delta : (i+1)\delta[: e(s,t)$$
$$\text{dis}_E(s)(i) = \langle e \rangle \Leftarrow \exists e \in E,\ t \in [i\delta : (i+1)\delta[: e(s,t)$$

Note that this definition is consistent since due to our construction every (t, δ)-footprint carries at most one event.

The time distance δ determines the time granularity of the stream $\text{dis}_E(s)$. If a finer or coarser time granularity is needed for $\text{dis}_E(s)$ the time granularity can be

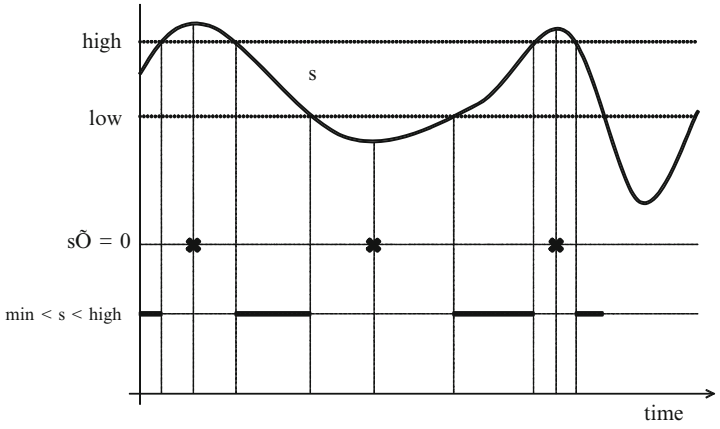


Fig. 1.3 Continuous stream s and discrete and durable events

changed according to [1]. Then after deriving $dis_E(s)$ we may coarsen $dis_E(s)$ which may lead to histories with more than one message in one time interval.

In principle, we may use the same construct to deal with durable events. Then durable events e are replaced by a sequence of discrete events e_E that are repeated every δ-time step as long as event e lasts. This is called *sampling* and provides an alternative to the approach treating durable events by introducing the event e_α and the event e_ω indicating the end of the durable event.

Fact 1.1. Associating discrete streams with hybrid streams

Let E′ be an arbitrary set of events and E be defined as above, then for every stream s ∈ PTS the set of events from E defines a discrete stream $dis_E(s) \in (E^*)^\infty$ of events in E.

Note that this form of associating discrete streams with continuous ones is essentially different from the techniques of discretisation used in numerical analysis or in control theory, where continuous functions are approximated by discrete step functions, where the distances between the discrete time points are chosen fine enough such that the functions are approximated precisely enough.

Figure 1.3 shows a continuous stream s and some examples of discrete and durable events.

Given event sets for all channels in set C, we get this way a function

$$\mathrm{Dis} : \vec{\vec{C}} \to \vec{C}$$

that maps histories of hybrid streams onto histories of discrete streams of events.

1.3.2 Assertions Specifying Hybrid Systems

To formulate properties about hybrid state machines with input and output we use interface assertions that refer to streams communicated via the input and output channels of the state machine. An interface assertion is a formula in predicate logic that contains the input and output channels of the state machines as logical identifiers for timed streams. The validity of such assertions for state machines is described in the following.

We work with templates to specify systems very much along the lines of [2] as demonstrated in the following example:

Example 1.1. Specification of hybrid systems

As a simple example we specify a hybrid system called Amplifier with a permanent and a discrete input stream and a permanent output stream.

System Amplifier (δ: Real: $\delta > 0$)

in v: Prm Real, c: Dsc Real
out r: Prm Real
$\forall t \in \mathbb{R}_+$:
$r(t + \delta) = lt(c, t)^* v(t)$
$0 \leq t < \delta \Rightarrow r(t) = 0$
where
$\forall t \in \mathbb{R}_+$:
$lt(c, t) = c(\max\{s" \in dom(c) : s" \leq t - \delta\}) \Leftarrow \{s" \in dom(c) : s" \leq t - \delta\} \neq \emptyset$
$lt(c, t) = 0 \Leftarrow \{s" \in dom(c) : s" \leq t - \delta\} = \emptyset$

This example shows an amplifier that amplifies the permanent input on channel v by the last actual value received on the discrete channel c and sends it as output on channel r with a time delay δ.

As the example shows, we use a mixture of plain higher order predicate logic and functional calculus. This leads to a specific logic that may be supported by interactive theorem provers (see [3]). Another possibility is domain specific logical calculi. Duration calculus (DC), for instance, is an interval logic for real-time systems (see [4]).

1.4 Composition

So far we have introduced a mathematical model for systems. Systems can be composed to larger systems by composition.

1.4.1 Composing Systems

In this chapter we study the composition of systems. We introduce the composition operator for composing two systems. Systems are composed by parallel composition with feedback following the approach of [2].

1.4.1.1 Composition of Systems in Terms of Their Interface Behaviour

The definition of composition of systems given by their interface behaviour reads as follows:

Definition 1.5. Composition of systems

Given two interfaces $F_1 \in IF[I_1 \blacktriangleright O_1]$ and $F_2 \in IF[I_2 \blacktriangleright O_2]$, with type consistent channels and where $O_1 \cap O_2 = \emptyset$, we define a composition for the feedback channels $C_1 = O_1 \cap I_2$ and $C_2 = O_2 \cap I_1$ by the expression

$$F1 \otimes F2$$

The system $F_1 \otimes F_2 \in IF[I \blacktriangleright O]$ is defined as follows (let $C = I_1 \cup O_1 \cup I_2 \cup O_2$, where $I = (I_1 \backslash C_2) \cup (I_2 \backslash C_1)$ and $O = (O_1 \backslash C_1) \cup (O_2 \backslash C_2)$):

$$\forall x \in \vec{I} : (F1 \otimes F2)(x) = \{y \in \vec{O} : \exists z \in \vec{C} : y = z|O$$
$$\wedge x = z|I \wedge z|O_1 \in F1(z|I_1) \wedge z|O_2 \in F_2(z|I_2)\}$$

The channels in set $C_1 \cup C_2$ are called *internal* for the composed system $F_1 \otimes F_2$. □

The idea of the composition of systems as defined above is graphically illustrated in Fig. 1.4.

In a composed system $F_1 \otimes F_2$, the channels in the channel sets C_1 and C_2 are used for internal communication.

Given specifying assertions S_1 and S_2 for the systems F_1 and F_2, the specifying assertion for $F_1 \otimes F_2$ is given by the assertion $\exists\, C_1, C_2 \colon S_1 \wedge S_2$, where internal channels C_1 and C_2 are hidden by the existential quantifier.

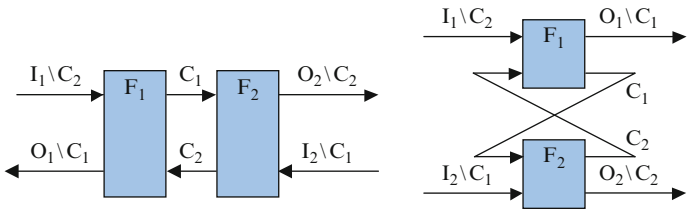


Fig. 1.4 Composition $F_1 \otimes F_2$ (in two layouts)

Parallel composition of systems with disjoint sets of input channels and disjoint sets of output channels is commutative and associative. The proof is straightforward.

The set of systems together with the introduced composition operators form an algebra. The composition of systems (strongly causal stream processing functions) yields systems.

Composition is a partial function on the set of all systems and the set of all services. It is only defined if the syntactic interfaces fit together. Syntactic interfaces fit together if there are no contradictions or conflicts in their channel names and types.

1.4.1.2 Composition of Hybrid State Machines

Consider Moore machines $\mathrm{M_k} = (\Delta_\mathrm{k}, \Lambda_\mathrm{k})$ with $\mathrm{k} = 1, 2$:

$$\Delta_\mathrm{k} : \Sigma_\mathrm{k} \times \widehat{\mathrm{I}}_k \to \wp(\Sigma_k \times \widehat{\mathrm{O}}_k)$$

We define the composed state machine (let O_1 and O_2 be disjoint; $I = I_1 \cup I_2$, $O = O_1 \cup O_2$)

$$\Delta : \Sigma \times \widehat{\mathrm{I}} \to \wp(\Sigma \times \widehat{\mathrm{O}})$$

as follows: the composed state is given by the set

$$\Sigma = \Sigma_1 \times \Sigma_2$$

For $\mathrm{x} \in \mathrm{I}$ and $(\sigma_1, \sigma_2) \in \Sigma$ we define the state transition function:

$$\Delta((\sigma_1, \sigma_2), \mathrm{x}) = \{((\sigma_1{}', \sigma_2{}'), \mathrm{z}|\mathrm{O}) : \mathrm{x} = \mathrm{z}|\mathrm{I} \wedge \forall \mathrm{k} : (\sigma_\mathrm{k}{}', \mathrm{z}|\mathrm{O_k}) = \Delta_\mathrm{k}(\sigma_\mathrm{k}, \mathrm{z}|\mathrm{I_k})\}$$

This definition is based essentially on the fact that we consider Moore machines. Note that for Mealy machines it is not guaranteed that appropriate histories z, as used in the definition above, exist.

We write

$$\Delta = \Delta_1 || \Delta_2$$

$$\mathrm{M} = \mathrm{M_1} || \mathrm{M_2} = (\Delta_1 || \Delta_2, \Lambda_1 \times \Lambda_2)$$

This way we get a composition for state machines that is compatible with the composition of behaviours.

1.5 Refinement and Abstraction of Time

Going from continuous to discrete streams is an abstraction step. The function dis maps a continuous and also a hybrid stream onto a discrete stream. The function Dis extends this mapping to histories. In the following we study how to go from discrete to continuous and also to hybrid behaviour in a refinement step based on these functions.

Refinement of properties is a simple and basic notion. Given a behaviour

$$F : \vec{\vec{I}} \to \wp(\vec{\vec{O}})$$

a behaviour

$$F' : \vec{\vec{I}} \to \wp(\vec{\vec{O}})$$

is called a *property refinement* if for all histories $x \in \vec{\vec{I}}$ we have

$$F'(x) \subseteq F(x)$$

Then every output history of F' is an output history of F.

Interaction refinement is the refinement notion for modelling development steps between levels of abstraction. Interaction refinement allows us to change for a component:

- The number and names of its input and output channels
- The types of the messages on its channels determining the granularity of the messages.

An interaction refinement is described by a pair of two functions

$$A : \vec{\vec{C}}' \to \wp(\vec{C}) \quad R : \vec{C} \to \wp(\vec{\vec{C}}')$$

that relate the interaction on an abstract level (in our case discrete system models) with the corresponding interaction on the more concrete level (in our case continuous or hybrid system models). This pair specifies a development step leading from one level of abstraction to the next as illustrated by Fig. 1.5. Given an abstract

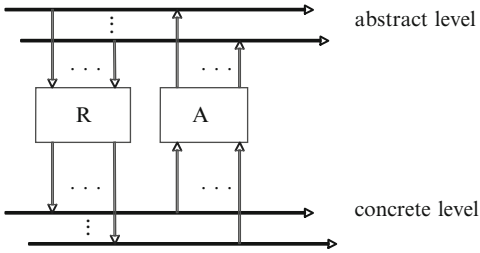


Fig. 1.5 Communication history refinement

history $x \in \vec{C}$ each history $y \in R(x)$ denotes a concrete history representing abstract history x. Calculating a representation for a given abstract history and then its abstraction yields the original abstract history again. Using pipelining composition, defined by

$$(R \circ A)(x) = \{z \in A(y) : y \in R(x)\}$$

this is expressed by the requirement:

$$R \circ A = Id$$

where Id denotes the identity relation. A is called the *abstraction* and R is called the *representation*. R and A are called a *refinement pair*.

Interaction refinement allows us to refine components, given appropriate refinement pairs for their input and output channels. The idea of an interaction refinement is visualized in Fig. 1.6. Note that the components (boxes) A_I and A_O are no longer definitional in the sense of specifications, but rather methodological, since they relate two levels of abstraction. Nevertheless, we may specify them also by the specification techniques introduced so far.

Given refinement pairs

$$A_I : \vec{\vec{I}}_2 \to \wp(\vec{I}_1) \qquad R_I : \vec{I}_1 \to \wp(\vec{\vec{I}}_2)$$
$$A_O : \vec{\vec{O}}_2 \to \wp(\vec{O}_1) \quad R_O : \vec{O}_1 \to \wp(\vec{\vec{O}}_2)$$

for the input and output channels we are able to relate abstract to concrete channels for the input and for the output. We call the I/O-behaviour

$$F' : \vec{\vec{I}}_2 \to \wp(\vec{\vec{O}}_2)$$

an *interaction refinement* of the I/O-behaviour

$$F : \vec{I}_1 \to \wp(\vec{O}_1)$$

if the following proposition holds:

$$F' \circ A_O \subseteq A_I \circ F \quad \textit{simulation}$$

This formula essentially expresses that the system $F' \circ A_O$ is a property refinement of the system $A_I \circ F$. Thus, for every "concrete" hybrid input history $x' \in \vec{\vec{I}}_2$ every concrete hybrid output history $y' \in F'(x')$ can be also obtained by translating the concrete hybrid history x' via abstraction A_I onto an abstract discrete input history $x \in A_I(x')$ with an abstract discrete output history $y \in A_O(y')$ for which $y \in F(x)$ holds.

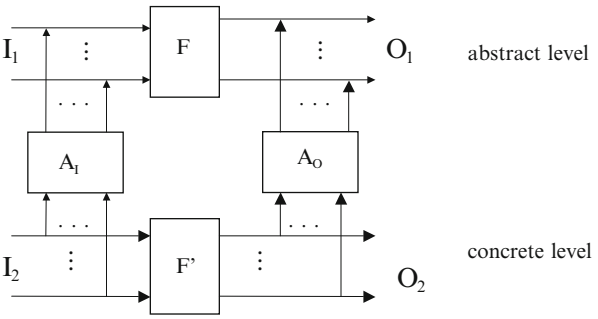


Fig. 1.6 Interaction refinement

This construction is here applied to the case of discrete and continuous system models by using the function Dis that maps continuous and hybrid histories to discrete ones. Hence, a discrete behaviour F is called an abstraction of the continuous or hybrid behaviour F' if system $F' \circ Dis_O$ is a property refinement of system $Dis_I \circ F$. Thus, for every continuous or hybrid input history $x' \in \overset{\Rightarrow}{I_2}$ every discrete output $y \in F(Dis_I(x'))$ can also be obtained from the continuous or hybrid output $y' \in F'(x')$ by translating the continuous or hybrid history x' via Dis_I onto a discrete input history $x \in Dis_I(x')$ such that there exists a discrete output history $y \in F(x)$ with $y \in Dis_O(y')$.

1.6 Concluding Remarks

The motivation for this work is twofold

1. Extending the approach of FOCUS (see [5]) to compositional hybrid models of systems in terms of interfaces, architectures and state machines as a generalisation of discrete to continuous streams of communication, supporting also hybrid models.
2. Defining refinement and abstraction relations between discrete and continuous models.

This approach aims at a general modelling technique for cyber-physical systems that can be combined with existing approaches to modelling hybrid systems such as hybrid automata, differential dynamic logic for hybrid systems, control theory, and duration calculus.

What we are particular aiming at is a formal basis for the translation of informal requirements in natural language into formalized logical requirements, that capture their logical contents and then further onto the specification of systems by differential equations as typical for control theory. A simple example would be a requirement from adaptive cruise control stating for instance "*As soon as a distance between the car and the object in front of it is less than the required safety distance*

relative to the current speed the car is slowed down." Such a requirement, which, by the way, is safety critical, then can be translated into a discrete event formalization. By defining appropriate concepts of discrete events for the continuous input and output streams for the adaptive cruise control, we can even prove its correctness for a hybrid system.

Acknowledgements It is a pleasure to thank David Trachtenherz and Radu Grosu for stimulating discussions and helpful remarks on draft versions of the manuscript.

References

1. Broy M (2008) Relating time and causality in interactive distributed systems. Marktoberdorf Summer School
2. Broy M, Stølen K (2001) Specification and development of interactive systems: Focus on streams, interfaces, and refinement. Springer, Berlin
3. Platzer A (2008) Differential dynamic logic for hybrid systems. J Automated Reasoning 41(2), 143–189
4. Chaochen Z, Hoare CAR, Ravn PA (1991) A calculus of durations. Inform Process Lett 40(5), 269–276
5. Broy M (2006) A theory of system interaction: Components, interfaces, and services. In: Goldin D, Smolka S, Wegner P (eds) The new paradigm. Springer, Berlin, pp 41–96
6. Henzinger TA (1996) The theory of hybrid automata. In: Proceedings of the 11th annual symposium on logic in computer science (LICS), IEEE Computer Society Press, pp 278–292. An extended version appeared in Verification of digital and hybrid systems (Inan MK, Kurshan RP, eds), NATO ASI series F: Computer and systems sciences, vol 170, Springer, Berlin, 2000, pp 265–292
7. Henzinger TA, Manna Z, Pnueli A (1993) Towards refining temporal specifications into hybrid systems. In: Hybrid systems I, Lecture notes in computer science 736, Springer, Berlin, pp 60–76
8. Alur R, Henzinger TA, Lafferriere G, Pappas GJ (2000) Discrete abstractions of hybrid systems. Proc IEEE 88:971–984
9. Lee EA (2009) Computing needs time. Commun ACM 52(5), 70–79
10. Sifakis J, Tripakis S, Yovine S (2003) Building models of real-time systems from application software. Proc IEEE (Special issue on modelling and design of embedded) 91(1), 100–111
11. Grosu R, Stauner Th., Broy M (1998) A modular visual model for hybrid systems. FTRTFT, 75–91

Chapter 2
Temporal Uncertainties in Cyber-Physical Systems

Hermann Kopetz

2.1 Introduction

A Cyber-Physical System (CPS) consists of two interacting sub-systems, the physical-subsystem(the *P-system*) and a (distributed) computer sub-system, (the *C-system*) (Fig. 2.1). The C-system, consisting of interface nodes and computation nodes that are interconnected by a real-time communication system, is a *real-time system*, i.e. the correctness of its results depends on the delivery of the intended values at the intended points in time. The behavior of the P-system is governed by the respective laws of the physical world, while the behavior of the C-system depends on programs executed on digital processors. We assume that the C-system observes the events of the P-system (the *P-events*), builds a model of the P-system, and acts on the P-system.

There is a fundamental difference in the *model of time* among the two system: While the model of time in the P-system is *dense* and time changes in infinitesimal steps, the model of time in the C-system is *discrete* and time changes abruptly in discrete steps. Since most C-systems are distributed and it is impossible to perfectly synchronize clocks in a distributed computer system, the finite synchronization error of distributed clocks constrains the granularity of a *reasonable* discrete global time base in the C-system [1].

The difference in the model of time of the P-system compared to the C-system and the jitter of the communication system within the C-system lead to significant phenomena concerning simultaneity, causality and determinism, the investigation of which is the topic of this paper.

We will show that, due the different models of time in the P-system and the C-system, it is impossible to recover the true sequence of events that happen close to

H. Kopetz (✉)
TU Wien, Austria
e-mail: hk@vmars.tuwien.ac.at

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_2,

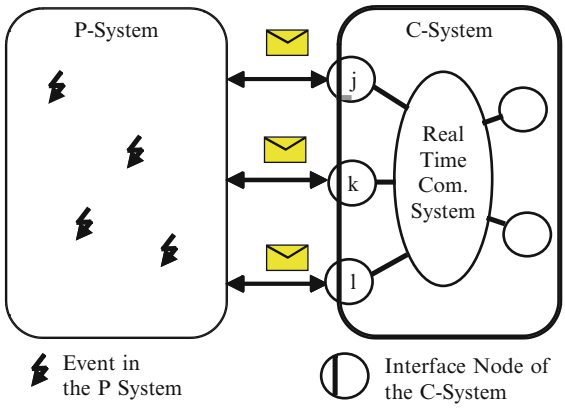


Fig. 2.1 Example of a cyber-physical system

each other in the *P-system* in the *model of the P-system* that is part of the *C-System*. It may even happen that the temporal order of the events as they are perceived by the *C-system* is in conflict with the true temporal order of *P-events*.

The topics analyzed in this paper are substantially different – although related to – the problems of sampling a (continuous) controlled P-object by a discrete C-system. In this case the duration of a sampling period will be orders of magnitude larger than the precision of the clock synchronization. Consider the example of a *drive-by-wire automotive system*: the sampling period may be 1 ms, while the precision of the clocks may be 1 μs. A jitter of the samplings instants of the order 10^{-3} of the sampling period of a distributed control system will be considered a *second order effect* that – in most cases – can be neglected.

The rest of this paper is structured as follows. Section 2.2 elaborates on the basic concepts that are used in this paper, such as *node*, *time* and *message*. Section 2.3 introduces the concept of the *observation uncertainty* that arises because of the synchronization error and the digitalization error of a discrete global time in a distributed C-system. Section 2.4 is devoted to the temporal inconsistency that is caused by the jitter of the communication system within the C-system. Section 2.5 discusses the consequences of the temporal uncertainties on causality and determinism. Based on the gained insights, Sect. 2.6 proposes some guidelines for the design of cyber-physical systems. The paper terminates with a short note on timeless systems and a conclusion in Sect. 2.7.

2.2 Basic Concepts

In this Section we elaborate on those concepts that must be clearly defined before any analysis of distributed real-time systems can be performed.

2.2.1 Node

A *node* is a hardware/software unit of the C-system that is aware of the progression of time. A node accepts input messages, provides a useful service, maintains internal state, and produces after some *elapsed time* output messages containing the results. A node that provides an interface to the *P-system* is called an *interface node*.

We call a property *consistent* if all nodes of the C-system agree on the value of this property. For example, if two interface nodes of the C-system observe and timestamp the occurrence of a single P-event then the timestamps are *consistent* if they have the same value. If the two timestamps for the same P-event are different in two interface nodes of the C-system, the timestamps are *inconsistent*.

If properties in the C-system are inconsistent then the reasoning about the behavior of the C-system is more complex, i.e., requires more mental effort [2]. Since we should avoid anything that increases the complexity of the C-system, the consistency of properties of the C-system is of paramount importance.

2.2.2 Time

When we use the word *time* we mean physical time as defined in the international standard of time TIA [3]. (In this paper, the words *time* and *real-time* are used synonymously, since they refer to the same concept.) A cut of the timeline is called an *instant*. The interval between two instants is called a *duration*. An occurrence at an instant is called an *event*. We call an event that occurs in the P-system a *P-even* and an event that occurs in the C-system a *C-event*.

We assume that every node j of a distributed computer system has its own local digital clock c_j. We generate a time-stamp of an *event e* observed by *node j* by assigning the value of the state of clock c_j immediately after the observation to the event, generating its timestamp $ts_j(e)$. We further assume that there exists an *omniscient observer* with a perfect clock z that is impeccably synchronized with TIA and has such a small (an infinitesimal small) granularity that second order effects that are related to the granularity can be neglected.

Timestamps generated by different nodes can only be related to each other if the respective clocks are synchronized to form a global notion of time. We call the maximum difference of respective ticks of any two clocks of our ensemble of synchronized clocks, as measured by the *reference clock z*, the *precision* π of the global time. We call the global time *reasonable*, if the *granularity g* of the global time (i.e., the *duration* between two adjacent ticks of the global time) is *longer* than the *precision* π of the ensemble [1].

2.2.3 Message

A *message* is an atomic data structure that is formed in the C-system for the purpose of *transmitting data* and *control signals* from a sending node at a given instant to one or more receiving nodes that receive the message at a later instant. The message concept does not make any assumption about (abstracts form) the physics of the specific transport mechanism (e.g., wire-bound or wireless transmission) or about the meaning of the bit-vector contained in the data field of the message. However, the time it takes to transport a message from the sending node to the receiving node is part of the control aspect of the message concept. A message that is intended to arrive at its receiver(s) within a given time-interval is called a *real-time message*; otherwise it is a *non-real-time message*. A real-time message is *correct* if it contains the *intended* data in its data field and is sent and received at the *intended* time.

The message concept can be generalized to cover the interactions between interface nodes of the C-system and the P-system. The flow of information within the P-system can be modeled by messages within the P-system, which we call *hidden messages* [1].

2.3 Observation Uncertainty

Even if the global time is *reasonable*, we cannot avoid that the observations of a single P-event, performed by two different nodes of the C-system, differ by one tick.

Let us explain this phenomenon by the following Fig. 2.2, where we denote the progress of the reference clock by the small marks on the uppermost line. Every tenth tick of the reference clock is a *global-time tick*, denoted by the respective number. Since the *precision* π of the clocks in the *nodes j, k* and *l* in the following figures is *nine* ticks, the *reasonableness condition* is met.

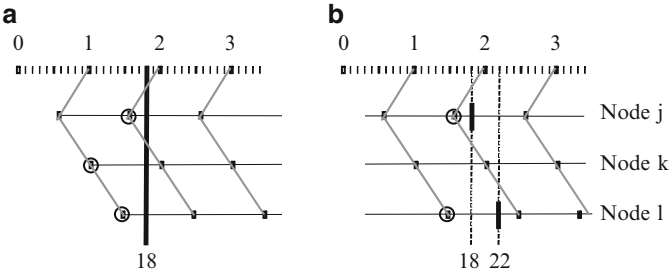


Fig. 2.2 The effect of the digitalization error and synchronization error on the time-stamping of events

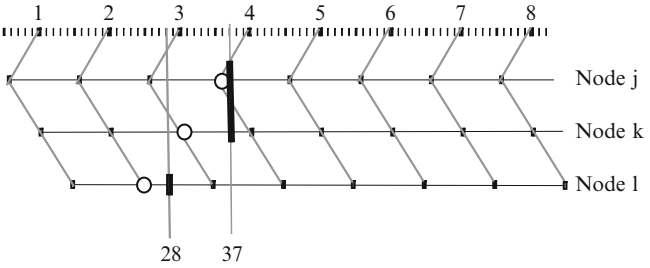


Fig. 2.3 Multiple observations of an event

In Fig. 2.2a the *P-event 18* (i.e., an event that happens on the z-time-scale – top of Fig. 2.2 – at instant18) is time-stamped by *node j* with the global time 2, by *node k* with the global time 1 and by *node l* with the global time 1.

Given a *reasonable* global time, the real temporal order of two P-events can only be recovered consistently in the C-system, if the *time-stamps* of these two events differ by at least *two global time ticks*. This will always be the case if the events occur at least *three global time ticks apart*. This minimum distance of events that is required in order to be able to recover the temporal order of P-events by the C-system is called the *observation inaccuracy*. This *observation inaccuracy* depends on the precision of the clock synchronization in the C-system.

Let us now look at the scenario of Fig. 2.2b, where the global *event 18* is observed by *node j*, generating the timestamp $ts_j(18)$ of 2 and *event 22* is observed by *node l*, generating the timestamp $ts_l(22)$ of 1. Although *event* 18 happened in the P-system before event 22, its *time-stamp* $ts_j(18)$ in the C-system is larger than the *timestamp* $ts_l(22)$ of the later event 22. This implies that the *perceived temporal order* of temporally close *P-events* in the C-system can differ from the *real temporal order* of the events in the P-system. *We cannot do better* in the case the events in the P system are quite close together.

Let us now analyze a scenario (Fig. 2.3) where a singe P-event is observed by more than one C-node. *P-event 28* happens at the z-time of 28 and is observed by *node l* that assigns $ts_l(28) = 2$ to this P-event. *Node l* sends a message with this observation to *node k* and *node j*. *Node k* and *node j* observe the same *P-event 37*. *Node k* timestamps this event $ts_k(37) = 3$, while *node j* timestamps this event $ts_j(37) = 4$. *Node k* concludes that *P-event 28* occurred before *P-event 37*, since the difference of the timestamps is 2. *Node j* cannot order these two events, since the difference of the timestamps is only *1*. It follows that the two nodes j and k have an *inconsistent view* about the temporal order of these two P-events.

In order to eliminate any inconsistency about the temporal order of P-events within the C-system it is proposed that all C-nodes that observe a single P-event must arrive at a *common view* as to which time-granule this P-event is assigned to. In the example of Fig. 2.4, *node j* and *node k* must agree on whether they both assign the *time-stamp 3* or the *timestamp 4* to event 37 (by the execution of an *agreement protocol*). This is an arbitrary decision that is needed to maintain consistency within

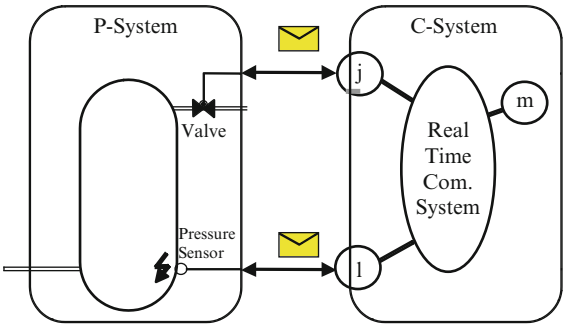


Fig. 2.4 Simple monitoring PCS

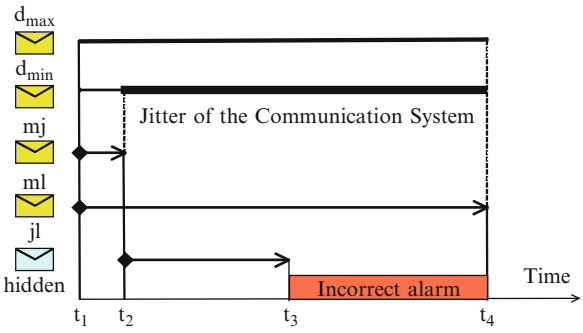


Fig. 2.5 The effect of Jitter on the permanence of messages

the C-system. Agreement is also needed in the value domain if the P-value of a variable is an analog quantity, since the least significant bit in any analog-to-digital conversion is an *arbitrary* value in the C-value of that variable.

2.4 Permanence of Messages

The temporal uncertainty caused by the jitter of the message communication system in the C-system can be the source of an inconsistency in the C-system which can result in severe system failures. The example in Figs. 2.4 and 2.5 are used to analyze this inconsistency and introduce the concept of *permanence of messages*.

Let us analyze the simple monitoring system depicted in Fig. 2.4. A pressurized vessel contains a *pressure release valve* controlled by *node j* and a pressure sensor monitored by *node l*. The pressure in the vessel can drop for two reasons, either the operator, connected to the man-machine interface at *node m*, sends a message to *node j* to open the pressure release valve (*expected pressure drop*) or a defect in the P-system, such as a rupture of a pipe, causes a pressure drop (*unexpected pressure drop*). In case of an unexpected pressure drop *node l* should raise an alarm and initiate an emergency shut-down procedure of the plant.

The real-time message communication system in the C-system of Fig. 2.4 is characterized by a maximum message delay of d_{max} and a minimum message delay of d_{min} resulting in a *jitter* of d_{max} *minus* d_{min} as shown in Fig. 2.5.

Let us assume that the operator sends at *instant* t_1 a message from *node m* to *node j* and *node l*, commanding *node j* to open the pressure release valve and informing *node l* that the following pressure drop is an *expected pressure drop* caused by the opening of the pressure release valve by *node j*. Given the *message mj* arrives at *node j* at *instant* t_2 (after the minimum delay) *node j* will open the pressure release valve at *instant* t_2 and the consequent pressure drop will be relayed to *node l* by a *hidden message* in the P-system, arriving at *node l* at *instant* t_3. Since at this instant, *node l* has not received the related *message ml* (which is still in transit since the related *message ml* takes the maximum message delay of d_{max}), *node l* will assume at *instant* t_3 an unexpected pressure drop and initiate the emergency shutdown procedure. Later, at *instant* t_4, *node l* will realize that is made a mistake – but it is too late to undo the emergency shutdown. The jitter of the communication system that has messed up the causality chain is at the source of the failure.

This example is a good illustration of the fundamental conflict between *speed-of-action* and *consistency*. Let us call a message that is supposed to cause an action in the P-system an *action message*. A node should delay an action on an *action message* until after the action message has become *permanent*. *An action message becomes* ***permanent****, as soon as no other message that is related to this action message is in transit any more*. The set of *related* messages must be derived from an application specific analysis. In the previous example, the information message from *node m* to *node l* is *related* to the action message from *node m* to *node j*.

How long must *node j* wait, after the arrival of an action message, until the action message becomes *permanent*? Let us assume that all related messages are sent at the same instant. In the extreme case, if the message delay of the *action message mj* is d_{min}, then the required *wait duration* is d_{max} minus d_{min}, i.e., the *jitter* of the communication system (see Fig. 2.5). In the other extreme case, if the message delay is d_{max}, then the wait duration is *zero*. However, *node j* can only decide on the basis of its local information.

Given that *node j* has no knowledge about the send-time of an action message, it must assume the worst case and always delay the action for the full *jitter duration* – even in the case that the action message delay was already d_{max} (and the node could have acted immediately after the receipt of the action message). Viewed from our omniscient outside observer, the duration between the send time of the action message and the instant when the receiver is sure that the action message is permanent is $(d_{max} + d_{max} - d_{min})$, *i.e.* d_{min}, *plus two times the jitter of the communication system*.

In case the node has knowledge about the send-time of the message (e.g., a send-timestamp is part of the action message) and a reasonable global time of granularity g is available, then the wait duration is always until *send-time* $+ d_{max} + 2g$, i.e. d_{min}, *plus the jitter of the communication system plus* $2g$. In the example of Fig. 2.5, *node j* should have waited until t_4 before executing the *open-valve command*. From the above analysis we conclude that it is the *jitter of the communication protocol*

that determines to a significant degree the responsiveness of a distributed real-time system.

The knowledge about the *send time-stamp* of an action message helps to reduce the duration a receiver has to wait until he is sure that the message has become *permanent* [1]. It follows that a C-system where all nodes have access to a global time has a better response time characteristic than a C-system without a global time base.

2.5 Causality and Determinism

It is the objective of many cyber-physical system to control the P-system by the C-system in order to achieve a desired effect in the physical world. For this purpose, a model of the behavior of the P-system is constructed within the C-system. This model takes the C-timestamps of significant P-events as input. Because of the *observation uncertainty,* the C-model of the P- system is *not fully faithful*. For example, *it is impossible to observe the simultaneity of P-events*, no matter how fine a granularity of the global time has been chosen. We cannot build a fully faithful C-model of a P-system, because the models of time that are *innate* in each one of these systems are different. What we can do is limit the effect of the *observation uncertainty* by choosing an appropriate granularity of the global time base.

2.5.1 Causal Analysis of Events

In many scenarios (e.g., alarm analysis in a widely distributed electric power distribution system) the identification of the causal sequence of events in the P-system by the C-system is of importance. Consider, for example, the following remark in the report about the *US-Canada power blackout* of August 14, 2003: '*A valuable lesson from the August 14 blackout is the importance of having time-synchronized system data recorders. The Task Force's investigators labored over thousands of data items to determine the sequence of events, much like putting together small pieces of a very large puzzle. That process would have been significantly faster and easier if there had been wider use of synchronized data recording devices. [4], p. 162*'.

Since there is a close relationship *between causal* order and *temporal order*, the establishment of the correct temporal order of events is the starting point for any causal analysis. Necessary (but not sufficient) for a causal dependence to exist is that the *cause-event* has happened prior to the *effect-event*. If an event has happened before or simultaneously to the effect-event, it cannot be the cause.

If the temporal distance between two events in the P-system is less than the *observation uncertainty* then the correct (physical) temporal order of these events cannot be established in the C-model of the P-system, due to the digitalization and

synchronization error that are inherent parts of the timing in the C-system. *The observation granularity thus limits the causal analysis of the observed P-events.*

The temporal uncertainty introduced by the jitter of the communication system in the C-system can also mess up the causal analysis of events, as shown by the example of Fig. 2.4. It is therefore necessary to delay an action until the *action message* has become *permanent*.

2.5.2 State

In many C-models of a P-system the concept of *state* is introduced to capture the effects of past behavior on future behavior. We follow the definition of *Mesarovic* [5], p. 45.

The state enables the determination of a future output solely on the basis of the future input and the state the system is in. In other word, the state enables a "decoupling" of the past from the present and future. The state embodies all past history of a system. Knowing the state "supplants" knowledge of the past.

... Apparently, for this role to be meaningful, the notion of past and future must be relevant for the system considered...

The borderline between the past and the future is the event *now*. A consistent distributed C-system-wide notion of state can only be introduced if all nodes have an identical view about the event-set that exists before *now*.

What happens if a *P-event* and the event *now* have the same time-stamp, i.e. a *P-event* and the event *now* happen *simultaneously* in the C-model? In this case, a rule is introduced that says that the effect of the P-event on the state has to be processed first such that the state at *now* includes the effect of the simultaneous P-event. Note that the notion of state is an artificial construct in the C-model of the P-system to capture the effects of the past on the future in a *well-defined data structure*. State in its digital form does not exist in the P system.

2.5.3 Determinism

We call a system *deterministic*, if its time evolution can be predicted. For the following reason *determinism* is a desirable property of C-system behavior:

1. *Timeliness*: Many C-system must carry out timely responses. The notion of *determinism* subsumes predictable timing
2. *Complexity reduction:* Logical reasoning, i.e. *modus ponens* is based on a deterministic relationship between *cause* and *effect*. The human mind is ill-equipped to reason along probabilistic dependencies.
3. *Testing*: The testability of a C-system is improved, if the system will produce the same outputs given it has been offered identical inputs [6].

4. *Active Redundancy*: The implementation of active redundancy requires a deterministic behavior of the replicated nodes.

For our purpose, a more specific definition of determinism is derived from [7]: A model behaves deterministically if and only if, given a full set of initial conditions (the initial state) at the discrete time t_o, and a sequence of future timed inputs, then the outputs and the system state at selected future instants are entailed.

Let us now investigate the following scenario: A *P-system* that is assumed to be deterministic is interfaced with a *C-system* that is deterministic (the C-system must obey the rule about the permanence of messages and contain no other mechanism that causes indeterminism) to form a cyber-physical system (CPS). Is the behavior of this CPS *deterministic*? As long as any two events in the P-system are further apart than the *observation uncertainty*, the CPS behavior will be *deterministic*. If events in the P-system are closer together than the *observation uncertainty,* then the randomness of observations can cause a non- deterministic behavior of the CPS. Two runs, starting at the same initial state of the P-system can evolve differently in the C-system, due to the possible different event orderings perceived by the C-system at the P-system/C-system interface.

Does this non-determinism of the CPS matter? *Not really*, since the reasons for introducing determinism, as stated in the beginning of this section, relate to the behavior of the C system and not to the behavior of the full CPS. To take the argument further, with every physical system there is a finite *assumption coverage* of less than one that the *assumed deterministic model* of the physical system (that is the subject of our investigation) is a valid abstraction of the behavior of the *real physical system* (Consider, e.g., quantum mechanic effects). Even if the physical system behaves non-deterministically – and in many instances even the models of physical system that we investigate will be non-deterministic, there is a strong case for making the C-system behave deterministically. We definitely need the C-system determinism if we intend to mask a failure of a C-system channel at the logical level [8] by active redundancy, such as triple-modular redundancy (TMR).

2.6 Guidelines for System Design

The insights gained from the analysis in the previous sections lead to the following design guidelines

2.6.1 Observation Uncertainty

Whenever we design a timing system for a Cyber-Physical System, we must start with an analysis of the *tolerable observation uncertainty* in the given application context. The tolerable observations uncertainty depends on the dynamics of the

P-system. Depending on the intended task that the CPS is to perform, we must decide on the minimal temporal distance of P-events, the temporal order of which must be faithfully represented in the C-model of the P-system. P-events that are closer together than this minimal distance will still be consistently temporally ordered in the C-system, but this C-system temporal order might not be faithful. The *tolerable observation uncertainty* determines the required precision of the global time, i.e., the granularity of the global time must be better than one third of the tolerable observation uncertainty.

2.6.2 Precision

The theory of clock-synchronization is well developed [1], such that the design of the clock synchronization system is a straightforward engineering task. This design depends on whether the P-system and the C-system are local or geographically widely distributed.

In a local system, such as a car or an airplane, clocks can be synchronized via a local area network. There are local communication protocols, such as TTP [9] or Flexray [10] where clock synchronization is an integrated service of the protocol. With these protocols a precision of better than a micro-second can be achieved. If this protocol-inherent precision is not sufficient, a special synchronization protocol, such as the standardized IEEE 1588 clock synchronization protocol [11]can be deployed.

In a widely distributed system, such as the before-mentioned electric grid control (see Sect. 2.5.1), external clock synchronization via the GPS service is the preferred alternative, since the jitter of messages in a best-effort wide area network can be significant. The GPS-time is precise to better than 1 μs. The combination of external and internal clock synchronization is the alternative of choice if local islands have to be globally synchronized [12].

2.6.3 Permanence

Given a global time, every *action message* should contain a send-timestamp such that a receiver can determine at what instant the message becomes *permanent* without having to wait for the full *jitter* of the communication system. When selecting a real-time communication protocol for the C-system, a protocol should be selected that exhibits a small jitter. In real- time systems, a small jitter is more important than a small average transmission delay. This is in contrast to non-real-time protocols, where a small average transmission delay is aimed for.

2.6.4 Simultaneity

In order to maintain a consistent view of the events in the C-system, multiple C-System observations of the same P-event must be consolidated by the execution of *agreement protocols* in order to assign a single time-stamp to every single P-event.

The consistent handling of simultaneous C-events by all nodes of the C-system warrants special attention. As mentioned before, simultaneity is a property of C-events that is not necessarily a true property in the P-system. Nevertheless, all C-nodes must have an *identical view* of simultaneity and must resolve simultaneity in the same way. Extending the timestamp by assigning a unique natural number to each node as a node-ID and appending this node-ID to the time-stamp of the P-event can achieve this [13]. Simultaneous C-events can now be consistently temporally ordered on the basis of the extended time-stamps.

2.6.5 Reintegration State

The robustness of a node can be considerably improved, if a node can recover after the occurrence of a transient fault within a short recovery interval. For this purpose it is necessary to specify during system design *reintegration instants* where the relevant state of the node is contained in a declared data structure – we call it the *reintegration state* of the node. In order to minimize the size of the *reintegration state*, the application has to be carefully analyzed to find an instant, where the effects of the past on the future are small and where all past activity that is relevant for the future behavior is captured in the declared data structure. For example, in a cyclic control system, the interval before the start of a new control cycle is a good candidate for a reintergration instant. This reintegration instant should be visited periodically with a period that is smaller than the tolerable recovery interval. The reintegration state should be stored periodically in stable storage, such that after a fault a node can be reset and be restarted with the most recently saved reintegration state.

2.6.6 A Note on Timeless Systems

What is the *observation uncertainty* of a distributed C-model in a CPS that does not contain a global time, i.e. is *timeless* (or *asynchronous*)? The establishment of the *observation uncertainty* in such a system requires a cumbersome analysis of all involved *communication jitters* and *middleware jitters*. Since in an asynchronous distributed system with a shared communication channel a *critical instant*, i.e., an instant where all senders send simultaneously a message to the same receiver, cannot be avoided, the jitter of the communication system in such a timeless system will be significant. As long as a rigorous analysis of this *observation uncertainty* is not

available, the faithfulness of the respective C-model remains an *open question*. Of what utility is a *model-based analysis* of the causality of events in the P-system, if the faithfulness of the corresponding C-model has not been established?

2.7 Conclusion

The different models of time in the physical subsystem, the P-system, and the computation subsystem, the C-system, of a cyber-physical system cause a temporal observation uncertainty at the P-system/C-system interface. This observation uncertainty can be reduced, but not eliminated, by providing a more precise global time-base. As a consequence of this observation uncertainty the temporal order of events that occur close together in the P-system may not be properly reflected in the C-system model of the P-system, limiting the causal analysis of P-event. Furthermore, the jitter of the communication system within the C-system can cause inconsistencies in the C-system state. These inconsistencies can be avoided, if an action is delayed until the message is permanent. This delay can be reduced, if a global time of appropriate precision is available. At the end of this paper, some practical guidelines for the design of Cyber-Physical Systems have been given.

Acknowledgements This work was supported in part the EU Project GENESYS under project number FP 7/213322. Many discussions within the project and the research group on distributed real-time systems at the TU Vienna are warmly acknowledged.

References

1. Kopetz H (2011) Real-time systems–design principles for distributed embedded applications. Second Edition. Springer Verlag, 2011
2. Kopetz H (2008) The complexity challenge in embedded system design. In ISORC 2008, IEEE Press
3. Wikipedia (2008) International atomic time
4. Final Report on the August 14, 2003 Blackout in the United States and Canada, 2004
5. Mesarovic MD, Takahara Y (1989) Abstract systems theory. Lecture notes in control and information science, vol 116. Springer, Berlin
6. Schütz W (1993) The testability of distributed real-time systems, vol ISBN 0–7923–9386–4, Kluwer, Boston, MA, p 160
7. Hoefer C (2004) Causality and determinism: Tension, or outright conflict. Revista de Filosofia 29(2):99–225
8. Avizienis A (1982) The four-universe information system model for the study of fault tolerance. In: Proceedings of the 12th FTCS symposium, IEEE Press, Los Angeles, 1982
9. Kopetz H, Gruensteidl G (1993) TTP – A time- triggered protocol for fault-tolerant real-time systems. In: Proceedings of the 23rd IEEE international symposium on fault-tolerant computing (FTCS-23), IEEE Press, Toulouse, France, 1993
10. Berwanger J et al (2001) FlexRay – the communication system for advanced automotive control systems. In: SAE World Congress, SAE Press, Detroit, 2001, paper 2001001–0676

11. IEEE (2002) 1588 standard for a precision clock synchronization protocol for network measurement and control systems
12. Kopetz H, Ademaj A, Hanslik A (2004) Integration of internal and external clock synchronization by the combination of clock state and clock rate correction in fault tolerant distributed systems. In: RTSS 04, IEEE Press, Lissabon, 2004
13. Lamport L (1978) Time, clocks, and the ordering of events. Comm. ACM b(7):558–565

Chapter 3
Large-Scale Linear Computations with Dedicated Real-Time Architectures

Patrick Dewilde and Klaus Diepold

3.1 Introduction

Understanding the connection between algorithms and solvers for large scale systems on the one hand, and appropriate architectures that execute them efficiently on the other is key to the effective design of modern signal processing applications. This trend has started with the emergence of digital media, large scale signal processing for image coding and analysis, digital mobile telephony and digital processing in medical imaging. In many cases dedicated, hardware or software dominated methods on a single processor have been used, only in recent times more generic methods based on general purpose array architectures, either dedicated to media processing or for general computing have become realizable. Massive use of parallelism becomes attractive and should, in the future, allow us to tackle large scale problems in a streamlined fashion. It is no exaggeration to state that this trend was started a long time ago when Georg Färber proposed parallel processing schemes for signal processing and control using coprocessors on standard busses and proceeded to prove his ideas in practice, thereby creating a company that set a standard in the field [28] – he was clearly many decades ahead of the field!

In this contribution to this volume to honor the attainment of the status of "Professor Emeritus" by Georg Färber, we review a number of what we consider very attractive cases where the connection between algorithm and executing architecture proves to be very effective. Luckily, these cases handle some of the most important algorithms in numerical linear algebra. The resulting combination

K. Diepold (✉)
Technische Universität München, Department of Electrical Engineering and Information Technology
e-mail: kldi@tum.de

P. Dewilde
Technische Universität München, Institute for Advanced Study
e-mail: p.dewilde@me.com

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_3,

algorithms-architectures brings the fields of signal processing and linear algebra together, a dream that has been somewhat elusive over the years, because programming paradigms and hardware design methods were not well adapted to each other. Work has to be done, both at the side of the choice of algorithm and at the side of the architectural design. Although we cannot go into the details of neither the most sophisticated algorithms nor the details of the design methodology, we can easily illustrate the principles. Essential is the combination of the choice of algorithm with the architectural consequences, the designer handles algorithm and architecture at the same time, to achieve a result in which both sides are optimally adapted on each other.

An important issue in signal processing is numerical stability and robustness of the algorithms used. The system designer should not deteriorate the numerical properties of his problem by introducing computational steps that degrade the conditioning and hence the quality of the computed results. The conditioning of a problem is defined as a measure of the sensitivity of the result to variations of the input data, or more precisely, how much errors in the input data are propagated to the output by the implemented mathematical function.

Jacobi's QR factorization is one of the numerical algorithms, which exhibits very favorable numerical properties, which it combines with great opportunities for parallelization [14]. The QR factorization sits at the core of solutions for many important technical problems: the solution of linear equations, channel estimation and signal identification in telecommunication, Kalman filtering (in the square root version) and H-infinity control. It also represents a core function in an iterative loop for computing eigenvalue and singular value decompositions (abbreviated as EVD and SVD, respectively). Although the QR factorization is used a lot, often hidden in embedded software, it is not as well known as it deserves. To honor Jacobi, we start the paper with a description of the algorithm, and an account of its main applications.

The matrices associated with real technical problems exhibit often special structural properties. These properties are exploited by an alternative class of algorithms for improved performance or for reduced computational complexity. Foremost for large system solvers are the iterative algorithms used to handle e.g. sparse matrices. We give an account of some of the major methods and the resulting architectures. Structure also plays a role in direct solvers. A very important type of structure is called "semi-separable" or "quasi-separable" (the terminology has not stabilized.) Here, both direct and iterative methods play a role, we give a brief account.

To connect algorithms to architectures, we make systematic use of "data flow graphs", sometimes called dependence or precedence graphs. These are actually generalizations of the signal flow graphs classically used in the signal processing literature. They have been adopted in commercial design packages [24] and give the designer a convenient way to control the parallelization process without having to resort to complicated and often wieldy design tools.

3.2 Algorithms and Architectures

3.2.1 Technical Applications of Numerical Linear Algebra

Solving linear systems of equations or computing a least squares solution for an overdetermined system of equations belong to the most common computational tasks in science and engineering. Engineered products and services are relying on the capability of real-time systems to solve such problems fast, accurate and in a robust way. Examples for this statement are Kalman-filtering [12], rake-receivers in mobile communications [38], adaptive channel equalizers, adaptive beamforming [34], in stereo vision systems [16], to name just a few. Another set of applications using linear systems of equations originates from CAD tools and circuit simulators [5], structural analysis using finite element methods or partial differential equations [37].

Researchers in the domain of Numerical Linear Algebra have worked since the dawn of the modern computer on devising effective numerical methods for solving systems of equations of ever increasing dimension. Therefore, we suffer no shortage of published results and practical implementations of system solvers, which are readily available and widely used in terms of highly optimized software packages such as the LINPACK, LaPACK or IMSL libraries, as well as in software packages like Matlab, Mathematica, Octave, or Scilab.

3.2.2 Dense Matrices and Direct Algorithms

Direct solution schemes use factorizations of the coefficient matrix which allow to map the original problem on a problem involving a triangular matrix. Examples are the LU factorization $T = L \cdot U$ for a square coefficient matrix T and the lower and upper triangular factors L and U, respectively. For a symmetric positive definite coefficient matrix (prime indicates transpose) $T = T', T > 0$ we can compute the Cholesky factorization $T = R' \cdot R$, where R is upper triangular. Note that both factorizations require pivoting schemes to achieve numerical stability [14]. Pivoting is an effective method to improve numerics, but it destroys the regular data flow because of additional control structures and branching.

Factorization algorithms, which are based on orthogonal elementary operations, such as the QR decomposition, satisfy the need for numerically reliable computations without resorting to pivoting [14]. A solution strategy which computes the QR-decomposition of the coefficient matrix using elementary Jacobi (Givens) rotations has the added benefits to be amenable for parallelization on highly local and regular architectures.

Solving dense systems of linear equations takes $\mathcal{O}(n^3)$ operations [14], where n denotes the size of the coefficient matrix. This computational effort may be overwhelming in case of large n. In many signal processing applications the

matrices involved may have moderate values for n, for which the computational burden may be challenging for real-time application, but they comprise "structure", that is, the matrices have only $\mathcal{O}(n)$ parameters. The structure in the matrix allows for solution algorithms, which require only $\mathcal{O}(n^2)$ operations. Typical examples for such structured matrices are diagonal matrices or Toeplitz or Hankel matrices [22].

3.2.3 Square-Root and Array Algorithms

Real-time computer systems gain additional advantages in terms of numerical robustness and reliability if the implemented algorithms operate directly on the data of the coefficient matrix instead of setting up normal equations. Computing the normal equation squares the condition number of the problem; this leads to a dramatic loss in precision for the result, if the coefficient matrix is badly conditioned and hence more sensitive to rounding errors and other imperfections of finite word length computations. Algorithms which work directly on the data are often referred to as "square-root algorithms", because they avoid "squaring" the data when determining the covariance matrices that come with approaches based on solving the normal equation. The combination of "square-root" approaches and orthogonal elementary transformations leads to a family of so-called "array" algorithms, which exploit an elementary identity known as Schur-complements and which are highly suitable for being mapped onto parallel computer architectures [23, 31].

3.2.4 Algorithms and Architectures

Applications running on real-time computer systems require that computations are completed with a pre-determined deadline, that the results are computed in a reliable way having an accuracy that is robust against perturbations and noise. Furthermore, favorable algorithms shall provide a high level of locality and parallelism [27]. For large scale real time architectures it would be very attractive to dispose of a generic high level algorithm that at the same time solves major problems, and at the other is amenable to real time realization, utilizing the available resources to a maximum. The requirements one can put on such an algorithm are:

- *Parametrizable*: the algorithm should be able to handle problems of any size, even though the available resources are limited.
- *Numerically accurate*: the algorithm should be numerically backward stable with errors close to the conditioning of the problem.
- *Parallelizable*: there should be a natural way to partition the algorithm in chunks that operate in parallel and can be mapped to the underlaying architecture.
- *Localizable*: both data transport and memory usage should be highly local, no massive storage needed during the algorithm nor massive data transport or data manipulation.

- *Generic*: the algorithm should be able to handle many if not most large scale computations – much like multiplication and addition, which can handle most numerical problems as well.
- *Incremental*: when additional data becomes available, the algorithm gracefully adapts.

3.3 The QR Algorithm as a Generic Method

The algorithm for computing the QR factorization was originally proposed by Jacobi and is an excellent candidate algorithm to satisfy the requirements list given in the previous section. We start with a description of the algorithm, followed by a methodology to derive attractive, real-time architectures for it. We stop at the architectural level, but enough detail of the strategy will transpire to allow for attractive, concrete realizations.

3.3.1 The QR Algorithm

The most attractive way to present QR is on a very common example. Suppose T is a rectangular, tall matrix of dimension $m \times n$, y a given vector of dimension m and suppose we are interested in finding a vector u of dimension n such that Tu is as close as possible to y (we assume real arithmetic throughout. With slight modifications complex arithmetic or even finite field calculations are possible as well but beyond our present scope.) Numerical analysts call such a problem "solving an overdetermined system". It occurs in a situation where u is a set of unknown parameters, row i of T consist of noisy data, which, when combined linearly with u produce the measured result y_i. The situation occurs very often in measurement setups, where repeated experiments are done to reveal the unknown parameters, or in telecommunication where transmitted signals have to be estimated from the received signals (we omit the details). The most common measure of accuracy is "least squares", for a vector u with components u_i we write

$$\|u\|_2 = \sqrt{\sum_{i=1}^{n} |u_i|^2} \tag{3.1}$$

There may be more than one solution to the minimization of $\|Tu - y\|_2$, often one is interested in the least squares, so one tries to solve

$$u_{\min} = \text{argmin}_w \|(w = \text{argmin}_u \|Tu - y\|_2)\|_2 \tag{3.2}$$

the actual minimum being the estimation error. The strategy is to perform a QR decomposition of T. The matrix Q has to be an orthogonal matrix (a generalized rotation), which keeps the norm of the vectors to which it is applied, while R should be an upper triangular matrix. Let Q' be the transpose of Q, then orthogonality means $Q'Q = QQ' = I$, Q' is actually the reverse rotation. Let us just assume that Q and R can be found (see further), and let us apply Q' to y, to obtain $\eta = Q'y$, then we have $QRu = y$ and hence $Ru = Q'y = \eta$. R has the same dimensions as T, meaning that it is a tall matrix. It is also upper triangular, meaning that it has the form

$$R = \begin{bmatrix} R_u \\ 0 \end{bmatrix} \tag{3.3}$$

in which R_u is now $n \times n$ square and upper triangular. We find that the solution u must minimize

$$\| \left(\begin{bmatrix} R_u u \\ 0 \end{bmatrix} - \eta \right) \|_2. \tag{3.4}$$

If we partition $\eta = \begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix}$ with η_1 of dimension n, we see that the minimal solutions must satisfy the square system $R_u u = \eta_1$ and that η_2 certainly contributes wholly to the error, there is nothing we can do about it. If R_u is non-singular, i.e. if the original system has a full row basis, then the solution will be unique, i.e. $u = R_u^{-1}\eta_1$ and the error will be $\|\eta_2\|_2$. If that is not the case, further analysis will be necessary, but the dimension of the problem is reduced to n, the number of parameters, from m, the number of measurements (usually much larger.) It may appear that the QR factorization step is not sufficient, it has to be followed by a "back substitution" to solve $R_u u = \eta_1$, but this difficulty can be circumvented (see further). Here we want to concentrate just on the QR step and its possible architectures.

3.3.2 The Basic Step: Jacobi Rotations

The elementary Jacobi matrix is a rotation over an angle θ in the 2D plane (for convenience we define Q'):

$$Q' = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \tag{3.5}$$

Let's abbreviate to $Q' = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$ and apply the rotation to two row vectors:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_1 & a_2 \cdots a_n \\ b_1 & b_2 \cdots b_n \end{bmatrix} = \begin{bmatrix} \sqrt{a_1^2 + b_1^2} & ca_2 + sb_2 \cdots \\ 0 & -sa_2 + cb_2 \cdots \end{bmatrix} \tag{3.6}$$

which is achieved by choosing $c = \frac{a_1}{\sqrt{a_1^2+b_1^2}}$, $s = \frac{b_1}{\sqrt{a_1^2+b_1^2}}$ (and hence automatically $\tan\theta = \frac{b_1}{a_1}$). In this way one can treat the entries of the original matrix T row by row and create all the zeros below the main diagonal. With a 4×3 matrix this works as shown below. The only thing one must do is embed the 2×2 rotation matrices in the 4×4 schema, so that unaffected rows remain unchanged. We label the rotation matrices with the indices of the rows they affect – we indicate affected entries after each step with a $\star$:

$$\begin{bmatrix}\cdot&\cdot&\cdot\\ \cdot&\cdot&\cdot\\ \cdot&\cdot&\cdot\\ \cdot&\cdot&\cdot\end{bmatrix} \xrightarrow{Q'_{1,2}} \begin{bmatrix}\star&\star&\star\\ 0&\star&\star\\ \cdot&\cdot&\cdot\\ \cdot&\cdot&\cdot\end{bmatrix} \xrightarrow{Q'_{1,3}} \begin{bmatrix}\star&\star&\star\\ 0&\cdot&\cdot\\ 0&\star&\star\\ \cdot&\cdot&\cdot\end{bmatrix} \xrightarrow{Q'_{1,4}} \begin{bmatrix}\star&\star&\star\\ 0&\cdot&\cdot\\ 0&\cdot&\cdot\\ 0&\star&\star\end{bmatrix} \xrightarrow{Q'_{2,3}} \begin{bmatrix}\cdot&\cdot&\cdot\\ 0&\star&\star\\ 0&0&\star\\ 0&\cdot&\cdot\end{bmatrix} \xrightarrow{Q'_{2,4}} \begin{bmatrix}\cdot&\cdot&\cdot\\ 0&\star&\star\\ 0&0&\cdot\\ 0&0&\star\end{bmatrix} \tag{3.7}$$

and the final step is a $Q'_{3,4}$ which annihilates the $4,3$ entry. In each of these subsequent steps, the first operation determines the rotation matrix and then applies it to all the entries in the respective rows, skipping the already computed zero entries (which remain zero). The overall rotation matrix is then $Q = Q_{1,2}Q_{1,3}Q_{1,4}Q_{2,3}Q_{2,4}Q_{3,4}$. In most cases it need not be put in memory (and if so there are tricks.) It turns out that this algorithm leads to a very regular computational schema, now known as the "Gentleman-Kung array", which we discuss in the next section.

The result of a QR factorization need not be in strict triangular form, the actual more general form is called an i.e., echelon form. It may happen that in the course of the algorithm, when the processor moves from one column to the next, an actual sub-column of zeros is discovered. In that case no rotation is necessary and the processor can move to the next sub-column, which again might be zero etc... until a sub-column is reached with non zero elements. The result will then have the form shown in Fig. 3.1. The QR algorithm compresses the row data of the matrix in the North-East corner, leaving the norm of each relevant sub-column as leftmost non zero element. There is a dual version, called the LQ algorithm that compresses the columns in the South-West corner, and one can of course also construct versions for the other corners (but these will not bring much additional information). One can take care of the zero or kernel structure exemplified by the QR or LQ algorithms by testing on zero inputs when Jacobi rotations are applied, this can be arranged for automatically, we shall henceforth just assume that these provisions have been taken.

$$\begin{bmatrix}0&*&\bullet&\bullet&\bullet&\bullet&\bullet&\bullet\\ 0&0&0&0&*&\bullet&\bullet&\bullet\\ \vdots&0&0&0&0&0&*&\bullet\\ \vdots&\vdots&\vdots&&\cdots&&0&0\\ 0&0&0&0&0&0&0&0\end{bmatrix}$$

Fig. 3.1 Example of an echelon form. Elements indicated with a "$\star$" are strictly positive. The QR algorithm compresses the rows to the North-Eastern corner of the matrix and generates a basis for the rows of the matrix

3.3.3 The Gentleman-Kung Array

An important step for connecting the algorithmic information with potential architectures is the definition of a "dependence graph", sometimes called a "sequencing graph". The graph exhibits operations as nodes and data transport as dependencies between nodes. In the case of the QR algorithm just defined, we have two types of operations: Type 1 consists in the computation of sine and cosine of an angle given two values (say a and b) and the subsequent rotation of these two values to $\sqrt{a^2 + b^2}$ and 0, while the second operation just applies the rotation to subsequent data on the same rows. The array is upper triangular, for each position (i, j) in the upper part of the resulting matrix R_u there is one processor, the processors on the diagonal are of the first type, while the others are of the second type. Angle information is propagated along the rows, while the data is inputed, row by row, along the columns. It gives a precise rendition of the operation-data dependencies in the original algorithm. From the view of a compiler, the graph represents a "Single Assignment Code (SAC)" rendition of the original algorithm, there is no re-use neither of operations nor of memory elements. Actually, the local memory in this representation is reduced to the end result, the $r_{i,j}$ of the final matrix, one per processor, all other data is communicated either from the environment to the area, or from one processor to the next. In this algorithm, some data can actually be "broadcast", namely the information about the angles (c_i, s_i), along the rows, it might not be a bad idea to make special arrangements for this. The array shows what the architecture designer has to know about the algorithm. If only the type of the various data is specified and the operation in each node, he has enough information to design the architecture – we describe briefly what the architecture designer can do next in a further section, but before doing so we show how various standard problems can be solved with the array or interesting extensions of it (Fig. 3.2).

3.4 Architecture Design Strategies for Parallelization of the QR Algorithm

Once a high level algorithm can be represented as a regular array, in which nodes represent operations and arrows data transport, the issue arises how these operations and data can best be mapped to an architecture consisting of actual processors, memories, busses and control. In the following sections we shall see that (extensions) of the Gentleman-Kung array succeeds in solving major problems of linear algebra, estimation and control. These problems may have any dimension, they are parametrized by their size, often indicated as n. It seems logical that the most attractive type of concrete architecture to map to would be an array itself, consisting of processing nodes that contain provisions for the necessary

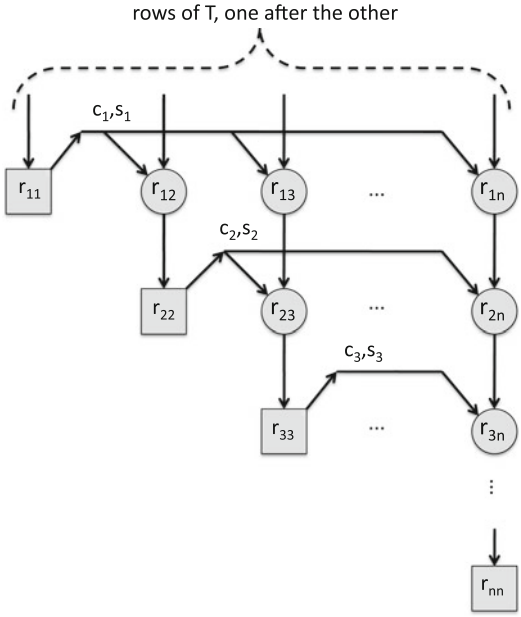


Fig. 3.2 The Gentleman-Kung or Jacobi array. The squares represent "vectorizing rotors", they compute the Jacobi angle information from the data and perform the first rotation, the circle are "rotors", they rotate a two-dimensional vector over the given angle

operations (in our case vectorization and rotation), local memory in each processor for intermediate results, and an infrastructure surrounding the array, whose task it is to provide the array with data and to collect results when they become available. Such a processor array will typically have a small dimension, e.g. 3×3 to mention a commercial size. So the issue becomes the mapping of an arbitrarily (parametrized) SAC array onto potential, architecturally viable sizes. When the computational array is regular, this assignment can be done in a regular fashion as well. As data transport becomes a paramount issue in performance, it becomes important to use local memory as much as possible. The architectural problem becomes in the first place a problem of efficient memory usage, or, to put it more simply, to perform the partitioning and mapping to resources of the original array in such a way that local memory is fully used before data are sent to background memory. We can offer two general partitioning strategies to achieve this feat, whose combination allows to first exhaust local memory and then map well chosen remainder to background. They have been called with various names in the literature, here we call them "LPGS" and "LSGP" – for "Local Parallel Global Sequential" and "Local Sequential Global Parallel" [20]. We suffice here to describe these two strategies, there are many more, more detailed ones, but these would go beyond the scope of the present paper.

In each of the two strategies, the original array will first be "tesselated", i.e. decomposed in tiles. Although not strictly necessary, we shall assume our tiles to be square, just to illustrate the methods. In a first approach, we have two choices: either we choose the tiles so big that the total array of tiles actually equals the processor array we envisage, and then we map each tile just to one processor, or

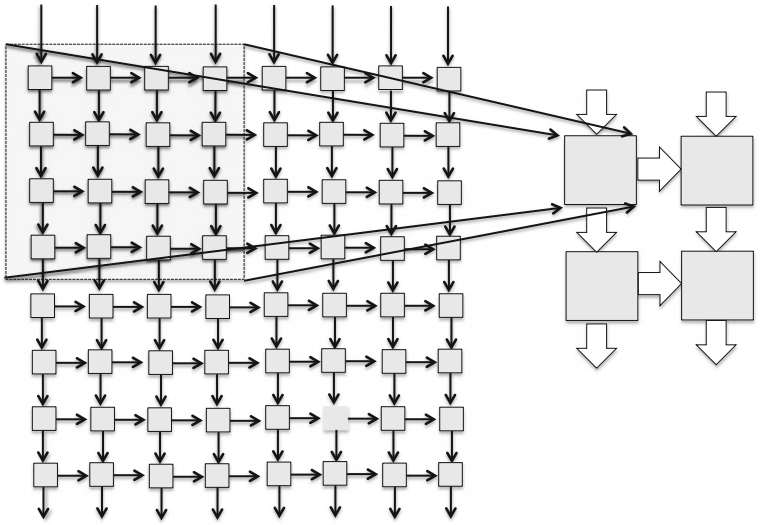


Fig. 3.3 The "Local Sequential, Global Parallel" strategy: The array is partitioned in subarrays, the partitions are mapped to the processor array

we take tiles so large that they are exactly isomorphic to the processor array. The first strategy is LSGP – the operations within one tile will be executed sequentially on one processor. As a consequence of this strategy, the intermediary data that is produced by one operation either has to be mapped to local memory (if the operation that shall use the data belongs to the same tile) or it has to be shoveled to background memory (if the operation that shall use the data is in another tile). Clearly, LSGP will use a lot of local memory and only be feasible if the size of the tiles are small enough so that all local intermediary results can also be stored locally. The opposite strategy is LPGS. Here the size of the tile would be chosen exactly equal to the size of the processor array, and the only local storage needed is what was already assigned to the individual nodes, all the rest goes to the background memory. The two strategies are illustrated in Figs. 3.3 and 3.4 respect.

What could then be an optimal strategy? The solution is almost obvious: choose tiles so big that LSGP is feasible on them, condensate the operations and then use LPGS on the result – LSGP by choice followed by LPGS by necessity.

3.5 Solving Problems with a QR Array

3.5.1 One Pass Equation Solver: The Faddeev/Faddeeva Array

To obtain a direct, one pass solution for the system of linear equations $Ty = u$ using a modification of the Gentleman-Kung array, one just has to perform QR

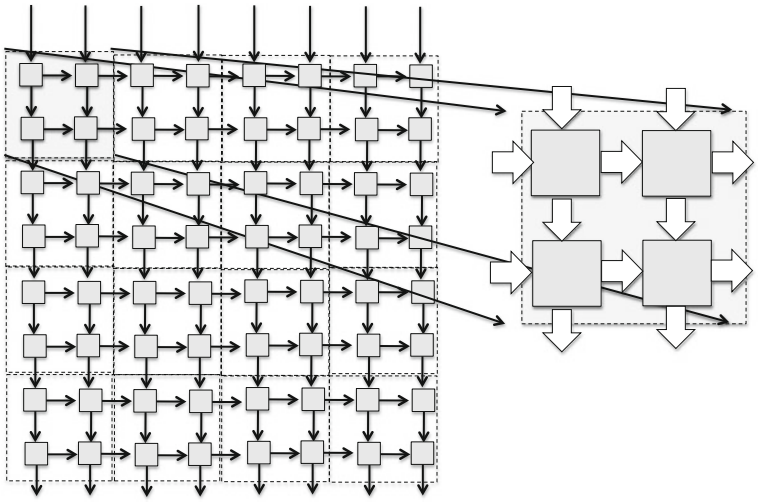


Fig. 3.4 The "Local Parallel, Global Sequential" strategy: Tiles are mapped directly on the processor array and executed sequentially

on the following, extended matrix (we use $\cdot'$ to indicate the transpose of a matrix, namely $T'_{i,j} := T_{j,i}$) [18]:

$$F := \begin{bmatrix} T' & I & 0 \\ -u' & 0 & 1 \end{bmatrix} \tag{3.8}$$

Instead of having a tall matrix as in the original Gentleman-Kung array, we now have a flat one, the only thing we must do is extend the algorithmic array with a number of rotors to the right hand side. The dimensions have also increased, it is now a processor array of dimensions $(n+1)\times(2n+1)$, with only $(n+1)$ vectorizing rotors on the main diagonal, but otherwise just a similar regular array as before, now rectangular. The Q matrix will now have dimensions $(n+1)\times(n+1)$, and to amplify this we partition it accordingly:

$$Q = \begin{bmatrix} Q_{11} & q_{12} \\ q_{21} & q_{22} \end{bmatrix} \tag{3.9}$$

In this representation, q_{12} is a tall vector of dimension n, q_{21} a flat vector of dimension n and q_{22} just a scalar quantity. One can immediately verify that the QR factorization of this matrix produces:

$$\begin{bmatrix} T' & I & 0 \\ -u' & 0 & 1 \end{bmatrix} = Q \begin{bmatrix} R & Q'_{11} & q'_{21} \\ 0 & y'q_{22} & q_{22} \end{bmatrix} \tag{3.10}$$

in which R is some matrix (which we do not use further), and the result appears as the pair $(y'q_{22}, q_{22})$. One interprets q_{22} as a normalizing factor, it is actually equal to $1/\sqrt{1+\|y\|^2}$ flowing out of the array.

3.5.2 Channel and Signal Estimation in Telecommunications

The typical equation governing a telecommunication situation has the form

$$x = h * s + n. \tag{3.11}$$

Herein x is the received signal, h represents the channel, s the signal to be transmitted and n the noise. All these can be viewed as "signals", in the most current situation they are functions of time. "$*$" is convolution, it represents the action of a channel on an input signal, convolution being typical for a linear, time-invariant (LTI) situation wherein the output is the sum of similar responses modulated by the signal and shifted according to their occurrence. The technical situation is often pretty complicated, but from a numerical point of view, relating original input sequences to the output, the overall model can be captured by two equivalent equations:

1. The *channel estimation model* $x = Sh + n$, in which

$$S = \begin{bmatrix} s_0 & & \\ \vdots & \ddots & \\ s_{K-1} & & s_0 \\ & \ddots & \vdots \\ & & s_{K-1} \end{bmatrix}, \quad h = \begin{bmatrix} h_0 \\ \vdots \\ h_{L-1} \end{bmatrix}, \tag{3.12}$$

2. The *signal estimation model* $x = Hs + n$, in which

$$H = \begin{bmatrix} h_0 & & \\ \vdots & \ddots & \\ h_{L-1} & & h_0 \\ & \ddots & \vdots \\ & & h_{L-1} \end{bmatrix}, \quad s = \begin{bmatrix} s_0 \\ \vdots \\ s_{K-1} \end{bmatrix}. \tag{3.13}$$

s is now a sequence of original symbols that are being transmitted (we assume them to be real numbers), h is the impulse response of the transmission medium and n is the overall "noise" being added at each reception (the noise is a combination of interferences and processing noise). The first situation is the "training" part, in which we assume that s is known while h has to be estimated, the system is adapting

to the transmission situation by using a learning signal. In the second situation knowledge about the channel is assumed and it is used to transmit information s that has to be estimated.

Classical estimation theory provides a number of techniques, called "Best Linear Unbiased Estimator – BLUE", "Minimum Variance Unbiased Estimator – MVU", "Maximum Likelihood Estimator – MLE" and even Bayesian estimators, such as the "Linear Minimum Mean Square Error Estimator – LMMSE". Most popular are BLUE and LMMSE, which we briefly pursue. In the case of the channel estimator, assume the noise to have covariance $\sigma^2 I$ and the signal matrix S to be sufficiently rich so as to have full row rank, then optimization theory shows that the BLUE and least square estimators are given by

$$\hat{h} = (S'S)^{-1}S'x. \tag{3.14}$$

$(S'S)^{-1}S'$ is the so called *Moore-Penrose inverse* of S. This is precisely the situation we have described in the section on QR-factorization. Indeed, when $S = QR$, then $(S'S)^{-1}S' = R^{-1}Q'$, we have called $Q'x = \eta_1$ and we find the result $\hat{h} = R^{-1}\eta_1$ – exactly the algorithm presented earlier. There is also a one-pass version, due to Jainandunsing and Deprettere [18].

The data situation is not much different. In the BLUE situation and with the white noise assumption in force, the signal estimator becomes

$$\hat{s} = (H'H)^{-1}H'x \tag{3.15}$$

and a QR factorization of H will produce the result.

If a so-called Bayesian estimator is desired, then known covariance information on the result has to be brought into play. The formulas get to be a little more complicated, for example

$$\hat{s} = (H'H + \sigma^2 C_s^{-1})H'x \tag{3.16}$$

for the data estimator (in which C_s is the assumed known covariance of s (this may purely be belief data!). Also in this case the QR factorization is the way to go, as $H'H = R'R$ and $H'x = R'\eta_1$ with η_1 defined as before. Typically, the R matrix will be much smaller than the H matrix, the former has the size of the data vector, while the size of the latter is determined by the number of experiments.

3.5.3 The Kalman Filter

The Kalman estimation filter attempts to estimate the actual state of an unknown discrete dynamical system, given noisy measurements of its output, for a general introduction see Kailath [21], see also Kailath, Sayed and Hassibi [23], here we give a brief account to connect up with the previous section on QR and the

following section on structured matrices. The traditional set up makes a number of assumptions, which we summarize.

3.5.3.1 Assumptions

1. We assume that we have a reasonably accurate model for the system whose state evolution we try to estimate. Let x_i be the evolving state at time point i – it is a vector of dimension δ_i. We further assume that the system is driven by an unknown noisy, zero mean, vectorial input u_i, whose second order statistical properties we know (as in the BLUE, we work only with zero mean processes and their variances). We assume the dynamical system to be linear and given by three matrices $\{A_i, B_i, C_i\}$, which describe respectively the maps from state x_i to next state x_{i+1}, from input u_i to next state x_{i+1} and from state x_i to output y_i. The latter is contaminated by zero mean, vectorial *measurement noise* v_i, whose second order statistics we know also. The model has the form given by

$$\begin{cases} x_{i+1} = A_i x_i + B_i u_i \\ y_i = C_i x_i + v_i \end{cases} \tag{3.17}$$

   A data flow diagram of the state evolution is shown in Fig. 3.5. The *transition matrix* of this filter is defined as the matrix $\begin{bmatrix} A_i & B_i \\ C_i & 0 \end{bmatrix}$, it defines the map from all inputs $\begin{bmatrix} x_i \\ u_i \end{bmatrix}$ to all outputs $\begin{bmatrix} x_{i+1} \\ y_i \end{bmatrix}$.
2. Concerning the statistical properties of the driving process u_i and the measurement noise v_i, we need only to define the second order statistics (the first order means is already assumed zero, and no further assumptions are made on the higher orders). We always work on the space of relevant, zero means stochastic variables, using "**E**" as the expectation operator. In the present summary, we assume that u_i and v_i are uncorrelated with each other and with any other u_k, v_k, $k \neq i$, and that their covariances are given respectively by $\mathbf{E} u_i u_i' = Q_i$ and $\mathbf{E} v_i v_i' = R_i$, both non singular, positive definite matrices (this assumes that

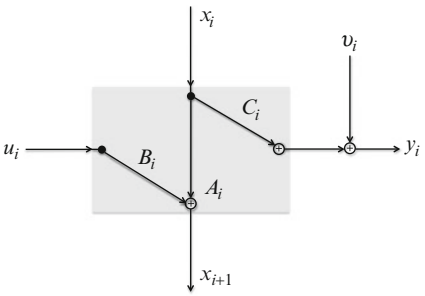


Fig. 3.5 The model filter

there is enough noise in the system), we assume again real matrices throughout, otherwise one must use Hermitian transposition. On a space of scalar stochastic variables with zero mean we define a (second order statistical) inner product as $(x, y) = \mathbf{E}(xy)$. This can be extended to vectors by using outer products such as

$$\mathbf{E}xy' = \mathbf{E}\left(\begin{bmatrix} u_1 \\ \vdots \\ u_m \end{bmatrix}\begin{bmatrix} y_1 \cdots y_n \end{bmatrix}\right) = \begin{bmatrix} \mathbf{E}u_1y_1 & \mathbf{E}u_1y_2 & \cdots & \mathbf{E}u_1y_n \\ \mathbf{E}u_2y_1 & \mathbf{E}u_2y_2 & \cdots & \mathbf{E}u_2y_n \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{E}u_my_1 & \mathbf{E}u_my_2 & \cdots & \mathbf{E}u_my_n \end{bmatrix}. \quad (3.18)$$

3. We also assume that the process whose state is to be estimated starts at the time point $i = 0$. The initial state x_0 has known covariance $\mathbf{E}x_0x_0^T = \Pi_0$.

3.5.3.2 The Recursive Solution

We start with summarizing the classical solution, based on the "innovations model" pioneered by Kailath e.a., see [21]. Let us assume that we have been able to predict x_i and attempt to predict x_{i+1}. The least squares predictor $\hat{x}_i$ is the one that minimizes the prediction error $e_{x,i} = x_i - \hat{x}_i$ in the covariance sense, assuming linear dependence on the data (the same assumptions will hold for the next predictor). The Wiener property asserts that $\hat{x}_i$ is a linear combination of the known data (in our case all the y_k for $k = 0 \cdots i-1$) and that the error, also called the state innovation, e_i is orthogonal on all the known data so far. These properties will of course be propagated to the next stage, given the (noise contaminated) new information y_i. It turns out that the only information needed from the past of the process is precisely the estimated state $\hat{x}_i$, the new estimate being given by

$$\hat{x}_{i+1} = A_i\hat{x}_i + K_{p,i}(y_i - C_i\hat{x}_i). \quad (3.19)$$

In this formula $K_{p,i}$ denotes the "Kalman gain", which has to be specified, and which is given by

$$K_{p,i} = K_iR_{e,i}^{-1},\ R_{e,i} = R_i + C_iP_iC_i',\ K_i = A_iP_iC_i'. \quad (3.20)$$

In these formulas, the covariances $P_i = \mathbf{E}e_{x,i}e_{x,i}^T$ and $R_{e,i}$ are used, in view of the formula for the latter, only P_i has to be updated to the next step, and is given by

$$P_{i+1} = A_iP_iA_i' + B_iQ_iB_i' - K_{p,i}R_{e,i}K_{p,i}' \quad (3.21)$$

The covariance P_{i+1} is supposed to be positive definite for all values of i, a constraint which may be violated at times because of numerical errors caused be the subtraction in the formula. In the next paragraph we shall introduce the square root version of the Kalman filter, which cannot create this type of numerically

caused problems. So far we have only given summaries the known results, we give a simple direct proof in the next paragraph. Starting values have to be determined since this solution is recursive, and they are given by

$$\hat{x}_0 = 0,\ P_0 = \Pi_0. \tag{3.22}$$

Proof. We give a recursive proof based on the parameters of the model at time point i. We assume recursively that the error $e_i = x_i - \hat{x}_i$ at that time point is orthogonal on the previously recorded data, and that the new estimate $\hat{x}_{i+1}$ is a linear combination of the data recorded up to that point. We first relax the orthogonality condition, and only ask e_{i+1} to be orthogonal on $\hat{x}_i$ (a linear combination of already recorded previous data) and y_i, the newly recorded data at time point i. We show that this estimator already produces an estimate that is orthogonal (of course in the second order statistical sense) on all the previously recorded data. From our model we know that $x_{i+1} = A_i x_i + B_i u_i$. We next ask that $\hat{x}_{i+1}$ be a linear combination of the known data $\hat{x}_i$ and y_i, i.e. there exist matrices X_i and Y_i, to be determined, such that

$$\hat{x}_{i+1} = X_i \hat{x}_i + Y_i y_i. \tag{3.23}$$

Requesting second order statistical orthogonality of e_{i+1} on $\hat{x}_i$ we obtain

$$\mathbf{E}(x_{i+1} - \hat{x}_{i+1})\hat{x}_i' = \mathbf{E}(A_i x_i + B_i u_i - X_i \hat{x}_i - Y_i y_i)\hat{x}_i' = 0. \tag{3.24}$$

We now observe that $\mathbf{E}u_i \hat{x}_i' = 0$ by assumption and that $\mathbf{E}\hat{x}_i \hat{x}_i' = \mathbf{E}x_i \hat{x}_i'$ because $\mathbf{E}e_i \hat{x}_i = 0$ through the recursive assertion. The previous equation then reduces to

$$(A_i - X_i - Y_i C_i)\mathbf{E}(x_i \hat{x}_i') = 0, \tag{3.25}$$

which shall certainly be satisfied when $X_i = A_i - Y_i C_i$.
Next we request orthogonality on the most recent data, i.e.

$$\mathbf{E}e_{i+1} y_i' = 0. \tag{3.26}$$

In fact, we can ask a little less, by using the notion of "innovation". The optimal predictor for y_i is simply $\hat{y}_i = C_i \hat{x}_i$, and its innovation, defined as $e_{y,i} = y_i - \hat{y}_i$, is $e_{y,i} = C_i e_i + \nu_i$. We now just require that e_{i+1} is orthogonal on $e_{y,i}$, as it is already orthogonal on $\hat{x}_i$ and $\mathbf{E}e_{i+1} y_i' = \mathbf{E}[e_{i+1}(x_i' C_i' + \nu_i' - \hat{x}_i' C_i')]$. We now obtain an expression for the innovation e_{i+1} in term of past innovations and the data of section i

$$e_{i+1} = A_i e_i + B_i u_i - (A_i - Y_i C_i)\hat{x}_i - Y_i y_i = A_i e_i + B_i u_i - Y_i e_{y,i}, \tag{3.27}$$

which we now require to be orthogonal on $e_{y,i}$. With $P_i = \mathbf{E}e_i e_i'$, we have $\mathbf{E}e_i e_{y,i}' = P_i C_i'$ and $\mathbf{E}e_{y,i} e_{y,i}' = C_i P_i C_i' + R_i$. The orthogonality condition becomes therefore

$$Y_i(R_i + C_i P_i C_i') = A_i P_i A_i'. \tag{3.28}$$

Hence the formulas given for the Kalman filter, after identifying $K_{p,i} = Y_i$ and $R_{e,i} = R_i + C_i P_i C_i'$ (actually the covariance of $e_{y,i}$.)

Concerning the propagation of the covariance of the innovation P_i, we rewrite the formula for e_{i+1} as (reverting back to the notation in the previous paragraph)

$$e_{i+1} + K_{p,i} e_{y,i} = A_i e_i + B_i u_i. \tag{3.29}$$

Remarking that the terms of the left hand side are orthogonal to each other, and those of the right hand side as well, we obtain the equality

$$P_{i+1} + K_{p,i} R_{e,i} K_{p,i}' = A_i P_i A_i' + B_i Q_i B_i', \tag{3.30}$$

which shows the propagation formula for the innovation covariance. □

Finally, when y_k is some data collected at a time point $k < i$, we see that $\mathbf{E}e_{i+1} y_k^T = \mathbf{E}[(A_i e_i + B_i u_i - K_{p,i}\hat{y}_i) y_k']$. The recursion hypothesis states that e_i is orthogonal to all past collected data, in particular to y_k. Hence we see that the expression is equal to zero, after working out the individual terms.

3.5.3.3 The Square Root (LQ) Algorithm

The square root algorithm solves the Kalman estimation problem efficiently and in a numerical stable way, avoiding the Riccati equation of the original formulation. It computes an *LQ factorization* on the known data to produce the unknown data. An LQ factorization is the dual of the QR factorization, rows are replaced by columns and the order of the matrices inverted, but otherwise it is exactly the same and is done on the same architecture. Not to overload the symbol "Q", already defined as a covariance, we call the orthogonal transformation matrix at step i, U_i, acting on a so called *pre-array* and producing a *post-array*

$$\begin{bmatrix} C_i P_i^{1/2} & R_i^{1/2} & 0 \\ A_i P_i^{1/2} & 0 & B_i Q_i^{1/2} \end{bmatrix} U_i = \begin{bmatrix} R_{e,i}^{1/2} & 0 & 0 \\ \bar{K}_{p,i} & P_{i+1}^{1/2} & 0 \end{bmatrix}. \tag{3.31}$$

The square root algorithm gets it name because it does not handle the covariance matrices P_i and $R_{e,i}$ directly, but their so called square roots, actually their Cholesky factors, where one writes, e.g. $P_i = P_i^{1/2} P_i^{\prime/2}$ assuming $P_i^{1/2}$ to be lower triangular, and then $P_i^{\prime/2}$ is its upper triangular transpose (this notational convention is in the benefit of reducing the number of symbols used, the exact mathematical square root is actually not used in this context). The matrix on the left hand side is known from the previous step, applying U_i reduces it to a lower triangular form and hence defines all the matrices on the right hand side. Because of the assumptions on the

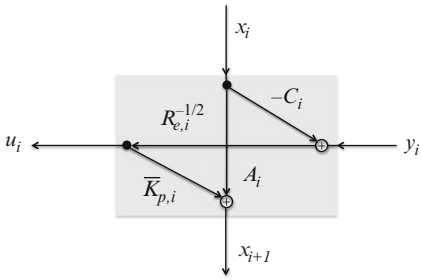


Fig. 3.6 The Kalman filter, alias innovations filter

non singularity of R_i, $R_{e,i}$ shall also be a square matrix, the non-singularity of P_{i+1} is not directly visible from the equation and is in fact a more delicate affair, the discussion of which we skip here.

The right hand side of the square root algorithm actually defines a new filter with transition matrix

$$\begin{bmatrix} A_i & \bar{K}_{p,i} \\ C_i & R_{e,i}^{1/2} \end{bmatrix} \tag{3.32}$$

One obtains the original formulas in the recursion just by *squaring* the square root equations (multiplying to the right with the respective transposes). In particular this yields $A_i P_i A_i' = \bar{K}_{p,i} R_{e,i}'$ and hence

$$\bar{K}_{p,i} = K_{p,i} R_{e,i}^{1/2} = K_i R_{e,i}^{-\prime/2} \tag{3.33}$$

(different versions of the *Kalman gain*). This form is called an *outer filter*, i.e. a filter that has a causal inverse. The inverse can be found by arrow reversal (see Fig. 3.6) and it can rightfully be called both the Kalman filter (as it produces $\hat{x}_{i+1}$) and the (normalized) innovations filter, as it produces $\epsilon_i = R_i^{-1/2}(y_i - C_i \hat{x}_i)$, the normalized innovation of y_i given the preceding data summarized in $\hat{x}_i$.

3.5.4 Solving Symmetric Positive Definite Systems with the Schur Algorithm

We consider the solution of a linear least-squares problem via the normal equation

$$T'Tu = T'y,$$

where the coefficient matrix $C = T'T$ is a symmetric positive definite matrix. We split up the symmetric matrix according to $C = U + D + U' \in \mathbb{R}^{m \times m}$, and define the intermediate matrices

$$V = \frac{1}{2}(D + 2U + \mathbf{1}_n), \quad W = \frac{1}{2}(D + 2U - \mathbf{1}_n)$$

such that C can be represented as

$$C = V'V - W'W = \begin{bmatrix} V \\ W \end{bmatrix}' J \begin{bmatrix} V \\ W \end{bmatrix}, \qquad J = \begin{bmatrix} \mathbf{1}_n & \\ & -\mathbf{1}_n \end{bmatrix}.$$

The Schur-Cholesky algorithm, as presented in [19] and [17] solves this symmetric linear system determines a J-orthogonal Θ satisfying

$$\Theta \begin{bmatrix} V \\ W \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix}, \quad \Theta' J \Theta = J.$$

The matrix Θ being J-orthogonal results in the identity

$$\begin{bmatrix} V \\ W \end{bmatrix}' \Theta' J \Theta \begin{bmatrix} V \\ W \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix}' J \begin{bmatrix} R \\ 0 \end{bmatrix} = V'V - W'W = R'R,$$

where the matrix R is an upper triangular matrix, hence, a triangular factor of C. If we complete this map by extending it to the orthogonal space we arrive at the full equation

$$\Theta \begin{bmatrix} V & W' \\ W & V' \end{bmatrix} = \begin{bmatrix} R & 0 \\ 0 & L \end{bmatrix}, \quad C = LL',$$

where L is supposed to be a lower triangular factor of C. Assuming that all necessary matrices are invertible, then we can devise an expression for the transformation

$$\Theta = \begin{bmatrix} (R')^{-1}V' & -(R')^{-1}W' \\ -(L')^{-1}W & (L')^{-1}V \end{bmatrix}. \tag{3.34}$$

Similar to the approach for computing the QR decomposition using Jacobi rotations (compare with Sect. 3.3.2) we can compute the overall transformation Θ using elementary J-orthogonal or hyperbolic rotations. The elementary hyperbolic matrix is a rotation over an angle θ in the 2D plane (for convenience we define H'):

$$H' = \begin{bmatrix} c_h & s_h \\ -s_h & c_h \end{bmatrix}, \quad \text{using} \quad \begin{matrix} c_h := \cosh\theta \\ s_h := \sinh\theta \end{matrix} \tag{3.35}$$

Let's apply such a rotation H' to two row vectors such as

$$\begin{bmatrix} c_h & s_h \\ -s_h & c_h \end{bmatrix} \begin{bmatrix} a_1 & a_2 & \cdots & a_n \\ b_1 & b_2 & \cdots & b_n \end{bmatrix} = \begin{bmatrix} \sqrt{a_1^2 - b_1^2} & c_h a_2 + s_h b_2 & \cdots & c_h a_n + s_h b_n \\ 0 & -s_h a_2 + c_h b_2 & \cdots & -s_h a_n + c_h b_n \end{bmatrix} \tag{3.36}$$

which is achieved by choosing

$$c_h = \frac{a_1}{\sqrt{a_1^2 - b_1^2}}, \quad s_h = \frac{b_1}{\sqrt{a_1^2 - b_1^2}},$$

and hence automatically $\tanh\theta = \frac{b_1}{a_1}$.

In this way one can treat the entries of the original matrix T row by row and create all the zeros below the main diagonal. With a 3×3 matrix C this works as shown below. The only thing one must do is embed the 2×2 rotation matrices in the 4×4 schema, so that unaffected rows remain unchanged. We label the hyperbolic rotation matrices with the indices of the rows they affect

$$\begin{bmatrix} \cdot & \cdot & \cdot \\ & \cdot & \cdot \\ & & \cdot \\ \hline \cdot & \cdot & \cdot \\ & \cdot & \cdot \\ & & \cdot \end{bmatrix} \xrightarrow{H'_{1,4}} \begin{bmatrix} \star & \star & \star \\ & \cdot & \cdot \\ & & \cdot \\ \hline 0 & \star & \star \\ & \cdot & \cdot \\ & & \cdot \end{bmatrix} \xrightarrow{H'_{2,5}} \begin{bmatrix} \cdot & \cdot & \cdot \\ & \star & \star \\ & & \cdot \\ \hline 0 & \cdot & \cdot \\ & 0 & \star \\ & & \cdot \end{bmatrix} \xrightarrow{H'_{3,6}} \begin{bmatrix} \cdot & \cdot & \cdot \\ & \cdot & \cdot \\ & & \star \\ \hline 0 & \cdot & \cdot \\ & 0 & \cdot \\ & & 0 \end{bmatrix} \xrightarrow{H'_{2,4}} \tag{3.37}$$

$$\xrightarrow{H'_{2,4}} \begin{bmatrix} \cdot & \cdot & \cdot \\ & \star & \star \\ & & \cdot \\ \hline 0 & 0 & \star \\ & 0 & \cdot \\ & & 0 \end{bmatrix} \xrightarrow{H'_{3,5}} \begin{bmatrix} \cdot & \cdot & \cdot \\ & \cdot & \cdot \\ & & \star \\ \hline 0 & 0 & \cdot \\ & 0 & 0 \\ & & 0 \end{bmatrix} \xrightarrow{H'_{3,4}} \begin{bmatrix} \cdot & \cdot & \cdot \\ & \cdot & \cdot \\ & & \star \\ \hline 0 & 0 & 0 \\ & 0 & 0 \\ & & 0 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ & r_{22} & r_{23} \\ & & r_{33} \\ \hline 0 & 0 & 0 \\ & 0 & 0 \\ & & 0 \end{bmatrix} \tag{3.38}$$

The final step is a $H'_{3,4}$ which annihilates the $4,3$ entry. In each of these subsequent steps, the first operation determines the rotation matrix and then applies it to all the entries in the respective rows, skipping the already computed zero entries (which remain zero hence avoiding fill-ins). The overall J-orthogonal matrix is then determined as the product of the elementary hyperbolic rotation $\Theta = H_{1,4}H_{2,5}H_{3,6}H_{2,4}H_{3,5}H_{3,4}$. Note that rotation steps with non-overlapping row indices can be carried out in parallel, i.e. the sequence of rotations $H_{1,4}$, $H_{2,5}$, and $H_{3,6}$ can be executed simultaneously on a dedicated processor array as well as the rotations $H_{2,4}$ and $H_{3,5}$.

After having computed the J-orthogonal matrix Θ by a sequence of hyperbolic rotations we can take the representation of the J-orthogonal transformation as given in (3.34), and determine the effect of applying Θ to two tacked identity matrices, i.e. we can compute

$$\Theta \begin{bmatrix} \mathbf{1}_n \\ \mathbf{1}_n \end{bmatrix} = \begin{bmatrix} (R')^{-1}(V' - W') \\ (L')^{-1}(V - W) \end{bmatrix} = \begin{bmatrix} (R')^{-1} \\ (L')^{-1} \end{bmatrix}.$$

This expression shows that the process of elementary elimination steps implicitly creates the inverses of the triangular factors of the matrix C. The symmetric system of equation can be used in a straight forward manner with appropriate back-substitution steps. However, back-substitution destroys the homogenous data flow and is therefore not desirable for parallel processing hardware. Applying the Fadeeva approach to this algorithm [17], similar to the approach used for the QR solver, allows us to devise a purely feed forward algorithm that avoids all back-substitution steps. This version of the algorithm is based on performing the following two stages in an elimination process that employs hyperbolic rotations. Stage 1 executes the elimination of the matrix W by hyperbolic rotations, as explained previously creating the matrix Θ_1 according to

$$\Theta_1 \begin{bmatrix} V & \mathbf{1}_n & 0 \\ W & \mathbf{1}_n & 0 \\ -b' & 0 & 1 \end{bmatrix} = \begin{bmatrix} R & (R')^{-1} & 0 \\ 0 & (L')^{-1} & 0 \\ -b' & 0 & 1 \end{bmatrix}.$$

Stage 2 of the elimination process annihilates the entries of the vector $-b'$ while creating the matrix Θ_2

$$\Theta_2 \begin{bmatrix} R & (R')^{-1} & 0 \\ 0 & (L')^{-1} & 0 \\ -b' & 0 & 1 \end{bmatrix} = \begin{bmatrix} \tilde{R} & (\tilde{R'})^{-1} & \star \\ 0 & (L')^{-1} & 0 \\ 0 & ku' & k \end{bmatrix}.$$

After both stages of elimination are completed, the solution vector $k \cdot u'$ can be read off from the resulting array as well as scalar parameter k. Figure 3.7 depicts the processing array to implement the Schur-Cholsky algorithm for solving symmetric positive systems of equations in a highly parallel and regular way and without a need to perform back-substitution [17].

3.6 Sparse Matrices and Iterative Algorithms

Large scale linear systems, which are characterized by a coefficient matrix of enormous size are a topic of particular interest from a practical point of view. Luckily, such matrices are mostly sparse, which means that only a small fraction of the matrix entries are different from zero. Sparse matrices exhibit very large values for n, but only $\mathcal{O}(n)$ matrix entries are different from zero. The sparsity allows the matrices to be stored with $\mathcal{O}(n)$ memory locations [36]. This type of matrices originate for example from finite-element computations, from solving partial differential or Euler-Lagrange equations to name a few examples [36]. Applying standard matrix algebra does not preserve the sparsity pattern, that is, it destroys the sparsity pattern when adding, multiplying and inverting sparse matrices. For example, the inverse of a tri-band matrix is in general not a tri-band matrix, but

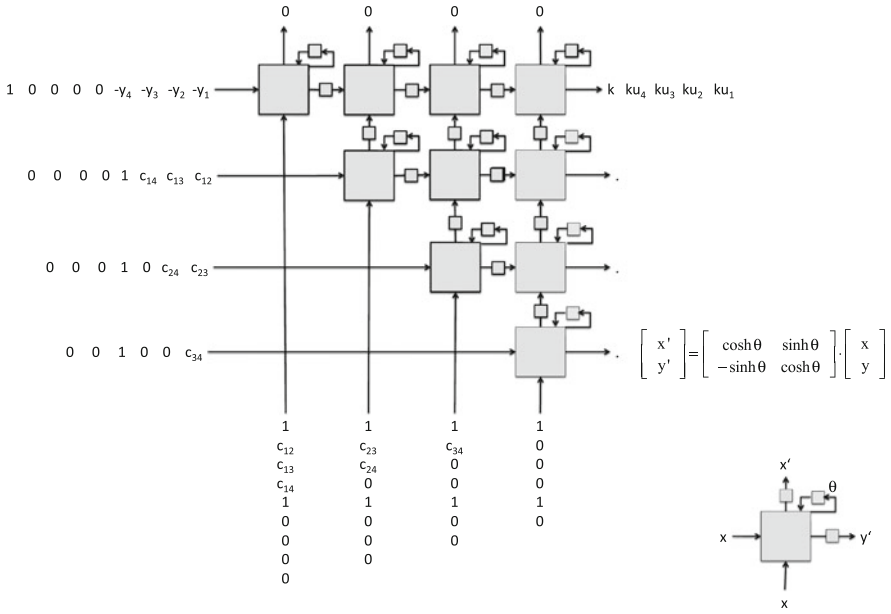


Fig. 3.7 The array for the Schur-Cholesky algorithm

a full matrix. Similar statements hold if matrix factorizations and dense solvers are applied to sparse matrices; elementary transformations tend to fill up the matrix by creating fill-ins, i.e. by overwriting zero matrix entries with non-zero values [2, 36].

Algorithms to efficiently solve systems of equations with a sparse coefficient matrix use iterative approaches, which are using a sequence of matrix vector multiplications of the form $u_{k+1} = T \cdot u_k$. This way, the sparsity pattern of the coefficient matrix is preserved and the sequence of vectors u_k converges, under certain conditions to the solution vector u. Matrix-vector multiplication with a sparse matrix $T \in \mathbb{R}^{m \times n}$ amounts to a computational load of $\mathcal{O}(m + n)$ operations and $\mathcal{O}(m + n)$ memory requirement. Iterative schemes such as a Conjugate Gradient algorithm require $\mathcal{O}(n)$ iterations, resulting in an solution method with a computational complexity of $\mathcal{O}(n^2)$ operations overall. For a comprehensive coverage of iterative solution methods we refer the interested reader to [36].

3.6.1 Sparse Matrices in Motion Analysis

For many applications in the domain of computer vision or in the field of digital video signal processing the task of estimating the apparent motion of objects or pixels throughout a video sequence is a fundamental task. Motion estimation is an expensive calculation, in particular when considering to deal with standard

definition resolution images (576×720 pixels per image) or moving on to even handle High Definition resolutions ($1,080 \times 1,920$ pixels per image) [8].

3.6.1.1 Optic Flow Constraint

We discuss how to compute the optical flow according to the approach proposed by Horn & Schunck [35]. The brightness of a pixel at point (x, y) in an image plane at time t is denoted by $I(x, y, t)$. Let $I(x, y, t)$ and $I(x, y, t + 1)$ be two successive images of a video sequence. Each image is comprised of a rectangular lattice of $N = m \times n$ pixels. Optic flow computation is based on the assumption that the brightness of a pixel remains constant in time and that all apparent variations of the brightness throughout a video sequence are due to spatial displacements of the pixels, which again are caused by motion of objects. We denote this brightness conservation assumption as

$$\frac{dI}{dt} = 0.$$

This equation is called the optical flow constraint. Using the chain rule for differentiation the optic flow constraint is expanded into

$$\frac{\partial I}{\partial x} \cdot \frac{dx}{dt} + \frac{\partial I}{\partial y} \cdot \frac{dy}{dt} + \frac{\partial I}{\partial t} = 0.$$

Using the shorthand notation $v_x = \frac{dx}{dt} \quad v_y = \frac{dy}{dt}, I_x = \frac{\partial I}{\partial x}, I_y = \frac{\partial I}{\partial y}, I_t = \frac{\partial I}{\partial t}$, the optic flow constraint can be written as

$$E_{of} = I_x \cdot v_x + I_y \cdot v_y + I_t = 0. \tag{3.39}$$

Equation (3.39) is only one equation for determining the two unknowns v_x and v_y, which denote the horizontal and the vertical component of the motion vector at each pixel position. Hence the optical flow equation is an underdetermined system of equation. Solving this equation in a least squares sense only produces the motion vector component in direction of the strongest gradient for the texture. Therefore a second constraint has to be found to regularize this ill-posed problem.

3.6.1.2 Smoothness Constraint

To overcome the underdetermined nature of the optic flow constraint, Horn & Schunck introduced an additional smoothness constraint. Neighboring pixels of an object in a video sequence are likely to move in a similar way. The motion vectors v_x and v_y are varying spatially in a smooth way. Spatial discontinuities in the motion vector field occur only at motion boundaries between objects, which move in different directions and which are occluding each other. Therefore, the motion

vector field to be computed is supposed to be spatially smooth. This smoothness constraint can be formulated using the Laplacian of the motion vector field v_x and v_y

$$E_{sc} = \nabla^2 v_x + \nabla^2 v_y = \frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2} + \frac{\partial^2 v_y}{\partial x^2} + \frac{\partial^2 v_y}{\partial y^2}. \tag{3.40}$$

The Laplacians of v_x and v_y can be calculated by the approximation

$$\nabla^2 v_x \approx \bar{v}_x - v_x \quad \text{and} \quad \nabla^2 v_y \approx \bar{v}_y - v_y.$$

The term $\bar{v}_{x,y} - v_{x,y}$ can be computed numerically as the difference between the central pixel $v_{x,y}$ and a weighted average of the values in a 2-neighborhood of the central pixel. The corresponding 2-dimensional convolution for performing this filtering operation is given as

$$\bar{v}_x(x, y) - v_x(x, y) = \mathcal{L}(x, y) * v_x(x, y)$$

$$\bar{v}_y(x, y) - v_y(x, y) = \mathcal{L}(x, y) * v_x(x, y),$$

where the convolution kernel $\mathcal{L}(x, y)$ is given by a 2D-filtering mask, such as

$$\mathcal{L}(x, y) = \begin{bmatrix} 1/12 & 1/6 & 1/12 \\ 1/6 & -1 & 1/6 \\ 1/12 & 1/6 & 1/12 \end{bmatrix}. \tag{3.41}$$

The Horn & Schunck approach uses the optic flow equation (3.39) along with the smoothness constraint (3.40) to express the optic flow computation as the optimization problem for the cost function

$$E^2 = E_{of}^2 + \alpha^2 \cdot E_{sc}^2,$$

which needs to be minimized in terms of the motion vector $[v_x v_y]^T$. The parameter α is a scalar regularization parameter, which controls the contribution of the smoothness constraint (3.40). The optimization problem finally expands into the equation

$$E^2 = \left(I_x v_x + I_y v_y + I_t\right)^2 + \alpha^2 \left((\bar{v}_x - v_x)^2 + (\bar{v}_y - v_y)^2\right). \tag{3.42}$$

Applying the calculus of variations to (3.42) results in the following two equations

$$I_x^2 v_x + I_x I_y v_y + I_x I_t - \alpha^2(\bar{v}_x - v_x) = 0$$
$$I_x I_y v_x + I_y^2 v_y + I_y I_t - \alpha^2(\bar{v}_y - v_y) = 0,$$

which need to be solved for the motion vector components $v_x(x, y)$ and $v_y(x, y)$.

3.6.1.3 Linear System of Equations

Based on (3.39), the optic flow equation for all pixels can be written as a matrix equation [8]

$$\left[\mathbf{I}_x\ \mathbf{I}_y\right] \cdot \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \end{bmatrix} = -\mathbf{I}_t, \tag{3.43}$$

where $\mathbf{I}_x$ and $\mathbf{I}_y$ are diagonal matrices of size $N \times N$ and $\mathbf{I}_t$ is a vector of length N. The x- and y-components of the motion vector field are given as the vectors $\mathbf{v}_x$ and $\mathbf{v}_y$, each with the dimension N. The effect of the convolution kernel $\mathcal{L}(x, y)$ on the motion vector field can be represented by a constant and sparse $N \times N$ band matrix $\mathbf{L}$, which has Toeplitz structure. We will use the symbol $\boldsymbol{C}$ to denote the negative Laplacian, i.e. $\boldsymbol{C} = -\mathcal{L}$. The specific banded structure of C is depicted in Fig. 3.8,which can be symbolically denoted as

$$\boldsymbol{C} = \begin{bmatrix} \boldsymbol{M} & \boldsymbol{U} & & & & \\ \boldsymbol{U} & \boldsymbol{M} & \boldsymbol{U} & & & \\ & \boldsymbol{U} & \boldsymbol{M} & \boldsymbol{U} & & \\ & & \boldsymbol{U} & \ddots & \ddots & \\ & & & \ddots & \ddots & \boldsymbol{U} \\ & & & & \boldsymbol{U} & \boldsymbol{M} \end{bmatrix}.$$

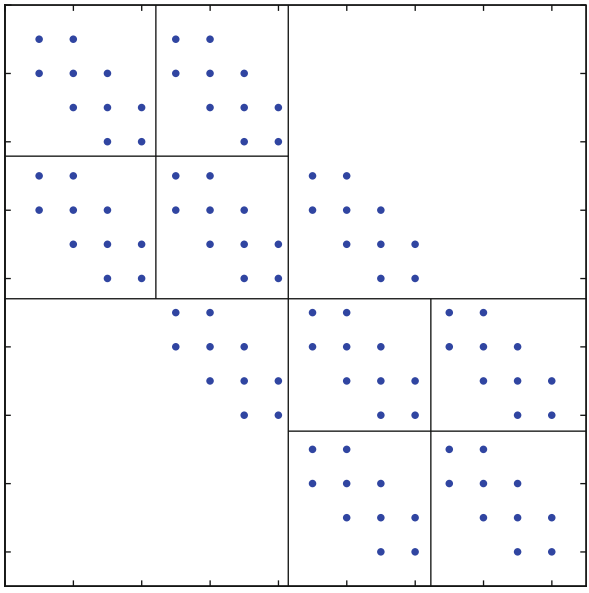


Fig. 3.8 Structure for the Laplace operator

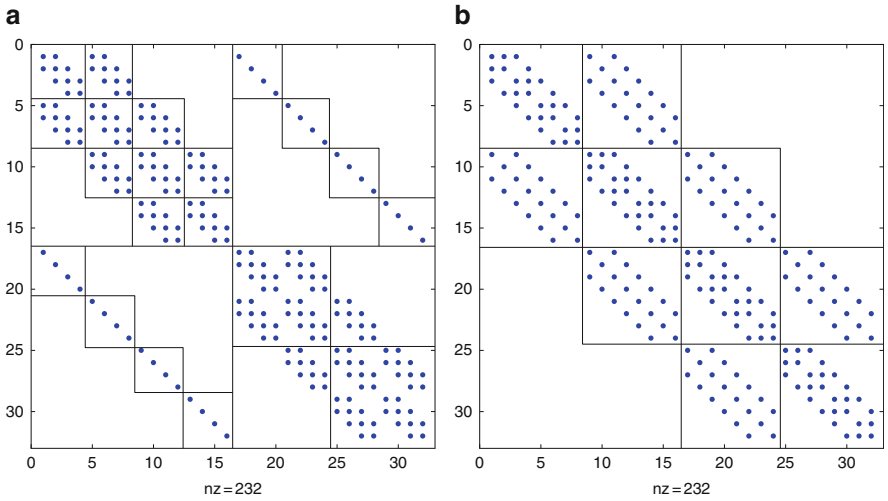


Fig. 3.9 Structure for original matrix (*left*) and re-ordered matrix (*right*)

The minimization of the cost function (3.42) for all pixels in the image leads to the set of regularized linear equations,

$$\left(\alpha^2 \begin{bmatrix} C & 0 \\ 0 & C \end{bmatrix} + \begin{bmatrix} \mathbf{I}_x \\ \mathbf{I}_y \end{bmatrix} \cdot \begin{bmatrix} \mathbf{I}_x & \mathbf{I}_y \end{bmatrix}\right) \cdot \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \end{bmatrix} = \begin{bmatrix} \mathbf{I}_x \\ \mathbf{I}_y \end{bmatrix} \cdot \mathbf{I}_t . \tag{3.44}$$

The term in brackets of (3.44) represents a $2N \times 2N$ band matrix, the structure of which can be seen on the left hand side of Fig. 3.9. This structured matrix needs to be solved efficiently for computing the motion vector fields $\mathbf{v}_x$ and $\mathbf{v}_y$. We can modify the structure of the matrix by re-ordering the variables and hence the matrix entries. In the right hand side of Fig. 3.9 the resulting matrix structure is shown if the variables v_x and v_y are re-ordered by interleaving them. Such a change of structure for the matrix entries has influence on the efficiency of sparse linear system solvers [2, 36].

Lucas and Kanade proposed an alternative scheme for computing optical flow [29], which is better suitable for implementation on parallel processors such as a Graphic Processor Unit (GPU) [10].

In [9] an approach is presented to solve the sparse linear system arising from optical flow computations in an efficient way by exploiting the structure of the matrix. This structure is called *hierarchically semi-separable*, a notion that will be explained in more detail in Sect. 3.7.

3.6.2 Iterative Matrix Solvers

Iterative methods take an initial approximation of the solution vector and successively improve the solution by continued matrix vector multiplications until an acceptable solution has been reached [2]. Since matrix-vector multiplications can be computed efficiently for sparse matrices iterative matrix solvers are attractive. However, iterative methods may have poor robustness and often are only applicable for a rather narrow range of applications. With view to implementing such schemes on real-time computer systems we need to understand the convergence properties of iterative schemes, which strongly depend on the input data. Therefore it is difficult to determine worst case deadlines for the iterations to be completed [2].

3.6.2.1 General Approach

Iterative algorithms do not explicitly compute the term $T'T$, a term that appears in the context of normal equation, but only propagate the effects and T' and T in a factored form, and hence exploit the sparsity of the coefficient matrix T. This leads to the following representation of the standard least-squares problem

$$T' \cdot (Tu - b) = 0.$$

Stationary iterative methods for solving linear systems of equations run the iteration of the form

$$Mu_{k+1} = Nu_k + b, \quad k = 1, 2, 3, \ldots,$$

where u_0 is an initial approximation for the solution vector. In this context we have the "splitting" of the positive definite matrix $T'T = M - N$ where M is assumed to be non-singular. The matrix M should be chosen such, that solving the linear system with coefficient matrix M is easy to do. Analyzing the equation

$$u_{k+1} = M^{-1}Nu_k + M^{-1}b = Gu_k + c, \quad k = 1, 2, 3, \ldots,$$

reveals that the iterative scheme is convergent if the spectral radius of the matrix $G \in \mathbb{R}^{m \times m}$

$$\rho(G) = \max |\lambda_i(G)|, \quad 1 \leq i \leq m$$

satisfies $\rho(G) < 1$. Taking the additive splitting of the coefficient matrix as

$$T'T = L + D + L', \quad D \geq 0,$$

then we get the Jacobi method if we choose $M = D$, whereas the choice $M = L + D$ will lead us to the Gauss-Seidel iterative scheme. The Gauss-Seidel has better convergence properties when compared with Jacobi-methods (one Gauss-Seidel

iteration step corresponds to two Jacobi iterations). However, Jacobi-type methods are better suited for parallel implementation [2].

For executing iterative matrix solvers on parallel computing architectures it is essential to implement an efficient way of matrix vector multiplication. This is possible only for a predetermined sparsity pattern. See [15] for a corresponding array to execute iterative matrix solvers.

3.6.2.2 LSQR Algorithm

An attractive alternative to Jacobi- or Gauss-Seidel-iterations is the family of quasi-iterative methods such as Conjugate Gradient techniques. One very interesting algorithm from this family is Paige and Saunder's LSQR algorithm for solving least squares problems [32]. The LSQR technique combines matrix-vector multiplication based steps (Lanczos and Arnoldi methods) with a QR decomposition step. The starting point for LSQR is to compute the factorization

$$T = V \begin{bmatrix} B \\ 0 \end{bmatrix} W', \quad V'V = \mathbf{1}_m, \quad W'W = \mathbf{1}_n,$$

where

$$B = B_n = \begin{bmatrix} \alpha_1 & & & \\ \beta_2 & \alpha_2 & & \\ & \beta_3 & \ddots & \\ & & \ddots & \alpha_n \\ & & & \beta_{n+1} \end{bmatrix} \in \mathbb{R}^{(n+1)\times n}$$

is a lower bi-diagonal matrix. The orthogonal matrices $V = (v_1, v_2, \ldots, v_m)$ and $W = (w_1, w_2, \ldots, w_n)$ can be computed as a product of Householder reflections or Jacobi rotations for the elimination of the matrix entries in T. However, this process will destroy the sparsity pattern of the coefficient matrix. For the case of sparse matrices, Golub and Kahan suggested an alternative procedure to compute this factorization, which is based on a Lanczos process. This process is again using matrix-vector multiplications to propagate vectors and to leave the coefficient matrix T unchanged, hence preserving its sparsity pattern. The Lanczos process starts by setting $\beta_1 w_0 = 0$ and $\alpha_{n+1} w_{n+1} = 0$. If we initialize the process with the vector $\beta_1 v_1 = u \in \mathbb{R}^m$ and $\alpha_1 w_1 = T' v_1$ then, for $k = 1, 2, \ldots$ the recurrence relations

$$\beta_{k+1} v_{k+1} = T w_k - \alpha_k v_k, \quad \alpha_{k+1} w_{k+1} = T' v_{k+1} - \beta_{k+1} w_k,$$

continue to produce the sequence of vectors $w_1, v_2, w_2, \ldots, v_{m+1}$ and the corresponding entries in the bi-diagonal matrix B. Note that the parameters $\alpha_{k+1} \geq 0$ and $\beta_{k+1} \geq 0$ are determined such that $\|v_{k+1}\|_2 = \|w_{k+1}\|_2 = 1$. After k steps the algorithm produces the matrices $V = (v_1, v_2, \ldots, v_k)$ and $W = (w_1, w_2, \ldots, w_{k+1})$

as well as the bi-diagonal matrix B_k. We now can go for an approximate solution vector $u_k \in \mathcal{K}_k$ lying in the Krylov subspace $\mathcal{K}_k = \mathcal{K}_k(T'T, T'y)$, where a Krylov subspace is defined as $\mathcal{K}_k(T,u) = span\left[u, Tu, T^2u, \ldots, T^{k-1}u\right]$. Since in the present case we have $\mathcal{K}_k = span(W_k)$ it is possible to write $u_k = W_k\psi_k$.

For computing the vector ψ_k we have to determine a solution of the least-square problem

$$\min_{\psi_k} \|B_k\psi_k - \beta_1 e_1\|_2.$$

The LSQR algorithm calculates this solution via the QR decomposition of B_k

$$Q_k B_k = \begin{bmatrix} R_k \\ 0 \end{bmatrix}, \quad Q_k(\beta_1 e_1) = \begin{bmatrix} f_k \\ \phi_{k+1} \end{bmatrix},$$

where R_k is upper bi-diagonal

$$R_k = \begin{bmatrix} \rho_1 & \theta_1 & & & \\ & \rho_2 & \theta_2 & & \\ & & \ddots & \ddots & \\ & & & \rho_{k-1} & \theta_{k-1} \\ & & & & \rho_k \end{bmatrix} \in \mathbb{R}^{k\times k}, \quad f_k = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{k-1} \\ \phi_k \end{bmatrix}.$$

Notice the similarity of this approach with the algorithm described in Sect. 3.3.1. The matrix Q_k is computed as a product of Jacobi rotations parametrized to eliminate the subdiagonal elements of B_k. The solution vector ψ_k and the residual vector r_{k+1} can be determined from

$$R_k\psi_k = f_k, \quad r_{k+1} = Q_k' \begin{bmatrix} 0 \\ \phi_{k+1} \end{bmatrix}.$$

The iterative solution u_k is then computed via

$$u_k = \left(W_k R_k^{-1}\right) f_k = Z_k f_k,$$

where the matrix Z_k satisfies the lower triangular system $R_k' Z_k' = V_k'$ such that we can computed the column vectors $(z_1, z_2, \ldots, z_k)$ by forward substitution. With $z_0 = u_0 = 0$ the process proceeds as

$$z_k = \frac{1}{\rho_k}\left(w_k - \theta_k z_{k-1}\right), \quad u_k = u_{k-1} + \phi_k z_k.$$

The LSQR algorithm produces the same sequence of intermediate solution vectors u_k as the Conjugate Gradient Least Squares algorithm [2]. The algorithm only accesses the coefficient matrix T only to produce the matrix-vector products Tw_k and $T'v_k$ and exhibits preferable numerical properties, in particular for ill-conditioned matrices T [32].

3.6.2.3 Iterative Algorithms and Memory Bandwidth

An efficient design of an numerical algorithm strives to minimize the amount of arithmetic operations along with the amount of memory space needed to store the data. Besides those two important design criteria we have to consider the amount of data that needs to be moved in and out of main memory during the execution of an algorithm; this is denoted as memory bandwidth. Even a modern real-time computer system may be challenged to provide excessive sustained memory bandwidth if it executes an iterative matrix solver for very large and sparse matrices. The challenge originates from iterating the solution vector, that is, from moving a potentially very large vector in and out of main memory for each iteration as it will not fit into the cache memory anymore.

For calculating a simple example we consider a problem where the solution vector u has length n. Each vector entry is represented by a double precision floating point number using a word length of $B = 64$ bits. We assume that the iterative algorithm requires N iterations until convergence. Hence, the algorithm needs to move $n \cdot N \cdot B$ bits of data between the CPU and Memory. In a real-time application such as in video signal processing (deconvolution, motion estimation, scan conversion, etc.) the algorithm has to complete its calculations within a time interval of T seconds, which leads to a required memory bandwidth of $\frac{n \cdot N \cdot B}{T}$ bits per second. For a modern HDTV application ($1,920 \times 1,080$ pixels per frame) with a frame rate of 25 Hz, the necessary memory bandwidth amounts to over 41 GByte/s, if we assume that the computation for one system of equation converges after $N = 100$ iterations. Even under these optimistic assumptions for the number of iterations the memory bandwidth becomes the critical element [33].

If the real-time computer system employs modern Graphical Processing Units (GPU), which offer high computational performance, this big amount of data needs to be transported over a bus between CPU and GPU [3,26]. Hence, the data transport becomes a bottleneck to achieve the required system performance. Furthermore, the internal structure of a GPU, i.e. the shader does not support iterative computations very well. Taking all this together leads us to reconsider the use of direct solution methods for large scale systems as the pure operation count is not the most restricting factor [10]. If there are direct solution methods, which can take advantage of special structures in the coefficient matrix, such as sparsity, then this is beneficial.

3.7 Structure in Matrices

The classical QR algorithm on an $n \times n$ matrix T has computational complexity of order $\mathcal{O}(n^3)$. Better computational complexity, at the cost of numerical stability, is offered by "Strassens" method, but is certainly not advisable in the context of embedded processing, where numerical complexity plays only a minor role as compared to data transport and where good numerical properties are of paramount importance. There is, however, another much more promising possibility, which

consists in exploiting intrinsic structure of the matrix. In the literature, many diverse types of matrix structure have been considered, such as Toeplitz or Hankel, in the present discussion we shall only consider the semi-separable or time-varying structure as it connects up nicely with QR factorization and is very important from an applications point of view. In our discussion on the Kalman filter, we were given a model for the system to be considered. That meant that the overall covariance matrix of the output data is not arbitrary, although the system parameters vary from one time point to the next. As a result, the Kalman filter in its square root version allowed state estimation on the basis not of the overall covariance data, but just on the basis of properties local to each point in time. The computational complexity was therefore determined, not by the size of the overall system, but by the dimension of the local state representation (called δ_i.)

The structure we encountered implicitly in the Kalman filter is known by various terms, depending on the relevant literature, semi-separable systems, time-varying systems or quasi-separable systems. They appeared for the first time in [13], where it was shown that LU factorization of such a system would have $\mathcal{O}(n\overline{\delta_i}^3)$ complexity instead of $\mathcal{O}(n^3)$. Later this idea was generalized to QR and other factorizations in [6]. We summarize the main results. It should be remarked that the complexity of the method we shall describe can be considerably better than $\mathcal{O}(n\overline{\delta_i}^3)$, actually $\mathcal{O}(n\overline{\delta_i}^2)$ when state space representations are judiciously chosen, but this topic goes far beyond our present purpose.

To work comfortably with time-varying systems, we need the use of sequences of indices and then indexed sequences. When $\mathcal{M} = [m_k]_{k=-\infty}^{\infty}$ is a sequence of indices, then each m_k is either a positive integer or zero, and a corresponding indexed sequence $[u_k] \in \ell_2^{\mathcal{M}}$ will be a sequence of vectors such that each u_k has dimension m_k and the overall sum

$$\sum_{k=-\infty}^{\infty} \|u_k\|^2 \tag{3.45}$$

is finite, the square root of which is then the quadratic norm of the sequence. When $m_k = 0$, then the corresponding entry just disappears (it is indicated as a mere "place holder"). A regular n-dimensional finite vector can so be considered as embedded in an infinite sequence, whereby the entries from $-\infty$ to zero and $n+1$ to ∞ disappear, leaving just n entries indexed by $1 \cdots n$, corresponding e.g. to the time points where they are fed into the system. On such sequences we may define a generic shift operator Z. It is also convenient to represent sequences in row form, underlying the zero'th element for orientation purposes, taking transposes of the original vectors if they are in column form. Hence:

$$[\cdots, u'_{-2}, u'_{-1}, \underline{u'_0}, u'_1, u'_2, \cdots]Z = [\cdots, u'_{-2}, \underline{u'_{-1}}, u'_0, u'_1, \cdots] \tag{3.46}$$

Z is then a unitary shift represented as a block upper unit matrix, whose inverse Z' is then a lower matrix with first lower block diagonal consisting of unit matrices

(notice that the indexing of the rows is shifted w.r. to the indexing of the columns.) Typically we handle only finite sequences of vectors, but the embedding in infinite ones allows us to apply delays as desired and not worry about the precise time points. Similarly, we handle henceforth matrices in which the entries are matrices themselves. For example, $T_{i,j}$ is a block of dimensions $m_i \times n_j$ with $[m_i] = \mathcal{M}$ and $[n_j] = \mathcal{N}$, and, again, unnecessary indices are just placeholder, with the corresponding block entries disappearing as well – also consisting of place holders (interesting enough, MATLAB now allows for such matrices, the lack of which was a major problem in previous versions. Place holders are very common in computer science, here they make their entry in linear algebra).

In this convention we define a causal system by the set of equations

$$\begin{cases} x_{i+1} = A_i x_i + B_i u_i \\ y_i \quad = C_i x_i + D_i u_i \end{cases} \tag{3.47}$$

very much as before, but now with a direct term $D_i u_i$ added to the output equation (in the Kalman filter this term is zero, because the prediction is done strictly on past values). $\begin{bmatrix} A_i & B_i \\ C_i & D_i \end{bmatrix}$ is called the *system transition matrix* at time point i (A_i being the state transition matrix). What is the corresponding input/output matrix T? As is tradition in system theory, we replace the local equations above with global equations on the (embedded) sequences $u = [u_i]$, $y = [y_i]$ and $x = [x_i]$, and define "global" block diagonal matrices $A = \text{diag}(A_i)$, $B = \text{diag}B_i$, etc... to obtain

$$\begin{cases} Zx = Ax + Bu \\ y \quad = Cx + Du \end{cases} \tag{3.48}$$

and for the input–output matrix

$$T = D + CZ'(I - AZ')^{-1}B. \tag{3.49}$$

This represents a block lower matrix in semi-separable form. A block upper matrix would have a similar representation, now with Z replacing Z':

$$T = D + CZ(I - AZ)^{-1}B \tag{3.50}$$

Remark. it is not necessary to change to a row convention to make the theory work as in [6]. Instead of defining Z as pushing forward on a row of data, we have defined Z' as pushing forward on a column of data. Hence, $Z' = Z^{-1}$ is the causal shift, and a causal matrix is lower (block) triangular, rather than upper.

Such representations, often called *realizations*, produce in a nutshell the special structure of an upper, semi-separable system. When T is block banded upper with two bands, then $A = 0$ and $B = I$ will do, the central band is represented by D

and the first off band by C. With a block three band, one can choose $A = \begin{bmatrix} 0 & 0 \\ I & 0 \end{bmatrix}$, $C = \begin{bmatrix} C_1 & C_2 \end{bmatrix}$ and $B = \begin{bmatrix} I \\ 0 \end{bmatrix}$, with $Z := \begin{bmatrix} Z & \\ & Z \end{bmatrix}$ because the state splits in two components. We find, indeed, $Z(I - AZ)^{-1} := \begin{bmatrix} Z & 0 \\ Z^2 & Z \end{bmatrix}$, and hence $T = D + C_1 Z + C_2 Z^2$. This principle can easily be extended to yield representations for multi-band matrices or matrix polynomials in Z.

State space representations are not unique. The dimension chosen for x_i at time point i may be larger than necessary, in which case one would call the representation "non minimal" – we shall not consider this case further. Assuming a minimal representation, one could also introduce a non singular state transformation R_i at each time point, defining a transformed state $\hat{x}_i = R_i^{-1} x_i$. The transformed system transition matrix now becomes

$$\begin{bmatrix} \hat{A}_i & \hat{B}_i \\ \hat{C}_i & D_i \end{bmatrix} := \begin{bmatrix} R_{i+1}^{-1} A_i R_i & R_{i+1}^{-1} B_i \\ C_i R_i & D_i \end{bmatrix}. \tag{3.51}$$

for a lower system, and a similar, dual representation for the upper.

Given a block upper matrix T, what is a minimal system representation for it? This problem is known as the *system realization problem*, and was solved for the first time by Kronecker (for representations of rational functions [25]), and then later by various authors, for the semi-separable case, see [6] for a complete treatment. An essential role in realization theory is played by the so called i^{th} Hankel matrix H_i defined as

$$H_i = \begin{bmatrix} \vdots & \vdots & \ddots \\ T_{i-1,i+1} & T_{i-1,i+2} & \cdots \\ T_{i,i+1} & T_{i,i+2} & \cdots \end{bmatrix} \tag{3.52}$$

i.e. a right-upper corner matrix just right of the diagonal element $T_{i,i}$. It turns out that any minimal factorization of each H_i yields a minimal realization, we have indeed

$$H_i = \begin{bmatrix} \vdots \\ C_{i-2} A_{i-1} A_i \\ C_{i-1} A_i \\ C_i \end{bmatrix} \begin{bmatrix} B_{i+1} & A_{i+1} B_{i+2} & A_{i+1} A_{i+2} B_{i+3} \cdots \end{bmatrix} \tag{3.53}$$

where, as explained before, entries may disappear when they reach the border of the matrix e.g. This decomposition has an attractive physical meaning. We recognize

$$\mathcal{O}_i = \begin{bmatrix} \vdots \\ C_{i-2}A_{i-1}A_i \\ C_{i-1}A_i \\ C_i \end{bmatrix} \tag{3.54}$$

as the i^{th} *observability operator*, and

$$\mathcal{R}_i = \begin{bmatrix} B_{i+1} & A_{i+1}B_{i+2} & A_{i+1}A_{i+2}B_{i+3} & \cdots \end{bmatrix} \tag{3.55}$$

as the i^{th} *reachability operator* – all these related to the (anti-causal) upper operator we assumed. $\mathcal{R}_i$ maps inputs after the time point i to the state x_i, while $\mathcal{O}_i$ maps state x_i to actual and outputs before index point i, giving its linear contribution to them. The rows of $\mathcal{R}_i$ form a basis for the rows of H_i, while the columns of $\mathcal{O}_i$ form a basis for the columns of H_i in a minimal representation. When e.g. the rows are chosen as an orthonormal basis for all the H_i, then a realization will result for which $A_i A_i' + B_i B_i' = I$ for all i. We call a realization in which $\begin{bmatrix} A_i & B_i \end{bmatrix}$ has this property of being part of an orthogonal or unitary matrix, in *input normal form*.

It may seem laborious to find realizations for common systems. Luckily, this is not the case. In many instances, realizations come with the physics of the problem. Very common are, besides block banded matrices, so called smooth matrices [4], in which the Hankel matrices have natural low-rank approximations, and ratios of block banded matrices (which are in general full matrices), and, of course, systems derived from linearly couples subsystems.

3.7.1 Solving Semi-Separable Systems with QR: The URV Method

The goal of an URV factorization is a little more ambitious than the QR factorization presented as the beginning. As we saw, the factorization only works well when T is non-singular, otherwise we end up with an R factor that is not strictly triangular but only upper and in so called "echelon" or staircase form. Clearly, row operations are not sufficient. To remedy the situation, one also needs column operations that reduce the staircase form to purely triangular. This is in a nutshell the URV factorization, in which U is a set of columns of an orthogonal matrix, V a set of rows of another, and R is strictly upper triangular and invertible. When $T = URV$ and T is invertible, then U and V will be unitary, and $T^{-1} = V'R^{-1}U'$. However, when T is general, then the solution of the least squares solution for $y = Tu$ is given by $u = T^{\dagger}y$ with $T^{\dagger} = V'R^{-1}U'$ (the same would be true for $y = uT$, now with $u = yV'R^{-1}U'$!) $T^{\dagger}$ is called the "Moore-Penrose inverse" of T.

The URV recursion would start with orthogonal operations on (block) columns, transforming the mixed matrix T to the upper form – actually one may alternate (block) column with (block) row operations to achieve a one pass solution. However, the block column operations are completely independent from the row operations, hence we can treat them first and then complete with row operations. We assume a semi-separable representation for T where the lower and upper parts use different state space realizations (all matrices shown are block diagonal and consisting typically of blocks of low dimensions):

$$T = C_\ell Z'(I - A_\ell Z')^{-1} B_\ell + D + C_u Z(I - A_u Z)^{-1} B_u \tag{3.56}$$

This corresponds to a "model of computation" shown in Fig. 3.10. The URV factorization starts with getting rid of the lower or anticausal part in T by post-multiplication with a unitary matrix, like in the traditional LQ factorization, but

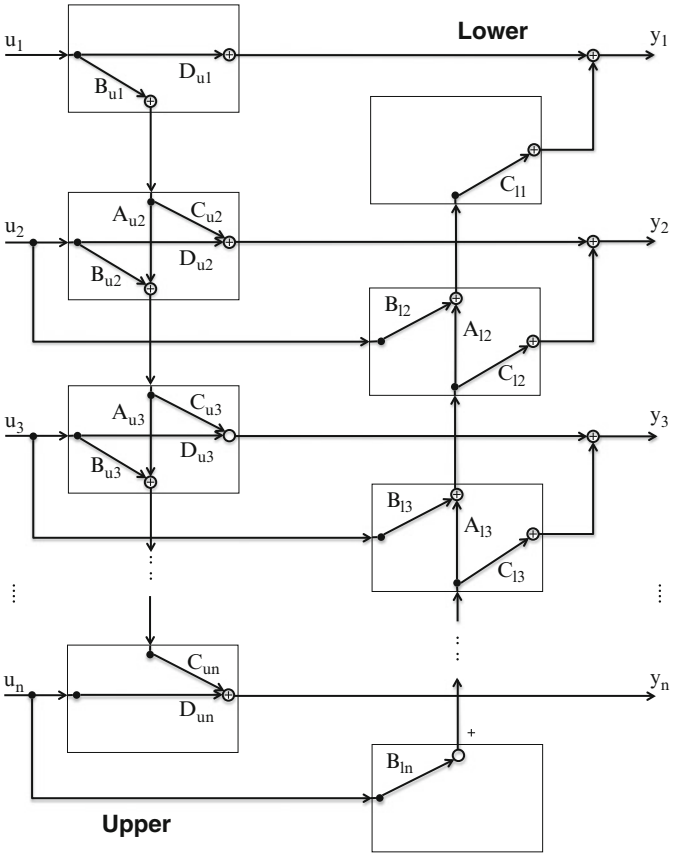


Fig. 3.10 The semi-separable model of computation

now working on the semi separable representation instead of on the original data. If one takes the lower part in input normal form, i.e. $\hat{C}_\ell Z'(I - \hat{A}_\ell Z')^{-1}\hat{B}_\ell = C_\ell Z'(I - A_\ell Z')^{-1}B_\ell$ such that $\hat{A}_\ell\hat{A}'_\ell + \hat{B}_\ell\hat{B}'_\ell = I$, then the realization for (upper) V is given by

$$V \approx \begin{bmatrix} \hat{A}_\ell & \hat{B}_\ell \\ C_V & D_V \end{bmatrix} \tag{3.57}$$

where C_V and D_V are formed by unitary completion of the isometric $\begin{bmatrix} \hat{A}_\ell & \hat{B}_\ell \end{bmatrix}$ (for an approach familiar to numerical analysts see [4].) V is a minimal causal unitary operator, which pushes T to upper: $\begin{bmatrix} T_u & 0 \end{bmatrix} := TV$ can be checked to be upper (we shall do so further on where we show the validity of the operation) and a realization for T_u follows from the preceding as

$$T_u \approx \left[\begin{array}{cc|c} \hat{A}'_\ell & 0 & C'_V \\ B_u\hat{B}'_\ell & A_u & B_uD'_V \\ \hline \hat{C}_\ell\hat{A}'_\ell + D\hat{B}'_\ell & C_u & \hat{C}_\ell\hat{C}'_V + DD'_V \end{array}\right]. \tag{3.58}$$

As expected, the new transition matrix combines lower and upper parts and has become bigger, but T_u is now upper. Numerically, this step is executed as an LQ factorization as follows. Let $x_k = R_k\hat{x}_k$ and let us assume we know R_k at step k, then

$$\begin{bmatrix} A_{\ell,k}R_k & B_{\ell,k} \\ C_{\ell,k}R_k & D_k \end{bmatrix} = \begin{bmatrix} R_{k+1} & 0 & 0 \\ \hat{C}_{\ell,k}\hat{A}'_{\ell,k} + D_k\hat{B}'_{\ell,k} & \hat{C}_{\ell,k}\hat{C}'_{V,k} + D_kD'_{V,k} & 0 \end{bmatrix} \begin{bmatrix} \hat{A}_{\ell,k} & \hat{B}_{\ell,k} \\ C_{V,k} & D_{V,k} \end{bmatrix} \tag{3.59}$$

The LQ factorization of the left handed matrix computes everything that is needed, the transformation matrix, the data for the upper factor T_u and the new state transition matrix R_{k+1}, all in terms of the original data. Because we have not assumed T to be invertible, we have to allow for an LQ factorization that produces an echelon form rather than a strictly lower triangular form, and allows for a kernel as well, represented by a block column of zeros.

The next step is what is called an inner/outer factorization on the upper operator T_u to produce an upper and upper invertible operator T_o and an upper orthogonal operator U such that $T_u = UT_o$. The idea is to find an as large as possible upper and orthogonal operator U such that $U'T_u$ is still upper – U' tries to push T_u back to lower, when it does so as much as possible, an upper and upper invertible factor T_o should result. There is a difficulty here that T_u might not be invertible, already in the original QR case one may end up with an embedded R_u matrix. This difficulty is not hard to surmount, but in order to avoid a too technical discussion, we just assume invertibility at this point and we shall see that the procedure actually produces the general formula needed. If the entries of T_u would be scalar, then we would already have reached our goal. However, the operation of transforming T to a block upper matrix T_u will destroy the scalar property of the entries, and the inverse of T_u may now have a lower part, which will be captured by the inner operator U that we shall now determine.

When $T_u = UT_o$ with U upper and orthogonal, then we also have $T_o = U'T_u$. Writing out the factorization in terms of the realization, and redefining for brevity $T_u := D + CZ(I - AZ)^{-1}B$ we obtain

$$\begin{aligned} T_o &= \left[D'_U + B'_U(I - Z'A'_U)^{-1}Z'C'_U\right]\left[D + CZ(I - AZ)^{-1}B\right] \\ &= D'_U D + B'_U(I - Z'A'_U)^{-1}Z'C'_U D + D'_U CZ(I - AZ)^{-1}B \\ &\quad + B'_U\{(I - Z'A'_U)^{-1}Z'C'_U CZ(I - AZ)^{-1}\}B \end{aligned} \tag{3.60}$$

This expression has the form: "direct term" + "strictly lower term" + "strictly upper term" + "mixed product". The last term has what is called 'dichotomy', what stands between $\{\cdot\}$ can again be split in three terms:

$$(I - Z'A'_U)^{-1}Z'C'_U CZ(I - AZ)^{-1} = (I - Z'A'_U)^{-1}Z'A'_U Y + Y + YAZ(I - AZ)^{-1} \tag{3.61}$$

with Y satisfying the "Lyapunov-Stein equation"

$$ZYZ' = C'_U C + A'_U YA \tag{3.62}$$

or, with indices: $Y_{k+1} = C'_{U,k}C_k + A'_{U,k}Y_k A_k$. The resulting strictly lower term has to be annihilated, hence we require $C'_U D + A'_U YB = 0$, in fact U should be chosen maximal with respect to this property (beware: Y depends on U!) Once these two equations are satisfied, the realization for T_o results as $T_o = (D'_U D + B'_U YB) + (D'_U C + B'_U YA)Z(I - AZ)^{-1}B$ – we see that T_o inherits A and B from T and gets new values for the other constituents C_o and D_o. Putting everything together in one matrix equation and in a somewhat special order, we obtain

$$\begin{bmatrix} YB & YA \\ D & C \end{bmatrix} = \begin{bmatrix} B_U & A_U \\ D_U & C_U \end{bmatrix}\begin{bmatrix} D_o & C_o \\ 0 & ZYZ' \end{bmatrix}. \tag{3.63}$$

Let us interpret this result without going into motivating theory (as in done in [4,6]). We have a pure QR factorization of the left hand side. At stage k one must assume knowledge of Y_k, and then perform a regular QR factorization of $\begin{bmatrix} Y_k B_k & Y_k A_k \\ D_k & C_k \end{bmatrix}$. $D_{o,k}$ will be an invertible, upper triangular matrix, so its dimensions are fixed by the row dimension of Y_k. The remainder of the factorization produces $C_{o,k}$ and Y_{k+1}, and, of course, the Q factor that gives a complete realization of U_k. What if T is actually singular? It turns out that then the QR factorization will produce just an upper staircase form with a number of zero rows. The precise result is

$$\begin{bmatrix} Y_k B_k & Y_k A_k \\ D_k & C_k \end{bmatrix} = \begin{bmatrix} B_{U,k} & A_{U,k} & B_{W,k} \\ D_{U,k} & C_{U,k} & D_{W,k} \end{bmatrix}\begin{bmatrix} D_{o,k} & C_{o,k} \\ 0 & Y_{k+1} \\ 0 & 0 \end{bmatrix}, \tag{3.64}$$

in which the extra columns represented by B_W and D_W define an isometric operator $W = D_W + C_W Z(I - A_W Z)^{-1} B_W$ so that

$$T_u = \begin{bmatrix} U & W \end{bmatrix} \begin{bmatrix} T_o \\ 0 \end{bmatrix}. \tag{3.65}$$

In other words, W characterizes the row kernel of T.

Remarkably, the operations work on the rows of T_u in ascending index order, just as the earlier factorization worked in ascending index order on the columns. That means that the URV algorithm can be executed completely in ascending index order. The reader may wonder at this point (1) how to start the recursion and (2) whether the proposed algorithm is numerically stable. On the first point and with our convention of empty matrices, there is no problem starting out at the upper left corner of the matrix, both A_1 and Y_0 are just empty, the first QR is done on $\begin{bmatrix} D_1 & C_1 \end{bmatrix}$. In case the original system does not start at index 1, but has a system part that runs from $-\infty$ onwards, then one must introduce knowledge of the initial condition on Y. This is provided, e.g., by an analysis of the LTI system running from $-\infty$ to 0 if that is indeed the case, see [7] for more details. On the matter of numerical stability, we offer two remarks. First, propagating Y_k is numerically stable, one can show that a perturbation on any Y_k will die out exponentially if the propagating system is assumed exponentially stable. Second, one can show that the transition matrix Δ of the inverse of the outer part will be exponentially stable as well, when certain conditions on the original system are satisfied [6].

To obtain the Moore-Penrose inverse (or the actual inverse when T is invertible) one only needs to specify the inverse of T_o, as the inverses of U and V are already known (they are just U' and V' with primed realizations as well.) By straightforward elimination, and with the knowledge that T_o is upper invertible, we find with $\Delta = A - BD_o^{-1}C_o$,

$$T_o^{-1} = D_o^{-1} - D_o^{-1}C_o Z(I - \Delta Z)^{-1} B D_o^{-1}. \tag{3.66}$$

One does not need to compute these matrices, the inverse filter can easily be realized directly on the realization of T_u by arrow reversal, as shown in Fig. 3.11.

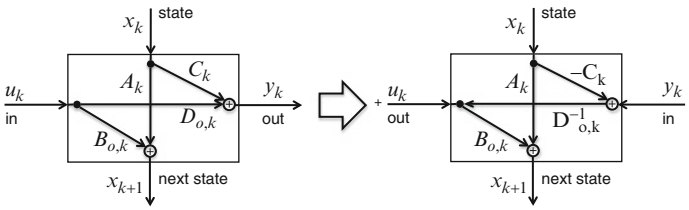


Fig. 3.11 Realization of the inverse of an outer filter in terms of the original

With a bit of good will, one recognizes the square root algorithm for the Kalman filter as a (very) special case of inner-outer factorization. Now, we actually have the dual case, due to the fact that the anti-causal part first had to be eliminated, forcing the inner-outer factorization on the rows rather than on the columns (we would have found the exact same formula as for the Kalman filter, if we had started with an upper matrix and then had done an outer-inner rather than an inner-outer factorization). There is an alternative to the URV algorithm presented here, namely working first on the rows and then on the columns, but that would necessitate a descending rather than an ascending recursion.

3.8 Concluding Remarks

The interplay of signal processing applications, numerical linear algebra algorithms and real-time computing architectures is a fascinating cross-road of interdisciplinary research, which is under constant change due to technological progress in all associated fields. Besides all the research aspects this is also an essential aspect for designing engineering curricula – students need to learn about this interplay, to command a good understanding of the associated domains. This understanding is essential for them to successfully design real-time computer systems implementing state of the art products in terms of highly integrated embedded systems.

We discussed the QR decomposition of a coefficient matrix as a versatile computational tool that is central to many such algorithms because of its superior numerical properties matching the requirements for implementation on parallel real-time architectures. Besides its preferable properties, the QR decomposition provides for an elegant computational framework that allows for an improved understanding of many related concepts, including Kalman filtering and time-varying system theory.

Many of the mentioned arguments and features of algorithms for solving linear systems have been investigated and discussed in the context of algorithm specific systolic VLSI arrays for signal processing applications [27]. Even though the hey-days of systolic arrays are gone, the topics are still relevant, since programmable Graphical Processing Units (GPU) are relevant parallel data processing targets for implementing number crunching algorithms in real-time computer systems [3, 10, 26].

Fragment shaders of programmable GPUs provide for a high performance parallel computing platform, but algorithms have to be able to fully exploit the performance offered by GPUs. Besides the costs for arithmetic operations and static memory designers need to consider the cost associated with data transport (memory bandwidth) as another important design criterium. Iterative matrix solvers for large-scale sparse matrices are attractive for many large-scale computational problems running on mainframe computers, however, for real-time applications running on embedded systems it may in fact be more interesting to use other algebraic means to exploit the structure of matrices. The advent of concepts to

exploit the semi-separable or hierarchically semi-separable matrix structure holds new promise for attacking large-scale computational problems using embedded real-time computer systems.

References

1. Ahmed HM, Delosme J-M, Morf M (1982) Highly concurrent computing structures for matrix arithmetic and signal processing. IEEE Comp 15(1):65–86
2. Björck A (1996) Numerical methods for least squares problems. SIAM, Philadelphia, PA
3. Bolz J, Farmer I, Grinspun E, Schröder P (2003) Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. In: SIGGRAPH 2003, ACM, New York, pp 917–924
4. Chandrasekaran S, Dewilde P, Gu M, Pals T, van der Veen A-J, Xia J (2002) A fast backward stable solver for sequentially semi-separable matrices, volume HiPC202 of Lecture notes in computer science, Springer, Berlin, pp 545–554
5. Dewilde P (1988) New algebraic methods for modelling large-scale integrated circuits. Int J Circ Theory Appl 16(4):473–503
6. Dewilde P, van der Veen A-J (1998) Time-varying systems and computations. Kluwer, Dordrecht
7. Dewilde P, van der Veen A-J (2000) Inner-outer factorization and the inversion of locally finite systems of equations. Linear Algebra Appl 313:53–100
8. Dewilde P, Diepold K, Bamberger W (2004a) Optic flow computation and time-varying system theory. In: Proceedings of the international symposium on mathematical theory of networks and systems (MTNS). Katholieke Universiteit Leuven, Belgium
9. Dewilde P, Diepold K, Bamberger W (2004b) A semi-separable approach to a tridiagonal hierarchy of matrices with applications to image analysis. In: Proceedings of the international symposium on mathematical theory of networks and systems (MTNS). Katholieke Universiteit Leuven, Belgium
10. Durkovic M, Zwick M, Obermeier F, Diepold K (2006) Performance of optical flow techniques on graphics hardware. In: IEEE international conference on multimedia and expo (ICME)
11. Fadeeva VN (1959) Computational methods of linear algebra. Dover, New York
12. Gastona FMF, Irwina GW (1989) Systolic approach to square root information kalman filtering. Int J Control 50(1):225–248
13. Gohberg I, Kailath T, Koltracht I (1985) Linear complexity algorithms for semiseparable matrices. Integral Equations Operator Theory 8:780–804
14. Golub G, van Loan Ch (1989) Matrix computations. John Hopkins University Press, Baltimore, MD
15. Götze J, Schwiegelshohn U (1988) Sparse matrix-vector multiplication on a systolic array. In: International Conference on Acoustics, Speech, and Signal Processing (ICASSP), vol 4. IEEE, pp 2061–2064
16. Hartley RI (1997) In defence of the 8-point algorithm. IEEE Trans Pattern Anal Machine Intell 19(6):580–593
17. Jainandunsing K, Deprettere EF (1989a) A new class of parallel algorithms for solving systems of linear equations. SIAM J Scientific Comput 10(5):880–912
18. Jainandunsing K, Deprettere EdF (1989b) A new class of parallel algorithms for solving, systems of linear equations. SIAM J Sct Stat Comput 10(5):880–912
19. Jean-Marc D, Ipsen Ilse C.F. (1986) Parallel solution of symmetric positive definite systems with hyperbolic rotations. Linear Algebra Appl 77:75–111.
20. Jichun Bu, Deprettere F, Dewilde P (1990) A design methodology for fixed-size systolic arrays. In: Proceedings of the international conference on application specific array processors

21. Kailath T (1981) Lectures on Wiener and Kalman Filtering. CISM Courses and Lectures No. 140, Springer, Wien, New York
22. Kailath T, Sayed A (1999) Fast reliable algorithms for matrices with structure. SIAM, Philadelphia, PA
23. Kailath T, Sayed A, Hasibi B (2000) Linear esimtation. Prentice Hall, Upper Saddle River, NJ
24. Kienhuis B e.a. (2010) Hotspot parallelizer. Compaan Design. http://www.compaandesign.com/technology/overview, visitied on Nov.14.2011.
25. Kronecker L (1890) Algebraische Reduction der schaaren bilinearer Formen. S.B. Akad. Berlin, pp 663–776
26. Krüger J, Westermann R (2005) Linear algebra operators for gpu implementation of numerical algorithms. In: SIGGRAPH 2005. ACM, New York
27. Kung SY (1988) VLSI array processors. Prentice Hall, Englewood Cliffs, NJ
28. Larsson D, Schinner e.a. P (1986) The CADMUS 9230 ICD Graphic Workstation 1, volume The integrated design handbook, chapter 8, Delft University Press, Delft, pp 8.1–8.29
29. Lucas BD, Kanade T (1981) An iterative image registration technique with an application to stereo vision. In: Proceedings of imaging understanding workshop, pp 121–130
30. Misraa M, Nassimib D, Prasannaa VK (1993) Efficient vlsi implementation of iterative solutions to sparse linear systems. Parallel Comput 19(5):525–544
31. Nash JG, Hansen S (1988) Modified faddeeva algorithm for concurrent execution of linear algebraic operations. IEEE Trans Comp 37(2):129–137
32. Paige ChC, Saunders MA (1982) LSQR: An algorithm for sparse linear equations and sparse least squares. ACM Trans Math Softw 8:43–71
33. Polka LA, Kalyanam H, Hu G, Krishnamoorthy S (2007) Package technology to address the memory bandwidth challenge for tera-scale computing. Intel Technol J 11(3):197–206
34. Proudler IK, McWhirter JG, Shepherd TJ (1991) Computationally efficient qr decomposition approach to least squares adaptive filtering. IEE Proc F, Radar Signal Processing 138(4): 341–353
35. Schunck BG, Horn BKP (1981) Determining optical flow. Artif Intell 17(1-3):185–203
36. Saad Y (2003) Iterative methods for sparse linear systems. SIAM, Philadelphia, PA
37. Strang G (2007) Computational science and engineering. Wellesley-Cambridge Press, Wellesley, MA
38. Tong L, van der Veen A-J, Dewilde P (2002) A new decorrelating rake receiver for long-code wcdma. In: Proceedings 2002 conference on information sciences systems. Princeton University, NJ
39. Vanderbril R, van Barel M, Mastronardi N (2008) Matrix computations and semi-separable matrices. John Hopkins University Press, Baltimore, MD

Chapter 4
Interface-Based Design of Real-Time Systems

Nikolay Stoimenov, Samarjit Chakraborty, and Lothar Thiele

4.1 Introduction

Today most embedded systems consist of a collection of computation and communication components that are supplied by different vendors and assembled by a system manufacturer. Such a component-based design methodology is followed in several domains such as automotive, avionics, and consumer electronics. The system manufacturer responsible for component assembly has to take design decisions (related to the choice of components and how they are to be connected, e.g. using a bus or a network-on-chip) and perform system analysis. Such important analysis is usually related to verifying that buffers in components never overflow or underflow, end-to-end delays or worst-case traversal times (WCTTs) of data through a component network fall within given deadlines, and others.

Typically performance analysis methods are used for the analysis of a component-based real-time system design a posteriori. This means that a real-time system is designed and dimensioned in a first step, and only after completion of this first step, the performance analysis is applied to the system design in a second step. The analysis result will then give an answer to the binary question whether the system design that was developed in the first step meets all real-time requirements, or not. A designer must then go back to the first step, change the design, and iterate on the two steps until an appropriate system design is found.

N. Stoimenov
Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland
e-mail: stoimenov@tik.ee.ethz.ch

S. Chakraborty
Institute for Real-Time Computer Systems, TU Munich, 80290 Munich, Germany
e-mail: samarjit@tum.de

L. Thiele (✉)
Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland
e-mail: thiele@tik.ee.ethz.ch

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_4,

Unlike this two-step approach is the idea of *interface-based design* described by de Alfaro and Henzinger in [1, 2]. It proposes a holistic one-step approach toward design and analysis of systems where components have interfaces, and a designer can decide whether two components can be connected and work together based only on the information exposed in their interfaces. Interface-based design avoids modeling the internals of each component, which is often difficult because of the complexity of the components and their proprietary nature.

In interface-based design, components are described by component interfaces. A component interface models how a component can be used, which is in contrast to an abstract component that models what a component does. Through *input assumptions*, a component interface models the expectations that a component has from the other components in the system and the environment. Through *output guarantees*, a component interface tells the other components in the system and the environment what they can expect from this component. The major goal of a good component interface is then to provide only the appropriate information that is sufficient to decide whether two or more components can work together properly. In the context of component interfaces for real-time system performance analysis, the term "properly" refers to questions like: Does the composed system satisfy all requested real-time properties such as delay, buffer, and throughput constraints?

Consequently, in an interface-based real-time system design approach, the compliance to real-time constraints is checked at composition time. That is, the successful composition of a set of components and their interfaces to a complete system design already guarantees the satisfaction of all real-time constraints, and no further analysis steps are required. This leads to faster design processes and partly removes the need for the classical binary search iteration approach to find appropriate system designs.

Additionally, an interface-based real-time system design approach also benefits from the properties of incremental design and independent implementability that are elementary features of an interface-based design. The support for *incremental design* ensures that component interfaces can be composed one-by-one into subsystems in any order. And if at any step a component interface cannot be composed with a subsystem, this already excludes the possibility that the complete system can be composed successfully, and therefore can not work properly. *Refinement* on the other hand is very similar to subtyping of classes in object-oriented programming. A component interface can be refined by another component interface if it accepts at least all inputs of the original interface and produces only a subset of the original outputs. Fulfilling these constraints ensures that components with compatible interfaces can be refined independently and still remain compatible, thus supporting *independent implementability*.

In this chapter, we develop an *interface algebra* for verifying buffer overflow and underflow constraints, and estimating worst-case traversal times in order to verify their compliance with provided upper bounds, where the different components exchange data through first-in first-out (FIFO) buffers. Such architectures are common for streaming applications, e.g. audio/video processing and distributed

controllers where data flows from sensors to actuators while getting processed on multiple processors. Our proposed interface algebra is based on *Rate and Real-Time Interfaces* as proposed in [5, 9], respectively, and is motivated by the concept of *assume/guarantee* interfaces from [1]. In particular, we cast the *Real-Time Calculus* framework from [4,8] within the assume/guarantee interface setting. At a high level, two interfaces are *compatible* when the guarantees associated with one of them comply with the assumptions associated with the other. We describe how to compute the *assumptions* and *guarantees* associated with interfaces based on a *monotonicity* property. This result is then used to verify buffer underflow and overflow constraints, compute the WCTT in a component network incrementally, and validate that it complies to a given deadline. Our approach may be summarized in the following three steps:

1. Define an abstract component that describes the real-time properties of a concrete system hardware/software component. This involves defining proper abstractions for component inputs and outputs, and internal component relations that meaningfully relate abstract inputs to abstract outputs. Such components can be composed together to create a system model, and together with a model of the environment can be used to perform timing analysis. Several such abstract components are described in Sect. 4.2.
2. To derive the interface of an abstract component, we need to define interface variables as well as input and output predicates on these interface variables. In Sect. 4.3 we describe how one can do this for the abstract components introduced in Sect. 4.2.
3. Derive the internal interface relations that relate incoming guarantees and assumptions to outgoing guarantees and assumptions of a component's interfaces. This is also described in Sect. 4.3.

4.2 Timing Analysis of Component Networks

In this section, we describe a timing analysis framework in particular, for verifying buffer overflow and underflow constraints and computing worst-case traversal times for component networks. The framework is based on Real-Time Calculus [4, 8]. This calculus is an adaptation of Network Calculus [6] that was designed to analyze communication networks. Here, we consider three basic types of abstract components: Processing Element (PE), Playout Buffer (PB), and Earliest Deadline First (EDF) component. Component models for greedy shapers, time division multiple access, polling servers, and hierarchical scheduling are described in [10–12]. In Sect. 4.3 we will lift this framework to an interface-based design setting. First we need to introduce the basic models and abstractions that underlie Real-Time Calculus before we describe the three basic abstract components that we consider.

4.2.1 Basic Models

In the setting we study here, event streams are processed on a sequence of components. An event or data stream described by the cumulative function $R(t)$ enters the input buffer of the component and is processed by the component whose availability is described by the cumulative function $C(t)$. Formally, the cumulative functions $R(t) \in \mathbb{R}^{\geq 0}$ and $C(t) \in \mathbb{R}^{\geq 0}$ for $t \geq 0$ denote the number of events/data items that have been received or could be processed within the time interval $[0, t)$, respectively. After being processed, events are emitted on the component's output, resulting in an outgoing event stream $R'(t)$. The remaining resources that were not consumed are available for use and are described by an outgoing resource availability trace $C'(t)$. The relations between $R(t)$, $C(t)$, $R'(t)$ and $C'(t)$ depend on the component's processing semantics. For example, Greedy Processing (GP) denotes that events are always processed when there are resources available. Typically, the outgoing event stream $R'(t)$ will not equal the incoming event stream $R(t)$ as it may, for example, exhibit more or less jitter.

While cumulative functions such as $R(t)$ or $C(t)$ describe one concrete trace of an event stream or a resource availability, *variability characterization curves* (VCCs) capture all possible traces using upper and lower bounds on their timing properties. The *arrival* and *service curves* from Network Calculus [6] are specific instances of VCCs and are more expressive than traditional event stream models such as the periodic, periodic with jitter, sporadic, etc. Arrival curves $\alpha^{\mathrm{l}}(\Delta)$ and $\alpha^{\mathrm{u}}(\Delta)$ denote the minimum and the maximum number of events that can arrive in *any* time interval of length Δ, i.e. $\alpha^{\mathrm{l}}(t-s) \leq R(t) - R(s) \leq \alpha^{\mathrm{u}}(t-s)$ for all $t > s \geq 0$. In addition, $\alpha^{\mathrm{l}}(0) = \alpha^{\mathrm{u}}(0) = 0$. We also denote the tuple $(\alpha^{\mathrm{l}}, \alpha^{\mathrm{u}})$ with α. Service curves characterize the variability in the service provided by a resource. The curves $\beta^{\mathrm{l}}(\Delta)$ and $\beta^{\mathrm{u}}(\Delta)$ denote the minimum and the maximum number of events that can be processed within *any* time interval of length Δ, i.e. $\beta^{\mathrm{l}}(t-s) \leq C(t) - C(s) \leq \beta^{\mathrm{u}}(t-s)$ for all $t > s \geq 0$. In addition, $\beta^{\mathrm{l}}(0) = \beta^{\mathrm{u}}(0) = 0$. We also denote the tuple $(\beta^{\mathrm{l}}, \beta^{\mathrm{u}})$ with β. An event stream modeled by $\alpha(\Delta)$ enters a component and is processed using the resource modeled as $\beta^{\mathrm{l}}(\Delta)$. The output is again an event stream $\alpha'(\Delta)$, and the *remaining* resource is expressed as $\beta'^{\mathrm{l}}(\Delta)$. Note that the domain of the arrival and service curves are events, i.e. they describe the number of arriving events and the capability to process a certain number of events, respectively. The generalization towards physical quantities such as processing cycles or communication bits can be done by means of *workload curves* which is another instance of a VCC, for details refer to [7].

4.2.2 Processing Element

The PE component can be used to model a single processing element which processes one input stream. However, it can also be composed with other components of

the same type, and model components processing more than one input stream using a fixed priority (FP) scheduling. Consider a concrete GP component that is triggered by the events of an incoming event stream. A fully preemptive task is instantiated at every event arrival to process the incoming event, and active tasks are processed in a FIFO order, while being restricted by the availability of resources. The completion of each task execution results in the corresponding event being removed from the input buffer and an event being emitted on the outgoing event stream.

Following results from Real-Time and Network Calculus [4, 6], such a component can be modeled by an abstract component *PE* with the following internal component relations[1]:

$$\alpha'^{u}(\Delta) = (\alpha^{u} \oslash \beta^{l})(\Delta), \tag{4.1}$$

$$\alpha'^{l}(\Delta) = (\alpha^{l} \otimes \beta^{l})(\Delta), \tag{4.2}$$

$$\beta'^{l}(\Delta) = \sup_{0 \le \lambda \le \Delta} \left\{ \beta^{l}(\lambda) - \alpha^{u}(\lambda) \right\} := RT(\beta^{l}, \alpha^{u}), \tag{4.3}$$

and the backlog of the input buffer is bounded by:

$$\sup_{0 \le \lambda \le \Delta} \left\{ \alpha^{u}(\lambda) - \beta^{l}(\lambda) \right\}.$$

If the available buffer space in the input buffer is constrained by $b_{\max}$, the backlog should never become bigger than the buffer size, i.e. we have a *buffer overflow constraint*. In this case, we can obtain the following component-based constraint on the admissible arrival and service curves:

$$\alpha^{u}(\Delta) \le \beta^{l}(\Delta) + b_{\max}, \quad \forall \Delta \in \mathbb{R}^{\ge 0}. \tag{4.4}$$

If the input arrival and service curves satisfy the above constraint, the backlog will never be bigger than $b_{\max}$. Before continuing with the computation of the WCTT for a PE component, we need to define the following shift function:

$$r(\alpha, c, \Delta) = \begin{cases} \alpha(\Delta - c) & \text{if } (\Delta > c) \wedge (\Delta \neq 0) \\ 0 & \text{if } (\Delta \le c) \vee (\Delta = 0) \end{cases} \tag{4.5}$$

which simply shifts a given curve $\alpha(\Delta)$ by the amount c to "the right".

The WCTT experienced by an event in the component, defined as its finishing time minus its arrival time, can be computed as:

$$\mathrm{Del}(\alpha^{u}, \beta^{l}) := \sup_{\lambda \ge 0} \left\{ \inf\{\tau \ge 0 : \alpha^{u}(\lambda) \le \beta^{l}(\lambda + \tau)\} \right\}.$$

[1] See the Appendix at the end of the chapter for definitions of the operators $\otimes$ and $\oslash$.

If all events of the input stream must be processed by a PE component within a relative deadline D, then for the stream to be schedulable we must have that $\mathrm{Del}(\alpha^{\mathrm{u}}, \beta^{\mathrm{l}}) \leq D$ which can be written as:

$$\beta^{\mathrm{l}}(\Delta) \geq r(\alpha^{\mathrm{u}}, D, \Delta), \quad \forall \Delta \in \mathbb{R}^{\geq 0}.$$

The above inequality gives us an expression for the minimum service in component PE that is required in order to meet a deadline constraint.

It is also possible to have systems with processing elements that process more than one data stream. For this purpose, the remaining service output of a higher priority PE component, computed with (4.3), can be connected to the service input of a lower priority PE component. This way we can model an FP resource sharing between PE components.

4.2.3 Playout Buffer

The PB component models a playout buffer. It receives data and stores it in a buffer which is read at a constant (usually periodic) rate. The buffer has a maximum size $B_{\max}$. We make the assumption that at the start of the system, there are already B_0 initial data items in the playout buffer, e.g. due to a playback delay. Data items in the playout buffer are removed at a constant rate. In particular, $P(t)$ data items are removed within the time interval $[0, t)$. This behavior can be described by the readout VCC $\rho(\Delta) = (\rho^{\mathrm{l}}(\Delta), \rho^{\mathrm{u}}(\Delta))$, i.e. $\rho^{\mathrm{l}}(t-s) \leq P(t) - P(s) \leq \rho^{\mathrm{u}}(t-s)$ for all $t > s \geq 0$. What needs to be guaranteed is that the playout buffer never *overflows or underflows*. Following results from Real-Time and Rate Interfaces [5], for a PB component with input and readout event streams characterized by the VCCs α and ρ, respectively, and B_0 initial events, the playout buffer size $B(t)$ is constrained by $0 \leq B(t) \leq B_{\max}$ at all times if the following constraints are satisfied:

$$\alpha^{\mathrm{l}}(\Delta) \geq \rho^{\mathrm{u}}(\Delta) - B_0, \quad \forall \Delta \in \mathbb{R}^{\geq 0}, \tag{4.6}$$

$$\alpha^{\mathrm{u}}(\Delta) \leq \rho^{\mathrm{l}}(\Delta) + B_{\max} - B_0, \quad \forall \Delta \in \mathbb{R}^{\geq 0}. \tag{4.7}$$

These are component-wise constraints that guarantee for a PB component to never overflow or underflow.

The WCTT experienced by an event in the component can be computed as $\mathrm{Del}(\alpha^{\mathrm{u}}, \rho^{\mathrm{l}}_{\tau})$ where

$$\rho^{\mathrm{l}}_{\tau}(\Delta) = r(\rho^{\mathrm{l}}, \tau, \Delta) \tag{4.8}$$

is the lower readout curve "shifted to the right" by the initial playback delay $\tau \geq 0$ necessary to accumulate B_0 events. Similarly to a PE component, meeting a relative deadline constraint of D in a PB component would require for the input stream that we have:

$$\rho^{\mathrm{l}}_{\tau}(\Delta) \geq r(\alpha^{\mathrm{u}}, D, \Delta), \quad \forall \Delta \in \mathbb{R}^{\geq 0},$$

where r is the shift function defined in (4.5).

4.2.4 Earliest Deadline First Component

The EDF component is similar to the PE component but it models processing of several data streams with a resource shared using the earliest deadline first scheduling policy. This requires a new abstract component with different internal relations [10]. Such a component processes N input event streams and emits N output event streams. Each input event stream i, $1 \leq i \leq N$, is associated with a fully preemptive task which is activated repeatedly by incoming events. Each input event stream i has an associated FIFO buffer with maximum size $b_{i\,\max}$ where events are backlogged. Tasks process the head events in these buffers and are scheduled in an EDF order. Each task has a best-case execution time of BCET_i, a worst-case execution time WCET_i, and a relative deadline D_i where $0 \leq \mathrm{BCET}_i \leq \mathrm{WCET}_i \leq D_i$. The completion of a task execution results in the corresponding input event being removed from the associated buffer and an output event being emitted on the associated output event stream.

For an EDF component with a service curve β and event streams characterized by arrival curves α_i, all tasks are processed within their deadlines if and only if:

$$\beta^{\mathrm{l}}(\Delta) \geq \sum_{i=1}^{N} r(\alpha_i^{\mathrm{u}}, D_i, \Delta), \quad \forall \Delta \in \mathbb{R}^{\geq 0}, \tag{4.9}$$

using the shift function r from (4.5). The output streams can be characterized by arrival curves computed for all streams i as:

$$\alpha'^{\mathrm{u}}_i(\Delta) = r(\alpha_i^{\mathrm{u}}, -(D_i - \mathrm{BCET}_i), \Delta), \tag{4.10}$$

$$\alpha'^{\mathrm{l}}_i(\Delta) = r(\alpha_i^{\mathrm{l}}, (D_i - \mathrm{BCET}_i), \Delta), \tag{4.11}$$

and the number of events in input buffers do not exceed their capacity $b_{i\,\max}$ if:

$$\alpha_i^{\mathrm{u}}(D_i) \leq b_{i\,\max}, \quad \forall i. \tag{4.12}$$

The EDF component schedulability condition (4.9) can be related to the demand bound functions described by Baruah et al. in [3]. Given that this condition is satisfied, the service curve provided to each stream can be modeled with a burst-delay function as defined in [6] which is computed for each stream i as:

$$\beta^{\mathrm{l}}_{D_i}(\Delta) = \begin{cases} +\infty & \text{if } \Delta > D_i \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

The WCTT experienced by an event from stream i can be computed as $\mathrm{Del}(\alpha_i^{\mathrm{u}}, \beta^{\mathrm{l}}_{D_i})$ which is upper bounded by D_i.

4.2.5 Worst-Case Traversal Times of Component Networks

The worst-case traversal time for an event from an input stream which is processed by a sequence of components can be computed as the sum of the worst-case traversal times of the individual components. However, this would lead to a very pessimistic and unrealistic result as it would assume that the worst-case traversal times occur in all components for the same event. A better bound on the worst-case traversal time can be achieved by considering a concatenation of the components. This is a phenomenon known as "pay bursts only once" [6]. Following results from Network Calculus, this leads to the following computation for the WCTT.

For an input event stream α traversing a sequence of components which consists of a set of PEs, a set of PBs, and a set of EDF components denoted as $\mathscr{PE}$, $\mathscr{PB}$ and $\mathscr{EDF}$, respectively, the worst-case traversal time that an event can experience can be computed as $\mathrm{Del}(\alpha^{\mathrm{u}}, \beta_{\mathscr{PE}} \otimes \rho_{\mathscr{PB}} \otimes \beta_{\mathscr{EDF}})$ with $\beta_{\mathscr{PE}} = \bigotimes_{c \in \mathscr{PE}} \beta^{\mathrm{l}}_c$, $\rho_{\mathscr{PB}} = \bigotimes_{c \in \mathscr{PB}} \rho^{\mathrm{l}}_{\tau\, c}$, and $\beta_{\mathscr{EDF}} = \bigotimes_{c \in \mathscr{EDF}} \beta^{\mathrm{l}}_{D_i\, c}$, where β^{l}_c is the service availability of PE component c, $\rho^{\mathrm{l}}_{\tau\, c}$ is the lower readout curve for PB component c as defined with (4.8), and $\beta^{\mathrm{l}}_{D_i\, c}$ is the service availability provided to the stream served with relative deadline D_i by EDF component c as defined with (4.13). A WCTT constraint on the sequence of components $\mathrm{Del}(\alpha^{\mathrm{u}}, \beta_{\mathscr{PE}} \otimes \rho_{\mathscr{PB}} \otimes \beta_{\mathscr{EDF}}) \leq D$ can be written as follows:

$$\beta_{\mathscr{PE}} \otimes \rho_{\mathscr{PB}} \otimes \beta_{\mathscr{EDF}} \geq r(\alpha^{\mathrm{u}}, D, \Delta), \quad \forall \Delta \in \mathbb{R}^{\geq 0}, \tag{4.14}$$

using the shift function r from (4.5).

4.3 Interface Algebra

In this section, we develop an interface-based design approach which will allow us by only inspecting the interfaces of two components to check whether WCTT and buffer underflow/overflow constraints would be satisfied if the components are composed together. The proposed interface algebra includes concepts from Real-Time Calculus, Assume/Guarantee Interfaces [1], and constraint propagation.

In our setup each component has two disjoint sets of input and output ports I and J. The actual input and output values of an abstract component are VCC curves. A connection from output j of one component to the input i of some other component will be denoted by (j, i). The interface of a component makes certain *assumptions* on I, which are specified using the predicate $\phi^I(I)$. Provided this predicate is satisfied, the interface *guarantees* that the component works correctly and its outputs will satisfy a predicate $\phi^O(J)$. Here, working correctly means that the component satisfies all real-time constraints such as buffer underflow/overflow or delay constraints [9].

In order to simplify the presentation, we introduce the *complies to* relation $\vdash$ between two VCC curves $a(\Delta)$ and $b(\Delta)$ as follows:

$$a \vdash b = (\forall \Delta : (a^{\mathrm{l}}(\Delta) \geq b^{\mathrm{l}}(\Delta)) \wedge (a^{\mathrm{u}}(\Delta) \leq b^{\mathrm{u}}(\Delta))).$$

In other words, a complies to b ($a \vdash b$) if for all values of Δ the interval $[a^{\mathrm{l}}(\Delta), a^{\mathrm{u}}(\Delta)]$ is enclosed by $[b^{\mathrm{l}}(\Delta), b^{\mathrm{u}}(\Delta)]$.

In the following, we will just use α to denote the characterization of *any* VCC that is an input or an output of an abstract component.

Following the introduced notation, for any VCC α, we can define the input and output predicates for some component input i and output j as $\phi_i^I(\alpha_i) = (\alpha_i \vdash \alpha_i^{\mathrm{A}})$ and $\phi_j^O(\alpha_j) = (\alpha_j \vdash \alpha_j^{\mathrm{G}})$, respectively, where α^{A} and α^{G} are assume and guarantee curves provided by the component interface.

We would like to have that if the input predicates of a component are all satisfied, then it works correctly and all output predicates are satisfied. In other words the condition $\bigwedge_{\forall i \in I} \phi_i^I(\alpha_i) \Rightarrow \bigwedge_{\forall j \in J} \phi_j^O(\alpha_j)$ must be satisfied by the interfaces of all components.

If we now connect several components, we want to be able to check if the whole system can work correctly by just checking whether their interfaces are compatible. This can be done by testing whether the relation $\bigwedge_{\forall (j,i)} \phi_j^O(\alpha_j) \Rightarrow \bigwedge_{\forall (j,i)} \phi_i^I(\alpha_i)$ is satisfiable. In other words, we must check if there exists *some* environment in which the components can be composed. The relation is hence the weakest precondition on the environment of the system.

We also need to propagate information about the predicates between the interfaces, see [9]. This way, we combine interface theory with constraint propagation, which enables parameterized design of component-based systems. We propagate the assume and guarantee curves of the input and output predicates through the interfaces. Each interface connection would have both assume and guarantee curves propagated in opposite directions as shown in Fig. 4.1. We connect the interfaces, i.e. the corresponding guarantee and assume curves, as $\forall (j,i) : \quad (\alpha^{\mathrm{G}}(i) = \alpha^{\mathrm{G}}(j)) \wedge (\alpha^{\mathrm{A}}(j) = \alpha^{\mathrm{A}}(i))$.

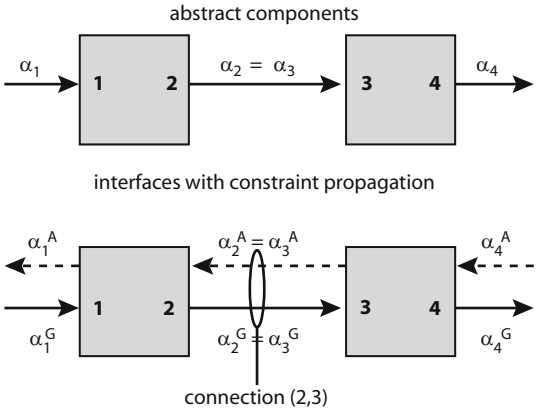


Fig. 4.1 Relation between input/output values of abstract components and assume/guarantee variables in interfaces with constraint propagation

Now, we can determine whether two abstract components are compatible by checking the compatibility of their interfaces. Let us suppose that the assume and guarantee variables of an interface of any component and their relation to the input and output values of the corresponding abstract component satisfy that:

$$(\forall i \in I : \alpha_i \vdash \alpha_i^{\mathrm{G}} \vdash \alpha_i^{\mathrm{A}}) \Rightarrow (\forall j \in J : \alpha_j \vdash \alpha_j^{\mathrm{G}} \vdash \alpha_j^{\mathrm{A}}), \tag{4.15}$$

where the component has inputs I and outputs J. Then if for a network of components, the relation $\alpha_i^{\mathrm{G}} \vdash \alpha_i^{\mathrm{A}}$ is satisfied for all inputs i, we can conclude that the system works correctly.

Now we need to develop the relations between guarantees and assumptions in order to satisfy (4.15) for every component. We will first describe a general method how these relations can be determined and then apply it to the abstract components described in Sect. 4.2.

To this end, as we are dealing with stateless interfaces, I and J can be related by a *transfer function*, e.g. $J = F(I)$. The actual function depends on the processing semantics of the modeled component.

We need to define the concept of a monotone abstract component. Note that the "complies to" relation $\vdash$ has been generalized to tuples, i.e. $(a_i : i \in I) \vdash (b_i : i \in I)$ equals $\forall i \in I : a_i \vdash b_i$.

Definition 4.1. An abstract component with a set of input and output ports, I and J, respectively, and a transfer function F that maps input curves to output curves, is *monotone* if $((\widetilde{\alpha}_i : i \in I) \vdash (\alpha_i : i \in I)) \Rightarrow ((\widetilde{\alpha}_j : j \in J) \vdash (\alpha_j : j \in J))$ where $(\alpha_j : j \in J) = F(\alpha_i : i \in I)$ and $(\widetilde{\alpha}_j : j \in J) = F(\widetilde{\alpha}_i : i \in I)$.

In other words, if we replace the input curves of an abstract component with curves that are compliant, then the new output curves are also compliant to the previous ones. Note that all components described in Sect. 4.2 satisfy this monotonicity condition, see for example the transfer functions (4.1), (4.2), (4.3), (4.10), and (4.11).

The following theorem leads to a constructive way to compute the input assumes and output guarantees from the given input guarantees and output assumes. We make use of the individual components of the transfer function F, i.e. $\alpha_j = F_j(\alpha_i : i \in I)$ for all $j \in J$ where I and J denote the input and output ports of the corresponding abstract component, respectively. The theorem establishes that we can simply determine the output guarantees using the components of a given transfer function of an abstract component. For the input assumes we need to determine inverses of the transfer function F_j with respect to at least one of its arguments. All arguments of some F_j are determined by the input guarantees but one, say for example α_{i*}^{G}. This one we replace by α_{i*}^{A} and try to determine this curve such that the result of the transfer function still complies to the given output assumes. If we choose the same i^* for several components of the output function, then the resulting α_{i*}^{A} needs to comply to all partial "inverses".

Theorem 4.1. *Given a monotone component with input ports I, output ports J, and a transfer function F that maps input curves to output curves, i.e.* $(\alpha_j : j \in J) = F(\alpha_i : i \in I)$.

Let us suppose that we determine the output guarantees using:

$$\alpha_j^{\mathrm{G}} = F_j(\alpha_i^{\mathrm{G}} : i \in I), \quad \forall j \in J, \tag{4.16}$$

and the input assumes are computed such that:

$$\forall j \in J \;\; \exists i^* \in I : \left(F_j(\alpha_i^{\mathrm{G}} : i \in I)\Big|_{\alpha_{i^*}^{\mathrm{G}} \leftarrow \alpha_{i^*}^{\mathrm{A}}} \;\vdash\; \alpha_j^{\mathrm{A}} \right), \tag{4.17}$$

where $\alpha_{i^*}^{\mathrm{G}} \leftarrow \alpha_{i^*}^{\mathrm{A}}$ denotes that in the preceding term $\alpha_{i^*}^{\mathrm{G}}$ is replaced by $\alpha_{i^*}^{\mathrm{A}}$.

Then (4.15) holds.

Proof. Let us assume that for all input ports $i \in I$ we have $\alpha_i \vdash \alpha_i^{\mathrm{G}}$, see (4.15). Using the monotonicity of F, we can now see that $(\forall i \in I : \alpha_i \vdash \alpha_i^{\mathrm{G}}) \Rightarrow F(\alpha_i : i \in I) \vdash F(\alpha_i^{\mathrm{G}} : i \in I) \Rightarrow (\forall j \in J : \alpha_j \vdash \alpha_j^{\mathrm{G}})$.

We still need to show that $(\forall i \in I : \alpha_i^{\mathrm{G}} \vdash \alpha_i^{\mathrm{A}}) \Rightarrow (\forall j \in J : \alpha_j^{\mathrm{G}} \vdash \alpha_j^{\mathrm{A}})$ using the construction in (4.16). At first note that this expression is equivalent to $\forall j \in J \;\; \exists i^* \in I : \left((\alpha_{i^*}^{\mathrm{G}} \vdash \alpha_{i^*}^{\mathrm{A}}) \Rightarrow (\alpha_j^{\mathrm{G}} \vdash \alpha_j^{\mathrm{A}}) \right)$. We also know that for any $i^* \in I$ we have $(\alpha_{i^*}^{\mathrm{G}} \vdash \alpha_{i^*}^{\mathrm{A}}) \Rightarrow ((\alpha_i^{\mathrm{G}} : i \in I) \vdash (\alpha_i^{\mathrm{G}} : i \in I)\,|_{\alpha_{i^*}^{\mathrm{G}} \leftarrow \alpha_{i^*}^{\mathrm{A}}})$.

Because of the monotonicity of F we can derive that for any $i^* \in I$ we have $(\alpha_{i^*}^{\mathrm{G}} \vdash \alpha_{i^*}^{\mathrm{A}}) \Rightarrow (F(\alpha_i^{\mathrm{G}} : i \in I) \vdash F(\alpha_i^{\mathrm{G}} : i \in I)\,|_{\alpha_{i^*}^{\mathrm{G}} \leftarrow \alpha_{i^*}^{\mathrm{A}}})$, and using (4.16) we find $\forall j \in J \;\; \exists i^* \in I$ such that $((\alpha_{i^*}^{\mathrm{G}} \vdash \alpha_{i^*}^{\mathrm{A}}) \Rightarrow (F_j(\alpha_i^{\mathrm{G}} : i \in I) \vdash F_j(\alpha_i^{\mathrm{G}} : i \in I)\,|_{\alpha_{i^*}^{\mathrm{G}} \leftarrow \alpha_{i^*}^{\mathrm{A}}}) \Rightarrow (\alpha_j^{\mathrm{G}} \vdash \alpha_j^{\mathrm{A}}))$. □

Next, we show how to compute the largest upper curve and smallest lower curve for which the respective relations still hold. This leads to the weakest possible input assumptions. We do this for the three types of components introduced in Sect. 4.2.

4.3.1 Processing Element

Now, using the relation between interface values, assumptions and guarantees in (4.15), and following the results from Theorem 4.1, we can deduce that the equations describing the output guarantees are equivalent to those for the abstract component, i.e. (4.1) and (4.2), but instead of using values, we use interface guarantees. Therefore, we have:

$$\alpha'^{\mathrm{uG}}(\Delta) = (\alpha^{\mathrm{uG}} \oslash \beta^{\mathrm{lG}})(\Delta),$$
$$\alpha'^{\mathrm{lG}}(\Delta) = (\alpha^{\mathrm{lG}} \otimes \beta^{\mathrm{lG}})(\Delta).$$

In order to calculate the input assumptions of the PE abstract component, we need to determine inverse relations corresponding to (4.1), (4.2), and (4.4). Following results from Network Calculus [6], we can do this by determining the pseudo-inverse functions which have the following definition $f^{-1}(x) = \inf\{t : f(t) \geq x\}$.

In order to guarantee that all relations hold if the input and output predicates are satisfied, we then need to use the minimum (in case of the upper curves) or the maximum (in case of the lower curves) of all the determined pseudo-inverses.

From the pseudo-inverses of (4.2), we get the inequalities $\alpha^{\mathrm{lA}} \geq \alpha'^{\mathrm{lA}} \oslash \beta^{\mathrm{lG}}$ and $\beta^{\mathrm{lA}} \geq \alpha'^{\mathrm{lA}} \oslash \alpha^{\mathrm{lG}}$. Here we use the duality relation between the $\oslash$ and $\otimes$ operators (see the Appendix). Similarly, from the pseudo-inverses of (4.1), we get the inequalities $\beta^{\mathrm{lA}} \geq \alpha^{\mathrm{uG}} \oslash \alpha'^{\mathrm{uA}}$ and $\alpha^{\mathrm{uA}} \leq \beta^{\mathrm{lG}} \otimes \alpha'^{\mathrm{uA}}$. Inverting the buffer overflow constraint (4.4) is trivial and we get the inequalities $\alpha^{\mathrm{uA}} \leq \beta^{\mathrm{lG}} + b_{\max}$ and $\beta^{\mathrm{lA}} \geq \alpha^{\mathrm{uG}} - b_{\max}$.

If a PE component shares the service it receives with other lower priority PE components, the remaining service is bounded by (4.3). In terms of output guaranteed values, this can be expressed as $\beta'^{\mathrm{lG}}(\Delta) = RT(\beta^{\mathrm{lG}}, \alpha^{\mathrm{uG}})$ where the RT operator is defined in (4.3). In order to obtain the input assumptions of a component using FP scheduling, we need to use the inverses of the RT operator (see the Appendix).

After combining all inverses, the assumptions related to component PE can be determined as follows:

$$\begin{aligned}
\alpha^{\mathrm{uA}} &= \min\left\{\beta^{\mathrm{lG}} \otimes \alpha'^{\mathrm{uA}}, \beta^{\mathrm{lG}} + b_{\max}, RT^{-\alpha}\left(\beta'^{\mathrm{lA}}, \beta^{\mathrm{lG}}\right)\right\}, \\
\alpha^{\mathrm{lA}} &= \alpha'^{\mathrm{lA}} \oslash \beta^{\mathrm{lG}}, \\
\beta^{\mathrm{lA}} &= \max\left\{\alpha'^{\mathrm{lA}} \oslash \alpha^{\mathrm{lG}}, \alpha^{\mathrm{uG}} \oslash \alpha'^{\mathrm{uA}}, \alpha^{\mathrm{uG}} - b_{\max}, RT^{-\beta}\left(\beta'^{\mathrm{lA}}, \alpha^{\mathrm{uG}}\right)\right\}. \qquad (4.18)
\end{aligned}$$

The interface connections for two PE components are illustrated in Fig. 4.2a.

4.3.2 Playout Buffer

For a PB component, the relations are simpler. We only need to determine the inverse relations for the buffer constraints (4.6) and (4.7), which directly yield the following relations:

$$\begin{aligned}
\alpha^{\mathrm{uA}} &= \rho^{\mathrm{lG}} + B_{\max} - B_0, \\
\alpha^{\mathrm{lA}} &= \rho^{\mathrm{uG}} - B_0, \\
\rho^{\mathrm{uA}} &= \alpha^{\mathrm{lG}} + B_0, \\
\rho^{\mathrm{lA}} &= \alpha^{\mathrm{uG}} - (B_{\max} - B_0). \qquad (4.19)
\end{aligned}$$

The interface connections for a single PB component are illustrated in Fig. 4.2b.

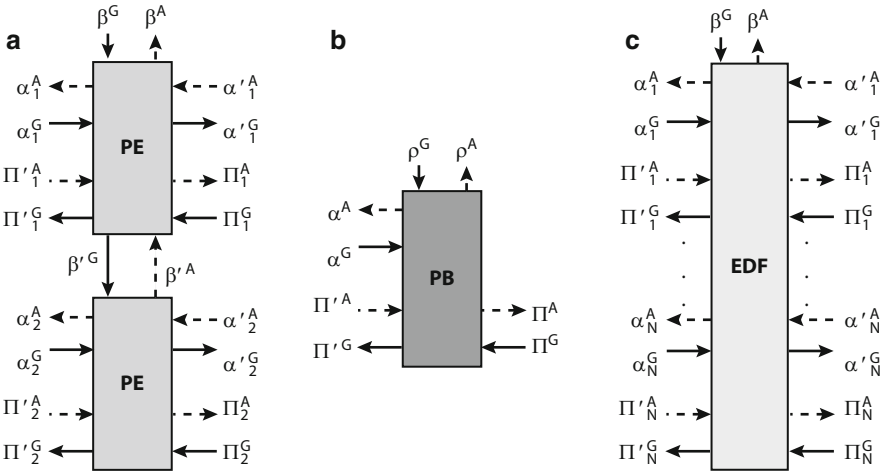


Fig. 4.2 Interface models for: (**a**) two PE components processing two streams with FP scheduling, (**b**) PB component, and (**c**) EDF component processing N streams

4.3.3 Earliest Deadline First Component

Similarly to the PE component, equations describing the output guarantees are again equivalent to those for the abstract component, i.e. (4.10) and (4.11). They only need to be expressed in terms of interface variables instead of values for all streams i:

$$\alpha'^{\mathrm{uG}}_i(\Delta) = r(\alpha^{\mathrm{uG}}_i, -(D_i - \mathrm{BCET}_i), \Delta),$$

$$\alpha'^{\mathrm{lG}}_i(\Delta) = r(\alpha^{\mathrm{lG}}_i, (D_i - \mathrm{BCET}_i), \Delta),$$

using the definition of the shift function r in (4.5).

Similarly, for the resource and buffer constraints, (4.9) and (4.12), we obtain:

$$\beta^{\mathrm{lG}}(\Delta) \geq \sum_{i=1}^{N} r(\alpha^{\mathrm{uG}}_i, D_i, \Delta), \quad \forall \Delta \in \mathbb{R}^{\geq 0},$$

$$\alpha^{\mathrm{uG}}_i(D_i) \leq b_{i\,\max}, \quad \forall i.$$

Determining the input assumptions of the EDF component also involves finding the pseudo-inverse functions of the relations. Finding the input assumptions for the upper arrival curves involves inverting (4.9), (4.10), and (4.12). Again, we need to compute the largest upper curves for which the relations still hold. Finding the

inverses and combing them, we find for all streams i:

$$\alpha_i^{\mathrm{uA}}(\Delta) = \min \left\{ \beta^{\mathrm{lG}}(\Delta + D_i) - \sum_{\substack{j=1 \\ j \neq i}}^{N} r\left(\alpha_j^{\mathrm{uG}}, (D_j - D_i), \Delta\right), \right.$$
$$\left. s\left(\alpha_i'^{\mathrm{uA}}, (D_i - \mathrm{BCET}_i), \Delta\right),\ t(D_i, b_{i\,\max}, \Delta) \right\},$$

using functions $s(\alpha, c, \Delta)$ and $t(d, b, \Delta)$ defined as:

$$s(\alpha, c, \Delta) = \begin{cases} \alpha(\Delta - c) & \text{if } \Delta > c \\ \lim_{\epsilon \to 0}\{\alpha(\epsilon)\} & \text{if } 0 < \Delta \leq c \\ 0 & \text{if } \Delta = 0 \end{cases} \qquad t(d, b, \Delta) = \begin{cases} \infty & \text{if } \Delta > d \\ b & \text{if } 0 < \Delta \leq d \\ 0 & \text{if } \Delta = 0 \end{cases}$$

Calculating the input assumption for the lower curve is much simpler as it involves finding the smallest lower curve solution to the pseudo-inverse of (4.11) or $\alpha_i^{\mathrm{lA}}(\Delta) \geq \alpha_i'^{\mathrm{lA}}(\Delta + (D_i - \mathrm{BCET}_i))$ for all i. Therefore, we can determine the following assume interface function for the lower curve of each input data stream:

$$\alpha_i^{\mathrm{lA}}(\Delta) = r(\alpha_i'^{\mathrm{lA}}, -(D_i - \mathrm{BCET}_i), \Delta), \quad \forall i,$$

using the shift function r as defined in (4.5).

Similarly, for the assume of the lower service curve we invert (4.9) which yields the inequality $\beta^{\mathrm{lA}}(\Delta) \geq \sum_{i=1}^{N} r(\alpha_i^{\mathrm{uG}}, D_i, \Delta)$. Therefore, the input assume for the lower service curve of an EDF component can be determined as:

$$\beta^{\mathrm{lA}}(\Delta) = \sum_{i=1}^{N} r(\alpha_i^{\mathrm{uG}}, D_i, \Delta). \tag{4.20}$$

The interface model for the EDF component is illustrated in Fig. 4.2c.

4.3.4 Worst-Case Traversal Time Interface

We develop an additional type of interface to alleviate design of systems with WCTT constraints that can span a network of components. It is an interface-based interpretation of the analytical computation of WCTTs with (4.14).

The "complies to" relation $\vdash$ for this interface connection is defined as $\Pi^{\mathrm{G}}(\Delta) \vdash \Pi^{\mathrm{A}}(\Delta) = (\forall \Delta : \Pi^{\mathrm{G}}(\Delta) \geq \Pi^{\mathrm{A}}(\Delta))$, where Π^{A} expresses the minimum service requested from all subsequent components such that the WCTT constraint

is satisfied, and Π^{G} expresses the minimum service guaranteed by all subsequent components.

Computing the guarantee for a sequence of components follows directly from (4.14) and can be done with $\Pi^{\mathrm{G}} = \beta^{\mathrm{G}}_{\mathscr{P}\mathscr{E}} \otimes \rho^{\mathrm{G}}_{\mathscr{P}\mathscr{B}} \otimes \beta^{\mathrm{G}}_{\mathscr{E}\mathscr{D}\mathscr{F}}$. Connecting a PE component to the sequence would change the combined service to $\Pi'^{\mathrm{G}} = \beta^{\mathrm{lG}} \otimes \Pi^{\mathrm{G}}$ where β^{lG} is the lower service guaranteed by the PE. Similarly, connecting a PB component we would have $\Pi'^{\mathrm{G}} = \rho^{\mathrm{lG}}_{\tau} \otimes \Pi^{\mathrm{G}}$, where $\rho^{\mathrm{l}}_{\tau}(\Delta)$ is the lower guaranteed shifted readout curve as defined with (4.8). For an EDF component, we have $\Pi'^{\mathrm{G}} = \beta^{\mathrm{lG}}_{D_i} \otimes \Pi^{\mathrm{G}}$ where $\beta^{\mathrm{lG}}_{D_i}$ is the service curve provided to the stream when processed with a relative deadline D_i as defined in (4.13).

From (4.14), we can also compute the assume on the combined service of a sequence of components as $\Pi^{\mathrm{A}} = r(\alpha^{\mathrm{uG}}, D, \Delta)$ which expresses the minimum necessary service in order to meet a WCTT constraint of D for the input α^{uG}. Propagating the assume value through a sequence of components can be done for the three types of components by inverting (4.14) as follows:

$$\mathrm{PE} : \Pi^{\mathrm{A}} = \Pi'^{\mathrm{A}} \oslash \beta^{\mathrm{lG}}, \quad \mathrm{PB} : \Pi^{\mathrm{A}} = \Pi'^{\mathrm{A}} \oslash \rho^{\mathrm{lG}}_{\tau}, \quad \mathrm{EDF} : \Pi^{\mathrm{A}} = \Pi'^{\mathrm{A}} \oslash \beta^{\mathrm{lG}}_{D_i}.$$

We can also compute component-wise constraints on the resources provided to each component given the resource assumption from preceding components Π'^{A}, and the resource guarantee from subsequent components Π^{G}. We can do this for three types of components as follows:

$$\mathrm{PE} : \beta^{\mathrm{lA}} \geq \Pi'^{\mathrm{A}} \oslash \Pi^{\mathrm{G}}, \quad \mathrm{PB} : \rho^{\mathrm{lA}}_{\tau} \geq \Pi'^{\mathrm{A}} \oslash \Pi^{\mathrm{G}}, \quad \mathrm{EDF} : \beta^{\mathrm{lA}}_{D_i} \geq \Pi'^{\mathrm{A}} \oslash \Pi^{\mathrm{G}}.$$

The above constraints can be combined with the previously computed input assumes for the resources of the three components with (4.18), (4.19), and (4.20). By doing this, satisfying all interface relations of components composed in a sequence will guarantee that the WCTT constraint on the sequence of components is satisfied too. The WCTT interfaces for the PE, PB, and EDF components are shown in Fig. 4.2a–c.

4.4 Illustrative Example

In this section we show how our proposed theory can be applied to an example system shown in Fig. 4.3. It represents a typical distributed embedded system for video-/signal-/media-processing. Communication is not modeled explicitly however, this can be done by adding additional components if necessary.

Each PE, PB, and EDF component is considered to be an independent component, and our objective is to connect them together to realize the architecture shown in the figure. In order to decide whether two components can be connected together, we would only inspect their interfaces. Two compatible interfaces implicitly

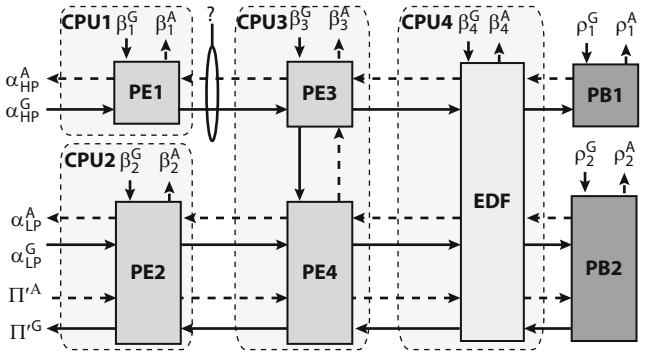


Fig. 4.3 Interface model of an example stream processing system

guarantee that the buffers inside their respective components will never overflow or underflow, and in addition, the WCTT constraints are satisfied.

The main message in this section is an illustration of how the internal details of a component (e.g. its buffer size, scheduling policy, processor frequency, deadline) are reflected (or summarized) through its interfaces. We show that if these internal details are changed then the component's interfaces also change and two previously compatible components may become incompatible (or vice versa).

Experimental Setup We consider the system illustrated in Fig. 4.3. It consists of a multiprocessor platform with four CPUs. A distributed application is mapped to the platform. It processes two data streams, a high priority one denoted as *HP*, and a low priority one denoted as *LP*. The application consists of six tasks. Streams are preprocessed by the tasks modeled with components *PE*1 and *PE*2 which are mapped separately to *CPU*1 and *CPU*2, respectively. Afterwards, they are processed by components *PE*3 and *PE*4 which are mapped to *CPU*3. The tasks share the resource using FP scheduling where stream *HP* is given higher priority. Additionally, streams are processed by two tasks mapped to *CPU*4 which they share with the EDF policy. This is modeled with the EDF component. The fully processed streams are written to playout buffers which are read by an I/O interface at a constant rate. The buffers are modeled with components *PB*1 and *PB*2. For simplicity, the communication is not modeled here. If necessary, it can be taken into account by additional components in the model.

Data packets from the streams have bursty arrivals described with period p, jitter j, and minimum inter-arrival distance d. For the HP stream the parameters are $p = 25$, $j = 40$, $d = 0.1$ ms, and for *LP* stream they are $p = 25$, $j = 30$, $d = 0.1$ ms. Each data packet from the two streams has a constant processing demand of 1 cycle for all tasks. *CPU*1 is fully available with service availability of 0.3 cycles/ms. For *CPU*2, *CPU*3, and *CPU*4, the respective service availabilities are 0.3, 0.4, and 0.4 cycles/ms. Components *PE*3 and *PE*4 have internal buffer sizes of 2 and 3 packets, respectively. These buffers should never overflow. The EDF

component schedules tasks processing streams *HP* and *LP* with relative deadlines of 8 and 10 ms, respectively, with both buffers being limited to 3 packets. These buffers should also never overflow. Components *PB*1 and *PB*2 are read at a constant rate of 25 packets/ms. Both components have maximum buffer sizes of 8 data packets, and initially they contain 4 data packets. Both buffers should not underflow and overflow. Additionally, we have a WCTT constraint on the *LP* stream of 200 ms.

Results We consider three different scenarios of the system's parameters. In each of them, we check the compatibility of component *PE*1 with the partially designed system when all other components are already connected. Compatibility is checked by only inspecting the interface connection between *PE*1 and the system which is marked with "?" in Fig. 4.3. Compatibility meaning that the output guarantee is fully "enclosed" by the input assumption.

Case I The system is considered with the specified parameters. The components turn out to be compatible. The interface connection is illustrated in Fig. 4.4a. It shows that the guarantee on the output stream rate $\alpha^{\mathrm{G}}_{PE1}$ expressed by *PE*1s interface is compatible with the assumption on the input rate $\alpha^{\mathrm{A}}_{PE3}$ expressed by *PE*3s interface.

Case II The WCTT constraint on the *LP* stream is decreased to 192 ms. This leads to incompatibility between components *PE*1 and *PE*3 which reveals in the interface connection as shown in Fig. 4.4b.

Case III The maximum buffer size of component *PB*2 is decreased to 5 packets which leads to incompatibility as shown in Fig. 4.4c. This reveals how constraints in component *PB*2 are propagated to multiple interfaces in the rest of the system in order to guarantee their satisfaction.

In summary, we have shown through a concrete example how incremental compatibility checking can be done using the proposed interfaces. Clearly, such interfaces can also be used in a straightforward manner for resource dimensioning and component-level design space exploration. Typical questions that one would ask are: What is the minimum buffer size of a component such that its interface

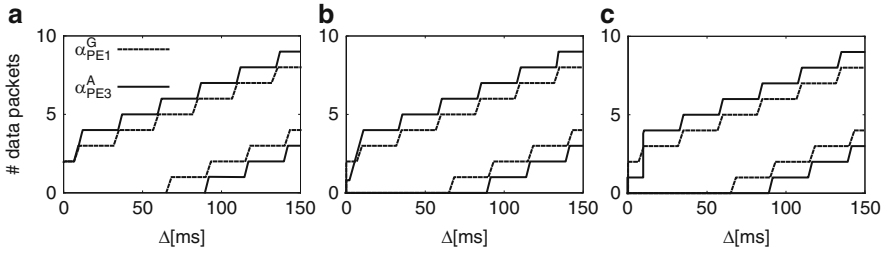


Fig. 4.4 Interface connection between the output guarantee of component $PE1$ and the input assumption of component $PE3$ shows: (**a**) compatibility, (**b**) incompatibility when WCTT for stream LP is reduced to 192 ms, and (**c**) incompatibility when buffer of component $PB2$ is decreased to 5 packets

is compatible with a partially existing design? What is the minimum processing frequency such that the interface is still compatible? Or what are the feasible relative deadlines in an EDF component? In this chapter, we are concerned with buffer and WCTT constraints however, one can imagine developing similar interfaces for power, energy, and temperature constraints.

4.5 Concluding Remarks

In this chapter we proposed an interface algebra for checking whether multiple components of an embedded system may be composed together while satisfying their buffer and worst-case traversal time (WCTT) constraints. The main advantage of such an interface-based formulation is that component composition only requires a compatibility checking of the interfaces of the components involved, without having to compute the WCTT of the entire component network from scratch, each time a new component is added or an existing component is modified. This has a number of advantages. It significantly reduces design complexity, it does not require components to expose the details of their internals, and it allows a correct-by-construction design flow.

The interfaces studied here were purely functional in nature, i.e. they do not contain any state information. This might be restrictive in a number of settings, e.g. when the components implement complex protocols. As an example, the processing rate of a component might depend on the "state" or the fill level of an internal buffer. As a part of future work, we plan to extend our interface algebra to accommodate such "stateful" components. This may be done by describing an automaton to represent an interface, with language inclusion or equivalence to denote the notion of *compatibility* between components.

Appendix

The min-plus algebra convolution $\otimes$ and deconvolution $\oslash$ operators are defined as:

$$(f \otimes g)(\Delta) = \inf_{0 \le \lambda \le \Delta} \{f(\Delta - \lambda) + g(\lambda)\},$$

$$(f \oslash g)(\Delta) = \sup_{\lambda \ge 0} \{f(\Delta + \lambda) - g(\lambda)\}.$$

The duality between $\otimes$ and $\oslash$ states that: $f \oslash g \le h \iff f \le g \otimes h$.
The inverses of the $RT(\beta, \alpha)$ are defined as:

$$\alpha = RT^{-\alpha}(\beta', \beta) \Rightarrow \beta' \le RT(\beta, \alpha),$$

$$\beta = RT^{-\beta}(\beta', \alpha) \Rightarrow \beta' \le RT(\beta, \alpha),$$

with solutions:

$$RT^{-\alpha}(\beta', \beta)(\Delta) = \beta(\Delta + \lambda) - \beta'(\Delta + \lambda) \text{ for } \lambda = \sup\left\{\tau \ : \ \beta'(\Delta + \tau) = \beta'(\Delta)\right\},$$

$$RT^{-\beta}(\beta', \alpha)(\Delta) = \beta'(\Delta - \lambda) + \alpha(\Delta - \lambda) \text{ for } \lambda = \sup\left\{\tau \ : \ \beta'(\Delta - \tau) = \beta'(\Delta)\right\}.$$

References

1. de Alfaro L, Henzinger TA (2001) Interface theories for component-based design. In: Proceedings of the first international workshop on embedded software, EMSOFT '01, Springer, London, pp 148–165
2. de Alfaro L, Henzinger TA (2005) Interface-based design. In: Broy M, Gruenbauer J, Harel D, Hoare C (eds) Engineering theories of software-intensive systems, NATO Science Series: Mathematics, physics, and chemistry, vol 195. Springer, Berlin, pp 83–104
3. Baruah S, Chen D, Gorinsky S, Mok A (1999) Generalized multiframe tasks. Real-Time Syst 17(1):5–22
4. Chakraborty S, Künzli S, Thiele L (2003) A general framework for analysing system properties in platform-based embedded system designs. In: Proceedings of the conference on design, automation and test in Europe, vol 1. IEEE Computer Society, Washington, DC, pp 10,190–10,195
5. Chakraborty S, Liu Y, Stoimenov N, Thiele L, Wandeler E (2006) Interface-based rate analysis of embedded systems. In: Proceedings of the 27th IEEE international real-time systems symposium, RTSS '06, IEEE Computer Society, Washington, DC, pp 25–34
6. Le Boudec JY, Thiran P (2001) Network calculus: A theory of deterministic queuing systems for the internet, LNCS, vol 2050. Springer, Berlin
7. Maxiaguine A, Künzli S, Thiele L (2004) Workload characterization model for tasks with variable execution demand. In: Proceedings of the conference on design, automation and test in Europe, vol 2. IEEE Computer Society, Washington, DC, pp 21,040–21,045
8. Thiele L, Chakraborty S, Naedele M (2000) Real-time calculus for scheduling hard real-time systems. In: Circuits and Systems, 2000. Proceedings of the 2000 IEEE international symposium on ISCAS 2000 Geneva, vol 4, pp 101–104 (2000)
9. Thiele L, Wandeler E, Stoimenov N (2006) Real-time interfaces for composing real-time systems. In: Proceedings of the 6th ACM & IEEE international conference on embedded software, EMSOFT '06, ACM, New York, pp 34–43
10. Wandeler E, Thiele L (2006a) Interface-based design of real-time systems with hierarchical scheduling. In: Proceedings of the 12th IEEE real-time and embedded technology and applications symposium, RTAS '06, IEEE Computer Society, Washington, DC, pp 243–252
11. Wandeler E, Thiele L (2006b) Optimal TDMA time slot and cycle length allocation for hard real-time systems. In: Proceedings of the 2006 Asia and South Pacific design automation conference, ASP-DAC '06, IEEE Press, Piscataway, NJ, pp 479–484
12. Wandeler E, Maxiaguine A, Thiele L (2006) Performance analysis of greedy shapers in real-time systems. In: Proceedings of the conference on design, automation and test in Europe: Proceedings of the European Design and Automation Association, 3001 Leuven, Belgium, pp 444–449

Chapter 5
The Logical Execution Time Paradigm

Christoph M. Kirsch and Ana Sokolova

Since its introduction in 2000 in the time-triggered programming language Giotto, the Logical Execution Time (LET) paradigm has evolved from a highly controversial idea to a well-understood principle of real-time programming. This chapter provides an easy-to-read overview of LET programming languages and runtime systems as well as some LET-inspired models of computation. The presentation is intuitive, by example, citing the relevant literature including more formal treatment of the material for reference.

5.1 LET Overview

Logical Execution Time (LET) is a real-time programming abstraction that was introduced with the time-triggered programming language Giotto [23, 24, 27]. LET abstracts from the actual execution time of a real-time program. LET determines the time it takes from reading program input to writing program output regardless of the time it takes to execute the program. LET is motivated by the observation that the relevant behavior of real-time programs is determined by when input is read and output is written and not when programs just execute any code.

Before the introduction of LET two other rather different real-time programming abstractions had been around for quite some time that originated from two largely disjoint communities: the Zero Execution Time (ZET) abstraction [31, 35] as the foundation of synchronous reactive programming [18] and the Bounded Execution Time (BET) abstraction [31,35] as the foundation of real-time scheduling theory [7]. Figure 5.1 shows the three abstractions.

C.M. Kirsch (✉) · A. Sokolova
Department of Computer Sciences, University of Salzburg, Austria
e-mail: ck@cs.uni-salzburg.at; anas@cs.uni-salzburg.at

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_5,

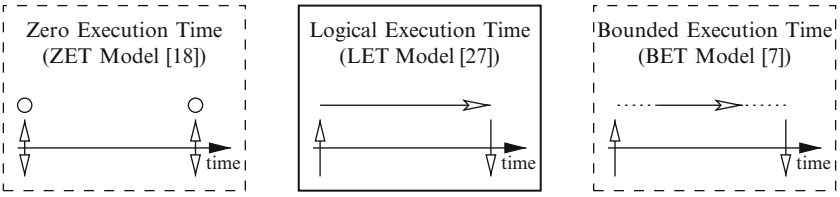


Fig. 5.1 Fundamental real-time programming abstractions [31, 35]

Similar to the LET abstraction, ZET also abstracts from the actual execution time of a real-time program yet even more than LET. With ZET the execution time of a program including reading input and writing output is assumed to be zero, or equivalently, the execution platform of a program is assumed to be infinitely fast. ZET is the key abstraction of synchronous reactive programming. ZET programs are reactive, i.e., always react to input with some output, and synchronous, i.e., do so in zero time. The execution of ZET programs is correct if the program always writes output before new input becomes available. Establishing correctness typically involves fixed-point analysis since ZET programs written in synchronous reactive programming languages such as Lustre [19] and Esterel [5] may contain cyclic dependencies.

Unlike the ZET and LET abstractions, BET does not abstract execution times but instead bounds them using deadlines. Strictly speaking, BET is therefore a model for temporal constraint programming rather than a programming abstraction. With BET a real-time program has a deadline, which constrains correct program execution to the instances when the program completes execution before or at the deadline. In the BET model, an execution of the program is incorrect if the program does not complete within the deadline, even if the program eventually completes with a functionally correct result. Correct execution of concurrent real-time programs with multiple, possibly different and recurring deadlines requires real-time scheduling. Rate-monotonic (RM) and earliest-deadline-first (EDF) scheduling [36] are prominent examples of scheduling strategies in the BET model.

LET is inspired by both the ZET abstraction and the BET model. LET program execution is correct, i.e., time-safe [24, 27], if the program reads input, in zero time, then executes, and finally writes output, again in zero time, exactly when the LET has elapsed since reading input. The end of the LET thus corresponds to a deadline in the BET model but only for program execution without reading input and writing output. In other words, if the program completes execution before the deadline, writing output is delayed until the deadline, i.e., until the LET has elapsed. The deadline is therefore not only an upper bound, like in the BET model, but also a lower bound, at least, logically. In the LET model, using a faster machine does therefore not result in (logically) faster program execution but only in decreased machine utilization, which makes room for more concurrency. Conversely, more concurrency on the same machine has no effect on input and output times as long as the machine is sufficiently fast to accommodate all deadlines.

5.1.1 Giotto

LET programs may be event- or time-triggered (or both) in the sense that the time when input is read and thus when the LET begins is determined by the occurrence of an event or the progress of time, respectively. Giotto programs, for example, are time-triggered LET programs while xGiotto [16] programs may be both event- and time-triggered LET programs, and even the LET itself in xGiotto programs may be determined by events rather than time.

There are two key results on Giotto programs. Checking time safety of Giotto programs is easy [25] and time-safe Giotto programs are time-deterministic [24, 27], i.e., the relevant input and output (I/O) behavior of time-safe Giotto programs does not change across varying hardware platforms and software workloads. Time-safe execution of Giotto programs requires real-time scheduling, similar to BET programs, but no fixed-point analysis, unlike ZET programs, since reading input and writing output is cycle-free. Note that all real-time programming paradigms have yet in common that they require some form of hardware-dependent, worst-case execution time (WCET) analysis [14] for establishing correctness.

Giotto programs may specify multiple modes and mode switching logic but can only be in one mode at a time during execution. Checking time safety of Giotto programs is easy, i.e., fast, because time safety of the individual modes of a Giotto program implies time safety of the whole program regardless of mode switching [25]. The converse is not true since the mode switching logic may prevent modes that are not time-safe from ever being executed. Checking time safety of a mode is linear in the size of its description using a standard, utilization-based schedulability test based on EDF.

LET programs may be distributed across multiple machines, just like ZET and BET programs. However, the key difference is that LET, unlike ZET and BET, is a natural, temporally robust placeholder not just for program execution but also for program communication [29]. Distributing LET programs is thus easy, in particular, onto architectures that provide time synchronization such as the Time-Triggered Architecture (TTA) [34]. Even more importantly, the relevant behavior of a time-safe, distributed version of a time-safe, non-distributed LET program is equivalent to the relevant behavior of the non-distributed program.

However, Giotto has a scalability issue in the sense that each mode of a Giotto program needs to specify the whole behavior of the program while being in that mode. For example, if a Giotto programmer would like to maintain some behavior of a mode when switching to another mode, both modes need to specify their common behavior. In other words, a Giotto program is a flat, non-hierarchical description of a real-time program. Giotto programs may therefore become large for non-trivial, in particular distributed applications. Giotto programming by example is discussed in more detail in Sect. 5.2.

5.1.2 Hierarchical Timing Language

The Hierarchical Timing Language (HTL) [15, 30] is a Giotto successor that aims at improving succinctness while keeping time safety checking easy. Modes in HTL are partial specifications of LET programs that may be hierarchical in the sense that some modes may be abstract placeholders reserving computation time for refining, more concrete modes that specify actual program behavior. Intuitively, a concrete mode refines an abstract mode if the LETs in the concrete mode start later and end earlier than the LETs in the abstract mode. The key result is that time safety of an abstract HTL program implies time safety of any concrete HTL program that refines the abstract program [15]. The converse is again not true. There may be time-safe concrete HTL programs that refine abstract HTL programs that are not time-safe. Note that checking refinement is easier than checking time safety [30].

Refinement in HTL enables modular real-time programming [30]. A time-safe HTL program may be changed locally without the need for re-checking time safety globally, with the exception of the top level of the program. Modifications to the most abstract portion of an HTL program may require re-checking time safety for the whole program. Since correctness of modifications below top level can be checked fast, i.e., independently of the size of the whole program, HTL programs may even be modified at runtime through a process called runtime patching while preserving the real-time behavior of the unmodified parts [33]. Runtime patching is a semantically robust method for introducing flexibility into real-time programming. An HTL programming example is discussed in more detail in Sect. 5.2.

5.1.3 Model-Driven Development

LET programming is part of a larger, model-driven development (MDD) methodology [26] depicted in Fig. 5.2. LET programs may be modeled and validated in a simulation environment such as Simulink [10] and then translated to executable code. Here, the key idea is that LET model, program, and code are equivalent with respect to their relevant real-time behavior [26]. Changes on one level have therefore a well-understood effect on the other levels enabling compositional implementation and validation with LET-based toolkits such as FTOS [6] and TDL [13]. LET-oriented runtime systems are discussed next.

5.1.4 The Embedded Machine

LET code generators may target general purpose programming languages such as C or virtual machines that have been specifically designed for LET semantics such as the Embedded Machine [20, 22], or E machine, for short, which is an interpreter for

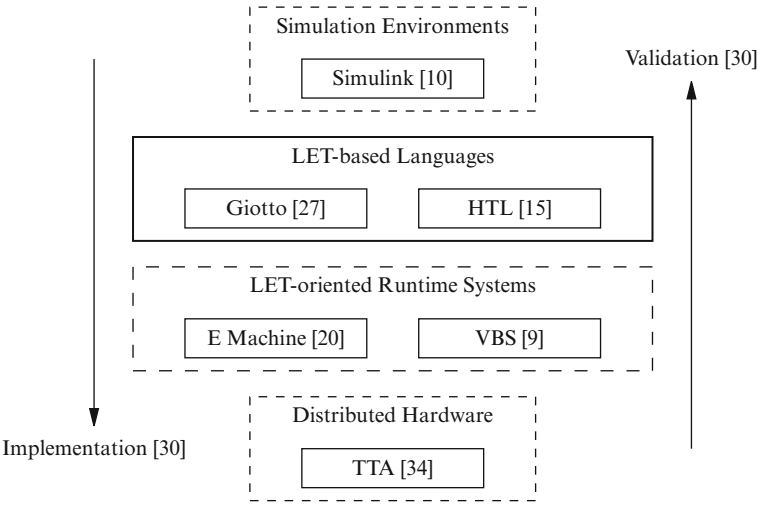


Fig. 5.2 Context of LET-based languages and LET-oriented runtime systems

E code. Similar to Giotto and HTL programs, time-safe E code is time-deterministic. We also say that time-safe E code is time-portable [1, 3] to any platform for which an E machine implementation exists. E code may also be executed on distributed systems such as TTA by running an E machine on each host [29].

Checking time safety of arbitrary E code may be difficult but remains easy for non-trivial classes of E code [21] such as E code generated from Giotto and HTL [25]. Executing E code requires an E machine as well as an EDF-scheduler. However, scheduling decisions may also be computed at compile time and represented by Schedule-Carrying Code (SCC) [28], which is E code extended by specific scheduling instructions. SCC is an executable witness of time safety. Checking whether SCC is time-safe is in general easier than generating SCC. An E machine extended for SCC support may therefore verify time safety prior to execution and does not require an EDF-scheduler [32].

E code generated from a Giotto program requires pseudo-polynomial space in the size of the program, i.e., the numerically represented program periods [25]. E code execution time is linear in the size of the program in this case. E code generated from an HTL program may even be exponentially larger than the program regardless of the periods because any hierarchy in the program is flattened prior to code generation. Flattening can be avoided by targeting a hierarchical version of the E machine [17]. The resulting E code requires then again only pseudo-polynomial space in the size of the program and can be executed in linear time. The E machine is discussed in more detail in Sect. 5.2.

5.1.5 Variable-Bandwidth Servers

Giotto and HTL are languages in which LET programs are constructed around the notion of modes. However, a LET program may also be understood as a specification of a set of concurrent processes where each process performs I/O as fast as possible, i.e., logically in zero time, and then computes as predictable as possible, i.e., logically for a given amount of time, and then performs I/O again and so on. After performing I/O the process may decide, based on the previous compution and the new input, what to compute next, which is essentially another way of switching modes. The logical execution time of each computation phase may also change as long as it is determined by the process itself.

Variable-Bandwidth Servers (VBS) [9] may provide a natural scheduling platform for executing concurrent processes specified by a LET program. A VBS process is a sequence of actions. Each action is sequential code, which is executed in temporal isolation of any other process in the system, i.e., lower and upper bounds on the time to execute the action are solely determined by the invoking process. The bounds may change from one action to the next to accommodate different types of actions such as latency-oriented as-fast-as-possible I/O actions and throughput-oriented yet as-predictable-as-possible computation actions. The bounds can be seen as a generalization of LET from an exact logical value of duration to a realistic interval of permitted values. Running LET programs on VBS remains to be a subject of future work.

5.1.6 Models of Computation

Figure 5.3 shows a selection of LET-inspired models of computation. Exotasks [1,3] implement a real-time scheduling framework in Java using the real-time garbage collector Metronome [4] for real-time performance. The framework has been instantiated to provide LET semantics in Java. A more general version called

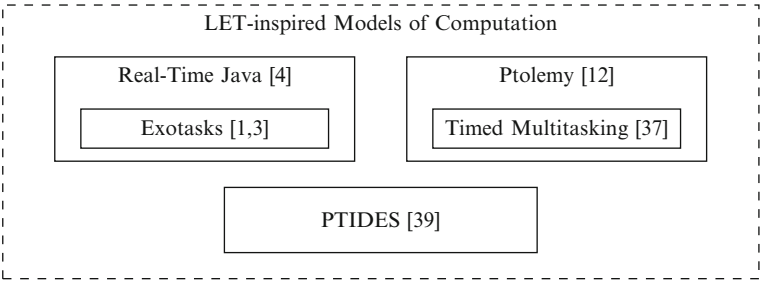


Fig. 5.3 LET-inspired models of computation

Flexotasks [2] provides even more freedom to implement and integrate temporal and spatial isolation policies.

The key motivation of LET programming is to develop systems that maintain their relevant real-time behavior across changing hardware platforms and software workloads. However, requiring all program parts to follow the LET regime may be unnecessarily restrictive. For example, program parts that are independent of I/O behavior may be scheduled in a more flexible manner without loosing guarantees on relevant behavior [11]. Timed multitasking [37] in Ptolemy [12] provides LET guarantees relative to the occurrences of events, similar to the previously mentioned xGiotto [16]. However, event scoping in xGiotto enables structured specification of event handling policies such as implicit assumptions on interarrival times to facilitate time safety analyses. Discrete-event models in PTIDES [39] provide another model of computation that enforces LET but only at communication boundaries, i.e., sensing, actuating, and other relevant I/O is performed at time instants that are independent of execution order and speed.

5.2 LET Programming by Examples

5.2.1 Giotto

A Giotto program consists of a functionality part and a timing part. The functionality part contains port, driver, and task declarations, which interface the Giotto program to a functionality implementation, written in C, for example. For the examples below, we show the timing part of a Giotto program.

5.2.1.1 Single-Mode Giotto Program

As a first example, we present a highly simplified version of the control program for a model helicopter such as the JAviator [8]. Consider the helicopter in hover mode m. There are two tasks, both given in native code, possibly autogenerated from Matlab/Simulink models [26]: the control task t_1, and the navigation task t_2. The navigation task processes GPS input every 10 ms and provides the processed data to the control task. The control task reads additional sensor data (not modeled here), computes a control law, and writes the result to actuators (reduced here to a single port). The control task is executed every 20 ms. The data communication requires three drivers: a sensor driver d_s, which provides the GPS data to the navigation task; a connection driver d_i, which provides the result of the navigation task to the control task; and an actuator driver d_a, which loads the result of the control task into the actuator. The drivers may process the data in simple ways (such as type conversion), as long as their WCETs are negligible. In general, since E code execution is synchronous and can thus not be interrupted by other E code, we say

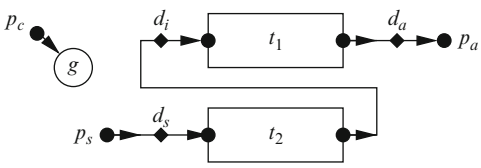


Fig. 5.4 The dataflow of the example with two periodic tasks [22]

that the WCET of an E code block (i.e., the sum of the WCETs of all drivers as well as all E code instructions called in that block) is negligible if it is shorter than the minimal time between any two events that can trigger the execution of E code. In the case of the helicopter software, the WCETs of all E code blocks are at least one order of magnitude shorter than 10 ms, which is the time between two consecutive invocations of E code in this example.

There are two environment ports, namely, a clock p_c and the GPS sensor p_s; two task ports, one for the result of each task; and three driver ports – the destinations of the three drivers – including the actuator p_a. Figure 5.4 shows the dataflow of the program: we denote ports by bullets, tasks by rectangles, drivers by diamonds, and triggers by circles. It therefore presents an abstract functional implementation of the program. Here is a Giotto description of the program timing:

```
mode m() period 20 {
  actfreq 1 do pa(da);
  taskfreq 1 do t1(di);
  taskfreq 2 do t2(ds);
}
```

The "`actfreq` 1" statement causes the actuator to be updated once every 20 ms; the "`taskfreq` 2" statement causes the navigation task to be invoked twice every 20 ms; etc. Note that the LET of each task is specified by the ratio of the mode period over the task frequency (20 ms for t_1 and 10 ms for t_2). Here is a simplified version of the E code generated by the Giotto compiler:

```
a1:  call(da)          a2:  call(ds)
     call(ds)               release(t2)
     call(di)               future(g, a1)
     release(t1)
     release(t2)
     future(g, a2)
```

The E code consists of two blocks. The block at address a_1 is executed at the beginning of a period, say, at 0 ms: it calls the three drivers, which provide data for the tasks and the actuator, then hands the two tasks to the scheduler, and finally activates a trigger g with address a_2. When the block finishes, the trigger queue of the E machine contains the trigger g bound to address a_2, and the ready queue of

the scheduler contains two tasks, t_1 and t_2. Now the E machine relinquishes control, only to wake up with the next input event that causes the trigger g to evaluate to true. In the meantime, the scheduler takes over and assigns CPU time to the tasks in the ready queue according to some scheduling scheme. When a task completes, the scheduler removes it from the ready queue.

There are two kinds of input events, one for each environment port: clock ticks, and changes in the value of the sensor p_s. The implementation of the trigger g is part of the functional code. In the example, the trigger g: $p_c' = p_c + 10$ specifies that the E code at address a_2 will be executed after 10 clock ticks. Logically, the E machine wakes up at every input event to evaluate the trigger, finds it to be false, until at 10 ms, the trigger is true. An efficient implementation, of course, wakes up the E machine only when necessary, in this case at 10 ms. The trigger g is now removed from the trigger queue and the associated a_2 block is executed. It calls the sensor driver, which updates a port read by task t_2. There are two possible scenarios: the earlier invocation of task t_2 may already have completed, and is therefore no longer in the ready queue when the a_2 block is executed. In this case, the E code proceeds to put another invocation of t_2 into the ready queue, and to trigger the a_1 block in another 10 ms, at 20 ms. In this way, the entire process repeats every 20 ms. The other scenario at 10 ms has the earlier invocation of task t_2 still incomplete, i.e., in the ready queue. In this case, the attempt by the sensor driver to overwrite a port read by t_2 causes a runtime exception, called time-safety violation. At 20 ms, the end of the mode period, when ports read by both tasks t_1 and t_2 are updated and ports written by both t_1 and t_2 are read (via the drivers), a time-safety violation occurs unless both tasks have completed, i.e., the ready queue must be empty. In other words, an execution of the program is time-safe if the scheduler ensures the following: (1) each invocation of task t_1 at $20n$ ms, for $n \geq 0$, completes by $20n + 20$ ms; (2) each invocation of task t_2 at $20n$ ms completes by $20n + 10$ ms; and (3) each invocation of task t_2 at $20n + 10$ ms completes by $20n + 20$ ms. Therefore, a necessary requirement for time safety is $\delta_1 + 2\delta_2 < 20$, where δ_1 is the WCET of task t_1, and δ_2 is the WCET of t_2. If this requirement is satisfied, then a scheduler that gives priority to t_2 over t_1 guarantees time safety. Figure 5.5 presents a time-safe EDF schedule of the two-task Giotto example, with $\delta_1 = 10$ ms and $\delta_2 = 4$ ms.

The E code implements the Giotto program correctly only if it is time-safe: during a time-safe execution, the navigation task is executed every 10 ms, the control task every 20 ms, and the dataflow follows Fig. 5.4. Thus the Giotto compiler needs to ensure time safety when producing E code. In order to ensure this, the compiler needs to know the WCETs of all tasks and drivers (cf., for example, [14]), as well as the scheduling scheme used by the operating system. With this information, time safety for E code produced from Giotto can be checked [25]. However, for arbitrary E code and platforms, WCET analysis and time-safety checking may be difficult, and the programmer may have to rely on runtime exception handling, see [22] for more details. At the other extreme, if the compiler is given control of the system scheduler, it may synthesize a scheduling scheme that ensures time safety [28].

The time-safe executions of the E code example have an important property: assuming the two tasks compute deterministic results, for given sensor values that

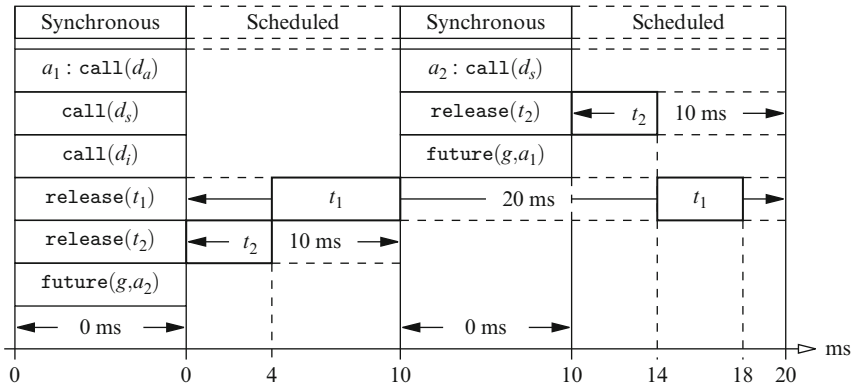


Fig. 5.5 Earliest-deadline-first (EDF) schedule of the two-task Giotto program (adapted from [22])

are read at the input port p_s at times 0, 10, 20, ... ms, the actuator values that are written at the output port p_a at times 0, 20, 40, ... ms are determined, i.e., independent of the scheduling scheme. This is a consequence of the LET paradigm, because each invocation of the control task t_1 at $20n$ ms operates on an argument provided by the invocation of the navigation task t_2 at $20n-10$ ms, whether or not the subsequent invocation of t_2, at $20n$ ms, has completed before the control task obtains the CPU. Time safety, therefore, ensures not only deterministic output timing, but also deterministic output values; it guarantees predictable, reproducible real-time code.

5.2.1.2 Multiple-Mode Giotto Program

The helicopter may change mode, say, from hover to descend, and in doing so, apply a different filter. In this case, the navigation task t_2 needs to be replaced by another task t_2'. We show how to implement different modes of operation using Giotto and the generated E code with control-flow instructions. Note that E code can also be changed dynamically, at runtime, still guaranteeing determinism if no time-safety violations occur. This capability enables the real-time programming of embedded devices that upload code on demand, of code that migrates between hosts, and of code patches [22].

Consider the following timing part of a Giotto program with two modes, m_a (representing the helicopter in hover mode) and m_b (descend mode):

```
start m_a {
  mode m_a() period 20 {
    actfreq 1 do p_a(d_a);
    exitfreq 2 do m_b(c_b);
    taskfreq 1 do t_1(d_i);
    taskfreq 2 do t_2(d_s);
  }
```

```
    mode m_b() period 20 {
      actfreq 1 do p_a(d_a);
      exitfreq 2 do m_a(c_a);
      taskfreq 1 do t_1(d_i);
      taskfreq 2 do t'_2(d_s);
    }
}
```

The program begins by executing mode m_a, which is equivalent to the (single) mode m of the Giotto program from the previous subsection except for the mode switch to mode m_b. A mode switch in Giotto has a frequency that determines at which rate an exit condition is evaluated. The exit condition c_b of mode m_a is evaluated once every 10 ms (the ratio of the period over the exit frequency). If c_b evaluates to true, then the program switches to mode m_b, which is similar to mode m_a except that task t_2' replaces task t_2. Task t_2' applies a different filter on the same ports as t_2. The mode switch back to m_a evaluates the exit condition c_a also once every 10 ms.

This example consists of two modes with equal periods. Programs with multiple nodes of different periods are also possible in Giotto but mode switching is restricted at the end of the node period, i.e., only exit frequency of 1 is allowed.

In order to express mode switching in E code, we use a conditional branch instruction $\texttt{if}(c, a)$. The first argument c is a condition, which is a predicate on some ports. The second argument a is an E code address. The $\texttt{if}(c, a)$ instruction evaluates the condition c synchronously (i.e., in logical zero time), similar to driver calls, and then either jumps to the E code at address a (if c evaluates to true), or proceeds to the next instruction (if c evaluates to false). Here is the E code that implements the above Giotto program:

```
a_1:  call(d_a)                 a_3:  call(d_a)
      if(c_b, a'_3)                   if(c_a, a'_1)
a'_1: call(d_s)                 a'_3: call(d_s)
      call(d_i)                       call(d_i)
      release(t_1)                    release(t_1)
      release(t_2)                    release(t'_2)
      future(g, a_2)                  future(g, a_4)

a_2:  if(c_b, a'_4)             a_4:  if(c_a, a'_2)
a'_2: call(d_s)                 a'_4: call(d_s)
      release(t_2)                    release(t'_2)
      future(g, a_1)                  future(g, a_3)
```

The two E code blocks in the left column implement mode m_a; the two blocks on the right implement m_b. Just like in the single-mode example, the code is free of deadlines and exception handlers for the three tasks, see [22] for more details on exception handlers. Note that, no matter which conditional branches are taken, the

execution of any block terminates within a finite number of E code instructions, i.e., the code corresponding to a mode is finite and therefore the execution of each mode instance is finite.

Generating E code, as in the above examples (with additional deadlines and exception handlers) is the result of the first, platform-independent phase of the Giotto compiler. The second, platform-dependent phase of the Giotto compiler performs a time-safety check for the generated E code and a given platform. For single-CPU platforms with WCET information and an EDF-based scheduling scheme, and for the simple code generation strategy illustrated in the example, the time-safety check is straightforward [25]. For distributed platforms, complex scheduling schemes, or complex code generation strategies, this, of course, may not be the case. The code generation strategy has to find the right balance between E code and E machine annotations, see [22] for details. An extreme choice is to generate E code that at all times maintains a singleton task set, which makes the scheduler's job trivial but E code generation difficult. The other extreme is to release tasks as early as possible, with precedence annotations that allow the scheduler to order task execution correctly. This moves all control over the timing of software events from the code generator to the scheduler. In other words, the compiler faces a trade-off between static (E machine) scheduling and dynamic (RTOS) scheduling. The strategy used in the examples and implemented in the Giotto compiler chooses a compromise that suggests itself for control applications. The generated code releases tasks and imposes deadlines according to the "logical semantics" of the Giotto source. To achieve controller stability and maximal performance, it is often necessary to minimize the jitter on sensor readings and actuator updates. This is accomplished by generating separate, time-triggered blocks of E code for calling drivers that interact with the physical environment. In this way, the time-sensitive parts of a program are executed separately [38], and for these parts, platform time is statically matched, at the E code level, to environment time as closely as possible. On the other hand, for the time-insensitive parts of a program, the scheduler is given maximal flexibility.

5.2.2 Hierarchical Timing Language

In this section we present HTL on one example, the multi-mode control program for a model helicopter of Sect. 5.2.1. In Giotto, the control task t_1 is part of both modes, since Giotto programs are flat. In contrast to that HTL, as the name suggests, allows for hierarchical models.

An HTL program is built out of four building blocks: program, module, mode, and task. A program is a set of concurrently running modules. A module consists of a set of modes and some mode-switching logic between them. Like in Giotto, each mode has a period and contains tasks. Unlike in Giotto, the periods of all tasks in a mode are equal to the mode period. Also different from Giotto, some of the tasks may be abstract tasks, schedulability placeholders for concrete tasks that may

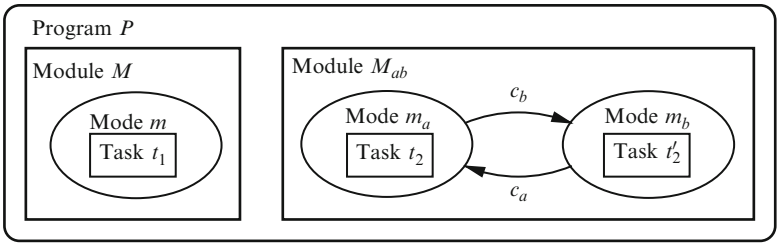


Fig. 5.6 HTL program for the multi-mode helicopter control example

refine them. If a mode contains abstract tasks, then it also specifies a refinement program which refines the abstract tasks. Moreover, another difference to Giotto is that mode switching can only happen at the end of a mode (at a period instance). This is not a restriction, since the otherwise richer structure of HTL provides the same expressiveness, as we will see in the example of the multi-mode helicopter control program. The HTL program for our example has two modules: the module M contains a single mode with period 20 ms containing the concrete control task t_1; the module M_{ab} has two modes each with period 10 ms containing a single concrete task (t_2 and t_2', respectively). The program does not involve refinement since there are no abstract modes and tasks. Figure 5.6 depicts a graphical representation of the HTL program.

In HTL, input is read from and output is written to so-called communicators which are periodic global variables. A value can be read from or written to a communicator at period instances. Communicators have periods that divide the periods of tasks using them. LET in HTL is therefore specified by the time interval between the latest communicator period instance that a task reads, and the earliest communicator period instance that a task writes to (for distributed programs this needs to be slightly adjusted for modularity [30]). Tasks are linked to communicator period instances via ports. In the example, each of the three drivers corresponds to a communicator and the driver ports are the ports used for the link. Due to the simplicity of the example, there is no need to mention the ports in the code for the HTL program of Fig. 5.6. Here is a simplified version of the HTL (pseudo) code:

```
program P {
  comm d_a(20), d_s(10), d_i(10)
  module M {
    mode m(20) {
      task t_1 in (d_i, 1) out (d_a, 2)
    }
  }
```

```
module M_ab {
  start m_a
  if m_a ∧ c_b then m_b
  if m_b ∧ c_a then m_a
  mode m_a(10) {
    task t_2 in (d_s, 1) out (d_i, 2)
  }
  mode m_b(10) {
    task t'_2 in (d_s, 1) out (d_i, 2)
  }
 }
}
```

The mode switching rules are expressed with `if-then` statements. An instruction of the form `in` (d, i) within a task instruction specifies that the task reads from the ith period instance of communicator d within the task period. In particular, $i = 1$ corresponds to the beginning of the task period. Similarly, an instruction `out` (d, i) specifies a communicator and its period instance when output is written. HTL also allows for specifying task precedences within a mode since an input port of a task may be linked to an output port of a preceding task, but our example does not illustrate this (as there are not even multiple tasks per mode).

The example program does not involve refinement so far. Therefore, for time safety one needs to check schedulability of all possible combinations of active modes in a program, which in this case amounts to two combinations: (1) mode m and mode m_a, and (2) mode m and mode m_b. Since both m_a and m_b have the same

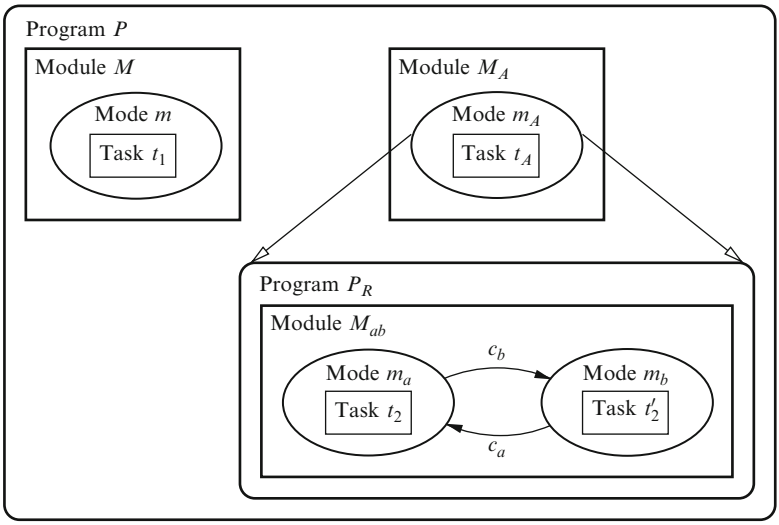


Fig. 5.7 HTL program with refinement for the multi-mode helicopter control example

timing (apart from the maybe different WCETs) and I/O behavior, they can be seen as refining a single abstract mode m_A, as in the program presented in Fig. 5.7.

The program still consists of two modules, the module M as before, and the module M_A containing a single abstract mode m_A with period 10 ms and a single abstract task t_A (with input, output, and LET equal to the ones of t_2 and t_2' and WCET equal to the maximum of the WCETs of the two concrete tasks). The abstract mode has an associated refinement program P_R with a single module M_{ab} containing two modes m_a and m_b as it was the case with the original module M_{ab}. Both tasks, t_2 and t_2' refine the abstract placeholder task t_A. Refining tasks need to have same or more LET, same or less WCET, and same or weaker task precedences [15,30]. Now time safety is guaranteed if the tasks of m and m_A are schedulable, since m_a and m_b both refine the abstract mode m_A. Here is a simplified version of the HTL (pseudo) code for the example including refinement:

```
program P {
  comm d_a(20), d_s(10), d_i(10)
  module M {
    mode m(20) {
      task t_1 in (d_i, 1) out (d_a, 2)
    }
  }
  module M_A {
    mode m_A(10) {
      abstract task t_A in (d_s, 1) out (d_i, 2)
      refinement program P_R {
        module M_2 {
          start m_a
          if m_a ∧ c_b then m_b
          if m_b ∧ c_a then m_a
          mode m_a(10) {
            task t_2 in (d_s, 1) out (d_i, 2)
          }
          mode m_b(10) {
            task t_2' in (d_s, 1) out (d_i, 2)
          }
        }
      }
    }
  }
}
```

For more details on HTL and its modular properties we refer the interested reader to [30]. E code can be generated for HTL programs in a flat way, as for equivalent Giotto programs, or in a hierarchical way. Details on HTL code generation can be found in [17].

5.3 Conclusion

We have discussed the notion of logical execution time (LET) and provided an overview of LET programming languages and runtime systems as well as some LET-inspired models of computation. We have also highlighted the key features of Giotto and its successor, the Hierarchical Timing Language (HTL), using program examples. The purpose of the rather informal presentation is to encourage the readers to study the LET paradigm further through original sources.

Acknowledgements The idea of logical execution time came up in 2000 in Thomas A. Henzinger's research group at UC Berkeley and has since been advanced by the hands of many people. Our bibliography lists quite some, but probably not all for which we apologize. We thank Eduardo R.B. Marques for his ongoing work on HTL and discussions related to it. The writing of this chapter has been supported by the EU ArtistDesign Network of Excellence on Embedded Systems Design and the Austrian Science Funds P18913-N15 and V00125.

References

1. Auerbach J, Bacon DF, Iercan D, Kirsch CM, Rajan VT, Röck H, Trummer R (2007) Java takes flight: Time-portable real-time programming with exotasks. In: Proceedings of the ACM SIGPLAN/SIGBED conference on languages, compilers, and tools for embedded systems (LCTES). ACM, New York
2. Auerbach J, Bacon DF, Guerraoui R, Spring JH, Vitek J (2008) Flexible task graphs: A unified restricted thread programming model for java. SIGPLAN Not 43:1–11
3. Auerbach J, Bacon DF, Iercan D, Kirsch CM, Rajan VT, Röck H, Trummer R (2009) Low-latency time-portable real-time programming with Exotasks. ACM Trans Embedded Comput Syst (TECS) 8(2):1–48
4. Bacon DF, Cheng P, Rajan VT (2003) A real-time garbage collector with low overhead and consistent utilization. In: Proceedings of the symposium on principles of programming languages (POPL), ACM, pp 285–298
5. Berry G (2000) The foundations of Esterel. In: Stirling C, Plotkin G, Tofte M (eds) Proof, language and interaction: Essays in honour of Robin Milner. MIT, Cambridge, MA
6. Buckl C, Sojer D, Knoll A (2010) Ftos: Model-driven development of fault-tolerant automation systems. In: Proceedings of the international conference on emerging techonologies and factory automation (ETFA). IEEE
7. Buttazzo G (1997) Hard real-time computing systems: Predictable scheduling algorithms and applications. Kluwer, Dordrecht
8. Craciunas SS, Kirsch CM, Röck H, Trummer R (2008) The JAviator: A high-payload quadrotor UAV with high-level programming capabilities. In: Proceedings of the AIAA guidance, navigation and control conference (GNC)
9. Craciunas SS, Kirsch CM, Payer H, Röck H, Sokolova A (2009) Programmable temporal isolation through variable-bandwidth servers. In: Proceedings of the symposium on industrial embedded systems (SIES). IEEE
10. Dabney J, Harmon T (2003) Mastering simulink. Prentice Hall, Englewood Cliffs, NJ
11. Derler P, Resmerita S (2010) Flexible static scheduling of software with logical execution time constraints. In: Proceedings of the international conference on embedded systems and software (ICESS). IEEE

12. Eker J, Janneck JW, Lee EA, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y (2003) Taming heterogeneity–the ptolemy approach. Proc IEEE 91(1):127–144
13. Farcas E, Pree W (2007) Hyperperiod bus scheduling and optimizations for tdl components. In: Proceedings of the international conference on emerging technologies and factory automation (ETFA). IEEE
14. Ferdinand C, Heckmann R, Langenbach M, Martin F, Schmidt M, Theiling H, Thesing S, Wilhelm R (2001) Reliable and precise WCET determination for a real-life processor. In: Proceedings of the international workshop on embedded software, vol 2211 of LNCS, Springer, pp 469–485
15. Ghosal A, Henzinger TA, Iercan D, Kirsch CM, Sangiovanni-Vincentelli AL (2006) A hierarchical coordination language for interacting real-time tasks. In: Proceedings of the international conference on embedded software (EMSOFT), ACM
16. Ghosal A, Henzinger TA, Kirsch CM, Sanvido MAA (2004) Event-driven programming with logical execution times. In: Proceedings of the international workshop on hybrid systems: Computation and control (HSCC), vol 2993 of LNCS, Springer, pp 357–371
17. Ghosal A, Iercan D, Kirsch CM, Henzinger TA, Sangiovanni-Vincentelli A (2010) Separate compilation of hierarchical real-time programs into linear-bounded embedded machine code. Sci Comp Program doi:10.1016/j.scico.2010.06.004
18. Halbwachs N (1993) Synchronous programming of reactive systems. Kluwer, Dordrecht
19. Halbwachs N, Caspi P, Raymond P, Pilaud D (1991) The synchronous dataflow programming language Lustre. Proc IEEE 79(9) 1305–1320
20. Henzinger TA, Kirsch CM (2002) The embedded machine: Predictable, portable real-time code. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI), ACM, pp 315–326
21. Henzinger TA, Kirsch CM (2004) A typed assembly language for real-time programs. In: Proceedings of the international conference on embedded software (EMSOFT), ACM, pp 104–113
22. Henzinger TA, Kirsch CM (2007) The Embedded Machine: Predictable, portable real-time code. ACM Trans Program Languages Syst (TOPLAS) 29(6):33–61
23. Henzinger TA, Horowitz B, Kirsch CM (2001a) Embedded control systems development with Giotto. In: Proceedings of the ACM SIGPLAN workshop on languages, compilers, and tools for embedded systems (LCTES). ACM
24. Henzinger TA, Horowitz B, Kirsch CM (2001b) Giotto: A time-triggered language for embedded programming. In: Proceedings of the international workshop on embedded software (EMSOFT), vol 2211 of LNCS, Springer, pp 166–184
25. Henzinger TA, Kirsch CM, Majumdar R, Matic S (2002) Time safety checking for embedded programs. In: Proceedings of the international workshop on embedded software (EMSOFT), vol 2491 of LNCS, Springer, pp 76–92
26. Henzinger TA, Kirsch CM, Sanvido MAA, Pree W (2003) From control models to real-time code using Giotto. IEEE Contr Syst Mag (CSM) 23(1):50–64
27. Henzinger TA, Horowitz B, Kirsch CM (2003a) Giotto: A time-triggered language for embedded programming. Proc IEEE 91(1):84–99
28. Henzinger TA, Kirsch CM, Matic S (2003b) Schedule-carrying code. In: Proceedings of the international conference on embedded software (EMSOFT), vol 2855 of LNCS, Springer, pp 241–256
29. Henzinger TA, Kirsch CM, Matic S (2005) Composable code generation for distributed Giotto. In: Proceedings of the ACM SIGPLAN/SIGBED conference on languages, compilers, and tools for embedded systems (LCTES), ACM
30. Henzinger TA, Kirsch CM, Marques ERB, Sokolova A (2009) Distributed, modular HTL. In: Proceedings of the real-time systems symposium (RTSS), IEEE
31. Kirsch CM (2002) Principles of real-time programming. In: Proceedings of the international workshop on embedded software (EMSOFT), vol 2491 of LNCS, Springer, pp 61–75

32. Kirsch CM, Sanvido MAA, Henzinger TA (2005) A programmable microkernel for real-time systems. In: Proceedings of the ACM/USENIX conference on virtual execution environments (VEE), ACM
33. Kirsch CM, Lopes L, Marques ERB (2008) Semantics-preserving and incremental runtime patching of real-time programs. In: Proceedings of the workshop on adaptive and reconfigurable embedded systems (APRES)
34. Kopetz H (1997) Real-time systems design principles for distributed embedded applications. Kluwer, Dordrecht
35. Lee I, Leung J, Son SH (eds) 2007 Handbook of real-time and embedded systems, The evolution of real-time programming. CRC Press, Boca Raton, FL
36. Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard real-time environment. J ACM 20(1):46–61
37. Liu J, Lee EA (2003) Timed multitasking for real-time embedded software. IEEE Contr Syst Mag (CSM) 23(1):65–75
38. Wirth N (1977) Towards a discipline of real-time programming. Comm ACM 20:577–583
39. Zhao Y, Liu J, Lee E.A (2007) A programming model for time-synchronized distributed real-time systems. In: Proceedings of the real-time and embedded technology and applications symposium (RTAS), IEEE, pp 259–268

Part II
Connecting Theory and Practice

Chapter 6
Improving the Precision of WCET Analysis by Input Constraints and Model-Derived Flow Constraints

Reinhard Wilhelm, Philipp Lucas, Oleg Parshin, Lili Tan, and Björn Wachter

6.1 Introduction

6.1.1 Timing Analysis of Embedded Systems

Hard real-time embedded systems are subject to stringent timing constraints. The proof of their satisfaction requires upper bounds on the *worst-case execution time (WCET)* of tasks. This requires taking into account properties of the software, such as potential control flow, loop bounds and maximal recursion depths, as well as of the hardware, such as the state of caches or pipeline units. Therefore it is extremely hard to derive sound upper bounds by measurement-based approaches [30].

Static timing analysis guarantees safe upper bounds on the WCET derived from an over-approximation of the set of possible program executions. The derivation of execution-time bounds for programs is a bottom-up process starting with the determination of execution-time bounds for instructions and basic blocks. To bound the execution time of instructions, which can vary depending on the execution state of the execution platform, static analysis computes invariants about the set of possible execution states at all program points. The execution times of basic blocks are then combined into the overall WCET bound according to the control-flow structure of the program, taking loop bounds and other feasibility constraints into account.

Methods based on static analysis and the corresponding tools have proved successful in industrial practice. They are in routine use in parts of the avionics and the automotive domains [1, 39]. However, the embedded systems in these domains have characteristics that can be exploited to further improve the precision of timing analysis.

R. Wilhelm (✉) · P. Lucas · O. Parshin · L. Tan · B. Wachter
Compiler Design Lab, Universität des Saarlandes, Saarbrücken, Germany
e-mail: wilhelm@uni-saarland.de; wilhelm@cs.uni-sb.de; phlucas@cs.uni-sb.de; oleg@cs.uni-sb.de; lili@cs.uni-sb.de; bwachter@cs.uni-sb.de

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_6,

Embedded control software is often implemented as a task reading from inputs, computing reactions to these inputs, thereby transforming its internal state, and writing outputs. This task is periodically or sporadically invoked. Static timing analysis is applied to determine a bound on the execution time for any task invocation under all possible circumstances.

Besides the execution-time differences arising from different states of architectural components, e.g., contents of caches or occupancy of execution units, another variation appears at the level of the software, e.g., loop iteration bounds or input-dependent control flow. In general, this concerns the *control flow* within the task. A conservative analysis needs to take all paths through the control-flow graph into account that it cannot safely exclude. Although there exist provisions to recognise dead code or to derive mutually exclusive branches, the peculiarities of embedded systems enable the analysis to work with a restricted and thus more precise view of the possible control flow.

One such restriction is due to *operating modes*. Briefly speaking, an operating mode defines a particular behaviour of the overall system and its constituting tasks. For example, a task may reside in a startup-mode until the operating temperature reaches a certain level and then work in normal operational mode. Thus, the possible control flow is restricted in each mode and the task might have different timing *properties* in these modes, which have to meet potentially different timing *constraints* imposed upon it from the environment. An analysis not oblivious to operating modes is able to provide several mode-specific WCET bounds, as opposed to one conservative bound for all possible modes. Current timing analyses compute a safe overall upper bound and do not on their own specialise this information.

Another restriction of the control flow can be derived by *examining the models* from which code is synthesised. The high-level information available from the model helps to analyse its development over time (*inter-run* analysis), whereas timing analysis of a single task invocation (*intra-run* analysis) is confined to a safe over-approximation of the set of states at invocation time. Such a high-level analysis reveals logical relations between branching conditions, in particular when the highly regular structures of automata (e.g., Stateflow) are used to model control logic. If a dedicated analysis of automata finds out a constraint such as "If automaton A is in state a_1, then automaton B cannot be in state b_2.", this constrains the possible logical relations of conditions in the surrounding part of the control system (e.g., Simulink): automata states govern the control logic, after all. These relations translate into knowledge about infeasible paths, again restricting the control-flow graph. Relations which need inter-run analysis are out of the scope of traditional intra-run timing analysis.

6.1.2 Overview of This Chapter

Conventional WCET analysis is neither aware of operating modes nor of the specific characteristics of model-based code. In this work, we show

- What kind of information is *useful*
- How this information can be *gathered* and
- How it can be *transferred* to the timing analysis tool.

For operating-mode analysis, we propose a semi-automatic procedure which heuristically identifies operating modes from C code. This entails not only the various restrictions on the control flow, but also the predicates on input variables giving rise to these paths. We lay out different ways of exploiting mode information in timing analysis and explain how the mode-specific WCET bounds can be used in scheduling.

To specifically improve the timing analysis of code synthesised from models, we introduce techniques that leverage model information to derive infeasibility constraints in the control-flow graph. To this end, we develop analyses at the model level, which derive relations between branching conditions. This entails dedicated analyses for different types of model subcomponents including Stateflow automata and Simulink components.

We also show how to use results from one analysis in the other, exploiting synergies to increase precision even further.

6.1.3 Contents of This Chapter

Section 6.2 provides the technical background on timing analysis. Sections 6.3 and 6.4 form the main part of this paper, explaining our work on improving the WCET analysis of programs with operating modes and programs synthesised from Simulink/ Stateflow. These sections flesh out the ideas and techniques mentioned in Sect. 6.1.1. Section 6.5 explains differences and synergies between the approaches. Section 6.6 gives an overview of related work, and Sect. 6.7 concludes the paper and gives an outline on future work.

Parts of this work have been published in [26, 34].

6.2 Timing Analysis

Embedded systems nowadays employ modern high-performance processors with features such as caches, pipelines, and speculation. The execution time of embedded software therefore exhibits a significant variation depending on the hardware state. For example, the execution time of an instruction that requires a memory access can vary by at least two orders of magnitude depending on whether it results in a cache hit. For that reason, timing analysis has to take the development of the system state during the actual execution of the compiled program into account. External factors such as input data and interference with other system parts (e.g., side-effects of other tasks) introduce further variability. In general, the state space of input data, initial

state and effects from interference is too large to exhaustively explore all possible executions to determine the exact worst-case execution time. Instead, the crux is to represent all possible executions symbolically and in a compact way by abstract interpretation, e.g., to statically predict cache contents [7]. By this process we derive sound but precise *bounds* on the WCET. We now give an overview of static timing analysis; for more about concepts and tools, see [40].

6.2.1 Timing Analysis Framework

Over the last several years, a more or less standard architecture for timing-analysis tools has emerged [4, 17, 37] as depicted in Fig. 6.1. The architecture is divided into the following phases:

Control-flow reconstruction [35] takes a binary executable to be analysed and reconstructs the program control flow.

Value analysis [2, 31] computes an over-approximation of the set of possible values in registers and memory locations by an interval analysis and/or congruence analysis. This information is among others used for a precise data-cache analysis.

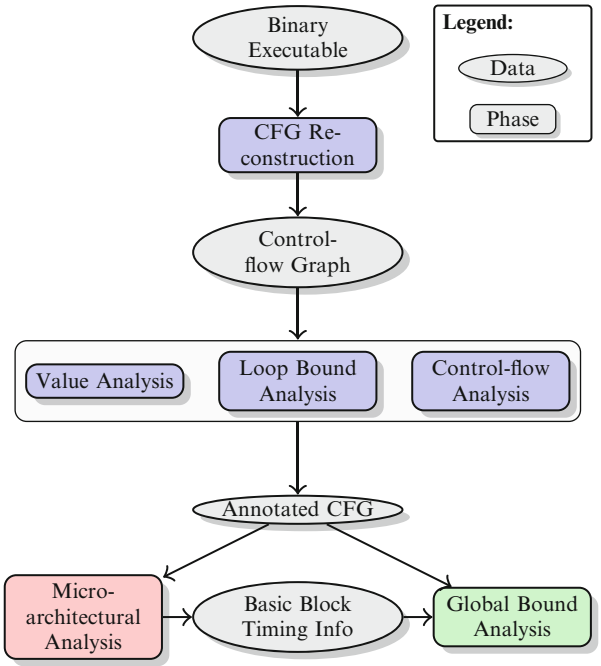


Fig. 6.1 Standard architecture of timing analysis tools

Loop bound analysis [5,16] identifies loops in the program and tries to determine bounds on the number of loop iterations, information indispensable to bound the execution time.

Control-flow analysis [5,32] narrows down the set of possible paths through the program by eliminating infeasible paths or to determine correlations between the number of executions of different blocks using the results of value analysis. These constraints will tighten the obtained timing bounds.

Micro-architectural analysis [3,8,38] determines bounds on the execution time of basic blocks for a specific processor. It takes architectural details such as the pipeline, caches, and speculation into account.

Global bound analysis [25,36] finally determines bounds on execution time for the whole program. To this end, this phase uses the results of microarchitectural analysis, namely upper bounds on the execution times of basic blocks, and results from control-flow and loop bound analysis. Algorithmically, the analysis determines a longest path through the control-flow graph of the program based on basic block execution-time bounds. The control flow and loop bounds are expressed according to implicit path enumeration in an integer linear program. The target function computes an upper bound on a path's execution time, and the optimal solution is the WCET bound.

The commercially available tool `aiT`[1] by AbsInt implements this architecture. It is used in the aeronautics and automotive industries and has been successfully used to determine precise bounds on execution times of real-time programs [8,9,18,33,39].

6.2.2 Annotations

There are different ways to adjust and improve the precision of timing analysis. Here we focus on the control-flow graph (CFG), which guides the different analysis phases. The CFG may contain paths that do not correspond to any possible execution of the system, also called *infeasible paths*. Figure 6.2 shows an example program with its control-flow graph. Although the fragment does not contain dead code,

```
if(a>0)
  long_calculation1();
else
  short_calculation1();

if(a<=0)
  long_calculation2();
else
  short_calculation2();
```

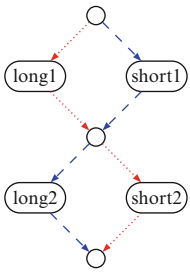


Fig. 6.2 C code with infeasible paths and its control-flow graph. Only the dashed blue path and the dotted red path are feasible

[1] cf. http://www.absint.de/wcet.htm

only two of the four statically possible paths are feasible because of the correlation between the branch conditions `a>0` and `a<=0`. Without interpreting the branch conditions, all four paths through the graph have to be considered, leading to an over-approximation of the WCET.

Correlations between branch conditions as in the example can be used to exclude certain infeasible paths, which improves tightness of timing analysis. In previous work, these correlations have been discovered at different levels: directly from the executable [32], in C code [15], or in Esterel specifications [20]. Technically, these correlations can be used as additional constraints, so called *flow constraints*, in the ILP formulation. Let l_1, l_2 and s_1, s_2 be the variables corresponding to the execution counts of the blocks with the long/short calculations in Fig. 6.2, respectively. The feasibility information would then be expressed by additional constraints to the implicit path enumeration: $l_1 = s_2$ and $s_1 = l_2$.

Flow constraints will be used in two ways in this work. In the case of operating modes, we *specialise* the timing analysis to certain paths by declaring control flow paths as infeasible for a particular analysis and thereby derive different WCET bounds for different sets of possible paths. In the case of Simulink analysis, we compute *additional constraints* on the feasibility which cannot be derived as easily as the ones in Fig. 6.2 without access to the model.

Another way of influencing the timing analysis is by starting at the value analysis phase. The analysis needs to start with a superset of the initial values of registers and memory locations. Although the values of initialised memory cells are known at start-up time, these may change over time. Timing analysis has to work with an over-approximation of all possible states at the task entry point during the execution of the main loop.

Thus, annotations on the ranges of values in memory or registers are necessary for safety of the analysis. But equally well, they can be employed to improve precision. For example, if it is known that due to physical restrictions an input variable will take on values only within a certain interval, this can be specified and will influence all of the following parts of the timing analysis. For specialisation to certain inputs such as those defining operating modes, starting by annotations at the value analysis phase is also a natural way.

6.3 Input Constraints: Operating Mode Analysis

6.3.1 Motivation

In the most general sense, we speak of *operating modes* if a software can exhibit different behaviour in different circumstances and if this behaviour is determined at runtime. As a trivial example, consider a mode for normal operation and an exceptional (error) mode which is triggered if a sensor fails. Because the sensor failure status is not statically fixed, the same piece of software may enter both modes during its running time. In this sense, a dynamic mode is defined by a *constraint on*

input variable valuations: If the variable denoting sensor input is in range $[0, \infty)$, the software is in normal mode; if the variable has the value -1, the software is in error mode. As such, mode-specific analysis is a kind of *refinement of context*: The context of the mode-specific analysis is defined by the applicable set of possible inputs.

This shows one usage for modes in timing analysis: *Within a single task*, different pieces of code are executed depending on the operating mode. Mutual exclusivity of modes translates into specific subsets of code executed per mode, and thus a WCET analysis can give *different* mode-specific WCET bounds, in contrast to a conservative overall bound; indeed, the maximal mode-specific bound on the execution time may even be lower than the conservative overall one due to less opportunities for loss of information arising from over-approximation.

This is useful and even necessary because tasks also may have different timing *requirements* in different situations. Consider a task which controls the fuel injection. Because it is triggered by the crankshaft sensor, the deadlines are stricter at higher RPMs (revolutions per minute). If the task would uphold its precise computation which meets the deadlines up until some RPM threshold, e.g., 4,000, it would miss the deadlines above that threshold, so it has to switch to a less computationally-demanding mode. If only one overall WCET bound were considered, then the task would become formally unschedulable after 4,000 RPM, which is not the case. Thus, the task has a WCET motivated mode distinction, and this needs to be identified and used in the analysis to ensure success of schedulability analysis.

Another use of modes for timing analysis comes from considering not solely modes within a single task, but from the combined mode-specific behaviour of a set of tasks. If one task needs to perform more computations in start-up mode (e.g., initialisation of some system), and the other one in run-time mode (e.g., monitoring some parameters), then their overall WCETs are mutually exclusive. Combining these WCETs would unnecessarily restrict schedulability analysis. Generally speaking, the modes of the tasks are not occurring in arbitrary combinations, but form a *global mode* determining the behaviour of the complete system. Figure 6.3 gives an example of this usage of modes.

In this section, we report on work to semi-automatically identify modes which are relevant for WCET analysis within a single task. The result of this identification is a set of mutually disjoint ranges of the input variable valuations. Here, we confine ourselves to the mode analysis and merely note that the results are of value to scheduling tools and may be transferred to them [21].

6.3.2 Heuristical Mode Analysis

Deriving modes automatically is non-trivial. The main problem is that a static analysis can only *heuristically* figure out mode *candidates*. Designers and engineers of the software would naturally attribute certain modes to the system: Some

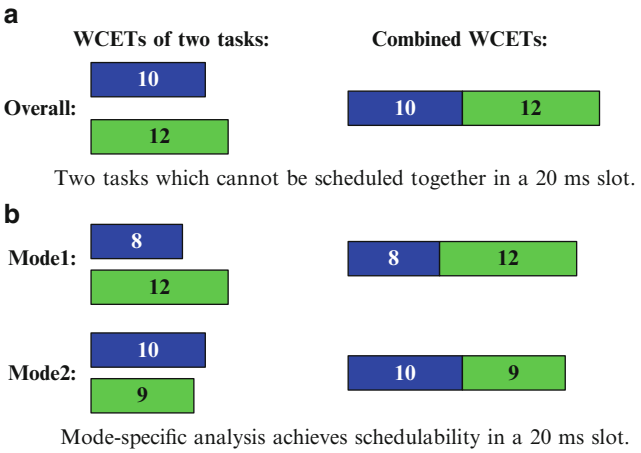


Fig. 6.3 Operating modes

pre-conditions for differing behaviour of a set of tasks are mutually exclusive, or the behaviour of a single task is largely determined by a small number of state variables. Examples for such modes are *machine initialisation* or *sensor failure*.

In the absence of explicit mode specifications, it might be unknown precisely which conditions are sufficient for a mode or which are guaranteed to hold, and it is up for interpretation what actually constitutes a mode. Thus, modes as known to the system creators have to be *approximated* by an analysis of the system.

Because of this inherent fuzziness, a fully automated process cannot analytically derive modes which precisely capture the notions of modes of all people involved. What is a mode to one person is an arbitrary combination of input variables to another one. But if the heuristics arrives at a mode which is useful but unknown to the designers, i.e., a *hidden mode* that is neither present by design nor an obvious artifact from the physical environment of the software, this improves knowledge about the software and thereby is useful itself.

6.3.2.1 General Idea

Consider a parking assistant. If the vehicle speed is below a certain threshold, ultrasonic sensors are evaluated to measure the distance to obstacles. Let the code governing that be implemented by a conditional:

```
if(speed<THRESHOLD){
  complicated_calculation();
} else {
  return;
}
```

Heuristically, this is a candidate for a switch depending on an operating mode, because it significantly influences the program's computation. Guarding that conditional is the value of the input variable `speed`: Value ranges $(-\infty, t)$ or $[t, \infty)$, where t is the value of a compile time constant `THRESHOLD`.

A static analysis has to trace back this causality chain from the source code. Thus, the rough idea of mode estimation is:

1. Find out which conditionals are likely to govern modes.
2. Find out which input variables control these conditionals.
3. Find out which valuations on the input variables give rise to the varying evaluations of the conditional.

We briefly describe these general ideas in the following.

6.3.2.2 Mode Conditionals and Mode Variables

The question "When does a conditional govern a mode?" can be answered only heuristically. Would one regard each and every conditional as signifying a mode, this would lead to an absurd diversification of modes, where each feasible path through the program corresponds to one mode. Thus, we need to establish a concept of *importance* for conditionals.

We call mode-governing conditionals *mode conditionals* and the input variables influencing their evaluation *mode variables*. Heuristics for *mode conditionals* include, but are not limited to, the prediction of their impact on execution time, as statically derived from the source code. (We shall explain later why source code analysis is more useful than analysis of the binary.) This heuristics is not only pragmatically motivated, but also serves well to approximate operating modes: It is more likely that a conditional influencing a significant part of the program would be regarded as signifying a mode than a conditional choosing among only slightly different arithmetic instructions. The impact of a branch may be estimated simply by examining the length of the statement sequence or by checking for the presence of certain patterns or calls:

- The familiar implementation of a function body being guarded completely by an `if`-statement would lead to a significant difference in implementation.
- Occasional loops are present in the implementation of lookup tables with binary or linear search. The conditional execution of such structures also hints at mode-dependent behaviour.
- External functions may be annotated by the user if they implement mode-significant behaviour. For example, such functions may be those that communicate with external devices.

The approach can be summarised as checking for unbalanced branches. All in all, basic blocks have certain *weights*, for example, corresponding roughly to the execution time, and a conditional with highly unbalanced branches is regarded a mode conditional.

Heuristics for *mode variables* involve:

- If a variable is used for control in largely disparate parts of the source code, this hints to its use as a mode variable.

- Naming conventions may enforce special names for mode variables.
- User annotations can specifically identify mode variables.

In all cases, it is furthermore possible to exploit *synergies* with higher levels of the tool chain. On one hand, conditionals which are already present in the high-level models, such as explicit switch blocks, are more natural candidates for mode conditionals than `if` statements arising as artifacts of the code generation process. On the other hand, naming patterns can make the identification of state variables easier, if one assumes that automata states are used to encode modes.

Determination of mode conditionals and mode variables is interweaved: Mode conditionals lead to mode variables by backward slicing, and mode variables lead to mode conditionals by forward slicing. Thus, an iteration of the slicing phases leads to an extension and refinement of mode determination.

If we now disregard those conditionals which are no mode conditionals and abstract them out of the control-flow graph, we arrive at the statically possible set of *significantly different* paths through the reduced control-flow graph, and for each one the set of mode variables determining whether it is taken. What remains to be done is to *cluster* the paths into coarser modes, to determine which state of the environment leads to which of those clusters and to pass this information to the timing analysis tool.

6.3.2.3 Mode Determination

Let us consider a single path in the reduced control-flow graph. By backward interpretation from the mode conditionals to their governing mode variables (in the complete program), one can gain a specification of the valuations of these variables giving rise to the one or the other choice (e.g., [14, 19]). Even in the absence of perfectly precise information, both *must* and *may* information is useful: An *under-approximation* of the input state, what we have considered before in our examples, ensures that this valuation definitely gives rise to the behaviour as determined by one path; an *over-approximation* ensures that at least some paths are not taken in this state.

The use of approximations comes into play in particular in the optional post-processing phase of *clustering*. Clustering can improve the usability of mode analysis and help better distinguish those modes which are pertinent to the user at hand. For usage in timing analysis, for example, not all modes have to be treated differently: If their predicted WCETs do not differ significantly, they can be handled together. To this end, the paths in the reduced program can be assigned a variety of properties to be used for classification into modes. Such properties include:

- The very control-flow choices defining the paths
- The read and write accesses to communication variables
- The static length, including presence of loops
- Calls to specified external functions

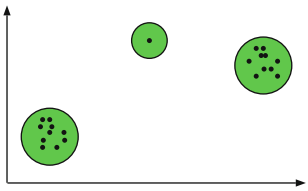


Fig. 6.4 Clustering

More properties corresponding to different heuristics can be added into this framework. For a discussion of possible heuristics, see also [23].

To ensure that paths can be compared to each other, we need these properties to be comparable. If the comparability is ascertained, the paths can be sorted into *clusters* by a gravitational clustering algorithm (like nearest neighbour classification) [29].

For example, with only two linear dimensions, consider the situation in Fig. 6.4. By visual inspection, three natural candidates for modes present themselves. A similar operation can be computed by regarding the points in the multidimensional space as bodies gravitating to each other and setting the attraction across the dimensions to allow the bodies to gravitate to certain points.

This approach has the benefit that sensitivity to certain properties can be changed by fiddling with the gravitational constants. For example, if a certain conditional is already known to be mode defining, then the clustering can be made to not let paths which contain different control-flow choices at that point cluster at all; or the sensitivity between path length and communication patterns can be adjusted by balancing the respective constants. In this way, the importance of modes can be established with respect to other criteria such as the communication fabric. Furthermore, the clustering can also directly be used for *sub-mode* determination, by running the clustering step again for the set of paths forming a mode with different gravitational constants.

Deducing constraints on the input variables giving rise to the paths of a cluster is more involved, however. Although it is reasonably easy to specify the clusters by exhaustive enumeration of properties and property combinations, be it an under- or an over-approximation, this is not likely to be a relevant information for the users. Future research needs to investigate whether existing approaches yield sufficient results in our setting.

6.3.3 Usage of Mode Information

With several modes identified, the WCETs specific to each mode need to be computed. This is done with the help of annotations (Sect. 6.2.2). The obvious way to do so is to conduct several timing analyses with the inputs specifically fixed according to the modes – a costly process. A second possibility is to only solve different ILPs for deriving the overall WCET bound from the results for the basic blocks. In this way, flow constraints would be generated from the mode governing

conditionals. These two approaches thus enable a choice between precision and speed.

Thirdly, *trace partitioning* [27] can be a means for implementing mode analysis in another way. Partitioning according to a specific mode leads to a separate instance of timing analysis for each mode. This yields not only a special execution time bound for each mode, one also specialises the whole of the analysis to the mode from that point on. If several mode conditionals are congenerous, trace partitioning takes care of exploiting this similarity automatically. Mode analysis thus provides the split points for trace partitioning and the information to exploit the different results.

The reader may now ask why the mode *determination* works on the source code level whereas the *exploitation* uses binary analysis. It has been motivated in Section 6.2 that accurate WCET analysis can be conducted only on the binary level due to the paramount importance of considering the hardware state. Although in principle mode determination could be performed on the same level, the benefits would be negligible and could be dwarfed by the problems. We have already established that the modes resulting from the analysis shall approximate the natural notions of operating modes. Anything which is not expressible in terms of the source code or the model is of very limited use to the programmers. For example, if a mode arises purely because one branch of a conditional contains a memory access modifying the cache in such a way that subsequently there is a larger amount of cache misses, this mode is unlikely to be of any interest or use to the user of the tool.

We stress that confinement to source-level mode analysis cannot provide *wrong* or *worse* timing analysis results; it might merely fail to achieve better WCET estimates.

For the usage of mode information together with the analysis of synthesised code, the reader is referred to Section 6.5, which explains synergies after the nomenclature and the analysis for Simulink models have been explained.

6.4 Flow Constraints: Simulink/Stateflow Analysis

Today, embedded systems are predominantly developed using model-based design tools such as Matlab Simulink/Stateflow. Current timing analysis tools analyse compiled executables. Model information is not leveraged. We present initial results with the exploitation of Simulink/Stateflow models showing that timing analysis can benefit significantly from model information in terms of both automation and precision.

6.4.1 Matlab Simulink/Stateflow and Generated Code

Simulink/Stateflow is a hierarchical modelling language for control software with a sequential, imperative semantics. The underlying methodology is to design control computation within Simulink and control logic within Stateflow. Simulink

offers building blocks for proportional, integral and differential (PID) control computations and estimations, e.g., filters, look-up tables, and arithmetic operators. Stateflow is an automata specification language that can be used to express transitions between different system states. Blocks communicate with each other via signals and receive external inputs from the environment. For deployment, code generators generate production C code, in which the internal states of Stateflow and Simulink blocks are encoded by state variables. Signals and internal inputs also map to C variables. The standard code generators ensure that the implementation of blocks can be traced in the source code.

Model information can already quite directly influence the timing analysis. For example, for input variables which might change during execution, a *volatile* annotation can be derived automatically, ensuring the safety of the analysis; variable ranges also can be generated automatically [10]. Here, we concentrate on more involved model analyses improving precision of timing analysis.

6.4.2 Problem Statement

We investigate where precision is lost due to infeasible paths. To this end, we focus on typical patterns at the level of the model that lead to infeasible paths by analysing the inter-run development of a model's implementation. As an example, we consider the fuel-rate controller which is a Simulink/Stateflow demo model that contains typical features of embedded controllers. The controller estimates airflow rate, and calculates the fuel injection rate based on PID control principle.

We analysed the controller with the `aiT` WCET Analyser. `aiT` produces a worst-case path to explain the execution time bound it has computed. Without providing flow constraints, the execution time is over-approximated and the computed worst-case path is infeasible. The reason is that static timing analysis is not aware of certain dependencies in the model. Internal states and signals received from the environment are often in some logical relation, e.g., they exclude or imply each other. Depending on the current state, signals, and their logical combinations, different look-up tables or computations are triggered.

As discussed in Sect. 6.2, the timing analyser generally does not interpret conditions. Hence it has to take the longer branch of a conditional, even if execution history of the path does not admit so. As a result, the worst-case path resolves branches spuriously: it "switches" between operating states where this is not possible in an execution of the program.

We illustrate some typical spurious resolutions of conditions on the worst-case path. Some resemble the infeasible-path example in Section 6.2.2, e.g., they involve conditions like `mode==LOW` and `mode==RICH`. Other conditions are more involved. For example, the condition `O2_fail==0 && mode==LOW` checks if the oxygen sensor is operating correctly and the system is in `LOW` mode, while condition `pressure_fail==1` checks if the pressure sensor has failed. Due to analysis of the control logic implemented in an Stateflow automaton, we can

derive that these seemingly unrelated conditions are, in fact, mutually exclusive: The Stateflow automaton would not set `mode` to `LOW` if any sensor had sent a failure signal. Such *entailed relations* need to be derived by analysing the model semantics. In the source code or executable, dependencies are more implicit and even harder to track than in the model.

In this example, we see how the role of modes in this section relates to the concept of *operating modes* in Sect. 6.3. Mode analysis on C code might find out that the different conditions are significant and might order them into different modes according to the variable values, but could not derive how these conditions *relate*. Only by interpretation of the possible transitions of the system become the relations between the model states and thereby of the mode variables visible.

6.4.3 Deriving Flow Constraints

In the following, we show how to construct flow constraints from the model to achieve a more precise timing analysis.

We aim at conditions that determine whether a piece of the model is executed. These conditions on external inputs, and internal signals, e.g., states, guard signal transformation and control computation. In Simulink/Stateflow, this is expressed by conditional blocks, similar to conditionals in C, e.g., triggered and enabled subsystems, guarded transitions in Stateflow and switch-blocks. We uniformly refer to the conditions as *trigger conditions*.

6.4.3.1 Flow Constraints from Definition-Use Dependencies

We formulate flow constraints that relate a definition, e.g., a state variable, and uses of that variable. Certain definitions always make a trigger condition false. Trivially, a program execution cannot pass through such a condition *and* the branch guarded by the trigger condition. This can be expressed by flow constraints. One example for such constraints in the fuel-rate controller are signals that indicate a failure of a sensor. These signals are set in a Stateflow block and are used in a Simulink block to trigger the evaluation of a lookup table.

6.4.3.2 Flow Constraints from Correlations between Trigger Conditions

Relations between trigger conditions can be formulated as flow constraints, e.g., independent, equivalence, implication, antivalence, and exhaustion can be expressed. To be effective, entailed relations need to be considered. The analysis of entailed relations requires information about deep semantic properties of Stateflow and Simulink blocks. To this end, we anticipate that relational abstract domains from static analysis may be helpful.

6.4.3.3 Significant Branches

Eliminating infeasible paths does not per se improve precision. For example, if branches of conditionals have approximately the same execution time, there can be little gain in precision by eliminating an infeasible path that takes the 'wrong' branch. Therefore, similarly to mode analysis on C code, we focus on *significantly* unbalanced branches. For example, the invocations of look-up tables and state-dependent discrete filters give rise to such branches. In general, determination of infeasible paths pays off more in the Simulink part of a model than in Stateflow.

In contrast to the analysis in Section 6.3, however, here we are hindered by the fact that the code generator adds an additional step between the model and the eventually executed code. If an expensive block is used as an input of two different switches, for example, the code generator may schedule the evaluation of the block exactly once before the switches are evaluated. The flow constraints becomes useless in this case. Thus, although the *derivation* of constraints on the model is independent from the code generator, their *applicability* depends on its peculiarities.

6.4.3.4 Experimental Results

Initial results with the fuel-rate controller are promising. For the fuel-rate controller, we have formulated the flow constraints above according to the `aiT` timing specification language. In the first stage, we considered flow constraints from definition-use dependencies which reduced the execution time bound by 4%. Adding both kinds of flow constraints yields an overall reduction by 19% and a feasible worst-case path. If we give an execution-time bound for each internal mode, we achieve a reduction between 20% and 48%. In future work, we will automate the generation of flow constraints and apply our approach to industrial examples.

6.5 Comparison of the Approaches

Let us recapture the basic strengths and the differences between the approaches presented in Sects. 6.3 and 6.4 and then point out possible synergies between them. Figure 6.5 graphically illustrates the improvements of WCET analysis by both approaches. The first case in Fig. 6.5 shows the WCET bound computed by a state-of-the-art analysis without additional path or input constraints.

As can be seen in the second case, in the Simulink analysis, we strive to compute a *single*, but more *precise* WCET bound, by exclusion of infeasible paths. In the analysis of *operating modes*, we strive to compute a *multitude* of WCET bounds, by *specialisation* to particular inputs. The third case in Fig. 6.5 shows this. Note that it is possible for the operating mode analysis to compute a lower overall WCET bound, but this cannot always be guaranteed.

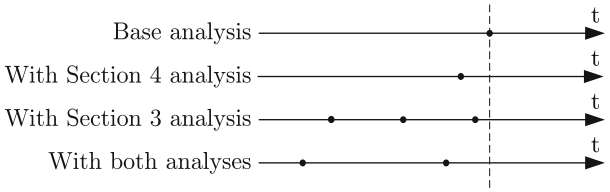


Fig. 6.5 Comparison of mode analysis and Simulink analysis to improve WCET computations. Dots on the time axes represent computed WCET bounds, the dashed line represents the single overall WCET bound for reference

The fourth case shows a possible result after applying both approaches together. Note the improved WCET bounds of every operating mode and also the absence of the second mode: by using additional information extracted from the model by the Simulink analysis the operation mode analysis can automatically exclude some infeasible modes (e.g., using constraints on Stateflow automata). This is one example of possible synergies between the approaches.

Another difference of the approaches needs to be stressed. Whereas the operating mode analysis aims to compute *meaningful* operating modes, this need not be the case in Simulink analysis: A flow constraint which arises from some arbitrary restriction of states is fine, as long as it rules out certain infeasible paths.

The general overview of both approaches and the information flows are depicted in Fig. 6.6. As we can see, the operating mode analysis can profit from the additional information gained by the feasibility analysis: some modes could be excluded because of infeasibility (the last example in Fig. 6.5); or some mode variable candidates can be provided by model analysis, which works as a starting point for mode determination. In the other direction, C code analysis can find out significantly unbalanced branches. This enables the Simulink analysis to take the peculiarities of the generated code into account, if that specialisation is so desired.

6.6 Related Work

6.6.1 Analysis of Synthesised Code

The approach presented here derives flow constraints from the model. Previous work on flow constraints focused on the executable [32], or C level. In [20], the authors consider timing analysis of code synthesised from Esterel. They identify flow constraints to eliminate infeasible paths. The principal ideas concerning the two kinds of flow constraints are related, however Esterel is significantly different from Matlab Simulink/Stateflow. Hence rules to derive flow constraints differ significantly.

[24] describes early work on timing analysis for Simulink models in general. They do not consider the correlations arising from Stateflow, but add annotations

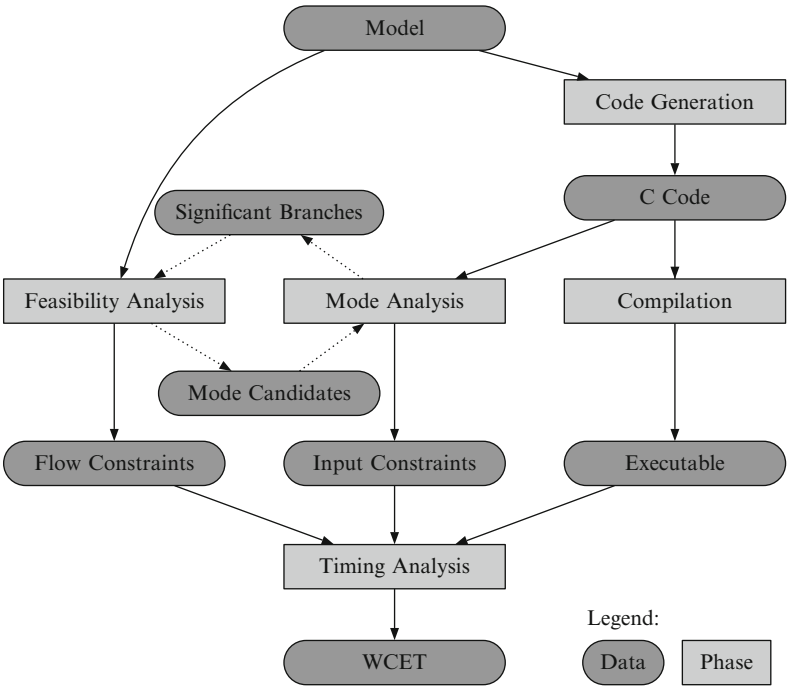


Fig. 6.6 Overview of the two approaches. Dotted arrows denote synergies

for basic Simulink blocks. Model information such as loop bounds is passed to the underlying timing analysis tool. The authors modified the code generator to generate annotated C code which is then processed by a compiler based on GCC. The mapping process ensures that the computed timing information can be attributed to the model blocks. Resulting annotations such as bounds for simple counting loops could be inferred automatically by modern analysis tools, however.

Integrations of `aiT` with ASCET and SCADE are described in [10, 11], respectively. Apart from providing feedback about the behaviour of the generated code to developers, the integration also improves precision by passing additional information from the modelling tool to aiT. This concerns, e.g., variable ranges, maximal depths of searches through tables or other loop bounds. [23] shows how to pass information from ASCET-MD models to the timing analysis tool, providing, e.g., information about the values of calibration parameters. They also consider mode-dependent timing analysis (see below).

6.6.2 Operating Modes

Operating modes have been studied before in the literature, with differing foci and differing definitions of what constitutes a mode. For a discussion of various notions of modes, see [28]. Apart from a body of literature on specifying and using

operating modes, there are several papers concerned with *derivation* of task-level modes.

[22, 23] fall into the intersection of mode analysis and model analysis: The authors have developed a tool to semi-automatically derive operating modes from ASCET-MD models. A mode is defined as a semantic context information influencing software behaviour or performance; more concisely, "an interesting set of states", where the definition of "interesting" varies among the stakeholders. Because of this, similar to our approach, various heuristics are employed to arrive at important modes, taking properties such as syntactic patterns, naming conventions and differences of measured execution times into account. The modes are then used to visualise mode-dependent signal flow and also for mode-dependent timing and schedulability analysis by specific annotations.

In [19], modes are derived from C code. The authors identify semantically feasible paths and must-preconditions on input variables for these paths and consider these different paths as modes for which different WCET bounds are calculated. They do not attempt to filter according to the significance of conditionals or to cluster the paths according to importance. Instead, modes are used to exploit the call context information of functions. For simple processors where the execution time of an instruction depends only on its operands, they also proceed to derive symbolic expressions for mode-specific WCETs.

In [12], those invocations of a component which lead to similar execution times are grouped into modes (*clusters*, in that paper's terminology). Starting from one cluster representing the complete input space and the overall BCET and WCET bounds, clusters are subdivided in a blind search process, for as long as these bounds are too different and the number of clusters is not too large. Due to the (very costly) full $^{W}_{B}$CET analyses performed during the search process for each cluster candidate, the result is tied to a particular architecture. Furthermore, without adequate heuristics guiding the search process, the usability of the resulting clusters itself is unclear. In [6], the input space is divided in order to explicitly find the worst-case input. This restricted setting allows for some optimisations in the search process.

In [13], scenarios, which roughly correspond to our modes, are derived from C code. To this end, the influence of an input variable on the WCET is estimated by considering its uses in control-flow splits, similar to our unbalanced branches. For input variables with large predicted influence on the WCET, multiple scenarios are derived, each corresponding to particular valuations. These valuations are derived by splitting the input space according to the relations in which that variable is used in the source code. Then, the source code is instantiated for each scenario anew, and the WCET estimation phase can proceed on each scenario's reduced program.

Related to (dynamic) operating modes is the notion of *static* modes: Modes which are fixed at creation time of the software as well as those that are fixed at initialisation time, such as calibration parameters. In a program, these correspond to parameterisation via preprocessor directives and source level constants, and to initialisation of constants from constant memory in a start-up phase, respectively.

Both the identification of modes and their usage alluded to in this paper work likewise for such static modes.

6.7 Conclusion

We have shown that the information about operating modes of embedded software as well as available model information can be exploited to refine the way control flow is used in the timing analysis of hard real-time systems. Automata as present in model-based design and operating modes partition the control flow. Timing analysis of the different parts of these partitions yields more precise results than an upper bound computed over the whole unpartitioned control flow. The same information can be used to eliminate infeasible paths, which could contribute to the over-approximation of WCETs. The knowledge about restrictions of the control flow is communicated to the timing-analysis tool in the form of annotations.

We are currently developing a method and a tool to semi-automatically identify operating modes, which are not made explicit neither in models nor in hand-written code. The flow constraints which are currently derived manually from Simulink/Stateflow models will be derived automatically. The final goal is a highly precise, mode-specific and model-aware timing analysis.

Acknowledgements The research leading to these results has received funding from the following projects (in alphabetical order): The European Network of Excellence *ArtistDesign*, the Deutsche Forschungsgemeinschaft in SFB/TR 14 *AVACS*, the ITEA 2 project number 06042 (*ES_PASS*), and the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement number 216008 (*Predator*).

References

1. Byhlin S, Ermedahl A, Gustafsson J, Lisper B (2005) Applying static WCET analysis to automotive communication software. In: Proceedings of ECRTS, pp 249–258
2. Cousot P, Cousot R (1977) Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of POPL, pp 238–252, DOI http://doi.acm.org/10.1145/512950.512973
3. Engblom J (2002) Processor pipelines and static worst-case execution time analysis. PhD thesis, Uppsala University
4. Ermedahl A (2003) A modular tool architecture for worst-case execution time analysis. PhD thesis, Uppsala University
5. Ermedahl A, Gustafsson J (1997) Deriving annotations for tight calculation of execution time. In: Proceedings of Euro-Par, pp 1298–1307
6. Ermedahl A, Fredriksson J, Gustafsson J, Altenbernd P (2009) Deriving the worst-case execution time input values. In: Proceedings of ECRTS, pp 45–54
7. Ferdinand C (1997) Cache behavior prediction for real-time systems. PhD Thesis, Universität des Saarlandes

8. Ferdinand C, Wilhelm R (1999) Efficient and precise cache behavior prediction for real-time systems. Real-Time Syst 17(2–3):131–181
9. Ferdinand C, Heckmann R, Langenbach M, Martin F, Schmidt M, Theiling H, Thesing S, Wilhelm R (2001) Reliable and precise WCET determination for a real-life processor. In: Proceedings of EMSOFT, LNCS, vol 2211, pp 469–485
10. Ferdinand C, Heckmann R, Wolff HJ, Renz C, Parshin O, Wilhelm R (2006) Towards model-driven development of hard real-time systems. In: Proceedings of ASWSD, pp 145–160
11. Ferdinand C, Heckmann R, Sergent TL, Lopes D, Martin B, Fornari X, Martin F (2008) Combining a high-level design tool for safety-critical systems with a tool for WCET analysis on executables. In: Proceedings of ERTS
12. Fredriksson J, Nolte T, Ermedahl A, Nolin M (2007) Clustering worst-case execution times for software components. In: Proceedings of WCET
13. Gheorghita SV, Stuijk S, Basten T, Corporaal H (2005) Automatic scenario detection for improved WCET analysis. In: Proceedings of DAC, pp 101–104
14. Gupta N, Mathur AP, Sofia ML (1998) Automated test data generation using an iterative relaxation method. In: Proceedings of SIGSOFT, pp 231–244
15. Gustafsson J, Ermedahl A, Lisper B (2005) Towards a flow analysis for embedded System C programs. In: Proceedings of WORDS, pp 287–300
16. Healy C, Sjödin M, Rustagi V, Whalley D, van Engelen R (2000) Supporting timing analysis by automatic bounding of loop iterations. J Real-Time Syst Vol. 18 (2/3) 129–156
17. Healy CA, Whalley DB, Harmon MG (1995) Integrating the Timing Analysis of Pipelining and Instruction Caching. In: Proceedings of RTSS, pp 288–297
18. Heckmann R, Langenbach M, Thesing S, Wilhelm R (2003) The influence of processor architecture on the design and the results of WCET tools. Proc RTS 91(7):1038–1054
19. Ji ML, Wang J, Li S, Qi ZC (2009) Automated worst-case execution time analysis based on program modes. Comp J 52(5):530–544, online 2007
20. Ju L, Huynh BK, Roychoudhury A, Chakraborty S (2008) Performance debugging of Esterel specifications. In: Proceedings of CODES/ISSS, pp 173–178
21. Kästner D, Wilhelm R, Heckmann R, Schlickling M, Pister M, Jersak M, Richter K, Ferdinand C (2008) Timing validation of automotive software. In: Proceedings of ISOLA, communications in computer and information science, vol 17, pp 93–107
22. Kim JE, Kapoor R, Herrmann M, Härdtlein J, Grzeschniok F, Lutz P (2008) Software behavior description of real-time embedded systems in component based software development. In: Proceedings of ISORC, pp 307–311
23. Kim JE, Rogalla O, Kramer S, Hamann A (2009) Extracting, specifying and predicting software system properties in component based real-time embedded software development. In: Proceedings of ICSE, pp 28–38
24. Kirner R, Lang R, Freiberger G, Puschner P (2002) Fully automatic worst-case execution time analysis for Matlab/Simulink models. In: Proceedings of ECRTS, pp 31–40
25. Li YTS, Malik S (1995) Performance analysis of embedded software using implicit path enumeration. In: Proceedings of DAC, pp 456–461
26. Lucas P, Parshin O, Wilhelm R (2009) Operating mode specific WCET analysis. In: Proceedings of JRWRTC, pp 15–18
27. Mauborgne L, Rival X (2005) Trace partitioning in abstract interpretation based static analyzers. In: Proceedings of ESOP, LNCS, vol 3444, pp 5–20
28. Pedro PSM (1999) Schedulability of mode changes in flexible real-time distributed systems. PhD thesis, University of York
29. Ravi TV, Gowda KC (1999) Clustering of symbolic objects using gravitational approach. IEEE Trans Syst, Man Cybernetics B 29(6):888–894
30. Reineke J (2008) Caches in WCET analysis. PhD thesis, Universität des Saarlandes
31. Sen R, Srikant YN (2007) Executable analysis using abstract interpretation with circular linear progressions. In: Proceedings of MEMOCODE, pp 39–48
32. Stein I, Martin F (2007) Analysis of path exclusion at the machine code level. In: Proceedings of WCET

33. Tan L (2009) The worst-case execution time tool challenge 2006. Int J Softw Tools Technol Transfer (STTT) 11(2):133–152
34. Tan L, Wachter B, Lucas P, Wilhelm R (2009) Improving timing analysis for Matlab Simulink/Stateflow. In: Proceedings of ACES-MB, pp 59–63
35. Theiling H (2002) Control flow graphs for real-time systems analysis. PhD thesis, Universität des Saarlandes
36. Theiling H (2002) ILP-based interprocedural path analysis. In: Proceedings of EMSOFT, Springer, LNCS, vol 2491, pp 349–363
37. Theiling H, Ferdinand C, Wilhelm R (2000) Fast and precise WCET prediction by separated cache and path analyses. Real-Time Syst 18(2/3):157–179
38. Thesing S (2004) Safe and precise WCET determination by abstract interpretation of pipeline models. PhD thesis, Universität des Saarlandes
39. Thesing S, Souyris J, Heckmann R, Randimbivololona F, Langenbach M, Wilhelm R, Ferdinand C (2003) An abstract interpretation-based timing validation of hard real-time avionics software systems. In: Proceedings of DSN, pp 625–632
40. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mueller F, Puaut I, Puschner P, Staschulat J, Stenström P (2008) The determination of worst-case execution times – overview of methods and survey of tools. ACM Trans Embedded Comput Syst (TECS) 7(3) pp. 36:1–36:53

Chapter 7
Reconciling Compilation and Timing Analysis

Heiko Falk, Peter Marwedel, and Paul Lokuciejewski

7.1 Why Should Compilers and Timing Analysis Be Integrated?

According to forecasts such as a report published by the National Research Council in the US [21], embedded devices will be a main application area of information technology in the future. Therefore, we can observe an increased interest into embedded systems. Funding of embedded systems research in Europe by the European Community (see "Objective ICT-2009.3.4 Embedded Systems Design" in [3]) is a clear indicator of this trend. Also, market statistics [12] demonstrate the increasing market for certain embedded devices.

This leads to the question: is there some way of defining the term "embedded system"? This would help us to separate the embedded market from the non-embedded market. According to Marwedel [20], embedded systems are *"information processing systems embedded into a larger product."* Examples include information processing systems in cars, trains, airplanes, and fabrication equipment. A more recent definition is the result of the work performed by Edward A. Lee. He wrote [14]: *"Embedded software is software integrated with physical processes. The technical problem is managing time and concurrency in computational systems."* The first sentence defines the term "embedded software", but can be easily extended into a definition of the term "embedded system" by just replacing the term "software" by "system". This new definition stresses the link to physical systems and time. Actually, to model time as a "first class citizen" is a key distinction between embedded and non-embedded systems. According to Edward A. Lee, *"The*

H. Falk (✉)
Institute of Embedded Systems/Real-Time Systems, Ulm University, D-89069 Ulm, Germany
e-mail: Heiko.Falk@uni-ulm.de

P. Marwedel · P. Lokuciejewski
Computer Science 12, TU Dortmund University, D-44221 Dortmund, Germany
e-mail: Peter.Marwedel@tu-dortmund.de; Paul.Lokuciejewski@tu-dortmund.de

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_7,

lack of timing in the core abstraction (of computer science) is a flaw, from the perspective of embedded software" [13]. Most embedded systems are indeed real-time systems and most real-time systems are embedded. In the following, we will only consider embedded systems that are also real-time systems, and call them embedded real-time systems. The importance of the link to physics, including the need to model time, also leads to the introduction of the term *"cyber-physical systems"* [15]. We assume that the terms "cyber-physical systems" and "embedded real-time systems" are equivalent.

How are embedded real-time systems designed? For embedded real-time systems, hardware and software are both important, but in this chapter we will just consider software. How is embedded real-time software designed? Embedded real-time software may be manually written or may be generated from some model (for example some data flow model [4]). This software, typically written in some imperative programming language such as C, is then compiled.

How is timing taken into account in this process? Typically, timing is not considered before executable software is available. Once it is available, we can try to derive a *worst-case execution time (WCET)*. Towards this end, the software can be executed on a real or simulated processor or it can be analyzed for its timing behavior. Recently, powerful timing analyzers became available for this purpose [1]. In contrast to measurement-based WCET estimators, formal tools such as aiT derive safe upper bounds on the execution times.

In an attempt to analyze the impact of using *scratchpad memories (SPMs)*, we connected such an analyzer to our experimental memory-aware optimizing compiler *encc* [23]. Figure 7.1 shows the setup.

SPMs are small memories, which are mapped into the memory address space. Due to their small size, the energy required per access as well as the access times are smaller than for larger memories [2]. *encc* tries to take advantage of that by mapping frequently used objects to the SPM, instead of mapping them to some larger "main" memory. The mapping strategy is based on a knapsack-like model. We compared software optimized in this way with software using caches. Caches

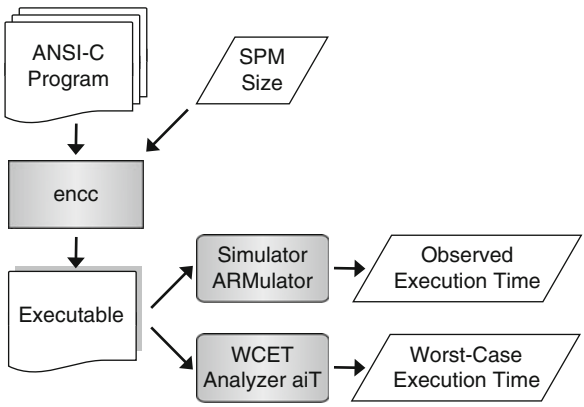


Fig. 7.1 Computing worst-case and observed execution times after code generation

are designed to improve the *average-case execution time (ACET)*, but may be rather slow in the worst case. In the worst case, a cache-based system may actually be slower than a system without a memory hierarchy. We expected that a scratchpad based system would have a much better worst case, if a compiler like *encc* was used for an optimized mapping to the SPM. Therefore, we ran experiments using the setup of Fig. 7.1. We compared the results for two different architectures:

- The first architecture is comprised of an ARM7 processor, a cache and a "main memory". Consistent with a real ARM7-based system, we assume that loads require two cycles for reading the instruction, three cycles for processing, and two cycles for fetching data from memory in the worst case. For store instructions, we assume that, in the worst case, we need two cycles for reading the instruction, two cycles of processing and two cycles for actually storing the results. For all other instructions, we need two cycles for reading the instruction, and one cycle of processing. Rather large numbers are the result of potential cache block (re)-loadings. In the worst case, every access to the cache is a miss, resulting in a (re)-loading of the cache block. This (re)-loading requires several cycles.
- The second architecture is comprised of an ARM7 processor, a scratchpad memory and a "main memory". Consistent with a real ARM7-based system, we assume that loads require zero cycles for reading the instruction, three cycles for processing, and two cycles for fetching data from memory in the worst case. For store instructions, we assume that, in the worst case, we need zero cycles for reading the instruction, two cycles of processing and zero cycles for actually storing the results. For all other instructions, we need zero cycles for reading the instruction, and one cycle of processing. The low overhead results from the fact that instruction fetches are well integrated into the pipeline in this case.

Figure 7.2 shows the results for the G.721 benchmark [24]. We realize that, for SPM-based systems (cf. Fig. 7.2a), worst-case and observed execution times actually decrease with an increasing size of the SPM. For cache-based systems

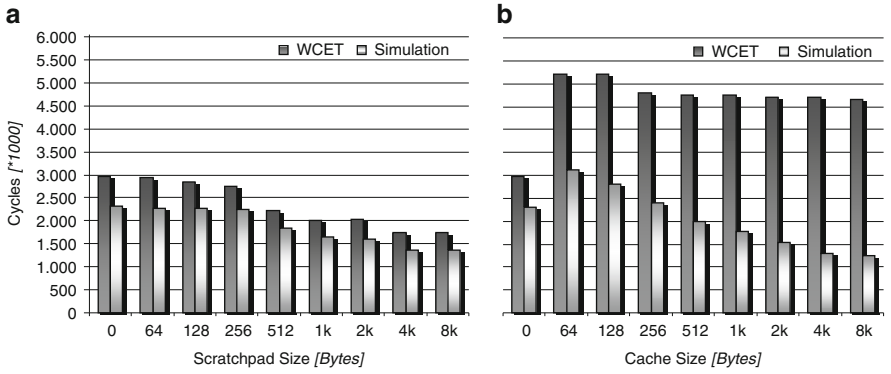


Fig. 7.2 Impact of (**a**) scratchpads and (**b**) caches on worst-case and observed execution times

(cf. Fig. 7.2b), the observed execution time decreases, but the worst-case execution time stays at a rather high level. For small caches, it is even larger than for systems without a cache. Rather large numbers can at least partially be attributed to using a unified cache: this means that we have a single cache for both instructions and data, and there may be conflicts between instructions and data.

The approach just described computes worst-case execution times after code generation. Similar approaches are frequently used in code generation for embedded real-time systems. These approaches use an iterative procedure:

- Specification of embedded software
- Generation of imperative code, e.g., C code
- Compilation of the code for the specific target architecture
- Computation of the WCET, either by using measured worst-case execution times or by formally deriving a safe upper bound on the execution time
- If the derived execution times exceed the real-time constraint, the procedure must be repeated with some changed imperative code or compiler options.

There are no clear rules for generating this changed code. Changes are frequently based on intuition, rather than on clear rules. The number of iterations required may be unknown. Therefore, we are calling this a *trial-and-error-based development of real-time code*. Additional problems are resulting from the use of measured execution times, as these do not provide safe upper bounds of execution times. Therefore, timing constraints may be violated in the real application, while design-time checks revealed no problem.

How can we avoid this trial-and-error-based development of real-time code? Typically, time constraints for real-time code are known *before* compilation. Iterations could be avoided if we used this knowledge already during compilation, instead of checking time constraints *after* compilation. Such an approach opens many opportunities. The compiler could already check if time constraints are met. Going one step further, the compiler could already perform optimizations aiming at a reduction of the worst-case execution time. Figure 7.3 compares the traditional approach to use timing information in the software generation process.

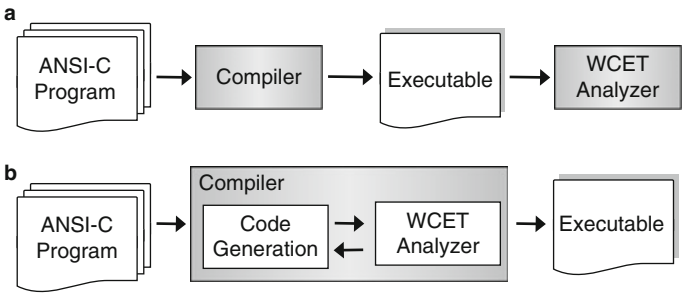


Fig. 7.3 (**a**) Late WCET-checking (**b**) reconciled WCET analysis/compilation

In the traditional approach, WCET-checking takes place *after* compilation (cf. Fig. 7.3a). In the proposed new approach (cf. Fig. 7.3b), WCET information is already used *during* compilation to enable WCET-based optimizations. Actually, this solves one problem which currently exists with compilers: compiler designers are claiming to design optimizing compilers. However, optimizations need a cost function. Without such a cost function, "optimizations" can at best be based on guesses of the impact of "optimizations" on the code quality. Currently available compilers can very easily use code size as a cost function. However, the execution time, both in the form of an average-case execution time as well as in the form a worst-case execution time, can hardly be predicted.

This is what we would like to change with our *WCET-aware C Compiler WCC*. WCC uses an explicit worst-case execution time model of the target processor to steer optimizations. How do we obtain such a worst-case execution time model? One of the golden principles of computer science is to implement a certain functionality only once. Execution time models – if available at all – are typically available in the form of worst-case execution time analyzers. Therefore, instead of designing a new timing model, we propose integrating an available worst-case execution time analysis tool into a compiler optimizing worst-case execution times, and in this way reconciling compilation and timing analysis.

The remainder of this chapter is structured as follows: Sect. 7.2 presents the overall structure of our WCET-aware C Compiler WCC, followed by the description of the particular challenges a compiler writer faces when developing WCET-aware optimizations in Sect. 7.3. Sections 7.4–7.9 briefly highlight some of the WCET-aware optimizations currently integrated into WCC: function inlining, loop unrolling, loop unswitching, register allocation, scratchpad allocation and cache partitioning. Finally, Sect. 7.10 summarizes this chapter and gives an outlook on our future work.

7.2 Structure of the WCET-Aware C Compiler WCC

The WCET-aware C Compiler WCC [6] described in this chapter is an ANSI-C compiler for the Infineon TriCore TC1796 processor which is heavily employed in the automotive industry. WCC's overall structure is depicted in Fig. 7.4. Those stages of the compiler connected with solid arrows resemble a typical optimizing compiler:

ICD-C: The ICD-C framework [10] is a compiler front-end providing a machine-independent IR for C code. It features machine-independent code analyses and optimizations. WCC uses ICD-C's code selector interface to couple the front-end with a tree-pattern matching based code selector for the TC1796 processor.

Code Selector: The code selector translates the source-level IR ICD-C to TriCore assembly code. WCC's tree grammar for the TC1796 consists of approx. 33,000 lines of C++ code resulting in the generation of highly efficient machine code.

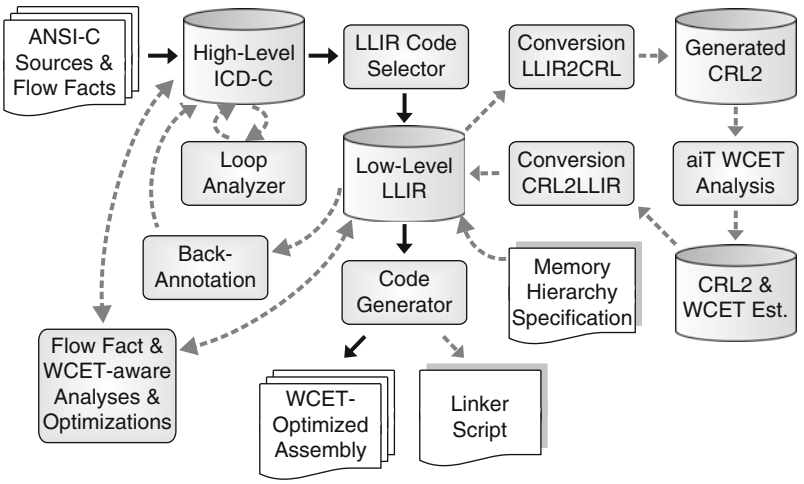


Fig. 7.4 WCC compiler infrastructure

ICD-LLIR: ICD-LLIR [11] is a retargetable low-level IR for compiler back-ends. It includes various fully retargetable assembly-level analyses and optimizations. WCC's TC1796 processor description for ICD-LLIR consists of approx. 13,000 lines of C++ code, capturing all aspects of the complex TriCore architecture.

Code Generator: The code generator finally emits valid assembly code from the class structures of the TriCore ICD-LLIR within WCC's back-end.

The key components turning WCC into a unique WCET-aware C compiler are depicted with dashed arrows in Fig. 7.4. The following Sects. 7.2.1–7.2.5 describe these modules in more detail. They deal with memory hierarchy specification, WCC's integration of WCET analysis, flow facts, loop bound analysis and back-annotation of WCET data.

7.2.1 Specification of Memory Hierarchies

The performance of many systems is largely dominated by the memory subsystem. Due to the large speed gap between slow memories and fast processors, execution times of software widely depend on the memories. Obviously, static WCET estimates also heavily depend on the memories. In the WCC compiler environment described in this section, where the WCET analyzer is tightly integrated into the compilation process, it is in the duty of the compiler to provide the WCET analyzer with detailed information about the underlying memory hierarchy. For this reason, WCC includes an infrastructure to specify memory hierarchies. Furthermore, WCC uses this memory hierarchy infrastructure to do memory-aware optimization by moving parts of a program's code and data onto fast memories.

WCC's memory hierarchy infrastructure is designed to be lightweight and to only support WCET analysis and optimizations moving parts of a program across memories. In a conventional code generation environment, this kind of memory allocation is usually performed by the linker during the final step of generating a binary executable. Thus, the information usually available only while linking needs to be provided already to the WCC compiler itself. This is because it is up to WCC in our setup to decide on a program's memory layout, and no longer up to the linker.

WCC provides a simple text file interface to specify memory hierarchies. Such a memory specification describes different regions of a processor's physical memory hierarchy. For each physical memory region, the following attributes can be defined:

- The region's base address and absolute length
- Access attributes like e.g., read, write, executable
- Memory access times, specified in processor cycles
- Assembly-level sections that are allowed to be mapped to a memory region.

For caches, various attributes like e.g., size, line size or associativity can be specified, too. Now that WCC is aware of the processor's physical memories, program fragments need to be moved to the present memories. For this purpose, assembly-level sections serving as containers for code, data, constants etc. are attached to the available physical memories. Memory allocation of program fragments is now done in WCC's back-end by assigning functions, basic blocks or data to these assembly-level sections. The compiler's infrastructure provides a convenient API to do such memory assignments of code and data. Symbol tables allow to retrieve physical memory addresses per function, basic block or data object.

Finally, the memory allocation decided by WCC must be respected by subsequent linkage stages. The binary executable generated by WCC must exactly match the memory layout decided by WCC. Since the executables are produced outside WCC by an external linker, WCC automatically generates a GNU *ld* compatible linker script and invokes the linker using this linker script. This way, the binary executable is fully equivalent to the memory layout determined by WCC's optimizations.

7.2.2 Integration of Static WCET Analysis into the Compiler

Static WCET analysis takes place at the assembly/binary code level since processor-specific information and machine code is unavailable at higher abstraction levels. Thus, the WCET analyzer aiT is coupled to the WCC compiler in its back-end at machine code level (cf. Fig. 7.4).

The WCET analyzer aiT uses its very own intermediate code representation (*IR*) which is called CRL2. During WCET analysis, CRL2 serves as exchange format storing the application under WCET analysis and all of aiT's analysis results. Since both ICD-LLIR and CRL2 are low-level IRs, a mutual translation of their

control flow graphs (CFGs) is straightforward. The CFGs of both IRs consist of functions. Each function is a list of basic blocks connected via edges. Basic blocks in turn are a sequence of instructions. In both IRs, an instruction consists of several operations executed in parallel by VLIW machines. Due to the analogy of both LLIR and CRL2, it is basically sufficient to traverse the LLIR CFG and to generate corresponding CRL2 components to construct an equivalent CRL2 CFG.

Moreover, physical memory addresses provided by WCC's memory hierarchy infrastructure (cf. Sect. 7.2.1) are exploited to construct the CRL2 CFG. Using WCC's memory hierarchy API, physical addresses for LLIR basic blocks are determined and passed to aiT. In addition, branch targets of jump operations, which are represented by symbolic block labels, are translated into physical addresses.

Using the conversion step from LLIR to CRL2, WCC produces a CRL2 file representing the program for which WCET timing data is required. Fully transparent to the compiler user, WCC invokes aiT on this CRL2 file. The compiler takes control over the WCET analyzer and performs all required static WCET analyses. As a consequence, the WCET analyzer is completely encapsulated within WCC. The compiler user is unaware of the fact that timing analysis is performed in the background.

After the transparent invocation of aiT, the results of the static WCET analysis are imported back into the compiler. This is done by traversing the final CRL2 file produced by aiT containing all valuable analysis data, extracting this WCET data and attaching it back to the compiler's ICD-LLIR IR. The following list gives an overview about the WCET data made available within WCC this way [8]:

- WCET of the entire program, of each function, and each basic block
- Worst-case call frequency per function
- Worst-case execution frequency per basic block or CFG edge
- Execution feasibility of each CFG edge
- Safe approximation of register values
- Encountered I-cache misses per basic block.

7.2.3 Flow Fact Specification and Transformation

A program's execution time (on a given hardware) is largely determined by its control flow, i.e., the execution order of instructions or basic blocks, as modeled by the CFG. Usually, constructs like e.g., loops or conditionals express control flow. In general, static WCET analysis is undecidable since it is undecidable to compute how many times a general loop iterates. Since loop iteration counts are crucial for a precise WCET analysis, and since they can not be computed for arbitrary loops in general, loop iteration counts need to be specified by the user of a WCET analyzer.

Besides loops known from high-level programming languages, any cycle in a program's CFG needs to be annotated manually by the user. These user-provided annotations specifying the control flow are called *flow facts*. Flow facts describe

the set of possible execution paths of a program. To make WCET analysis feasible, there must be enough flow facts to limit the execution count of every statement of a program. User-provided flow facts should be specified inside the source code since this way, only the code base needs to be maintained, and not the source codes plus some external flow fact files which are potentially forgotten. The WCC compiler fully supports source-level flow facts by means of ANSI-C pragmas.

Loop bound flow facts limit the iteration counts of regular loops. These are *for*-, *while-do*- and *do-while*-loops of ANSI-C having a single entry point and a well-defined termination condition. For such loops, loop bound flow facts allow to specify the minimum and maximum iteration counts. For example, the following C code snippet specifies that the shown loop body is executed 50–100 times:

```
_Pragma( "loopbound min 50 max 100" )
for ( i = 1; i <= maxIter; i++ )
  Array[ i ] = i * fact * KNOWN_VALUE;
```

Allowing to provide a minimum and maximum loop iteration count enables to annotate data-dependent loops as shown above. In order to annotate irregular loops or recursions, WCC provides *flow restriction* flow facts allowing to relate the execution frequency of one C statement with that of other statements. Flow restrictions allow to specify linear dependencies between arbitrary positions in the C source code. For instance, the following piece of code annotates a triangular loop:

```
_Pragma( "marker outermarker" )
Statement A;

for ( i = 0; i < 10; i++ )
  for ( j = i; j < 10; j++ )
    _Pragma( "marker innermarker" )
    Statement B;

_Pragma( "flowrestriction 1*innermarker <= 55*outermarker" );
```

The execution frequency of the code denoted as `innermarker` is at most 55 times larger than that of statement `A` labeled as `outermarker` for this example.

Due to the fact that source-level flow facts are highly desirable, there is a semantic gap between the place where flow facts are specified (C code) and where they are actually used for WCET analysis (assembly code). WCC is inherently aware of this semantic gap and automatically transforms specified flow facts whenever the compiler's IRs are modified or optimized. All optimizations of WCC are made fully flow-fact aware using built-in *flow fact update* techniques. They ensure that always safe and precise flow facts are maintained, irrespective of how and when the optimizations modify the intermediate code.

7.2.4 Polyhedral Loop Bound Analysis

Manual flow fact annotation as described in Sect. 7.2.3 becomes tedious and error-prone even at the source code level if the program to be annotated is long and complex. It becomes even infeasible if the program's source code is automatically

generated by some high-level specification tool like e.g., SCADE or Matlab. Manual user intervention for flow fact annotation should be avoided. To relieve the user form this burden and to establish a fully automated compiler framework for WCET minimization, a static loop analyzer producing flow facts was integrated into WCC.

For loop bound analysis, possible values of the loop counter variable need to be determined. However, it is infeasible to compute all actual values of loop counters in an exact fashion for general loops. WCC's loop analyzer bases on *abstract interpretation* whose fundamental idea is to find a compromise between analysis precision and analysis runtime. In the context of loop bound analysis, an abstraction is achieved by representing values of loop counters as interval instead of taking all possible concrete values into account. This abstraction results in a loss of information during abstract interpretation, turning the loop bound analysis feasible.

The main drawback of abstract interpretation is that it performs an iterative fixed-point analysis. This fixed-point iteration can consume a significant amount of time for loops with large iteration counts. WCC's loop bound analyzer combines abstract interpretation with mechanisms avoiding this iterative behavior.

First, a standard technique called *program slicing* is applied to the body of the currently analyzed loop. Usually, loop bodies mostly contain instructions not affecting the calculation of the loop counter. Using program slicing, these instructions not influencing neither the concrete nor the abstract semantic of the loop counter are recognized to be meaningless for loop bound analysis and are not considered in the following. This frequently results in loops with almost empty loop bodies.

Second, it is verified after program slicing whether the remaining relevant instructions basically represent affine expressions over the loop counter variables. If this precondition is met, these affine expressions are translated into a *polyhedral model* where a polyhedron P is an N-dimensional geometrical object defined as a set of linear inequations: $P := \{x \in \mathbb{Z}^N \mid Ax = a, Bx \geq b\}$ for $A, B \in \mathbb{Z}^{m \times N}$ and $a, b \in \mathbb{Z}^m$ and $m \in \mathbb{N}$. The problem of computing loop iteration counts is then equivalent to computing the number of integer points in a polytope. To efficiently count the integer points of polyhedra, *Ehrhart polynomials* are used within WCC.

Using this combination of program slicing and polyhedral models [17], WCC's loop analyzer is able to exactly compute loop bounds for very large numbers of loops of standard benchmark suites. Furthermore, loop bound analysis within WCC usually requires significantly less runtime than approaches solely based on abstract interpretation. WCC's loop analyzer has proven to be of superior quality – among all tools participating in the WCET Tool Challenge 2008, it was the only one which solved all flow fact related analysis problems [9].

7.2.5 Back-Annotation of WCET Data

The technical infrastructure of WCC described so far allows the effective WCET minimization by optimizations applied at assembly/ICD-LLIR level where WCET estimates provided by the WCET analyzer are imported and made accessible to

the compiler. In contrast, high-level WCET-aware optimizations taking place at the source code / ICD-C level are not yet supported due to the lack of WCET timing information at this abstraction level. However, high-level optimizations focusing on function call and loop transformations exhibit a large potential for WCET reduction. Thus, a worst-case timing model for ICD-C is highly desired. To transform WCET timing data from assembly to the source code level, a bridge between both abstraction levels of the code is required. This is realized by WCC's *back-annotation*.

In order to raise the abstraction level of the WCET timing model from assembly to source code level, a connection between objects of the ICD-LLIR and ICD-C IRs must be established. For coarse-grained objects such as assembly/C files and functions, this is trivial since a 1:1 correlation exists.

Correlating fine-grained basic blocks is more complicated since a 1:1 mapping does not always exist. By definition, a basic block is a code fragment with a single entry and exit point where jumps can only occur at the block's end. Function calls implicitly modifying the control flow can be handled in two different ways. They can either represent a basic block boundary, i.e., a new basic block begins after a function call, or they are considered as regular statements/instructions that do not explicitly modify the control flow. The former definition is used within ICD-LLIR, while the latter is used for ICD-C blocks. Due to the varying definitions and assembly-level optimizations modifying the basic block structure, the relationship of basic blocks represents an n:m mapping in general. For example, one LLIR basic block may correspond to m ICD-C basic blocks, or n LLIR basic blocks may correspond to one ICD-C basic block.

Since WCC's code selector is the interface between the source- and assembly-level IRs, it is also the location where the determination of the relationship between ICD-C and ICD-LLIR basic blocks as well as the establishment of corresponding mappings is integrated. All WCC optimizations taking place at the ICD-LLIR level are extended to automatically update all such mappings when modifying assembly-level basic blocks. Using these mappings, the following WCET timing data (among others) can be back-annotated from assembly to C level:

- WCET of the entire program, of each function, and each basic block
- Worst-case execution frequency per CFG edge
- Execution feasibility of each CFG edge
- Encountered I-cache misses per basic block
- Code size and amount of spill code per basic block.

To the best of our knowledge, WCC is the very first compiler framework providing highly accurate WCET timing information within the compiler front-end.

7.3 Challenges Imposed to WCET-Aware Compilers

In typical complex programs, there usually exist several alternative execution paths from a program's entry point to an exit point where the program terminates. That path among all these alternative paths within a program's CFG having the maximal

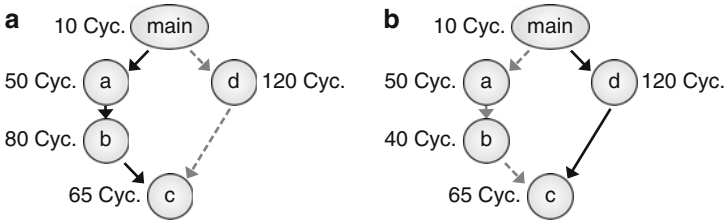


Fig. 7.5 (**a**) Original example CFG (**b**) example CFG after optimization of `b`

WCET is called the *worst-case execution path (WCEP)*. Hence, the WCET of a program is equal to the WCET of its WCEP. In the following, a path's WCET will also be called the path's length. To minimize WCETs by a WCET-aware compiler, optimizations must exclusively focus on those parts of a program lying on the WCEP. Optimization of parts of the program aside the WCEP is ineffective since it does not shorten the WCEP and thus does not reduce the WCET. Therefore, optimization strategies for WCET minimization must have detailed knowledge about the WCEP.

A static WCET analyzer as used within WCC provides detailed information about a program's WCEP, but the sole knowledge of a WCEP is insufficient for effective WCET minimization. Consider the CFG of a function `main` in Fig. 7.5a, consisting of five basic blocks each of them having the indicated WCETs given in clock cycles. Obviously, the longest path through this CFG is `main`, `a`, `b`, `c`. This WCEP, highlighted with solid arrows in Fig. 7.5a, has a WCET of 205 cycles.

Assuming that some optimization is able to reduce `b`'s WCET from 80 down to 40 cycles, the CFG shown in Fig. 7.5b results from this optimization. As can be seen, the WCEP after optimization of `b` is `main`, `d`, `c`. This example shows that the WCEP is very unstable during optimization – it can switch from one path within the CFG to a completely different one in the course of optimizations. In summary, a WCET-aware compiler is faced with the following challenges, turning the development of WCET-aware optimizations into an even more demanding area of research compared to traditional compiler optimization:

- During the entire optimization process, WCET-aware optimizations must have detailed knowledge of the current WCEP at any point in time.
- Such optimizations must be aware of possible WCEP switches in the course of an optimization and they thus have to recompute the WCEP whenever necessary.

The remainder of this chapter presents optimizations of the WCC compiler that explicitly consider switching WCEPs and thus reduce WCETs successfully.

7.4 Machine Learning Based WCET-Aware Function Inlining

Function inlining is a well-known transformation replacing a function call by the body of the callee while storing the arguments in variables that correspond to

function parameters. Function inlining originally intends to reduce average-case execution times (ACETs) for the following reasons:

- By copying the callee's code into the caller, the calling overhead is reduced since the function call and return instructions as well as parameter handling is removed.
- Removing calls and returns potentially yields a smoother pipeline behavior.
- Inlining potentially enables other compiler optimizations which are unable to do global code analysis and transformation across several functions.

One of the main drawbacks of this optimization is the increased register pressure. By inserting additional variables from the inlined function into the caller, possibly more registers are required which may result in additional spill code degrading a program's performance. Furthermore, I-cache performance may be degraded due to the increased code size and reduced spatial locality of memory accesses.

The evaluation of the impact of inlining on ACETs is challenging due to the side-effects of the register allocator and the caches. This complicates the decision whether a function should be inlined. Up to now, more or less simple compiler heuristics try to predict if inlining of a given function will be beneficial or not. The WCC compiler includes mechanisms for function inlining which, on the one hand, focus on WCETs instead of ACETs. On the other hand, simple inlining heuristics are replaced by sophisticated *machine learning (ML)* techniques [19].

ML techniques provide a flexible and adaptive way to automatically generate compiler heuristics handling complex search spaces. Furthermore, such ML based approaches often outperform hand-crafted heuristics. The application of inlining poses a typical *classification problem*, i.e., one is interested in a classification rule that decides if a particular function should be inlined or not for a particular call site.

WCC extracts numerous features from the program to be optimized to drive the classification process for inlining. These include among others:

- Size of caller/callee
- WCET of the caller/callee
- Worst-case execution frequency per call site
- WCET of a callee for a particular call site, i.e., the product of the callee's WCET and the worst-case execution frequency of the call site
- Number of call sites of a callee lying on the current WCEP
- Whether a callee is called from only one call site
- Number of registers whose lifetimes span a call. Inlining of calls crossing a high number of lifetimes increases the probability for spilling
- Maximal number of registers that are simultaneously live within a function.

These features are extracted by heavily using WCC's infrastructure (cf. Sect. 7.2). Using the tight integration of a timing analyzer into the compiler back-end, all WCET- and WCEP-related features are collected. Features related to liveness of registers are computed by a dedicated register pressure analyzer in the compiler back-end. These assembly code-specific features are then propagated to the ICD-C level via WCC's back-annotation. All other features dealing with the number and positions of call sites are directly computed at C code level within ICD-C.

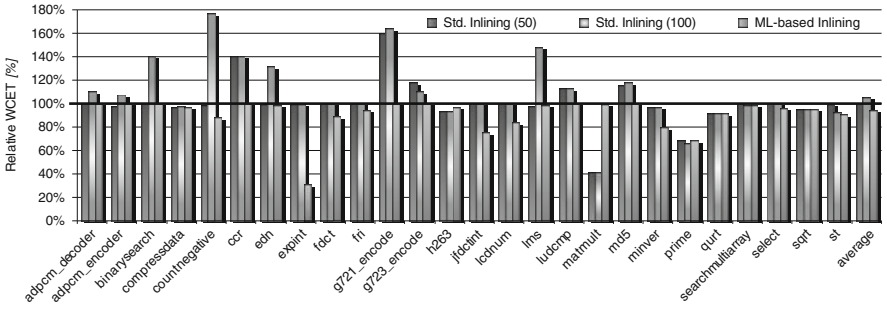


Fig. 7.6 Relative WCETs after different function inlining strategies

Hereafter, a random forests classifier integrated in the ML tool RapidMiner uses these features at ICD-C level. The knowledge base of this classifier was generated off-line by means of a training set of 41 realistic benchmarks. If the classifier decided to inline a function at a particular call site, this optimization was applied to WCC's IR, followed by a WCET analysis of the resulting optimized program. This way, all WCET-related data used by the classifier's features is updated and possible WCEP changes (cf. Sect. 7.3) are automatically captured.

Figure 7.6 shows the WCETs for several inlining strategies and 26 real-life benchmarks. The first two bars per benchmark denote standard, WCET-unaware heuristics inlining only functions with less than 50 and 100 expressions, respectively. The third bar denotes the WCETs achieved by WCC's novel ML-based inlining which does not use any hard-coded size threshold. All results are given as a percentage, with 100% corresponding to the benchmarks' WCETs if no function inlining is applied. All results are generated using WCC's highest optimization level *-O3*.

As can be seen, inlining of functions with less than 50 expressions only has a marginal impact on the benchmarks' WCETs. Only for very few benchmarks, significant WCET reductions were achieved. For most benchmarks, WCETs did not change after inlining. On average over all considered benchmarks, a WCET of 100.3% of the WCET without inlining was obtained. When inlining functions with less than 100 expressions, the WCETs of 11 benchmarks increased significantly. On average, this inlining strategy increases WCETs by 5.5%. Unlike these standard inlining approaches, ML-based inlining never significantly increases WCETs in any experiment. Instead, ML-based inlining outperforms both WCET-unaware strategies by up to 11.4% on average over the entire benchmark set.

7.5 WCET-Aware Loop Unrolling Using Static Loop Analysis

In analogy to function inlining (cf. Sect. 7.4), *loop unrolling* is also a well-known optimization requiring sophisticated control mechanisms. It replicates the body of a loop a number of times and adjusts the loop control accordingly. The number of

replications is called the *unrolling factor u* and the original loop is often termed *rolled loop*. If the loop iteration count of a rolled loop is not an integral multiple of *u*, additional code must be generated to correctly handle left-over loop iterations. The benefits of loop unrolling are similar to those of function inlining:

- Loop overhead, i.e., incrementing and testing of the loop counter, is reduced.
- Reducing jumps back to a loop's entry might improve pipeline behavior.
- Unrolling makes instruction-level parallelism in loops explicit and thus potentially enables other compiler optimizations.

The main drawbacks of unrolling are the inherent code size increases potentially augmenting I-cache capacity misses, and additional spill code if the working set of registers of an unrolled loop does no longer fit into a processor's register file. Thus, the central question of loop unrolling is which unrolling factor to use per loop. The unrolling factor depends on several parameters, like e.g., the iteration count of a loop, I-cache memory constraints and an approximation of spill code generation.

Due to lacking sophisticated loop analyses, most compilers only use a constant, small unrolling factor (usually 2 or 4) which does not sufficiently exploit the optimization potential. In contrast, WCC uses its integrated polyhedral loop analyzer (cf. Sect. 7.2.4) which is extended to also support context-sensitive loop iteration counts. In many real-life applications, loop bounds depend on function parameters, i.e., the bounds are variable and depend on the context in which a function is called. Simple loop analyzers only support purely constant loop bounds and thus cannot handle such classes of loops. WCC's loop analyzer uses parameterized polyhedra to model loops depending on function parameters. Thus, this loop analyzer provides many different iteration counts per single loop, depending on a loop's contexts.

Within WCC, loop unrolling takes place at C code level in ICD-C. To circumvent negative unrolling effects due to I-cache thrashing, each loop is translated into assembly code. For assembly code, it is easy to determine the size of the loop header and of its body in bytes. Using WCC's back-annotation (cf. Sect. 7.2.5), this data is shifted back to the C code level. By combining these code sizes with I-cache related data from WCC's memory hierarchy infrastructure (cf. Sect. 7.2.1), WCC's unrolling is able to estimate how large a loop unrolled by factor *u* will become at assembly level so that increases of I-cache misses are avoided.

The possibly many iteration counts per loop provided by WCC's loop analyzer are finally used to estimate whether unrolling leads to the generation of additional spill code during register allocation. For this purpose, a rolled loop L is virtually unrolled by a factor *u* being equal to the *smallest common prime factor (SCPF)* of all possible iteration counts of L. The resulting unrolled loop is denoted L_{SCPF}. For both L and L_{SCPF}, assembly code is generated and the amount of spill code in both loops is counted. If L_{SCPF} contains more spill code than *u* times L's amount of spill code, unrolling led to the generation of additional spill code. Again, this spilling-related data is back-annotated into WCC's front-end. Likewise, WCC performs a WCET analysis of each loop L_{SCPF} and also back-annotates this WCET data.

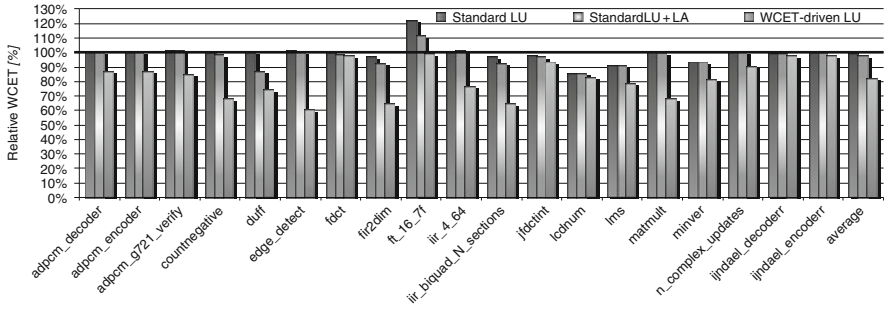


Fig. 7.7 Relative WCETs after different loop unrolling strategies

By combining the available information about iteration counts, code sizes, I-cache properties, spill code and WCETs, the final unrolling factor u^L_{final} is computed for each loop L. Therefore, all innermost loops of a program are considered as unrolling candidates which do not lead to increased spill code. Each unrolling candidate is assigned a profit. The profit represents the expected WCET reduction and code size increase when the loop is unrolled by factor u^L_{final}. Loops with larger profits promise a higher benefit and are thus unrolled first as long as no additional I-cache misses are expected. Loops with negative profit will likely have a negative impact on the WCET and are thus excluded from unrolling [16].

Figure 7.7 shows the WCETs for different unrolling strategies and 19 real-life benchmarks. The first bar per benchmark denotes a standard, WCET-unaware loop unrolling *(LU)* as it can be found in many compilers nowadays. Unrolling is done as long as the unrolled loop is not larger than 50 ANSI-C expressions. Furthermore, only a simple, context-insensitive loop analysis is used here. The second bar per benchmark combines the WCET-unaware standard LU with WCC's context-sensitive loop analyzer *(LA)*. The third bar represents WCC's WCET-aware LU, including its context-sensitive LA. All results are given as a percentage, with 100% corresponding to the benchmarks' WCET if no unrolling is done. All results are generated using WCC's highest optimization level *-O3* and assuming a 2 kB I-cache.

As can be seen, standard LU has minimal positive effects on WCETs. Only for few benchmarks, significant WCET reductions were achieved. On average over all benchmarks, standard LU reduces WCETs by less than 1%. Integrating WCC's sophisticated loop analysis into standard unrolling slightly improves the average WCETs of all benchmarks by 2.9%. Notably improved results are achieved by the proposed WCET-aware LU. WCET reductions of up to 39.5% were obtained. On average, WCET-aware unrolling reduces WCETs by 18.3%.

7.6 Invariant Path Based WCET-Aware Loop Unswitching

As motivated in Sect. 7.3, WCET-aware optimizations must be aware of WCEP switches in the course of an optimization, and they thus have to recompute

the WCEP whenever necessary. In the field of WCET-aware code optimization, this requirement is usually met by pessimistically recomputing the WCEP after each individual code modification. Thus, a time consuming WCET analysis is performed after any code modification. However, this exhaustive WCEP update is not necessary. In most situations, the WCEP provably remains stable during optimization so that many WCEP recomputations can be saved. WCC's concept to model those sub-paths of the WCEP that always remain part of the WCEP independent of any code modification is called the *invariant path* [18].

Only branches in a program's CFG are places where the WCEP can possibly switch (cf. Fig. 7.5). In usual high-level programming languages, such branches are modeled by *if-then-* or *if-else*-statements. The challenge is now to classify each if-then- and if-else-statement as being part of the invariant path or not.

Depending on the conditional expression of an if-then-statement, either the path through the then-part is executed, or the mutually exclusive path bypassing the then-part is taken. For such an if-then-statement, the WCEP either goes along the then-part, or the then-part does not contribute to the WCET. In terms of the invariant path paradigm, a WCEP traversing the then-part is also part of the invariant path since a modification of the code in the then-part can not lead to path switching. This is because the other feasible path of the if-then-statement does not contain any code that might become the new WCEP.

For if-else-statements, a context-sensitive WCET analyzer may determine that the WCEP traverses both the then- and else-part in different execution contexts. In this situation, WCEP switches cannot emerge since it is known from static WCET analysis that both parts always contribute to the program's WCET. Thus, such an if-else-statement can entirely be declared as part of the invariant path.

In two possible cases, WCET analysis may detect that one of the two paths through the then- or else-part is infeasible – under all circumstances, the WCEP traverses exactly one of the two mutually exclusive paths. The first case occurs if static WCET analysis is able to compute that the condition of the if-else-statement always evaluates to *true* or *false*, respectively. One of the two paths is never executed and is thus dead code. This dead code along an infeasible path can be eliminated and the remaining path is part of the invariant path. The second case occurs if the WCET analyzer is unable to statically analyze the condition of the if-else-statement. In this situation, it conservatively assumes that the longest of both alternative paths is the WCEP which is always taken. This is the only situation where a WCEP path switch can occur. Thus, these types of if-else-statements are not part of the invariant path.

Similar to inlining and unrolling, *loop unswitching* is a typical ACET optimization where a trade-off between the execution time improvement and the resulting code size increase must be taken into account. It shifts loop-invariant if- or if-else-statements out of a loop at the cost of loop body duplications. For this reason, loop unswitching can not be applied exhaustively if strict code size constraints must be met. Rather, potential unswitching candidates should be evaluated beforehand, thus enabling to unswitch only those loops leading to maximal runtime improvement while keeping code size increases minimal. Since the way how loop unswitching is made WCET-aware is very similar to the techniques already discussed in previous

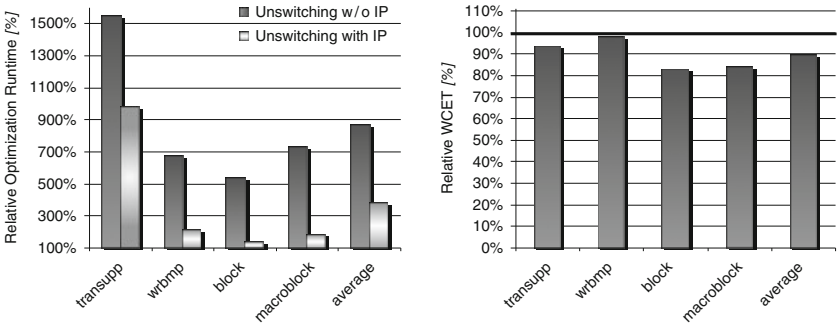


Fig. 7.8 (**a**) Relative optimization runtimes (**b**) relative WCETs after unswitching

sections and heavily relies on back-annotation of WCET-related data from WCC's back-end into the ICD-C IR, the details of this optimization are omitted here. In contrast to inlining and unrolling, WCET-aware loop unswitching uses the invariant path to avoid superfluous WCEP updates.

An analysis of 42 real-life benchmarks revealed that between 85.4% and 88.8% of the benchmarks' total WCETs are contributed by pieces of code lying on the invariant path. Thus, only less than 15% of the average benchmarks' WCETs stem from code where WCEP switches may occur.

To show the effectiveness of the invariant path paradigm, the optimization runtimes of WCET-aware loop unswitching were measured for the most interesting of these 42 real-life benchmarks, once with and once without exploiting invariant path information. The 100% line of Fig. 7.8a corresponds to the optimization time of standard loop unswitching without any WCET heuristics. WCET-aware unswitching without using invariant paths took on average 872% more runtime than standard unswitching. Exploiting invariant paths reduces runtimes down to 379% which corresponds to a speed-up by a factor of 2.3.

Figure 7.8b shows the WCET reductions achieved by WCC's loop unswitching. Again, all results are given as a percentage, with 100% corresponding to the WCET of the original benchmark code after dead code elimination. All results are generated assuming an 8 kB I-cache. As can be seen, an average WCET reduction of 10.4% was achieved for all benchmarks. The maximal WCET reduction of 17.3% was achieved for the `block` benchmark. It contains seven loop-invariant if-then- or if-else statements executed between 4 and 16 times in the worst case. By unswitching, their execution frequencies could be reduced significantly.

7.7 WCET-Aware Register Allocation Based on Graph Coloring

Among all compiler optimizations, *register allocation* is considered the most important one. It intends to use a processor's *physical registers (PHREGs)* most

efficiently to minimize slow main memory accesses. However, memory accesses can not be totally avoided, since the amount of temporary variables (aka *virtual registers, VREGs)* in a program place can exceed the number of available PHREGs. In such a situation, *spill code* is inserted swapping a register out to memory and back.

Currently, register allocators usually decide heuristically where to insert spill code. Due to a lack of precise timing models, current compilers are unaware of the impact of generated spill code on a program's execution time. Thus, badly placed spill code can have a dramatic impact on a program's WCET.

The WCC compiler is the first one to include a WCET-aware graph coloring register allocator. To design a WCET-aware register allocator, the worst-case execution frequencies per CFG node need to be known. However, WCET analysis can not be applied to the program P being input for register allocation to obtain this data. This is because P is not executable since it uses VREGs instead of PHREGs. WCET analysis needs executable and thus register-allocated code to take the mutual influences of P and the processor hardware into account. Hence, there are cyclic dependencies between register allocation and WCET analysis which need to be broken.

Conventional register allocators try to keep as many VREGs in PHREGs as possible, and to move a VREG to memory only if this is really necessary. The traditional way of thinking thus assumes optimistically that all VREGs fit into the physical register file and that only exceptionally, a VREG is allocated to memory. However, this traditional approach is not applicable for a WCET-aware register allocator. The intermediate code produced in the course of all the iterations of traditional graph coloring is not executable and thus, no WCEP can be determined (cf. Sect. 7.3).

For WCET-aware graph coloring, the opposite way of thinking is proposed here: it is pessimistically assumed that all VREGs reside in memory. During register allocation, VREGs are thus moved from memory to PHREGs. This approach has the advantage that the intermediate code generated in the course of register allocation is always executable so that WCEPs can be determined. WCC's WCET-aware graph coloring allocator bases on the following two key characteristics [5]:

- Due to its instability, the WCEP has to be recomputed regularly. Since it is practically infeasible to recompute the WCEP after each individual spill decision, WCEPs are recomputed after deciding on the allocation of one single basic block.
- For a given WCEP, that basic block b with the highest execution of spill code in the worst case is chosen. All VREGs v of b are sorted by the number of occurrences of v in b. This sorting order is passed to a standard graph coloring allocator allocating b and spilling only those registers with least occurrences, if necessary.

Figure 7.9 shows the WCETs for WCET-aware register allocation for 46 different real-life benchmarks as a percentage of the WCET resulting from a traditional graph coloring allocator spilling that node with the highest degree in the interference graph. All results are generated using WCC's highest optimization level *-O3*.

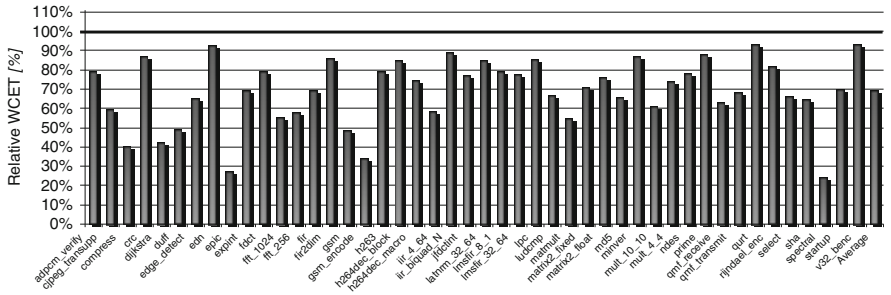


Fig. 7.9 Relative WCETs after WCET-aware register allocation

WCC's register allocator reduces the WCETs of all benchmarks considerably. For `qurt`, a WCET reduction of 6.9% was achieved. For all other benchmarks, even higher gains were observed. The largest gain in terms of WCET was achieved for `spectral` where the WCET after WCET-aware register allocation amounts to only 24.1% of the original WCET, leading to savings of 75.9%. On average over all 46 benchmarks, a total average WCET reduction of 31.2% was obtained.

7.8 WCET-Aware Scratchpad Memory Allocation

Many processors are equipped with software-controllable memories which are tightly integrated into the CPU to achieve best possible performance. Such so-called scratchpad memories (SPMs) are highly efficient and are therefore in general well-suited for optimizations regarding energy consumption and execution times.

For WCET-centric optimizations, SPMs are ideal since the timing of such memories is fully predictable – in contrast to caches (cf. Sect. 7.1). Within WCC, scratchpads are exploited for WCET minimization by placing assorted parts of a program into an SPM. This section describes an approach based on *integer linear programming (ILP)* for moving parts of a program's code onto scratchpads [7].

In order to move individual basic blocks of a program onto a scratchpad memory, WCC's ILP uses a binary decision variable x_i per basic block b_i. b_i is moved onto the scratchpad memory if x_i is equal to 1. The overall goal of this ILP is to find an assignment of values to the variables x_i such that the resulting scratchpad allocation leads to a global WCET minimization.

A scratchpad assignment is legal if the size of all basic blocks allocated to the SPM does not exceed the scratchpad's capacity. This is ensured by the following capacity constraint: $\sum_{i=1}^{n} x_i * s_i \leq S_{SPM}$, where s_i denotes block b_i's size in bytes.

In addition, other constraints need to be added modeling the structure of a program's CFG. The generation of these constraints starts with all those basic blocks b_i located in the innermost loops including no other loops inside them. For each such block b_i and each successor b_{succ} of b_i in the CFG, a constraint is set up bounding

the WCET w_i of b_i: $w_i \geq w_{succ}\ cost_{i,main\_mem} - gain_i * x_i$. This constraint states that the WCET of b_i has to be larger than that of any of the successors of b_i, plus the contribution of b_i to the WCET itself with b_i located in main memory, minus the potential gain when moving b_i from main memory onto the scratchpad memory. This way, constraints for all blocks of the innermost loops can be set up.

For reducible loops L with a unique entry basic block b_L^{entry}, the WCET of the entire body of loop L is thus represented by the variable w_L^{entry}, under the assumption that L is executed exactly once. Since a loop executes its body several times in general, the entire loop's WCET is defined by: $w_L = w_L^{entry} * C_L^{max}$ where the iteration count C_L^{max} of loop L is taken from WCC's flow facts (cf. Sect. 7.2.3). Whenever an outer loop L' includes an inner loop L, this inner loop L is treated as a single CFG node with a WCET represented by the ILP variable w_L as defined above.

Analogously, the WCET of a program's function F is represented within the ILP by the variable w_F^{entry} if basic block b_F^{entry} is F's unique entry point. Whenever a basic block b_i calls some function F, variable w_F^{entry} has to be added to w_i in order to model the interprocedural control flow correctly.

Finally, an entire C program's WCET is represented in the ILP by the variable $w_{\texttt{main}}^{entry}$. In order to minimize a program's WCET by the ILP, the following simple objective function is used: $w_{\texttt{main}}^{entry} \rightsquigarrow \min$.

Furthermore, the ILP formulation produced by the WCC compiler also takes care of adjusted branch instructions making sure that a basic block located in main memory can still branch to a successor placed onto the SPM, and vice versa.

Figure 7.10 shows the average WCETs for SPM allocation of program code for 73 different benchmarks and various scratchpad sizes. Since the TriCore's SPM is large enough to entirely hold all benchmarks, scratchpad sizes are artificially limited here. Figure 7.10 uses SPM sizes relative to the total code size of the benchmarks. Hence, the category "20%" shows the WCETs for a scratchpad size of 20% of a benchmark's code size. Once again, the 100% base line reflects the WCETs of all benchmarks without using the SPM at all. All results are generated using WCC's

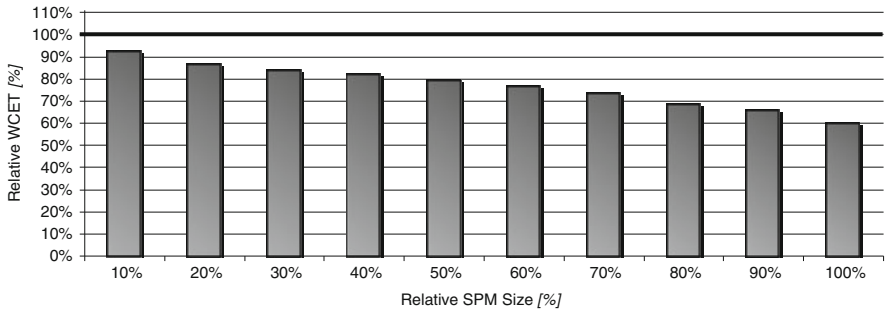


Fig. 7.10 Relative WCETs after WCET-aware scratchpad allocation

highest optimization level *-O3* and assuming an access latency of 20 cycles for main memory and of 1 cycle for the scratchpad.

On average over all 73 benchmarks, steadily decreasing WCETs were finally obtained with increasing scratchpad sizes. Already for small SPMs, WCETs decrease to 92.6% of the WCET without any SPM, corresponding to a WCET reduction of 7.4%. For large SPMs storing the entire benchmark, average WCETs of only 60% of the original WCET were obtained, leading to overall savings of 40%.

7.9 Cache Partitioning for Multi-Task Real-Time Systems

Caches are a source of unpredictability since it is very difficult to predict if a memory access results in a cache hit or miss. In systems running multiple tasks steered by a preemptive scheduler, it is even impossible to determine the cache behavior since interrupt-driven schedulers lead to unknown points of time for context switches. Furthermore, it is even unknown at which address the execution of a preempted task continues, hence it is unknown which cache line is evicted next. An unknown cache behavior forces a static WCET analyzer to conservatively assume a cache miss for every memory access, thus implying a highly overestimated system's WCET.

Cache partitioning is a technique to make the I-cache behavior more predictable. Each task of a system is assigned to a unique cache partition. The tasks in such a system can only evict cache lines residing in the partition they are assigned to. As a consequence, multiple tasks do not interfere with each other any longer w.r.t. the cache during context switches. This allows to apply static WCET analyses for each individual task of the system in isolation. The overall WCET of a multi-task system using partitioned caches is then composed of the WCETs of the single tasks given a certain partition size, plus the overhead required for scheduling including additional time for context switches.

Cache partitioning can be realized in software without any modification of a cache's hardware. For this purpose, the code of each task has to be scattered over the address space such that tasks are solely mapped to only those cache lines belonging to the task. Thus, a task's executable code is split into many chunks which are stored in non-consecutive regions in main memory. To make sure that a task's control flow remains correct after splitting it into chunks, additional jump instructions between these chunks must be inserted into the code. To generate non-contiguous chunks of a task's code, the linker needs to be provided with dedicated relocation information for each task. Since WCC automatically generates linker scripts (cf. Sect. 7.2.1), software-based cache partitioning is easy to realize in this compiler.

The remaining challenge consists of determining a partition size per task such that a multi-task system's overall WCET is minimized. In the following, preemptive round robin scheduling of tasks is assumed and the period p_i of each task $t_i \in \{t_1, \ldots, t_m\}$ is known a priori. The length of the entire system's hyper-period is equal to the least common multiple of all tasks' periods p_i. The schedule count c_i

then reflects the number of times, each task t_i is executed within a single hyper-period. Furthermore, possible cache partition sizes s_j measured in bytes are given: $s_j \in \{s_1, \ldots, s_n\}$.

WCET-aware software based cache partitioning is modeled within WCC using integer linear programming [22]. A binary decision variable $x_{i,j}$ is set to 1 iff task t_i is assigned to a partition of size s_j. To ensure that a task is assigned to exactly one partition, a constraint of the following shape is added for each task t_i: $\sum_{j=1}^{n} x_{i,j} = 1$.

Another constraint in the ILP ensures that the amount of used cache partitions does not exceed the I-cache's total capacity S: $\sum_{i=1}^{m} \sum_{j=1}^{n} x_{i,j} * s_j \leq S$.

It is assumed here that the WCET $WCET_{i,j}$ of each task t_i if executed once using a cache partition of each possible size s_j is given. This is achieved by performing a WCET analysis of each task for each partition size before generating the ILP for software based cache partitioning. A task t_i's WCET $WCET_i$ depending on the partition size used by the task can thus be expressed as: $WCET_i = \sum_{j=1}^{n} x_{i,j} * WCET_{i,j}$.

The objective function of the ILP models the WCET of the entire task set for one hyper-period. This overall WCET is defined as: $WCET = \sum_{i=1}^{m} c_i * WCET_i$. The ILP finally only has to minimize this variable: $WCET \rightsquigarrow \min$.

Figure 7.11 shows the WCETs for WCET-aware cache partitioning and three different benchmark suites. Since no multi-task benchmark suites currently exist, randomly selected task sets from single-task benchmark suites were used. Figure 7.11 shows results for task sets consisting of 5, 10 and 15 tasks, respectively. Each individual point in the figure's curves denotes the average value over 100 randomly selected task sets of a certain size. Benchmarking was done for I-cache sizes ranging from 256 bytes up to 16 kB. All results are given as a percentage, with 100% corresponding to the WCETs achieved by a standard heuristic that uses a partition size per task which depends on the task's code size relative to the code size of the entire task set. All results are generated using WCC's highest optimization level *-O3*.

As can be seen, substantial WCET reductions of up to 36% were obtained. In general, WCET savings are higher for small caches and lower for larger caches.

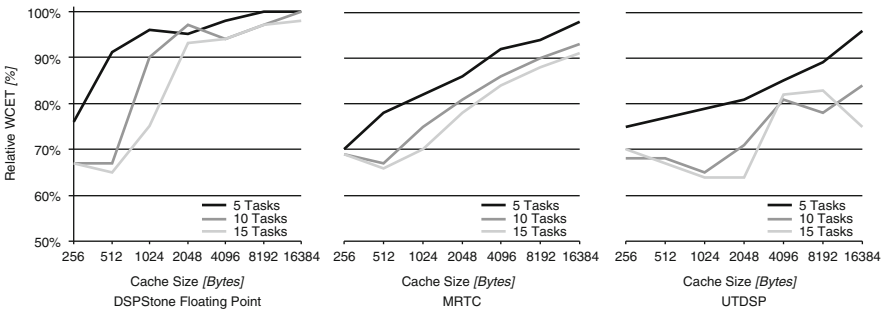


Fig. 7.11 Relative WCETs after cache partitioning for different benchmark suites

For DSPstone, WCET reductions between 4% and 33% were achieved. For the MRTC benchmarks, an almost linear correlation between WCET reductions and cache sizes was observed, with maximal WCET savings of 34%. For the large UTDSP benchmarks, WCET reductions of up to 36% were finally observed. In most cases, larger task sets exhibit a higher optimization potential so that WCC's cache partitioning achieves higher WCET improvements as compared to smaller task sets.

7.10 Conclusions and Future Work

Real-time code needs to meet real-time constraints. Optimization potential is lost if these time constraints are considered only *after* code has been generated. We propose to reconcile compilers and timing analysis such that WCET information can already be exploited *during* compiler optimizations. Up till now, not much was known about the gains in terms of WCET which can be achieved in this way. This chapter provides an overview over research work exploring the potential of such integrated compilation and timing analysis. The chapter presents the WCET-aware C Compiler WCC aiming at code optimization minimizing worst-case execution times. WCET-aware optimization requires a complex compiler infrastructure. The exploitation of memory hierarchies for WCET reduction requires detailed information about memories, usually available only to the linker, inside the compiler. Obviously, a tight integration of static WCET analyses into the compiler is key for WCET-aware optimization. Since WCET analysis relies on flow facts, WCC provides sophisticated mechanisms for source-level flow fact specification. Besides manual flow fact specification by the user, WCC includes a highly performant loop analyzer deriving flow facts automatically. A back-annotation module is finally required to perform WCET-aware optimization at source code level.

On top of this infrastructure, the WCET-aware optimizations function inlining, loop unrolling, loop unswitching, register allocation, scratchpad allocation and cache partitioning are integrated into WCC. Each of them reduces the WCETs of typical benchmarks between 6% and 40% on average, clearly showing the performance of the entire WCC framework.

In the future, WCET-aware optimizations for multi-task and multi-core systems will be integrated into WCC. Besides pure WCET-aware optimizations, we will consider multi-objective optimizations to achieve trade-offs between real-time constraints and other optimization criteria like e.g., energy dissipation.

Acknowledgements The authors would like to thank AbsInt Angewandte Informatik GmbH for their support concerning WCET analysis using the aiT framework. The research leading to these results has received funding from the European Community's ArtistDesign Network of Excellence and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement no. 216008 and from the German Research Foundation DFG under reference number FA 1017/1-1.

References

1. AbsInt Angewandte Informatik GmbH: aiT: Worst-case execution time analyzers. http://www.absint.com/ait (2012)
2. Banakar R, Steinke S, Lee BS, Balakrishnan M, Marwedel P (2002) Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In: Proceedings of the international symposium on hardware/software codesign (CODES), Estes Park, USA, pp 73–78
3. Commission of the European Community: Updated Work Programme 2009 and Work Programme 2010, ICT – Information and Communications Technologies. http://cordis.europa.eu/fp7/ict (2009)
4. Edwards SA (2001) Dataflow languages. http://www.cs.columbia.edu/~sedwards/classes/2001/w4995-02/presentations/dataflow.ppt (2001)
5. Falk H (2009) WCET-aware register allocation based on graph coloring. In: Proceedings of the 46th design automation conference (DAC), San Francisco, pp 726–731
6. Falk H (2010) WCET-aware compilation. http://ls12-www.cs.tu-dortmund.de/research/activities/wcc (2010)
7. Falk H, Kleinsorge JC (2009) Optimal static WCET-aware scratchpad allocation of program code. In: Proceedings of the 46th design automation conference (DAC), San Francisco, pp 732–737
8. Falk H, Lokuciejewski P, Theiling H (2006) Design of a WCET-aware C compiler. In: Proceedings of the 4th IEEE workshop on embedded systems for real-time multimedia (ESTIMedia), Seoul, Korea, pp 121–126
9. Holsti N, Gustafsson J, Bernat G et al (2008) WCET tool challenge 2008: Report. In: Proceedings of the 8th international workshop on worst-case execution time analysis (WCET). Prague, Czech Republic
10. Informatik Centrum Dortmund e. V.: ICD-C Compiler framework. http://www.icd.de/es/icd-c (2012)
11. Informatik Centrum Dortmund e. V.: ICD-LLIR Low-level intermediate representation. http://www.icd.de/es/icd-llir (2012)
12. IT Facts: Home page. http://www.itfacts.biz (2012)
13. Lee EA (2005) Absolutely positively on time: What would it take? Embedded Syst Column, IEEE Comp 38(7):85–87
14. Lee EA (2006) The future of embedded software. ARTEMIS conference, Graz, http://ptolemy.eecs.berkeley.edu/presentations/06/FutureOfEmbeddedSoftware_Lee_Graz.ppt (2006)
15. Lee EA (2007) Computing foundations and practice for cyber-physical systems: A preliminary report. Tech. Rep. UCB/EECS-2007-72, EECS Department, University of California, Berkeley
16. Lokuciejewski P, Marwedel P (2009) Combining worst-case timing models, loop unrolling, and static loop analysis for WCET minimization. In: Proceedings of the 21st euromicro conference on real-time systems (ECRTS), Dublin, Ireland, pp 35–44
17. Lokuciejewski P, Cordes D, Falk H, Marwedel P (2009a) A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In: Proceedings of the international symposium on code generation and optimization (CGO), Seattle, pp 136–146
18. Lokuciejewski P, Gedikli F, Marwedel P (2009b) Accelerating WCET-driven Optimizations by the Invariant Path – a case study of loop unswitching. In: Proceedings of the 12th international workshop on software & compilers for embedded systems (SCOPES), Nice, France, pp 11–20
19. Lokuciejewski P, Gedikli F, Marwedel P, Morik K (2009) Automatic WCET reduction by machine learning based heuristics for function inlining. In: Proceedings of the 3rd workshop on statistical and machine learning approaches to architectures and compilation (SMART), Paphos, Cyprus, pp 1–15
20. Marwedel P (2011) Embedded system design. 2nd edition, Springer, Heidelberg
21. National Research Council (2001) Embedded, everywhere. National Academies Press

22. Plazar S, Lokuciejewski P, Marwedel P (2009) WCET-aware software based cache partitioning for multi-task real-time systems. In: Proceedings of the 9th international workshop on worst-case execution time analysis (WCET), Dublin, Ireland, pp 78–88
23. Steinke S, Wehmeyer L, Lee BS, Marwedel P (2002) Assigning program and data objects to scratchpad for energy reduction. In: Proceedings of design, automation and test in Europe (DATE), Paris, France, pp 409–415
24. Wehmeyer L, Marwedel P (2005) Influence of memory hierarchies on predictability for time constrained embedded software. In: Proceedings of design automation and test in Europe (DATE), Munich, Germany, pp 600–605

Chapter 8
System Level Performance Analysis for Real-Time Multi-Core and Network Architectures

Jonas Rox, Mircea Negrean, Simon Schliecker, and Rolf Ernst

8.1 Introduction

Advances in chip design and communication technology allow the integration of a growing number of functions in distributed embedded systems, ranging from mobile phones through multimedia home platforms to automotive systems. The resulting system complexity makes it a major challenge to build reliable systems, in particular in the context of permanently decreasing time-to-market and production costs. Embedded systems often have to also satisfy real-time requirements, which makes performance verification necessary to exclude critical system failures.

These problems can especially be observed in automotive systems, in which the number and complexity of electronic functions is rapidly increasing. Functions that were introduced as innovations in luxury class cars become standard functions a few years later. With this increase in functions, network topologies are continuously evolving, be it by adding new segments and routing functionalities (gateways) or by changing the protocol (e.g. from CAN to FlexRay). In addition, the semiconductor industry has started offering multi-core solutions for automotive processors which aim at providing new functionality by clustering previously distributed applications into a single chip or to allow the parallelization of complex computations over multiple cores, for example in high-performance domains such as engine control or advanced driver assistance systems. Another source of complexity comes from the supply chains that often contain several companies which design their individual components based on requirement definitions from the OEMs. Clearly, systems' integration has become a key challenge.

J. Rox (✉) · M. Negrean · S. Schliecker · R. Ernst
Institute of Computer and Network Engineering, Technische Universität Braunschweig, D-38106 Braunschweig, Germany
e-mail: rox@ida.ing.tu-bs.de; negrean@ida.ing.tu-bs.de; schliecker@ida.ing.tu-bs.de; ernst@ida.ing.tu-bs.de

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_8,

To ease the design of automotive systems in the future, the AUTOSAR partnership [1], an alliance of OEM manufacturers and Tier-1 automotive suppliers with many associates, has established an industry standard for automotive E/E architectures. AUTOSAR not only provides a new degree of flexibility giving new opportunities for system optimization, but does also define a common formal model used to describe automotive systems.

Such modeling standards can build the foundation for serious application of formal methods, i.e. to investigate the timing and explore mapping options. Thus, methods that have been suggested in research for some time, now become applicable to actual productive environments. Different formal methods are available, with simulation still being the method of choice for performance verification in industrial designs today. But its applicability is limited when no executable code or architecture is available. Furthermore, it fails to deliver robust guarantees, as corner-case behavior is not covered. Instead, formal analysis has been introduced as a modeling alternative to close these verification gaps. By computing conservative performance bounds, formal analysis guarantees that the system's performance behavior during any execution scenario completely lies within these bounds. This is especially important for critical real-time systems. Since it can cope with incomplete data specification, formal analysis can be applied early in the design process. It is, hence, suitable for early design space exploration and optimization.

In the past decade, several approaches to formal analysis have been proposed, based on different system models. Each of these approaches has shown high modeling and analysis accuracy for specific application domains, but less applicability for other application areas. Examples are holistic performance analysis [2, 3] and compositional performance analysis [4,5]. Despite its better ability to take global system effects into account, the holistic performance analysis approach lacks flexibility due to the need to adapt scheduling analysis to each potential system configuration. The compositional performance analysis approach on the other hand, is less accurate for system configurations requiring global system knowledge, but it offers higher flexibility and scalability making it more appropriate for the analysis of typical heterogeneous industrial designs. However, to obtain valuable results, the underlying analysis model used for compositional performance analysis must fit to the system under verification. This makes the continuous development of extensions necessary to adapt the used models to the system structures we find in industry today and in the near future. In this chapter, we will highlight two recent extensions to an existing formal performance analysis approach that cover the analysis of multi-core architectures and the incorporation of modern communication stacks into system analysis. First, we give a brief overview of existing compositional system level performance analysis techniques for distributed systems and, then, briefly introduce the common system model on which these methods rely, in Sect. 8.2. In Sect. 8.3 we identify the shortcomings of the traditional analysis models with respect to capturing the timing effects of the communication stack, as defined in AUTOSAR and we present an appropriate extension to an existing analysis model to remedy these shortcomings. In Sect. 8.4 we discuss performance analysis challenges which arise with the advent of the multi-core components and elaborate on another extension of the existing

analysis model dedicated to such architectures. Finally, in Sect. 8.5 we survey experiments from academic and industrial use cases and afterwards we draw conclusions.

8.2 Compositional System Level Performance Analysis

The foundation for compositional system level performance were layed by Gresser [6] in the group of Georg Frber, TU Munich. Tasks were defined as locally analyzable entities connected via event streams. Event vectors were used to model the task activations sequences. Events stream models suitable to capture the time relation between events and local analysis coupled by such event stream models are the key ingredients of compositional performance analysis today.

Gresser performed EDF scheduling analysis for tasks sets with periodic, jitter and burst activations to demonstrate the efficiency of his model. From the mathematical point of view, the stream representations are used to capture the dependencies between the set of equations that describe the individual components timing. Compared to the holistic approach, the compositional models are modularly structured with respect to the architecture. This not only significantly helps the designers to understand the complex dependencies in the system, but it also enables a surprisingly simple solution.

Note that the compositional analysis approach is conceived to be suitable also for addressing systems that do not adhere to a composition strategy (as suggested in [7] or [8]), i.e. where dependencies between components exist that make an isolated component-based verification intractable. The event-model based analysis allows to efficiently capture component inter-dependencies (for example due to the use of shared resources), allowing the composition of the local *analysis* modules (i.e. local timing analyses). The analysis composition then works by integrating different local scheduling analysis techniques into an iterative system-level analysis (see Fig. 8.1).

The output event stream of a given component which is derived based on the input event stream and the results of the local analysis, turns into an input event stream of a connected component. Schedulability analysis can be seen as a flow-analysis problem for event streams, and, in principle, can be iteratively solved using event stream propagation. Following the same principle, two other compositional approaches have emerged.

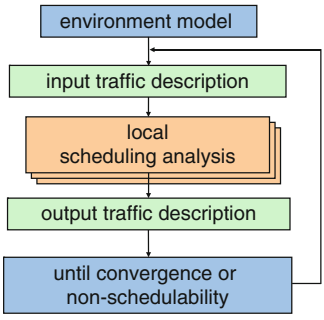


Fig. 8.1 Compositional system level performance analysis loop

Thiele et al. [9] defined a similar model, in which Gresser's event vectors are replaced with arrival curves. Furthermore, the processing capacity of the resources has been modeled using service curves. The new performance analysis combines the arrival and service curves to determine the timing behavior at component's outputs. The model was named Real-Time Calculus (RTC), due to its close relations to Network Calculus [10].

Richter et al. [11] proposed a compositional analysis model based on event model interfaces, called SymTA/S. The interfacing between the local components is realized using a set of comprehensible standard event models. Event model interfaces (EMIF) and event adaptation functions (EAF) are used to perform transformations between event models. The analysis compositionality is realized by combining local scheduling analysis techniques and event model propagation into a system-level analysis loop as depicted in Fig. 8.1.

Since their first introduction, these approaches have been constantly extended. For example, based on the event vectors introduced by Gresser, Albers et al. [12,13] defined a new event model to describe arbitrary event arrivals in a very compact way, by using a hierarchical parameterization. For RTC, as well as for SymTA/S, there have also been numerous extensions, e.g. [14–17].

Despite their different way of actually implementing the analysis, the approaches introduced above share a common underlying way of modeling a system and capturing its properties:

- A software description, typically a set of tasks or a set of communication entities, along with parameters such as priority, core execution time, etc. for tasks, and time slots, frame length, etc. for communication
- An activation model for the *primary stimulation* of tasks and communication (e.g. periodic), i.e. stimulation not depending on any other tasks in the system under consideration, as well as the *secondary stimulation* of tasks and communications (and vice versa) depending on other tasks along event-triggered paths
- For system-level approaches: a hardware description, typically a set of CPU nodes, buses/networks, and connectivity and mapping information (tasks to CPU nodes, communication to buses), along with parameters such as processing power, scheduling strategy, protocol configuration, etc.

In the next sections, we take a closer look on two specific features of upcoming automotive systems which make adaptations to the existing analysis model mandatory to obtain valuable analysis results. We also discuss recently developed corresponding extensions to compositional system level performance analysis.

8.3 Model Extension for Communication Stacks

The traditional compositional analysis models used by the formal approaches introduced in the previous section, models bus communication by a simple communication task that is directly activated by the sending task, and which directly

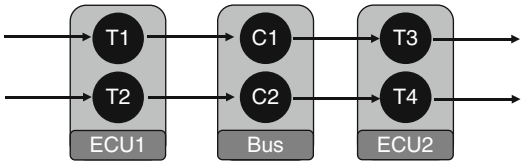


Fig. 8.2 Traditional model

activates the receiving task. Figure 8.2 shows a simple example system that uses this model for communication, where each output event streams of the sending tasks, $T1$ and $T2$, directly becomes the input event streams of the communication tasks on the bus.

However, modern communication stacks employed in todays embedded control units (ECU) make this abstraction inadequate. For instance, AUTOSAR defines a detailed API for the communication stack [18], including several frame transmission modes (direct, periodic, mixed, none) and signal transfer properties (triggered, pending) with key influences on communication timing. Hence, the transmission timing of messages over the bus doesn't have to be directly connected to the output behavior of the sending tasks anymore, but it may be completely independent of the tasks output behavior, e.g. the sending of a message is triggered by a periodic timer. It is even possible to send several output signals in one message.

8.3.1 Timing Implications of Modern Communication Layers

Figure 8.3 shows an adequate model accounting for the presence of communication layers on the sending and receiving ECUs. Instead of directly triggering the message transmission (activating the communication task), $T1$ and $T2$ write their output data, called signals here, into a register provided by the communication layer, overwriting previous output data. Each register is assigned a fixed position in a so called frame. The communication layer triggers the sending of a frame (modeled by the execution of the communication task C), which then transmits all register values assigned to that frame. A frame can be of different types: *periodic*, *direct* or *mixed*. The signals generated by the sending tasks can either be defined *triggering* or *pending*. The sending of frames is triggered according to the following rules: When the frame type is *periodic*, frames are just send periodically, not influenced by the signal output timing of the sending tasks. If the frame type is *direct*, for each arrival of a *triggering* signal, a frame is send. And finally a mixed frame is a combination of the two first ones, so it is transmitted periodically and also whenever a *triggering* signal arrives.

When a frame is received by the communication layer of the receiving ECU, it transmits the included data into registers, again overwriting the values stored there previously. Either does the receiving task fetch the register value from time to time or each time new data is written into the register, the process is activated, e.g. by an interrupt.

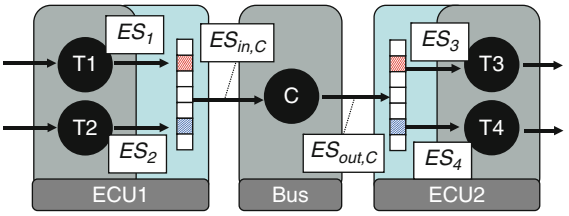


Fig. 8.3 Communication via ComLayer

Hence, depending on the configuration, the activation timing of the communication task C in our analysis model, may or may not be influenced by the output event streams ES_1 and ES_2 of the tasks $T1$ and $T2$. Some kind of *join* operation is needed that, based on the communication layers configuration determines the activation timing of the communication task. Since, it is also possible to map several signals to the same frame, there isn't necessarily a one-to-one relation between signal and frame transmission, instead, each frame can contain one, several or no new signal. This information must somehow be captured to enable the *decomposition* of the previously merged streams on the receiving side to determine viable bounds for the signal arrival times. All the described are not only permitted but actually used in automotive software.

By using traditional event stream models like the ones proposed in [4, 6, 19] or [12] these effects cannot be appropriately captured. Using event stream concatenations as presented in [20] or [5] the input event stream of the communication task could be appropriately determined and described with the existing flat event streams. But the individual timing of signal arrivals on the receiving side can only be bounded by the timing of arriving frames. Using type rate curves [16] or different execution modes [21], different execution modes of the communication task could be considered, but the problem of decomposing the individual streams on the receiving side still persists.

That means, to conservatively bound the maximum number of signal arrivals in a given time interval when only flat event streams are used, it must be assumed that every arriving frame contains a new signal. To conservatively bound the minimum number of signal arrivals it must be assumed that none of the frames contains a new signal, since no other information is available in the output event stream of the communication task. Obviously, this can lead to large overestimation.

8.3.2 *A Model for Composition and Decomposition of Event Streams*

To be able to cover the previously described hierarchical structure of the communication behavior, in [22] we introduced hierarchical event streams (HES) modeled by a hierarchical event model (HEM). This HEM determines the timing of message transmissions (activation timing of the communication task), captures the timing

of the signals transmitted by this messages, and most importantly, defines how effects on the message timing influences the timing of the transmitted signals. The latter allows to derive the timing of the individual embedded signal streams when unpacking the signals on the receiving side, giving much tighter timing bounds for arrival timing of signals.

To avoid confusion here, we note that these hierarchical event streams do not require or imply that the scheduling itself is hierarchical. Quite to the contrary, the scheduling of frames on a CAN bus follows a single, non-hierarchical policy, i.e. static priority non-preemptive. We have introduced hierarchical event models to account for the multiplexing nature of the signals that are grouped into one frame within the COM layer. We then use these hierarchical event streams within the analysis of non-hierarchical scheduling using known techniques.

Figure 8.4 illustrates the structure of the hierarchical input event stream of the communication task C resulting from the combination of the output streams of tasks $T1$ and $T2$ from the example depicted in Fig. 8.3. The general idea is, that a hierarchical event stream has one *outer* representation in form of an event stream and for each combined event stream it has one *inner* representation, also in form of an event stream. The relation between the *outer* event stream and the *inner* event streams depends on the hierarchical event stream constructor (HSC) that combines the event streams.

The HSC not only allows to incorporate timing effects of the communication layer, but also defines how the *inner* streams are affected if the timing characteristics of the *outer* event stream are changed, e.g. due to scheduling on the bus. Each of the involved event streams is defined by the functions $\delta^-(n)$ and $\delta^+(n)$ defining

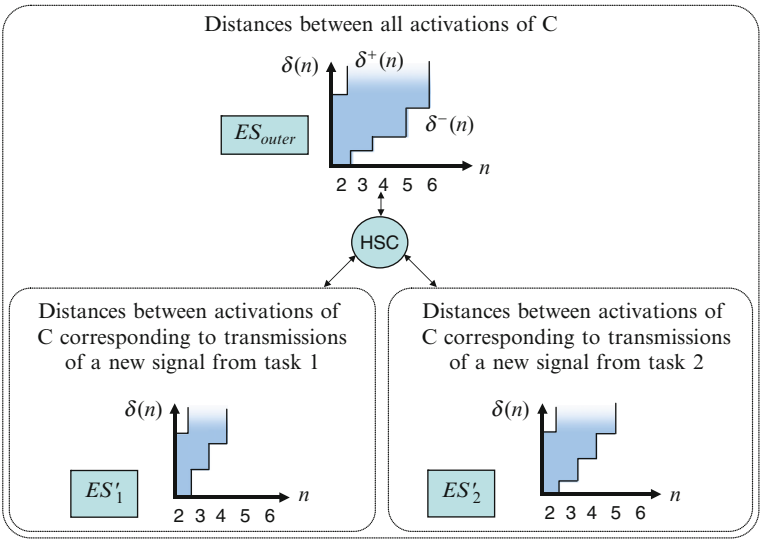


Fig. 8.4 The hierarchical input event stream $ES_{in,C}$ (as denoted in Fig. 8.3)

the minimum, respectively maximum distance between n consecutive events. The events of a specific *inner* event stream model the timing of only those events that are somehow associated with the corresponding input event stream that was combined. Here, the *outer* event stream ES_{outer} models the timing of all activations of the communication task C the two *inner* event streams ES_1' and ES_2' model the timing of only those activations of the communication task C that represent the transmission of a new signal from the task $T1$, respectively task $T2$. An important feature of the HEM is that the local component analysis do not have to be adapted in any way. If a task has a hierarchical input stream, for the local analysis of the component this task is mapped on, the *outer* representation of its hierarchical input stream is used as description for the tasks activation timing. Thus, the response times can be obtained by using the same techniques as used for non-hierarchical event streams, e.g. from the algebraic solution of the response time formulas using the sliding window technique presented by Tindell [23].

The timing of message arrivals on the receiving side is modeled by the *outer* event stream of the hierarchical output model of the communicational task, which is obtained by traditional output model calculation as described in [15], based on the *outer* stream of the hierarchical input model and the analysis results. The timing of signal arrivals from a specific runnable is modeled by the corresponding *inner* stream of the hierarchical output model of the communicational task. These streams are obtained by applying so called update functions on the corresponding *inner* streams of the hierarchical input stream. These update functions essentially adapt the *inner* streams, taking into account the changes to the message timing, e.g. due to scheduling effects.

This output calculation for hierarchical streams has been generalized in [24]. The key idea of the generalized procedure is to express the output model calculation of an event stream as the composition of the two, so called, elementary functions ϑ and σ. Furthermore, for each of the elementary functions, the HSC defines a corresponding update function to calculate the adapted *inner* streams: The function B^{ϑ} and the function B^{σ}.

An important property of these update functions is that they change the *inner* streams only by decreasing (increasing) the minimum (maximum) distance between events and by enforcing a minimum separation between two consecutive events; changes that can again be expressed using only ϑ and σ . Therefore, if one of the *inner* streams is the *outer* stream of another hierarchical event stream, it is assured that its HSC provides some update functions to propagate the changes to its *inner* streams. This allows to handle streams with multiple hierarchical layers in a uniform manner. Figure 8.5 depicts the hierarchical input and output streams of the communication task C from the example shown in Fig. 8.3. On the left side the hierarchical input stream, which consists of the *outer* event stream ES_{outer}, modeling the timing of all activations of the task C and two *inner* event streams ES_1' and ES_2', modeling the timing of only those activations of the task C due to an event originated form the task $T1$, respectively task $T2$, is shown. The scheduling analysis of the communication resource the task C is mapped on uses the *outer* event stream as activating event stream of the task and delivers a corresponding

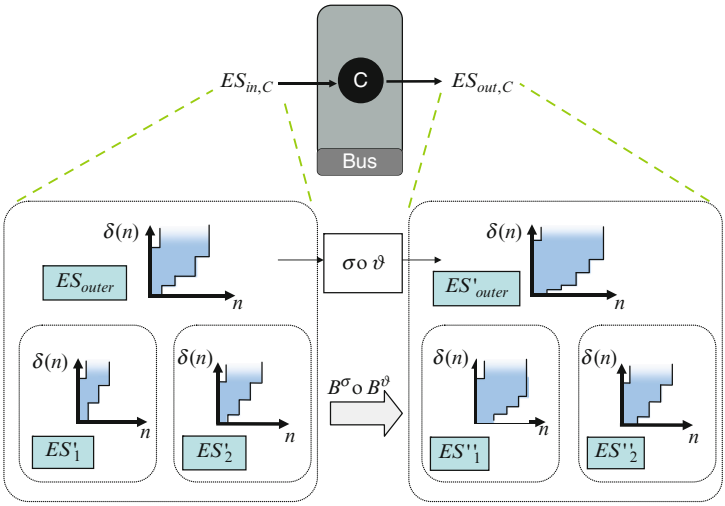


Fig. 8.5 From hierarchical input stream to hierarchical output stream

WCRT and a BCRT. To obtain the hierarchical output stream, first its *outer* stream is calculated by applying the composition of the two elementary operations ϑ and σ with the response time jitter and the minimum distance as parameters on the *outer* stream of the hierarchical input stream. Then, the *inner* streams are adapted accordingly with the composition of the update functions B^{ϑ} and B^{σ} which are defined by the HSC that was previously used to merge the streams.

With the information available in the hierarchical stream at the output of the communicational task, we are now able to determine the activation timing of the individual receiving tasks $T3$ and $T4$ much more precisely. Each *inner* stream directly bounds the timing of those message arrivals that contain a new signal value from the corresponding sending task. So we can directly map these *inner* streams to the input streams of the receiving tasks. This mapping is performed by an hierarchical stream deconstructor (HSD) [24] which also allows to consider additional effects, e.g. a delay of the communication layer of the receiving ECU. It has been shown in [22], that when communication layers are considered in the system performance analysis, the use of the above presented hierarchical event streams can lead to significantly tighter bounds of the activation timing, especially for the receiving tasks. This in turn leads to tighter performance estimations for system characteristics including end-to-end deadlines.

8.4 Performance Estimation of Multi-Core Systems

The demand for powerful domain controllers in real-time embedded systems encourages the introduction of multi-core processors. While these generally deliver additional performance energy and cost efficiently, their application also introduces

a new level of inter-core dependencies that was not previously observed in distributed systems. The use of physically shared hardware (such as the shared memory) or synchronization via logical resources (i.e. semaphores) introduces dependencies between task executions on different cores which may lead to hard-to-find timing problems including missed deadlines that can make the entire system fail [25,26]. Applying such components in reliable real-time systems, for example in the automotive domain, requires careful investigation of the implications on system timing.

8.4.1 Timing Implication of Multi-Core Components

Multi-core components provide a new dimension of scheduling freedom, as tasks can now be assigned over time *and* over different processors. A multitude of scheduling policies takes advantage of this freedom to achieve high utilization under real-time guarantees [27]. For most, synchronization schemes are available to support the use of shared resources [28]. The choice of the optimal scheduling scheme highly depends on the actual set of tasks in the system, in particular their number, local execution requirements, use of the shared resources, or inter-task data dependencies [26]. Literature has shown that scheduling schemes are incomparable in general, in the sense that different schemes provide optimal solutions for different task sets (e.g. [29, 30]). Considering the automotive dedicated architectures, static task partitioning is expected to first find its way into automotive multi-core real-time systems [31], as the benefits of reusing existing applications, tools, and methods outweighs the potential loss in efficiency. But also under the partitioned scheme, a new look at system timing is required, as the use of shared resources introduces previously unknown dependencies. This is illustrated in the following example.

Assume that a multi-core system has been statically partitioned into two task sets on fixed priority preemptive operating systems as depicted in Fig. 8.6. The tasks mapped on the two available processors make use of a common shared resource, which we assume to be a memory in the following example.

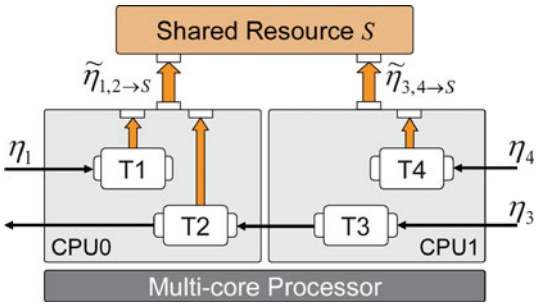


Fig. 8.6 Dual-core system with four tasks and a shared resource

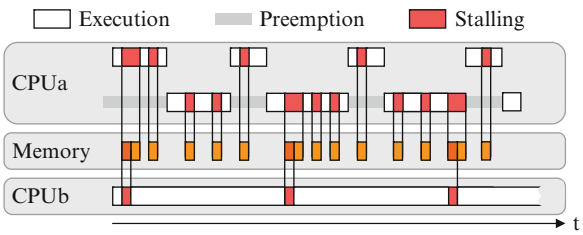


Fig. 8.7 Conflicting accesses among tasks mapped on different processors

Under these assumptions, the worst case response time of a task mapped on any of the processor cores (CPUa or CPUb in Fig. 8.6) is classically determined by the task's worst case execution time plus the maximum amount of time the task can be kept from executing due to preemptions by higher priority tasks. Furthermore, when a task performs accesses to shared resources it is additionally delayed (blocked) when waiting for the required resources. When tasks mapped on the same processor, as can be seen in Fig. 8.7 on CPUa, access the same remote memory, the finishing time of the lower priority task increases, due to itself fetching data from the remote memory, and due to the prolonged preemptions by the higher priority task (as in the given example, its requests also stall the processor). Additionally, whenever the memory is also used by a task on the other processor (CPUb in Fig. 8.7), the first CPU can be stalled for a longer time, further increasing the task response times. Moreover, because the execution of the low priority task is now stretched over time, it may also suffer from additional preemptions by the high priority task. These effects can add up and may lead to deadline violations.

This example shows that the task's response times are influenced by the delay caused by the use of shared resources. In the general case, this depends on the amount of traffic imposed on the shared resources by e.g. *other* processors or hardware units in the system. The respective local analysis of the other processors however also requires the shared resource delay, which closes a cycle: The timing interference in the system translates into a mutual dependency between the local analyses of the different cores. To provide the reliable upper bound on the resource access timing which is required in real-time systems, additional measures must be taken.

Previous approaches addressing the use of shared memories rely on a bounded influence between the processors during run-time e.g. by time division of the bus schedule [32]. For priority-based arbitration, analytical methods are also available [33]. For this, the amount of resource accesses per task activation need to be known, as well as the number of task activations. The latter however can not trivially be bounded in event driven multiprocessor systems where, for example, a task activation is the result of a message produced on another processor.

The problem is aggravated when the shared resource is a memory, and accesses to it are the result of local cache misses. In this case, timing of the memory accesses is the highly dynamic and the joint scheduling will lead to competition for the local cache lines. Apart from cache partitioning or locking [34], a solution to this

problem is to rely on non-preemptable tasks segments [35], or to rely on results from the research on single-processor worst case execution time analysis, in which caches and cache-related preemption delays have already been investigated [36]. For logical protection of shared resources, i.e. the execution of exclusive critical sections, a number of high-level protocols have been proposed [37, 38]. These methods provided an arbitration scheme, possibly tailored towards a particular task scheduler, and an analysis to calculate the upper bounds on the time before a lock is granted. As opposed to single processor protocols (such as the priority ceiling protocol [37]), the blocking time in multiprocessor setups usually depends on the pattern of task activations.

This dependency is a major hurdle in a general multiprocessor shared resource analysis. Given event-driven task activations and dynamic scheduling, the amount of interfering task activations is unknown without a system level analysis, which may only be done when every task's resource usage is known — this again leads to a cyclic problem dependency. Thus the problem can not be generally solved with classical methods.

8.4.2 System-Level Performance Analysis in the Presence of Multi-Core Systems

The previous section has highlighted the complex dependencies that arise in multi-core setups through the common use of shared resources. Other than in traditional system analysis, a component, such as a task or a processor, can not be verified in isolation, because elementary properties are lost when the components are integrated. In order to facilitate a structured design process that leads to reliable multi-core systems, we have developed a methodology that allows to capture the inter-core timing dependencies and calculate bounds on the task response times and output event models even in the presence of dynamic scheduling and *secondary* shared resources.

For modeling purposes, we adopt the concept of *requesting tasks* as in [39]. In addition to the local execution, such a task performs *operations* during its execution, whenever it requires access to a *shared resource*. The task issues a *request* and may only continue execution after the request was e.g. transmitted over the bus, processed on the remote component and transmitted back to the requesting source. We differentiate between local *scheduling* and remote shared resource *arbitration*.

The key idea of the proposed general solution is to rely on the event model propagation concept of the compositional analysis (as mentioned in Sect. 8.2), to express not only the task activations but also the load imposed on the secondary resources. This allows to separate the analysis into three major building-blocks which will be part of the extended system-level analysis procedure as depicted in Fig. 8.8 and discussed in what follows.

1. *The derivation of bounds on the amount of shared resource operations.*

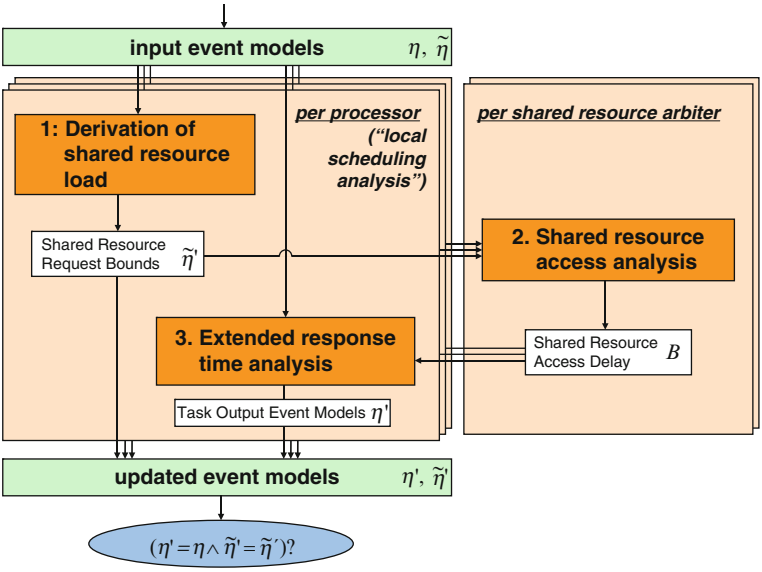


Fig. 8.8 Refined analysis procedure for shared secondary resources

To be able to compute the possible interference on a shared resource, event models on the maximum amount of requests issued by each task are required. By considering the pattern of task activations and the distance between requests issued by each task instance, the overall load imposed on the shared resource can be derived for each task and all tasks on a processor.

The load imposed on the shared resource from a single processor is the aggregation of the traffic inflicted by all concurrent task activations. This can be expressed using the functions $\tilde{\eta}^+(\Delta t)$ and $\tilde{\eta}^-(\Delta t)$ denoting the maximum and minimum number of requests in any time interval of size Δt. In Fig. 8.6, the traffic to the shared resource is denoted with $\tilde{\eta}$ to differentiate it from task activating event models η.

For each individual task instance, the amount of issued requests can be bounded by closely investigating the task's internal control flow as discussed in [12, 40]. For example, a task may fetch data each time it executes a for-loop that is repeated several times. By multiplying the maximum number of loop iterations with the amount of fetched data, a bound on the total number of memory accesses can be derived.

Depending on the actual system configuration, using solely the upper bound on the number of shared resource requests per task instance may translate into an assumed burst of requests that may not occur in practice, resulting in an overestimated shared resource load. Improved shared resource loads can be obtained by considering that there is a minimum execution time between successive requests. Tasks scheduled on the same processor will then execute in alternation, which leads to to a joint load as derived in [36].

2. *The computation of the shared resource access delay.*

Furthermore, it is necessary to derive the latency for a set of requests to the shared resource, given that the load imposed by other processors is bounded by event models $\tilde{\eta}$ as discussed above.

On the shared resource, coinciding requests are arbitrated according to a specific policy, which leads to specific delays for each request. In order to conservatively bound the worst-case shared resource delay one can rely on the derivation of the *aggregate busy time* that represents the total time for which at least one of the task's requests is not finished [25]. A straight-forward solution to compute the aggregate busy time is to sum the times during which any component in the system may occupy the shared resource within a given time window. This load-based concept is conservative for any work-conserving resource arbitration protocol (such as first-come-first-served).

Considering more sophisticated arbitration policies requires refined aggregate busy time calculation. For example the worst-case latency per transaction in a lock-free database was investigated in [41], and the aggregate latency of multiple shared resource operations under the lock-based Multiprocessor Priority Ceiling Protocol (MPCP) [37] was investigated in [25]. Alternatively, the required parameters can also be obtained from measurements (as proposed for memory accesses in [42]).

3. *The inclusion of the aggregated shared resource delay in the worst-case response time analysis of each task.*

As a further building-block, the delay of accessing the shared resource needs to be considered in the tasks' local response time calculations. When the shared resource is not immediately available, this may be treated differently by various processor or operating system implementations. In the case of memory accesses for example, many processors offer a multi-cycle operation that stalls the complete processor until the transaction has been processed by the system [43]. Unfortunately, such active waiting increases the processor load, possibly leading to infeasible schedules. In multithreading cores, a set of hardware threads may allow to perform a quick context switch to another thread that is ready, effectively keeping the processor utilized [44]. While this behavior usually has a beneficial effect on the average throughput of a system, multithreading demands for caution in priority based systems with reactive or control applications. There, the worst case response time of high priority tasks may even increase [39].

The classical response time calculations using the busy window approach, can appropriately be extended to include the previously calculated aggregate busy time of the resource accesses [25]. For priority based systems this can be again done by a simple addition [39]. Note that the aggregate busy time of the requests is a function of the size of the time window during which a task executes, i.e. a long execution of a task can experience more interference on the shared resource than a short one. Thus, the aggregate busy time needs to be calculated iteratively with the response time in most setups.

The aggregate busy time model of resource delays allows a useful orthogonalization of concerns, as now the timing of the shared resources, and its effect on the

local response time can be investigated separately. This allows to apply specialized analyses to either and, more importantly, to compose a multi-core system with different scheduling policies, e.g. the resource arbitration protocol (and its analysis) may be changed due to an update while the local scheduler (and its analysis) remains unchanged.

The final building block of the extended analysis procedure is *the derivation of an updated output event model*. Using the task's overall timing behavior bounded in the previous steps, one can derive the task's output event model as explained in Sect. 8.2.

All of these steps can be based on the current models of the event timing in the systems provided by the iterative compositional analysis. As these are possibly refined during the analysis procedure, both the analysis of the shared resource timing and local scheduling have to repeated for all dependent resources. This procedure may have to be repeated several times, until a change of a task activating event model does not result in a change in the interference of a given task (illustrated in the cycle of Fig. 8.8). As in the basic approach of Fig. 8.1 all considered event streams become increasingly more generic with each analysis iteration, the procedure either converges to a fixed-point, or the system can not be deemed schedulable.

This section has provided a method and details on the basic building blocks to deliver real-time guarantees for systems comprising multi-core components. The presented methods allow the application of heterogeneous processor scheduling and resource arbitration policies in hard real-time systems by relying on the event stream model to express not only the task activation pattern but also the resource accesses.

8.5 Comparisons and Applications

The previous sections have provided solutions for current challenges in performance verification of embedded safety critical systems. We have highlighted the need to extend existing performance models to map to the complex reality of modern systems. Additionally, two extensions have been presented to extend the applicability of compositional performance analysis to cover hierarchical communication and multi-core components with shared resources. This approach has been evaluated and used in a various synthetic and real-world examples, some of which have been published.

General comparisons with other approaches are always difficult due to their diverging capabilities. An effort has been made in [45], where the compositional analysis approaches SymTA/S and MPA, the holistic analysis with MAST, a timed automata based analysis with Uppaal, and pure simulation have been compared with means of simple examples that each approach is able to handle. As can be expected, different accuracy is achieved for different scenarios. Only timed automata consistently capture the exact behavior, but at the cost of high computational effort even for the very small examples. SymTA/S captures all examples with pretty good

accuracy at a very fast analysis time. A similar case study was conducted in [46] on a multimedia system, essentially leading to the same observations.

These comparisons can only capture the common set of tool functionalities, which leads to very simple system models. In practice, systems are far more complex and typically show a diverse amount of modeling challenges, which have been identified in [47]. Automotive systems in particular continue to consist of heterogeneous components, with hierarchical layers of hardware, software, and communication. To combine the strengths of different approaches, the compositional analyses allow to be cross-integrated, as has been done with MPA and SymTA/S [48], or MPA and simulation [49]. This easy composability is key to tackle complex real-world systems. While other approaches may also be extensible, the composable structure greatly simplifies this process. For automotive systems, this allows to integrate existing, specialized methods and tools for commercial libraries (such as OSEK, CAN, FlexRay) and upcoming architectures (such as multi-core).

Besides the industrial use cases the framework has also been applied in research case studies, such as on a hierarchical multiprocessor, multithreading setup [50]. Moreover, several experiments for synthetic systems have been published, which have stressed the applicability in a number of challenging setups, e.g. considering distributed systems [5], complex task interaction [24], or shared resource optimization [51].

The extended system-level performance analysis procedure presented in Sect. 8.4 has been applied in a number of hypothetical multi-core setups. Different multicore systems have been investigated by assuming different number of cores, tasks and shared resources and different resource arbitration policies [26]. These experiments have underlined the timing implication of multi-core components and have proven the applicability of the proposed approach.

Compositional system-level performance analysis methods have shown to be well suited to perform design space exploration and to investigate "what-if" scenarios. For example, the methodology presented in [52] can be used to quickly explore a great number of system configurations (e.g. different task mappings, priorities, time wheel sizes) with optimization objectives such as minimizing the end-to-end delay of critical paths or minimizing the overall buffer requirements. Moreover, this allows to evaluate and optimize a system with respect to its robustness towards unanticipated parameter changes, which can occur during the design process or also after the system has been deployed [53].

8.6 Conclusion

It is now widely accepted that appropriate performance and timing analysis methods is key to building reliable embedded systems. In the automotive domain, the advent of standards, like AUTOSAR, creates a basic ground for the application of formal performance methods in the industrial design process. The research community

has already proposed several formal system-level performance analysis models and methods.

We have identified two key limitations of the existing formal analysis solutions introduced by modern communication structures and complex multi-core components. For both we have discussed extensions to an existing system-level performance analysis approach. These extensions allow reliable predictions of the system performance, providing an important building-block for the design of future safety-critical systems.

References

1. http://www.autosar.org, "Autosar partnership," Internet.
2. Tindell K, Clark J (1994) Holistic schedulability analysis for distributed hard real-time systems, Microproc Microprogram 40(2-3):117–134
3. Gutiérrez J, García J, Harbour M (1997) On the schedulability analysis for distributed hard real-time systems. Proceedings of the 9th euromicro workshop on real-time systems, Toledo, Spain, pp 136–143
4. Chakraborty S, Künzli S, Thiele L (2003) A general framework for analysing system properties in platform-based embedded system designs. Design, Automation and Test in Europe Conference and Exhibition, pp 190–195
5. Henia R, Hamann A, Jersak M, Racu R, Richter K, Ernst R (2005) System Level Performance Analysis – The SymTA/S Approach. IEE Proc Comput Digital Techniq 152(2):148–166
6. Gresser K (1993) An event model for deadline verification of hard real-time systems. In: Proceedings of the 5th euromicro workshop on real-time systems, Oulu, Finland, pp 118–123
7. Bensalem S, Bozga M, Sifakis J, Nguyen T (2008) Compositional verification for component-based systems and application. Automated Technol Verification Anal 5311:64–79
8. Puschner P, Schoeberl M (2008) On composable system timing, task timing, and WCET analysis. In: Proceedings of the 8th international workshop on worst-case execution time (WCET) analysis, Prague, Czech Republic
9. Thiele L, Chakraborty S, Naedele M (2000) Real-time calculus for scheduling hard real-time systems. Circuits and Systems, 2000. Proceedings of the international symposium on ISCAS 2000, Geneva, vol 4, pp 101–104
10. Le Boudec J, Thiran P (2001) Network calculus: a theory of deterministic queuing systems for the internet Springer-Verlag Berlin, Heidelberg ©2001
11. Richter K, Racu R, Ernst R (2003) Scheduling analysis integration for heterogeneous multiprocessor SoC. In: Proceedings of the 24th IEEE real-time systems symposium (RTSS), Cancun, Mexico, December 2003
12. Albers K, Bodmann F, Slomka F (2006) Hierarchical event streams and event dependency graphs: A new computational model for embedded real-time systems. In: Proceedings of the 18th euromicro conference on real-time systems (ECRTS). IEEE Computer Society, Washington, DC, pp 97–106
13. Albers K, Bodmann F, Slomka F (2008) Advanced hierarchical event-stream model. In: Proceedings of the euromicro conference on real-time systems (ECRTS)
14. Henia R, Ernst R (2006) Improved offset-analysis using multiple timing-references. In: Proceedings of the conference on design, automation and test in Europe: Proceedings, pp 450–455
15. Schliecker S, Ivers M, Staschulat J, Ernst R (2006) A framework for the busy time calculation of multiple correlated events. In: Sixth International Worst Case Execution Time Wworkshop

16. Maxiaguine A, Künzli S, Thiele L (2004) Workload characterization model for tasks with variable execution demand. In: Proceedings of design automation and test in Europe, Paris, France
17. Wandeler E, Maxiaguine A, Thiele L (2006) Performance analysis of greedy shapers in real-time systems. In: Proceedings of the conference on design, automation and test in Europe: Proceedings, pp 444–449
18. AUTOSAR (2006) Autosar specification of communication v. 2.0.1, autosar partnership
19. Richter K (2004) Compositional scheduling analysis using standard event models, Ph.D. dissertation, Technical University of Braunschweig
20. Wandeler E (2006) Modular performance analysis and interface-based design of embedded systems, Ph.D. dissertation, Swiss Federal Institute of Technology
21. Jersak M, Henia R, Ernst R (2004) Context-aware performance analysis for efficient embedded system design. In: Proceeding of design automation and test in Europe
22. Rox J, Ernst R (2008) Modeling event stream hierarchies with hierarchical event models. In: Proceedings of the design, automation and test in Europe (DATE 2008) Munich, Germany, March 2008
23. Tindell KW, Burns A, Wellings AJ (1994) An extendible approach for analyzing fixed priority hard real-time tasks, Real-Time Syst 6(2):133–151
24. Rox J, Ernst R (2008) Construction and deconstruction of hierarchical event streams with multiple hierarchical layers. In: Proceedings of the euromicro conference on real-time systems (ECRTS 2008), Prague, Czech Republic, July 2008
25. Schliecker S, Negrean M, Ernst R (2009) Response time analysis on multicore ECUs with shared resources, IEEE Trans Industrial Inform 5(4):402–413
26. Negrean M, Schliecker S, Ernst R (2010) Timing implications of sharing resources in multicore real-time automotive systems. In: SAE world congress. SAE International, Detroit, MI
27. Carpenter J, Funk S, Holman P, Srinivasan A, Anderson J, Baruah S (2003) A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms. In Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Joseph Y-T Leung (ed). Chapman Hall/ CRC Press. 2004.
28. Brandenburg B, John M, Aaron Leontyev H, James H (2008) Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? Real-Time and Embedded Technology and Applications Symposium, RTAS'08. IEEE, pp 342–353
29. Andersson B, Jonsson J (2000) Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. Proceedings of the seventh international conference on real-time systems and applications (RTCSA'00), p. 337
30. Baker T (2006) A comparison of global and partitioned EDF schedulability tests for multiprocessors. International Conference on Real-Time and Network Systems (RTSN), pp 119–130
31. AUTOSAR GbR, "AUTOSAR Release v4.0," http://www.autosar.org/, January 2010
32. Rosen J, Andrei A, Eles P, Peng Z (2007) Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip, Real-Time Systems Symposium (RTSS 2007). 28th IEEE International, pp 49–60
33. Henriksson T, van der Wolf P, Jantsch A, Bruce A (2007) Network calculus applied to verification of memory access performance in SoCs. Workshop on Embedded Systems for Real-Time Multimedia, 2007
34. Puaut I, Decotigny D (2002) Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In: Proceedings of the 23rd IEEE real-time systems symposium (RTSS). Citeseer, pp 114–123, 2002
35. Pellizzoni R, Caccamo M (2010) Impact of peripheral-processor interference on WCET analysis of real-time embedded systems, IEEE Trans Comput 59(3):400–415
36. Schliecker S, Negrean M, Ernst R (2010) Bounding the shared resource load for the performance analysis of multiprocessor systems. In: Proceedings of design, automation, and test in Europe (DATE), Dresden, Germany, March 2010
37. Rajkumar R (1991) Synchronization in real-time systems: A priority inheritance approach. Kluwer, Norwell, MA

38. Devi U, Leontyev H, Anderson J (2006) Efficient synchronization under global edf scheduling on multiprocessors. Proceedings of the 18th euromicro conference on real-time systems, pp 75–84, 2006
39. Schliecker S, Ivers M, Ernst R (2006) Integrated analysis of communicating tasks in MPSoCs. Proceedings of the 4th international conference on hardware/software codesign and system synthesis (Codes-ISSS), pp 288–293, 2006
40. Schliecker S, Ivers M, Ernst R (2006) Memory access patterns for the analysis of MPSoCs, Circuits and systems, 2006 IEEE North-East Workshop on, pp 249–252
41. Münnich A, Färber G (2000) Calculating worst-case execution times of transactions in databases for event-driven, hard real-time embedded systems. In: IDEAS, 2000, pp 149–157
42. Stohr J, von Bulow A, Farber G (2005) Bounding worst-case access times in modern multiprocessor systems, pp 189–198, July 2005
43. Segars S (1998) The ARM9 family-high performance microprocessors for embeddedapplications. Proceedings of the international conference on computer design: VLSI in computers and processors, ICCD'98, pp 230–235, 1998
44. Adiletta M, Rosenbluth M, Bernstein D, Wolrich G, Wilkinson H (2002) The next generation of Intel IXP Network Processors. Network Processors 6(3): 6–18
45. Perathoner S, Wandeler E, Thiele L, Hamann A, Schliecker S, Henia R, Racu R, Ernst R, Harbour MG (2007) Influence of different system abstractions on the performance analysis of distributed real-time systems. In Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT '07). ACM, New York, NY, USA, pp. 193–202
46. Hendriks M, Verhoef M (2006) Timed automata based analysis of embedded system architectures. Workshop on Parallel and Distributed Real-Time Systems, 2006
47. Racu R, Hamann A, Ernst R, Richter K (2007) Automotive software integration. In: Proceedings of the 44th annual conference on design automation. ACM, New York, pp 545–550
48. Künzli S, Hamann A, Ernst R, Thiele L (2007) Combined approach to system level performance analysis of embedded systems. Proceedings of the 5th IEEE/ACM international conference on hardware/software codesign and system synthesis, pp 63–68, 2007
49. Künzli S, Poletti F, Benini L, Thiele L (2006) Combining simulation and formal methods for system-level performance analysis. In: Proceedings of design, automation and test in Europe, 2006
50. Schliecker S, Negrean M, Nicolescu G, Paulin P, Ernst R (2008) Reliable performance analysis of a multicore multithreaded system-on-chip. Proceedings of the 6th international conference on hardware/software codesign and system synthesis (Codes-ISSS), 2008
51. Schliecker S, Hamann A, Racu R, Ernst R (2008) Formal methods for system level performance analysis and optimization. In: Proceedings of the design verification conference (DVCON), San Jose, CA
52. Hamann A, Jersak M, Richter K, Ernst R (2006) A framework for modular analysis and exploration of heterogeneous embedded systems. Real-Time Syst J 33(1-3):101–137
53. Hamann A, Racu R, Ernst R (2006) A formal approach to robustness maximization of complex heterogeneous embedded systems. In: Proceedings of the IEEE/ACM international conference on HW/SW codesign and system synthesis (CODES-ISSS), Seoul, South Korea, October 2006

Chapter 9
Trustworthy Real-Time Systems

Stefan M. Petters, Kevin Elphinstone, and Gernot Heiser

9.1 Introduction

The market of embedded processors far surpasses the market of personal computers and servers. While being more prolific than their desktop counterparts, the progress in semiconductor technology has also brought unprecedented computing power to embedded systems. On the back of these opportunities the complexity of embedded applications is rising dramatically. Two typical examples are today's smartphones or cars. The amount of software contained in these devices is impressive, as for example 100 million lines of code (LOC) in a modern high end car [7] in 2009, while the Android operating system without applications weighs in at around 12 million LOC in 2010.

With the rising complexity there is also increased scope for problems leading to system failure. While it is clear that a degree of functional correctness is required, temporal behaviour must also be considered even in the absence of safety-critical temporal requirements, as the utility of most systems is subject to degradation in the presence of temporal failures. In this paper we consider a *bug* to be any kind of design or implementation error. Aside from bugs, system failures may also be triggered by deliberate attacks or transient errors. Examples of the latter are electromagnetic interference or single-event upsets induced by ionised particles in space.

S.M. Petters (✉)
CISTER/ISEP, Polytechnic Institute of Porto, Portugal,
e-mail: smp@isep.ipp.pt

K. Elphinstone
NICTA and UNSW, Sydney, Australia
e-mail: kevine@cse.unsw.edu.au

Gernot Heiser
NICTA, UNSW and Open Kernel Labs, Sydney, Australia
e-mail: gernot@nicta.com.au

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_9,

Another trend in embedded systems is towards mutable or open systems [14]. This ranges from upgrade facilities available in factory machinery, to tunable cars, to downloadable extensions and games on personal digital assistants or mobile phones. This post-deployment installation of applications not only increases the attack surface for a device, it also implies that the set of active tasks cannot be determined a-priori, which requires a dynamic approach to managing processor time. Furthermore, not the entire task set of a system will have hard real-time character, but some will be of best-effort or soft–real-time character. Such systems are usually referred to as mixed-criticality systems and require that the system components having real-time requirements be protected from those which have a best-effort character.

In summary we perceive the following requirements for modern embedded systems. Reliability and security are the central pillars of the increasing number of complex embedded systems we surround ourselves with. This covers not only the functional aspects, but also the temporal aspects of these systems. While not all requirement lapses will lead to catastrophic consequences, issues like confidentiality of personal and commercially sensitive data, as well as simple usability aspects, motivate the construction of trustworthy systems beyond the traditional safety domain. The lowest software layer or core needs to provide the basic mechanisms for allowing secure, reliable and feature rich embedded systems to be built. While the availability of such mechanisms alone is not sufficient to prevent unreliable systems to be developed, it is a precondition for reliable systems.

The following sections will provide an overview of past research at the ERTOS group at NICTA and the considerations which lead to this research agenda.

9.2 Comparison of Approaches

9.2.1 Real-Time Executives and Desktop Operating Systems

Currently two operating system (OS) paradigms dominate the embedded systems market: classical real-time executives and (more recently) stripped-down versions of desktop operating systems. The major advantage of real-time executives is their size. They fit on small devices and consume minimal resources. A real-time executive can usually be bypassed, as it is rarely used with memory protection enabled, even if that is available on the given platform. This leaves the executive unprotected, and a bug in an application may crash the entire system. Sometimes state is corrupted slowly, and the offending instructions can be very hard to track down. As a consequence, real-time executives are mostly deployed on classical single-purpose devices of low to moderate complexity.

Stripped-down desktop operating systems have been introduced in embedded systems some time ago, but have become more mainstream to handle more complex embedded applications as well as harness the computational power available.

This is driven by the realisation that embedded devices often take on tasks similar to a desktop computer. Examples of such stripped-down operating systems are embedded versions of Windows or Linux (such as Android). A major advantage of such operating systems is that – unlike real-time executives – the kernel is to some degree protected against misbehaving applications.

However, even a minimal configuration of Linux or Windows still features a large code base, hundreds of thousands of lines, and dubious real-time performance. While there are attempts to address real-time behaviour, such as the preempt-RT [30] effort in Linux, these are highly dependent on (for Linux kernel code) unusual coding standards, such as not disabling interrupts, which can be easily violated for example by legacy drivers.

9.2.2 Trusted vs. Trustworthy

While many embedded systems are trusted to some degree to perform according to their specification, it may be questioned whether these systems deserve the trust placed in them. Recalls of cars for *software updates* or required reboots of mobile phones indicate that there are indeed issues which need to be addressed.

Trustworthiness can be established in several ways, collectively referred to as *verification*. Firstly functional correctness can be assured by performing exhaustive testing. While this is possible for trivial examples, the approach fails for more demanding code. Systematic code inspection helps to increase the confidence that the code performs as expected, but as complexity increases, this also fails to provide real assurance, as it is in general not practical to consider all interactions between different parts of a complex system. The logical extension of systematic code inspection is employing *formal verification* methods. Formal verification provides a mathematical proof of correctness, but becomes infeasible beyond a few thousand lines of code.

The poor scalability of thorough verification approaches requires a system that is "small" (of limited conceptual complexity). A small system is not only easier to verify, it is less likely to be faulty (or vulnerable) in the first place. However, smallness is not necessarily required for the whole system, only the "critical" bits. This is encapsulated in the concept of the *trusted computing base* (TCB) [28]. The TCB is the collection of components and mechanisms in hardware, firmware, and software that are critical to enforce the security policy of an entire computer system. A bug in the TCB has the potential to undermine the security or safety of the whole system.

The TCB contains, among others, all software executing in the hardware's privileged mode, this is generally referred to as the *operating system kernel*. In the case of traditional microcontrollers that do not offer memory protection hardware, or when employing a real-time executive which does not utilise memory protection, *all software* is part of the TCB. In such a system of non-trivial size, security or safety become practically impossible to assure.

9.2.3 Minimising and Managing the TCB

To enable any meaningful verification, the size of the TCB needs to be minimised. This requires minimising the OS kernel. The most practical way to minimize kernel size is to restrict it to provide just basic mechanisms for securely managing the hardware, and move all actual system services (including the implementation of application-specific policies) into user-mode code. Such a minimised OS kernel is called a *microkernel* [20].

Some non-kernel components of such a microkernel-based system are still part of the TCB, despite executing without hardware privileges. However, they are now themselves subject to kernel-mediated memory protection and forced to interact via well-defined interfaces. This greatly aids their verification, as they can often verified independently of the rest of the system. The kernel itself can be made small enough that it is within reach of the most rigorous verification approaches. The main prerequisites for using this approach is that the hardware provides memory protection (in the form of a memory-management or memory-protection unit) and dual-mode execution (the distinction between privileged and unprivileged modes).

Figure 9.1 provides a comparison of the three different approaches to system structure. In systems built on top of traditional real-time executive, all code is part of the TCB (indicated by the grey background) and as such are safety/security-critical.

A (stripped-down) desktop OS removes application code from the TCB, as the OS is protected by hardware mechanisms. However, the OS itself is still large (100,000s of lines of code) and infeasible to verify completely. In the microkernel-based system, the TCB consists of the (much smaller) kernel plus whatever essential services the critical system operations depend on, the total can be as small as 10,000 lines of code.

While a microkernel-based OS does not ensure that a system is well-designed, it provides the basic mechanisms for building robust systems, by encapsulating services into individual address spaces [15]. In general, the microkernel provides

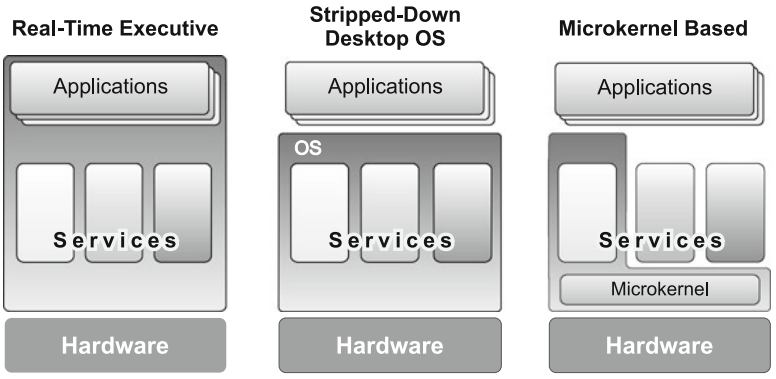


Fig. 9.1 Comparison of approaches

a good base for componentised systems, as envisaged by, for example, the Autosar consortium. Finally it simplifies reasoning about fault isolation properties of the system, and lends support for advanced features such as hot swapping and hot upgrades.

Device drivers are an interesting case in point. Traditionally they are in the kernel and therefore part of the TCB. In a microkernel-based system, they are user-mode components and might be removed from the TCB. There are two reasons why even user-mode drivers may be part of the TCB: Firstly, the correct operation of the system may depend on the correct operation of the device (e.g. a safety-critical actuator). Secondly, if the device is capable of direct memory access (DMA), it could overwrite arbitrary memory, including the microkernel. The latter problem can be avoided if the hardware platform features an *input/output memory management unit* (IOMMU), which makes DMA subject to memory protection just as any other memory access.

9.3 Design and Verification

When setting out to create a trustworthy operating system, a basic question is which design approach to use. Traditionally two different approaches, representative of different mindsets, have been pursued. In a bottom-up approach, depicted in Fig. 9.2, expert programmers design and implement an operating system, which will later be subject to verification by formal-methods experts. Given the expensive nature of formal verification, it is performed as a last step after the system is mature and thoroughly tested to ensure acceptable performance. Since expert kernel programmers are not normally formal-verification experts, this likely leads to a system which in the best case is not very amenable to proving the required properties of the system, and in the worst case does not even expose the properties stipulated in the initial requirements, or be practically intractable to formally reason about due to complexity.

An alternative approach usually favoured by formal methods experts is to start with a formal specification and high-level design to ensure the desired properties will be achieved [16]. The design is then implemented by kernel programmers, this approach is illustrated in Fig. 9.3. While ensuring that the implementation does not impinge on the properties proved for the design is reasonably straight

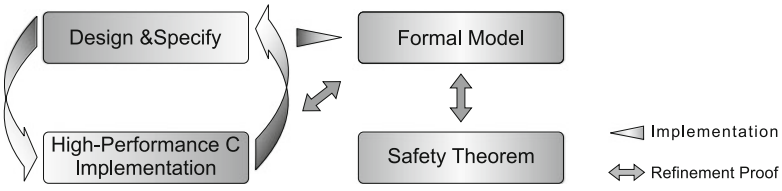


Fig. 9.2 Expert programmer design flow

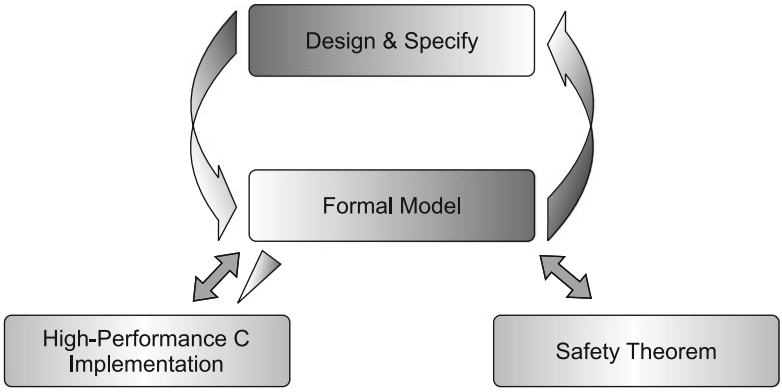


Fig. 9.3 Formal methods centered design flow

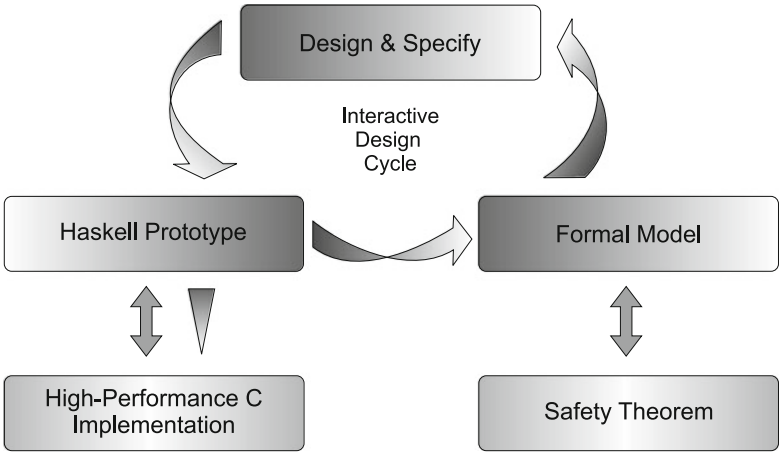


Fig. 9.4 Iterative design approach

forward, the design decisions taken on such high abstraction levels likely result in an operating system which might be correct, but exhibits unacceptable performance, or is impractical for programmers build applications upon.

A third way, employed by NICTA, is to assemble a team of developers with both formal methods and programming experts willing to at least partially learn the respective expertise of their counterparts, and work together to achieve a provably trustworthy high-performance operating system (Fig. 9.4). However, development of expertise alone is not enough, as one must reconcile the different development approaches of both groups. Additionally, the requirement of an iterative approach to the design still exists. This has two implications. On the one hand it requires a quick assessment of the performance impact of design decisions, on the other hand it requires a low barrier to formally reasoning about the design, in order to ensure optimisations aimed at improving performance still maintain the required formal properties.

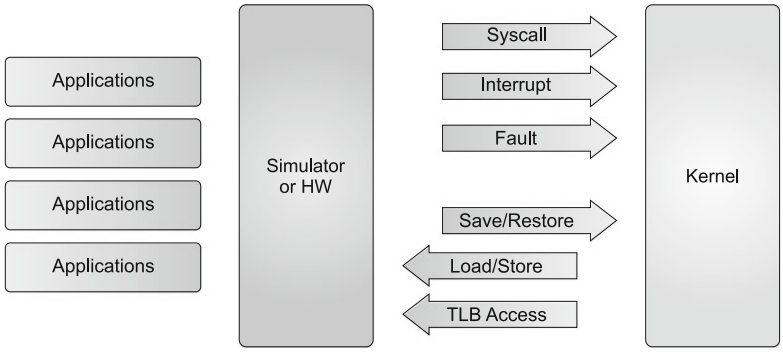


Fig. 9.5 Kernel in the loop [11]

The team at NICTA has approached this problem by choosing Haskell as the specification and design language [13]. Its functional style makes it amenable to automatic translation into the input of an interactive theorem prover. It also forms part of an executable specification which may be used to assess the performance impact of design decisions by requiring a concrete implementation of data structures and algorithms, and enables a much faster turn-around of the design of the API, compared to implementing any specification change in a low-level programming language like C.

With an executable specification, an operating system in-the-loop can be used to assess the utility, usability and general performance of API design decisions without resorting to the time-consuming implementation associated with bare hardware prototypes. Figure 9.5 illustrates the approach taken at NICTA. A set of test applications are be ported to the operating system. The applications themselves execute on a simulator, which is modified such that it only directly simulates user-mode (i.e. application) execution. Transitions to kernel mode (via system calls, interrupts, and other exceptions) result in transfer of control to the Haskell model, which replaces the kernel-mode portion of the simulator itself. The Haskell model manipulates the user-level state such that to applications it appears that an operating system running in kernel mode had serviced the exception.

The approach of running the specification also avoids ambiguities of an plain English specification. Haskell is a functional programming language with well-defined semantics, which enables automatic translation into an input usable by theorem provers. This then provides the starting point for the verification effort. This is eased by the fact that the theorem prover Isabelle/HOL [24] is based on lambda calculus, which also forms the core of Haskell.

Some parts of the kernel, such as the exact scheduling algorithm, are left underspecified in the NICTA work, to allow a later exploration of design choices. Difficulties in the verification process can be addressed, if needed, by changes to the kernel design or API. This flexibility proved to be of fundamental importance in reducing the time to proceed through design iterations.

Despite the common roots of Haskell and Isabelle/HOL, some restrictions were required to facilitate the automatic translation. The executable model avoided the use of lazy evaluation, unlimited recursion, and other problematic Haskell features. Additionally, to support the later reimplementation of the production design in C, the model also avoided use of garbage collection beyond automatic stack variables. The interested reader is referred to relevant publications for further details [17, 18].

9.4 Security and Safety

Security comprises the aspects of confidentiality, integrity and availability; in a particular application scenario some of these tend to be more important than others. Safety is similar, except that confidentiality is not relevant but the (data) integrity aspect is extended to functional correctness.

In many cases of embedded systems, security and safety requirements can be fulfilled using isolation of components. Isolation can be used to prevent a misbehaving application from bringing down another application, or prevent a malicious application from intruding on private data or denying critical actions.

With respect to security and safety requirements, isolation relates to the spatial as well as temporal domains. In the spatial domain it leads to address-space isolation as the basic mechanism. Temporal isolation as it relates to scheduling is discussed in Sect. 9.5. In this section examine temporal isolation in the context of avoidance of denial-of-service attacks through carefully crafted system calls to, for example, cause a very long running kernel operation and thus deny the other applications of the service of the operating system.

Memory management needs to be carefully considered to avoid starvation of this resource at runtime and consequent unbounded or long running system calls. One challenge is that resources such as pages and threads not only require the memory that is part of the user-visible resource, but additional memory for the in-kernel metadata required for management of the resource. Implicit allocation of this memory creates a potential exploit, by forcing the kernel to allocate large amounts of memory for the metadata, which leads to the starvation of memory needed for other tasks. One way to address the starvation issue is to avoid dynamic memory allocation in kernel. In order to achieve this in the seL4 microkernel [12], seL4's design team promoted all metadata to explicitly allocated first-class objects. The metadata is included in the object allocation process and is part of the memory for the object, thus enabling direct control of memory consumption.

A further building block in providing spatial isolation is managing the authority of object creation and authority transfer between applications. The seL4 kernel is implemented as a capability-based system. Capabilities [10] are a tamper-proof representation of the authority to access and/or modify certain data and invoke kernel services. A capability, and thus the corresponding authority, can be passed from application to application allowing concepts like shared memory or task creation to be implemented.

During bootstrapping, all free memory not used by the kernel is handed to the first task as *untyped memory* (UM). Such UM may be subdivided, passed to another application or cast into some other type (e.g. a thread-control block) via a *retype* invocation. The capability, and thus authority to perform these operations, comes with the UM. After invoking the retype method and creating an object of different type, the invokee receives full authority over the object. When it passes the capability to another task it may do so with reduced authority. The key observations to note are that all kernel data structures are objects of a specific type; all typed objects are allocated (via retype) using authority to UM, and UM is a finite resource. Hence applications cannot exceed the memory footprint beyond the authority they possess.

A typical system architecture based on the seL4 microkernel is depicted in Fig. 9.6. The system is comprised of a sensitive application which needs to be protected from unknown or untrusted convenience and legacy functions also running on the same hardware. This might, for example, separate the security-critical crypto-functionality and processing of unencrypted data from wireless networking functionality, or the life-supporting functions in a medical device from the GUI stack. Spatial isolation is assured between all parts of the system. However, trusted services and drivers can be accessed from the untrusted subsystem through well-defined and enforced interfaces. A potential use of a Linux server allows legacy or comfort Linux applications to be deployed alongside critical applications without porting to a new environment. Further information on seL4's security mechanisms can be found in the relevant publications [12, 17].

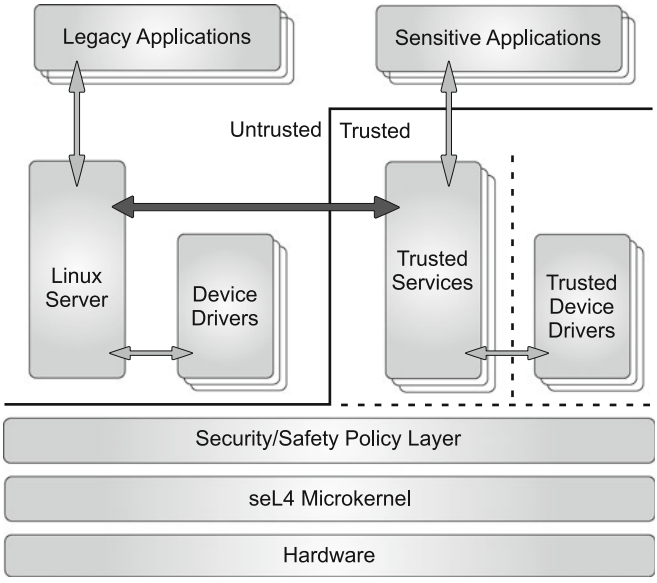


Fig. 9.6 seL4-based system architecture

9.5 Temporal Isolation

The functional isolation and its verification discussed so far is instrumental to building trustworthy embedded systems. This needs to be augmented by temporal isolation, which we now look at in detail in the context of resource scheduling.

Most embedded operating systems in industrial use are either time-driven or fixed-priority–driven. The time-driven approach provides isolation of different applications at the cost of responsiveness. For example, OSEKTime [27] implements a table-driven scheduling approach that allows interrupt delivery only in one slot of the schedule, and otherwise requires interrupts to be disabled. Ultimately this leads to a potential latency of interrupt delivery of an entire scheduling frame. In contrast, fixed-priority schedulers offer the flexibility of fast response time for high-priority applications. However, fixed priority scheduling suffers from poor temporal isolation, as there is no inherent contract about computation time available to an application. Furthermore, relative urgency as well as importance of the application have to be integrated into a single priority value.

For traditional tightly-integrated systems with a known task set, these limitations were acceptable. However, the increasing integration of applications of different and potentially unknown vendors, as well as legacy subsystems, into one device highlights the issue of lack of temporal isolation.

Dependent on the circumstances temporal fault isolation may have different goals. Obviously it should allow critical components to continue to operate correctly in the presence of misbehaving component. It may also ease fault location by avoiding knock-on effects which hide the source of a problem.

Another aspect of temporal isolation is that subsystem may have different scheduling requirements. Best-effort type applications are traditionally scheduled using a fairness-based scheduling scheme, while real-time systems require strict guarantees and the implementation of which are often in juxtaposition to fairness. However, modern embedded systems have often both types of scheduling needs, leading to mixed-criticality systems with conflicting requirements [23].

A number of scheduling approaches have been proposed to address these issues. Examples include sporadic servers [31], deferrable servers [32], or constant-bandwidth servers [1]. In our discussion we will focus on the constant-bandwidth server. Rather than assigning tasks priorities or specific time slots, tasks are assigned a budget of execution time, which will be replenished periodically. This effectively assigns tasks a certain share of the execution time. While the approach guarantees the execution time budget to be available before the next replenishment, it does not stipulate in which order unused budgets may be consumed, allowing a secondary scheduling algorithm to be applied.

Depending on the criticality and the worst-case execution time (WCET) of the task, different sizes of the budget can be chosen. For hard real-time applications, budgets will essentially be dictated by the WCET of a task. For soft real-time applications, budgets might be chosen to be less than the WCET but higher than the average execution time. In this case, a slack-management approach, as described

below, can use leftover budget to fulfill the needs of a task in most cases where the execution time exceeds the assigned budget. Finally, best-effort tasks with a long execution time and aperiodic inter-arrival may be assigned budgets and periods which are a fraction of the needs of an individual job of a task. Such time slicing in combination with slack management ensure fairness and progress among the multiple best-effort tasks.

Research on scheduling performed at NICTA is based on earlier work by Brandt et al. [6, 21]. A fundamental observation underlying their work is that neither a fairness-based approach retrofitted with some real-time mechanisms, nor a real-time scheduling approach with an emulated fairness layer will provide acceptable service for both classes of application needs. Instead an integrated solution offering native fairness and real-time guarantees is required.

In principle, Brandt's proposal has many similarities with other approaches based on bandwidth servers [2]. The distinguishing factors of this work are the detailed consideration of temporal reconfiguration constraints, integration of an albeit simple real-time analysis, and finally, effective dynamic slack management. Brandt's work uses *earliest deadline first (EDF)* [22] as the secondary scheduling algorithm. EDF has been proven to be optimal for a set of independent non-blocking tasks. Besides assuming independent deadlines of all tasks, deadlines were assumed to be equal to the release periods of tasks, which was also used as replenishment period.

In the work of Brandt et al., the traditional scheduling approach is split into a fundamental choice of how much bandwidth is allocated, and a dispatcher to decide when. The former step is taken by the resource allocator, which is active in any mode change of the system to recalculate available shares. The latter is performed by a more traditional scheduler. However, the scheduler enforces the budget allocation and manages unused budget, as well as overruns. Unused budget or slack can be collected and passed to tasks in need of extra budget.

Different aspects of this slack management are discussed by Lin and Brandt [21]. Slack can be passed around in the system under the presumption that slack is scheduled with either the deadline of the task it was associated with, or a more relaxed deadline. This property is required to ensure that the assumptions made in the resource allocator regarding budget use are maintained. A very fundamental corollary is the observation that a task exhausting its budget can be assigned the budget of the next release, if the deadline of the tasks is adjusted such that it reflects the deadline relative to this future release.

The core elements of this approach are depicted in Fig. 9.7. Application arrival or departure in the system triggers a reconfiguration of the budget allocation. During this stage, a real-time analysis is performed, and, after successful completion, the task is provided with a budget and period and is added to the set of schedulable tasks. The EDF scheduler assigns budgets and thus associates applications with enforced timeouts. A slack manager integrated with the scheduler performs the collection and assignment of slack.

We have extended the approach of the resource allocator to perform a demand-bound–based schedulability test [3]. Fundamentally, these tests compute the worst-case request for computation in a given interval. If the worst-case demand does

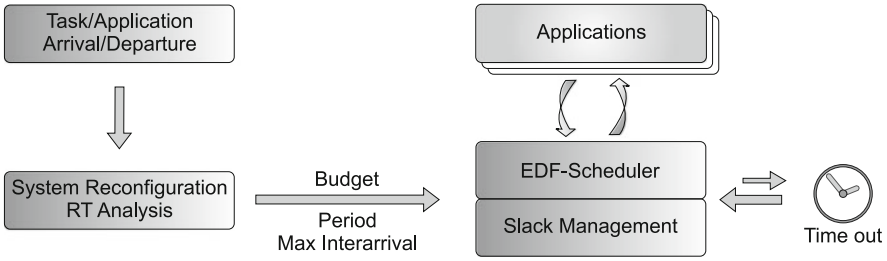


Fig. 9.7 Separation of budget allocation and dispatch

not exceed what can be provided in the same interval, the schedulability test is successful. An expressive event model, which describes the worst-case number of releases of a task – including release jitter and bursty releases – underpins this analysis.

Besides adding the capability to analyse bursty tasks, the resource allocator may also allow for deadlines which are different from the minimum inter-arrival times. This is relevant for rare, but highly important tasks, such as tasks responding to exceptional external input. We have also investigated this approach for critical sections, such as interrupt-service routines and critical sections in user-mode drivers [26]. Further work exploits the explicit knowledge of slack to facilitate practical dynamic voltage and frequency scaling in a real-time environment [19].

9.6 Worst-Case Execution-Time Analysis

The knowledge of the WCET is a fundamental building block for reliable real-time systems. The resource allocator requires such values to perform an overall schedulability analysis. The analysis of the WCET needs to be performed on all parts of the system involved in the delivery of a timely service for soft and hard real-time systems. This obviously covers the real-time applications, but also the kernel and services involved in the process.

The fundamental steps of WCET analysis are depicted in Fig. 9.8. The code for which the analysis is to be performed is firstly analysed for its control-flow structure and operations performed along the nodes of that control-flow graph. In a second step, the constraint and flow analysis aims at restricting the set of possible flows through the program to those which are feasible. This is, for example, necessary to provide bounds on the number of iterations of a loop or to identify nodes in the control-flow graph which may be structurally executable, but are proven to be mutually exclusive due to data dependencies.

In a third step, the execution time of individual nodes in the control-flow graph are established. This may take into account restrictions developed in the flow analysis, as for example, the identified mutual exclusivity of nodes may imply that

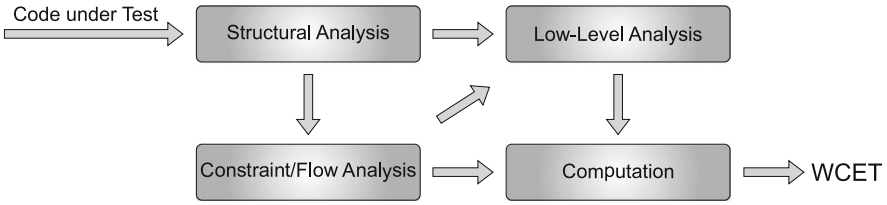


Fig. 9.8 Principal steps of WCET analysis

certain hardware states are not possible when one of the nodes is entered, which may avoid otherwise possible execution paths. Finally, the low-level timing results are combined using the constraints provided by the flow analysis, e.g. to bound the number of iterations for a given loop.

Academic research on WCET is mostly based on static analysis, which lends itself well to the approach shown in Fig. 9.8. In contrast, analysis of safety-critical systems in industry is largely performed using end-to-end measurements augmented with safety margins. The latter approach works well for simple architectures such as those employed by most 8- and 16-bit microcontrollers. It fails for more complex processors, such as the widely-deployed ARM architecture, which features instruction pipelining, branch prediction and multiple levels of cache.

Static WCET analysis on its own has significant limitations too. For one, it is dependent on detailed hardware models, which are not available especially for the more complex architectures. The complexities of modern architectures also mean that the latency of many instructions is highly dependent on internal processor state, which is frequently infeasible to model completely, forcing static analysis to adopt a pessimistic view which leads to gross over-estimation of WCET, often by orders of magnitude.

Fortunately, debugging interfaces available for modern processors can support fine-grained execution tracing and latency measurement. This enables an approach that measures the execution time of small sections of code and combines the results for these to obtain an estimate of the WCET for the overall program using techniques borrowed from static analysis. This approach works under two assumptions: Firstly, the variation in execution time for a small section of code without branches (i.e. a basic block) is small and able to be determined by measurements, even if these are not exhaustive. Secondly, the dominant variation of execution times is caused by the variability of executed paths through the program, which is where static analysis helps [4, 8]. For high-end processors, cache analysis is required to augment the measurements at the basic-block level.

In order to further refine the WCET analysis, it was performed not only on the maximum execution time observed, but also on the distribution and thus providing the frequency of observed execution times. Such execution-time profiles [25] can be interpreted as probability distributions and subsequently used in performing a probabilistic analysis [5].

Compared to application code, WCET analysis of an operating-system kernel presents extra challenges [9, 29], specifically in the structural and flow analysis depicted in Fig. 9.8. Kernel code is highly (manually) optimised, and typically includes parts written in assembly language for efficiency. The combination of optimisation and assembly code means that the complexity of the structural analysis, which is one of the fundamental steps in WCET analysis, becomes extremely difficult to automate. Furthermore, OS code presents the challenge that a single point of entry – an exception-handler address – vectors to many different code paths, such as interrupt-service delivery, exception handling and a variety of system calls. This affects structural as well as flow analysis.

Dealing with the challenges especially of system-call vectors would require a substantial investment into the automatic translation of kernel code into a control-flow graph. However, such a tool would likely be even more complex than a complete hardware simulator, as it would need to consider a very large state space of possible input values (in architected registers). Such a complex tool would be almost impossible to get right, which undermines the point of determining WCET as part of the for safety analysis of a system.

The (for now) preferred alternative is to rely heavily on programmer annotations. While this scales poorly, it is acceptable in the case of a microkernel, owing to its small size, relatively slow evolution (especially in the case of a formally-verified kernel) and the fact that the structural and constraint analysis needs to be done only once. In the case of seL4, there is additional benefit from the many invariants which have been proved in the course of the formal verification. These can partially replace manual annotations, and the existing verification framework makes it relatively easy to add (and prove!) further invariants. In essence, the functional verification provides (very precise) knowledge about properties of the code way beyond what is normally available.

9.7 Conclusions

The rising complexity of embedded systems creates challenges in achieving trustworthiness in security- or safety-critical deployments. We have provided on overview of some of these challenges, and indicated ways of addressing them. Recent work at NICTA is a substantial step towards overcoming these challenges, with contributions in the areas of proofs of functional correctness of a microkernel, isolation in resource management and scheduling, and analysis of worst-case execution times.

While our aim has been to create a trustworthy foundation for embedded software systems in the form of a formally-verified microkernel, this is only the first step towards trustworthy systems. Given a precisely-specified, -behaved, and -analysable operating system, the challenge becomes how to lift the guarantees of the foundation to the level of whole systems. This includes systems composed of software of varying degrees of trustworthiness that rely on the operating system

to protect safety- or security-critical components from malfunctions introduced by other components, which may make up the vast majority of the overall code base. Our on-going research aims at developing frameworks, methodologies, and tools to compose demonstrably trustworthy embedded systems from components of varying criticality and trustworthiness.

Acknowledgements NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. This work was also supported by the Portuguese Fundação para a Ciência e a Tecnologia (CISTER Research Unit – FCT UI 608).

References

1. Abeni L, Buttazzo G (1998) Integrating multimedia applications in hard real-time systems. In: Proceedings of the 19th IEEE real-time systems symposium, IEEE Computer Science Press, Madrid, Spain, pp 4–13
2. Abeni L, Lipari G, Buttazzo G (1999) Constant bandwidth vs. proportional share resource allocation. In: Proceedings of the 5th IEEE international conference on multimedia computing and systems, vol 2. IEEE Computer Science Press, Florence, Italy, pp 107–111
3. Albers K, Slomka F (2004) An event stream driven approximation for the analysis of real-time systems. In: Proceedings of the 16th euromicro conference on real-time systems, IEEE Computer Science Press, Catania, Italy
4. Bernat G, Colin A, Petters SM (2002) WCET analysis of probabilistic hard real–time systems. In: Proceedings of the 24th IEEE real-time systems symposium, Austin, Texas, pp 279–288
5. Bernat G, Newby M, Burns A (2005) Probabilistic timing analysis: An approach using copulas. J Embedded Comput 1(2):179–194
6. Brandt SA, Banachowski S, Lin C, Bisson T (2003) Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In: Proceedings of the 24th IEEE real-time systems symposium, Cancun, Mexico
7. Charette RN (2009) This car runs on code. IEEE Spectrum 46(2), http://www.spectrum.ieee.org/feb09/7649
8. Colin A, Petters SM (2003) Experimental evaluation of code properties for WCET analysis. In: Proceedings of the 24th IEEE international real-time systems symposium, Cancun, Mexico
9. Colin A, Puaut I (2001) Worst case execution time analysis of the RTEMS real-time operating system. In: Proceedings of the 13th euromicro conference on real-time systems, Delft, Netherlands, pp 191–198
10. Dennis JB, Van Horn EC (1966) Programming semantics for multiprogrammed computations. Communications ACM 9:143–155
11. Derrin P, Elphinstone K, Klein G, Cock D, Chakravarty MMT (2006) Running the manual: An approach to high-assurance microkernel development. In: Proceedings of the ACM SIGPLAN haskell workshop, Portland, OR
12. Elkaduwe D, Derrin P, Elphinstone K (2008) Kernel design for isolation and assurance of physical memory. In: 1st workshop on isolation and integration in embedded systems, ACM SIGOPS, Glasgow, UK, pp 35–40
13. Elphinstone K, Klein G, Derrin P, Roscoe T, Heiser G (2007) Towards a practical, verified kernel. In: Proceedings of the 11th workshop on hot topics in operating systems, San Diego, CA, pp 117–122
14. Heiser G (2009) Hypervisors for consumer electronics. In: Proceedings of the 6th IEEE consumer communications and networking conference, Las Vegas, NV, pp 1–5

15. Herder JN, Bos H, Gras B, Homburg P, Tanenbaum AS (2006) MINIX 3: A highly reliable, self-repairing operating system. ACM Operating Syst Rev 40(3):80–89
16. Klein G (2009) Operating system verification – an overview. Sādhanā 34(1):27–69
17. Klein G, Derrin P, Elphinstone K (2009) Experience report: seL4 – formally verifying a high-performance microkernel. In: Proceedings of the 14th international conference on functional programming, ACM, Edinburgh, UK, pp 91–96
18. Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S (2009) seL4: Formal verification of an OS kernel. In: Proceedings of the 22nd ACM symposium on operating systems principles, ACM, Big Sky, MT, pp 207–220
19. Lawitzky MP, Snowdon DC, Petters SM (2008) Integrating real time and power management in a real system. In: Proceedings of the 4th workshop on operating system platforms for embedded real-time applications, Prague, Czech Republic
20. Liedtke J (1995) On μ-kernel construction. In: Proceedings of the 15th ACM symposium on operating systems principles, Copper Mountain, CO, pp 237–250
21. Lin C, Brandt SA (2005) Improving soft real-time performance through better slack management. In: Proceedings of the 26th IEEE real-time systems symposium, Miami, FL
22. Liu C, Layland J (1973) Scheduling algorithms for multiprogramming in a hard real-time environment. J ACM 20:46–61
23. Lin C, Kaldewey T, Povzner A, Brandt SA (2006) Diverse soft real-time processing in an integrated system. In: Proceedings of the 27th IEEE real-time systems symposium, IEEE Computer Science Press, Rio de Janeiro, Brazil
24. Nipkow T, Paulson L, Wenzel M (2002) Isabelle/HOL – A proof assistant for higher-order logic, Lecture notes in computer science, vol 2283. Springer
25. Petters SM (2007) Execution-time profiles. Technical report, NICTA, Sydney, Australia
26. Petters SM, Lawitzky M, Heffernan R, Elphinstone K (2009) Towards real multi-criticality scheduling. In: Proceedings of the 15th IEEE conference on embedded and real-time computing and applications, Beijing, China, pp 155–164
27. Poledna S et al (2000) OSEKTime: a dependable real-time, fault-tolerant operating system and communication layer as an enabling technology for by-wire applications. In: SAE 2000 world congress, Detroit, MI, pp 51–70
28. Rushby J (1984) A trusted computing base for embedded systems. In: Proceedings of 7th DoD/NBS computer security conference, pp 294–311
29. Singal M, Petters SM (2007) Issues in analysing L4 for its WCET. In: Proceedings of the 1st international workshop on microkernels for embedded systems, NICTA, Sydney, Australia
30. Siro A, Emde C, Mc Guire N (2007) Assessment of the realtime preemption patches (rt-preempt) and heir impact on the general purpose performance of the system. In: Proceedings of 9th real-time Linux workshop, Linz, Austria
31. Stanovich M, Baker TP, Wang AI, Harbour MG (2010) Diverse soft real-time processing in an integrated system. In: Proceedings of the 16th IEEE real-time and embedded technology and applications symposium, IEEE Computer Science Press, Stockholm, Sweden
32. Strosnider JK, Lehoczky JP, Sha L (1995) The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. IEEE Trans Comp 44(1):179–194

Chapter 10
Predictably Flexible Real-Time Scheduling

Gerhard Fohler

Historically, real-time systems have been focussed on providing single, specific solutions to single, specific applications, treating all activities with the same methods, geared towards the most demanding scenarios. While the high cost of such an approach is acceptable for applications with dramatic failure consequences it is no longer justified in a growing number of new applications. In these, real-time behavior is demanded only for parts of the systems, few faults can be tolerated. Instead of strict real-time behavior for the entire system, these applications demand "also real-time", or some temporal control.

In this chapter, we analyze two fundamental activation paradigms for scheduling, time and event triggered, which have been considered as having contradicting assumptions, but each providing important attributes, i.e., determinism and flexibility. We will present a scheduling method, which overcomes this traditional all-or-nothing approach to provide a combination of both even for parts of the system. Thus, the system can be designed such that appropriate methods can be used for individual parts of the system. The design no longer has to select either one paradigm or the other, with the respective advantages or disadvantages, but can take the best selection independent of the paradigm.

10.1 Activation Paradigms

The discussion about event triggered vs. time triggered real-time systems has been lively and going on for a while. This text does not wish to contribute to it, rather present and analyze the concepts with respect to scheduling, as needed for presentation. Examples of the discussion can be found in [3, 15, 19].

G. Fohler (✉)
University of Kaiserslautern, Germany
e-mail: fohler@eit.uni-kl.de

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_10,

10.1.1 Scheduling

This section introduces the two paradigms and discusses some scheduling relevant properties and differences. One of the central choices in real-time systems design is that of the *activation paradigm*, i.e., when are events recognized, who initiates activities, when are these decisions taken? In conventional systems, the *event triggered* approach is prevalent, in which occurrences of events initiate activities in the system immediately. In *time triggered* systems, activities are initiated at predefined points in time [15].

Time Triggered

We start with the time triggered approach, contrasting some properties already here to those of event triggered. Initiating activities in the system with the progression of time requires thorough, complete understanding of the system and the environment it will operate in. Scheduling for TT is usually carried out via a scheduling table, which lists tasks and their activation times. An offline algorithm[1] takes complete information about the system activities, which reflect the knowledge about anticipated environmental situations and requirements, and creates a single table, representing a feasible solution to the given requirements. As the algorithm is performed offline, fairly complex task sets can be handled, e.g., precedence constraints, distribution and communication over networks, task allocation, mutual exclusion, separation of tasks, etc. – e.g. [1, 5, 7, 9, 14, 18, 20] Should a feasible solution not be found, retries are possible, e.g., by changing the parameterization of the algorithm or the properties of the task set. At runtime, a very simple runtime dispatcher executes the decisions represented in the table, i.e., which (portion of a) task to execute next. Typically, a minimum granularity of time is assumed for the invocations of the runtime scheduler, so called *slots* [16].

As a consequence, a number of *advantages* can be achieved from a scheduling perspective:

- *Determinism*: Given the schedule represented in the table and a point in time, it can be determined which task will execute in that slot. Note the difference to *predictability*, which allows to predict properties of task sets, e.g., whether they will meet their deadlines, but cannot determine exactly when tasks will execute.
- *Constructive schedulability test*: The scheduling table provides a "proof by construction" that all timing constraints will be met. In contrast to a proof that no timing constraints could be violated in any situation, an offline approach

[1] A number of terms have been used to describe scheduling methods which construct scheduling tables offline, such as initially *static, pre-runtime, offline* scheduling. The term table driven appears the most general.

only needs to show that one situation exists, i.e., the scheduling table, in which the timing constraints are met: instead of a "for all" proof, considering even situations which may never occur during the runtime of a system, a "there exists one" suffices. While, e.g., an explicit schedulability test needs to consider the worst case blocking time of tasks involved in mutual exclusion, even when they might never enter their critical sections at runtime together, offline scheduling, can simply separate the critical sections in the scheduling table. Consequently, pessimism in the schedulability test is reduced.

- *Complex constraints*: As scheduling is done before the system is deployed, sufficient time for solving complex constraints is available, such as precedence, distribution, etc. Still, at runtime, only the very simple table lookup suffices to meet all constraints. Furthermore, most algorithms are based on search, which provides for the simple inclusion of new constraints. In the event triggered case, explicit schedulability tests need to be developed anew for additional constraints, if that is possible at all.
- *Runtime overhead*: The very simple runtime scheduling, i.e., table lookup, incurs very little overhead. Explicit tests can require significant overheads [21].
- *Non temporal benefits* such as simple fault tolerance, receiver based error detection have been discussed in detail e.g., in [15].

On the *downside*, table driven scheduling reveals

- *Inflexibility*: Anything not completely known before runtime cannot be handled by a pure offline approach.
- *Pessimism*: Many parameters, such as execution times, arrival times, have to be based on worst case assumptions. As a consequence, a periodic world has to be assumed. Pure offline approaches are unable to reclaim resources if worst cases planned are not needed.
- *Design effort*: The cost and effort of obtaining complete knowledge may be considered too high for non critical applications.

Event Triggered

In event triggered systems, events invoke an online scheduler, which takes a decision based on a set of pre defined rules, e.g., represented as priorities. An offline schedulability test can be used to show that, if a set of rules is applied to a given task set at runtime, all tasks will meet their deadlines. Major representative lines of such algorithms are based on fixed priorities, e.g, rate monotonic or dynamic priorities [17].

The advantages and disadvantages appear to almost mirror the ones of TT Sect. 10.1.2:

- *Flexibility*: As decisions are taking at runtime, not completely known activities can be added easily.
- *Widely used*: Many commercial operating systems are based on priorities as well.

- *Simple constraints*: As decisions are taken at runtime, only simple constraints can be addressed, e.g., mutual exclusion. The necessary algorithms can be difficult [21] or costly [2] to implement. Changes in requirements will necessitate new algorithms and associated tests, which may not be available.
- *Limited predictability*: As opposed to the determinism of TT, ET schedulers can predict certain properties of task sets, e.g., whether deadlines will be met, but not determine exactly which activities will executing at specific points in time.

10.1.2 Levels of Determinism

We will now introduce levels of determinism to analyze the activation paradigms and their scheduling.

In general, computers cannot provide absolute determinism, as there is inherent uncertainty, e.g., due to different clock speeds, or physical properties of chips. We will look at various levels of computation, meaningful temporal granularities, and discuss the notion of temporal determinism for each, for use in comparison between TT and ET scheduling next in Sect. 10.1.3.

Task Level

It is generally not possible to predict which instructions of a task are being processed during the execution of a task. This information would also not be very helpful, as we are not interested in timing constraints on such a low level. The meaningful granularity commonly assumed is the worst case execution time (wcet) of the entire task.

Thus, task level determinism can state that when a task is started at time t, and we check again at time $t + wcet$, the task will have finished. Variations of finer granularity, e.g., branches, etc., leading to variations of execution times can be safely ignored from the task level point of view.

Slot Level

As described above, many TT systems assume slots as basic granularity for runtime scheduling decisions, with the scheduling table assigning tasks to these slots. We

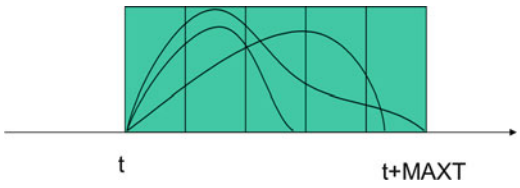


Fig. 10.1 Task level determinism

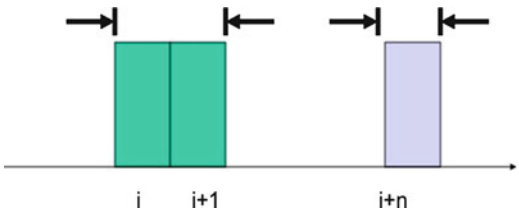


Fig. 10.2 Slot level determinism

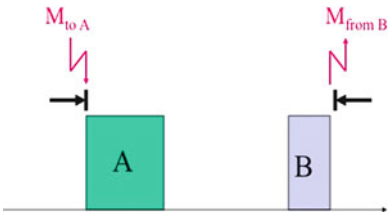


Fig. 10.3 Node level determinism

cannot say whether some instructions of a task will be processed in a slot as actual execution times will vary.

The determinism on the slot level is that if all tasks are executed at times as specified in the scheduling table (we look only at the slots) all temporal constraints will be met. Given a point in time and the scheduling table it can be determined which task has been assigned a slot.

Node Level

The relevant points at the node level are internode message transmissions; from the outside it is not relevant which tasks are executing when – provided constraints are met.

The determinism at the node level can seen as when an ingoing internode message is received by the node, sufficient processing time according to the schedule is waited, the resulting outgoing internode message will be sent in time with the desired contents.

System Level

Relevant points on the system level are events in the environment and actions performed. At this level, details about computations within the systems are not relevant, as long as the constraints between events and actions in the environment are met.

10.1.3 TT and ET Scheduling Fundamentally Different?

Given the different assumptions, approaches, and properties, TT and ET scheduling could be perceived as very different, or even opposing paradigms. We will take a closer look.

The ET *real-time scheduling process* takes sets of tasks with timing constraints and performs a test if these constraints can be met if a given algorithm is used at runtime. The algorithm may take properties of tasks, notably priority or deadline, as input, or determine them as directives, artifacts for the online scheduling algorithm.[2]

Thus, the process follows the steps: task set with timing constraints – schedulability test and determination of rules (e.g., via directives priority or deadline) – execution of rules by runtime scheduler – timing constraints met.

Looking closer, we can see that TT real-time scheduling works in the same way. Instead of a definition of rules, e.g., "earliest deadline first", the decisions on which task to execute are represented in the scheduling table, "schedule next task as given table".

While TT scheduling has to assume a *periodic world* and ET provides *flexibility* for tasks with not fully known parameters, e.g., aperiodic, the difference concerns mostly runtime execution without guarantees. When offline guarantees are required, task parameters have to be known offline: without worst case execution time, period or maximum arrival frequency, offline guarantees cannot be given, independent of the scheduling paradigm used.

Hence, we can conclude that the terms "offline" and "online" scheduling cannot be seen as completely disjoint in general. Real-time scheduling demands offline guarantees, which require assumptions about online behavior at design time. At runtime, both offline and online execute according to some (explicitly or implicitly) defined rules, which guarantee feasibility. Thus, both offline and online are based on a substantial offline part. The question is then where to set the tradeoff between determinism - all decisions offline - and flexibility - some decisions online .

Let us now look at how table driven and rule driven scheduling compare with respect to the levels of determinism. We include *best effort* methods, as applied in general purpose operating systems, as rule driven scheduling. Only statistical predictions can be made for large data sets, not about individual tasks.

Figure 10.4 shows the expected result that best effort provides high flexibility, with only statistical predictions; online real-time scheduling can provide some flexibility, guarantee deadlines and node level determinism; table driven scheduling with fixed time, i.e., the classic time triggered scheduling model, has no flexibility, but slot level determinism. In the figure, the crossing point of the flexibility and determinism lines lies within table driven scheduling, but not with fixed times - it represents table driven scheduling, with some flexibility on the actual slot scheduling. We will discuss this algorithm in the remainder of the chapter, and how it can provide *predictable flexibility*. Section 10.4 presents an example algorithm.

[2]Then, the task properties, "priority" are separated from the importance of a task, "deadline" from the timing constraint. Rather, they both serve only to direct the online scheduling algorithm to execute the proper rules for schedulability.

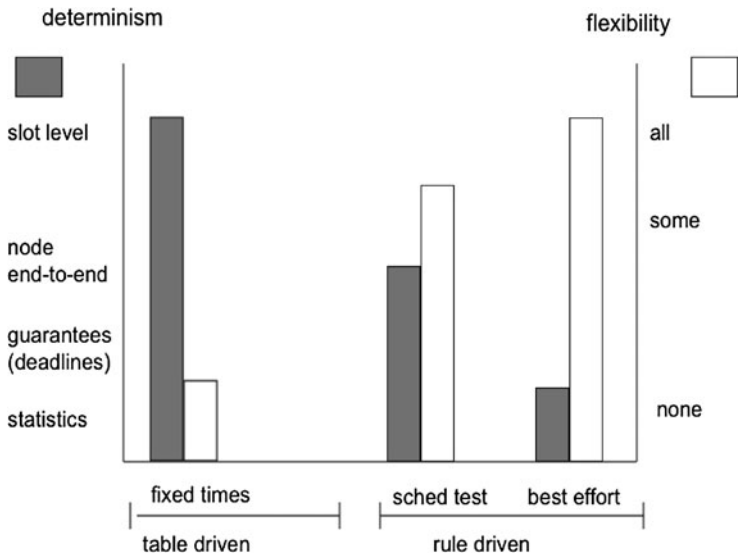


Fig. 10.4 Determinism vs. flexibility

10.1.4 Mixed Criticality Applications

Considering the different properties of ET and TT activation paradigms, as described in Sect. 10.1, the selection of one becomes a central design decision. It is considered an "either - or" decision, forcing designers to choose the advantages of one method at the expense of the other's. Demands outside the selected paradigm need to be "squeezed in", e.g., to suit periodic world requirements. The choice has also system wide implications, as the same properties, in particular cost, apply to all activities in the domain of the paradigm. Often, the highest level, e.g., of determinism has to be chosen, affecting even activities with no such demands.

This was acceptable for the simplicity and uniformity of historic monolithic approaches: single systems were executing single applications, a single paradigm sufficed for a single class of demands.

In modern systems, such uniformity cannot longer be maintained. Rather, systems exhibit a mix of activities and demands, *mixed criticalities*:

- *Core* system, which is essential for system survival with high demands for strict temporal behavior, has to be safety critical and fault tolerant. These properties have to be guaranteed and tested for the worst case. Possibly, certification is required as well.
- *Hard real-time applications* demand guarantees of temporal correctness, but are not part of the core system.
- *Flexible real-time applications* can tolerate the miss of some deadlines, exact behavior is not completely known, or too costly too obtain.

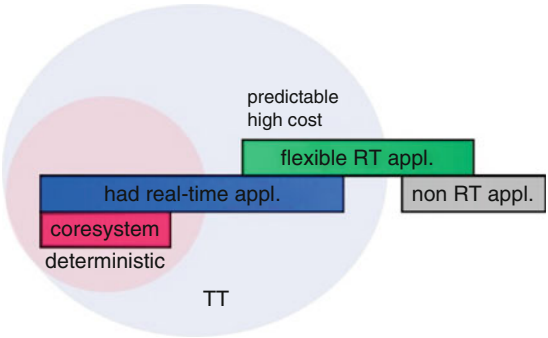


Fig. 10.5 Mixed criticality applications – TT

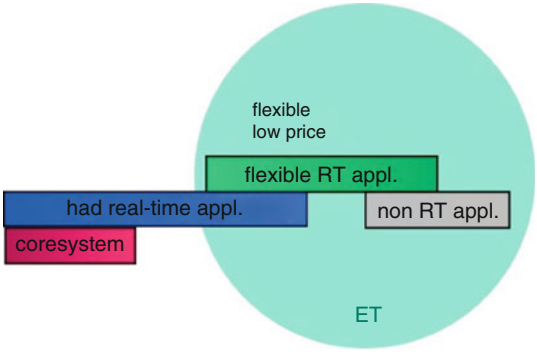


Fig. 10.6 Mixed criticality applications – ET

- *Non real-time activities* do not have temporal demands themselves, but must not disturb the real-time activities, while their behavior cannot be known.

Let's have a look at how pure TT or ET scheduling handle such mixed criticality systems.

Pure TT (Fig. 10.5) can provide determinism for the core system and hard real-time applications.

Some flexible real-time applications can be handled, but at high cost, e.g., due to transformation to fit the periodic model. Non real-time applications cannot be handled at all. Thus, TT scheduling incurs high cost even for non critical applications.

Pure ET (Fig. 10.6) deals well with flexible and non real-time applications, and can handle some hard real-time applications well. Core system requirements cannot be met. Thus, ET scheduling cannot provide for deterministic behavior of critical activities.

In summary, neither of the pure scheduling paradigms can handle the mixed criticality demands of modern system well; a combined approach is needed, which will be the topic of the rest of the chapter.

10.2 Combined TT – ET Scheduling Approach

The combined approach discussed here is based on table driven scheduling. Details of the actual algorithm have been presented in e.g. [6,13]. Here, we give the rationale and show how predictable flexibility can be achieved. A description is given in Sect. 10.4.

10.2.1 Table Driven Scheduling Revisited

As described in Sect. 10.1.2, table driven scheduling can handle general, complex constraints, as an offline scheduler is used. The resulting scheduling table describes an execution sequence of tasks, such that the constraints are met, i.e., a proof by construction. While a number of different schedules and tables could meet the constraints, a single one has to be created by the offline scheduler, due to the table lookup nature of the runtime dispatcher. Thus, slot level determinism is achieved at the expense of flexibility.

Let us look at the example in Fig. 10.7. The system consists of two nodes, N_0 and N_1, which are connected via a network NW. The input task set is a precedence graph consisting of four tasks, A, B, C, D. A, B are allocated to node N_0, C, D to N_1, a message is sent between B, C over the network. The precedence graph has a deadline $dl(PG)$, bounding the time between start of A and completion D. The resulting schedule meets these constraints.

We can see that the offline scheduler decided to leave some time between the execution of B and the transmission of the message over the network. The constraints would be met as well if B, or A as well, would be executed later. Thus, to achieve slot level determinism, the execution of A, B is fixed, although other

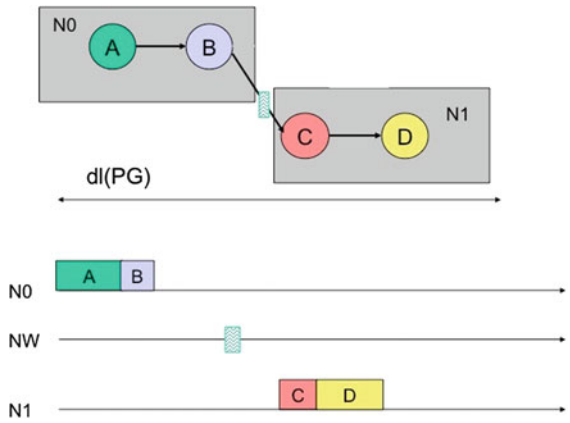


Fig. 10.7 Example precedence graph and schedule

times for their execution would be feasible as well. In fact, the transmission of the message, as well as of C, D could be shifted to later points in time, while still meeting the constraints.

Node level determinism can be achieved on N_0, as long as B finishes before the message is sent, on N_1, if C does not start before the message is received, and D completes before $dl(PG)$. Thus, the flexibility of the schedule can increase significantly, while maintaining node level determinism.

10.2.2 Target Windows

Given timing constrains and scheduling table, we can perform flexibility analysis. The result are limits on the execution of tasks, called *target windows* with earliest start times and deadlines: if tasks execute within their target windows, all constraints will be met. While meeting the original timing constraints in a distributed system is NP hard, tasks now only have to execute locally, independently, within their target windows to meet the constraints, which can be done with linear complexity, e.g., by earliest deadline first of fixed priority scheduling. Figure 10.8 shows the target windows for the previous example.

10.3 Predictable Flexibility

Target windows can now be used to control the flexibility of task executions:

- Target windows after the above described flexibility analysis provide flexibility while maintaining the original timing constraints and node level determinism.
- Setting target windows to the original task executions maintains the original scheduling table, no flexibility, but slot level determinism
- Reducing target windows for individual tasks reduces their flexibility, e.g., for jitter control.

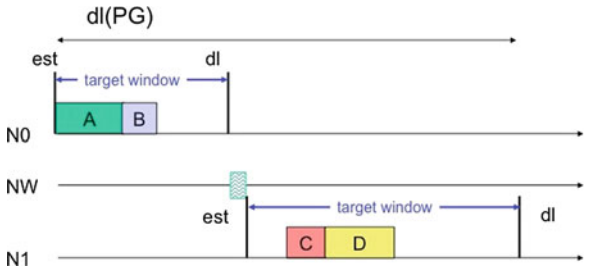


Fig. 10.8 Example target windows

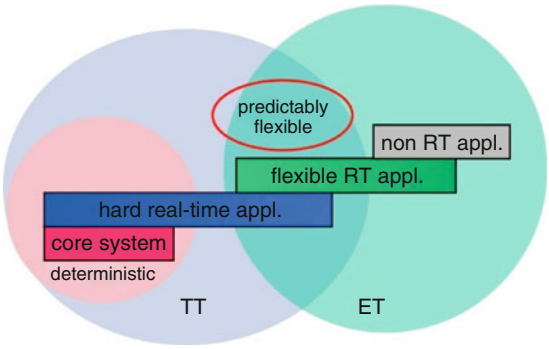


Fig. 10.9 Mixed criticality applications with predictable flexibility

Thus, target windows allow for setting flexibility for tasks *individually*, providing predictable flexibility. Slot level determinism as in the scheduling table can be set for tasks deemed worth the cost, while others can execute flexibly, and all timing constraints are met.

10.3.1 Mixed Criticality Applications

Predictable flexibility can be used to schedule mixed criticality applications described in 10.1.4 as follows:

- Core system: target windows set as originally in the scheduling table, providing slot level determinism
- Hard real-time applications: either table driven with flexible target windows or via ET scheduling on top
- Flexible real-time applications: via the combined offline/online approach
- Non real-time applications: together with the combined approach, which ensures they do not interfere with feasibility of the guaranteed tasks.

Figure 10.9 illustrates the change compared to pure TT or ET as described in Sect. 10.1.4.

10.3.2 Scheduling Overview

Figure 10.10 illustrates the steps involved for predictable flexibility.

First, we have the task set with original temporal constraints. Then, the offline scheduler creates a scheduling table. This step is not tied to a specific offline scheduler. Next, flexibility analysis as briefly described in Sect. 10.2 is performed to obtain the target windows.

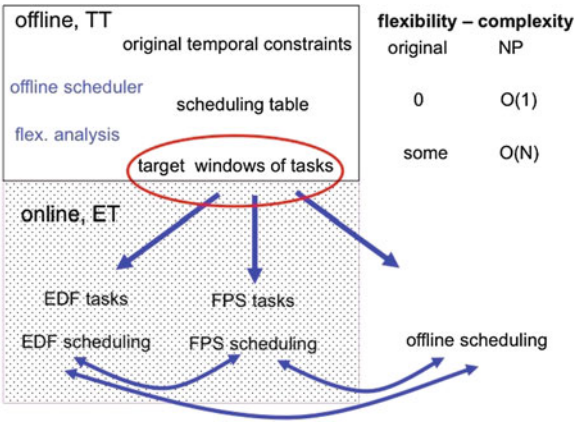


Fig. 10.10 Scheduling overview for predictable flexibility

The target windows express the original timing constraints via earliest start times and deadlines. As long as the offline scheduled tasks execute within their target windows, other tasks can be included using ET methods and the original constraints will be met. Thus, complexity is reduced significantly, compared to the original scheduling problem. The reduction comes at the price of loosing optimality as the scheduling table which forms the basis for flexibility analysis describes only a single solution out of possibly many other feasible ones (10.1).

Task execution within target windows can be guaranteed by a variety of algorithms: EDF based, as in slot shifting in Sect. 10.4, but also others, such as FPS. For completeness, offline scheduling can be applied as well. Then a different schedule from the original scheduling table will be executed, which can be used, e.g., for incremental scheduling, when tasks should be added with only minimum changes to an existing schedule.

Note that these scheduling steps can also be used to execute a schedule maintaining original timing constraints with a different scheduling method than originally anticipated. This provides more choice for designers, but also facilitates porting of applications to different platforms and scheduling algorithms.

10.4 Slot Shifting

In this section, we briefly describe the slot shifting method as example for the combined TT ET approach. It provides for the efficient handling and on-line guarantee of aperiodic tasks on top of a distributed offline schedule with general task constraints. Slot shifting extracts information about unused resources and leeway in an offline schedule and uses this information to add tasks feasibly, i.e., without

violating requirements on the already scheduled tasks. A detailed description can be found in [6].

10.4.1 Off-Line Preparations

First, an off-line scheduler, e.g., [5] creates scheduling tables for the periodic tasks. It allocates tasks to nodes and resolves precedence constraints by ordering task executions.

Start-Times and Deadlines
The scheduling tables list fixed start- and end times of task executions, that are less flexible than possible. The only assignments fixed by specification, however, are first and last tasks in the precedence graph, and, as we assume message transmission times to be fixed here,[3] tasks sending or receiving inter-node messages. These are the only fixed start-times and deadlines, all others are calculated recursively, as the execution of all other tasks may vary within the precedence order, i.e., they can be shifted.

Intervals and Spare Capacities
The deadlines of tasks are then sorted for each node and the schedule is divided into a set of *disjoint execution intervals* for each node. Spare capacities are defined for these intervals.

Each deadline calculated for a task defines the end of an interval I_i, $end(I_i)$. Several tasks with the same deadline constitute one interval.

The spare capacities of an interval I_i are calculated as given in formula (10.1):

$$sc(I_i) \;=\; |I_i| - \sum_{T \in I_i} wcet(T) + min(sc(I_{i+1}), 0) \tag{10.1}$$

The length of I_i minus the sum of the activities assigned to it is the amount of idle times in that interval. These have to be decreased by the amount "lent" to subsequent intervals: Tasks may execute in intervals prior to the one they are assigned to. Then they "borrow" spare capacity from the "earlier" interval.

10.4.2 On-Line Mechanisms

During system operation, the on-line scheduler is invoked after each slot. It checks whether aperiodic tasks have arrived, performs the guarantee algorithm, and selects a task for execution. This decision is then used to update the spare capacities. Finally the scheduling decision is executed in the next slot.

[3] We apply the same mechanisms to the network as well, i.e., shifting messages, as detailed in [6].

Guarantee Algorithm

Assume that an aperiodic task T_A is tested for guarantee. We identify three parts of the total spare capacities available:

- $sc(I_c)_t$, the remaining spare capacity of the current interval,
- $\sum sc(I_i),\ c < i \leq l,\ end(I_l) \leq dl(T_A) \wedge end(I_{l+1}) > dl(T_A),\ sc(I_i) > 0$, the positive spare capacities of all *full* intervals between t and $dl(T_A)$, and
- $min(sc(I_{l+1}), dl(T_A) - start(I_{l+1}))$, the spare capacity of the last interval, or the execution need of T_A before its deadline in this interval, whichever is smaller.

If the sum of all three is larger or equal to than $wcet(T_A)$, T_A can be accommodated, and therefore guaranteed. Upon guarantee of a task, the spare capacities are updated to reflect the decrease in available resources. This guarantee algorithm is $O(N)$, N being the number of intervals. It is shown in [5], that this acceptance test has equivalent results – but with simpler run-time handling – as to the ones presented in [4, 8], which are optimal for single processors.

On-Line Scheduling
On-line scheduling is performed locally on each node. If the spare capacities of the current interval $sc(I_c) > 0$, EDF is applied on the set of ready tasks. $sc(I_c) = 0$ indicates that a guaranteed task has to be executed or else a deadline violation in the task set will occur. Soft aperiodic tasks, i.e., without deadline, can be executed immediately if $sc(I_c) > 0$. After each scheduling decision, the spare capacities of the affected intervals are updated.

10.5 Conclusion

In this chapter we have introduced predictable flexibility for real-time system scheduling, which aims at combining time triggered and event triggered scheduling. Instead of forcing designers to take a choice early on which effects system design, predictable flexibility provides for selecting flexibility for individual activities in the system. Thus it is suited for mixed criticality applications, which involve a range of diverse requirements for different parts of the system.

We analyzed both TT and ET from a scheduling perspective, and with respect to the level of determinism vs. flexibility they can provide. We argued for a combined approach, which increases flexibility significantly over pure TT scheduling, while still providing a level of determinism, albeit less strict than pure TT. As example for such a combined approach, we reviewed slot shifting.

Acknowledgements The author wishes to thank the members of the research groups at TU Vienna, Austria, University of Massachusetts at Amherst, USA, MDH, Sweden, and TU Kaiserslautern, Germany for their valuable contributions and discussions of the line of research. Special thanks go to Damir Isovic for advancing the state of slot shifting and the many inspiration discussions.

References

1. Abdelzaher TF, Shin KG (1999) Combined task and message scheduling in distributed real-time systems. IEEE Trans Parallel Distributed Syst
2. Björn B. Brandenburg, John M. Calandrino, Aaron Block, Hennadiy Leontyev, James H. Anderson (2008) Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In: IEEE real-time and embedded technology and applications symposium, pp 342–353
3. Burns A (2004) Programming real-time systems. In: ECRTS 04–16th euromicro conference on real-time systems, Catania, Sicily
4. Chetto M, Chetto H (1989) Scheduling periodic and sporadic tasks in a real-time system. Inf Proc Lett
5. Fohler G (1994) Flexibility in statically scheduled hard real-time systems. PhD thesis, Technische Universität Wien, Austria, April 1994
6. Fohler G (1995) Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In: Proceedings of the 16th real-time systems ymposium, Pisa, Italy
7. Fohler G, Ramamritham K (1997) Static scheduling of pipelined periodic tasks in distributed real-time systems. In: Proceedings of the 8th euromicro workshop on real-time systems, June 1997
8. Garey MR, Johnson DS, Simons BB, Tarjan RE (1981) Scheduling unit-time tasks with arbitrary release times and deadlines. IEEE Trans Soft Eng
9. Hou C-J, Shin KG (1992) Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems. In: IEEE Proceedings of the 13th IEEE real-time systems symposium, pp 146–155, December 1992
10. Isovic D, Fohler G (1998) Handling sporadic tasks in off-line scheduled distributed hard real-time systems. In: Proceedings of the 10th euromicro onference on real-time systems, York, UK, June 1998
11. Isovic D, Fohler G (1999) Handling sporadic tasks in statically scheduled distributed real-time systems. In: Proceedings of the 10th euromicro conference on real-time systems, June 1999
12. Isovic D, Fohler G (2000) Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In: Proceedings of the 21st IEEE real-time systems symposium, Walt Disney World, Orlando, FL, November 2000
13. Isovic D, Fohler G (2009) Handling mixed sets of tasks in combined offline and online scheduled real-time systems. Real-Time Syst J 43(3)
14. Jonsson J, Shin KG (1997) A parametrized branch-and-bound strategy for scheduling precedence-constrained tasks on a multiprocessor system. In: ICPP, pp 158–165
15. Kopetz H (2011) Real-time systems – design principles for distributed embedded applications. Springer, Berlin
16. Kopetz H, Fohler G, Grünsteidl G, Kantz H, Pospischil G, Puschner P, Reisinger J, Schlatterbeck R, Schütz W, Vrchoticky A, Zainlinger R (1993) Real-time system development: The programming model of mars. In: Proceedings of the international symposium on autonomous decentralized systems, Kawasaki, Japan, March/April 1993
17. Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in hard real-time environment. J ACM 20:1
18. Ramamritham K, Stankovic JA, Shiah P (1990) Efficient scheduling algorithms for real-time multiprocessor systems. IEEE Trans Parallel Distributed Syst 2(1)
19. Verissimo P (2002) Fundamental questions in the et vs. tt debate? please look elsewhere. In: Next-TTA workshop on ET-TT integration, Grenoble, France, October 2002
20. Šůcha P, Hanzálek Z (2004) Scheduling with start time realated deadlines. In: IEEE international symposium on computer aided control systems design
21. Yodaiken V (1998) Rough notes on priority inheritance. Technical report, New Mexico Institut of Mining, 1998

Part III
Innovative Application Domains

Chapter 11
Detailed Visual Recognition of Road Scenes for Guiding Autonomous Vehicles

Ernst D. Dickmanns

11.1 Introduction

Almost a quarter of a century has passed by since road vehicle guidance at typical speeds by machine vision has been demonstrated for the first time [1]. Due to the high complexity of the vision task and to missing affordable mobile computing power, application of visual perception in assistance systems for road vehicles has been confined to single tasks like recognition of lanes or of some types of traffic signs; for almost all premium cars these systems are offered as options on the market today. Recently, combinations of radar and vision for (hybrid or advanced) Adaptive Cruise Control (ACC) have been introduced in the market. A good survey on the efforts to develop intelligent road vehicles may be obtained from the proceedings of the yearly *International Symposium 'On Intelligent Vehicles'* [2].

In the early 1990s, when the basic capability of convoy driving by vision had already been demonstrated in research vehicles [3], industry decided to use radar for market introduction due to longer experience available, all-weather capability, and lower overall costs. Missing direct road recognition with radar has been compensated (partially) by tracking other vehicles on the same road further away. Laser range finders (LRF) have also been tested in great varieties, e.g. [4–6]. Relative to radar LRF have the advantage of much better lateral resolution, however, like vision it does not guarantee all-weather capabilities. A review of these types of systems for perception of the environment may be found in [7]; laser systems are limited in range and can be rather costly. One such system is the Velodyne LRF that has been extensively used in the DARPA Urban Challenge in November 2007; remember that in this contest the vehicles did not have to perceive lane markings, traffic signs or negative obstacles but only "positive" obstacles sticking out of the driveway, which was given by GPS-waypoints [8]. For this support scenario in a

E.D. Dickmanns (✉)
UniBw Munich/LRT/TAS, D-85577 Neubiberg
e-mail: edd@dyna-vision.de

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_11,

well known environment dense laser ranging is well suited; the knowledge base for signal interpretation can be kept small compared to that needed for a powerful vision system with good capabilities in object discrimination.

However, machine vision on a level similar to the human visual perception system has quite a few advantages:

1. In multi-focal vision, lateral resolution can be an order of magnitude higher than for affordable lasers (a fraction of a milliradian (∼0.2 mrad) compared to 1.6 mrad horizontally, resp. 4.4 mrad vertically in the Velodyne system).
2. Different spectral ranges can be evaluated for easy understanding of complex scenarios (color); in each color channel good intensity resolution is available.
3. Passive vision exploits light intensities naturally available, or it uses light also needed for human vision (conventional head lights).
4. View fixation allows reducing motion blur for fast moving objects; even a set of three cameras is small nowadays, needing not much power for gaze control.
5. Inertial feedback of data from a perturbed platform (vehicle) allows good gaze stabilization under harsh environmental conditions (hardly affordable for Velodyne-type systems).
6. Detection of candidates for negative obstacles (like a ditch in a grass surface) can be achieved much earlier by intensity patterns than with stereo vision or laser range finders due to self-occlusion of the ditch at larger distances and to noise effects stemming from single herbs and straws in front of the ditch.

All these items indicate that technical vision is the way to go if systems with performance levels coming closer to the human one are looked for. Computing power needed is tremendous, but since the early 1980s a factor of about one million (10^6) in performance of microprocessors of same size and power consumption but lower costs has occurred. In the near future, multiple processors on a single chip (like used in Graphic Processing Units (GPUs) or similar) will allow an increase in computing power that can be run in automotive environments by several orders of magnitude. So it seems to be the right time now to develop high-performance technical vision systems for understanding dynamically changing traffic scenes.

The type of vision system needed has been investigated in [9, 10]; a multi-focal "vehicle eye" seems unavoidable if human-like performance levels are the goal. The technical eye should have

(a) A large simultaneous field of view nearby ($>\sim 110°$).
(b) The capability of stereo vision (a few degrees laterally, range $<\sim 10$ m).
(c) The capability of color vision in medium ranges ($<\sim 100$ m).
(d) High resolution (~ 5 cm per pixel) in a small field of view (covering ~ 30 m laterally at distances of ~ 300 m); this means 5 mm resolution at 30 m distance which is sufficient for reading traffic signs.
(e) At least points (b) and (d) require active gaze control.

To achieve good real-time performance, recursive estimation with feedback of prediction errors including spatio-temporal models for each object has shown to

be a very good approach [10, 11] almost universally accepted by now. It is the merit of G. Faerber to have initiated a DFG-"Transregio" on "Kognitive Automobile" in the first half of this decade, that has the goal of developing this approach further with additional knowledge and learning components. The software for "Expectation-based, Multi-focal, Saccadic (EMS-) vision" developed at UniBwM [12, 13] has been transferred to his Lab. at TUM and has become the starting point for the common system architecture in project "Kognitive Automobile" [14].

In the present contribution, the basic idea underlying the 4-D approach is developed one step further in concept: The visual interpretation processes monitor their outcome and initiate adaptation not only of model parameters and states, but they also can trigger switching of models and selecting other sets of features by them selves. This requires a store of models as knowledge background and criteria for assessing the actual quality of performance. The latter ones, of course, are the histories of prediction errors in the perception – action loop running for each object tracked. As long as the prediction errors are below a threshold for a single object, the model applied and the measurement data are in resonance, and the object as well as its motion is considered to be "understood". If only occasional discrepancies occur, trust in results of perception is reduced but the underlying model need not be changed immediately; feature extraction can be adjusted in this case.

If prediction errors increase in magnitude and/or in frequency and all feature sets available have been tried, a new hypothesis (model) for the object has to be tried. The history of this process is stored for later analysis and learning. The difficult and challenging part is the selection of a new model based on experience accumulated previously. The proposal here is to improve using combined sets of features for making intelligent choices of models and their parameters, taking the overall situation into account. Without the situational context, myriads of hypotheses could be checked; it is the actual knowledge about the situation given that allows choosing "reasonable" hypotheses for interpreting image sequences.

11.2 The Perception: Action Loop in Traffic Situations

Domain knowledge allows reducing the sets of objects likely to be encountered in certain scenes. These objects have characteristic features, usually, that allow their recognition when occurring in certain image regions under standard mapping conditions. Gravity pulls all objects towards the road surface; road vehicles in standard poses touch the ground with two to ten wheels that will be mapped as parts of approximate rectangles respectively ellipses. Vehicles with more than two wheels have an extended, almost planar area parallel to the ground at a relatively small elevation above the ground that obscures the region underneath the vehicle, both in sunshine and under overcast conditions; therefore, a dark spot below an assembly of various other features is a good indicator for a vehicle ahead. Unfortunately, a truck with a large cylindrical tank of round or elliptical cross-section above the axles,

especially in combination with the sun standing low (in the morning or evening), defies these characteristics; this case has to be checked separately depending on time of day and weather conditions.

The road surface is the main area of reference when driving, especially when other elements of standard roads can be recognized like lane markings, curbstones, almost homogeneous road shoulders or guide rails; reflection poles with regular spacing help recognizing road curvature, especially at low lighting conditions or after snowfall. The road surface appears as a collection of shaded gray patches. Due to changing aspect conditions with range and due to vertical curvature of the road cross-section, even roads with homogeneously gray appearance under orthonormal view will appear shaded in the image under other viewing conditions; the same is true for other objects. Therefore, image areas with approximately linear gray shading are chosen as easy to extract features for image understanding.

But it is not a collection of features of one type that "induces" hypotheses of certain objects; it is the combination of several features in certain arrangements and moving in conjunction that trigger hypotheses for objects typical in the situation given. Like in human perception, object hypotheses may be inferred from a typical (sparse) collection of features before the full feature set is applied for hypothesis verification. In road traffic, the presence of another vehicle may be inferred from the dark spot underneath and a collection of edge- and corner features above it; the lowest dark-to-bright edge in this collection allows estimating range to the object under the assumption of a planar ground in front of the own vehicle.

For distinguishing between a static obstacle and a vehicle, recognition of wheels is an important step [15]; here one has to distinguish between several cases: Does the vehicle appear under an oblique view (from left or right), or from straight behind or ahead. From straight behind or ahead, wheels appear as a single pair of dark rectangles at the outer sides underneath the vehicle body; in this case, features may tend to be symmetrically distributed [16, 17]. Looked at from the side, symmetry disappears and parts of two to three wheels (or up to six for a vehicle with more than two axles) may be visible as sections of ellipses with eccentricity depending on aspect angles. Whether a vehicle is moving or standing still has to be inferred from two or more images of a sequence by measuring relative feature positions of the vehicle and of some static features on or at the side of the road nearby (dashed lane markings or a pole, a building, or a tree at the side). The length of a vehicle can only be estimated under sufficiently oblique views; it has turned out that estimation of the length of a horizontal diagonal is more stable than that of the orthogonal values "width and length" under these conditions [18]. From right behind or in front, only vehicle width can be estimated, from the side it is only vehicle length that can be determined. These cases have to be treated separately, and their results have to be fused in memory. Once estimation of vehicle size parameters has settled to stable values, the state space for recursive estimation (including vehicle parameters) may be reduced, and the corresponding features should then be used for improving range estimation [19].

Lane markings carry information directly affecting behavior control: Solid lane markings on one side indicate that leaving the lane in this direction is not allowed under normal conditions; only dashed lane markings may be crossed. In case of a solid and a dashed lane marking side by side, crossing is only allowed from the side of the dashed markings. The width of the lane markings on multi-lane roads also carries information: Standard widths (depending on the country) separate two lanes while especially wide lane markings separate the driving area from the road shoulder (wide parking stripe or small safety zone). These wide markings are of special interest at entries and exits, and in regions where the number of lanes changes in between. In these regions, also diagonally hashed areas may be seen that mark parts of the road surfaces not available as lanes for standard driving.

Near crossings or entries/exits also arrows may be painted in the lanes indicating into which direction the lane will be leading; sometimes two options will be available from one lane containing arrows with two tips (straight ahead and turn-off). At crossings, usually, the start of the intersection area is marked by a white line orthogonal to the own lane markings; this line is only to be crossed when the vehicle can leave the intersection area immediately. Otherwise it has to wait in front of this line as in the case of a red or yellow traffic light.

Observation of all kinds of traffic signs (to the side of and above the road) and remembering the actually valid regulation state is another important task for a mature vision system [20]. Due to potential occlusion by other vehicles, the own stile of driving and of attention control is affected, especially in multi-lane traffic.

At a speed of $\sim$100 km/h a vehicle moves about 1 m per video cycle (from frame to frame); with a visual range of about 30 m for good resolution in standard images, a stationary object has appeared more than two dozen times in the video sequence before the vehicle comes close to it. Feature flow in the image increases with decreasing range, usually. Objects, all features of which move to one side in the image sequence will be bypassed at the opposite side. Most dangerous are objects, the feature flow of which goes to all sides of the image center when gaze direction is right onto the object (dubbed "looming"); this will lead to a crash with the camera if nothing is changed. The best reaction for evasion depends on the relation of egomotion to object motion (for another vehicle).

Table 11.1 summarizes the essential items making up a situation when driving on high-speed roads with unidirectional traffic; on other roads with oncoming traffic, many of the items to be observed remain, but look-ahead distance has to be increased since relative speed in neighboring lanes may increase even though speed in each direction is limited, usually.

Most of the items listed in Table 11.1 can best be perceived by looking for specific sets of features; beside edges, especially linearly shaded blobs and corners or textured areas are of interest. Radar or laser range finding provides no or only partial information of these features with less lateral resolution; multi-focal color vision can provide many more features for object distinction if computing power allows.

Table 11.1 Items and properties describing a traffic situation

Far left	Left neighboring lane	Own lane	Right neighboring lane	Far right
Lane or shoulder, para-meters; turn-off lane? Traffic signs? guide rails? buildings or trees nearby? shadows	Available/not -; lane width, lane continues/ends	Traffic signs above? width and curvature parameters	Available/not -; lane width, lane continues/ends	Lane or shoulder, para-meters; turn-off lane? Traffic signs? guide rails? buildings or trees nearby? shadows
	Second vehicle ahead: range, range rate, bearing, motion state, acceleration	Second vehicle ahead: range, range rate, bearing, motion state, acceleration	Second vehicle ahead: range, range rate, bearing, motion state, acceleration	
	Vehicle ahead: range, range rate, bearing, lane running, -change, long. acceleration brake lights on?	Vehicle ahead: range, range rate, bearing, lane running/-change, long. acceleration brake lights on?	Vehicle ahead: range, range rate, bearing, lane running, -change, long. acceleration brake lights on?	
	Occupied or free? type of vehicle? relative state? actual behavior? road surface?	Own vehicle **(self)** speed, behavioral mode running, own intentions; road surface state, lighting & weather	Occupied or free? type of vehicle? relative state? actual behavior? road surface?	
	Range & bearing, high range rate? actual behavior?	Vehicles behind**:** range, range rate, bearing, lane running/-change	Range & bearing, high range rate? actual behavior?	

11.3 Expectation-Based, Multi-focal, Saccadic Vision

What is the best type of vision system for the tasks mentioned in the previous section? Because of the relatively small increase in information from frame to frame due to the high image frequency (25 or 33 1/3 Hz) relative to speed driven, temporal integration of this information with the help of spatio-temporal models and recursive estimation with prediction error feedback [11] has proven to be both efficient and reliable. For a thorough evaluation including sensor fusion the reader is referred to [10]; saccadic vision is detailed in [21–23]. Here only an integration aspect with respect to the use of knowledge is added which may lead to improved overall systems capable of learning.

Similar to a development just happening in the understanding of biological systems [24], the 4-D approach and EMS-vision may be looked at as a method trying to avoid quasi-static states and – knowledge representations. Measurement data (including visual features) are used to bring the observation loop into resonance with spatio-temporal models in the perception process by adapting parameters in these models for a bunch of objects in the scene, including 3-D space, ego-motion, and changing perspective projection. This is done by the "subject" (the vehicle itself including sensors, data processors, background knowledge and actuators for control of gaze and ego-motion) taking part in the overall process of mutual multi-agent traffic control; the subject does have knowledge about own perceptual and behavioral capabilities and the resulting "maneuvers" occurring in space over time when applied [10]. These maneuvers constitute important knowledge elements in dynamic scenes; it is tacitly assumed that the other agents on the road all do have similar capabilities. Therefore, their behavior may be expected to some degree when typical maneuver initiations are observed.

This allows attention control by shifting high-resolution imaging to regions of special interest; saccades allow gaze shifts of tens of degrees within a fraction of a second. During the saccades, all images may be blurred so that feature extraction does not make sense, and the internal representation of the situation is derived from the spatio-temporal models, as is routine in recursive estimation anyway.

When all tracking loops for the selected number of objects constituting the situation show small prediction errors for their features, the scene is considered to be understood. Newly appearing features or large prediction errors request special attention: Either temporally limited noise effects or new objects entering the scene may be the reason. This calls for more thorough feature analysis or for a gaze shift leading to images with higher resolution in this region. Usually, in a dynamic environment there will always be some regions and objects needing adaptation. If this is only a small fraction of the overall scene, mission performance can be continued; if the uncertain parts increase, maybe mission performance has to be slowed down or even interrupted, and the vehicle has to stop in a safe manner to reorient itself. These topics of visual system integration will not be detailed here; in the rest of the chapter attention is focused on the extraction of sufficiently rich sets of features for solving these tasks.

11.4 Image Feature Extraction

In the development of machine vision, analysis had concentrated on snapshots (single images) without bothering about delay times until results became available. In real-time vision for active control of dynamic scenes (like road traffic), delay times can be of paramount importance. Fortunately, in parallel to the experience accumulated in the field of feature extraction, computing power of microprocessors has increased to a level that unified feature extraction even from multi-focal images in parallel becomes possible now.

A 25 Hz image evaluation rate (40 ms cycle time) is considered sufficient for road vehicle guidance up to very high speeds; at 180 km/h (50 m/s) images are thus taken every 2 m. In conventional (interleaved) video streams this means using single video-fields (odd or even) only. With a multi-focal look-ahead range of ∼200 m, 100 images can be evaluated until the vehicle reaches the location of first detection of an object at the range limit. Assuming a pixel resolution of 0.1 mrad for the large focal length of the "vehicle eye" means that one pixel maps 2 cm normal to gaze direction. With 1,000 pixels in an image row, the area mapped in the image cone of 5.7° is 20 m wide, sufficient for a bidirectional (two times) two-lane highway. Even with a 2:1 reduction in resolution for smoothing there still would be three pixels on a standard lane marking of width 12 cm.

Here, a new approach for unified feature extraction is presented that exploits the smoothing 1–2–1-averaging (needed for the Sobel-gradient operator) also for corner detection at every pixel. Keep in mind that with the carefully selected focal lengths and the high image frequency (25 Hz), scale aspects for corner detection and tracking play a minor role in a single image. High resolution and small time delays are important here; the new method serves this purpose.

In a tri-focal system with 4:1 separation of focal lengths the (divergent two) wide-angle cameras would have a field of view of about 60–90° each. A divergence angle of about one quarter of this value for the wide-angle cameras would allow a simultaneous field of view of around 110° and a stereo region of about 10–60°, of which only the central part may be analyzed. At 10 m range, pixel resolution of the wide-angle cameras is ∼1.6 cm; this looks like a good compromise for coming close to human visual capabilities. Keep in mind that with the tri-focal vehicle eye all images are evaluated in parallel and should be interpreted in conjunction; so, for traffic sign reading with color vision, the camera with medium focal length (resolution ∼0.4 mrad/pixel) has a resolution of ∼1 cm per pixel at 25 m range. Sign reading is performed with fixation-type vision [10, 21].

11.4.1 The Feature Set Extracted in a Unified Approach

As justified in [10], the following set is extracted: In addition to edges with adjacent average gray values, more precise regional gray-shadings (linearly shaded blobs

alleviating object recognition) and corners allowing 2-D tracking are of special importance. In the long run, color vision should be included. Before linear shading in one direction is applied, the region is tested whether planar shading is applicable at all. For this purpose, initially, a four point test with neighboring pixels or in adjacent rectangles with averaged intensity values has been performed [25, 10, Sect. 5.3] that turned out to be extremely simple and efficient; three sums and one compare to the threshold for planarity immediately lead to all four residues of the least-squares planar fit in the region covered. It is interesting to note that the Haar-wavelet I_{xy} used in [30] exactly represents this four-point planarity test. If the threshold is exceeded, planarity no more is a good approximation for the local intensity distribution, and the candidate region for linear shading has to be closed. Linearly shaded 1-D regions in search direction thus are found between two such locations with nonplanar intensity distribution.

On the other hand, corners characterized by stronger curvature of image intensity in two orthogonal directions can only occur in image regions satisfying the nonplanarity conditions. In typical traffic scenes, nonplanarity for a threshold value of 1% of the total intensity range (256 in standard video) occurred in less than 20% of image locations tested with the 4-point planarity test; lifting the threshold to 2% (typical for humans to notice differences without paying special attention, see [10, Fig. 5.24]) reduces these image regions to less than 5%. That means that the number of image locations where one should reasonably look for corners can be reduced to 1/5 resp. 1/20 by this simple planarity test. By saving intermediate values during gradient computation with the Sobel-operator, this 4-point planarity test costs only one subtract- and compare operation, and it reduces the average workload for corner feature extraction by about an order of magnitude. However, this simple nonplanarity test also picks up an edge when its direction sufficiently differs from the mask direction. These false candidates have to be removed in further steps confined to the small number of image locations found.

With respect to scale effects in corner detection, three routes have been considered:

1. The diagonal sums needed for the simple four-point planarity test directly invite to forming the next (2×2) pyramid level by addition. To have the same center for additional planarity tests on the next pyramid level, a 3×3 mask has to be chosen; this allows two such tests, the "diagonal" and the "cross" test with four pixels on the directions indicated by the name (see Fig. 11.1a). These tests together cover an area of 6×6 original pixels. However, depending on the four starting points possible for pyramid computation, *different pixel values* on the next higher pyramid level will result in image sequences with slightly different gaze angles from frame to frame as is quite natural from cameras onboard a vehicle. Experience with systematic variations of the starting point in the same image has shown that the number of corner candidates for a certain parameter set may vary up to 20%; this has led to discarding this approach.
2. Since the results with 3×3 masks for detecting and affirming nonplanarity have been promising, it was decided to start with these masks directly on the

original pixel level and to compute results for 1-step shifts in row and column direction. The central pixel is left out in the 4-point tests, but the difference between the measured intensity of the central pixel and the average of the four pixels used for the planarity test ("diagonal" or "cross") yields another, independent, nonplanarity test; it was dubbed "central deviation"-feature ε_{c4} and is used for forming binary or ternary (1-bit or 2-bit) nonplanarity-images [26] (see below).

Instead of looking at gradients and deriving expressions for curvature (as second-order derivatives of the 2-D intensity function) from these, as is usually done in a family of corner detectors (e.g. [27,28] and derivatives), here, curvature is detected directly from the definition as deviation from a linear or planar reference. Figure 11.1b shows the interpolation of three points by a second-order parabola; a higher curvature C_{1D} of the interpolated function is given only when the central point I_0 lies sufficiently off the straight line a_m between the two equidistant neighboring points I_{m1} and I_{p1}. By straightforward interpolation of the intensity function $I(x) = I_0 + a_m \cdot x + C_{1D}/2 \cdot x^2$ through the three points I_0, I_{m1} and I_{p1} the curvature parameter C_{1D} is obtained as

$$C_{1D} = (I_{m1} + I_{p1} - 2 \cdot I_0)/d^2. \tag{11.1}$$

For the cross mask, the grid points can be chosen as the same smoothed elements used for the Sobel gradient-operator as shown in Fig. 11.1c, both in row- (*left*) and in column direction (*right*); it is interesting to note that the sum of both one-dimensional curvature operators (without the division by d^2) yields eight times the "central deviation" feature (difference between the average of the corner

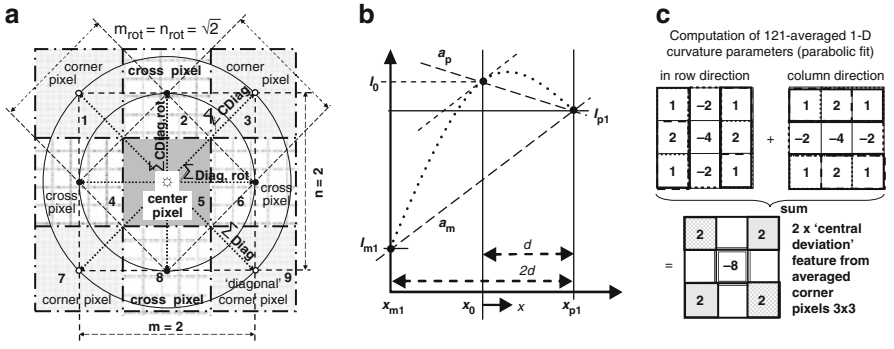


Fig. 11.1 (**a**) Test mask 3 × 3 with two pixel patterns ("diagonal" and "cross") for planarity tests. (**b**) Second order parabolic interpolation through three equidistant points (measured or smoothed) in any cross-section yields slope a_m of the linear least squares fit equal to the tangent direction of the interpolating parabola with curvature parameter C. (**c**) Use of Sobel-elements for computing the curvature parameters (see Fig. 11.2); the sum yields (eight times) the "central deviation" feature as initial planarity test

pixels of the 3 × 3 mask and the measured central pixel value, see Fig. 11.1c, *bottom*). So this central deviation feature is a good candidate for the initial nonplanarity test in the 3 × 3 mask. However, this "central deviation" feature as (4-point plus 1) planarity test leaves out the actual intensity values in the pixels left empty in Fig. 11.1c, *bottom*. If only the four elements of the Sobel-operator in the boundary regions of the 3 × 3 mask are summed, they yield 16 times the *average intensity* I_{MeanB3} *in this boundary region*. The difference to the measured central pixel yields another measure for a "boundary-central-deviation" nonplanarity test.

Random tests with road scene images have shown that this (8-point) test very closely yields about the same number of candidates for intensity-corners than the test with the average of the four corners of the 3 × 3 mask. The high evaluation frequency of 25 Hz under steadily changing conditions has led to discarding the differences without detrimental consequences. Confining generation of candidates for corner features to the small 3 × 3 region reduces workload for candidate merger later on; by precise localization of the center of gravity (*c.g.*) of candidates in a local region, the final corner position is determined. The measure for fusing candidates into a *c.g.* is the sum of the squared residues of the planar approximation.

In [26] the false alarms stemming from non-aligned edges have been reduced by 4-point planarity tests with rotated sets of masks in 5×5 or 7×7 regions. However, it has turned out to be more efficient to directly test bidirectional curvature with the scheme given above for only a few rotated masks by using orthonormal diagonals in a 5-point scheme (next section).

Coming back to the point of scale effects in corner detection, in our multi-focal approach this may be covered by performing the unified approach to feature detection on all multi-focal images in parallel and by crosschecking the results; active attention control in real-time vision then has to resolve uncertainties remaining over time. With a factor of 3–4 in spacing of focal lengths, a 3×3 mask on the next higher level (with less resolution) covers a square of 9–12 pixel side length on the lower level. Therefore, 7×7 masks on each level seem sufficiently large for approximately equal spacing in scales with our multi-focal approach.

3. A completely independent approach for dealing with scale effects has been pursued in [29]; computing an integrated image first and using the "SURF"-approach [30] for detecting interest points, their detection and tracking at any larger scale can be achieved efficiently with a fixed number of operations. Exploiting the parallel computing capabilities of modern Graphic Processing Units (GPUs), real-time performance up to 200 Hz has been demonstrated with a single image sequence. The approach discussed in the next section combined with this approach just cited may yield a good couple satisfying all requirements for multi-focal real-time vision.

11.4.2 Five-Point Planarity Tests

As in the 4-point test disregarding the central pixel, deviations from the best local planar fit are taken as reference for curvature here too. In general, in a 3 × 3 mask the best local planar approximation to the intensity function in the grid would have to use all nine pixels in the mask for interpolation. However, this yields relatively complex results needing many computational steps. Since curvature of the intensity function around the center of the mask will be checked in two orthogonal directions, five points (four corners and one center) are picked here for a least squares planar fit; in the cross mask in Fig. 11.1a the three intensity values of the cross-sections through the center are used for interpolating a second order parabolic function exactly through these values as shown in Fig. 11.1b. Additional test directions in wider regions (5 × 5 or 7 × 7) may be obtained as Fig. 11.2 shows; *on the left* (a), two sets of masks are formed from pixel pairs in a 5 × 5 field, while *on the right* (b) they are chosen from a 7 × 7 field.

The 5-point test tends to pick corners by the convex side, while the 4-point test without the central pixel in the 3 × 3 mask favors the concave side. Independent of the number of pixels in the square masks for this test, only the corner-pixels of the rotated mask and the central pixel are taken for determining a planar approximation based on these five samples. Since the initial nonplanarity test is based on a 3 × 3 local environment, the following tests only have to remove false alarms from edges; for this purpose it is of advantage to choose (averaged) pixel values further away from the center of the mask to reduce noise effects from digitization.

Choosing orthogonal coordinate frames in direction of the diagonal and the counter-diagonal of the mask applied, the formulation of the task is straightforward. The planar model for five discrete points on the diagonals at distance d from (respectively at) the origin is written

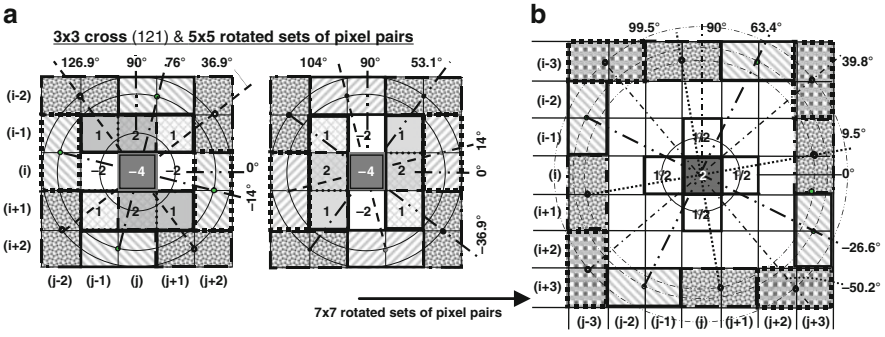


Fig. 11.2 Rotated mask sets from pixel pairs in 5 × 5 (*left*) and 7 × 7 local fields (*right*). In subfigure (**a**) the initial curvature test with Sobel-elements is included, while in (**b**) the central pixel smoothed in both directions with the same Sobel element is shown

$$\begin{aligned} I_{11M5D} &= I_0 - a_D \cdot d + 0 \\ I_{12M5CD} &= I_0 + 0 - a_{CD} \cdot d \\ I_{cM5} &= I_0 + 0 + 0 \\ I_{21M5CD} &= I_0 + 0 + a_{CD} \cdot d \\ I_{22M5D} &= I_0 + a_D \cdot d + 0. \end{aligned} \tag{11.2}$$

Let the measured values (index μ) from the image be $I_{11\mu}$, $I_{12\mu}$, $I_{c\mu}$, $I_{21\mu}$ and $I_{22\mu}$. Then the errors e_{ij} can be written:

$$\begin{bmatrix} e_{11} \\ e_{12} \\ e_{c5} \\ e_{21} \\ e_{22} \end{bmatrix} \begin{matrix} = I_{11M5D} - I_{11\mu} \\ = I_{12M5CD} - I_{12\mu} \\ = I_{cM5} - I_{c\mu} \\ = I_{21M5CD} - I_{21\mu} \\ = I_{22M5D} - I_{22\mu} \end{matrix} = \begin{bmatrix} 1 & -d & 0 \\ 1 & 0 & -d \\ 1 & 0 & 0 \\ 1 & 0 & +d \\ 1 & +d & 0 \end{bmatrix} \begin{bmatrix} I_0 \\ a_D \\ a_{CD} \end{bmatrix} - \begin{bmatrix} I_{11\mu} \\ I_{12\mu} \\ I_{c\mu} \\ I_{21\mu} \\ I_{22\mu} \end{bmatrix}$$

$$\text{or} \quad e = A \cdot p - I_\mu. \tag{11.3}$$

To minimize the sum of the squared errors, $e^T e$ shall be minimized by proper selection of $p^{\mathrm{T}} = [I_0\ a_{\mathrm{D}}\ a_{\mathrm{CD}}]$. The necessary condition for an extreme value is that the partial derivatives vanish; this leads to the solution via pseudo-inverse

$$p = (A^T A)^{-1} \cdot A^T \cdot I_\mu. \tag{11.4}$$

In a few steps the following results are obtained:

$$\begin{aligned} a_{D5} &= (I_{22\mu} - I_{11\mu})/(2d); \quad a_{CD5} = (I_{21\mu} - I_{12\mu})/(2d); \\ I_0 &= I_{Mean5} = (I_{11\mu} + I_{12\mu} + I_{c\mu} + I_{21\mu} + I_{22\mu})/5 = 0.8 \cdot I_{Mean4} + 0.2 \cdot I_{c\mu}. \end{aligned} \tag{11.5}$$

The two residues on the diagonal and the counter-diagonal are each equal and are dubbed ε_{D5} respectively ε_{CD5} here:

$$\begin{aligned} \varepsilon_{D5} &= 0.2 \cdot (I_{12\mu} + I_{21\mu} + I_{c\mu}) - 0.3 \cdot (I_{11\mu} + I_{22\mu}) = I_{\mathrm{Mean5}} - (I_{12\mu} + I_{21\mu})/2, \\ \varepsilon_{CD5} &= 0.2 \cdot (I_{11\mu} + I_{22\mu} + I_{c\mu}) - 0.3 \cdot (I_{12\mu} + I_{21\mu}) = I_{\mathrm{Mean5}} - (I_{11\mu} + I_{22\mu})/2. \end{aligned} \tag{11.6}$$

The residue at the center of the mask is

$$\varepsilon_{c5} = I_{\mathrm{Mean5}} - I_{c\mu} = 0.2 \cdot I_{\mathrm{Mean4}} - 0.8 \cdot I_{c\mu} = 0.8 \cdot \varepsilon_{c4}. \tag{11.7}$$

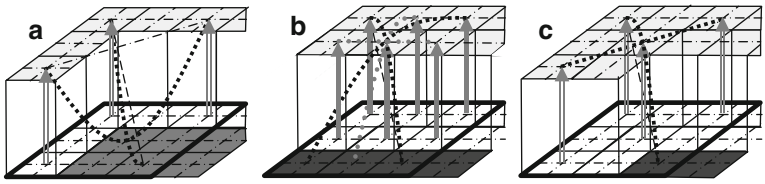


Fig. 11.3 Curvature test for eliminating false alarms from edges by parabolic second order data fit on the diagonals. (**a**) *left*: Central pixel of 3 × 3 mask on an aligned corner: ratio of curvature coefficients = 0.5; (**b**) *center*: aligned edge: strong curvatures, ratio = 1. The rotated test (45°) in cross direction shows one curvature to be zero (gray curve with round dots); (**c**) *right*: The 3 × 3 mask just picks up the dark corner with 1 pixel: one curvature is zero

Two things are interesting to note: (1) that the central residue ε_{c5} of the 5-point planar fit is 80% of the central deviation ε_{c4} in the 4-point fit, and (2) that the difference between the two residue values ε_{D5} and ε_{CD5} on the diagonals of the 5-point test (11.6) is always twice the residue value $|\varepsilon_4|$ of the 4-point fit, i.e. the difference between diagonal and counter-diagonal residues are the same

$$|\varepsilon_{D5} - \varepsilon_{CD5}| = |(I_{11\mu} + I_{22\mu}) - (I_{12\mu} + I_{21\mu})|/2 = 2 \cdot |\varepsilon_4|. \tag{11.8}$$

From (11.1), the curvature coefficients C for the diagonal and the counter-diagonal of the 5-point test are

$$C_{D5} = (I_{11} + I_{22} - 2 \cdot I_{c\mu})/d^2; \quad C_{CD5} = (I_{12} + I_{21} - 2 \cdot I_{c\mu})/d^2. \tag{11.9}$$

For the ratio of both, the denominators cancel; the first sums of intensities are the same as for the 4-point test, so that the computational load for determining the curvature parameters is small. Figure 11.3 visualizes the results for three cases with an ideal corner in alignment with the test mask used.

In case (a) *at left* the central pixel of the 3 × 3 mask covers the ideally hit dark corner, while in case (c) *at right* the central pixel is on the level of the bright convex side of the ideal corner, and the candidate is rejected. In the first case, both curvatures are large (squared ratio $C_{R2} = 0.25$); this is accepted as a corner feature. If the 3 × 3 mask covers the image corner with just one pixel (*right*), one curvature parameter is zero; the position is not accepted as a corner. If one row or column in the 3 × 3 mask is at a different (but constant) intensity level than the rest of the mask (also at a constant level, see (b) *center*), the 5-point test yields a curvature ratio of 1, but this is due to the straight edge. Therefore, both the 4-point test (which eliminates this case as candidate) and the 5-point test have to be passed; an alternative are more 5-point tests with rotated masks that have to be passed e.g. in cross direction (see curves with gray round dots in (3*b*) or with sets from Fig. 11.2).

To re-use the elements of the Sobel-gradient operator, in a first step the 121-smoothed pixels in the 3 × 3 cross mask are taken as shown in Figs. 11.1 and 11.2. This quite naturally induces the idea "Why not start with one of these curvature tests

right from the beginning?" The second curvature test would then be in orthogonal direction and also ask for a magnitude above a threshold. In environments with predominating edge directions (horizontal/vertical in civil engineering), testing these directions first eliminates most sources of false alarms from edges.

With less ideal orientations between mask direction and intensity patterns, several rotation angles of test masks and corresponding curvature ratios C_{R2} are necessary for separating false candidates from edges and real corner features. Usually, in a small local environment several candidates are found around a real corner in the images; these have to be fused for the single most likely corner location.

11.4.3 Fusion of Local Corner Candidates

Since the nonlinearities (residues to the best local planar fit) have been selected as measure of the curvature of the intensity function, it makes sense to fuse several candidates in a small local environment by computing the center of gravity of the sum of all residues squared; this conforms to using the *traceN*-values in the 4-point nonplanarity test for the same purpose [26].

With (11.6) and (11.7) the following result can be obtained:

$$\sum \text{Residues}^2 = \varepsilon_{c5}^2 + 2 \cdot \varepsilon_{D5}^2 + 2 \cdot \varepsilon_{CD5}^2. \tag{11.10}$$

This value is computed for the initial test with the 3×3 corner pixels. Fusion of candidates is performed when the neighbor is less than R_{cg} pixels away in row or column direction; values in the pixel range $1 \leq R_{cg} \leq 2.5$ have shown good results in connection with the 3×3 initial mask. If the initial nonplanarity test is done with larger masks, the fusion range R_{cg} has to be increased.

11.5 Experimental Results in Road Scene Recognition

Figure 11.4 shows a busy highway scene with at least seven vehicles easily recognizable by a human observer. The center-deviation feature (lower part) marks locations with stronger nonplanarity of the intensity function. A common gray background ($I = 175$) shows close to planar regions. Where the central pixel is much brighter than the average of the corner pixels in the 3×3 mask, white dots are painted; similarly, where it is much darker, black dots are shown (binary coding).

From these white or black candidates the corner locations marked by triangles in the top part of the figure are selected by four rotated orthogonal curvature tests. For efficiency reasons they only ask for the ratio of curvatures in orthogonal directions to exceed a lower threshold (0.4, or $C_{R2} > 0.16$); the absolute magnitude

Fig. 11.4 Highway scene with candidates for corner features (white and black, bottom) selected by the "center-deviation test" using the average of the four corner pixels of the 3×3 mask (Fig. 11.3, bottom, ∼14%). The orthogonal curvature cross-test reduces this number to ∼3.5%, the rotated tests to ∼0.8%. After fusion of ∼260 local candidates, about 600 corner features are obtained (top, region of special interest); they are marked by triangles

is no more checked. The corners detected mostly correspond to human visual observation; only a few real corners are missed, like the lower left corner of the left rectangle on the back side of the white truck ahead, and the corners of the faint lane markings. To pick these up, the threshold values have to be lowered. Due to temporally changing lighting conditions and to changing regions of special interest, threshold adaptation should be performed continuously.

Figure 11.5 shows a case where corners of dashed lane markings have been picked up. There are also many corner features on the truck in the right lane nearby and in the textured regions of the environment (bushes and trees). With respect to the truck far away in the right lane, the dark tires are very important features since they allow range estimation in conjunction with the vertical curvature profile of the road; by fusion of close regional candidates for corners (several white dots to a single triangle) precision in recursive estimation is improved.

Another detail of importance in convoy driving is observing the stop and blink lights of cars nearby. Figure 11.6 shows a case where the car between two trucks in the right neighboring lane might want to change lane to the left into the lane of the own vehicle carrying the camera. The figure *left* shows brake lights at both sides of the rear window detected by corner features as off; the *right-hand part* of the figure taken 1 s later shows them as bright (about 100 intensity steps higher than when not

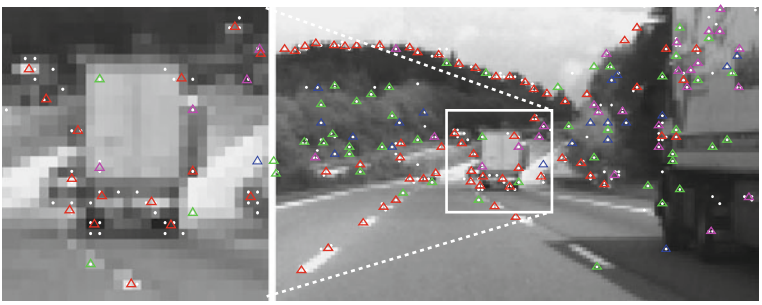


Fig. 11.5 Tracking of dashed lane markings (*right*) and of two trucks in the right neighboring lane. The zoomed region (*left*) shows fusion of candidates in a small local region (white dots) into a single corner candidate (triangle)

activated). This is not a one-sided blinking light for lane change but the indication that the vehicle is braking.

The sets of corner features on the car and on the truck at right allow precise tracking of relative speed and position of these vehicles nearby. In this case the reference is the left side of the large navigation sign extended down to the ground (*black dashed line*); with the camera looking in the direction of the own lane, a horizontal line orthogonal to the road direction is mapped into an image row (*also dashed black*). A comparison (*left* vs. *right* subfigure) of the distances on rays through the common vanishing point at infinity (not shown here) shows the distances traveled by the vehicles: The truck ahead moved only a bit; the car closed in to this truck and started braking, while the truck on the right moved into the image from

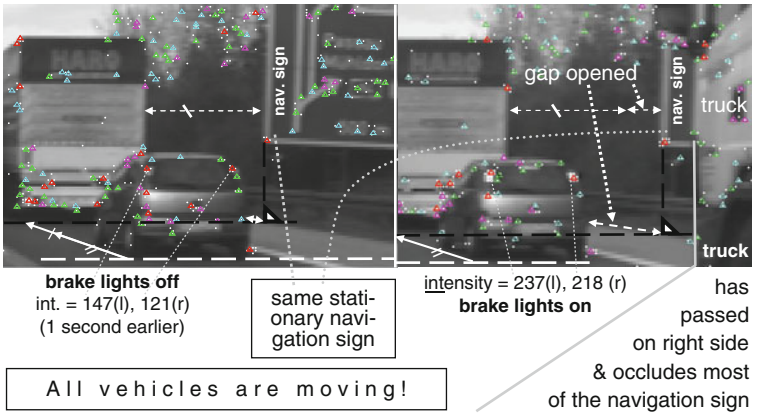


Fig. 11.6 Two frames 1 s apart in time; the camera stands still in a vehicle in the second lane. The large navigation sign (*top right*) is a stationary reference for motion of vehicles (see text); of special interest are the blink- and brake lights of the car. They are detected by corner features; their activation changes image intensity by ∼100 steps (of 256): *left*: brake lights off; *right*: on

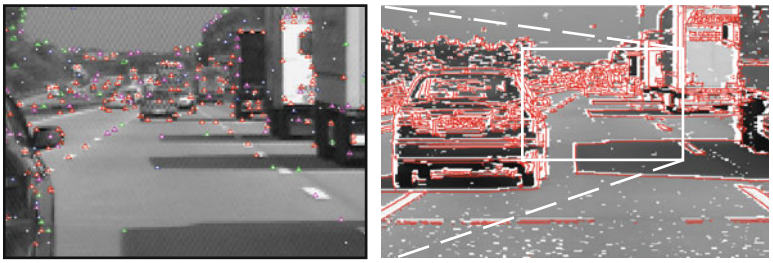


Fig. 11.7 Busy road scene with feature sets extracted: *Left*: Central part (marked right) of original (noise corrupted) image with corner features superimposed (triangles); *right*: edges and linearly shaded blobs. All features in conjunction alleviate interpretation; large blobs are efficient for recognition and tracking

the right side and occludes most of the navigation sign (hardly recognizable from its features alone).

Figure 11.7 shows the characteristics of the blob-edge-corner features extracted in the unified approach exploiting the smoothing properties of the 121-Sobel-elements. Due to intermediate storage the image contained ugly digitization noise (see *top left*). Despite this fact, satisfying feature extraction results have been achieved. The two subfigures visualize the benefit obtained when all these features are considered in conjunction during the deliberation process for associating features with real-world objects. Especially for tracking in image sequences the combination of larger blobs and corners is beneficial.

11.6 Conclusions

The smoothing properties of the 121-Sobel elements are exploited additionally in a new direct scheme for testing the curvature properties of the image intensity function in orthogonal directions in the same 3×3 mask. A threshold on the magnitude of curvature eliminates 80–95% of all locations as candidates for corners (depending on the threshold level); the second orthogonal test cuts the number of candidates remaining to one half or even to one fourth. Thus, only a few percent of the image remain as corner candidates. Stronger edges may yield false alarms; they have to be eliminated by further tests. Performing the curvature tests in row and column direction first, in road scenes with many edges in horizontal and vertical directions due to reasonable civil engineering (and, of course, the gravity vector) eliminates most false alarms from these edges right from the beginning. The remaining ones have to be removed by the same orthogonal curvature tests in rotated masks; these are selected in 5×5 or 7×7 pixel regions with the same center. For noise reduction, pixel pairs in row and column direction are used; these pairs are needed

anyway for building the Sobel elements; storing these in vectors allows avoiding 2-D coordinates after the initial grasp from the image.

In 1-D regions between nonplanar locations linearly shaded segments are assembled by a floating least squares fit; if the parameters obtained are within thresholds on intensity and gradients these segments are merged laterally to planar 2-D blobs. All these features: blobs, edges and corners, are stored in densely packed vectors, with a second vector set for efficient navigation [15].

Offline results as shown are very promising; real-time image sequence processing is in preparation. This fine-scale scheme nicely complements a new method favorable for larger scales that uses integrated images; for this scheme real-time results (200 Hz for a single image sequence) have recently been published [29].

References

1. Dickmanns ED, Zapp A (1987) Autonomous high speed road vehicle guidance by computer vision. 10th IFAC World Congr. Munich, Prepr, vol 4, pp 232-237
2. Masaki I (1992++) yearly International symposium on intelligent vehicles – Proceedings, in later years appearing under IEEE – ITSC sponsorship
3. Thomanek F, Dickmanns D (1992) Obstacle d, tracking and state estimation for autonomous road vehicle guidance. IEEE/RSJ international conference on intelligent robots and systems, IROS, vol. II, Raleigh, pp 1399-1406
4. Fuerstenberg KC, Dietmayer KCJ, Willhoeft V (2002) Pedestrian recognition in urban traffic using a vehicle based multilayer laser-scanner. IEEE intelligent vehicle symposium, Versaille
5. Wang CC, Thorpe C, Suppe A (2003) Ladar-based detection and tracking of moving objects from a ground vehicle at high speeds. IEEE-symposium on intelligent vehicles (IV)
6. Fuerstenberg KC, Dietmayer KCJ (2004) Object tracking and classification for multiple active safety and comfort applications using a multilayer laserscanner. IEEE-symposium on intelligent vehicles, Parma, pp 807-812
7. von Holt V (2004) Integrale Multisensorielle Fahrumgebungserfassung nach dem 4D-Ansatz. Dissertation, UniBwM, LRT
8. DARPA (2006) Urban challenge, route network definition file (RNDF) and mission data file (MDF) formats, May 12
9. Dickmanns ED (1995) Road vehicle eyes for high precision navigation. In: Linkwitz et al (eds) High precision navigation. Dümmler, Bonn, pp 329-336
10. Dickmanns ED (2007) Dynamic vision for perception and control of motion. Springer, London
11. Dickmanns ED (1987) 4-D dynamic scene analysis with integral spatio-temporal models. In: Bolles R, Roth B (eds) Robotics research, 4th international symposium, MIT Press, Cambridge, MA
12. Gregor R, Lützeler M, Pellkofer M, Siedersberger K-H, Dickmanns ED (2000) EMS-vision: A perceptual system for autonomous vehicles. IEEE intelligent vehicle symposium, Dearborn, pp 52–57
13. Gregor R, Dickmanns ED (2000) EMS-vision: Mission performance on road networks. IEEE intelligent vehicle symposium, Dearborn, pp 140-145
14. Goebl M, Faerber G (2007) A real-time-capable hard- and software architecture for joint image and knowledge processing in cognitive automobiles. Proceedings of the IEEE intelligent vehicle symposium, IEEE-Press, p 734-740
15. Hofmann U (2004) Zur visuellen Umfeldwahrnehmung autonomer Fahrzeuge. Dissertation, UniBw Munich, LRT

16. Kuehnle A (1991) Symmetry-based recognition of vehicle rears. In: Pattern recognition letters, vol 12. North-Holland, pp 249–258
17. Zielke T, Brauckmann M, von Seelen W (1993) Intensity and edge-based symmetry detection with an application to car following. CGVIP: Image Understanding 58:177–190
18. Schmid M (1993) 3-D-Erkennung von Fahrzeugen in Echtzeit aus monokularen Bildfolgen. Dissertation UniBw Munich, LRT Also: Fortschrittsberichte VDI Verlag, Reihe 10, Nr. 293
19. Thomanek F (1996) Visuelle Erkennung und Zustandsschätzung von mehreren Straßenfahrzeugen zur autonomen Fahrzeugführung. Dissertation, UniBw Munich, LRT. Also: Fortschrittsberichte VDI Verlag, Reihe 12, Nr. 272
20. Estable S, Schick J, Stein F, Janssen R, Ott R, Ritter W, Zheng YJ (1994) A real-time traffic sign recognition system. In: Proceedings of the international symposium on intelligent vehicles'94, Paris, pp 213–218
21. Schiehlen J (1995) Kameraplattformen fuer aktiv sehende Fahrzeuge. Dissertation, UniBw Munich, LRT. Also: Fortschrittsberichte VDI Verlag, Reihe 8, Nr. 514
22. Lützeler M (2002) Fahrbahnerkennung zum Manövrieren auf Wegenetzen mit aktivem Sehen. Dissertation, UniBw Munich, LRT
23. Pellkofer M (2003) Verhaltensentscheidung für autonome Fahrzeuge mit Blickrichtungssteuerung. Dissertation, UniBw Munich, LRT
24. Fuchs T (2008) Das Gehirn – ein Beziehungsorgan. Kohlhammer
25. Dickmanns ED (2006) Corner detection with minimal effort on multiple scales. Proceedings of Vision Application (VISAPP), Setubal
26. Dickmanns ED (2008) Generalized Nonplanarity Features. UniBwM/LRT/TAS/TR 2008–08
27. Harris C, Stephens M (1988) A combined corner and edge detector. In: Alvey vision conference, pp 147-151
28. Tomasi C, Kanade T (1991) Detection and tracking of point features. CMU, Tech. Rep. CMU-CS-91–132, Pittsburgh, PA
29. Schweitzer M, Wuensche H-J (2009) Efficient keypoint matching for robot vision using GPUs. In: Proceedings of the 5th IEEE workshop on embedded computer vision (ECVW), ICCV-Kyoto
30. Bay H, Tuytelaars T, Gool LV (2006) Surf: Speeded up robust features. In: Proceedings of ECCV

Chapter 12
System Architecture for Future Driver Assistance Based on Stereo Vision

Thomas Wehking, Alexander Würz-Wessel, and Wolfgang Rosenstiel

12.1 Introduction

The last two decades saw much research on computer vision in automotive environments. Since a few years the first and second generation of vision based driver assistance systems are available in luxury and middle-class vehicles. The offered functionality is limited to comfort tasks like road sign recognition, night view and lane keeping support. To develop more complex functions with a safety aspect like pedestrian protection, the required measurement data has to be more accurate. The scene depth information must be available and a robust model free approach is preferable.

The answer could be a stereo vision system (3D) combined with the measurement of optical flow (2D) tracked over time (1D) – so called 6D-stereo. In contrast to a monocular approach the extracted information provides much more knowledge about the surrounding of the vehicle. For this reason a possible architecture for real time automotive applications is developed and presented in this work.

12.1.1 Motivation

Today‘s driver assistance systems will improve in performance in case of more detailed and precise measurement data through 6D-stereo. The scenario

T. Wehking (✉) · A. Würz-Wessel
Robert Bosch GmbH, 71229 Leonberg
e-mail: thomas.wehking@de.bosch.com; alexander.wuerz-wessel@de.bosch.com

W. Rosenstiel
Eberhard-Karls-University, 72076 Tübingen
e-mail: rosenstiel@informatik.uni-tuebingen.de

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_12,

characteristics of a pedestrian protection system make an approach like the combination of stereo and optical flow indispensable. With this design it is possible to realize a robust and model free application. To reach this goal there are many challenges to overcome. One is, to implement the necessary algorithms within the usually limited automotive hardware and software resources (e.g. memory capacity and processing power). Due to this limitations the co-design between hardware and software is very important. A real time implementation has to process the complete analysis of an image pair within 40 milliseconds. This article describes the target system and the developed architecture for a real time 6D-stereo system.

12.1.2 Overview

The article is structured as follows. In Sect. 12.2 the 6D-stereo system is explained briefly. A short overview of the low- and high-level algorithms is given. Section 12.3 describes the system architecture in detail. System requirements are discussed and the development platform is explained. A safety concept for critical applications like emergency braking is presented. The result of the hardware/software co-design process is explained. The article closes with a conclusion and a look forward to further work.

12.2 6D-Stereo System

In the following subsections a brief overview of the 6D-stereo system is given. Details are described in [4, 5].

6D-stereo is designed as a sensor, able to support many different functions. The used measurement principles, explained in Sect. 12.2.1, generate a disparity map and a motion field. Based on this raw data an object detection procedure extracts obstacles in front of the vehicle – see Sect. 12.2.2. High level applications like pedestrian protection and emergency braking use this information to realize its functionality.

12.2.1 Measuring Base

The sensor configuration of an automotive stereo rig supports different approaches to generate raw data. 6D-stereo uses two of them.

The first is a stereoscopic 3D reconstruction via the epipolar geometry [1]. The correspondence problem between the two images is solved using the unique feature approach of [4]. For each pixel a signature is generated out of a surrounding pixel patch. All signatures are concentrated in one list per image. The comparison of

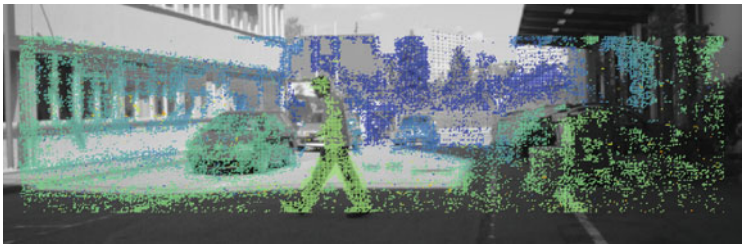


Fig. 12.1 Color coded stereo measurement in a typical road environment

Fig. 12.2 Flow measurement in the same scene like Fig. 12.1. The warmer the color the longer the flow vector

both signature lists provides the depth information in form of a disparity map. An example is shown in Fig. 12.1.

As a second principal component 6D-stereo uses the optical flow approach [2]. This motion field is induced by the ego-motion of the observer and any motion of independently moving objects. As shown in Fig. 12.2 a crossing pedestrian induces a flow dominant in horizontal direction. All other plotted measurements are static points because of a static observer.

As seen, the two measurement principals need to solve the correspondence problem in space and time. Due to the unique feature approach, the extracted features are reusable for optical flow and stereo. This advantage allows an efficient implementation.

12.2.2 Post Processing

The high-level algorithms are based on the stereo and optical flow information. To extract raised objects and their attributes is the main task. There are several algorithm modules needed to get this information. Clustering the stereo and flow measurements is encapsulated in segmentation modules. Also necessary is an estimation of the ego-motion, the compensation of it and a tracking of object hypotheses. Figure 12.3 shows the object detection result for the same scene as

Fig. 12.3 Object detection via 6D-stereo. White boxes denote those objects tracked over time

in Figs. 12.1 and 12.2. The white boxed regions denote the tracked objects. Their attributes include among others distance, dimensions and velocities.

Section 12.3 highlights the modular design of the post processing algorithms including some results. The challenge is a co-design of hardware and software to meet the requirement of real-time processing on embedded automotive qualified hardware.

12.3 Architecture

This section introduces the architectural concept of 6D-stereo. A hierarchical software structure is presented. The hardware/software co-design yields an optimized partitioning of the algorithms for the selected target-system.

The first subsection broaches essential requirements. In Sect. 12.3.2 the target-platform will be explained briefly. The part *safety concept* deals with the safeguarding of such a hardware-platform. A partitioning of the needed software algorithms is presented in the last subsection of this paragraph.

12.3.1 Requirements

Essential requirements of the stereo system, in particular the portability, real-time and safety aspect, are listed below. They are fundamental for the architectural design. Because of an other focus, the functionality, maintainability and usability are not discussed here.

12.3.1.1 Portability

For development and further improvements of algorithmic parts a system like 6D-stereo has to be manageable. The following requirements are indispensable for an effective architecture:

- The exchangeability of each algorithm and small logic processing step has to be guaranteed. That is possible in case of modularity and strict interfaces.
- Every logic unit should be established as a separate module. This makes sense till a defined granularity. Interfaces generate a substantial overhead for each module. The criteria to split an algorithm or not are explained in Sect. 12.3.4.
- Interaction between two software modules follows strict rules. Exact one interface has to manage the data transfer to the next processing instance.
- The algorithms have to be independent from the target system. For example, the realization of an obstacle detection algorithm has to use the same implementation even if it is integrated on a *personal computer* or an *electronic control unit*. This reduces the work load and the failure rate of the porting. Section 12.3.4 explains the used software structure.

12.3.1.2 Real-time

For embedded systems the real-time requirement is very strict. The refresh rate of 6D-stereo is 25 Hz, hence every 40 ms a new image pair with a resolution of 2×1,024×512 pixel and 2 MB data volume is taken. The complete process chain has to keep this cycle time. For minimal response time and maximum effectiveness of the assistance functionality it is important to fulfill this condition. To develop an optimized architecture for the target system the co-design between hardware and software is necessary. Algorithms which are in parts suitable for implementation in hardware have to be split into separate modules.

12.3.1.3 Safety

An ambition of 6D-stereo is to provide safety applications like emergency braking. The measuring base and the almost model free obstacle detection approach are a good basis. But there has to be a concept how a system failure is detected and handled. Section 12.3.3 discusses this requirement.

12.3.2 Development Platform

The platform which is used for the design process and implementation consists of one *field programmable gate array* (FPGA) and one *microcontroller*. They are connected via a *peripheral component interconnect (PCI)* interface. Such a setup stands for a maximum of flexibility and supports efficient realizations. Without any hardware support, a high-end consumer processor would be required. Thats not possible in an automotive environment. Figure 12.4 shows a schematic overview of the target.

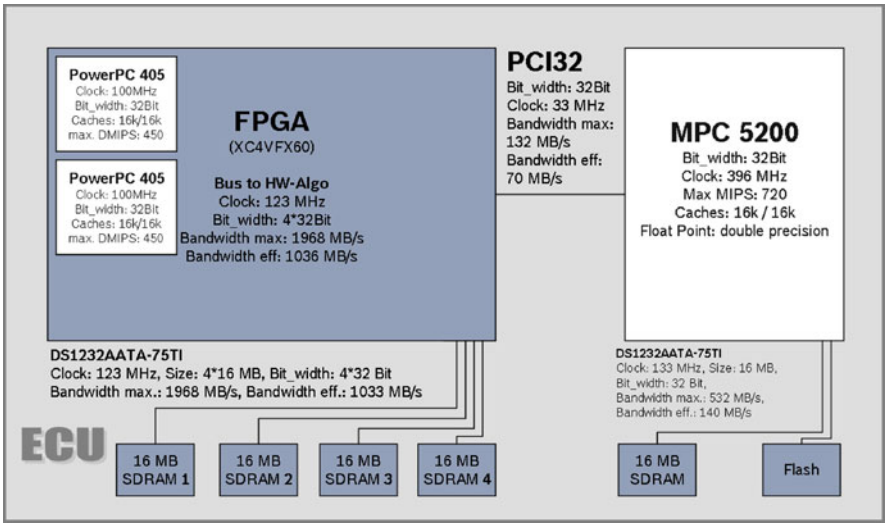


Fig. 12.4 Architecture of the used electronic control unit (ECU)

12.3.2.1 FPGA

The used FPGA is a Xilinx Virtex 4 FX60. Each of its 25,280 slices covers two registers and two look up tables. The two embedded Motorola PowerPC 405 cores are not included in the architecture because of transparency and system complexity. Four SDRAM modules, each with 16 MB, clocked with 123 MHz can read and write 32 Bit words at an effective bandwidth of 1033 MB/s.

12.3.2.2 Microcontroller

A Freescale MPC5200 microcontroller handles the post processing. With its RISC architecture and a floating-point-unit it accesses a 16 MB SDRAM module and a flash memory. Separate caches for data and instruction – each 16 k – are installed. At 396 MHz clock the microcontroller processes 32 Bit words.

12.3.2.3 PCI bus

The PCI connects the FPGA with the microcontroller. With a clock of 33 MHz and 32 Bit words the effective bandwidth of the PCI bus is 70 MB/s. Because of the low-level image processing a fast data reduction is done through the FPGA program. So the small bandwidth of PCI is no bottleneck.

12.3.3 Safety Concept

In case of safety critical applications, 6D-stereo must cope with unnoticed failures. The following section presents briefly a concept for the architecture consisting of FPGA and microcontroller.

12.3.3.1 FPGA

As seen in [3] FPGAs are sensitive against transient failures. One transient failure is negligible, because of no temporal logic correlation between two processing cycles. If the program code is affected, the failure is permanent till the next system reboot. This means that the FPGA configuration must be tested against transient-permanent and systematic failures.

To detect failures in the FPGA processing results, at the beginning and ending of every image a test pattern is integrated. This pattern is processed with the same algorithm like the current image. The correct results are stored in the microcontroller memory and compared with the current result. Only in case of equality the test is successful and the release is given to the safety controller (SCON). See Fig. 12.5 for a schematic illustration.

12.3.3.2 Microcontroller

The post processing tasks of the microcontroller contains tasks with temporal correlation. For this part, transient failures are not negligible. Permanent failures are also fatal. Both must be detected solidly. A concept with comparator and two different algorithm chains prevents false behavior. Two assumptions are required for this comparator concept:

- Based on the same FPGA preprocessing it is possible to develop two sufficient diverse algorithms for the complete post processing.
- Two diverse algorithms deliver not the same result on defective hardware.

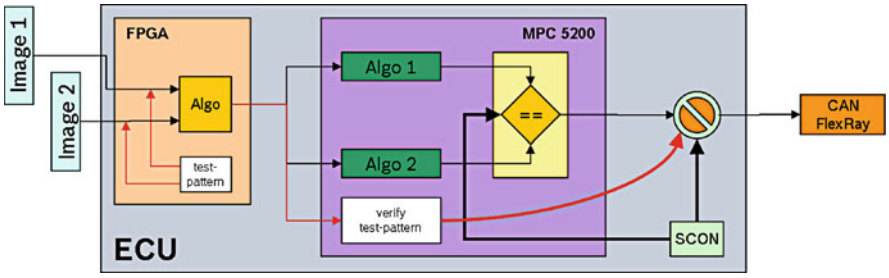


Fig. 12.5 Safety concept with test pattern and comparator

With these assumptions, the comparison of the results of the two algorithms is a single point of failure. Which situations are possible and how can we detect the failure of the comparison:

- Permanent equality (stuck at 1)
  For observation of the comparator a safety controller (SCON) is inserted. It tests the comparator cyclic on correct functionality. In case of a false response, the vehicle bus remains locked. The assistance functionality is not given, but the system is safe.
- Permanent inequality (stuck at 0)
  This is an uncritical case, because of the vehicle bus remains locked. The system is passive but safe.
- Transient failure
  Relating to the failure latency and the mechanical inertia, this is an insignificant and uncritical case.

12.3.4 Software Structure and Partitioning

This section discusses the hierarchical software structure and the partitioning of the algorithms needed for 6D-stereo.

12.3.4.1 Hierarchical software structure

In the development of a new driver assistance system there are two elementary different steps. At first a feasibility study of an idea – mostly implemented on a high-performance personal computer. At a next step the algorithms get optimized for execution on the development platform. Each system uses an proprietary, incompatible framework. Therefore we use a concept which simplifies this process.

Basic functionalities, interfaces, main algorithms and realizations in special frameworks are strictly divided. Figure 12.6 shows the hierarchical software structure as scheme. The left part displays a main algorithmic module which uses basic video functionality, basic stereo functionality and needed interfaces. In the block *main algorithmic module* the main logic of this module is implemented. A realization in PC or ECU environment – see right part – bases on the same algorithm components. So the essential logic is separated from the platform. The same code is used for PC and ECU realization.

12.3.4.2 Partitioning

To fulfill the requirements of Sect. 12.3.1 the 6D-stereo algorithms had to be partitioned. An iterative hardware/software co-design process was initiated to split

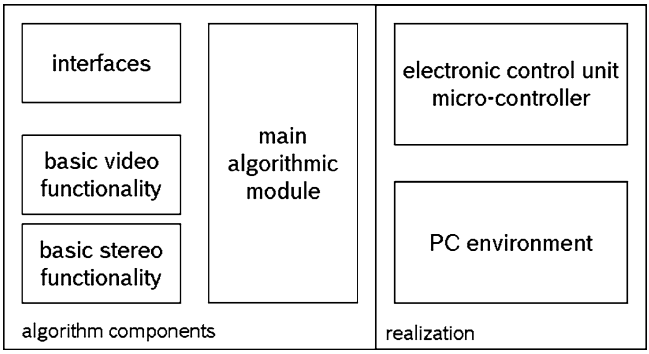


Fig. 12.6 Hierarchical structure of software modules

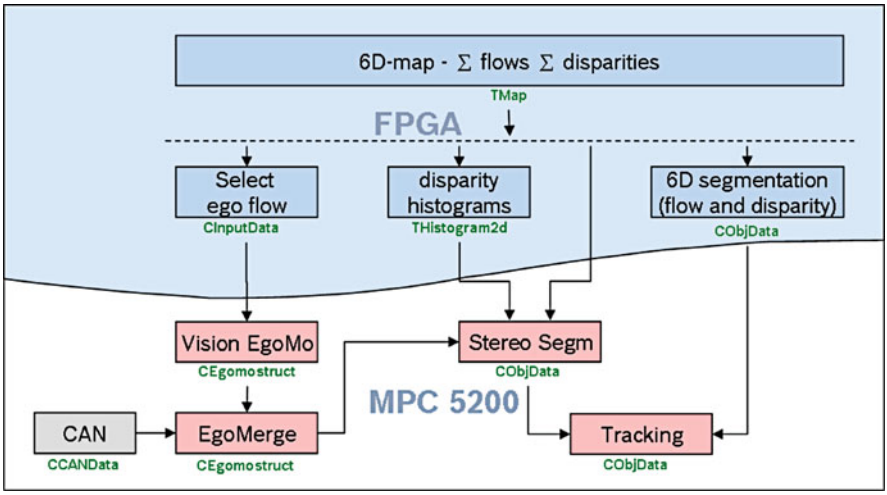


Fig. 12.7 Software module partitioning between FPGA and microcontroller

the tasks between FPGA and microcontroller. Parts which could be parallelised are applicable for hardware implementation. Tasks which work mostly sequential and need e.g. a floating point unit must run on the microcontroller. Figure 12.7 shows the resulting modules. For more detailed information on the algorithms themselves see [5]. For prototypical implementation and the feasibility study of real-time performance the safety concept is not integrated in the software partitioning.

The upper half of the chart in Fig. 12.7 denotes the modules to be implemented by means of a hardware description language. They were realized with FPGA resources. The dotted line symbolizes a dissociation between measurement program and further algorithms based on the result. Microcontroller tasks are shown in the lower half. The interfaces with its identifiers are listed below every module

12.3.4.3 Modules

- Vision EgoMo
  The vehicles ego-motion estimation is split for efficient implementation into two modules. *Select ego flow* is a FPGA task and preselects flow measurements. *Vision EgoMo* takes these preselection via the *CInputData* interface and estimates the ego-motion iteratively on the microcontroller. The result is presented via the *CEgomoStruct* interface.
- CAN
  Ego-motion and further vehicle information from inertial system sensors are provided by the module *CAN*. They are available without any calculation routines.
- EgoMerge
  Both inputs from *CCANData* and *CEgomoStruct* were fused by *EgoMerge*. A prioritization is given by the confidence of the input data. In case of equal confidence the vision based ego-motion has higher priority because of synchronisation relating to the current images and measurement data. The result is sent via *CEgomoStruct* interface to following modules.
- 6D-map
  The FPGA module *6D-map* holds the information described in Sect. 12.2.1. *TMap* is the used interface for this data.
- Stereo Segm
  Based on *6D-map* the stereo measurements are conditioned in the module *disparity histograms*. A structure of *THistogram2d* is applied and *Stereo Segm* called. An obstacle detection algorithm based on stereo information is implemented. Including ego-motion and optical flow information, the object attributes are calculated. The resulting object list is presented via the interface *CObjData*.
- 6D segmentation
  Another obstacle detection algorithm is integrated completely in FPGA resources. *6D segmentation* contains a connected component labeling approach based on optical flow and stereo disparity. The determined object information is provided via *CObjData*.
- Tracking
  To fuse the object information of *Stereo Segm* and *6D segmentation* the module *Tracking* is built in. A temporal history is integrated as well as unique identification, obstacle velocities and quality criterions. The final object list uses the interface *CObjData*.

12.4 Conclusion

The presented work gives a brief description about the 6D-stereo concept in Sect. 12.2. As one can see the measuring base established of stereo and optical flow offers the potential for model free obstacle detection. That allows robust driver assistance applications.

Main subject of this article is the architecture in Sect. 12.3. At first the architectural requirements portability, real-time and safety are discussed. With regard to a prototypical implementation a development platform is presented. Section 12.3.3 illustrates a concept for safeguarding the hybrid system consisting of FPGA and microcontroller. In the last subsection the partitioning of software algorithms is described. Within a hardware/software co-design process the split of modules is developed.

The described approach provides a good base for new driver assistance applications. Among others there are a robust obstacle detection, three-dimensional scene information and a concept to follow the safety requirement. With this base an emergency braking function is applicable. Furthermore existing functions like road sign recognition profits by 6D-stereo, too. Information about distance and motion of road signs could raise the robustness.

Future work on 6D-stereo will be concentrated on the ongoing integration of all algorithm modules onto the development platform. Therefore the requirement of real-time must be kept. In addition an extension of the confidence and robustness validation of the detected obstacles is necessary. Furthermore prototypic driver assistance systems should be realized.

References

1. Hartley R, Zisserman A (2003) Multiple view geometry in computer vision, 2nd edn. Cambridge University Press, Cambridge, p 237ff
2. Jähne B (2005) Digitale Bildverarbeitung, 6th edn. Springer, Berlin, p 423ff
3. Ken O'Neill (2004) Natürliche Strahlung verursacht Fehler in FPGA's, elektronik industrie, vol 11, pp 52–53
4. Lorei M, Würz-Wessel A, Heger T (2008) 6D-Stereo ein robuster und schneller Ansatz zur Objektdetektion, VDI-Berichte 2038, pp 85–92
5. Wehking T, Würz-Wessel A, Rosenstiel W (2009) 6D-Stereo Video für moderne Fahrerassistenzsysteme. Workshop Fahrerassistenzsysteme, Löwenstein/Hößlinsülz, pp 20–28

Chapter 13
As Time Goes By: Research on L4-Based Real-Time Systems

Hermann Härtig and Michael Roitzsch

13.1 Introduction

The ideas behind the L4 microkernel were born back in the mid-1990's when Jochen Liedtke reexamined the design of the earlier generation microkernels around Mach. Trying to prove that a minimal kernel can still provide high system performance, he developed first L3, then L4. The fundamental principle of his microkernels is that a concept will only be allowed inside the kernel, if user-land implementations would be unable to achieve the required functionality. This leads to truly minimalist kernels supporting only address spaces, threads and interprocess communication. These basic services are enough to run isolated user-level processes on top of L4. Any additional functionality must be implemented as a server process. This includes components like file systems, networking and even device drivers, all of which are usually subsumed as an operating system personality.

13.1.1 Diverse Platform Requirements

Roughly at the same time, multimedia applications were pushing forward into mainstream computing, because the required performance became increasingly available to consumers. The characterising new requirement of those systems was the coexistence of highly dynamic real-time and non-real-time workloads, sharing computer cores, disks, video subsystems and networks. Previously, real-time systems used to be dedicated, having the complete hardware for themselves. Now, both real-time and non-real-time applications are running side by side, launched and stopped at the user's discretion. But although the requirements towards the system

H. Härtig (✉) · M. Roitzsch
Technische Universität Dresden, Department of Computer Science, 01062 Dresden, Germany
e-mail: haertig@os.inf.tu-dresden.de; mroi@os.inf.tu-dresden.de

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_13,

changed, the basic architecture of the underlying operating systems stayed the same. Instead, software vendors tried to solve the emerging problems in middleware. We believe this approach is misleading, because no middleware can isolate real-time from non-real-time tasks or reliably enforce resource guarantees for real-time applications without proper core operating system support.

13.1.2 Resource Overprovisioning

Advancements in computer hardware achieved enormous performance improvements by using caches to exploit the locality of the applications' behavior. However, those techniques are not necessarily useful for real-time systems, because they tend to concentrate on improving the average case, whereas real-time applications must consider the worst case. For the longest time, dedicated real-time systems had been using less powerful, yet expensive specialized hardware to overcome this. On standard computer hardware, these problems were traditionally dealt with either not at all or by spending enormous amounts of resources. But all those overprovided resources are usually wasted, because the average case behavior is much more benign than the rare but devastating worst case situations, and this gap continues to widen as technology progresses. We intend to solve this problem by dealing with overload situations in ways other than the overprovisioning approach. With a task model that makes the powerful commodity hardware analyzable, we can allocate resources much closer to the average case.

13.1.3 Virtualisation

We do not want to explore our ideas just theoretically, but strive to build a system usable on a daily basis. To this end, we have to support existing commodity software without compromising on the real-time aspects of the system. One way to achieve this is by designing a real-time kernel from the ground up and reimplementing the interface of a commodity kernel in this system. This is the approach taken by the QNX [10] real-time operating system. However, matching the personality of an operating system by reimplementing it is expensive, because you are dealing with a moving target.

Alternative to enhancing a real-time kernel with a commodity personality, you can try to enhance a commodity kernel with real-time capabilities. This is the approach taken by RTLinux [18], which runs the legacy kernel next to high priority real-time processes on top of a small real-time executive. However, all real-time tasks run along with the real-time executive in kernel mode. Thus, there is no isolation between different real-time tasks, which weakens system security. A subverted real-time task alone can potentially take over the entire system.

The third alternative is to use two kernels in one system: a real-time kernel controls the actual hardware and a commodity kernel serves existing software. This is made possible by virtualisation technology. After being introduced by IBM in the 1960s, virtualization has experienced a renaissance in recent years. It has become a major industry trend in the server context and is also popular on consumer desktops. We believe virtualisation is a powerful complement for microkernels. Using L4 as the basis for virtualization and as an advanced microkernel provides a best-of-both-worlds combination.

13.1.4 Overview

We first present some design ideas for the whole system in Sect. 13.2. We then discuss our resource management ideas to handle overload conditions. We illustrate the main concepts with the CPU resource in Sect. 13.3. We continue to manage reservations for resources such as disk, network and graphics bandwidth in Sect. 13.4. With those key building blocks in place, we bring in support for commodity software using virtualisation in Sect. 13.5. We make sure the previously discussed real-time guarantees are preserved. In Sect. 13.6, we conclude by tying all the pieces together into a real-time component architecture that makes the research results readily available for the software development process.

13.2 Designing the System

A major change in computer systems was the emerging coexistence of real-time and non-real-time applications on the same machine. Today, a large variety of systems has to support a diverse set of such use cases:

- Multimedia applications are used on the average desktop. These applications have immediate real-time requirements, because frames need to be delivered to the display at fixed time intervals. Although deadline misses are not catastrophic, they diminish the user's media experience because they will be visible as motion judder or even frame drops.
  At the same time, non-real-time components may be running next to the player core. For example the media library and subscription management common to today's integrated client applications such as iTunes are clearly non-real-time tasks.
- Off-the-shelf computers are used for sound applications like multitrack editing and live recording of music instruments. Contrasting the media player scenario, enforcing a lower bound on throughput is not the only requirement, but a low latency is needed as well. Music artists can notice delay, if the sound from the speakers is more than 10 ms behind the key hit on the keyboard.

But arranging the sound in the user interface into multiple tracks, choosing instruments, tweaking the sound and managing media assets has no real-time requirements.

- Mobile phones are being used increasingly for personal information management. Calendar and address book applications are running, as well as sophisticated OpenGL games. Alongside, the phone still needs to handle the GSM protocol in a timely manner and once a call is accepted, the speech encoder needs to deliver data from the microphone to the mobile network within certain bandwidth and latency bounds.

From these examples, we can observe that applications with real-time requirements often have small, isolated real-time core functionality surrounded by a large and complex non-real-time part. Current mainstream operating systems do not honor this separation but treat both parts equally, often they are even co-located in the same address space, so proper isolation of resource reservations is impossible. Consequently, current systems have to overprovide resources so that the requirements of the real-time part are satisfied even if it is treated as a best-effort task only.

A system better suited for such applications would provide reliable real-time guarantees to some components while still providing support for the bulk of the existing non-real-time applications. Such a dual personality is made possible by a system with a real-time capable foundation and an environment for commodity applications. Due to its open-source nature, we chose Linux as the commodity environment, which we run virtualized on the Fiasco microkernel. Fiasco guarantees response times and schedules according to a fixed priority regime. Next to the Linux personality for non-real-time tasks, our system can thus provide a real-time personality. Every basic resource such as CPU time and main memory is wrapped by a manager which provides the resource to real-time and non-real-time system components and applications. Using this manager, real-time components like a filesystem can provide an interface with reservations and guarantees for real-time applications and a best-effort interface for Linux. This whole architecture on top of our Fiasco microkernel is codenamed the "Dresden Real-time Operating System", in short DROPS [7] (see Fig. 13.1).

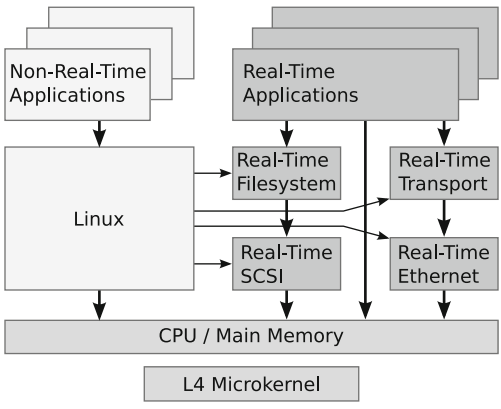


Fig. 13.1 The DROPS architecture [7]

Other than RTLinux, our DROPS architecture allows for strong separation of real-time tasks from each other and of real-time tasks from the kernel, because those tasks all run in their own, isolated address space.

13.3 Probabilistic Scheduling

As the basic architecture of the system is now in place, it is time to consider resource scheduling in real-time systems. The primary resource real-time system designers focus on is CPU time, because it provides the basis for all subsequent resource accesses. Scheduling the CPU is all about timeliness: Real-time applications provide a deadline and require sufficient CPU time to finish their job before the deadline is reached. The system scheduler must guarantee this property to all real-time tasks it admitted to run. However, the key problem here is how to determine, what "sufficient CPU time" means. Today's hardware makes a lot of effort to speedup applications in the average case by using caches to exploit locality in the application's behavior. Unfortunately, this widens the gap between the average case and the worst case time consumption. In addition, a common real-time application in desktop computing is video playback, which per se does not have a fixed execution time per job (Fig. 13.2).

These two factors cause execution time distributions to have a long tail. If the CPU resource was always allocated for the worst case, the system's utilization would be very low, because only few jobs can be admitted and large amounts of resources would be wasted. But media applications are an example for a class of real-time tasks that can tolerate occasional deadline misses, if this does not happen too frequently. With this observation, we devised a system that can handle overload predictably without dedicating enormous amounts of resources [4]. Our idea is to allow a percentage of deadlines to be missed; applications can configure this percentage as a quality level. Resource reservation is based on the distribution of the execution time instead of just the worst case value. Competing approaches in this area such as Imprecise Computation [12] and Statistic Rate Monotonic Scheduling [1] are either based on deterministic duration of resource usage or cannot guarantee a desired quality.

The task model of our Quality-Assuring Scheduling (QAS) allows each real-time task to be split into mandatory and optional parts. The mandatory parts are always guaranteed to be executed before their respective deadline, so worst case reservation is performed. Of the optional parts, only the percentage requested by the quality parameter is guaranteed to complete execution before the deadline. For these parts, admission and reservation is performed using the distribution of the execution time. Caused by the typical long tail of these distributions, even requested qualities only slightly below 100 % cause considerably less resources to be reserved, which greatly increases the overall utilization of the system. Although the scheduling is probabilistic, the requested quality levels are matched quite accurately, as the following table proves. The results in Table 13.1 were obtained for MPEG decoding

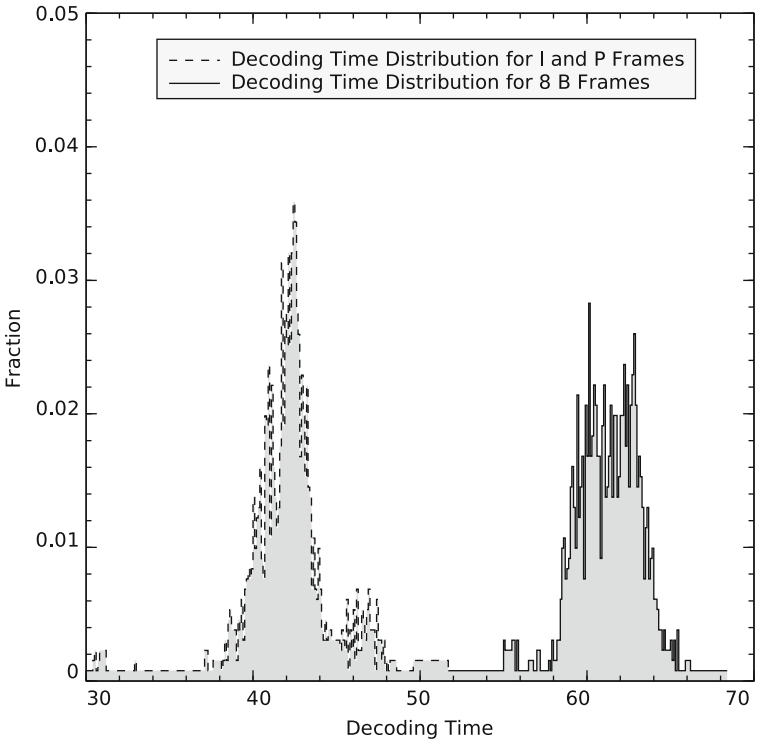


Fig. 13.2 Measured distribution of total decoding times per group of pictures [4]

Table 13.1 Requested quality, derived reservation time and measured quality of the optional parts [4]

Requested quality	Reservation time for optional parts	Achieved quality
0.95	55 ms	0.9506
0.90	53 ms	0.8588
0.80	47 ms	0.7875
0.70	39 ms	0.6740
0.60	32 ms	0.5804
0.40	23 ms	0.4063
0.20	9 ms	0.2451

with the I- and P-frames as mandatory parts and the B-frames as optional parts with the given quality.

Unfortunately QAS' applicability is limited because it only handles periodic tasks with uniform and harmonic periods and the admission is expensive, especially when the distributions are to be calculated with a high resolution. The model can handle arbitrary periods as well, but then the admission cost is increased beyond practical applicability. Therefore, the designated successor of QAS is QRMS, the Quality-Rate-Monotonic Scheduling [5]. It simplifies QAS by assigning the

Table 13.2 Requested and achieved quality of QAS and QRMS for a task system with three concurrent tasks [5]

Requested Quality	Quality achieved with QAS	Quality achieved with QRMS
0.70	0.7001	0.7024
0.50	0.5019	0.6742
0.7323	0.7326	0.7324

mandatory and optional parts a unified reservation time, which is regarded as constant in the admission control. Thus, QRMS ignores situations where jobs do not completely consume their reservation. QRMS is therefore more pessimistic than QAS, but has a tremendously simpler admission even for arbitrary periods. The results in Table 13.2 convince of the accuracy and feasibility of the model.

Another interesting property of both QAS and QRMS is that applications can be notified when the optional parts overrun their deadline. This way, an application can react, for example by reducing quality:

```
set_period(period);
reserve_time(mand_time, mand_priority);
reserve_time(opt_time, opt_priority);
do {
  begin_period();
  try {
    do_something();
  } catch {
  exceeded:
    adjust_quality();
  }
  next_reservation();
  try {
    do_something_else();
  } catch {
  exceeded:
    discard_result();
  }
} while (!end);
end_period();
```

With these unified admission and scheduling schemes, we can give probabilistic guarantees for periodic real-time tasks. Considering the variation of execution times allows us to admit far more applications and thus achieve better resource utilization than in systems based on worst-case admission.

13.4 Resource Management

The previous chapter dealt with scheduling the CPU, but this is not the only resource a real-time application might need. In our DROPS system architecture, all resources used concurrently by multiple tasks must be encapsulated and scheduled

by a resource manager running as a server in user-land. In the following, the design of such managers for resources like disk, network and graphics bandwidth is presented.

13.4.1 Disk Requests

Disk usage in modern systems combines traditional best-effort file access with storage and retrieval of real-time streams, such as audio and video data. Especially the latter must meet deadlines for high-volume disk requests. For good overall performance, the disk-request scheduler has to optimize the disk utilization as well. This is challenging, because the construction of disk drives causes a poor ratio of average and worst-case execution times. However, the same idea that has been successfully applied to CPU scheduling as discussed in the previous section also helps here: If an application can tolerate occasional deadline misses, probabilistic service guarantees can substantially improve the disk utilization compared to guarantees based on the worst case [16].

Thus, the basic idea is again to split real-time disk requests into mandatory and optional requests and to assign a quality parameter to the optional requests, which denotes the percentage of requests that must be completed. To optimize utilization, requests should be scheduled with the SATF (shortest access time first) algorithm, which is aware of the position of the drive's head on the disk. However, this scheduler does not know anything about deadlines. But instead of implementing a new scheduler, we devised a method to decouple the scheduling of the disk requests from the deadline and reservation enforcement [16]: The Dynamic Active Subset (DAS) always includes all pending requests that can be executed in any order without violating any deadline or reservation. This subset of disk requests is recalculated after every request completion and if enough time is available, the set even includes non-real-time requests to increase utilization. The SATF scheduler or any other scheduler can be run on this set to pick the request to execute next without having to know about deadlines.

With this technology, the disk request scheduler matches the desired quality levels of the tasks (see Table 13.3).

Table 13.3 Requested and achieved quality for a disk (IBM Ultrastar 36Z15) loaded with five concurrent streams [16]

Bandwidth	Requested quality	Achieved quality	Achieved bandwidth
640 KB/s	0.99	0.9973	638.54 KB/s
2,560 KB/s	0.95	0.9798	2,509.12 KB/s
1,280 KB/s	0.90	0.9444	1,209.36 KB/s
640 KB/s	0.85	0.9004	576.44 KB/s
1,280 KB/s	0.60	0.6705	858.65 KB/s

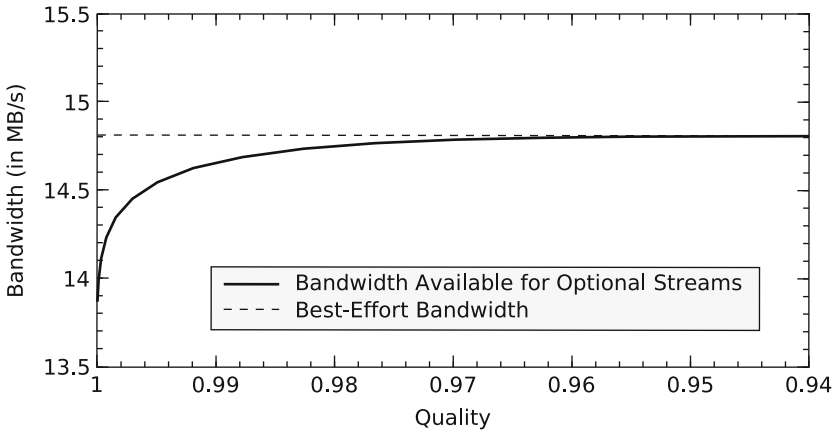


Fig. 13.3 Bandwidth that can be assigned to an optional stream [16]

Even quality levels only slightly below 100 % push the disk utilization close to the peak best-effort bandwidth (see Fig. 13.3).

13.4.2 Ethernet Transmission Delays

With the deployment of switches, Ethernet as the most widely used commodity network becomes interesting for real-time communication. Each port of a switch provides its own collision domain, so collisions do not occur in a star topology network. However, switches generally lack traffic policy features. Thus, if too many Ethernet frames are being sent to a machine that does not receive them fast enough, the switch will enqueue the frames internally, which causes transmission delays. If the internal queueing storage of the switch is depleted, it will even drop frames.

A mathematical model of the network traffic can be used to predict the buffer fill levels in the switch. If the nodes within the network cooperate, this model can be used to parametrize a traffic shaper running on each node that keeps buffer lengths and thus transmission delays within specified bounds [13].

The achievable delay bound (Table 13.4) mainly depends on the granularity of the traffic shaping. This results in a trade-off between delay bound and CPU load (Fig. 13.4).

Because all nodes on the network must cooperate to ensure the guaranteed delay bounds, each node must run an instance of the traffic shaper. However, not all nodes must run a real-time operating system. The shaping capabilities of the machines influence the delay bound, but we successfully shared the network with Linux machines while still observing predictable delays.

Table 13.4 Buffer bounds in the switch and transmission delay bounds [13]

Shaping interval	Buffer bound	Calculated max. delay	Observed max. delay
10 ms	111.8 KB	9,357 μs	8,759 μs
1 ms	15.7 KB	1,380 μs	1,300 μs
100 μs	6.1 KB	582 μs	438 μs

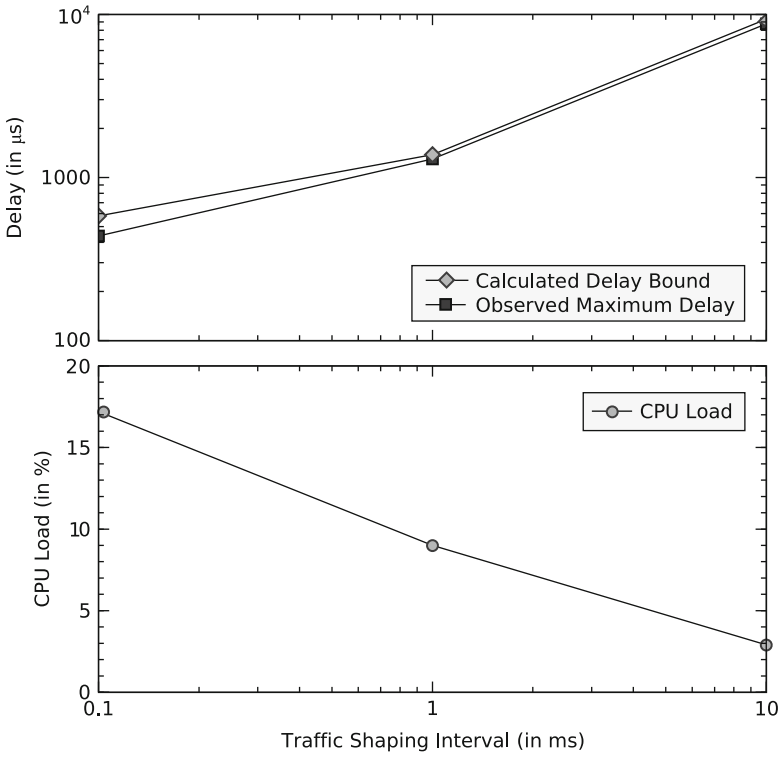


Fig. 13.4 Delay bound/CPU load trade-off [13]

13.4.3 Screen Real Estate and Screen Updates

Today's modern desktops feature a graphical user interface. Furthermore, modern real-time applications like media players also feature a graphical output. This drives the need for a real-time capable window manager that can provide guaranteed redrawing rates for the real-time windows while providing best-effort services to the remaining non-real-time windows and auxiliary operations such as the user reordering windows. Therefore, the design goal of our DOpE (Desktop Operating Environment) window server [3] was to multiplex the singleton resource of physical screen real estate to client applications. For real-time clients, quality of service is guaranteed even in overload situations, which can be caused by massive screen

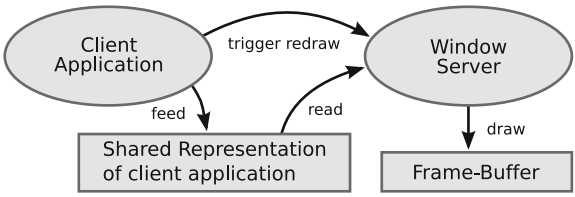


Fig. 13.5 Design of the DOpE window server [3]

updates of non-real-time applications. DOpE can therefore sustain real-time client windows running next to L^{4}Linux and X11 on the same desktop.

The architecture of DOpE (Fig. 13.5) separates the client's updates to the user interface from the server updates of the representation on screen. The client and the server share a description of the layout and content of the client's user interface. This allows the client to update the shared description without interference of the server and then trigger a redraw operation. The server can then interpret the shared window description and perform the necessary updates to the on-screen representation independently of the client. Because the execution time of such a redraw is known beforehand, the window server can guarantee previously negotiated refresh rates to admitted real-time clients. A real-time client can subscribe to periodic notifications of completed redraw operations. Updating the shared representation in a timely manner is entirely the responsibility of the client.

This separation of cause and execution of redraw requests allows us to display real-time graphics and windows of non-real-time clients seamlessly side by side.

13.4.4 Second-Level Cache

One easily overlooked resource used concurrently by real-time and non-real-time tasks are CPU caches. They are an especially interesting resource for real-time applications, because every task switch potentially disrupts cache working sets and thus makes execution times unpredictable. To avoid this, the CPU caches should be managed like all the other resources discussed above to isolate the real-time tasks from cache interference by other tasks or the operating system. A well-known solution for this problem is cache partitioning: portions of the cache are dedicated exclusively to specific applications. For our system, we developed a cache partitioning technique that operates without any hardware modifications [11].

A page size of 2^p divides the cache in banks of 2^p bytes, if the cache is direct-mapped. The least significant p bits are used to index an element within such a bank. Assuming a cache size of 2^c, the next $c - p$ bits in the address select the cache bank. The remaining part of the address is compared against the tag. For an n-way set-associative cache, a cache size of $n2^c$ and a bank size of $n2^p$ are to be used. The division of the cache into banks also divides the main memory into classes, whose physical page frames all fall into the same cache bank. Those classes are called colors. Cache conflicts can only occur between page frames of the same

color, so such conflicts can be avoided between any two tasks, if both tasks use disjoint colors. Since the L4 microkernel allows user level memory management, the mapping of physical to virtual addresses can be controlled by a memory server that assigns colors to tasks exclusively.

The problems with this approach are: Being based on the mapping of pages, it can only be applied to physically-indexed caches and only with page granularity. Additionally, if a certain percentage of the cache is to be dedicated to a task, the same percentage of the main memory is implicitly reserved for that task as well. On the other hand, the technique provides a way to close the gap between the average case and the worst case execution time for real-time tasks. This greatly helps when scheduling real-time tasks with hard deadlines, because less CPU resources need to be reserved.

13.5 Virtualisation

To provide a container for non-real-time and commodity applications, we want to reuse an existing operating system personality. This will ease splitting applications into a real-time and a non-real-time part that can then be treated by the OS differently. Because of its availability in source code and its wide range of application software, the operating system personality of choice is the POSIX personality of the Linux kernel.

13.5.1 L⁴Linux

At the time the DROPS project was conceived, hardware support for virtualisation was not available on the pervasive x86 architecture, so a software solution was needed to co-host two operating systems personalities on one machine. As the Linux kernel expects to run in CPU privileged mode, it has to be deprivileged to allow for resource control. This is achieved by replacing privileged instructions with calls to the virtualization layer, which can thus exercise control over the virtual machine. This approach is called *paravirtualization*.

We use Fiasco as the basis for paravirtualization and address spaces as the isolation primitive. Consequently, our port of Linux to L4 is called L⁴Linux [6]. To be usable for commodity software, paravirtualization must be fast. A performance decrease of a Linux application running on L⁴Linux instead of native Linux is expected, so we used the AIM multiuser benchmark suite VII to quantify the slowdown. The benchmark tests, how well multiuser systems perform under different application loads. Figure 13.6 compares monolithic Linux with L⁴Linux. To compare the performance of Linux on different microkernels, results for an in-kernel and a user-level version of MkLinux, a port of Linux to the Mach

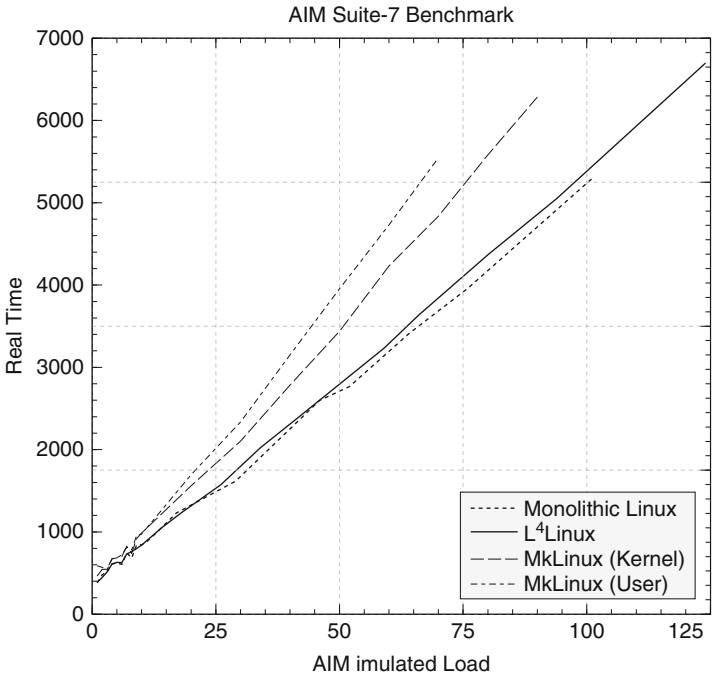


Fig. 13.6 Time per benchmark run depending on AIM load units [6]

microkernel, are also listed. The numbers in Fig. 13.6 were obtained in 1997 on a 133 MHz Pentium.

Averaged over all loads, L4Linux is 2.2 % slower than native Linux. User-mode MkLinux is on average 29 % slower than native Linux, the co-located in-kernel version of MkLinux is 21% slower. This demonstrates that L4Linux performs sufficiently close to native Linux, even under high load. Typical penalties range from 2% to 10%. The comparison with MkLinux shows that the performance of the underlying microkernel has a profound influence on the performance of the applications.

One of the goals of our architecture was to ensure guarantees for the real-time tasks even when they are running next to L4Linux. We confirmed the effectiveness of our solution by measuring the actual periodicity of an L4 real-time tasks that requests a 100 ms period from the system [8]. Running standalone on L4, the period length deviates by about 1–7 μs. With L4Linux running next to it, the real-time task shows deviations of 24 ms, so the response times do increase compared to a real-time task running standalone on L4, but the periodicity of 100 ms can be supported. These results were obtained in 1998 on the original L4 implementation by Jochen Liedtke and motivated the development of Fiasco as a real-time time microkernel to improve the response times.

To further evaluate the real-time performance of our system, we wanted to compare it against RTLinux. With our DROPS system using a separate address space for each real-time task to increase fault-tolerance, a degradation of response times is expected compared to RTLinux, which runs all real-time tasks as kernel-level threads. To compare both systems, we developed the L4RTL library, which implements the RTLinux API on DROPS and measured interrupt response times on 1.6 GHz Pentium 4 [14]. To actually get worst case behavior, we ensured that caches and TLBs were always cold when an interrupt occurred. The measurements yield a worst case latency of 24 μs on RTLinux and 33 μs on DROPS. This shows that the cost of using address spaces for real-time tasks is not significantly larger than uncertainties introduced by dirty caches or blocked interrupts, which designers of real-time systems seem to accept readily.

We summarize that modifying Linux to run on top of our L4 microkernel Fiasco allows a performance close to native Linux without compromising the real-time properties of the system. Using separate address spaces for real-time tasks increases the fault-tolerance and security of the system without a significant impact on response times. This allows running real-time and non-real-time applications side by side, which we believe to be a key feature for today's computing requirements. With the presented system architecture, we can also support tasks that want to use both real-time and non-real-time services. Those hybrid tasks can communicate with the L^{4}Linux part and with the real-time resource managers.

13.5.2 Full Virtualization

The one downside to the L^{4}Linux approach is its maintenance cost. L^{4}Linux started out based on Linux 2.0 and has since been adapted to new Linux versions as well as new L4 systems. Even though the required adaptations to Linux are small, updating to newer Linux versions calls for intimate knowledge on both the Linux kernel and L4's mechanisms. When hardware extensions for virtualization became available on x86, running completely unmodified Linux on Fiasco appeared increasingly interesting. Full virtualization, also called faithful virtualization, is a technique to run entire operating systems with their user environment completely unmodified in a virtual machine. The environment provided by full virtualization is designed to be indistinguishable from real hardware with the exception of temporal behavior. We therefore enhanced the Fiasco kernel to provide a new kernel abstraction: virtual machines [15]. True to the microkernel philosophy, only the basic mechanism to drive the virtualization hardware is included in the kernel, an addition of a mere 500 lines of code. The complex implementations of virtual devices is left to a userland virtual machine monitor. Currently, this support infrastructure is still based on L^{4}Linux, but work on a native microkernel virtualization solution is well underway. We can already run unmodified Linux on top of Fiasco on certain ARM platforms by utilizing the ARM TrustZone hardware extension to implement full virtualization.

To maintain our goal of running real-time applications next to commodity software, we have to ensure that running full virtualization on Fiasco is not inhibitive to the real-time guarantees. A study we performed shows that real-time latencies are indeed affected by a virtual machine, but on a 2.2 GHz machine, the effect was bounded to 5.73 μs [17]. This is no problem for typical response-time requirements in the millisecond magnitude.

One of the greatest obstacles to the adoption of microkernels is their lack of a native execution environment for general purpose applications. This can be overcome with virtualization. We have experimented with paravirtualization and full virtualization and not only showed their usefulness on microkernels, but also successfully demonstrated the undiminished real-time capabilities of the resulting system. With full virtualization, such a system can even run unmodified Windows combined with native L4 real-time applications.

13.6 Conclusion

With the Fiasco real-time L4 microkernel and the real-time enabled managers for various system resource, the DROPS system described in the previous sections provides all the building blocks for writing real-time applications. Legacy support for running non-real-time software and real-time tasks side by side is provided by virtualized Linux. The Quality Assuring Scheduling and Quality-Rate-Monotonic Scheduling provide the mathematical foundation to handle overload situations.

13.6.1 Real-Time in Software Development

However, what is still missing is a comprehensive way to open this technology to software developers. All the elegant solutions and advancements in real-time systems research are of limited use, if they are not accessible to the engineers in need. To this end, a joint team of members from our research group and from the software technology group of our department developed the COMQUAD component architecture. This architecture allows to specify non-functional properties like quality levels and resource usage of a component implementation in the component quality modelling language (CQML$^+$). These properties are then used to derive contracts between components which are translated by the component runtime environment into resource reservations. Our real-time operating system and its resource managers enforce these reservations at runtime. This way, a component-based software development process was created, that supports adaptive real-time systems from specification all the way to the running system [9].

13.6.2 Current State and Outlook

Most of the software discussed here is available for download under the terms of the GNU General Public License. The resource managers are still prototypes, but the foundation of the system is usable. A demo CD is available as well [2]. We hope to spark a wider interest amongst operating system enthusiasts for design, implementation and deployment of real-time systems on everyday computers. With the knowledge we gained from the DROPS architecture, we are currently exploring new ground in embedded systems, where requirements concerning security, real-time and backward compatibility now appear combined in one handheld device, which is additionally limited by its energy budget. We expect to present more fascinating research results and we will always ensure that our systems are designed to be useful beyond mere academic purposes.

Acknowledgements We want to thank all our colleagues at TU Dresden who participated in the work presented. Most notably we thank Michael Hohmuth, Jean Wolter and Sebastian Schönberg for their work on Fiasco and the initial L^{4}Linux, Adam Lackorzynski for his constant maintainership of L^{4}Linux and his extensive work on system components, Norman Feske for his work on the DOpE window server, Jork Löser for the real-time network theory and infrastructure, Martin Pohlack and Lars Reuther for the real-time disk scheduler and Claude-Joachim Hamann for his work on scheduling theory. We furthermore thank Ronald Aigner, Robert Baumgartl, Martin Borriss, Frank Mehnert, Udo Steinberg, Michael Peter, Henning Schild and of course Jochen Liedtke. We want to extend our thanks to our friends in the L4 community: the L4 groups in Karlsruhe and in Sydney. Our work was supported by the DFG in SFB 358, by several grants from Intel and by the European Union in the ROBIN and OpenTC projects.

References

1. Atlas A, Bestavros A (1998) Statistical rate monotonic scheduling. In: Proceedings of the IEEE real-time systems symposium (RTSS), p 123
2. Demo CD (2006) URL http://demo.tudos.org/
3. Feske N, Härtig H (2003) Demonstration of DOpE – a window server for real-time and embedded systems. In: 24th IEEE real-time systems symposium (RTSS), Cancun, Mexico, pp 74–77
4. Hamann CJ, Löser J, Reuther L, Schönberg S, Wolter J, Härtig H (2001) Quality assuring scheduling – deploying stochastic behavior to improve resource utilization. In: 22nd IEEE real-time systems symposium (RTSS), London, UK
5. Hamann CJ, Roitzsch M, Reuther L, Wolter J, Härtig H (2007) Probabilistic admission control to govern real-time systems under overload. In: Proceedings of the 19th euromicro conference on real-time systems (ECRTS 07), Pisa, Italy, URL http://os.inf.tu-dresden.de/papers_ps/hamann07-qrms.pdf
6. Härtig H, Hohmuth M, Liedtke J, Schönberg S, Wolter J (1997) The performance of μ-kernel-based systems. In: Proceedings of the 16th ACM symposium on operating system principles (SOSP), Saint-Malo, France, pp 66–77
7. Härtig H, Baumgartl R, Borriss M, Hamann CJ, Hohmuth M, Mehnert F, Reuther L, Schönberg S, Wolter J (1998) DROPS: OS support for distributed multimedia applications. In: Proceedings of the eighth ACM SIGOPS European workshop, Sintra, Portugal

8. Härtig H, Hohmuth M, Wolter J (1998) Taming Linux. In: Proceedings of the 5th annual Australasian conference on parallel and real-time systems (PART '98), Adelaide, Australia
9. Härtig H, Zschaler S, Ronald MP, Aigner, Göbel S, Pohl C, Röttger S (2007) Enforceable component-based realtime contracts: Supporting realtime properties from software development to execution. Real-Time Syst 35(1):1–31
10. Hildebrand D (1992) An architectural overview of QNX. In: 1st USENIX workshop on microkernels and other kernel architectures, Seattle, WA, pp 113–126
11. Liedtke J, Härtig H, Hohmuth M (1997) OS-controlled cache predictability for real-time systems. In: Third IEEE real-time technology and applications symposium (RTAS), Montreal, Canada, pp 213–223
12. Lin KJ, Natarajan S, Liu JWS (1987) Imprecise results: Utilizing partial computations in real-time systems. In: Proceedings of the IEEE real-time system symposium (RTSS), San Jose, CA, pp 210–217
13. Loeser J, Härtig H (2004) Low-latency hard real-time communication over switched ethernet. In: Proceedings of the 16th euromicro conference on real-time systems (ECRTS), Catania, Italy, pp 13–22
14. Mehnert F, Hohmuth M, Härtig H (2002) Cost and benefit of separate address spaces in real-time operating systems. In: Proceedings of the 23rd IEEE real-time systems symposium (RTSS), Austin, Texas, pp 124–133
15. Peter M, Schild H, Lackorzynski A, Warg A (2009) Virtual machines jailed: Virtualization in systems with small trusted computing bases. In: VDTS '09: Proceedings of the 1st EuroSys workshop on virtualization technology for dependable systems, ACM, Nuremberg, Germany, pp 18–23, DOI http://doi.acm.org/10.1145/1518684.1518688
16. Reuther L, Pohlack M (2003) Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In: 24th IEEE real-time systems symposium (RTSS), Cancun, Mexico, pp 374–385
17. Schild H, Lackorzynski A, Warg A (2009) Faithful virtualization on a real-time operating system. In: Proceedings of the 11th real-time Linux workshop, Dresden, Germany
18. Yodaiken V, Barabanov M (1997) A real-time linux. In: Proceedings of the Linux applications development and deployment conference (USELINUX), The USENIX Association, Anaheim, CA

Chapter 14
A Real-Time Capable Virtualized Information and Communication Technology Infrastructure for Automotive Systems

S. Drössler, M. Eichhorn, S. Holzknecht, B. Müller-Rathgeber, H. Rauchfuss, M. Zwick, E. Biebl, K. Diepold, J. Eberspächer, A. Herkersdorf, W. Stechele, E. Steinbach, R. Freymann, K.-E. Steinberg, and H.-U. Michel

14.1 Introduction

Embedded information technology (IT) is the dominating enabler for advanced driver assistance systems and for the continued introduction of innovations in automotive products. Today's Car-IT architecture is characterized by a large number of dedicated function electronic control units (ECUs) with relatively low-performance microcontrollers and a heterogeneous set of low-capacity, automotive-specific communication buses. Over the past decades, the approach to add one ECU per new function has led to a complex, difficult to maintain and costly Car-IT infrastructure (Fig. 14.1).

The following chapters address the potentials that arise from adapting structures and technologies found in standard business IT environments towards automotive requirements. High-performance multi-core processors allow the aggregation of multiple conventional ECU functionalities into a single physical resource. Virtualization technologies (e.g. Hypervisors) guarantee the secure separation of multiple operating system domains running real-time and non real-time applications concurrently on the same multi-processor system. Such a standard, high-performance processing platform enables flexible post-sale upgrades and plug-&-play of multimedia infotainment applications as well as dynamic migration of tasks between different ECUs for fault-tolerance or CPU load balancing reasons. A multi-path meshed Gigabit-Ethernet network with real-time and class of service enhancements acts as a uniform "data highway" interconnecting ECUs, network I/Os, data storage repositories and advanced, high data rate sensor modules like Lidar

A. Herkersdorf, (✉) · S. Drössler · M. Eichhorn · S. Holzknecht · B. Müller-Rathgeber · H. Rauchfuss · M. Zwick · E. Biebl · K. Diepold · J. Eberspächer · W. Stechele · E. Steinbach
Technische Universität München, Arcisstr. 21, D-80333 München, Germany
e-mail: herkersdorf@tum.de

R. Freymann · K.-E. Steinberg · H.-U. Michel
BMW Forschung und Technik GmbH

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_14,

Fig. 14.1 Electronic/electric system of a car

or high definition video cameras. A high bit-rate, low latency communication infrastructure is prerequisite for investigating qualitative differences between raw and pre-processed sensor data transmission in sensor fusion applications. Equally important as processing and communication technologies is the design process followed during the development of Car-IT architectures. A platform-centric approach with standard component interfaces is instrumental for maximizing design reuse, shortening development times, achieving "right-first-time" designs, and cutting down development expenses. System-level modeling and analytical / simulation-based design space exploration tools support Car-IT architecture evaluation during early phases of design.

Our vision of future Car-IT architectures consists of a cluster of homogeneous powerful computing nodes with automotive-specific peripherals, interconnected with a mesh of Ethernet-MAC-based communication links. Such a computing cluster offers virtualized computation, communication, and storage resources, which are extendable and upgradable. Hard real-time tasks may be implemented in protected virtual domains, while soft real-time tasks may be implemented on best effort base, such exploiting the full computational power of the computing nodes.

Failure-resilient system behavior can be achieved with modest overhead of communication and computing resources. Breakdown of a communication link could be solved by re-routing of communication channels. Upon breakdown of a computing node, a task migration between CPUs could be triggered. While safety relevant functions may be implemented in hot standby mode, non-safety functions might be migrated in a degraded performance mode on other CPUs with some available computing time. Such a trade-off between failure resilience and implementation cost can be achieved.

Future Car-IT architectures might become an attractive target for virus attacks. In order to cope with these attacks, protection mechanisms are needed to prevent uncontrolled resource utilization between separated virtual Operating System domains. A middleware layer, called Hypervisor, will support such protection mechanisms, and simultaneously enable efficient utilization of computing resources.

The development of these future Car-IT architectures requires to solve some key research challenges, including concurrent transmission of hard real-time and soft real-time messages over common Ethernet-MAC-based communication links,

concurrent computation of hard real-time and soft real-time tasks on a cluster of multi-core processors, while efficiently solving conflicting memory and I/O accesses on shared resources, strategies for dynamic reconfiguration of communication links and computing resources under real-time constraints, balancing of redundant resources and failure resilience, as well as strategies for task migration and application-specific degradation.

In order to exploit the described potentials of adapting standard business IT technologies and design processes for Car-IT, a seamless migration path from today's status and tight cooperation between OEMs, Tier-1 and semiconductor suppliers are a must. This transition, driven by German and European car industry, could set a landmark in defining new industry standards how Car-IT will be architected in the future.

14.2 Conceptual/Architectural Overview

We designed a new, flexible and future-proof concept for Car-IT architectures based on the following four principles derived from best practice in standard business IT environments:

Centralization: Centralization means consolidating applications and their communication on a few central ECUs and a single communication network, respectively.

Due to their increasing interaction, automotive functions need to be colocated. This allows to reduce communication overhead and overall system complexity. Furthermore, the automotive industry can participate on the soaring processing performance of multi-core platforms being introduced for embedded systems. The additional processing performance can be used to consolidate more functions or enhance them with additional features.

As sensors/actuators components have to remain on their locations their tasks and the actual processing on the centralized ECUs are connected by a communication network providing sufficient bandwidth and the real-time guarantees.

Homogenization: Homogenization stands for using only a few standardized components and interfaces for the Car-IT architecture.

To ease development and to utilize mass market effects the Car-IT architecture should utilize all-purpose HW and SW components as much as possible. Components like ECUs, communication network parts, etc. should be re-used in different car and model variants.

Virtualization: Virtualization comprehends of abstracting and partitioning the underlying HW and SW resources for the functions using them.

By virtualization of processing resources, communication and services, the infrastructure is shared in a transparent and secured way between the different functions. This allows flexible allocation and utilization of it.

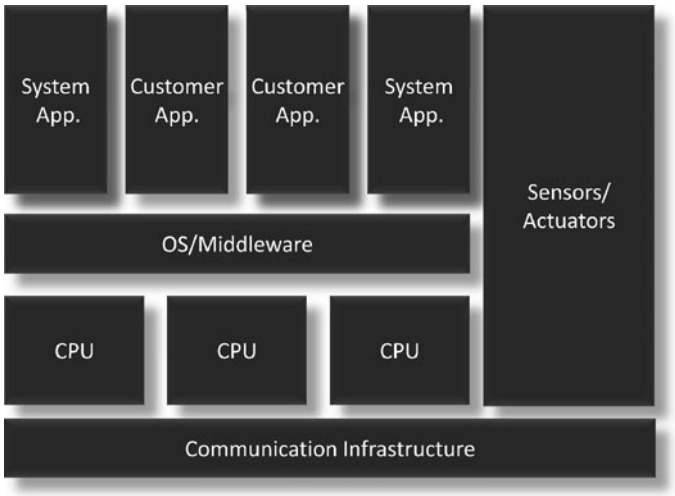


Fig. 14.2 Car-IT architecture

Relocatability: Functions are not longer fixed to a certain ECU or communication link, but can instead be (dynamically) relocated within the Car-IT architecture. Enabled by central and homogeneous ECUs, a single communication network and virtualization of resources, functions can be re-mapped easily. This allows more degrees of freedom during design-time, but also re-partitioning during run-time e.g, moving critical functions in case of a failure or switching to a more power-efficient application distribution on the ECUs.

The resulting Car-IT architecture is depicted conceptually in Fig. 14.2. The communication infrastructure is shared between the automotive applications and provides virtual channels. For details regarding this aspect see Sect. 14.4. The central ECUs are primarily providing an array of CPUs for the processing of the applications. The HW resources are partitioned and shared by virtualization (see Sect. 14.3). Communication infrastructure, ECUs and sensors/actuators are managed by a middleware (see Sect. 14.5) allocating and scheduling the logical resources of the virtualized and real-time capable infrastructure. Applications use this infrastructure; this is exemplary described for infotainment (see Sect. 14.6) and driving environment recognition (see Sect. 14.7).

14.3 Architectural Concepts for Automotive Control Units

Our concept and findings for automotive control units in the new Car-IT architecture are detailed in the followings paragraphs.

14.3.1 Overview

Future automotive control units will follow the general trend for embedded systems and contain multi-core processors. This higher performance is needed as automotive function from different car domains are more and more colocated due to their increasing interactions and have higher processing requirements. As multiple functions with different requirements regarding real-time, throughput or performance are consolidated on one shared platform, an effective set-up with low overhead has to be designed.

The Multiple Independent Levels of Security and Safety (MILS) architecture [2] which is utilized in aviation and virtualization techniques in data centers and desktop provide applicable solutions. On one platform several different domains can be executed concurrently. The physical resources are abstracted by logical resources. This provides a feature set required by the new Car-IT architecture. Domains can be migrated easily as they are only running on logical resources; virtualization has low overhead as no high level middleware is completely abstracting the underlying HW, but only separating and partitioning it; different domains are supported including legacy operating systems (OS).

With their increasing performance the number of ECUs will decrease and they will be integrated only at certain locations in an automotive structure. This results in no direct I/O connections to sensors/actuators by those ECUs i.e., the sensors/actuators are interconnected via communication network with the ECUs. The specific sensor/actuator tasks are performed on sensors/actuators components and the actual processing for the applications is performed on the ECUs. Therefore, the ECUs will not contain special or individual HW blocks, but rather a large array of homogeneous CPUs for performing processing and a high-bit data rate network interface.

14.3.2 Virtualization Set-Up

The set-up for virtualization of the automotive control unit resources is described in Fig. 14.3.

Central component is the hypervisor also called Virtual Machine Monitor (VMM). It has control over the whole HW resources and provides access to them for virtual machines. This includes resource provisioning, scheduling, partitioning and configuration. Examples for hypervisors are XEN [3] or KVM [15], but there are also dedicated hypervisors for automotive and embedded systems e.g., OpenSynergy Coqos [12] or Wind River hypervisor [34].

Virtual machines provide run-time environments for applications. A domain can contain anything from a general-purpose operating system with several applications down to a single task with only rudimentary kernel functions. This includes dynamically linked system and statically built system i.e., AUTOSAR [1]. There

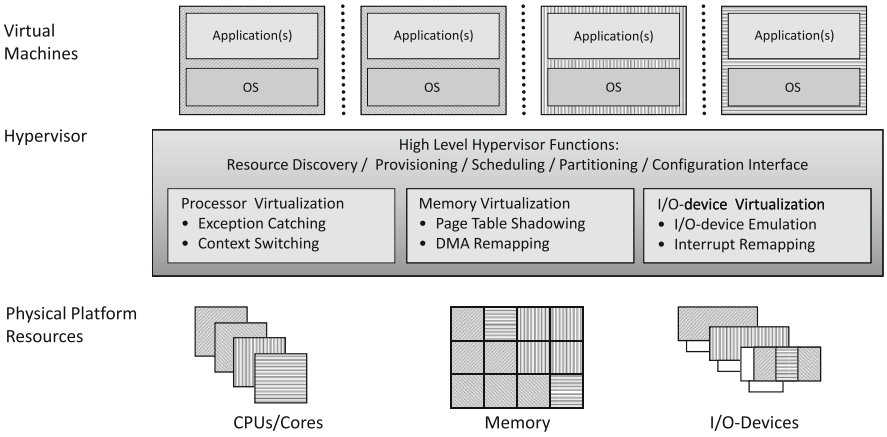


Fig. 14.3 Virtualization with hypervisor

can be special domains providing shared HW access (so called driver domains) or management interfaces to the hypervisor.

HW resources can be divided in three categories: processing units, memory and I/O(-devices).

- CPUs for processing are assigned to virtual machines. They can either be assigned exclusively or shared with other virtual domains in time slices. A domain can have more than one CPU. Virtualization has to catch CPU exceptions i.e., prevent the execution of not allowed instructions or conditions and switch (fast) between domains.
- Memory pages are mapped to different domains. Virtualization provides transparent access protection via page table shadowing and DMA remapping i.e., logical memory addresses are translated to the correct physical addresses without involvement of the domains.
- I/O-devices can be directly assigned to virtual domains or shared between them. Here, virtualization is responsible for the correct configuration of the devices and interrupt remapping.

All those actions are either performed entirely in SW by the hypervisor or are offloaded (partially) to HW with virtualization extensions e.g. for the Intel platform: VT-x for CPUs, Extended Page Tables (EPT) for memory and VT-d for chipset and I/O-devices [32].

14.3.3 Validation

We validated our concept for an automotive control unit with a prototype implementation (see Fig. 14.3). HW basis for this was an embedded board with an Intel

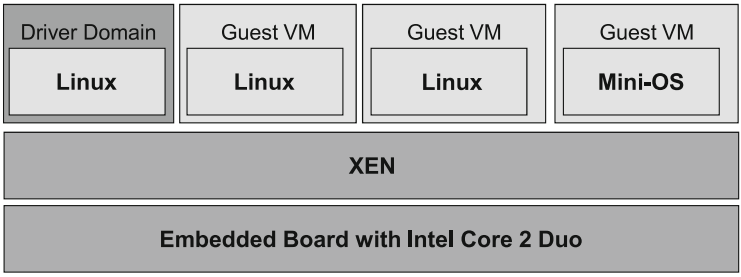


Fig. 14.4 Prototype set-up for an automotive control unit

Core 2 Duo CPU – representing both the performance grade of coming systems and a multicore set-up. An on-board Gigabit-Ethernet interface provided access to the communication network. We used XEN as VMM and Linux as driver domain. The driver domain also contained tasks for interaction and lifecycle management of guest domains providing applications run-time environments. A set of exemplary applications were deployed in three virtual machines. Two virtual machines were based on Linux containing an infotainment application with video-transcoding, a camera picture overlay processing and a soft real-time sensor data fusion. Those applications consisted each of several tasks running. Within a third VM containing Mini-OS – a simple embedded OS – a closed-loop control task was implemented receiving data from a sensor, processing it and sending control information back to an actuator over the network with a deadline and jitter of under 1 ms. All parts were stripped down to form a minimized system (Fig. 14.4).

In tests were able to provide the general throughput and performance in the Linux guest VMs for the infotainment and soft real-time applications and to meet the (hard) real-time constraints for the closed-loop control task.

This was only possible after carefully crafting the scheduling parameters. As the driver domain is in the critical path for network access to sensors/actuators, the deadline has to be practically split between this driver domain and the actual domain e.g., for the Mini-OS VM the period is 500 μs and the time slice 300 μs. The schedule of the driver domain inherits from the guest VM with the hardest requirements, here the Mini-OS domain. To reduce the requirements and the overhead in the scheduler, HW should dedicated directly to time or performance critical domains. This would take the driver domain out of the critical path, but it requires additional or self-virtualizing HW [27].

14.4 Next Generation Automotive Communication Network

Due to the trend towards more centralized processing in future automotive systems, the generation of information at the sensors and the processing will be more and more physically separated. To satisfy new requirements from *Homogenization,*

Virtualization, Centralization and Relocatability, a new approach for the car-internal communication network is needed to fulfill the following new challenges:

- Heterogeneous data, that means safety critical control data as well as convenience streaming data, have to be transmitted over the same cable while guaranteeing different service qualities.
- Centralisation leads to single-point-of-failures with increased probability of severe breakdowns.
- Communication paths change over car life time since new functions and devices are incorporated into existing systems.
- To control complexity and costs, well-known and proven IT-concepts and of-the-shelf components have to be adopted for use in the vehicle.

We developed a universal event based communication system using standard Ethernet physical layer and medium access technology in a star topology with cascaded switching nodes. We enhanced it with real-time capabilities and resilience features for the "mission-critical" automotive environment. *Real-time* means, for every planned transfer, an upper bound of the communication delay between source and sink can be specified. *Fault tolerant* means, with use of node disjoint paths between source and sink, data loss is avoided in case of a link or node failure or outage.

14.4.1 Requirements

Most of the transmitted messages can be classified into three categories: In category 1, messages are transmitted with cycle times of 100–200 ms, being typically control messages from body electronics with a size smaller than 8 Bytes. Messages which carry data from longitudinal dynamic control loops have to be transmitted every 20–50 ms. Whereas messages from lateral dynamic or chassis suspension control loops are transmitted with very short cycle times ranging from 2.5–10 ms. Through the physical behavior of a car [21] we know that the majority of the closed loops lead to timing requirements within these ranges. In the future, we expect increasing communication needs with 40–50 ms cycle times through new radar, lidar and camera based driving assistance systems and with 1–2.5 ms for new intelligent active suspensions and chassis with rising data rates.

So our system was designed to handle process data of 1 ms or 1 kHz control loops and validated with a realistic communication pattern extracted from an actual upper class car [24].

14.4.2 Real-Time Communication

The design is mostly using commercially available off-the-shelf hardware. It is based on the IEEE 802.3 Ethernet standard and does not require special end devices.

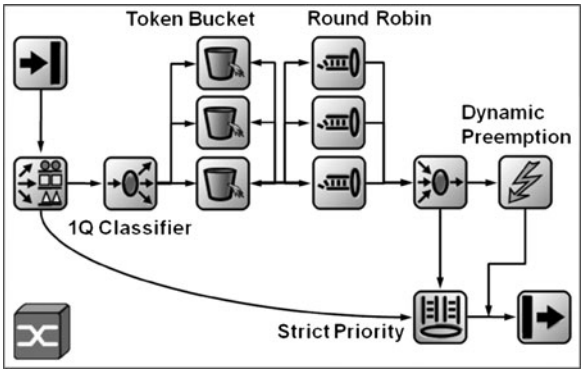


Fig. 14.5 Communication switch

The building blocks of this real-time enhanced Ethernet switch is shown in Fig. 14.5. With the help of a real-time capable scheduling system and partitioning, virtual channels are switched and an overlay network is built abstracting the logical path from the physical path.

We achieve this goal through separating the real-time connections into single flows with guaranteed resources through the informations in the 802.11Q Quality of Service Tag in the Ethernet Header. These virtual channels are, in a logical manner, on top of the network that itself offers connectionless access to the resources. Each flow has a defined path through the network, represented by a chain of *edges* and *nodes*. And each flow has guaranteed communication capabilities defined by a function $f(L_{P\,\max}, T_{A\,\min})$ where $L_{P\,\max}$ is the maximum packet size and $T_{A\,\min}$ the minimum packet inter-arrival time. With the help of this reservations and methods of queuing theory [11], a worst case transmission delay can be specified [25].

In an event driven system [16], the real-time calculus is complex compared to a time-triggered system, while allocation of the communication and processing resources is the other way round. Due to the limitations and hardware modifications mentioned above, the worst case transmission delay can be calculated analytically for a standardized Ethernet solution. Our simulations show, that this calculation delivers an upper bound for the transmission delay; the real delay is predominantly much lower. It is well-known, that in an event-triggered system, there is always an overestimation for the capacity of a communication network leading to a possible utilization $U << 1$ [17, 19]. The remaining capacity is consumed by the rising communication needs of audio, video and other information and communication services. In our approach, the traffic is separated from the control loop traffic with the help of a strict priority scheduling and a preemption mechanism to clear the channel if a high priority message arrives.

To avoid that the precalculated worst case transmission delay time bounds are violated when the flows exceed their agreed limits, a usage parameter control instance is needed for monitoring every flow to ensure the real-time behaviour. In our solution, we use a token bucket filter array centralized on the switches of the core network.

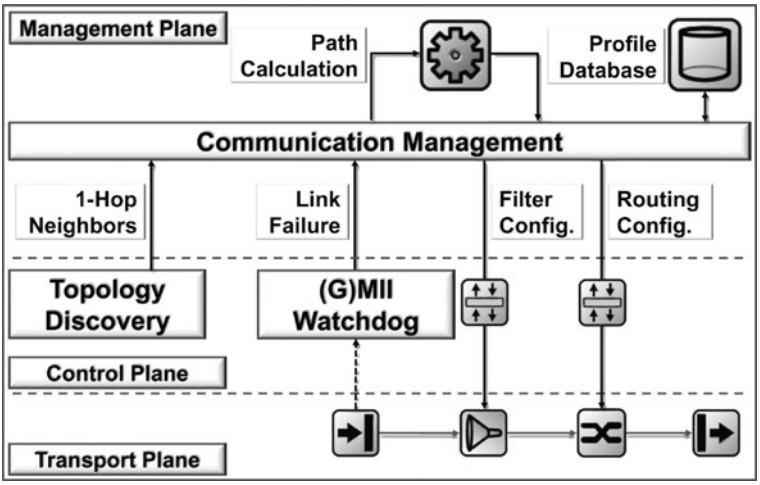


Fig. 14.6 Communication management

14.4.3 Fault Tolerance

In a centralized environment, the severity of failures raises because more and more functions are affected when a failure occurs. So a next generation architecture needs fault tolerance in its design, to reach comparable service quality. We provide this through a path-disjoint topology and dynamic rerouting for non-critical messages.

To achieve this, we build a distributed communication management system with different layers shown in Fig. 14.6. Every switch has a programmable switching fabric as transport plane. In the control plane of the switch, a small software based demon probes the connected nodes either with "hello" messages or with source address monitoring and generates a local view of the network topology. The local results of the topology discovery of all switches are transferred to a central management system which aggregates them to a global view of the whole topology. The management system calculates edge- and node-disjoint shortest paths for every critical data flow stored in the profile database and a shortest-path for all nodes. This information is splitted into several local configurations that the switches use to configure their switching fabric.

We separated the communication demand in three safety classes with different resilience mechanisms. The first class, safety critical data , are transferred over both paths in parallel (1+1 configuration), so even in case of errors no packet is lost. For the second class, critical data, the path is switched to the precalculated alternative path immediately after the error occurs. This results in more than one, but a limited number, of lost packets because the outage need to be detected and the changed topology information distributed to the switches. The management system then calculates new shortest paths for the remaining data flows and updates the configuration, leading to a short service outage of under 1 s for the third traffic

class, the non-critical systems. If the management system is not accessible, only the rerouting of non-critical data flows is affected and so this does not harm system safety. Apart from that, the management system itself should also be designed to be redundant.

14.4.4 Validation

To validate our approach, we use three ways: (i) an analytic verification, (ii) a simulation and (iii) a test setup [23]. To achieve this, the communication patters and traffic of a actual produced car was analyzed. This data serve as input values for the validation process. Our analytic approach verifies the real-time constraints and if the solution is feasible or any given boundaries are violated. A simulation with realistic traffic patterns predicts the occurring delay and jitter. The test setup helps to parameterize the models used in the simulation and in the analytical computation. The result is a worst case transmission time for every critical data flow and the delay distribution for every communication relation. The data can be used for certification as well as for estimating the Quality-of-Service for audio- and videostreams.

The plot in Fig. 14.7 shows the calculated worst case transmission time in nanoseconds for a vehicle with about 1,000 demand connections. With a Gigabit Ethernet (GE) network, the maximum delay time is at 160 μs. This means, that the approach is suitable to replace the state of the art vehicular communication architecture. Moreover, with its capacity reserves and functional capabilities, it

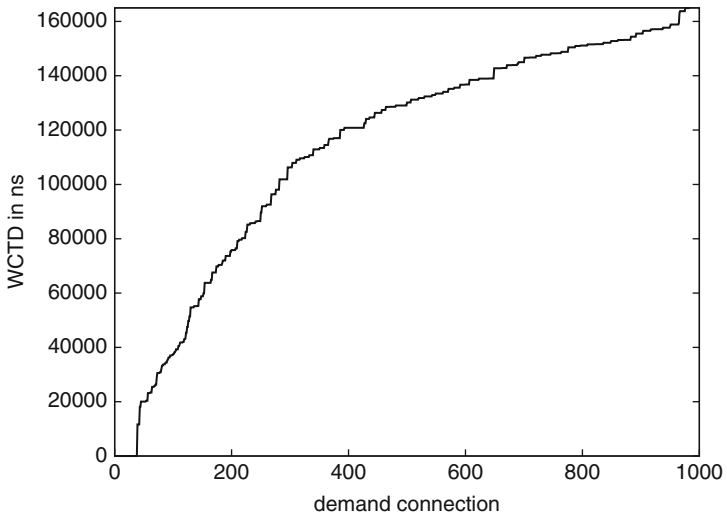


Fig. 14.7 Calculated worst case transmission time

is prepared to fulfill the requirements of next generation in-car communication systems.

14.5 Basic Software Architecture

The basic software uses virtualization capabilities provided by the underlying hardware and the communication network. It can be divided into a management layer, that has a *global view*, and a computation unit specific layer, that has a *local view* only. Additionally in this section, the influence of domain scheduling is addressed as well as fault-tolerance and dynamic reconfiguration issues.

Figure 14.8 shows a coarse block diagram of the layers and views of the basic software.

14.5.1 Basic Services

The software components of the *system layer* are located on every computation unit and have no knowledge about the global system. The computation unit configuration prepares a nominal local system state, informs the lifecycle manager about which

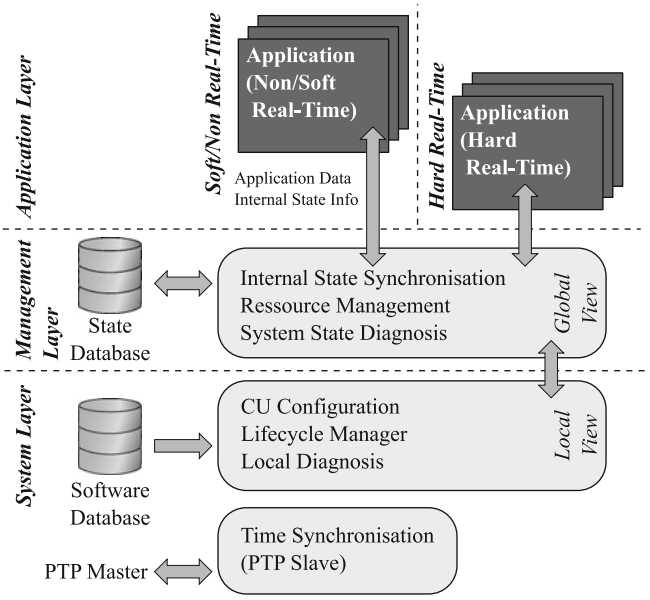


Fig. 14.8 Block diagram of basic software

domains should be in which sate and sets specific metadata like scheduling parameters, assigned memory, maximum allowed packet rate, etc. A *local diagnosis* subsystem is implemented to observe the behavior of all virtual domains running on top of the hypervisor. If an abnormal behavior, e.g. extraordinary cpu or memory consumption, is detected it informs the global state diagnosis and may also initiate minor local healing action like domain restart in case of non real-time domains. The *lifecycle manager* is an executive instance, which is able to start, stop, pause and resume virtual domains and applications running inside according to the input form the computation unit configuration. It is also responsible to execute a domain migration to another computation unit if requested by the global resource manager. If an execution binary of a domain or an application is not locally available, it is downloaded from a distributed software database.

In the *management layer*, all informations from the distributed computation units are gathered. The *global system manager* is located on at least two computation units for fault-tolerance reasons. There is only one manager active at the same time. If it fails, a new manager is voted among the others to get active. The *global system diagnosis* detects and classifies errors. Using the information from the resource manager, a recovery strategy is planned. It then informs the local computation unit managers of their new configuration. In Sect. 14.5.4, allocation strategies for non, soft and hard real-time domains are presented.

Applications are running in virtual domains. There is an API for exchanging internal state information, necessary for providing stateful dynamic reconfiguration.

14.5.2 Providing Fault-Tolerance

Real-time applications have a tight timing bound. We therefore use application specific fault-tolerance mechanisms for real-time applications, and the services provided by the basic software otherwise. The safety critical real-time applications are implemented as a triple modular redundancy (TMR) system with cloned domains (co-domains). If one domain repeatedly fails due to timing or value errors, it is (1) prohibited to furtherly produce outputs (error/fault masking). In this state there are still two co-domains active and operation can continue – as long as both results are equal. In the meantime, the following steps are taken: (2) the global diagnosis system is informed about the error, (3) a computation unit with enough spare resources is identified to take over the co-domain, (4) the configuration is sent to the chosen computation unit, (5) the local mechanisms of the basic software bring up the co-domain in passive state, (6) the co-domain is synchronized to the active ones, (7) the co-domain is set to active. At this point the real-time application has regained full health status.

If in step (3) no computation unit with enough resources can be found, non and soft real-time applications can be gracefully degraded. For every application, we added meta information that contain the supported degradation modes and their according processor, memory,... demand. We follow an approach to regain full

health status for the real-time applications as fast as possible: We set as much of the non and soft real-time applications to the most degraded mode as necessary to allocate the real-time domain. Then, we try to find an allocation where the degraded applications (see e.g. Sect. 14.6.4) may be upgraded to full service or at least to a less degraded mode.

14.5.3 Architecture Specific Real-Time Issues

There are some major architecture specific real-time issues that must be addressed. In the following we focus on the two most important issues: data consistency and real-time scheduling.

14.5.3.1 Providing Synchronous Data

In a distributed system, data consistency must be provided, esp. when different computation units are used for a system-wide control application. For example, the ESS (electronic stability system) reads wheelspeed and inertial sensors, calculates if some action on the brakes is required and sends the nominal actuator position to the four brakes; if these signals are not synchronized, the car may even get more unstable instead of being stabilized. Future driver assistant systems will be more sophisticated and a distributed control loop is conceivable. For the ESS that might be four different control loops calculating the actuator values for the brakes.

Although we have our system synchronized via PTP, slight clock deviations, different scheduling decisions and different communication delay require an additional effort to guarantee data consistency. We want to avoid a communication intensive agreement protocol. Instead, a technique introduced by Poledna et. al. [26] called timed messages is used. Every message, sensor value or actuator value is annotated by a validity time and not an acquisition or origination time. The validity time depends on network delay, domain scheduling strategy and clock deviation.

14.5.3.2 Influence of Domain Scheduling on Packet Processing

Using virtualization technologies in systems guaranteeing hard or firm real-time constraints is challenging. On one hand there are safety relevant real-time applications, like control applications, with short execution times and tight deadlines but generally low throughput demands. On the other hand soft or non real-time applications, like video or telematic services, with high throughput demands but less restrictive deadlines. In the present system, both domains are addressed.

Since hard real-time applications are activated by incoming packets, one has to take a deeper look at the virtualized network processing of the computation units. It turned out, that it is best to use a hybrid approach. Hardware support for virtual

network interfaces (vNIC) shall be used for all non and soft real-time domains. However, hard real-time domains communicate through the driver domain. With this configuration, non or soft real-time packet processing does not interfere with hard real-time packet processing. The other advantage is, that hard real-time domains do not need to implement an Ethernet or even a TCP/IP stack but use a lightweight standardized API[1].

Using Ethernet, the driver domain is invoked three times for one control cycle – assuming only one incoming sensor value and one outgoing value for the actuator: first by the incoming packet, second by the outgoing packet and third by an acknowledge signal from the NIC. Since the control applications have different periods and different deadlines, what deadline should be assigned to the driver domain in case of deadline scheduling? In case of using a fixed priority scheduling, what priority to assign?

For fixed priority scheduling, the driver domain gets the highest priority. In case of an incoming packet, there's no knowledge about its (VLAN-)priority – which is set according to the importance of the target domain – before it has been processed. Because the driver domain handles incoming as well as outgoing packets at the same time, there is no possibility of using priority inheritance strategies.

The reason for using deadline scheduling, i.e. earliest deadline first (EDF) scheduling in this project, rather than fixed priority scheduling is that its potential processor utilization is higher. Efficient feasibility testing depends on a proper model. Figure 14.9 shows three demand bound functions $dbf_x^{\mathrm{EDF}}(T,t)$ resulting of different modeling variants. The demand bound functions contain both invocations of the driver domain and the invocation of the real-timedomain. e_i and e_N are the worst case execution times for the real-timedomain and the driver domain respectively. d_i and d_N are their deadlines and p_i gives the minimum interarrival time of a packet for real-timedomain i, that we call the period of the real-timedomain i. The phase Φ_N is the shortest possible time between two invocations of the driver domain: $\Phi_N = \min(d_i, p_i - (d_N + d_i))$.

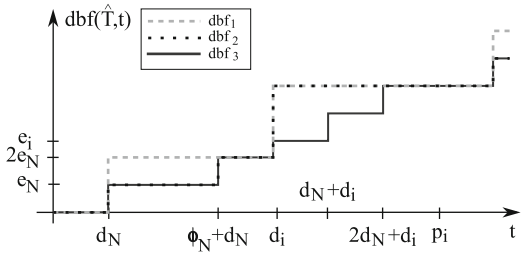


Fig. 14.9 Demand bound functions for one real-time application including packet processing

[1] In analogy to AUTOSAR [1], the real-time domains are equivalent to AUTOSAR Software Components (SW-C). The API is the connection to the Virtual Functional Bus (VFB), or the AUTOSAR Runtime Environment (RTE) respectively.

Using the density condition [20, 30], i.e. accumulating execution times e_N for both invocations of the driver domain and for the real-timedomain e_i without phases, results in dbf_1. An improvement can be achieved when Φ_N is taken into account [10] resulting in dbf_2. Best result gives dbf_3 which is based on the generalized multiframe task model [5, 6, 22, 31] and the recurring real-timetask model [4, 7].

To check feasibility of a set of real-timeapplication τ, $dbf^{\text{EDF}}(T_i, t)$ for all real-time applications must be accumulated. The schedule is feasible, if at any time $t \geq dbf_{\text{EDF}}(\tau, t)$ holds, with $dbf_{\text{EDF}}(\tau, t) = \sum_{\forall i} dbf^{\text{EDF}}(T_i, t)$.

14.5.4 Application Distribution

The before mentioned homogeneity of the computer cluster not only minimizes costs due to a high amount of common parts, but also enables the *dynamic migration of tasks*. If tasks can quickly be migrated from any ECU to another, then free computational power of any ECU can be misused as a kind of 'generic spare ECU' for ECUs that exhibit a failure. Compared to architectures that demand spare hardware for every critical component, dynamic task migration appears to be a promising way to reduce costs for spare hardware significantly.

Besides *network traffic* and *power consumption* [18], an optimization criterion to migrate soft-realtime and non-realtime tasks is the maximization of *application throughput performance* by placing applications on cores with respect to *cache performance*. The remaining of this section discusses this optimization criterion.

To motivate the migration of tasks with respect to an optimization of overall cache performance, we present Fig. 14.10a that shows the degradation of L2 cache

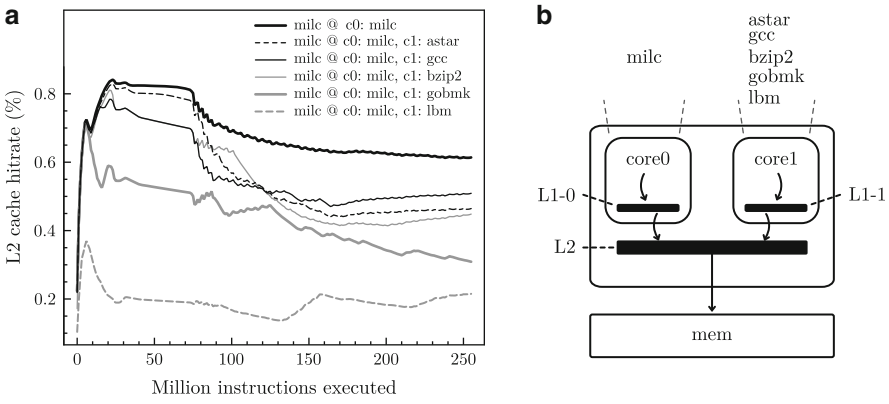


Fig. 14.10 (**a**) Cache contention introduced to the *milc* SPEC benchmark when co-scheduled with SPEC benchmarks *astar*, *gcc*, *bzip2*, *gobmk* and *lbm* respectively. (**b**) Simulation setup: Application *milc* on *core 0* is co-scheduled with either application *astar, ...* on *core 1* in a separate run

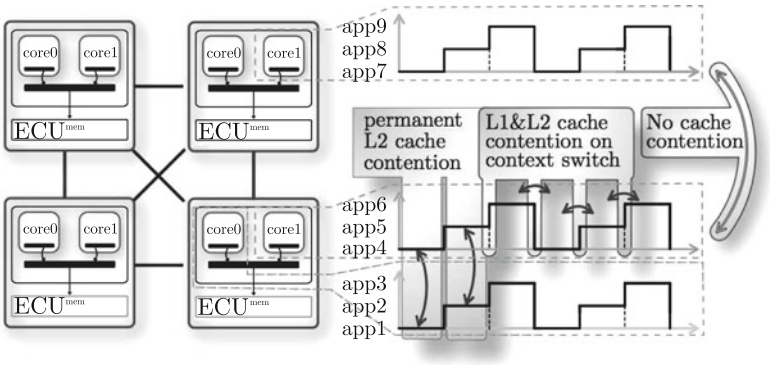


Fig. 14.11 Cache contention introduced when co-scheduling applications

performance for the *milc* SPEC benchmark due to *cache contention* when co-scheduled with SPEC benchmarks *astar*, *gcc*, *bzip2*, *gobmk* and *lbm* respectively in a multi-core scenario according to Fig. 14.10b, as it has been simulated with MCCCSim (Multi Core Cache Contention Simulator) [37]. [2]

One can easily see that the L2 cache hitrate degradation of the *milc* SPEC benchmark is maximized when co-scheduling *milc* with *lbm* and minimized when co-scheduling *milc* with *astar* and *gcc* respectively. As a consequence, an appropriate choice of co-scheduled applications improves overall execution performance significantly, as it has also been reported by [9].

Figure 14.11 shows how cache contention is depending on the placement of applications in the proposed architecture: Applications that share (timesliced) the same core (like *app4*, *app5* and *app6* in Fig. 14.11), also share the same *L1 and L2* cache. Therefore, any application's cached data might be replaced by any other application on the same core. However, the replacement of cached data only takes effect *at most once within a context switch*.

Applications that reside on *different cores*, but on the same ECU and are co-scheduled (such as app1 ↔ app4, app2 ↔ app5 or app3 ↔ app6 in Fig. 14.11) share only the L2 cache, but not the L1 cache. However, although only the L2 cache is shared in this case, our simulation results depicted in Fig. 14.12 show that performance degradation due to cache contention is generally worse, as the L2 cache is shared *permanently* during the whole timeslice execution. Therefore, such

[2]For a better understanding of our optimization concept, we assume an ECU architecture as depicted in Fig. 14.10b: An ECU consists of several cores that all share a common L2 (level 2) cache. Each core, however, has its own private L1 (level 1) cache that cannot be accessed by any other core. Further, each core can only execute one single thread – a constraint that can be omitted later on by a simple adaption of the proposed optimization scheme.

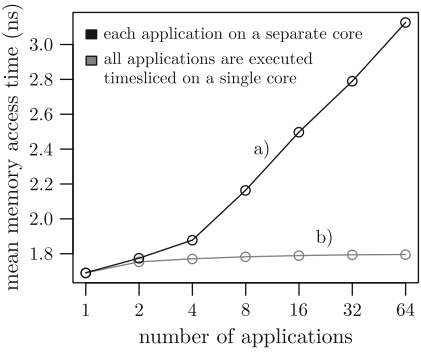


Fig. 14.12 Simulated accesstime when co-scheduling 1 ... 64 applications. In (**a**), each application is executed on a separate core, so there are up to 64 cores for up to 64 applications. In (**b**), all 1 ... 64 applications are executed on a single core using round robin scheduling. The mean memory accesstimes shown are averaged values gathered by the MCCCSim simulator executing 10 SPEC benchmarks. Although we simulated with very short timeslices of 2 ms resulting in many context switches, the higher accesstime in (**a**) denotes greater effect of permanent L2 cache contention than timesliced L1 *and* L2 cache sharing

co-scheduled applications (same timeslice, but different core) interfere at a much higher degree than applications that reside on the same core. [3]

To optimize application distribution for cache contention, a good method has to be provided to *predict* cache contention. This method should be able to take some program attributes as input and deliver an estimation of cache contention as a result. This way, optimization methods, such as the particle swarm optimization (PSO) [14] for example, can be applied to find optimal application distributions, as it has similarly been done in [28, 36] for job scheduling.

In [38], we proposed *setvectors* as a high performance method to predict cache behavior of *standalone* applications for phase classification [29]. In the following, we will show by an example, that this method is also a suitable approach to predict cache contention introduced by the *combination* of tasks.

In Fig. 14.13a we show simulation results of a typical prediction scenario for real programs: We tested, if it would be better to co-schedule application *Safari* with application *gzip* or with application *gunzip*.

To predict the optimal co-schedule for a given program interval, we calculate the predictors $p_{\text{Safari, gzip}}$ and $p_{\text{Safari, gunzip}}$ according to or proposal in [38], subtract $p_{\text{Safari, gunzip}}$ from $p_{\text{Safari, gzip}}$ and extract the sign of the result by the *signum* operation. Averaging the results over several intervals, we determined a mean value

[3] Certainly, if a multicore processor also includes SMT (simultaneous multi threaded; named *hyper-threaded* by Intel) cores, then cache contention is maximized on applications that run on such an SMT core at the same time, as those applications then share the L1 and L2 cache permanently for the duration of their timeslice.

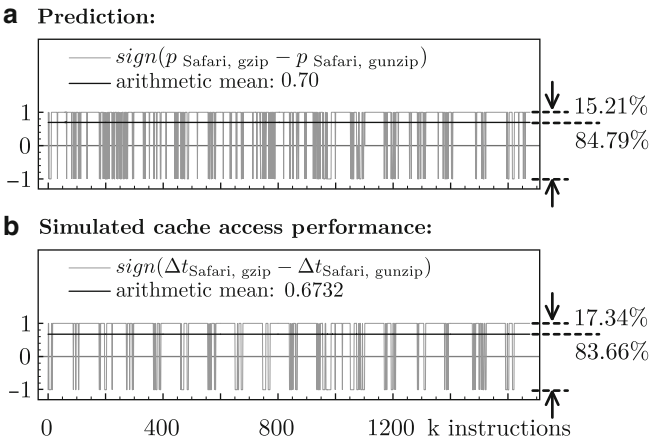


Fig. 14.13 (**a**) Prediction of *Safari* ↔ *gzip* and *Safari* ↔ *gunzip* co-scheduling. (**b**) Verification of the effect of right co-scheduling

of about 0.70. This means that the setvector method predicts a co-schedule of *Safari* and *gunzip* to be preferable over a co-schedule of *Safari* and *gunzip* for 84.79% of the accounted intervals, as it can be seen from Fig. 14.13a.

Figure 14.13b verifies this prediction: If $\Delta t_{\text{Safari, gzip}}$ summarizes the additional accesstime introduced to both *Safari* and *gzip* due to cache contention, when co-scheduled, and $\Delta t_{\text{Safari, gunzip}}$ the additional accesstime introduced to *Safari* and *gunzip*, then $\Delta t_{\text{Safari, gzip}} - \Delta t_{\text{Safari, gunzip}}$ is *positive*, if the co-schedule *Safari* ↔ *gunzip* would be a better deal than the co-schedule *Safari* ↔ *gzip* with respect to cache contention. Equivalently, if $\Delta t_{\text{Safari, gzip}} - \Delta t_{\text{Safari, gunzip}}$ is *negative*, then the co-schedule *Safari* ↔ *gzip* is preferable.

The arithmetic mean of $sign(\Delta t_{\text{Safari, gzip}} - \Delta t_{\text{Safari, gunzip}})$ for a set of intervals is therefore a measure for the amount of time each co-schedule is preferable. Figure 14.13b shows that the co-schedule *Safari* ↔ *gunzip* is preferable over the other co-schedule for 83.66% of the accounted intervals. The difference of the means pictured in Fig. 14.13a and b shows a prediction error of $84.79\% - 83.66\% < 1.2\%$, i.e., for the given intervals, the simulated accesstime introduced by cache contention differs from the prediction by only 1.2%.

For the given set of intervals, Fig. 14.14 shows a basic error quantification on a *per interval* basis: For 226 out of 1,677 intervals, the prediction from Fig. 14.13a coincides with the simulated accesstimes depicted in Fig. 14.13b.

In this chapter, we presented a method that optimizes task deployment in our IT_Motive car-IT-Architecture for soft-realtime applications and showed its effectiveness by a small example. In the next chapter, we introduce a new multimedia architecture that can be integrated in our IT_Motive car-IT-architecture.

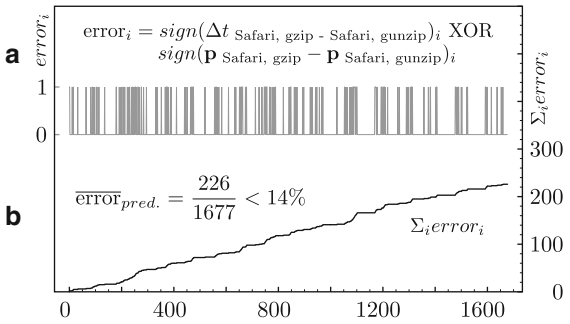


Fig. 14.14 Basic error quantification: In (**a**), the error is shown on a per-interval basis, i.e. for each interval, error_i denotes whether the prediction matches the relation of the simulated accesstimes for co-scheduling. A value of "0" declares a true prediction, "1" declares a false prediction. The mean prediction error ($\overline{\text{error}}_{\text{pred}}$) is calculated as *number of wrong predictions* divided by the *total number of predictions* and is about 14%. In (**b**), the number of wrong predictions vs. the total number of predictions shows a nearly linear characteristic

14.6 In-car Infotainment/Multimedia

Innovations in the area of in-vehicle infotainment/multimedia are typically driven by developments in the home entertainment and consumer electronics industry. Customers quickly get used to novel gadgets and applications and want to use them also in their cars. Hence, infotainment and multimedia functionalities become increasingly common in the automotive environment and keeping up with the customer demand generates many challenges. With our proposed architecture we are able to address many of these. Our concept offers the same four basic properties (*Homogenization, Virtualization, Centralization and Relocatability*) in the infotainment and multimedia domain.

14.6.1 Overview

Our infotainment/multimedia architecture builds on top of our car IT-infrastructure, uses its hardware components, the error-robust communication network and in general the mechanisms it provides. Also, it extends the hardware components by input and output devices. Our approach leads to increased flexibility, expandability and graceful degradation in the context of in-car infotainment and multimedia. The proposed architecture is divided in three main blocks (Fig. 14.15).

The concept for the Human Machine Interface (HMI) (left side of Fig. 14.15 and Sect. 14.6.2) is exclusively based on web technologies. It consists of several displays, one processing unit for the HMI and several controllers. The right-hand side of Fig. 14.15, described in detail in Sect. 14.6.3, is a Service-Oriented Architecture (SOA) for the integration of nomadic devices and car internal applications

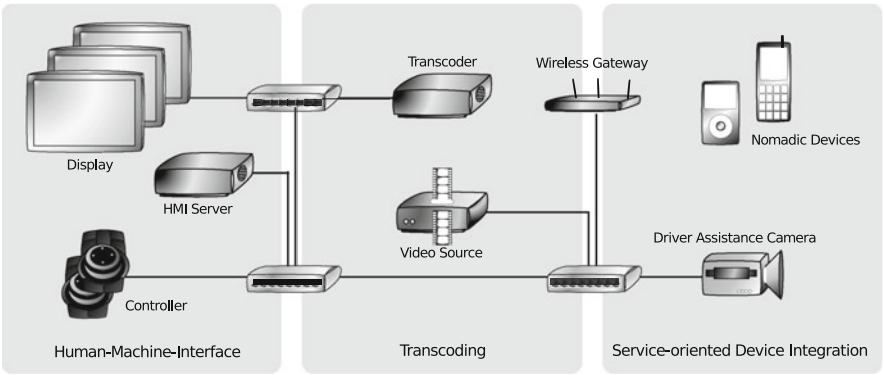


Fig. 14.15 Overview of the proposed infotainment/multimedia architecture: The three main building blocks are shown by means of an example network/setup

on Electronic Control Units (ECUs). In the figure some examples for SOA-based nomadic devices (wired or wireless) and a placeholder for SOA-based internal functions, e.g., a driver assistance camera are shown. In Sect. 14.6.4, the center block of Fig. 14.15 is explained which addresses the lifecycle mismatch between audio/video codecs and the car. In the middle of the figure a processing unit for video/audio transcoding and, exemplarily, one video source are shown.

14.6.2 Human Machine Interface

For HMIs, it is state of the art to use a combination of hardware and highly specialized software (think for example of a ticket vending machine). These systems are difficult to extend by a new interface or function. However, in the home or car environment, users want to integrate additional functions. Our approach offers the flexibility to adaptively change the number of output units, input units and even functions.

Web technologies form the basis of our architecture. All necessary data for the generation of the HMI are collected on a web server and the HMI is displayed in a web browser. This allows the *centralized* management of different HMIs, like simplified versions for the elderly or a more sophisticated version with more functionalities for expert users. Another advantage is the easy provisioning of different "look and feels". For the transmission of the generated HMI data, standard web protocols are used, e.g., HTTP and HTML. Thus we reduce the data transfer rate compared to a transmission of the HMI as a whole. We only assume that a web browser is installed at the user interface. By using a web browser to render our HMI, we can also access it from every PC. Also, we benefit from well tested software and configurations that are widely deployed in the Internet.

The input units can be, for instance, mouse, touchscreen or specialized controllers. If these are connected locally at the display, the input is directly processed by the browser. If these are remotely connected, they are handled as a service/ device (see Sect. 14.6.3), whose input is transmitted to the webserver and forwarded to the corresponding displays.

An additional advantage of using web technologies is the possibility of using SOA in our HMI. Functionality integration can, e.g., be achieved using web services.

14.6.3 Service-Oriented Device Integration

Todays upper class vehicles have up to 70 distributed ECUs, which offer a variety of different functionalities. The communication between all these autonomous units has become a challenge, as traditional automotive communication networks (e.g. CAN, MOST) are reaching their limits. This has initiated a gradual paradigm shift, replacing the collection of domain-specific network technologies by a single in-vehicle network, as proposed in our car IT-infrastructure. Flexibility is a key requirement in this context. For the infotainment/multimedia domain it is therefore necessary to investigate the use of distributed and dynamic control systems in an in-vehicle IP network *(Homogenization)*. For this purpose a SOA is adopted. In a web technology-based HMI, Webservices (WS) are well suited to achieve service integration and extensibility. However, Webservices offer no possibility for announcement or discovery of the services. Hence, in our SOA a combination of Universal Plug and Play (UPnP) and WS is used, since UPnP offers the missing discovery and announcement mechanisms.

In our approach (Fig. 14.16), a gateway, which takes the information of the UPnP service and generates a WS adaptation, is utilized. Additional SOA techniques can be adapted in the same way. The information of the WS for finding, integrating and connecting the services is stored in a database. Our HMI accesses this database and offers services to the user in a convenient way.

With SOA we achieve a *virtualisation* layer for an easy integration of devices and services. This layer is platform independent and hence, we can offer these services to everyone in the car.

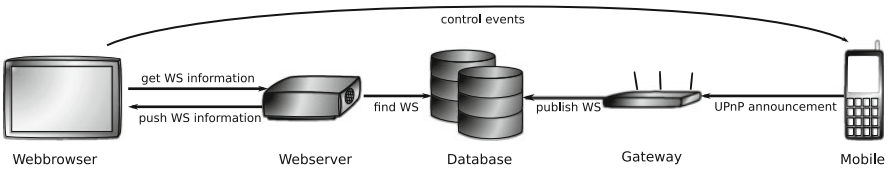


Fig. 14.16 Example for service integration in the HMI: A calendar service of a cellular phone offers the UPnP interface and is integrated via a gateway as a Webservice in the HMI

14.6.4 Transcoding

In order to be able to transfer multiple high quality audio and video streams in the car without exceeding the transmission capacity, efficient data compression is required. The number of available codecs for both video and audio is steadily increasing. Introducing compressed audio and video requires the integration of the corresponding codecs in the sender and/or receiver(s). Since, each compression standard requires a different codec, this implies extra costs for the car manufacturer. When comparing the compression standards' short life cycle of about 2-5 years with the car life cycle of around 10 years, it becomes obvious that frequent updating of the compression formats is inevitable. The frequent codec updates in all sources and sinks imply high costs and are therefore not desirable. The same issue occurs with an embedded hardware solution for every source and sink.

An alternative solution is to use one particular codec as the output format of every source. Hence, we can use a single standard inside the car that is valid throughout the entire life cycle. The drawback of this approach, however, is that transcoding units in every source, deviating from the standard codec, would be required. Hence, in our approach the transcoding is performed by a central transcoding unit that is easy to maintain and update.

In [8], we propose a transcoding unit where all compression standards are integrated and are regularly updated. If necessary, the compressed audio and video streams are processed by this unit. They are converted into the reference compression format and thus *homogenized*. Hence, the codec upgrading effort is reduced to only one unit and the production cost is decreased as well.

Another advantage of the centralized approach is that there is no transcoding complexity in the source. All transcoding operations are performed by one node in our IT-infrastructure. Although this leads to a single point of failure, integrating a corresponding module in every source (e.g in a DVD player) would be very expensive to implement. To deal with the fact of having a single point of failure, the proposed IT-Motive in-car architecture (see in Sect. 14.5) includes a reconfiguration mechanism. In case of a node failure, the transcoder software is *relocated* from the corresponding processing unit to another one. In case of insufficient resources on the new processing unit, a *graceful degradation* is possible. Rate and complexity adaptation is achieved by reconfiguration (compare Sect. 14.5.2) of the transcoding parameters. The relationship between the rate, complexity and quality for transcoding of video from MPEG-2 to MPEG-4 is illustrated in Fig. 14.17. The transmission rate is typically given in bits per second, the complexity in milliseconds per frame and as quality the unitless metric peak signal-to-noise ratio (PSNR) is used. Every surface represents one of three possible transcoding levels. In the first one, the original discrete cosine transform (DCT) coefficients and motion vectors (MV) are fully reused, in the second, only MVs are retained and in the third, full re-encoding is performed (for more details see [8]). As can be seen from Fig. 14.17 it is possible to adapt the used resources to the requirements in more than one way, either by adjusting the parameters or by switching transcoding levels.

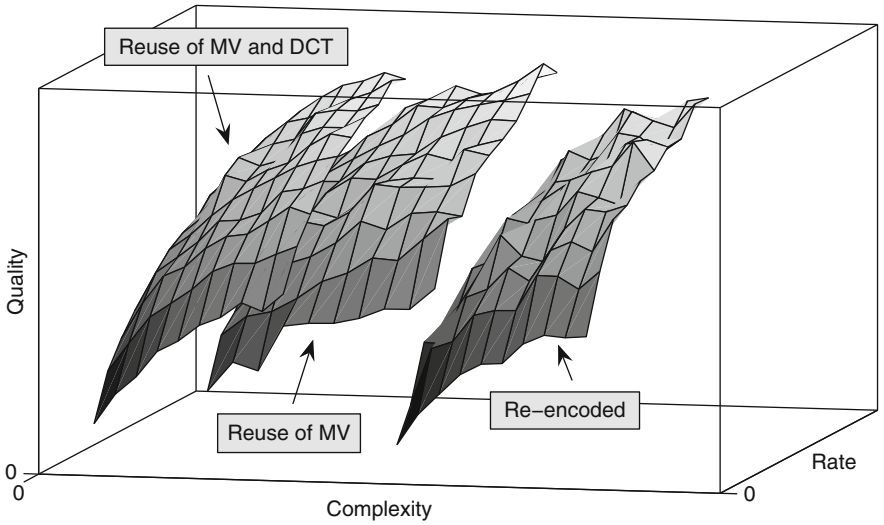


Fig. 14.17 Qualitative view for three different transcoding levels. Transcoding is performed for video from MPEG-2 to MPEG-4

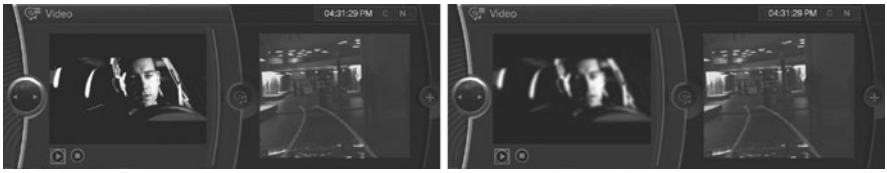


Fig. 14.18 Screenshots of the HMI with a transcoded video overlay (BMW design): The left figure shows the normal transcoding and the figure on the right shows the video after transcoding in the degradation mode

In Fig. 14.18 screenshots of the HMI before and after a node failure are shown. The left screenshot displays the transcoded video together with a realtime application (driving assistance application). The right one displays the same applications, but after a relocation. In this case the transcoding application is moved to the processing unit, where the assistance function is running. We can clearly see a decreased quality for the video, but no negative influence on the realtime application. This means that we can still use the function in its degraded version, as compared to the state of the art error handling, which would imply the failure of the function. The possibility of *relocation* further justifies implementing our transcoder module in software and not in hardware.

Another solution would be to decode every format at the display. However, this would require to implement the decoder scheme in software due to the updating requirements for every new codec. This in turn would increase the costs of the displays significantly compared to a solution that provides just the reference standard decoder implemented in hardware.

14.7 Sensor Architecture for Driving Environment Recognition

Current car sensor architectures have a fixed association of software and hardware, which means most of the signal processing is done within the used smart sensors and their dedicated embedded systems. Changing any step of signal processing to adapt to different situations is not possible. The processed sensor data is only accessible for interconnected ECUs. So the sensors can mostly be used for just one single function in one specific situation under optimum conditions (no malfunctions). In a today's sensor system, e.g. Bosch's Adaptive Cruise Control[35], there are five different ECUs involved and interconnected by CAN. In future, more and more sensors for driving environment recognition will be used due to safety and comfort enhancements. This leads to a highly complex integrated network, where our architecture proposal simplifies this structure and enables new features in signal processing and fault tolerance.

To meet the requirements of future driver assistance systems, the following features have to be realised:

- Centralized processing of raw data
- Multi-use of sensors for different functions (cruise control, park distance control, etc.)
- Possibility of data fusion with a wide variety of sensors
- Dynamic sensor reconfiguration for different situations (e.g. variable acquisition areas)
- Sensor and assistance system fault management.

These characteristics are achieved by building the sensor architecture on top of the proposed car IT-infrastructure. These features are described in the following subsections.

14.7.1 Raw Data Processing

In current sensor architectures the captured data is mostly processed within the sensor itself at microcontrollers, particularly at radar or lidar sensing. Working on raw data in combination with strong central ECUs allows advanced signal processing. Centralized CPU power is relatively cheap, so the envelope can be pushed in processing raw data.

Because of the possibility to replace applications (e.g. a new driver assistance function) an access to raw sensor data is needed for not having any quality loss caused by dropping parts of the signal in a previous processing step. In addition to that, some types of data fusion and the option of multi-use of sensors needs the central availability of raw data to be free of limitations in signal processing.

14.7.2 Multi-use of Sensors

The availability of environment data for different functions, acquired from one sensor, is aspired for different reasons. Up to now, nearly all customer functions have their own sensors, e.g. a radar sensor for ACC, ultrasound sensors for parking aids, video cameras for lane departure warning, etc. The need of data fusion for robust environment recognition leads to more and more sensors, e.g. fusion of long range and short range radar for ACC with traffic jam assist. In this examplary configuration, two types of short range surroundings acquisition are used: radar and ultrasound.

Making the surroundings data centrally available enables the operation of advanced algorithms using data fusion for every application that can use it at a better performance (additional information from extra sensors), compared to current car-IT-architectures. In our case, the short range radar can be replaced by (adapted) ultrasound sensors. A further alternative is the possibility to offer more customer functions which use environment sensor data. Since the sensors are already available for other functions, this could be done at low costs.

14.7.3 Data Fusion

Data fusion of any sensors will be an essential feature of prospective driving environment recognition systems. This can be done best by central accumulation and fusing of data. Due to this central data processing each step of the signal chain can be made available not only to any other application but also to data fusion units connected to further sensors. Thus, any combinations of fusion levels from early until late data fusion are feasible.

14.7.4 Dynamic Reconfiguration

Depending on different driving situations, like highway or rather city rides, reconfiguring the environment sensing expands the area of application. E.g. an acquisition angle of $\pm 4°$ suffices for Adaptive Cruise Control on highways, whereas a $\pm 25°$ aperture angle is necessary for stop and go assists used in urban areas[33]. Switching over between these two states for one sensor which has the ability to do so (for instance lidar or purpose-built radar) enables additional functions at the previous demand for only one sensor.

14.7.5 Fault Management

In case of a breakdown within our car-IT infrastructure which results in reduced available calculating power and/or reduced available communication bandwidth, the

original driver assistance application has to lower its demand for resources. This is done by *graceful degradation* [13]. The possibilities for this purpose are:

- Data acquisition working at lower performance (lower angular resolution, lower update rate, lower acquisition area, etc.)
- Modifying certain steps of the signal chain to consume less computing power
- Shifting parts of the signal chain into smart-sensors.

Accepting a lower overall function performance is a basic requirement for the concept of graceful degradation. At this, the infrastructure management system tells the affected application to go into degraded mode or rather activates an application in degraded mode, shown under Sect. 14.5.2.

In our first option, the program tells the sensor-internal embedded processor to reduce the amount of acquired data by changing one of the following points to unload the ECU:

Resolution	Resolution of captured angle, distance and other types of data
Acquisition area	Dimension of (each) scanned section
Update rate	Pictures per second. Can easily be changed e.g. at lidar sensors (rotation rate of scan head)
Types of data	There are different informations depending on type of sensor for transmitting available: reflectivity amplitudes, target velocity, angle, distance, current sensor configuration, etc.

Reducing the transmitted amount of data generally lowers the demand for network bandwidth as well as computing power.

The second option to degrade gracefully is modifying certain steps of the signal chain, shown in Fig. 14.19. Most potential for saving computing power, without degrading badly the whole function, provides the tracking block. In most cases a Kalman-Filter is used, which consumes a large part of the computing power used by the entire signal chain because of the necessary matrix-multiplications. Replacing a Kalman based tracker by an Alpha-Filter is an effective option to reduce the demand for resources. In a simulation of a two dimensional object tracking, the calculation time over 1,000 steps of random state changes (object motions) is 3.79 times better if an Alpha-Filter is used instead of a Kalman-Filter. The standard deviation of the prediction error is 1.35 times worse, which is an acceptable factor for graceful degradation.

Doing some (pre-)processing of captured raw data within the sensor is the third way to degrade gracefully. This makes the sensors work partly similar to current ones like Boschs ACC. Some preprocessing of data up to a nearly complete signal processing (of course at a lower performance) is done within the sensor and the

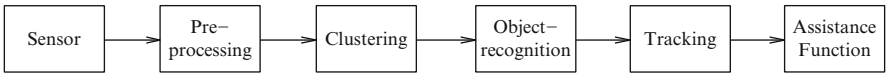


Fig. 14.19 Signal chain of a typical driving environment recognition system

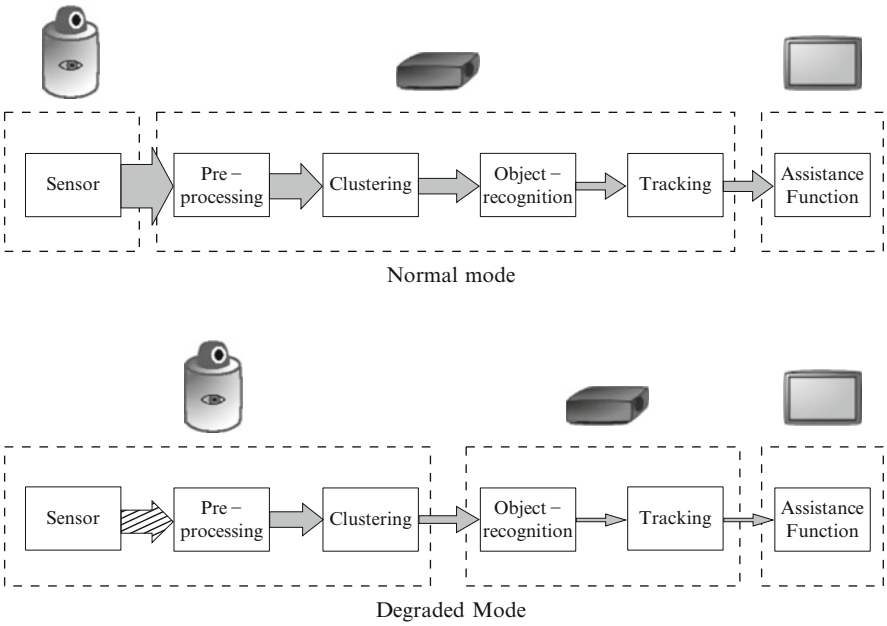


Fig. 14.20 Graceful Degradation by lowering the acquired data amount (hatched arrow) and shifting the first part of the signal chain (raw data preprocessing and clustering) into the smart sensor; dashed line framings corresponds to sensor, ECU and actuator

resulting data is transmitted to the electronic control unit. There, the remaining parts of the signal chain are processed, e.g. a brake- or accelerate-decision could be made with low calculation power. In Fig. 14.20 an example of this kind of graceful degradation is shown.

The width of the arrows corresponds to the network load. Shifting the preprocessing into the smart sensor results in a reduced demand for network bandwidth and a reduced demand for computing power. In this case, advanced algorithms that need raw data cannot work any more.

In principle any part of the signal chain can be shifted into the sensor to keep the function running under difficult conditions like a breakdown of more than one central ECU. Because of the poor performance compared to a classic ECU this can be done only with a massive degradation of the driver assistance system.

This degrading option by using ECU-external computing power can only be done at accepting a lower overall performance of the driver assistance system, the power of the embedded system is not sufficient to replace the whole signal chain with the previous preferences. Increasing the microcontroller performance for being able to this would not make sense because of high costs due to needless redundancy.

If a sensor fails, depending on system configuration we have other possibilities. If the assistance system uses more than one sensor (multi-use), the data fusion quality drops. If there is only one sensor, the last way to keep this function running is using

mostly equivalent data from other sensors covering almost the same area, accepting improper configuration (angular resolution etc.).

14.7.6 Sensor Architecture Implementation

The aforementioned characteristics are achieved in our novel driving environment recognition sensor architecture, see Fig. 14.21.

By our definition, actuators can be everything that uses processed data, e.g. an information display or the engine control. The used sensors are smart-sensors with an embedded system for the configuration of the physical sensor (like angular resolution, update rate and acquisition area) and optional internal data processing. The built-in microcontroller is used for dynamic reconfiguration and sensor fault management/graceful degradation.

Normally, raw data is transmitted to the active ECU, processed and passed to actuators. The availability of environment data for all functions can be guaranteed by an optional real-time database. Storing of already processed data to be accessed by further functions is possible, too.

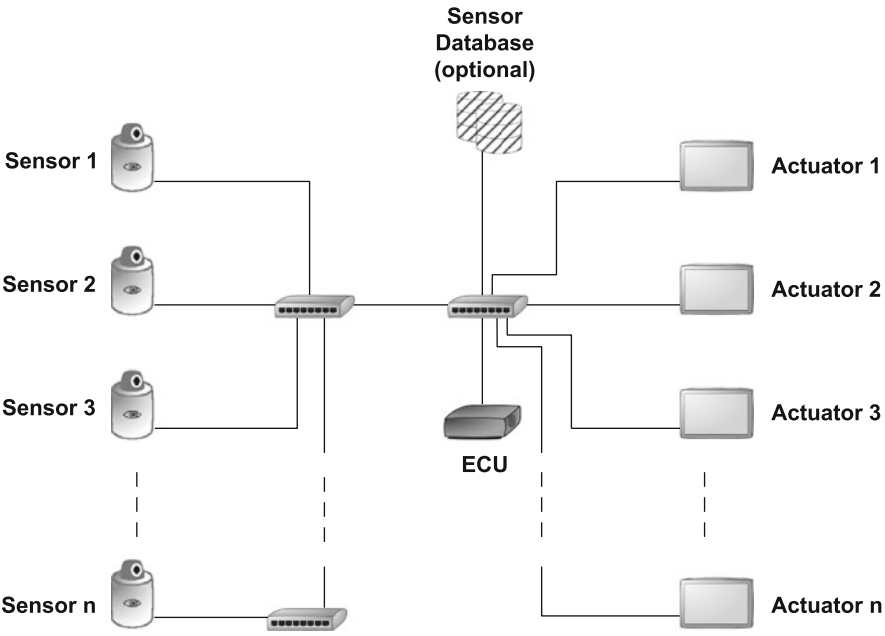


Fig. 14.21 Sensor architecture overview

In graceful degradation mode, the demand for calculation power on the active ECU consumed by driving environment recognition can be driven down, depending on available resources, till only passing through processed data to the actuators.

14.8 Summary

Motivated by high complexity, hard maintainability and high cost of today's Car-IT infrastructure, investigations for a future Car-IT infrastructure were presented, based on the following principles: *Centralization* of heterogeneous infrastructure, i.e. from up to 100 ECUs today down to a few high performance ECUs and a centralized communication network; *Homogenization* towards few standardized components and interfaces in order to benefit from the evolution of consumer and communications electronics; *Virtualization* of computing and communication resources in order to use them more flexibly; *Relocation*, i.e. functions are not statically mapped on ECUs, but might be dynamically relocated within the Car-IT infrastructure; and last but not least *fault tolerance*, i.e. error detection and error mitigation.

Special attention has been on the requirements of three different service classes supported concurrently on one common Car-IT infrastructure. These service classes comprise of *hard real-time* applications, e.g. closed control loops with 1 ms response time as for chassis control applications, *soft real-time* applications, e.g. video codecs for 25 frames per second or Lidar sensor data processing, and *non real-time* applications with best effort service.

Investigations include case studies on the virtualization of multicore ECUs with basic software layers and cache optimization strategies, communication network topologies and protocols, as well as two application studies from infotainment and sensor data processing domains. Throughout all investigations, the guideline was on exploiting the principles of centralization, homogenization, virtualization, relocation, and fault tolerance, while concurrently supporting real-time and non real-time applications.

Virtualization of multicore ECUs is based on a hypervisor for the management of CPU cores, memory resources and I/O devices. A prototype implementation was validated for an infotainment application, including video transcoding and camera picture overlay, a soft real-time sensor data fusion application, and hard real-time control loops running concurrently on two virtualized CPU cores with XEN hypervisor and both Linux and Windows domains, demonstrating to meet the required performance and real-time constraints.

The communication network was designed to support both real-time and best effort services, as well as fault tolerance, using commercially available off-the-shelf hardware. The communication traffic was separated into three safety classes with different resilience mechanisms, i.e. safety critical data is transmitted on redundant paths, critical data is switched to precalculated alternative paths in case errors were

detected, for non-critical data new paths are calculated online whenever errors occur. These mechanisms were validated analytically, by simulation and by a test setup.

On top of the virtualized CPU cores and the communication network, basic software services account for local configuration and diagnosis, as well as for global error detection and mitigation strategies. Safety critical real-time applications might be implemented using triple module redundancy, while non and soft real-time applications might be gracefully degraded. We investigated basic software services with a focus on data consistency, real-time scheduling, dynamic task migration, and cache performance, taking into account Autosar compatibility.

References

1. (2008) AUTOSAR (R3.1) Specification
2. Alves-foss J, Harrison WS, Oman P, Taylor C (2006) The mils architecture for high-assurance embedded systems. Int J Embedded Syst 2:239–247
3. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the art of virtualization. In: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, ACM, New York, pp 164–177, DOI http://doi.acm.org/10.1145/945445.945462
4. Baruah S (1998) A general model for recurring real-time tasks. In: Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE, pp 114–122
5. Baruah S, Chen D, Gorinsky S, Mok A (1999) Generalized multiframe tasks. Real-Time Syst 17:5–22
6. Baruah S, Chen D, Mok A (1999) Static-priority scheduling of multiframe tasks. In: Real-Time Systems. Proceedings of the 11th Euromicro Conference on, 1999, pp 38–45
7. Baruah SK (2003) Dynamic- and static-priority scheduling of recurring real-time tasks. Real-Time Syst 24(1):93–128
8. Eichhorn M, Schmid M, Steinbach E (2008) A realtime streaming architecture for in-car multimedia: Design guidelines and prototypical implementation. In: IEEE International Conference on Vehicular Electronics and Safety. ICVES 2008, pp 157–162
9. Fedorova A (2006) Operating system scheduling for chip multithreaded processors. Ph.D. thesis, Harvard University, Cambridge, MA
10. Gresser K (1993) Echtzeitnachweis ereignisgesteuerter realzeitsysteme. PhD thesis, Technische Universität München
11. Gross D, Harris C (1985) Fundamentals of queueing theory . Wiley, New York
12. Hergenhan A, Heiser G (2008) Operating systems technology for converged ECUs. In: 6th Embedded Security in Cars Conference (escar), ISITS, Hamburg, Germany
13. Holzknecht S, Biebl EM, Michel HU (2009) Graceful degradation for driver assistance systems. In: Advanced microsystems for automotive applications, Springer, Berlin, pp 255–265
14. Kennedy J, Eberhart R (1995) Particle swarm optimization. In: IEEE International Conference on Neural Networks, Perth, WA, Australia, vol. 4, pp. 1942–1948
15. Kivity A (2007) kvm: The linux virtual machine monitor. In: OLS '07: The 2007 Ottawa Linux Symposium, pp 225–230
16. Kopetz H (1991) Event-triggered versus time-triggered real-time systems. In: Proceedings of the international workshop on operating systems of the 90s and Beyond, Springer, London, UK, pp 87–101
17. Lehoczky J, Sha L (1986) Performance of real-time bus scheduling algorithms. Proceedings of the 1986 ACM SIGMETRICS joint international conference on Computer performance modelling, measurement and evaluation pp 44–53

18. Liu C (2005) Exploiting multi-threaded application characteristics to optimize performance and power of chip-multiprocessors. Ph.D. thesis, The Pennsylvania State University
19. Liu C, Layland J (1973) Scheduling algorithms for multiprogramming in a hard-real-time environment. J ACM (JACM) 20(1):46–61
20. Liu JW (2000) Real-time systems. Prentice Hall, Englewood Cliffs, NJ
21. Mitschke M, Wallentowitz H (2004) Dynamik der Kraftfahrzeuge. Springer Berlin, Heidelberg, New York
22. Mok A, Chen D (1996) A multiframe model for real-time tasks. In: Real-time systems symposium, 17th IEEE, pp 22–29
23. Mueller-Rathgeber B, Rauchfuss H (2008) A cosimulation framework for a distributed system of systems. In: IEEE 68th vehicular technology conference, VTC 2008-Fall, pp 1–5
24. Mueller-Rathgeber B, Eichhorn M, Michel H (2008) A unified Car-IT communication-architecture: Design guidelines and prototypical implementation. In: 2008 IEEE intelligent vehicles symposium (IV08), pp 709–714
25. Mueller-Rathgeber B, Eichhorn M, Michel H (2008) A unified Car-IT communication-architecture: Network switch design guidelines. In: IEEE international conference on vehicular electronics and safety (ICVES), 2008, pp 16–21
26. Poledna S, Burns A, Wellings A, Barrett P (2000) Replica determinism and flexible scheduling in hard real-time dependable systems. IEEE Trans Comp 49(2):100–111
27. Rauchfuss H, Wild T, Herkersdorf A (2010) A network interface card architecture for I/O virtualization in embedded systems. In: Second Workshop on I/O Virtualization (WIOV'10)
28. Salman A, Ahmand I, Al-Madani S (2002) Particle swarm optimization for task assignment problem. In: Elsevier (ed) Microprocessors and microsystems, Elsevier, Amsterdam, Netherlands, vol. 26, pp 363–371
29. Sherwood T, Perelman E, Hamerly G, Sair S, Calder B (2003) Discovering and exploiting program phases. IEEE Micro: Micro's top ricks from computer architecture conferences
30. Stankovic JA, Spuri M, Ramamritham K, Buttazzo GC (1998) Deadline scheduling for real-time systems: EDF and related algorithms. Kluwer, Dordrecht
31. Takada H, Sakamura K (1997) Schedulability of generalized multiframe task sets under static priority assignment. In: Proceedings – fourth international workshop on real-time computing systems and applications, pp 80–86
32. Uhlig R, Neiger G, Rodgers D, Santoni AL, Martins FC, Anderson AV, Bennett SM, Kgi A, Leung FH, Smith L (2005) Intel virtualization technology. Computer 38:48–56, DOI http://doi.ieeecomputersociety.org/10.1109/MC.2005.163
33. Wenger J (2005) Automotive radar – status and perspectives. In: Conference procceedings, compound semiconductor integrated circuit symposium, vol 29, pp 21–24
34. Wind River (2009) White paper: Wind river hypervisor, http://www.windriver.com/announces/hypervisor/
35. Winner H et al (2002) Adaptive Fahrgeschindigkeitsregelung ACC. Robert Bosch, GmbH
36. Xia W, Wu Z (2005) A hybrid particle swarm optimization approach for the job-shop scheduling problem. Computers & Industrial Engineering, Elsevier, Amsterdam, Netherlands, 48(2): 409–425.
37. Zwick M, Durkovic M, Obermeier F, Bamberger W, K D (2009) Mcccsim - a highly configurable multi core cache contention simulator. Tech. rep., Technische Universität München, https://mediatum2.ub.tum.de/doc/802638/802638.pdf
38. Zwick M, Durkovic M, Obermeier F, K D (2009) Setvectors for memory phase classification. In: International conference on computer science and its applications (ICCSA'09)

Chapter 15
Robot Basketball – A New Challenge for Real-Time Control

Georg Bätz, Kolja Kühnlenz, Dirk Wollherr, and Martin Buss

15.1 Introduction

Most of the industrial robots nowadays still employ strategies that neglect or minimize the effects of task dynamics. This simplifies task planning and reduces the required sensor capacities. In order to gain maximum performance with such an approach, these robotic systems are typically highly specialized and offer little flexibility. The potential areas of application for robots, however, gradually extend beyond the classical industrial settings in large scale enterprises. The envisioned range of use includes small-scale enterprises and households. This trend requires the development of efficient and flexible systems with advanced cognitive capabilities, such as perception, action, learning, or planning. The main research direction of the robotic basketball project is the development of advanced sensory and motor skills for robotic manipulators. With these skills, it is possible to actively use the dynamics of a given task. The paradigm of dynamic manipulation offers three potential benefits: first, the execution time of tasks can be reduced. Second, simpler gripper structures can be used. Third, the dexterity of the robotic system can be increased. In accordance with Mason, we refer the term dynamic manipulation to methods which actively use the task dynamics instead of merely tolerating them [1]. Manipulation techniques can be classified by the elements which are needed for a complete description. As depicted in Table 15.1, four classes of manipulation

G. Bätz, (✉) · D. Wollherr · K. Kühnlenz · M. Buss
Institute of Automatic Control Engineering, Technische Universität München,
D-80290 München, Germany
e-mail: georg.baetz@tum.de,mb@tum.de

D. Wollherr · K. Kühnlenz
Institute for Advanced Study, Technische Universität München, D-80290 München, Germany
e-mail: koku@tum.de; dw@tum.de

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_15,

Table 15.1 Taxonomy of manipulation by elements needed for analysis [1]

Class	Kinematic manipulation	Static manipulation	Quasi-static manipulation	Dynamic manipulation
Kinematics	✓	✓	✓	✓
Static forces	-	✓	✓	✓
Quasi-static forces	-	-	✓	✓
Acceleration forces	-	-	-	✓

are commonly distinguished. Compared to conventional manipulation, dynamic manipulation allows new motion paradigms such as throwing, catching, or batting.

For nonprehensile dynamic manipulation, the idea is to perform manipulation tasks without applying force or form closure grasps. Relative motion between end effector and object is actively employed and the object can perform a broader class of motions as compared to a static grasp.

The robot basketball project aims at providing a thorough control framework for dynamic manipulation. Such a framework facilitates the use of dynamic manipulation and hence provides new motion paradigms for robotic manipulation. In order to reach this goal, the project has the following objectives:

- Action planning for dynamic manipulation with special emphasis on sensor uncertainties.
- Investigation of nonprehensile dynamic manipulation and comparison of this methodology with conventional manipulation.
- Integrated hardware and control design. The concept of intrinsic compliance is used to minimize impact forces and to maximize peak output power.
- Engaging human robot interaction through a dynamic and flexible, and hence more *natural*, robot behavior.
- Environment modeling and creation of an object library providing geometrical (e.g. shape) and dynamic (e.g. mass, inertia matrix, stiffness) object properties.

This chapter summarizes the first steps towards reaching these objectives: first, an overview on related work in dynamic manipulation and environment perception is provided (Sect. 15.2). Second, an extensive control architecture for dynamic manipulation tasks based on multimodal sensor information is presented (Sect. 15.3). Third, the chapter outlines a method of object tracking and trajectory prediction with high frame rates (Sect. 15.3.1). Fourth, the optimal planning of robot actions is discussed (Sect. 15.3.2). Fifth, the proposed control design approach is experimentally validated for two dynamic manipulation tasks (Sect. 15.4).

15.2 Related Work

Dynamic Manipulation: robots with dynamic skills have been studied in various research works. Hodgins and Raibert were among the first and investigated dynamic

locomotion primitives such as hopping for robots [2]. With respect to manipulation, dynamic dexterity was first addressed by Mason and Lynch [1]. Following this idea, Huang investigated impulsive manipulation [3].

In addition, many works focused on specific dynamic manipulation tasks: the control of rhythmic tasks, for instance, has been a very active research area over the last two decades. These tasks are commonly summarized with the term juggling [4–6]. Bühler et al. coined this term for tasks that require interaction with one or multiple objects that would otherwise fall freely in the earths gravitational field [4]. The continuous motion of the actuator is used to control the continuous motion of the object through an intermittent contact. Bühler et al. proposed and experimentally verified the mirror algorithm to control the classic juggling task [7]. Schaal and Atkeson presented an open-loop sinusoidal actuation to obtain stable fixed points in the juggling task [5]. In their experimental setup, they used a one-joint robot with pantograph linkage to maintain a horizontal paddle orientation. Shiokata et al. discussed the strategy of dynamic holding and presented robotic dribbling with a ping-pong ball and a multi-fingered hand in [8]. Robotic throwing tasks were also investigated by several researchers: Kato and coworkers planned throwing motions and presented experimental results for a two DOF robot and an object with mass 0.01 kg [9]. Katsumata et al. discussed the throwing task with an underactuated two DOF robotic manipulator [10]. Considering more complex robotic structures, Lombai and Szederkenyi followed an offline approach to optimize throwing trajectories for a six DOF robot [11]. Senoo used a four DOF manipulator and a robotic hand to generate throwing trajectories for a tennis ball [12]. They proposed a rolling sequence of the ball on the hand in order to increase the release velocity. The task of robotic ball catching has also been studied by various researchers: Hove and Slotine presented three-dimensional catching with a four DOF manipulator [13]. Frese and Bäuml discussed kinematically optimal catching with a seven DOF robot and a robotic hand [14, 15]. Riley and Atkeson investigated ball catching with a humanoid robot using a baseball glove to catch [16].

Environment Perception: the approach of dynamic manipulation relaxes the requirements for the manipulator hardware. In return, an increased effort for motion/action planning is required. For effective task planning and execution, a detailed environment perception is preliminary. In order to obtain detailed and robust information, it is crucial to rely on different sensor modalities. Here, visual feedback from cameras is the most important source of information. High-speed vision improves control performance in vision-based control due to higher frame rates and lower latencies. Recent progress in sensors, bus systems and semiconductor technology has led to a few works on vision with more than 30 Hz frame rate during the last decade: Ishikawa et al. have developed high-speed vision systems with various resolutions and fast low-level image processing functions integrated on customized processing hardware allowing frame rates of up to 1 kHz. These vision systems were applied to various dynamic vision-based manipulation tasks, e.g. ball dribbling with multi-fingered hands, ball catching, ball batting, or regrasping [8, 17–19]. Various works in the field of visual servo control with higher frame rates than common 30 Hz NTSC exist, e.g. [20,21]. Other works in high-speed

vision in robotics are concerned with navigation and motion estimation issues, e.g. [22, 23].

Besides visual feedback, the knowledge of contact forces and torques is particularly crucial for the interaction with the environment. To this end, robotic systems are typically equipped with force/torque sensors at the wrist. For dynamic manipulation tasks, however, the problem arises that the inertial forces/torques of the end effector have a non-negligible effect on the measurements of the wrist sensor. This problem has been addressed by several researchers: dynamic force sensing for high-speed robot manipulation was first investigated by Uchiyama et al. [24]. The developed observer design was restricted to pose and F/T measurements. In the presented results, the complexity of the problem was reduced by considering a planar scenario with two translational and one rotational DOF. Garcia et al. investigated a sensor fusion approach for dynamic force/torque estimation [25, 26]. In addition to pose and F/T measurements, an inertial sensor was used. In the EKF-based observer design, the environment forces and torques were not considered and the estimation error was utilized to obtain a contact F/T estimator with low-pass properties. Considering the nonlinear process model for the torques, it seems problematic to use the estimation error of such an observer to determine the environment torques. In addition, the use of Euler angles for the tool orientation appears critical because of the well-known issues with representation singularities.

15.3 Control Design

The overall control framework is shown in Fig. 15.1. Here, $\boldsymbol{q}$ are the joint angles and $\boldsymbol{x}_M = \left[\boldsymbol{p}_M^T \; \boldsymbol{o}_M^T\right]^T$ the position and orientation (in quaternion notation) of the end effector. The subscripts a and d denote actual and desired quantities.

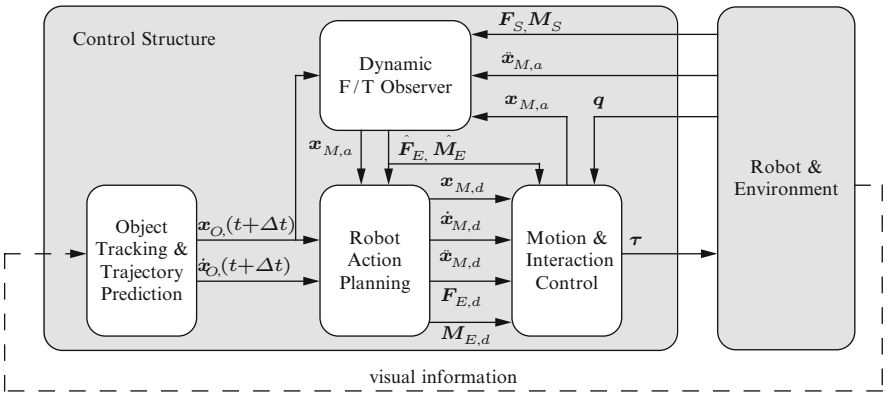


Fig. 15.1 Overall control structure consisting of four main elements: robot action planning, dynamic F/T observer, object tracking & trajectory prediction, and motion & interaction control

$\boldsymbol{F}_S$ and $\boldsymbol{M}_S$ denote the measured forces and torques, whereas $\boldsymbol{F}_E$ and $\boldsymbol{M}_E$ are the environment forces/torques. The predicted trajectories of the manipulated object are labeled $\boldsymbol{x}_{O,(t+\Delta t)}$. The control structure consists of four main elements: dynamic force/torque observer, object tracking and trajectory prediction, robot motion planning, and motion/interaction control. In the first module, the environment forces and torques are estimated based on the measurements of a force/torque and an acceleration sensor, see [27]. The second module realizes object tracking with high frame rates (appr. 150 Hz) and predicts future object trajectories, see Sect. 15.3.1. The motion planning and task-specific robot trajectory generation is performed in the third module, see Sect. 15.3.2. For the interaction with the environment, a hybrid force/position control in operational space is used. Here, the dynamic F/T observer is needed to provide reliable estimates of the contact forces during dynamic motions.

15.3.1 High-Speed Vision

The vision module contains two elements: object tracking and trajectory prediction, compare Fig. 15.2. In the following, tracking and trajectory prediction of a spherical object is described.

15.3.1.1 Object Tracking Module

The tracking algorithm assumes that color and shape information of the tracked object are available. First, a Gaussian smoothing filter is applied to the image in RGB-format. Then, the image is converted into the HSV color space. The HSV space consists of the three channels hue (H), saturation (S), and value (V) and the object is extracted based on the information in the H and V channel. For the hue channel, a look-up table is used containing the color information of the tracked object. For the value channel, a threshold value is set to exclude darker regions of the image from the analysis. This is helpful since the darker regions do not provide reliable color information. Combining the two images with an *and* operation results in a binary image. In the next step, an *opening* operation is performed on the binary image in order to filter out speckles, not related edges, and noise in the image [28]. The *opening* includes an elementary erosion operation followed by an dilation operation, both using a 3×3 pixel circle as structural element. Finally, by

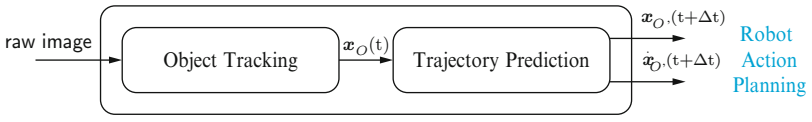


Fig. 15.2 Block diagram of the vision module

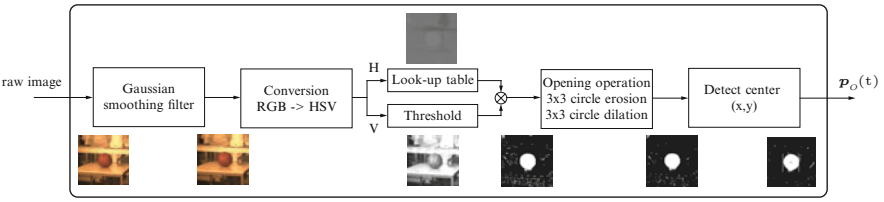


Fig. 15.3 Object tracking module: block diagram of the image processing

computing the first order moment of the image, the geometric center of the object is obtained. A block diagram with the processing steps of the tracking algorithm is depicted in Fig. 15.3. In order to increase the tracking frequency and to reduce the latency, a moving window search method is used: once the object is detected in the full size image (1280 × 1024 pixel), the search area is reduced to a 180 × 180 pixel window. The position of the window is determined by a linear position prediction and tracking is performed with approximately 150 Hz. If the object is lost, e.g. due to occlusion, the search algorithm is again applied to the full-size image.

15.3.1.2 Trajectory Prediction Module

The input for the module is the object position $\boldsymbol{p}_O(t)$ tracked by the camera system. The following two paragraphs describe the trajectory prediction of the object's trajectory for a free-flight phase and for an impact event. Again, a spherical object is considered.

Free-flight: first, the module determines whether the object is in free-flight. This can be realized in various ways: one approach is to check whether the vertical object acceleration matches the gravitational acceleration for a certain period of time. If that is the case, one can assume that the object is in free-flight. The drawback of this method is that the acceleration $\boldsymbol{a}_O(t)$ of the object has to be determined. However, if only position information is available, noisy and hence imprecise acceleration estimates impede this approach. Hence, a different approach is used in this work: a free-flight phase is detected when the objects position is outside the workspace of the two robots and the human (which has to be pre-specified). The prediction of the objects trajectory is realized with a recursive least squares fitting method for a sample period ΔT_s. The trajectory for free-flight is given by

$$\boldsymbol{p}_{[k]} = \begin{bmatrix} p_{x[k]} \\ p_{y[k]} \\ p_{z[k]} - 0.5gt_{[k]}^2 \end{bmatrix} = \begin{bmatrix} p_{x,0} & v_{x,0} \\ p_{y,0} & v_{y,0} \\ p_{z,0} & v_{z,0} \end{bmatrix} \begin{bmatrix} 1 \\ t_{[k]} \end{bmatrix} \tag{15.1}$$

where g denotes the gravitational acceleration. This results in the recursive least squares estimates for the parameters

$$\hat{\boldsymbol{m}}_{i[k+1]} = \hat{\boldsymbol{m}}_{i[k]} + \boldsymbol{h}_{[k]} \left(\bar{p}_{i[k+1]} - \boldsymbol{R}^T_{[k+1]} \hat{\boldsymbol{m}}_{i[k]} \right) \tag{15.2}$$

where $\bar{p}_{i[k+1]}$ is the measured i-coordinate of the object and

$$\hat{\boldsymbol{m}}_{i[k]} = \begin{bmatrix} p_{i,0[k]} & v_{i,0[k]} \end{bmatrix}^T \quad i \in \{x, y, z\}, \qquad \boldsymbol{R}_{[k+1]} = \begin{bmatrix} 1 & t_{[k+1]} \end{bmatrix}^T \tag{15.3}$$

$$\boldsymbol{h}_{[k]} = \frac{\boldsymbol{\Pi}_{[k]} \boldsymbol{R}_{[k+1]}}{1 + \boldsymbol{R}^T_{[k+1]} \boldsymbol{\Pi}_{[k]} \boldsymbol{R}_{[k+1]}}, \qquad \boldsymbol{\Pi}_{[k+1]} = \boldsymbol{\Pi}_{[k]} - \boldsymbol{h}_{[k]} \boldsymbol{R}^T_{[k+1]} \boldsymbol{\Pi}_{[k]}.$$

With the estimates for the initial state of the object, the future object trajectory $\boldsymbol{x}_O(t + \Delta t)$, $\dot{\boldsymbol{x}}_O(t + \Delta t)$ is predicted.

Impacts: for some tasks, e.g. the dribbling task, it is desirable to predict object trajectories that include impact events in addition to free-flight phases. The prediction model is based on the following assumptions: first, the angle of impact with respect to the normal direction is smaller than 30°. In this case, no sliding occurs at the contact point, see [29,30]. Second, rotational velocities of the object are negligible. With these assumptions, the objects tangential velocity after the impact is approximated as

$$\boldsymbol{v}^+_{t,O} \approx \frac{m_O r_O^2}{J_O + m_O r_O^2} \boldsymbol{v}^-_{t,O} \tag{15.4}$$

where J_O denotes the moment of inertia, m_O the mass, and r_O the radius of the ball. Inserting the expression for a spherical shell, $J_O = \frac{2}{3} m_O r_O^2$, in (15.4) leads to a proportionality constant $c_{r,t} = 0.6$ for the horizontal velocities before and after the impact. The translational object velocity after the ground impact is hence defined as

$$\boldsymbol{v}^+_O = \begin{bmatrix} c_{r,t} & c_{r,t} & -c_r \end{bmatrix}^T \boldsymbol{v}^-_O \tag{15.5}$$

where c_r is the coefficient of restitution for the ground impact.

The models for the free-flight phase and the impact event provide the possibility to predict the future object trajectory and use it for the robot action planning.

15.3.2 Robot Action Planning

The term *action planning* refers to the planning of desired motion ($\boldsymbol{x}_{M,d}$, $\dot{\boldsymbol{x}}_{M,d}$, $\ddot{\boldsymbol{x}}_{M,d}$) and/or desired force and torque ($\boldsymbol{F}_{E,d}$, $\boldsymbol{M}_{E,d}$) trajectories. For dynamic manipulation tasks, the action planning has to be performed online based on the provided sensor feedback. The inputs are the actual end effector pose ($\boldsymbol{x}_{M,a}$), the predicted object trajectory ($\boldsymbol{x}_{O,(t+\Delta t)}$, $\dot{\boldsymbol{x}}_{O,(t+\Delta t)}$), and the estimated forces/torques exchanged with the environment ($\hat{\boldsymbol{F}}_E$, $\hat{\boldsymbol{M}}_E$). The overall goal is to find motions that fulfill the desired optimization criteria and that are dynamically feasible. In

this context, dynamically feasible means that the resulting hybrid state trajectory can be realized with a control input $\boldsymbol{u}$ and does not exceed hardware limitations. The following subsection presents the constraints that need to be considered in the planning stage. It also discusses selection criteria that can be used to optimize the trajectory of the hybrid system.

15.3.2.1 Constraints

Two types of constraints are distinguished: two-sided equality constraints in the form $\boldsymbol{c}_e(\boldsymbol{\gamma}) = \mathbf{0}$ and one-sided inequality constraints $\boldsymbol{c}_i(\boldsymbol{\gamma}) \leq \mathbf{0}$. The former are used to specify positions, velocities, and accelerations at certain points in time. The latter consider hardware limitations such as maximum manipulator acceleration or joint limits.

15.3.2.2 Optimization Criteria

The following paragraphs outline selection criteria w_i for the optimal trajectory planning. Based on these selection criteria, task-specific cost functions

$$J(\boldsymbol{\gamma}, w_1, \ldots, w_n) \tag{15.6}$$

are defined. The optimization problem is given by

$$\begin{aligned} \min_{\boldsymbol{\gamma} \in \Gamma} \quad & J(\boldsymbol{\gamma}, w_1, \ldots, w_n) \\ \text{with} \quad \Gamma &= \{\boldsymbol{\gamma} \in \mathbb{R}^n, \boldsymbol{c}_i(\boldsymbol{\gamma}) \leq \mathbf{0}, c_e(\boldsymbol{\gamma}) = 0\} \\ \boldsymbol{\gamma} &= [a_1, \ldots, a_n]\,. \end{aligned} \tag{15.7}$$

The parameter set $\boldsymbol{\gamma}$ defines the trajectory and thus also the corresponding joint angles $\boldsymbol{q}$. Consequently, the selection criteria $w_i(\cdot)$ can be formulated either as function of $\boldsymbol{\gamma}$ or $\boldsymbol{q}$.

(a) *Distance from joint limits:* the distance from mechanical joint limits imposes constraints on the trajectory planning. However, it can also be utilized as an optimization criteria which ensures that the manipulator stays within preferred or feasible regions of its workspace.

$$w_j(\boldsymbol{q}) = \prod_{i=1}^{n} w_{j,i}(q_i, q_{i,min}, q_{i,max}) \tag{15.8}$$

where $\boldsymbol{q} \in \mathbb{R}^n$ is the vector of joint angles and $w_{j,i}$ corresponds to the joint limit measure for the i-th joint.

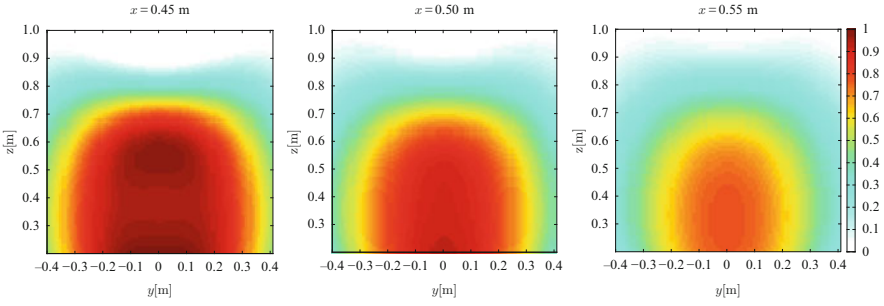


Fig. 15.4 Normalized dynamic manipulability measure w_d for different yz-planes: (**a**) $x = 0.45$ m, (**b**) $x = 0.50$ m, (**c**) $x = 0.55$ m

(b) *Dynamic manipulability measure:* a global measure for the manipulation ability has been proposed by Yoshikawa [31]. The measure quantifies the ability for arbitrarily changing position and orientation of the end effector in a given posture. The main drawback of this concept is the fact that it is a kinematic measure ignoring the arm dynamics. Hence, it is not suitable for planning dynamic motions. This shortcoming is addressed by the dynamic manipulability measure w_d introduced in [32]. The scalar measure is defined as

$$w_d(\boldsymbol{q}) = \frac{1}{w_{d,max}} \sqrt{\det\left(\boldsymbol{J}(\boldsymbol{q})\left(\boldsymbol{B}^T(\boldsymbol{q})\boldsymbol{B}(\boldsymbol{q})\right)^{-1}\boldsymbol{J}^T(\boldsymbol{q})\right)} \tag{15.9}$$

where $\boldsymbol{J}(\boldsymbol{q}) \in \mathbb{R}^{6\times n}$ is the manipulator Jacobian and $\boldsymbol{B}(\boldsymbol{q}) \in \mathbb{R}^{n\times n}$ is the inertia matrix of the manipulator. The scaling factor $w_{d,max}$ is used to normalize the measure. Figure 15.4 exemplarily shows the dynamic manipulability measure of the six DOF industrial robot used in this work for different yz-planes and a constant end effector orientation. The reference coordinate system is located in the robot base with the z-direction pointing vertically upwards.

(c) *Object orientation:* in accordance with the distance from mechanical joint limits, the object orientation can be regarded as both a constraint and an optimization criteria. For a specific object, there might exist a range of admissible contact points. Preferences within this range can be considered with a selection criteria $w_o(\boldsymbol{q}, \boldsymbol{o}_O)$, where $\boldsymbol{o}_O$ denotes the orientation of the object.

(d) *Energy consumption:* finally, the energy consumption of the system is another criteria for the evaluation of planned trajectories. It can be approximated by

$$w_u(\boldsymbol{\gamma}) = \left(\lambda \int_{t_s}^{t_e} \boldsymbol{u}(t)^T \boldsymbol{u}(t) dt\right)^{-1}, \tag{15.10}$$

where λ is a constant scaling factor and $[t_e,\ t_s]$ denotes the duration of the task execution.

15.3.2.3 Optimization Method

In order to generate near-optimal trajectories in limited time, the optimization method combines offline and online decisions. The method considers tasks that include a free-flight phase. This is a common feature for many dynamic manipulation tasks, the two best-known examples are catching and throwing. In a first step, equally spaced grid points are defined in the robot workspace, compare Fig. 15.5. For each grid point, the values of the optimization criteria w_j and w_d are determined by averaging the measures for different end effector orientations at that point. For the 14 DOF manipulator, the redundancy has been resolved so that distinct values can be assigned to each grid point. These values are stored in look-up tables. In the second step, motion trajectories are preplanned. Here, the initial state of the manipulator for a given task is assumed to be fixed. The trajectories are generated using fifth-order polynomials which coefficients are determined by position, velocity, and acceleration constraints at initial and final time. The resulting trajectories are checked for dynamic feasibility and their costs are determined based on a linear combination of w_u, w_d, and w_j. Again, each grid point is assigned a distinct value by averaging the costs for different final states and the costs are stored in a look-up table. The third step of the optimization is performed online: the module determines where the desired or predicted free-flight trajectory of the object intersects the workspace of the robot, compare Fig. 15.5. For this part of the free-flight trajectory, the nearest neighbors in the offline computed look-up tables are determined and the ten points with the lowest costs are considered as *candidate points*. For each of these points, the module generates trajectories using fifth-order polynomials. Based on the desired/predicted free-flight trajectory of the object, the coefficients of the polynomials are determined by the task-specific position, velocity, and acceleration constraints at initial and final time. The resulting trajectories of the candidate points are checked for workspace and acceleration constraints violations. Together with the overall task goal, these motion trajectories then determine force trajectories for remaining task directions.

15.3.2.4 Trajectory Generation

In the following, trajectory generation based on the presented selection criteria is exemplarily studied. Circular plates are used as end effector for the robotic manipulators. For the trajectory generation, the following assumptions are made:

A1 Air resistance and rotational ball velocity are negligible.
A2 Impacts between ball and end effector are instantaneous inelastic collisions described by the coefficient of restitution c_r.

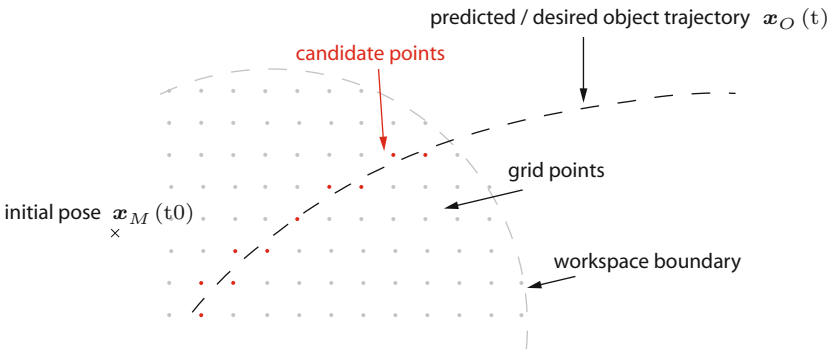


Fig. 15.5 Illustration of the optimization approach

A3 For an inelastic collisions, the angle of incidence is equal to the angle of reflexion.

Task 1 – Dual-Handed Throwing: the two-handed throwing of spherical objects with a basic force-closure grasp is considered. Hence, the object's orientation and its rotational velocity are neglected. The state of the system is then given by

$$\boldsymbol{x} = \left[\boldsymbol{x}_{M,1}^T \; \boldsymbol{x}_{M,2}^T \; \boldsymbol{p}_B^T \; \dot{\boldsymbol{x}}_{M,1}^T \; \dot{\boldsymbol{x}}_{M,2}^T \; \boldsymbol{v}_B^T \right]^T \quad \in \mathbb{R}^{32} \tag{15.11}$$

with $\boldsymbol{x}_{M,i} = \left[\boldsymbol{p}_{M,i}^T \; \boldsymbol{o}_{M,i}^T \right]^T \in \mathbb{R}^7$ and $\dot{\boldsymbol{x}}_{M,i} = \left[\boldsymbol{v}_{M,i}^T \; \boldsymbol{\omega}_{M,i}^T \right]^T \in \mathbb{R}^6$. A schematic of the of the dual-handed throwing task is depicted in Fig. 15.6. As first subtask, a grasping motion is executed if a stationary ball position within the robots workspace is tracked. The grasping is realized by *force closure*, applying a desired normal force on the contact surface. The imposed constraint for the normal force F_n is

$$||m_B(\boldsymbol{a}_{B,max} + \boldsymbol{g})|| \leq F_r = \mu_s F_n, \tag{15.12}$$

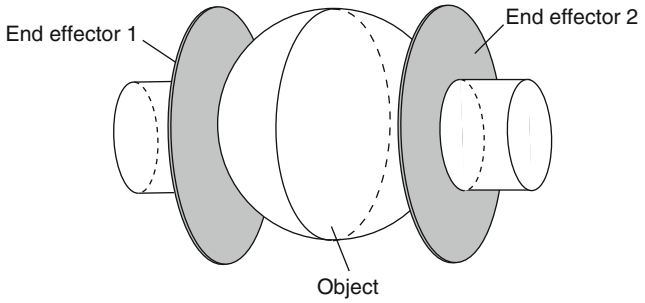


Fig. 15.6 Schematic of the dual-handed throwing with a force closure grasp

where μ_s is the static friction coefficient. For the two manipulators, the throwing trajectories in the directions tangential to the contact surfaces are planned based on the optimization method presented in Sect. 15.3.2.3. For the direction normal to the contact surfaces, the desired grasping force is specified.

Task 2 – Ball Dribbling: for this task, a compliant end effector design is used. The compliance is realized with a spring element that allows a temporary energy storage. During contact, the compliance is in series between the actuator and the ball, decoupling the actuator inertia from the end effector. Therefore, the velocity and acceleration requirements on the actuator are relaxed. Details on the mechanical design of the end effector and the trajectory planning are given in [33].

15.4 Experimental Results

For the experiments, a 6 DOF industrial robot and a 14 DOF dual-arm manipulator are used. The robots are equipped with either six DOF force/torque sensors or twelve DOF force/torque and acceleration sensors from JR3 that are located at the wrist. Circular plates are used as end effectors. The plates are attached to robot either through a rigid connection or through an elastic coupling which allows relative motion between plate and robot.

15.4.1 Task 1 – Dual-Handed Throwing

In order to evaluate the throwing strategy, a basketball was placed stationary in the workspace of an anthropomorphic dual-arm robot with 14 DOF. The robot

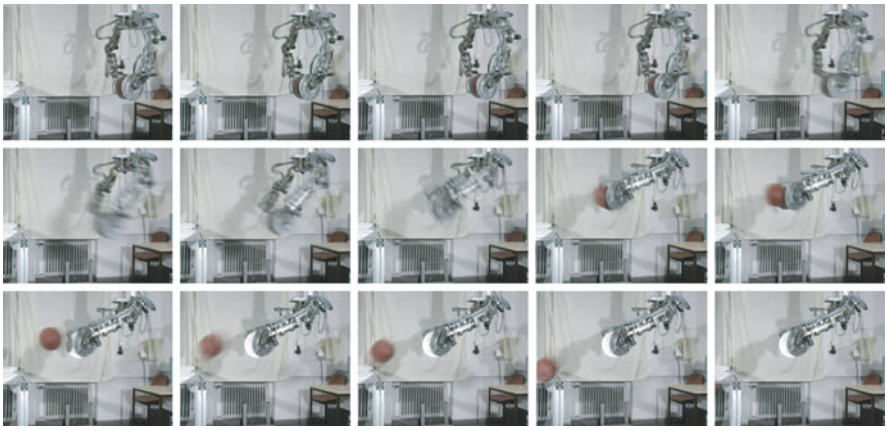


Fig. 15.7 Dual-handed throwing – experimental snapshots: motion sequence from 0 to 1 s

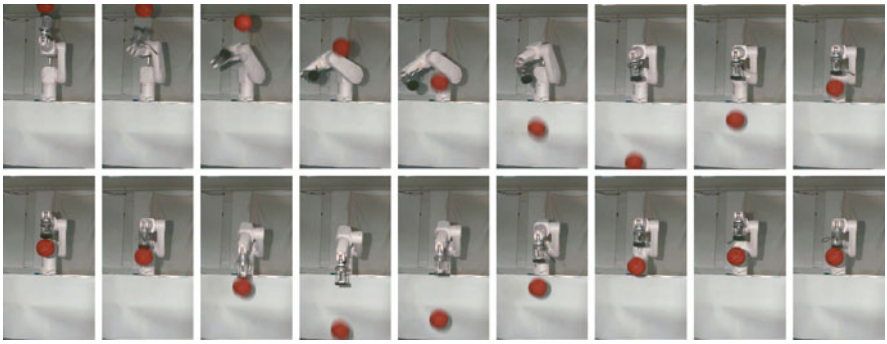


Fig. 15.8 Dribbling with a compliant actuator – experimental snapshots: autonomous task initialization and first dribbling cycle (sequence from 0 to 2.16 s)

then grasped the ball with a force closure grasp. The trajectory planning for the task was detailed in Sect. 15.3.2.4. Experimental snapshots of ball throwing with the dual-arm manipulator are depicted in Fig. 15.7. With the current manipulator hardware, the throwing distance is limited: for target points which are further away than 3 m from the robot base, all generated trajectories violate dynamic constraints (e.g. maximum end effector acceleration). However, for target points within this range the trajectory generation finds feasible solutions. During task execution, the dynamic force sensing is used to apply a constant grasping force and to maintain the contact between the ball and the two end effectors.

15.4.2 Task 2 – Ball Dribbling

The dribbling task is either initialized by the robot or the human operator. In the former case, which was used for the presented experiments, the ball is initially at rest on the plate and then dropped by the robot to start the dribbling cycle. Figure 15.8 illustrates this *autonomous* initialization. In the latter case, the task is triggered when the ball is dropped into a specified area of the robot workspace. The coefficient of restitution of the ball was experimentally determined to be $c_r = 0.84$. During the dribbling cycle, the ball position is tracked by the vision system. The compliant end effector design allows for a continuous-time control phase and relaxes the actuator requirements. A snapshot sequence of the initialization and the first dribbling cycle is depicted in Fig. 15.8. Currently, the performance is mainly limited by the following aspect: in the simulation, it was assumed that there is no energy loss in the elastic actuator. For the real actuator however, friction forces during spring compression/elongation cause energy dissipation. This, in turn, leads to a reduced amount of energy storage in the spring and energy transfer to the ball, see [33].

15.5 Conclusion

This chapter introduced robotic basketball as a new challenge for real-time control and as a demonstration scenario for dynamic object manipulation. A control framework was presented that addresses the challenges related to environment perception, action planning, and motion control. Two of these challenges were discussed: first, a method for object tracking with high frame rates (appr. 150 Hz) was detailed. In addition to the tracking algorithm, the subsection discussed the trajectory prediction for spherical objects during free-flight phases and impact events. Second, a method for online motion planning was detailed which generates trajectories based on different selection criteria: distance from mechanical joint limits, dynamic manipulability measure, and energy consumption. Task-specific trajectory planning was shown for two-handed ball throwing. Together with a motion & interaction control and a dynamic force/torque observer, the two modules were integrated in a thorough control architecture for dynamic object manipulation. To validate the overall control design, first experimental results were reported and two dynamic manipulation tasks were considered: ball throwing and dribbling. It was demonstrated that dynamic dexterity can be realized with simple end effector structures. The trade-off for the reduced hardware requirements is an increased effort in motion planning, modeling, and environment perception. With these results in mind, dynamic manipulation is not understood to replace conventional manipulation. However, it will extend the capabilities and dexterity of robotic systems in many situations and thus broaden the areas of application, especially with regards to human-robot collaboration.

Acknowledgements The authors would like to thank Xihua Lu, Uwe Mettin, Kwang-Kyu Lee, Thomas Schau, Lorenz Kniep, Alexander Schmidts, and Haiyan Wu for their valuable contributions. The first author gratefully thanks the German National Academic Foundation for their support.

References

1. Mason M, Lynch K (1993) Dynamic manipulation. In: Proceedings of the IEEE/RSJ international conference on intelligent robots and systems (IROS), pp 152–159, 1993
2. Hodgins J, Raibert M (1990) Biped gymnastics. Int J Robot Res 9:115–128
3. Huang W(1997) Impulsive manipulation. PhD thesis, Carnegie Mellon University
4. Bühler M, Koditschek D, Kindlmann P (1988) A one degree of freedom juggler in a two degree of freedom environment. In: Proceedings of the IEEE international workshop on intelligent robots, pp 91–97, 1988
5. Schaal S, Atkeson CG (1993) Open loop stable control strategies for robot juggling. In: Proceedings of the IEEE international conference on robotics and automation (ICRA), pp 913–918, 1993
6. Ronsse R, Lefevre P, Sepulchre R (2005) Timing feedback control of a rhythmic system. In: Proceedings of the IEEE conference on decision and control and the European control conference (CDC-ECC), pp 6146–6151, 2005

7. Bühler M, Koditschek D, Kindlmann P (1994) Planning and control of robotic juggling and catching tasks. Int J Robot Res 13(2):101–118
8. Shiokata D, Namiki A, Ishikawa M (2005) Robot dribbling using a high-speed multifingered hand and a high-speed vision system. In: Proceedings of the IEEE/RSJ international conference on intelligent robots and systems (IROS), pp 2097–2102, 2005
9. Kato N, Matsuda K, Nakamura T (1996) Adaptive control for a throwing motion of a 2 dof robot. In: Proceedings of the international workshop on advanced motion control, pp 203–207, 1996
10. Katsumata S, Ichinose S, Shoji T, Nakaura S, Sampei M (2009) Throwing motion control based on output zeroing utilizing 2-link underactuated arm. In: Proceedings of the American control conference (ACC), pp 3057–3064, 2009
11. Lombai F, Szederkenyi G (2009) Throwing motion generation using nonlinear optimization on a 6-degree-of-freedom robot manipulator. In: Proceedings of the IEEE international conference on mechatronics, pp 1–6
12. Senoo T, Namiki A, Ishikawa M (2008) High-speed throwing motion based on kinetic chain approach. In: Proceedings of the IEEE/RSJ international conference on intelligent robots and systems (IROS), pp 3206–3211, 2008
13. Hove B, Slotine J-J (1991) Experiments in robotic catching. In: Proceedings of the American control conference (ACC), pp 380–386, 1991
14. Frese U, Bäuml B, Haidacher S, Schreiber G, Schäfer I, Hähnle M, Hirzinger G (2001) Off-the-shelf vision for a robotic ball catcher. In: Proceedings of the IEEE/RSJ international conference on intelligent robots and systems (IROS), pp 1623–1629, 2001
15. Bäuml B, Wimböck T, Hirzinger G (2010) Kinematically optimal catching a flying ball with a hand-arm-system. In: Proceedings of the IEEE/RSJ international conference on intelligent robots and systems (IROS), pp 2592–2599, 2010
16. Riley M, Atkeson C (2002) Robot catching: Towards engaging human-humanoid interaction. Autonomous Robots 12(1):119–128
17. Imai Y, Namiki A, Hashimoto K, Ishikawa M (2004) Dynamic active catching using a high-speed multifingered hand and a high-speed vision system. In: Proceedings of the IEEE international conference on robotics and automation (ICRA), pp 1849–1854, 2004
18. Senoo T, Namiki A, Ishikawa M (2006) Ball control in high-speed batting motion using hybrid trajectory generator. In: Proceedings of the IEEE International conference on robotics and automation (ICRA), pp 1762–1767, 2006
19. Furukawa N, Namiki A, Taku S, Ishikawa M (2006) Dynamic regrasping using a high-speed multifingered hand and a high-speed vision system. In: Proceedings of the IEEE international Conference on robotics and automation (ICRA), pp 181–187, 2006
20. Namiki A, Hashimoto K, Ishikawa M (2004) A hierarchical control architecture for high-speed visual servoing. Int J Robot Res 22:873–888
21. Gangloff JA, de Mathelin MF (2003) High-speed visual servoing of a 6-d.o.f. manipulator using multivariable control. Adv Robot 17(10):993–1021
22. Gemeiner P, Vincze M (2005) Motion and structure estimation from vision and inertial sensor data with high speed cmos camera. In: Proceedings of the IEEE international conference on robotics and automation (ICRA), pp 1853–1858, 2005
23. Zhang T, Liu X, Kühnlenz K, Buss M (2009) Visual odometry for the autonomous city explorer. In: Proceedings of the IEEE/RSJ international conference on intelligent robots and systems (IROS), pp 3513–3518, 2009
24. Uchiyama M, Kitagaki K (1989) Dynamic force sensing for high-speed robot manipulation using kalman filtering techniques. In: Proceedings of the IEEE international conference on decision and control (CDC), pp 2147–2152, 1989
25. Garcia J, Robertsson A, Ortega J, Johansson R (2006) Generalized contact force estimator for a robot manipulator. In: Proceedings of the IEEE international conference on robotics and automation (ICRA), pp 4019–4024, 2006
26. Garcia J, Robertsson A, Ortega J, Johansson R (2008) Sensor fusion for compliant robot motion control. IEEE Trans Robot 24(2):430–441

27. Bätz G, Scheint M, Wollherr D (2011) Towards dynamic manipulation for humanoid robots: Experiments and design aspects. Int J Humanoid Robots (accepted)
28. Haralick RM, Shapiro LG (2002) Computer and Robot Vision. Addison-Wesley, Reading, MA
29. Domenech A (2005) A classical experiment revisited: The bounce of balls and superballs in three dimensions. Am J Phys 1:28–36
30. Cross R (2002) Grip-slip behavior of a bouncing ball. Am J Phys 11:1093–1102
31. Yoshikawa T (1993) Analysis and control of robot manipulators with redundancy. In: Proceedings of the international symposium on robotics research, pp 735–747, 1983
32. Yoshikawa T (1985) Dynamic manipulability of robot manipulators. In: Proceedings of the IEEE international conference on robotics and automation (ICRA), pp 1033–1038
33. Bätz G, Mettin U, Scheint M, Wollherr D, Shiriaev A (2010) Ball dribbling with an underactuated continuous-time control phase: Theory & experiments. In: Proceedings of the IEEE/RSJ international conference on intelligent robots and systems (IROS), pp 2890–2895

Chapter 16
FlexRay Static Segment Scheduling

Martin Lukasiewycz, Michael Glaß, Jürgen Teich, and Paul Milbredt

16.1 Introduction

The FlexRay protocol was introduced by an international consortium including several car manufacturers to cope with growing real-time requirements of advanced driver assistance functions and safety functions in the automotive domain. The FlexRay protocol offers a static and dynamic segment with a high data rate of 10 Mbit/s. While the event-triggered dynamic segment is used mainly for diagnosis, maintenance, and calibration data, the time-triggered static segment might be used for critical data with strict real-time requirements. In addition to standard linear bus and star topologies, the FlexRay bus allows hybrid topologies including a dual channel mode to increase the reliability. However, in contrast to the prevailing CAN bus [4] in the automotive domain, the configuration of the FlexRay bus is significantly more complex: It requires a large set of parameters and a predefined schedule. This chapter introduces a scheduling concept for the static segment of the FlexRay based on the transformation to a two-dimensional bin packing problem.

M. Lukasiewycz (✉)
TU Munich, Germany
e-mail: martin.lukasiewycz@rcs.ei.tum.de

M. Glaß and J. Teich
University of Erlangen-Nuremberg, Germany
e-mail: michael.glass@cs.fau.de; juergen.teich@cs.fau.de

P. Milbredt
AUDI AG, Germany
e-mail: paul.milbredt@audi.de

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_16,

16.1.1 FlexRay Protocol

The FlexRay communication is organized in *cycles*, as illustrated in Fig. 16.1. Since each frame has exactly 6 cycle count bits, the cycles are numerated from 0 to 63 and subsequently start over with 0 again. Each cycle is divided into four segments of configurable duration:

1. The *static segment* enabling a guaranteed real-time transmission of critical data
2. The *dynamic segment* (optional) for low-priority and event-triggered data
3. The *symbol window* (optional) used to transmit special symbols and
4. The *network idle time* used to perform a clock synchronization.

The focus of this chapter is put on new techniques for scheduling the static segment. The static segment is made up of n equally sized *slots* where each one is uniquely assigned to one *node* (or none). One node, however, may occupy more than one slot. Each slot consists of a header and trailer segment and a payload segment that is statically configured to carry between 0 and 254 bytes. By a predefined schedule, each slot is filled with the communication data of the applications. As illustrated in Fig. 16.1, the FlexRay protocol uses a Time Division Multiple Access (TDMA) to multiplex the communication into different slots.

Figure 16.2 gives an overview of data transmission on the FlexRay bus in the automotive domain. The basic unit are the *signals*, e.g., physical data like the vehicle speed. The signals are packed to *messages* which are measured in bytes as the basic unit. The messages are packed to *frames*. Since all participants of a FlexRay bus are aware of the current cycle number, the cycle is used for a second multiplexing dimension as suggested in the AUTomotive Open System ARchitecture (AUTOSAR) FlexRay Interface Specification [1]. A *slot* may contain different frames for specific cycles. Here, exactly one (or no) frame is transmitted in a *slot* at one specific cycle to increase the utilization of the bus. Finally, the slots are added to the *schedule*.

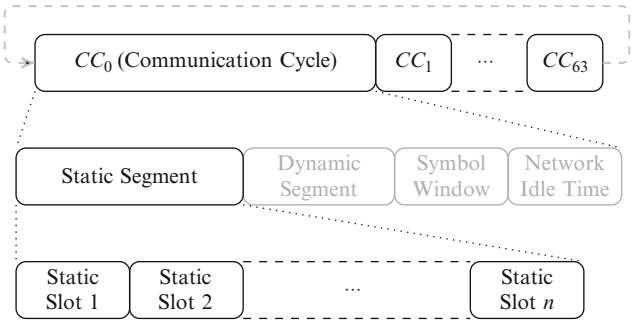


Fig. 16.1 The FlexRay communication protocol consisting of 64 communication cycles with a detailed illustration of the static segment

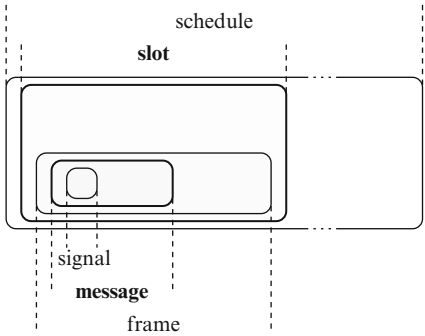


Fig. 16.2 Overview and terminology regarding the data transmission on the FlexRay bus in the automotive domain

In the work at hand, it is assumed that the basic communication units are messages, thus, the signals are already packed into messages. This is a common scenario, since messages are predefined by Electronic Control Unit (ECU) and gateway packing strategies. Thus, the goal is to pack messages into slots implicitly defining the frame packing. Note that a two-step approach where messages are first packed to frames and frames to slots might lead to suboptimal solutions with respect to the number of required slots.

16.1.2 Scheduling Requirements

In real-world implementations of the FlexRay bus, the periodic and safety-critical data is scheduled on the static time-triggered segment while the dynamic segment is mainly used for maintenance and diagnosis data [3, 17]. Though, in the first generation of the FlexRay bus in series-production vehicles, the static segment is not used at the full capacity [17], it is projected that the data volume on FlexRay buses will increase significantly in the future. Therefore, a schedule optimization that minimizes the number of used slots is necessary to allow a high flexibility for incremental schedule changes[1] and for future automotive networks with a higher data volume. Hence, an efficient schedule optimization of the static segment is the key to the success of the FlexRay bus.

The configuration of the FlexRay bus is defined by a large set of parameters. In particular, these parameters allow a configuration of the number and size of the slots in the static segment. Nevertheless, these values are mostly predefined by the manufacturer guided by existing data. For instance, the duration of a communication cycle is usually 5 ms due to the periods of the messages in the present automotive networks that are predominantly a multiple of 5 ms. For each message that is routed on the FlexRay bus, a fixed size in bytes is given and the minimal repetition is

[1] Incremental changes are common in the automotive area to decrease the testing exposure.

deduced from the period of the communication cycle and its own period, cf. [8]. In order to efficiently improve the tunable FlexRay parameters, fast scheduling techniques are necessary to allow for an effective parameter exploration.

16.1.3 AUTOSAR Interface Specification

As suggested in the AUTOSAR FlexRay Interface Specification [1] that is currently applied in all series-production vehicles, *cycle multiplexing* is used to increase the utilization of the FlexRay bus. An example of this *cycle multiplexing* for a single slot is illustrated in Fig. 16.3. The cycle multiplexing of messages is defined by the *base cycle* and the *cycle repetition*: The base cycle defines the offset in cycles for the first occurrence of the respective message. The cycle repetition denotes the frequency of a message among the communication cycles. The value of the cycle repetition is always a power of two $2^n, n \in \{0, \ldots, 6\}$ to allow a periodic occurrence in the 64 cycles. Thus, for a given base cycle b and repetition r, a message is existent in each communication cycle CC_i with

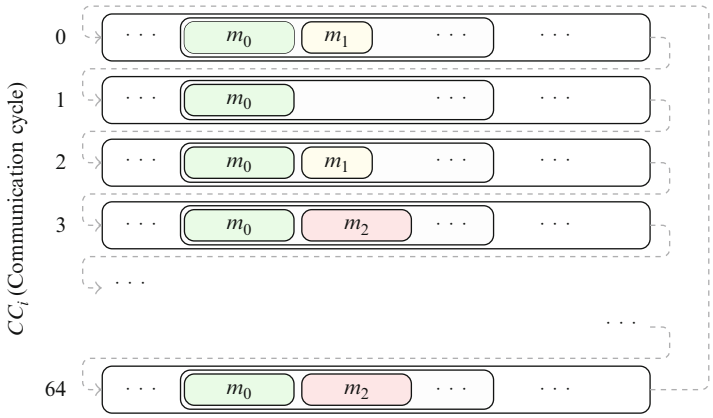
$$i = (b + r \cdot n)\%64 \text{ with } n \in \mathbb{N}_0. \tag{16.1}$$

Here, % is the modulo operation. An example of scheduling three messages m_0, m_1, and m_2 is given in Fig. 16.3. The base cycle values are 0 for m_0, 0 for m_1, and 3 for m_2. The repetition values are 1 for m_0, 2 for m_1, and 4 for m_2. Given a common duration of a single communication cycle of 5 ms, the message m_0 is sent each cycle with a period of 5 ms, the message m_1 each second cycle with a period of 10 ms, and the message m_2 each fourth cycle with a period of 20 ms. The cycle multiplexing technique maximizes the utilization of the static segment in compliance with the high requirements for reliability and robustness and, therefore, is integrated into real-world automotive implementations of the FlexRay bus based on the AUTOSAR specification.

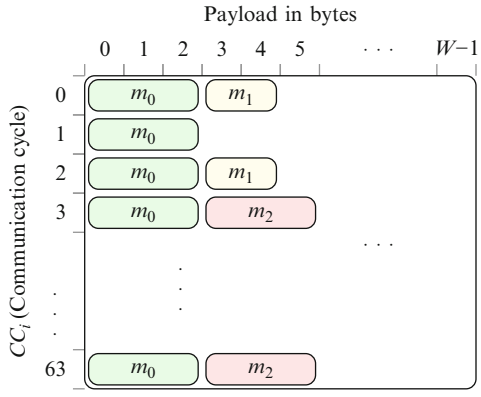
16.2 Related Work

The FlexRay specification [7] is under development by the *FlexRay Consortium* including *BMW*, *Daimler*, *General Motors*, and *Volkswagen*. Currently, the series-production vehicles using FlexRay are the *BMW X5*, *X6* and *7* series [2] and the *Audi A8* [10]. All these series-production vehicles are compliant with the FlexRay AUTOSAR Interface Specification [1]. Thus, this AUTOSAR specification is the de-facto industrial standard for the software specification of the FlexRay nodes.

Recent papers cover diverse FlexRay related topics. An introduction of the FlexRay protocol and the operating mode in real-world automotive systems is presented in [1,3,20]. In [16], a timing and performance analysis of FlexRay embedded



(a) Cycle multiplexing of three messages into a single slot in the context of the 64 communication cycles.



(b) Cycle multiplexing of three messages into a single slot in a two-dimensional representation.

Fig. 16.3 FlexRay cycle multiplexing of a single static slot. In order to achieve a high utilization, cycle multiplexing allows each slot to have an individual message scheduling in each communication cycle

systems is given, mostly focused on the dynamic segment. The determination and optimization of FlexRay schedules for the dynamic segment is discussed in [15]. The work in [11] presents a scheme for the acknowledgment and retransmission of data that is implemented on top of an existing FlexRay schedule in order to increase the reliability of FlexRay-based applications.

An approach that optimizes the static segment with a Genetic Algorithm (GA) is proposed in [6]. The approach in [21] introduces an Integer Linear Programming (ILP) approach for a proposed custom software architecture. In [24], the authors

present a Mixed Integer Linear Programming (MILP) approach for scheduling of messages and tasks in a synchronous architecture. However, these papers do not consider the AUTOSAR [1] software specification for the FlexRay bus and are, therefore, not applicable to solve current FlexRay scheduling problems in the automotive domain. Moreover, in the case of the exact ILP and MILP approaches, the scalability might form an obstacle for the applicability since the relevant papers only present relatively small case studies.

A recent work on scheduling the static segment is given in [8] that considers the optimization of the schedule with respect to cycle multiplexing. However, a predefined frame packing as described in [18, 19] is assumed. This approach is rather restrictive since a two-level packing of signals to messages and messages to frames as provided by AUTOSAR is common in the automotive domain. In contrast, the work at hand enables the scheduling of the messages directly into slots including the frame packing. Available tools for solving the scheduling problem are TTX PLAN [22] based on a heuristic approach and DAVINCI NETWORK DESIGNER FLEXRAY [23], a graphical user interface that only allows to build schedules manually. The scheduling algorithm of TTX PLAN is not published, but the experimental results in this work give evidence of an inferior behavior of TTX PLAN regarding runtime and quality of results.

One of the main contributions of the work at hand is a transformation scheme for the FlexRay scheduling problem into a special two-dimensional bin packing problem and its efficient solution. The general two-dimensional bin packing problem has been researched thoroughly, a brief summary is presented in [12]. Since an exact solution based on ILP results in a huge number of variables, cf. [5], heuristics are usually favored for unconstrained problems. However, in the presence of constraints like the level [13] or guillotine packing [17], an ILP can be formulated and solved efficiently. To the best of our knowledge, the special two-dimensional bin packing problem with individual level constraints, as presented in the work at hand, has not been topic of any research so far.

16.3 Schedule Optimization

The optimization flow as proposed in this work is illustrated in Fig. 16.4. The main goal is to minimize the number of used slots in order to maximize the utilization of the bus. Unused slots are still part of the final schedule, but these slots can be assigned to any ECU if the schedule is extended incrementally in further development. Moreover, a fast scheduling approach is advantageous, e.g., for the exploration of specific bus parameters.

First, the transformation of the original slot packing problem into a special *bin packing* problem is performed using the proposed transformation scheme. The bin packing is carried out in order to minimize the number of allocated slots. The work at hand introduces a fast heuristic and an exact ILP approach for this special bin packing problem. Finally, the transformation is inverted to convert the solution of

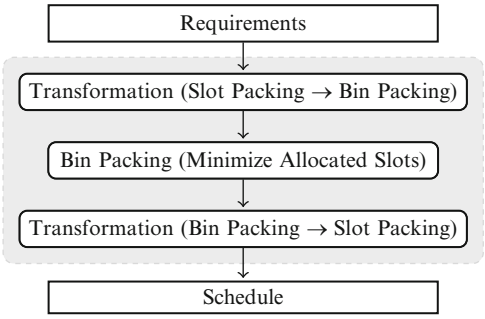


Fig. 16.4 Schedule optimization flow

the bin packing to a feasible FlexRay schedule. Since each slot is assigned to at most one ECU, the scheduling for each ECU is done independently and the slots are put together in the final schedule.

16.3.1 Problem Transformation

This section describes a one-to-one transformation between the slot packing problem that arises from the FlexRay cycle multiplexing and a special form of a two-dimensional bin packing problem. Since each slot corresponds to one bin, the transformation is presented for a single slot to a single bin and vice versa.

First, the general conditions for a feasible FlexRay slot packing are introduced: Each slot is defined by the payload size W (without the reserved load for the AUTOSAR specific update bits) and the number of cycles H which is 64 for the FlexRay bus. The set of messages is denoted M.
Each message $m \in M$ is defined by the following two values:

- $w_m \in \mathbb{N}$ - byte-length with $w_m \leq W$.
- $r_m \in \{2^n | n \in \{0, \ldots, 6\}\}$ - repetitions in the powers of two, defining the step-size for the multiplexing over the cycles. It holds that $r_m \leq H$.

For a feasible slot packing, two values for each message have to be determined:

- $x_m \in \mathbb{N}_0$ - the offset in bytes on the x-axis.
- $b_m \in \mathbb{N}_0$ - the base cycle that defines the offset on the y-axis. It holds that $b_m < r_m$.

A message is not allowed to exceed the slot ($x_m + w_m \leq W$) and no intersection between two messages is possible. The task of the transformation of the slot packing into a bin packing is to convert each message into a rectangular *element* and determine its position such that each feasible slot packing results in a feasible bin packing and vice versa. The bin size is the same as the slot size with the width W and height H. Also the position on the x-axis x_m and the width w_m for each element m correspond to the position and width of the message in the slot packing.

Therefore, the main task is to find a transformation that obtains the following two values for each element m:

- $y_m \in \mathbb{N}_0$ - the offset on the y-axis.
- $h_m \in \mathbb{N}_0$ - the height of an element.

The transformation for the height h_m is related to the repetition r_m:

$$h_m = \frac{H}{r_m} \tag{16.2}$$

$$r_m = \frac{H}{h_m} \tag{16.3}$$

Thus, the height of an element equals the number of appearances of the corresponding message in the H cycles.

Given $b_m < r_m$, it follows that in the bin packing problem, the position y_m is restricted to r_m individual levels, depending on the height of the element. It holds that the *level* of an element $l_m = y_m / h_m$ has to be in $\mathbb{N}_0$. This arises from the fact that two messages with the same repetition but different base cycles will never intersect each other. The same holds for elements of the same height but different levels.

Consider the following transformation function $t : \mathbb{N}_0 \times \mathbb{N} \to \mathbb{N}_0$:

$$t(x, y) = \begin{cases} 0, & x = 0 \\ t(\frac{x}{2}, \frac{y}{2}), & x \text{ is even} \\ t(\frac{x-1}{2}, \frac{y}{2}) + \frac{y}{2}, & x \text{ is odd} \end{cases} \tag{16.4}$$

with

$$y \in \{2^n | n \in \mathbb{N}_0\} \tag{16.5}$$

$$0 \leq x < y \text{ and } x \in \mathbb{N}_0 \tag{16.6}$$

It holds:

$$t(t(x, y), y) = x \text{ and } t(x, y) = t^{-1}(x, y) \tag{16.7}$$

The transformation function t directly transforms the level of an element l_m to the base b_m and vice versa, such that the following holds:

$$l_m = t(b_m, r_m) \text{ and } b_m = t(l_m, r_m) \tag{16.8}$$

and, thus,

$$y_m = l_m \cdot h_m = t(b_m, r_m) \cdot \frac{H}{r_m} \tag{16.9}$$

and

$$b_m = t(\tfrac{y_m}{h_m}, r_m) = t(\tfrac{y_m}{h_m}, \tfrac{H}{h_m}). \tag{16.10}$$

Thus, a transformation from the slot packing problem to a bin packing problem with individual level constraints based on the height of the elements is done

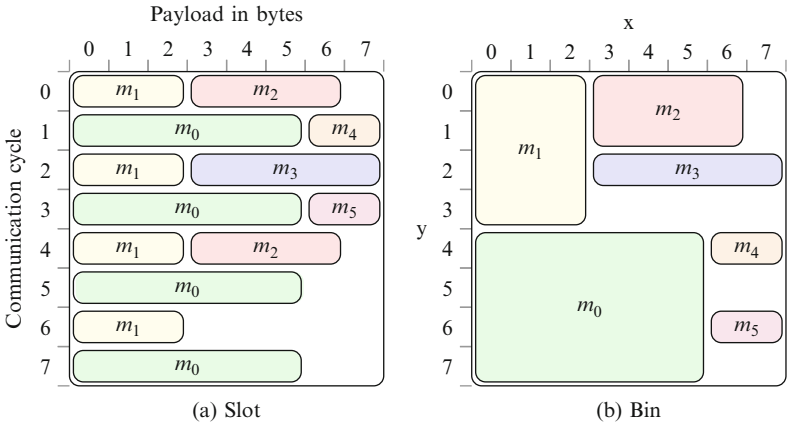


Fig. 16.5 Example of a transformation from a slot packing problem (**a**) to a bin packing problem (**b**) and vice versa

by applying (16.2) and (16.9) for each message. For the opposite direction, the transformation is performed by (16.3) and(16.10) for each element. An example for the transformation of a single slot is given in Fig. 16.5. A detailed proof for the correctness of this transformation is given in the following.

Theorem 16.1. *Two elements in the generated bin packing intersect if and only if the corresponding messages in the slot packing problem are conflicting, i.e., intersecting.*

Proof. For the x-transformation, this is trivial since the x-position and width w of the messages or elements, respectively, are identical. Thus, this has to be solved for the y-transformation as given by (16.2), (16.3), (16.9), and (16.10).

The function $t(x, y)$ from (16.4) performs a bitwise flip operation of the value x given in *Little-endian* encoding on its $\log_2 y$ bits. This satisfies the properties of function t in (16.7) under the assumptions in (16.5) and (16.6).
Two elements in a bin intersect if

$$\neg((y_m + h_m \leq y_{\tilde{m}}) \vee (y_m \geq y_{\tilde{m}} + h_{\tilde{m}})) \tag{16.11}$$

or by applying *De Morgan's law*

$$(y_m + h_m > y_{\tilde{m}}) \wedge (y_m < y_{\tilde{m}} + h_{\tilde{m}}). \tag{16.12}$$

Applying (16.2) to (16.12) results in

$$y_m - y_{\tilde{m}} > -\frac{H}{r_m} \text{ and } y_m - y_{\tilde{m}} > \frac{H}{r_{\tilde{m}}}. \tag{16.13}$$

A further transformation with (16.9) results in the following equations:

$$r_m \cdot t(b_{\tilde{m}}, r_{\tilde{m}}) < r_{\tilde{m}} \cdot (t(b_m, r_m) + 1) \tag{16.14}$$

$$r_{\tilde{m}} \cdot t(b_m, r_m) < r_m \cdot (t(b_{\tilde{m}}, r_{\tilde{m}}) + 1) \tag{16.15}$$

If (16.14) and (16.15) are both satisfied, the two elements are intersecting. On the other hand, if either (16.14) or (16.15) is violated, the two elements are not intersecting.

($\Rightarrow$) If the slot packing for two messages m and $\tilde{m}$ is conflicting, the corresponding elements in the bin packing intersect:

Without loss of generality, it is assumed that $r_m \leq r_{\tilde{m}}$. Two messages are conflicting in the slot packing if

$$\exists n \in \mathbb{N}_0 : b_m + r_m \cdot n = b_{\tilde{m}}. \tag{16.16}$$

This means, that the $\log_2 r_m$ least significant bits of b_m and $b_{\tilde{m}}$ are equal and, thus,

$$t(b_{\tilde{m}}, r_{\tilde{m}}) = \frac{r_{\tilde{m}}}{r_m}(t(b_m, r_m) + a) \tag{16.17}$$

holds with

$$0 \leq a \leq 1 - \frac{r_m}{r_{\tilde{m}}} < 1. \tag{16.18}$$

Here, a is the potential remainder by applying a shift of $\log_2 \frac{r_{\tilde{m}}}{r_m}$ bits. The upper bound 1 holds due to the problem-specific constraint $r_m \geq 1$ and the given assumption $r_{\tilde{m}} \geq r_m$.

Equation (16.17) is transformed to the following two equations:

$$r_m \cdot t(b_{\tilde{m}}, r_{\tilde{m}}) = r_{\tilde{m}} \cdot (t(b_m, r_m) + a) \tag{16.19}$$

$$r_{\tilde{m}} \cdot t(b_m, r_m) = r_m \cdot (t(b_{\tilde{m}}, r_{\tilde{m}}) - a \cdot \tfrac{r_{\tilde{m}}}{r_m}) \tag{16.20}$$

Given (16.19) with $a < 1$, (16.14) holds. At the same time, (16.15) holds due to (16.20) and $a \geq 0$. Thus, the elements intersect as required.

($\Leftarrow$) If the slot packing for two messages m and $\tilde{m}$ is not conflicting, the corresponding elements in the bin packing do not intersect:

Without loss of generality, it is assumed that $r_m \leq r_{\tilde{m}}$. Two messages are not conflicting in the slot packing if

$$\forall n \in \mathbb{N}_0 : b_m + r_m \cdot n \neq b_{\tilde{m}}. \tag{16.21}$$

This means, that the $\log_2 r_m$ least significant bits of b_m and $b_{\tilde{m}}$ are not equal and, thus, either

$$t(b_{\tilde{m}}, r_{\tilde{m}}) \leq \frac{r_{\tilde{m}}}{r_m}(t(b_m, r_m) - 1 + a) \tag{16.22}$$

or

$$t(b_{\tilde{m}}, r_{\tilde{m}}) \geq \frac{r_{\tilde{m}}}{r_m}(t(b_m, r_m) + 1 + a) \tag{16.23}$$

hold, both with a in the bounds from (16.18). From (16.22) it follows

$$r_{\tilde{m}} \cdot t(b_m, r_m) \geq r_m \cdot (\tfrac{r_{\tilde{m}}}{r_m}(1 - a) + t(b_{\tilde{m}}, r_{\tilde{m}})) \tag{16.24}$$

that violates (16.15) due to $\frac{r_{\tilde{m}}}{r_m}(1 - a) \geq 1$ that holds since $a \leq 1 - \frac{r_m}{r_{\tilde{m}}}$ as stated in (16.18). Equation (16.23) equals

$$r_m \cdot t(b_{\tilde{m}}, r_{\tilde{m}}) \geq r_{\tilde{m}} \cdot (t(b_m, r_m) + 1 + a) \tag{16.25}$$

that violates (16.14) due to $a \geq 0$. Thus, either (16.14) or (16.15) is violated and the elements do not intersect as required.

This proves (16.2) and (16.9). Equation (16.3) holds due to (16.2) and (16.10) holds due to (16.9) with the inverse properties of the t function given in (16.7).

16.3.2 Bin Packing

The task of a two-dimensional bin packing problem is to pack rectangular elements of different sizes defined by an individual width w and height h into a minimal number of rectangular bins without any intersection. Each bin has the fixed length W and height H. The transformation of the slot packing into a special two-dimensional bin packing problem determines a rectangular element m with the width w_m and height h_m for each message $m \in M$. The width of the bin equals the payload of a slot W and the height is the number of cycles which is 64 for FlexRay.

In contrast to the common two-dimensional bin packing, the transformed problem from the previous section contains two constraints:

1. Each element $m \in M$ has a height h_m that is a power of two, i.e., 2^n with $n \in \mathbb{N}_0$ and the bin height is at least the maximal height of all elements.
2. Each element $m \in M$ can be placed everywhere on the x-axis but only on a multiple of its height on the y-axis, i.e., $y_m = l \cdot h_m$ with the level $l \in \{0, \ldots, \frac{H}{h_m} - 1\}$.

This section introduces two optimization approaches for this specially constrained two-dimensional bin packing problem. The first approach is a fast greedy heuristic, the second is an efficient encoding as an ILP that allows to find the optimal solution.

16.3.2.1 Fast Greedy Heuristic

The presented bin packing problem can be solved by a fast greedy heuristic comparable to the approach presented in [17]. This heuristic is outlined in Algorithm 1.

Algorithm 1 Fast greedy heuristic for bin packing

```
1: S = {} //set of bins
2: for m ∈ M do
3:     for s ∈ S do
4:         if place(m, s) then
5:             continue with next m
6:         end if
7:     end for
8:     create new s and add it to S
9:     place(m, s)
10: end for
```

The algorithm starts with an empty set of bins S. Each element $m \in M$ is tried to be placed subsequently in a bin $s \in S$. Here, the function $place(m, s)$ is problem dependent and returns *true* if the placing is successful, and *false* otherwise. If an element is not placed in any of the allocated bins in S, a new bin s is allocated, added to S, and the element m is placed into this new empty bin.

Applied to the proposed special bin packing problem, the order of M influences the quality of the results. The elements in M are ordered first by their height h_m such that high elements are ordered to the front. The second criterion for the ordering of elements of the same height is the width w_m such that wide elements are ordered to the front. The function $place(m, s)$ tries to place each element m the most left void space in the bin s considering the individual level constraints of the proposed bin packing problem. This strategy tends to avoid the waste of void space of the bins. The complexity of this heuristic is polynomial.

16.3.2.2 Integer Linear Programming

Basic ILP. Solving a general two-dimensional bin packing problem with an ILP results in a high number of variables and constraints [5, 13]. However, the fact that each element has a height of a power of two and can only be placed on levels depending on its height can be exploited to deduce a compact and efficient ILP formulation with relatively few variables and constraints. The ILP formulation relies on the following binary variables:

- $\mathbf{m_{s,l}}$ - element m is placed at level l in bin s
- $\mathbf{s}$ - bin s is allocated (used)

The ILP is formulated as follows:

$$min \sum_{s \in S} \mathbf{s} \tag{16.26}$$

$$\forall\, m \in M : \sum_{s \in S} \sum_{l=0}^{\frac{H}{h_m}-1} \mathbf{m_{s,l}} = 1 \tag{16.27}$$

$$\forall\, s \in S, \{y = 0, \ldots, H-1\} : \sum_{m \in M} w_m \cdot \mathbf{m}_{\mathbf{s},\left\lfloor \frac{\mathbf{y}}{\mathbf{h_m}} \right\rfloor} \leq W \tag{16.28}$$

$$\forall\, m \in M, s \in S, \{l = 0, \ldots, \tfrac{H}{h_m} - 1\} : \mathbf{s} - \mathbf{m_{s,l}} \geq 0 \tag{16.29}$$

The objective function (16.26) of the ILP minimizes the number of allocated bins. Here, the set S has to contain a minimal number of bins that are necessary to solve the problem. This number is deduced from the presented fast heuristic approach. The constraints (16.27) state that each element m is placed in exactly one bin s at the specific level l. By adding the widths of the elements and restricting this sum by the width of a bin, the constraints (16.28) ensure that the size of each bin is not exceeded. The constraints (16.29) state that a bin s has to be allocated if at least one element m is placed in it.

Solving this ILP provides a bin s and level l for each element m (exactly one variable $\mathbf{m_{s,l}}$ is true). Placing the elements starting from the highest element to the most left void space in the bin s at the level l results in a feasible solution of the bin packing problem. This holds since the individual level constraints induce that each element that is sorted to the most left void space has at most one contact element on its left, see Fig. 16.5(b). Thus, the constraints (16.28) are sufficient to determine a feasible bin packing.

Though this is a very efficient ILP encoding in terms of the number of variables, one has to keep in mind that the complexity of an ILP is exponential in general. Moreover, in contrast to the heuristic approach, the presented ILP cannot be used incrementally, i.e., an already allocated bin cannot be filled with additional elements without moving the old elements.

Enhanced ILP. The stated ILP can be further improved by reducing the search space by applying domain-specific knowledge. First, the set of S is reduced by one bin, simplifying the ILP by omitting several variables and constraints. In case there exists no feasible solution of this simplified ILP, there also exists no feasible bin packing for $|S| - 1$ bins and, thus, the reference solution obtained by the heuristic is already optimal.

Furthermore, a lower bound for the objective is deduced by domain-specific knowledge to improve the runtime of the ILP. This lower bound is calculated as follows:

$$lb(M) = \frac{\sum_{m \in M} w_m \cdot h_m}{W \cdot H}. \tag{16.30}$$

The additional constraint

$$\sum_{s \in S} \mathbf{s} \geq \lceil lb(M) \rceil \tag{16.31}$$

sets the lower bound for the objective function. This constraint improves the runtime of the ILP: If the optimal solution is reached and equals the lower bound, the optimization process terminates immediately.

Regarding the problem of bin packing, a so-called *symmetry breaking* is applicable to reduce the search space. Consider the following one dimensional bin packing example: Given two elements m_1 and m_2 and two bins s_1 and s_2, there exist four possible distributions of the elements to the bins:

1. m_1, m_2 in s_1
2. m_1 in s_1 and m_2 in s_2
3. m_1, m_2 in s_2
4. m_1 in s_2 and m_2 in s_1

Since all bins have the same size and their order is negligible, there exists a symmetry between s_1 and s_2 regarding (1),(3) and (2),(4): Either (1) or (3) state that both elements are in the same bin, and correspondingly (2) or (4) state the both elements are in different bins. If the element m_2 is prohibited to be packed to bin s_2, (2) and (3) become invalid and the symmetry is broken. Thus, the search space is effectively reduced.

In order to generalize the symmetry breaking for the presented ILP formulation for the two-dimensional bin packing problem, two order functions for the elements and bins are used:

$$o : S \to \mathbb{N} \tag{16.32}$$

$$o : M \to \mathbb{N} \tag{16.33}$$

These functions assign to each element and bin, respectively, a unique integer value starting from 1 to $|M|$ and $|S|$, respectively. Given these functions, the symmetry breaking between bins is performed by adding the following constraints to the ILP formulation:

$$\begin{aligned} &\forall m \in M, s, s' \in S (s \neq s'), l = \{0, \ldots, \tfrac{H}{h_m} - 1\} \\ &\text{with } o(m) = o(s'), o(s') < o(s) : \mathbf{m_{s,l}} = 0 \end{aligned} \tag{16.34}$$

This ensures that an element is not allowed to be placed in a bin with a higher order.

Additionally, the two-dimensional bin packing leads to a possible horizontal symmetry through the middle of each bin. The symmetry breaking inside a bin is performed by adding the following constraints:

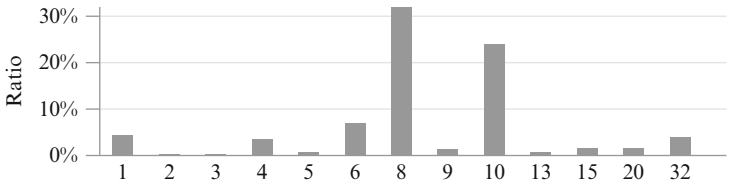
$$\begin{aligned} &\forall m \in M, s \in S, l = \{0, \ldots, \left\lceil \tfrac{H}{2 \cdot h_m} - 1 \right\rceil\} \\ &\text{with } o(m) = o(s) : \mathbf{m_{s,l}} = 0 \end{aligned} \tag{16.35}$$

This ensures that all elements with the same order value as the order value of a bin can only be placed in the lower half of this bin. Note that each symmetry breaking effectively accelerates the ILP solving without excluding optimal solutions.

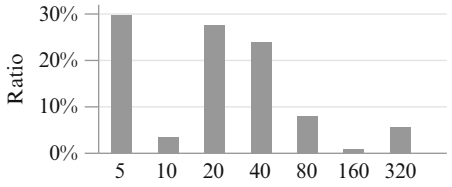
16.4 Case Study

A real-world example consisting of a FlexRay bus with 8 ECUs and an overall number of 220 messages is carried out as a case study to show the applicability of the proposed methods. The distribution of the sizes and periods of the messages is illustrated in Fig. 16.6. The messages are highly heterogeneous in terms of their period and size. The parameters of the FlexRay bus are predefined such that the static segment consists of 62 slots with each slot carrying a payload of 42 bytes. Effectively, only 41 bytes are used since one byte is reserved for the update bits. The duration of the communication cycle is 5 ms. The experiments were carried out on an Intel Pentium 4 3.20 GHz machine with 512 MB RAM. The ILP solver for the bin packing was the CPLEX solver in the version 10.5 [9]. Currently, the only available automatic scheduling approach compliant with the AUTOSAR specification is the commercial tool TTX PLAN [22]. For the introduced case study, a reference solution obtained by TTX PLAN is available.

The time for the transformation into a bin packing problem and vice versa might be omitted since it is negligibly small (less than 1 ms). The results for the case study are given in Table 16.1. The runtimes for the heuristic and ILP approach for the case study is a fraction of a second. The row ILP* in Table 16.1 shows the results that were obtained by the ILP approach without the presented enhancements.



(a) Distribution of the message sizes in bytes.



(b) Distribution of the message periods in milliseconds.

Fig. 16.6 Distributions for message sizes and periods of the real-world case study

Table 16.1 Results for the case study

Method	runtime [s]	slots
Heuristic	**0.065**	**27**
ILP* (CPLEX)	25.7	**27**
ILP (CPLEX)	0.080	**27**
TTX Plan	360	29

Here, the runtime is significantly higher with 25.7 s. In fact, the ILP enhancements are always advantageous since there arises no additional overhead. The heuristic and ILP approach both deliver solutions with 27 allocated slots. Moreover, the ILP approach proves that 27 allocated slots is the optimal solution.

The proposed algorithms were compared to a result obtained by the commercial available scheduling tool TTX PLAN [22] that is based on an undisclosed heuristic approach with a prospected polynomial scalability of the runtime. While the commercial tool returns a schedule with 29 allocated slots, the heuristic and the ILP improve this value by two slots, which is significant for a real-world application. These results show that the commercial tool delivers an inferior result in a comparatively large amount of time, in particular, in 6 min. The runtime of the commercial tool TTX PLAN and the presented approaches differs by four orders of magnitude. Since scheduling is typically just one of several tasks in a complete design flow, this enables a Design Space Exploration (DSE) [14] with reasonable runtimes using the proposed algorithms.

16.5 Summary

This chapter presented a scheduling optimization scheme for the static segment of the FlexRay bus in compliance with the AUTOSAR specification. First, the problem is transformed into a special two-dimensional bin packing problem using a proposed one-to-one transformation scheme. This constrained bin packing problem is solved either with a presented heuristic approach, delivering good results in a relatively small amount of time, or an introduced efficient ILP approach that delivers the optimal solution. The results of the case study show that the heuristic and ILP approach are superior to a commercial tool in runtime and quality. The scalability analysis studies the applicability of the proposed methods.

References

1. AUTOSAR: Specification of the FlexRay Interface Version 3.0.2 (2008). URL Http://www.autosar.org
2. Berwanger J, Peteratzinger M, Schedl A (2008) FlexRay startet durch. FlexRay-Bordnetz für Fahrdynamik und Fahrerassistenzsysteme (in German). In: Elektronik Automotive:

Sonderausgabe 7er BMW. Available at URL http://www.elektroniknet.de/home/automotive/bmw-7/flexray-startet-durch/
3. Broy J, Müller-Glaser KD (2007) The impact of time-triggered communication in automotive embedded systems. In: Proceedings of the international symposium on industrial embedded systems (SIES 2007), pp 353–356
4. CAN: Controller Area Network. URL Http://www.can.bosch.com/
5. Christofides N, Hadjiconstantinou E (1995) An exact algorithm for orthogonal 2-D cutting problems using guillotine cuts. Euro J Operat Res 83(1):21–38
6. Ding S, Murakami N, Tomiyama H, Takada H (2005) A GA-based scheduling method for flexRay systems. In: Proceedings of the international conference on embedded software (EMSOFT 2005), pp 110–113
7. FlexRay Consortium: FlexRay Communications Systems – Protocol Specification Version 2.1 Rev. A. 2005 URL Http://www.flexray.com
8. Grenier M, Havet L, Navet N (2008) Configuring the communication on flexRay: The case of the static segment. In: Proceedings of the 4th European congress on embedded real time software (ERTS 2008) (2008)
9. ILOG: CPLEX. URL Http://www.ilog.com/products/cplex/, Version 10.5
10. Kötz J, Poledna S (2008) Making flexRay a reality in a premium car. In: Proceedings of the society of automotive engineers international 2008 (SAE 2008)
11. Li W, Di Natale M, Zheng W, Giusto P, Sangiovanni-Vincentelli A, Seshia S (2009) Optimizations of an application-level protocol for enhanced dependability in flexRay. In: Proceedings of the conference on design, automation and test in Europe (DATE 2009), pp 1076–1081
12. Lodi A, Martello S, Vigo D (2002) Recent advances on two-dimensional bin packing problems. Discrete Appl Math 123(1-3):379–396
13. Lodi A, Martello S, Vigo D (2004) Models and bounds for two-dimensional level packing problems. J Combinatorial Opt 8(3):363–379
14. Lukasiewycz M, Glaß M, Milbredt P, Teich J (2009) FlexRay schedule optimization of the static segment. In: Proceedings of the 7th IEEE/ACM international conference on hardware/software codesign and system synthesis (CODES+ISSS 2009), pp 363–372
15. Pop T, Pop P, Eles P, Peng Z (2007) Bus access optimisation for FlexRay-based distributed embedded systems. In: Proceedings of the conference on design, automation and test in Europe (DATE 2007), pp 51–56
16. Pop T, Pop P, Eles P, Peng Z, Andrei A (2006) Timing analysis of the FlexRay communication protocol. In: Proceedings of the 18th euromicro conference on real-time systems (ERTS 2006), pp 203–216
17. Puchinger J, Raidl GR (2007) Models and algorithms for three-stage two-dimensional bin packing. Euro J Operat Res 127(3):1304–1327
18. Saket R, Navet N (2006) Frame packing algorithms for automotive applications. J Embedded Comput 2(1):93–102
19. Sandstrom K, Norstom C, Ahlmark M (2000) Frame packing in real-time communication. In: Proceedings of the seventh international conference on real-time computing systems and applications (RTCSA 2000), pp 399–403
20. Schedl A (2007) Goals and architecture of FlexRay at BMW. In: Slides presented at the Vector FlexRay Symposium (2007). Available at URL https://www.vector-worldwide.com/
21. Schmidt K, Guran Schmidt E (2009) Message scheduling for the FlexRay protocol: The static segment. IEEE Trans Vehicular Technol 58(5):2170–2179
22. TTTech: TTX Plan. URL Http://www.tttech-automotive.de/
23. Vector: DaVinci Network Designer FlexRay. URL Http://www.vector.com/
24. Zeng H, Zheng W, Di Natale M, Ghosal A, Giusto P, Sangiovanni-Vincentelli A (2009) Scheduling the FlexRay bus using optimization techniques. In: Proceedings of the 46th conference on design automation (DAC 2009), pp 874–877

Chapter 17
Real-Time Knowledge for Cooperative Cognitive Automobiles

Christoph Stiller and Oliver Pink

17.1 Real-Time Requirements in Automotive Applications

We are currently witnessing a rapid growth in the number of sensors in our automobiles. Accompanied by a trend towards higher data rate, storage capabilities, and processing power per sensor it seems safe to state that the role of information acquisition and processing is of increasing importance to the automotive domain. While radar, lidar, and video sensors were only introduced to selected upper class automobiles around the turn of the millennium,[1] such sensors are readily available for medium sized vehicles by now and are expected to become standard in any vehicle in the not so far future. Combined with vehicular communication systems it is expected that this trend will not only show quantitative effects on driving comfort, but in the long term will provide a totally new quality of traffic operation including concerted navigation for safe, comfortable, and efficient driving.

We define *cognitive automobiles* as automobiles that carry on-board sensors to acquire and process information about their traffic environment and are able to decide and conduct appropriate actions based on that information. Cognitive automobiles may provide support to human drivers through information, warning or sharing the control task between drivers and the vehicle, or in the long term,

[1]The European Automotive industry introduced radar sensors in the Mercedes-Benz S-Class and in Jaguar's XKR for *Adaptive Cruise Control* in 1999, followed by BMW's 7-series in early 2000.

C. Stiller (✉)
Institute of Measurement and Control Systems, Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
e-mail: stiller@kit.edu

O. Pink
Institute of Measurement and Control Systems, Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
e-mail: pink@kit.edu

S. Chakraborty and J. Eberspächer (eds.), *Advances in Real-Time Systems*,
DOI 10.1007/978-3-642-24349-3_17,

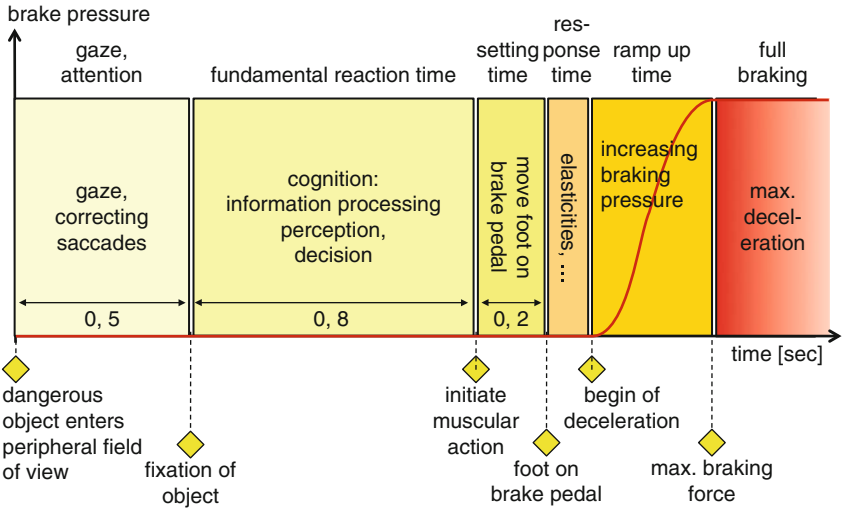


Fig. 17.1 Exemplaric reaction of a human driver in an immanent collision situation

autonomously decide for automatic vehicle control that does not require any human intervention.

The potential of cognitive automobiles to traffic safety may be illustrated at the example of a typical reaction of a human driver in an immanent collision situation as depicted in Fig. 17.1. After a new object enters the peripheral field of view, the human needs about 0.5 s to direct the gaze towards it, i.e. to direct the foveal sight onto it. About another 0.8 s are needed for the cognition process, i.e. to recognize the danger and to decide for a braking manoeuvre. Muscular action is initiated to move the foot from the accelerator to the brake pedal. Finally elasticities and slackness have to be compensated for before the vehicle finally begins deceleration. A typical time from presence of the object to full deceleration may be 1.9 s. Even when this duration is somewhat reduced when the object has been visually tracked before becoming a threat, this figure clearly reveals the potential to reduce the deceleration time through cognitive automobiles. In the European Union's e-Safety Report the authors project a reduction of fatal accidents by 50% through a gain of 0.5 s in critical situations [1].

While human drivers do definitely not belong to the class of real time systems, cognitive automobiles may reach such properties. Figure 17.2 sketches the above braking manoeuvre by a cognitive automobile. With a 360° surround view all objects in the field of view are permanently monitored and tracked, which typically happens far before entering a human's field of view. Furthermore, vehicular communication allows for telematic awareness, i.e. automobiles get information about other traffic participants that are not yet visible at all. The cognition process in machine vision can be speed up through usage of more processing power yielding time frames that go below 0.1 s. Muscular motion is completely avoided and

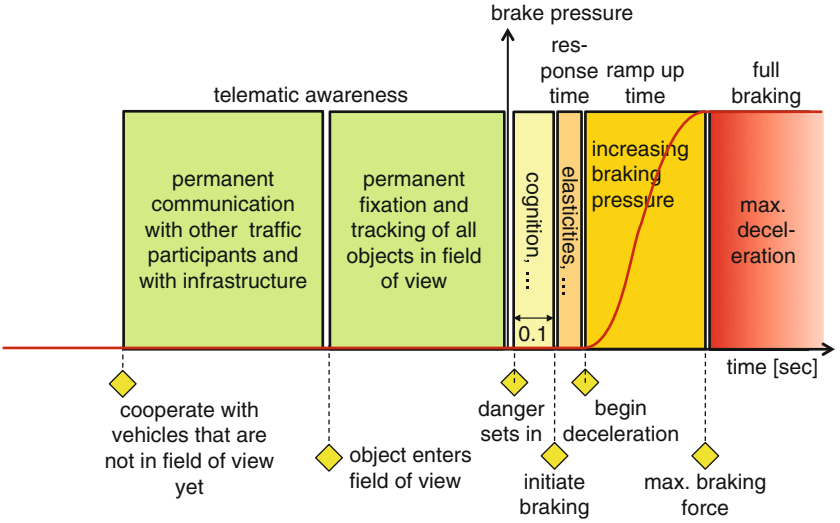


Fig. 17.2 Potential reaction of a cognitive automobile in an immanent collision situation, cf. Fig. 17.1

elasticities and slackness may be reduced through preconditioning of the braking system. In total, cognitive automobiles may react up to 1.5 s faster than a human which would reduce the vast majority of accidents.

While in the long term cognitive automobiles are expected to react safer and faster than human drivers, the basic decision modules may resemble human behaviour to a large extent. Figure 17.3 depicts a psychological model for human behaviour (cf. [11]) which may be related to a decision concept applied in automated driving as depicted in Fig. 17.4, e.g. [7, 10]. At the lowest level of skill-based behaviour, features are extracted from the scene via human senses or automotive sensors, respectively. These are implicitly mapped to appropriate action without requiring a conscious decision. An example for such a task is given by lane keeping on a highway. In technical systems, stabilization is achieved by low level controllers. At the intermediate level of rule-based behaviour, feature patterns are recognized and associated to previously known situations. Then a sequence of behaviour is recalled that has led to a successful result in related situations. In a technical system, tactical decisions may also follow predefined rules. On the knowledge-based behaviour level a conscious decision process takes place. Alternative behavioural options are established and a planning process predicts the result of each option. Finally, the option is chosen whose predicted result maximizes some quality measure. In a technical system, such strategic decisions are often conducted by optimizing over millions of alternative trajectories [7, 14]. It is worthwhile noting that the notion of real-time for decision making in autonomous automobiles does not generally refer to time periods in the range of some milliseconds. Such reaction times are required for some stabilization tasks as well as for some tactical decisions

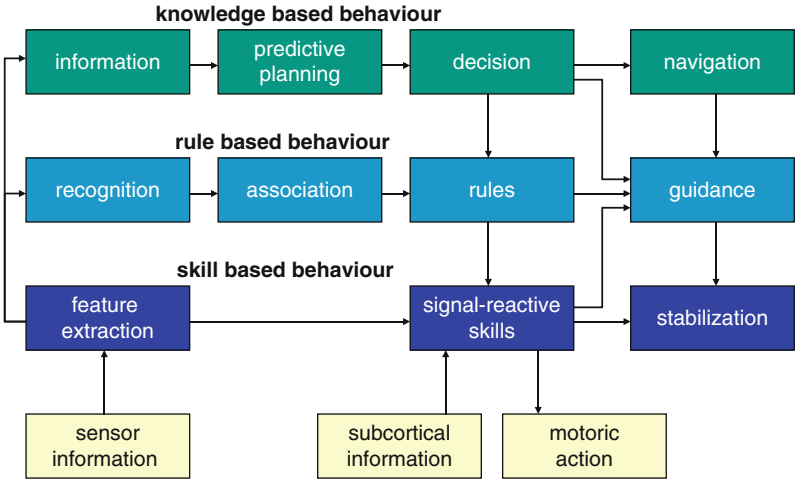


Fig. 17.3 Model for human driving behaviour

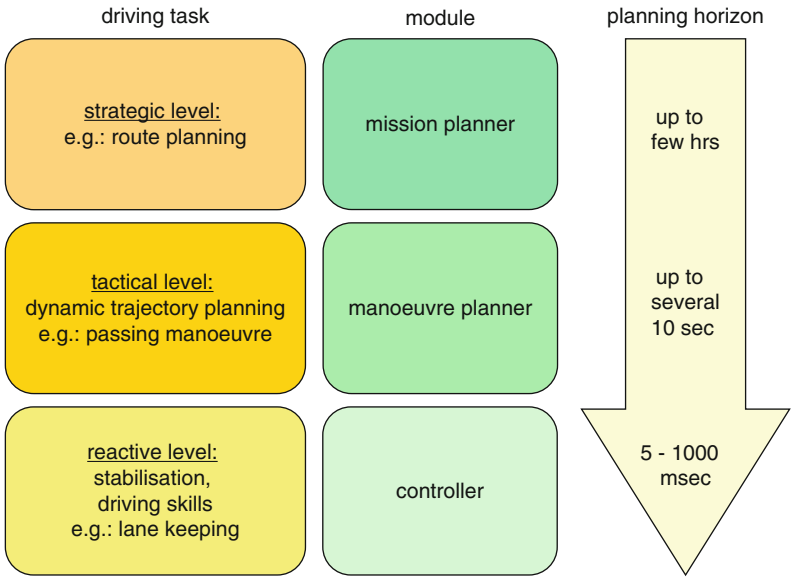


Fig. 17.4 Automated driving decision levels

in unexpected emergency situations. However, typical decisions on a tactical level evolve from perceiving a scene over a longer period in time that is in the range of a second or more. The final decision for an emergency manoeuvre may then be triggered when the trajectory of some traffic participants enters a safety margin. Finally, on a strategic level, driving decisions may be updated at very low rates.

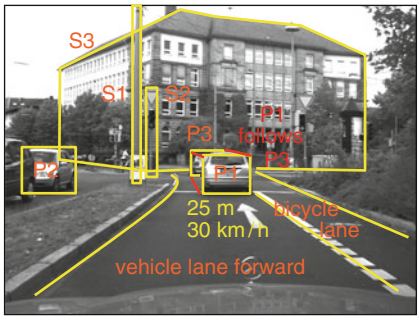


Fig. 17.5 Metric, symbolic, and conceptual knowledge for cognitive automobiles

17.2 Knowledge Representation for Automobiles

The knowledge basis for any decision is the perception of the actual situation. As illustrated in Fig. 17.5, driving – whether by a human or a cognitive machine – involves knowledge representation in various forms. *Metric knowledge*, such as the lane geometry and the position or velocity of other traffic participants is required to keep the vehicle on the lane at a safe distance to others. *Symbolic knowledge*, e.g. classifying lanes as either "vehicle lane forward", "vehicle lane rearward", "bicyle lane", "walkway", etc. is needed to conform with basic rules. Finally, conceptual knowledge, e.g. specifying a relationship between other traffic participants allows to anticipate the expected evolution of the scene to drive foresightedly.

The Transregional Collaborative Research Centre 28 *"Cognitive Automobiles,"* has focused on systematic and interdisciplinary research in this field. Founded in January 2006, partners from Karlsruher Institut für Technologie, Fraunhofer Institut IOSB Karlsruhe, Technische Universität München, and Universität der Bundeswehr München have investigated methods for machine cognition of mobile systems as the basis for automated machine behaviour [13, 15]. The partners have not only conducted analytic research accompanied by closed-loop simulations, but have also integrated their findings into experimental autonomous vehicles. These have successfully participated in international competitions such as the Grand and Urban Challenge [6, 7, 10].

17.3 Cooperative Automobiles

Cognitive automobiles shall exchange their knowledge on the traffic scene with another and negotiate cooperative behaviour. Through broader and more reliable information, cooperativity among automobiles enhances traffic safety and traffic flow at the same time.

Figure 17.6 depicts a mixed traffic scenario with cognitive automobiles as well as automobiles that are neither equipped with any sensors nor with vehicular

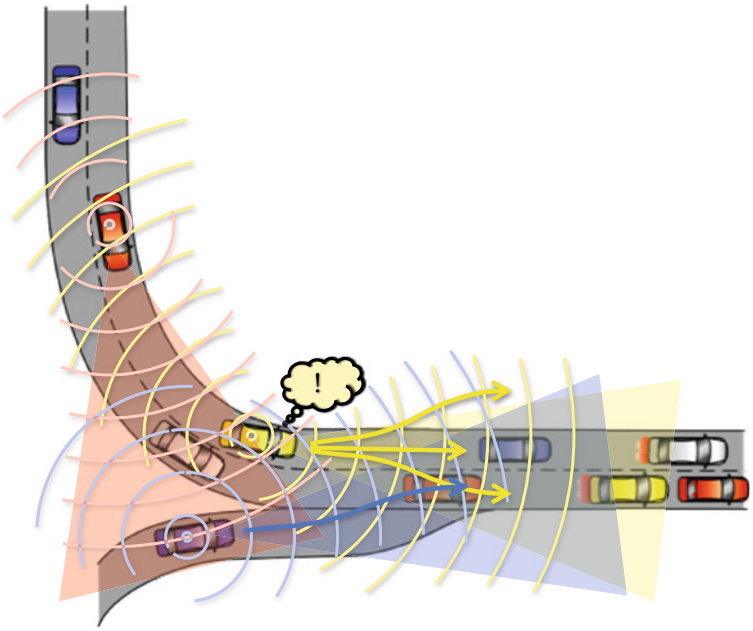


Fig. 17.6 Cooperative cognitive automobiles in mixed traffic

communication devices. Cognitive automobiles are equipped with video, lidar, or radar sensors and a communication device that allows the exchange of information with the infrastructure or with other vehicles. Sensor data analysis is performed to gather an instantiated model of the real world. Metric, symbolic, and conceptual information is used as a basis to continuously decide for appropriate driving commands. When the traffic scenario involves several cognitive automobiles, they communicate their knowledge and driving intentions one to another. Furthermore the automobiles may negotiate coordinated driving trajectories that are beneficial to the community of traffic participants. Simple examples for cooperative driving are coordinated trajectories at intersections or coordinated lane change manoeuvres.

17.3.1 Cooperative Perception

As indicated in Fig. 17.6 cooperative perception extends the field of view for each automobile to the union set of the individual fields of view. This extension is termed a *telematic horizon*. It provides information about the traffic scene long ahead, in blind spots, and in areas that may be occluded for some of the automobiles.

As depicted in Fig. 17.7, cooperative perception involves several stages that must be considered to access timing and uncertainties of information that is communicated from other vehicles. Let vehicle C' observe an obstacle in its

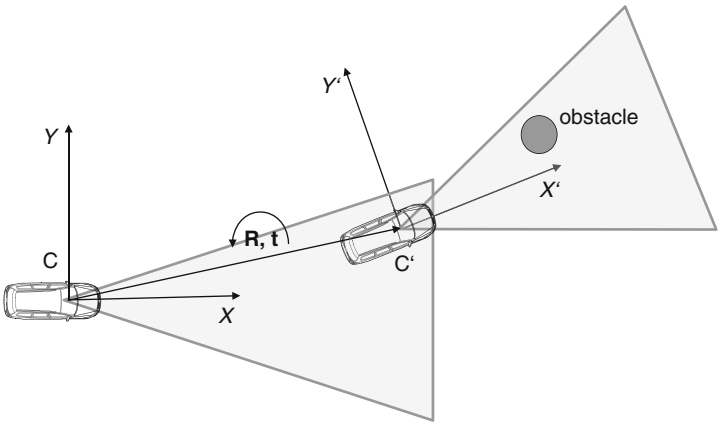


Fig. 17.7 An obstacle observed by C' is communicated to C via coordinate transformation $\mathbf{R}, \mathbf{t}$

coordinate frame at time t' and at position[2] $\mathbf{X}' = (X', Y', Z')^T$. The position uncertainty is expressed by the covariance $\Sigma_{\mathbf{X}'}$. This information is transmitted to vehicle C within a communication time t_c. This vehicle processes this information within time t_p. Processing includes coordinate transformation of the position to ego-coordinates $\mathbf{X} = (X, Y, Z)^T$. This transformation requires knowledge on the relative pose of C' wrt C expressed through the rotations $\boldsymbol{\omega} = (\omega_X, \omega_Y, \omega_Z)^T$ about the $X-$, $Y-$, and $Z-$axis, and the translation $\mathbf{t} = (t_X, t_Y, t_Z)^T$, respectively. Let their uncertainties be denoted by $\Sigma_{\boldsymbol{\omega}}$ and $\Sigma_{\mathbf{t}}$ and let all uncertainty vectors be mutually uncorrelated. The coordinate transform yields the position estimate in the ego coordinate system

$$\mathbf{X} = \mathbf{R}\mathbf{X}' + \mathbf{t}\,, \tag{17.1}$$

where $\mathbf{R} = \mathbf{R}(\boldsymbol{\omega})$ denotes the rotation matrix associated with $\boldsymbol{\omega}$. This information is available at time

$$t = t' + t_c + t_p\,. \tag{17.2}$$

Furthermore, the uncertainties in obstacle position and relative pose accumulate to

$$\Sigma_{\mathbf{X}} = \mathbf{R}\Sigma_{\mathbf{X}'}\mathbf{R}^T + [\mathbf{X}']_\times \Sigma_{\boldsymbol{\omega}} [\mathbf{X}']_\times^T + \Sigma_{\mathbf{t}} \tag{17.3}$$

$$\text{with } [\mathbf{X}']_\times = \begin{pmatrix} 0 & -Z' & Y' \\ Z' & 0 & -X' \\ -Y' & X' & 0 \end{pmatrix}\,.$$

[2]For the sake of simplicity, we restrict our consideration to a position estimate. Extension to other information like orientation or velocity is straightforward.

In practice, the second term may become dominant for distant objects. As information communicated from other vehicles may thus be deteriorated by additional time delay and pose uncertainty, the information that is selected for communication and the reference frame for this information must be carefully chosen. In particular, geo-referencing of information may significantly improve on this situation [8, 16].

17.3.2 Cooperative Behaviour

Based on all available sensor information, an autonomous vehicle has to assess the situation and determine a list of possible actions, from which an optimal manoeuvre is chosen based on a given set of quality criteria, such as e.g. collision avoidance, travel time, or fuel consumption. Beyond the exchange of sensor data cooperative vehicles will exchange information about this intended manoeuvre and last not least may negotiate possible alternative behaviours with another. Driving negotiated trajectories is termed *cooperative behaviour* and generally yields improved safety and traffic flow for all participants [13].

A possible way to implement cooperative behaviour in a convoy driving scenario is to exchange a prioritized list of planned vehicle trajectories and/or the current vehicle state, such as

- Accelerating, decelerating, constant speed
- Merging, intersection approach, lane change, vehicle following
- Convoy driving as lead/following vehicle.

Consideration of this information in vehicle control improves on the safety and flow of the complete convoy.

Another example for cooperative behaviour generation is illustrated in Fig. 17.8. Let us assume the yellow vehicle has three alternative trajectories, of which, given only the vehicle's own sensor data, a lane change to the right would be the best alternative. However, this would conflict with the intended merge manoeuvre of the blue vehicle coming from the lower left side. If the two vehicles are able to communicate their intended trajectories and possible alternatives, they can negotiate a behaviour which is acceptable for both vehicles: The yellow vehicle stays on the left lane and the blue vehicle merges into the right lane.

According to the decision levels in Fig. 17.4, cooperative negotiation of vehicle behaviour is typically performed on the tactical layer, which requires significantly lower data rates than an information exchange on sensor level. This reduces communication bandwidth and permits higher latencies and therefore reduces the required quality of service (QoS) of vehicle-to-vehicle communication [9].

On the other hand, a lower QoS and consequently reduced reliability requires additional concepts to ensure safe vehicle behaviour. This includes a reactive system for collision avoidance by emergency braking or escape manoeuvres that solely relies on on-board sensor data of the vehicle. The minimum safety distances for

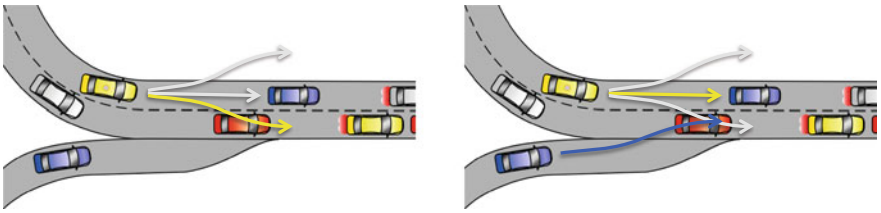


Fig. 17.8 Example scenario for cooperative behaviour

cooperative driving are therefore limited by the perception update rates and bus latencies as introduced in Sect. 17.1.

17.4 Hardware and Software Architecture

Following the considerations from the previous sections, real-time requirements in cognitive automobiles are dependent on the respective decision level and range from several milliseconds for the reactive level up to few hours for strategic mission planning. This induces different limitations for the hardware and software framework for a cognitive automobile. While the reactive level, e.g. vehicle control and low-level collision avoidance is mainly influenced by the sensor data processing time and sensor update rates, the update cycles of higher-level modules such as route planning or scene understanding are mainly limited by the available computing performance.

The large difference in update rates requires a modular software design for perception and decision making which is closely related to the decision levels in Fig. 17.4. Lower-level decision levels such as collision avoidance should only rely on low-level perception, e.g. an occupancy grid, which can be computed at high update rates, while higher-level decision modules can make use of more complex environmental representations.

On the behavioural side, the software architecture should allow lower-level modules to override the output of higher-level modules, i.e. short-term collision avoidance has priority over mission planning and path planning over a prolonged period. An example for such a software architecture is given in Fig. 17.9.

Depending on the task of a software module, real-time capability is limited by different hardware constraints, e.g.

- Available computing power
- Available communication bandwidth
- Sensor update cycles.

Lower-level systems such as vehicle control are mainly influenced by the communication bandwidths and resulting bus latencies whereas low data rates of

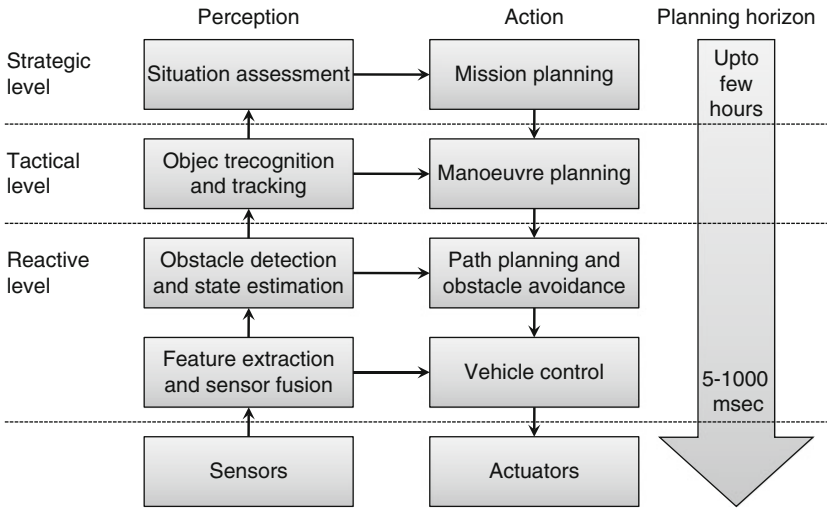


Fig. 17.9 Hierarchical software architecture for cooperative cognitive automobiles

the required sensors allow for high update rates. Higher-level tasks such as scene understanding on the other hand are mainly limited by the available computing power, while bus latencies and sensor update cycles are small compared to the processing time of these tasks. Especially for systems that make use of sensors with high data rates such as image processing, all three limitations have to be taken into account: Typical processing times, sensor update cycles and bus latencies lie all in the same order of magnitude. To obtain an overall image processing time of 0.1 s as discussed in Sect. 17.1, a good combination of image resolution, communication bandwidth and computing power has to be chosen.

To keep the inter-process communication overhead low and to gain optimal use of the available computing power, a centralized hardware architecture is advantageous, where computing time is distributed among all processes while real-time requirements especially of the lower-level tasks are guaranteed. In such a centralized system, all sensor information is available to all modules at any time. Furthermore, a centralized system offers high bandwidths for inter-process communication.

17.5 Experimental Cognitive Automobiles

Within the Karlsruhe-Munich Transregional Collaborative Research Centre 28 *'Cognitive Automobiles,'* researchers from several disciplines are working on the different challenges on the way to a cognitive automobile. The experimental validation of the findings requires a unified hardware and software framework for

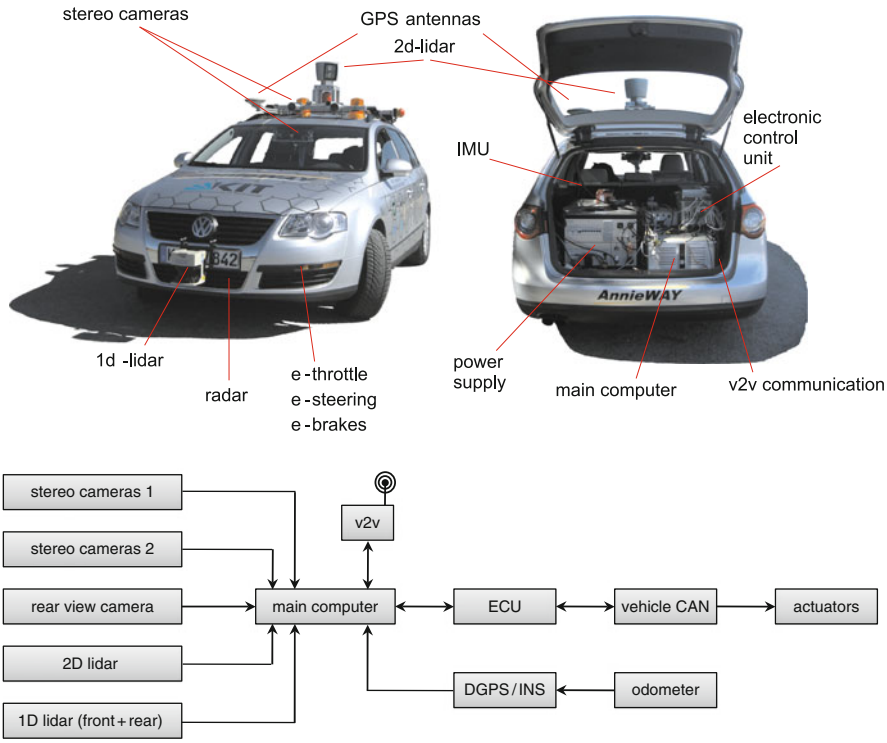


Fig. 17.10 Experimental hardware setup for cooperative cognitive automobiles

all partner institutions [4, 17]. Integrating the solutions of different research groups into one experimental platform requires simple and well-defined interfaces of the different modules.

Based on the hardware architecture depicted in Fig. 17.10, a total of six experimental cognitive automobiles were set up within the collaborative research center [5, 12, 15].

To ensure real-time requirements and low latencies, vehicle control is performed on a dedicated dSpace AutoBox which directly communicates with the actuators over the vehicle CAN. All other perception and planning modules as well as sensor data acquisition are performed by a single multicore multiprocessor computer system which delivers sufficient computing power to host all processes and at the same time provides low latencies and high bandwidth for inter-process communication.

The hardware architecture is complemented with a real-time capable software architecture as depicted in Fig. 17.11. Its central element is a real-time database for information exchange (KogMo-RTDB [3, 4]). It allows parallel operation of real-time processes and non-real-time processes at any update rate. As all inter-process communication is performed via the database, all modules have a centralized view on all available information at every time. The different driving and perception tasks

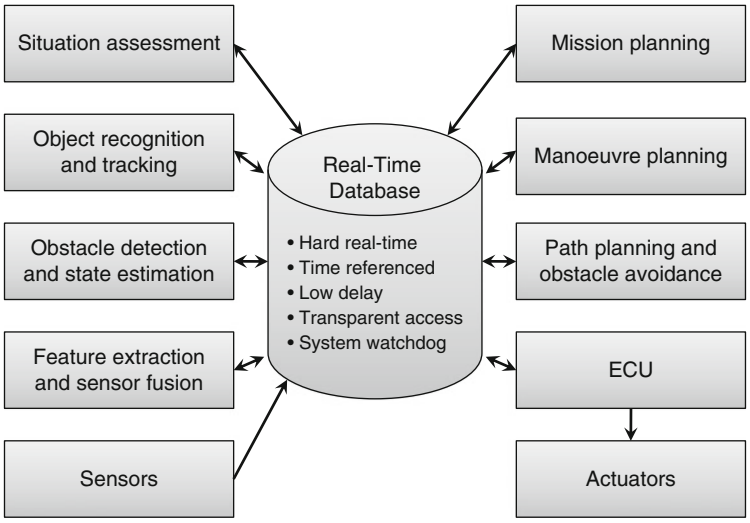


Fig. 17.11 Software architecture for cooperative cognitive automobiles with centralized real-time database KogMo-RTDB [4]

run in separate processes according to the decision levels and can read and write information from the real-time database at any update rate.

For temporal consistency, all information in the database is labeled with a time-stamp that indicates the time when the data was recorded. This allows higher-level processes with a particularly long computing time to relate their output to the point in time when the underlying sensor data was valid.

The concept of a data-centered real-time database with consistent timestamp indexing enables a straightforward extension to distributed systems or car-to-car communication [3]. For a distributed system, additional latencies have to be taken into account, whereas car-to-car communication requires additional considerations on quality-of-service [9].

The centralized database design has other advantages such as easy integration of e.g. recording and playback functionality or a process watchdog, as it was included in Team AnnieWAY's software framework. This team's entry to the 2007 DARPA Urban Challenge – a competition of autonomous vehicles – successfully reached the finals where it competed safe and accident-free [7].

The different behaviour levels are implemented as follows:

- *Mission Planning* is performed by an A*/D* search of the optimal route in a given graph representation of the road network. In general, mission planning is performed only once when a new target position is defined or if the vehicle has to diverge from the planned route, e.g. due to a road blockade.
- *Manoeuvre Planning* is implemented as a concurrent hierarchical state machine (CHSM) [2], where every possible driving behaviour, e.g. driving or intersection

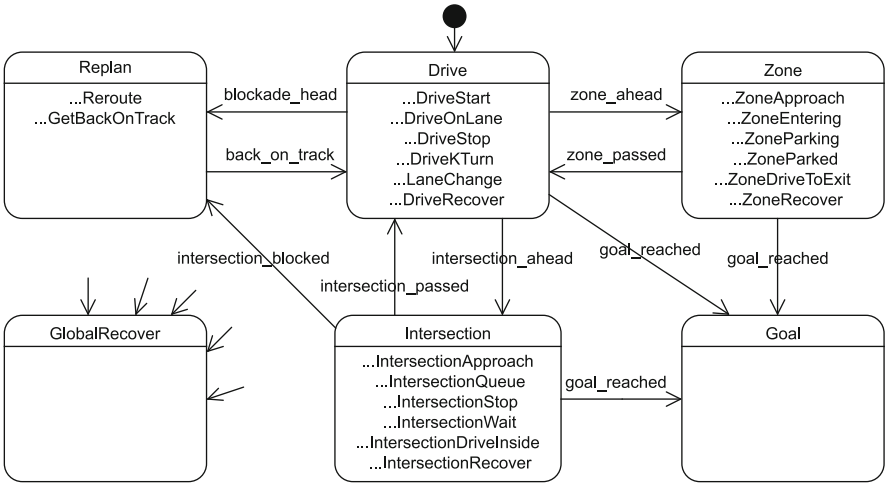


Fig. 17.12 Overview of the concurrent hierarchical state machine used to model traffic situations and behaviour

handling, is defined by a state which itself can have several substates, e.g. lane keeping or lane changing. A UML chart of the state machine is shown in Fig. 17.12. Output of the manoeuvre planning is the desired vehicle trajectory for approximately the next 10 s.

- As outlined in Sect. 17.4, the paths generated by the manoeuvre planning layer may be overridden by a *reactive layer* [6] for obstacle avoidance.

Figure 17.13 shows an example for a knowledge representation in a urban scene. It is composed of different perception layers that include the lane geometry, a static obstacle map, and tracked moving obstacles obtained from 2D lidar data. This example illustrates the idea of multiple perception layers that fulfill different real-time requirements: While the static occupancy grid is suitable for collision avoidance at low speed, higher-level behaviour generation such as handling the right-of-way or merging into moving traffic also requires a higher-level obstacle representation. On the other hand, detection and tracking of moving obstacles comes at a significantly higher computational cost as the generation of a static obstacle map.

17.6 Summary and Conclusions

Cooperative cognitive automobiles acquire and process information about their traffic environment using on-board sensors. This information serves as a decision basis to conduct appropriate driving actions. Time plays a crucial role in safe driving. Not only will a gain in time for decision of only half a second yield

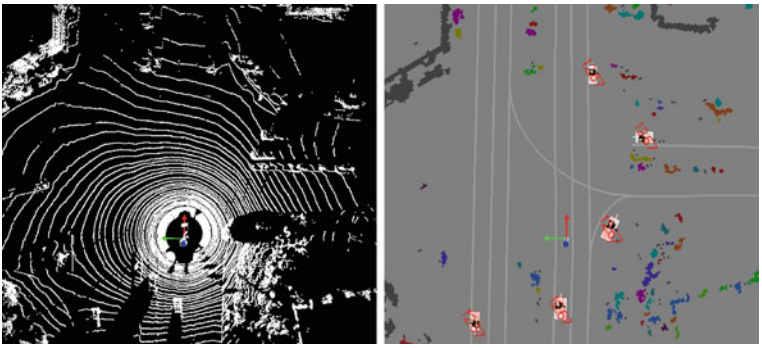


Fig. 17.13 Tracking of dynamic objects with occupancy grid map and linear Kalman Filter. *Left*: High resolution lidar data. *Right*: Tracked vehicles on segmented grid map [7]

a significant reduction of traffic fatalities, but real-time capabilities are the key to smooth vehicle control and enhanced traffic flow. It is argued that cognitive automobiles may save reaction time in a typical braking manoeuvre by as much as 1.5 s. We have discussed real-time requirements in automotive applications and have classified perception and action into strategic, tactical, and reactive levels. While reaction times may be as low as a few milliseconds on the reactive layer, tactical decisions are typically conducted in the range of a second, and strategic decisions may be allowed even longer periods of time. The exchange of information between vehicles opens potential for another dimension of traffic operation, namely cooperativity. Cooperative cognitive vehicles will exchange information on the perceived traffic scene and their intended trajectories. Negotiations among automobiles will yield concerted navigation with unprecedented safety, comfort, and efficiency. Clearly, driving on concerted trajectories will pose even higher importance on real-time capabilities. Preliminary experimental vehicles have been constructed in the Karlsruhe-Munich TCRC *Cognitive Automobiles* and have successfully demonstrated first automated driving manoeuvres.

Acknowledgements The authors gratefully acknowledge the fruitful collaboration of the partners from Karlsruhe Institute of Technology, Technische Universität München and Universität der Bundeswehr München within the Transregional Collaborative Research Centre 28 *Cognitive Automobiles*. Special thanks are directed to Georg Färber, one of the initiators, founders and member of the executive board of the Centre. The authors gratefully acknowledge support of the TCRC by the Deutsche Forschungsgemeinschaft (German Research Foundation).

References

1. European Union (2010) E-Safety. http://www.ec.europa.eu/information_society/activities/esafety

2. Gindele T, Jagszent D, Pitzer B, Dillmann R (2008) Design of the planner of Team AnnieWAY's autonomous vehicle used in the DARPA Urban Challenge 2007. In: Proceedings of the IEEE intelligent vehicles symposium, pp 1131–1136
3. Goebl M (2009) Eine realzeitfähige Architektur zur Integration kognitiver Funktionen. Dissertation, Technische Universität München, München
4. Goebl M, Färber G (2007) A real-time-capable hard- and software architecture for joint image and knowledge processing in cognitive automobiles. In: Proceedings of the IEEE intelligent vehicles symposium, Istanbul, Turkey, pp 734–739
5. Goebl M, Althoff M, Buss M, Färber G, Hecker F, Heißing B, Kraus S, Nagel R, Puente León F, Rattei F, Russ M, Schweitzer M, Thuy M, Wang C, Wuensche H (2008) Design and capabilities of the Munich cognitive automobile. In: Proceedings of the IEEE intelligent vehicles symposium, Eindhoven, the Netherlands, pp 1101–1107
6. Hundelshausen F, Himmelsbach M, Hecker F, Mueller A, Wuensche HJ (2008) Driving with tentacles: Integral structures for sensing and motion. J Field Robot 25(9):640–673
7. Kammel S, Ziegler J, Pitzer B, Werling M, Gindele T, Jagzent D, Schröder J, Thuy M, Goebl M, von Hundelshausen F, Pink O, Frese C, Stiller C (2008) Team AnnieWAY's autonomous system for the 2007 DARPA Urban Challenge. J Field Robot 25(9):615–639
8. Kämpchen N, Clauss M, Guenter Y, Schreier RM, Stiegeler M, Tischler K, Dietmayer K, Grossmann HP, Kabza H, Neumann H, Rothermel AL, Stiller C (2005) Vernetzte Fahrzeug-Umfelderfassung für zukünftige Fahrerassistenzsysteme. In: Maurer M, Stiller C (eds) Proceedings of the workshop Fahrerassistenzsysteme, Freundeskreis Mess- und Regelungstechnik Karlsruhe e.V., Walting, Altmühltal, pp 139–150
9. Nagel R, Eichler S, Eberspächer J (2007) Intelligent wireless communication for future autonomous and cognitive automobiles. In: Proceedings of the IEEE intelligent vehicles symposium, Istanbul, Turkey, pp 716–721
10. Özgüner Ü, Stiller C, Redmill K (2007) Systems for safety and autonomous behavior in cars: The DARPA grand challenge experience. IEEE Proc 95(2):1–16
11. Rasmussen J (1983) Skills, rules, and knowledge; signals, signs, and symbols, and other distinctions in human performance models. IEEE Trans Syst, Man, Cybernetics SMC-13(3):257–266
12. Schröder J, Gindele T, Jagszent D, Dillmann R (2008) Path planning for cognitive vehicles. In: Proceedings of the IEEE intelligent vehicles symposium, Eindhoven, Holland, pp 1119–1124
13. Stiller C, Färber G, Kammel S (2007) Cooperative cognitive automobiles. In: Proceedings of the IEEE intelligent vehicles symposium, Istanbul, Turkey, pp 215–220
14. Stiller C, Kammel S, Dang T, Duchow C, Hummel B (2007) Autonome Fahrzeugführung durchs Gelände – ION im Grand Challenge. at – Automatisierungstechnik 55(6):290–297
15. Thuy M, Althoff M, Buss M, Diepold K, Eberspächer J, Färber G, Goebl M, Heißing B, Kraus S, Nagel R, Naous Y, Obermeier F, Puente León F, Rattei F, Wang C, Schweitzer M, Wünsche H (2008) Kognitive Automobile – Neue Konzepte und Ideen des Sonderforschungsbereiches/TR-28. In: 3. Tagung Aktive Sicherheit durch Fahrerassistenz, Garching bei München
16. Tischler K, Hummel B (2005) Enhanced environmental perception by inter-vehicle data exchange. In: IEEE intelligent vehicles symposium, Las Vegas, USA
17. Werling M, Goebl M, Pink O, Stiller C (2008) A hardware and software framework for cognitive automobiles. In: Proceedings of the IEEE intelligent vehicles symposium 2008, Eindhoven, Niederlande, pp 1080–1085