Scheduling and Communication Synthesis for Distributed Real-Time Systems

Paul Pop

INSTITUTE OF TECHNOLOGY
LINKÖPINGS UNIVERSITET

ISBN 91-7219-776-5   ISSN 0280-7971
PRINTED IN LINKÖPING, SWEDEN
BY LINKÖPING UNIVERSITY

Lianei

Abstract

EMBEDDED SYSTEMS ARE now omnipresent: from cellular phones to pagers, from microwave ovens to PDAs, almost all the devices we use are controlled by embedded systems. Many embedded systems have to fulfill strict requirements in terms of performance and cost efficiency. Emerging designs are usually based on heterogeneous architectures that integrate multiple programmable processors and dedicated hardware components. New tools which extend design automation to system level have to support the integrated design of both the hardware and software components of such systems.

This thesis concentrates on aspects of scheduling and communication for embedded real-time systems. Special emphasis has been placed on the impact of the communication infrastructure and protocol on the overall system performance. The scheduling and communication strategies proposed are based on an abstract graph representation which captures, at process level, both the dataflow and the flow of control. We have considered non-preemptive static cyclic scheduling and preemptive scheduling with static priorities for the scheduling of processes, while the communications are statically scheduled according to the time triggered protocol. We have developed static cyclic scheduling algorithms for time-driven systems with control and data dependencies. We show that by considering aspects of the communication protocol, significant improvements can be gained in the schedule quality. In the context of event-driven systems we have proposed a less pessimistic schedulability analysis that is able to handle both control and data dependencies. Moreover, we have provided a schedulability analysis for the time-triggered protocol, and we have proposed several optimization strategies for the synthesis of communication protocol parameters. Extensive experiments as well as real-life examples demonstrate the efficiency of our approaches.

Acknowledgements

I WOULD LIKE to express thanks towards my advisors Petru Eles and Zebo Peng for their precious guidance during my graduate studies and their valuable comments on the thesis.

Many thanks also to the people at Volvo Technological Development in Gothenburg, especially to Jakob Axelsson, for their insightful ideas during the early stages of this work.

I am also grateful towards my colleagues at IDA and in the ARTES network for providing a creative and pleasant working environment, and towards the administrative and technical staff at IDA, that have always been supportive.

Last, but not least, my deepest gratitude towards my family and friends for their love and encouragement.

Contents

Chapter 1 Introduction

THIS THESIS CONCENTRATES on aspects related to the scheduling and synthesis of distributed embedded real-time systems consisting of programmable processors and application specific hardware components.

We have investigated the impact of particular communication infrastructures and protocols on the overall performance and how the requirements of such an infrastructure have to be considered for process and communication scheduling. Not only have particularities of the underlying architecture to be considered during scheduling, but the parameters of the communication protocol should also be adapted to fit the particular embedded application.

The approaches to scheduling and system synthesis are based on an abstract graph representation which captures, at process level, both dataflow and the flow of control.

This introductory chapter presents the motivation behind our research, the formulation of the research problems, and our contributions. An overview of the thesis is also presented.

1.1 Motivation

Figure 1.1 presents the microprocessor market share in the year 1999 [Tur99]. As we can see from the figure, less than 1% of the world's microprocessors are used in general purpose systems (i.e., computers). More than 99% are used in embedded real-time systems. Embedded real-time systems are now omnipresent: from cellular phones to pagers, from microwave ovens to PDAs, almost all the devices we use are controlled by embedded systems.

Many embedded systems have to fulfill strict requirements in terms of performance and cost efficiency. Emerging designs are usually based on heterogeneous architectures that integrate multiple programmable processors and dedicated hardware components. New tools which extend design automation to system level have to support the integrated design of both the hardware and software components of such systems.

During the synthesis of an embedded system the designer maps the functionality captured by the input specification on different architectures, trying to find the most cost efficient solution which, at the same time, meets the design requirements [Ern98]. This design process implies the iterative execution of several allocation and partitioning steps before the hardware and software components of the final implementation are generated. The term *hardware/software codesign* is often used to

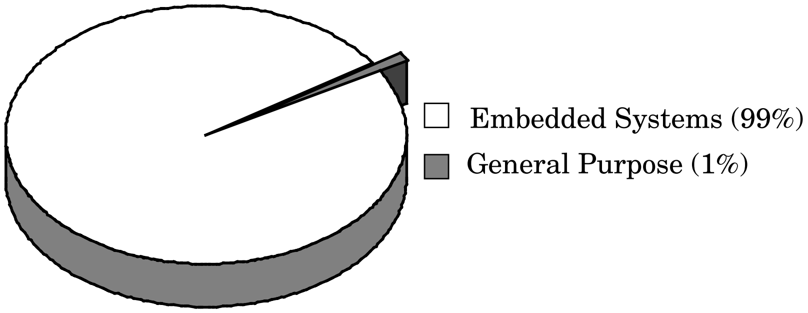

Figure 1.1: Microprocessor Market Shares

denote this system-level design process. Surveys on this topic can be found in [Mic96, Mic97, Ern98, Gaj95, Sta97, Wol94].

Figure 1.2 presents one possible codesign flow. The design starts with an abstract system specification. The initial system specification is implementation independent which means that no assumptions are made concerning how different parts will later be implemented. Thus, different implementation alternatives can be evaluated, including hardware/software trade-offs.

Moving further into the design process, the designer has to decide what components to include in the hardware architecture and how these components are connected. This is the so called *architecture selection* phase. Following the selection of the archi-

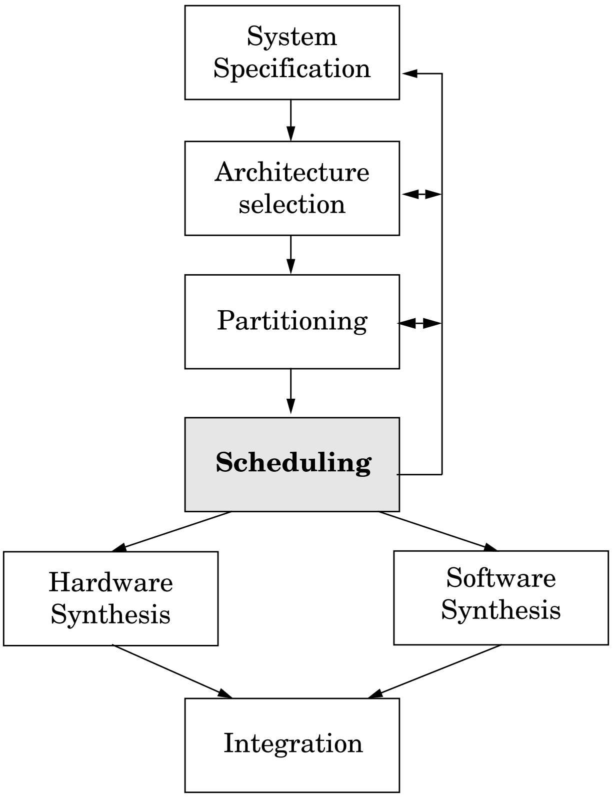

Figure 1.2: A Codesign Flow

tecture components, the designer has to decide what part of the functionality should be implemented on which of the selected components (*mapping*) and what is the execution order of the resulting tasks (*scheduling*).

Scheduling has to be performed during several phases of the design flow. We can, for example, use scheduling for performance estimation during the architecture selection and mapping phases where we are interested to quickly explore design alternatives and compare them in terms of timing behaviour. In addition, scheduling can also be used during the final stages of the design process when we are interested to synthesize the system such that time constraints are fulfilled.

Once a partitioning into hardware and software and a mapping have been decided on, the design process continues with the software synthesis and hardware synthesis phases. In the final phase, the hardware and software parts are integrated and tested. All these design steps can partially overlap, and they can be assisted by (semi)automatic synthesis tools.

In this thesis we concentrate on several aspects related to the scheduling and synthesis of systems consisting of communicating processes which are implemented on multiple processors and dedicated hardware components. In such a system, in which several processes communicate with each other and share resources like processors and buses, scheduling of processes and communications is a factor with a decisive influence on the performance of the system and on the way it meets its timing constraints.

1.2 Problem Formulation

The input for our problem is a model of a real-time system captured using a set of *conditional process graphs* [Ele98a, Ele00] described in detail in Section 2.1.1. Each node in this graph represents one process that can potentially be assigned to one of several programmable or hardware processors. Estimated worst case execution time for each process on each potential host processor is given. We assume that the amount of data to be transferred during communication between two processes has been determined in advance.

We consider a generic architecture consisting of *programmable processors* and application specific *hardware processors* (ASICs) connected through several *buses*. As the communication infrastructure for our distributed real-time system we consider the *time-triggered protocol* (TTP) [Kop94]. TTP is well suited for safety critical distributed real-time embedded systems and represents one of the emerging standards for several application areas like, for example, automotive electronics [Wir98]. Chapter 3 describes in more detail the system architectures considered, with Section 3.2.1 introducing the time-triggered protocol.

In our approach, process scheduling can use either a *non-preemptive static cyclic* or a *static priority preemptive* scheduling approach while the bus communication is statically scheduled according to the TTP. Only one process can be executed at a time by a programmable processor while a hardware processor can execute processes in parallel. Processes on different processors can be executed in parallel. Only one data transfer can be performed by a bus at a given moment. Data transfer on buses and computation can overlap.

Algorithms for automatic hardware/software partitioning have been presented in [Axe96, Ele97, Ern98]. The problems discussed in this thesis concern the performance estimation of a given design alternative and scheduling of processes and communications. Thus, we assume that each process has been assigned to a (programmable or hardware) processor and each communication channel which connects processes assigned to different processors has been assigned to a bus.

In this context, our goals are the following:

- derive a schedulability analysis for systems with both control and data dependencies,

- derive a schedulability analysis for systems where the communication takes place using the time-triggered protocol,
- determine an as small as possible worst case delay by which the system completes its execution and generate the static schedule such that this delay is guaranteed, and
- determine the parameters of the communication protocol so that the overall system performance is optimized and, thus, the imposed time constraints can be satisfied.

1.3 Contributions

In our approach, an embedded system is viewed as a set of interacting processes mapped on an architecture consisting of several programmable processors and ASICs interconnected by a communication channel.

Process interaction is not only in terms of dataflow but also captures the flow of control, since some processes can be activated depending on conditions computed by previously executed processes.

We have considered both the non-preemptive static cyclic scheduling and the static priority preemptive scheduling approaches for the scheduling of processes, while the communications are statically scheduled according to the TTP.

The scheduling strategies are based on a realistic communication model and execution environment. We take into consideration the overheads due to communications and to the execution environment and consider the requirements of the communication protocol during the scheduling process.

The main contributions of this thesis are:

- a less pessimistic schedulability analysis technique in order to bound the response time of a hard real-time system with both control and data dependencies (modelled as a condi-

tional process graph) [Pop00b, Pop00c];
- a schedulability analysis in the context of the time-triggered protocol, considering four different approaches to message scheduling [Pop00a, Pop99d];
- a static scheduling strategy for systems with both data and control dependencies, that takes into consideration the overheads due to communications and to the execution environment and considers the requirements of the communication protocol during the scheduling process [Pop99a, Pop99c]; and
- several optimization strategies for the synthesis of the bus access scheme in order to fit the communication particularities of a certain application [Pop00a, Pop99a, Pop99b].

1.4 Thesis Overview

This thesis has 7 chapters, and it is structured as follows:

- **Chapter 2** introduces the conditional process graph, describes the hardware and software architectures considered and presents the time-triggered protocol.
- **Chapter 3** presents the related work in the areas of scheduling and communication synthesis, as well as some basic approaches to hardware/software codesign.
- **Chapter 4** considers a non-preemptive static scheduling approach both for processes and messages. In such a context, we present previous work on the static cyclic scheduling of systems with data and control dependencies. This work is then extended to handle the scheduling of messages over the TTP. Several approaches to the synthesis of communication parameters for the TTP are proposed and they are later evaluated based on extensive experiments.
- **Chapter 5** assumes a preemptive fixed priority scheduling approach for the processes and a non-preemptive static cyclic scheduling approach for the messages, according to the TTP. A schedulability analysis of the TTP is developed considering

four message scheduling approaches. This analysis is then extended to systems with data and control dependencies. Optimization strategies that derive the parameters of the communication protocol are proposed. Extensive experiments evaluate the optimization strategies, and show that by considering both data and control dependencies we are able to reduce the pessimism of the analysis.

- **Chapter 6** presents a real-life example. We apply our scheduling and communication synthesis strategies to a vehicle cruise controller, and the results obtained validate our research.
- **Chapter 7** is the final chapter of the thesis and presents our conclusions and future work ideas.

Chapter 2 System Model and Architecture

THIS CHAPTER PRESENTS preliminaries for the later discussions. We start by introducing the conditional process graph that is used for system modelling, and then continue with the presentation of the hardware architecture considered. Our contribution is the software architecture designed for both time-driven and event-driven systems.

2.1 Design Representation

The specification that is at the input of the design process outlined in Figure 1.2 in the previous chapter could actually be very heterogeneous. Different formalisms are used to specify and model different parts of the system. Then, the information needed for the subsequent design phases, such as architecture selection, partitioning, scheduling, verification, etc., have to be extracted and mapped to internal representations that are more suited for that purpose. In certain cases, different internal models can be used for different tasks to be performed during system analysis and design.

There is a lot of research in the area of system modelling and specification, and an impressive number of representations have been proposed. An overview and classification of different design representations is given in [Edw97, Ern99].

In this thesis we use the conditional process graph [Ele98, Ele00] as an abstract model for system representation.

2.1.1 CONDITIONAL PROCESS GRAPH

A *process graph* is an abstract representation consisting of a directed, acyclic, polar graph $G(V, E_S, E_C)$. Each node $P_i \in V$ represents one process. E_S and E_C are the sets of simple and conditional edges respectively. $E_S \cap E_C = \varnothing$ and $E_S \cup E_C = E$, where E is the set of all edges. An edge $e_{ij} \in E$ from P_i to P_j indicates that the output of P_i is the input of P_j. The graph is polar, which means that there are two nodes, called *source* and *sink*, that conventionally represent the first and last process. These nodes are introduced as dummy processes, with zero execution time and no resources assigned, so that all other nodes in the graph are successors of the source and predecessors of the sink respectively.

A mapped process graph, $\Gamma(V^*, E_S^*, E_C^*, M)$, is generated from a process graph $G(V, E_S, E_C)$ by inserting additional processes (communication processes) on certain edges and by mapping each process to a given processing element. The mapping of processes $P_i \in V^*$ to processors and buses is given by a function M: $V^* \rightarrow PE$, where $PE = \{pe_1, pe_2, .., pe_{Npe}\}$ is the set of processing elements. $PE = PP \cup HP \cup B$, where PP is the set of programmable processors, HP is the set of dedicated hardware components, and B is the set of allocated buses. In certain contexts, we will call both programmable processors and hardware components simply processors. For any process P_i, $M(P_i)$ is the processing element to which P_i is assigned for execution. In the rest of this thesis, when we use the term *conditional process graph* (CPG), we consider a mapped process graph as defined here.

Each process P_i, assigned to a programmable or hardware processor $M(P_i)$, is characterized by an execution time t_{Pi}.

In the process graph depicted in Figure 2.1, P_0 and P_{15} are the source and sink nodes respectively. The nodes denoted $P_1, P_2, .., P_{14}$ are "ordinary" processes specified by the designer. They are assigned to one of the two programmable processors or to the hardware component (ASIC). The rest of the nodes are so called *communication processes* and they are represented in Figure 2.1 as solid circles. They are introduced during the generation of the system representation for each connection which links processes mapped to different processors. These processes model inter-processor com-

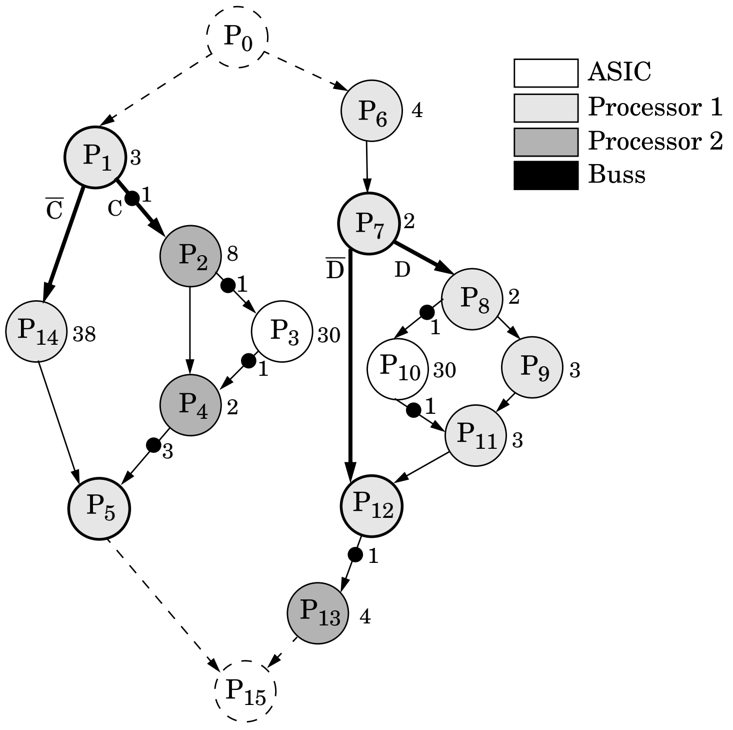

Figure 2.1: Conditional Process Graph

munication and their execution time $t_{i,j}$ (where P_i is the sender and P_j the receiver process) is equal to the corresponding communication time. All communications in Figure 2.1 are performed on one bus.

An edge $e_{ij} \in E_C$ is a *conditional edge* (represented with thick lines in Figure 2.1) and has an associated condition value. Transmission on such an edge takes place only if the associated condition value is *true* and not, like on simple edges, for each activation of the input process P_i. In Figure 2.1 processes P_1 and P_7 have conditional edges at their output.

We call a node with conditional edges at its output a *disjunction node* (and the corresponding process a *disjunction process*). A disjunction process has one associated condition, the value of which it computes. Alternative paths starting from a disjunction node, which correspond to complementary values of the condition, are disjoint and they meet in a so called *conjunction node* (with the corresponding process called *conjunction process*)[1]. In Figure 2.1 circles representing conjunction and disjunction nodes are depicted with thick borders. The alternative paths starting from disjunction node P_1, which computes condition C, meet in conjunction node P_5. We assume that conditions are independent and alternatives starting from different processes cannot depend on the same condition.

A process, that is not a conjunction process, can be activated only after all its inputs have arrived. A conjunction process can be activated after messages coming on one of the alternative paths have arrived. All processes issue their outputs when they terminate. If we consider the activation time of the source process as a reference, the activation time of the sink process is the delay of the system at a certain execution. This delay has to be, in the worst case, smaller than a certain imposed deadline. Release

1. If no process is specified on an alternative path, it is modelled by a conditional edge from the disjunction to the corresponding conjunction node (a communication process may be inserted on this edge at mapping).

times of some processes as well as multiple deadlines can be easily modelled by inserting dummy nodes between certain processes and the source or the sink node respectively. These dummy nodes represent processes with a certain execution time but which are not allocated to any processing element.

A boolean expression X_{Pi}, called a *guard*, can be associated to each node P_i in the graph. It represents the necessary conditions for the respective process to be activated. X_{Pi} is not only necessary but also sufficient for process P_i to be activated during a given system execution. Thus, two nodes P_i and P_j, where P_j is not a conjunction node, are connected by an edge e_{ij} only if $X_{Pj} \Rightarrow X_{Pi}$ (which means that X_{Pi} is true whenever X_{Pj} is true). This avoids specifications in which a process is blocked even if its guard is true, because it waits for a message from a process which will not be activated. If P_j is a conjunction node, predecessor nodes P_i can be situated on alternative paths corresponding to a condition.

The above execution semantics is that of a so called single rate system. It assumes that a node is executed at most once for each activation of the system. If processes with different periods have to be handled, this can be solved by generating several instances of the processes and building a CPG which corresponds to a set of processes as they occur within a time period that is equal to the least common multiple of the periods of the involved processes.

As mentioned, we consider execution times of processes, as well as the communication times, to be given. In the Figure 2.1 they are depicted to the right of each node. In the case of hard real-time systems this will, typically, be worst case execution times and their estimation has been extensively discussed in the literature [Eng99, Li95, Lun99, Mal97]. For many applications, actual execution times of processes are depending on the current data and/or the internal state of the system. By explicitly capturing the control flow in our model, we allow for a more fine-tuned modeling and a tighter (less pessimistic) assignment of worst case execution times to processes, compared to traditional data-flow based approaches.

2.2 System Architecture

As pointed out in the introductory chapter, real-time systems are nowadays omnipresent. Depending on the particular application implemented, real-time systems can be implemented as uniprocessor, multiprocessor, or distributed. Systems can be hard or soft, event-driven or time-driven, fault-tolerant, autonomous, etc. A good classification of real-time systems is given in [Kop97a].

This chapter describes the architecture we consider in this thesis for the implementation of a distributed real-time system. Our hardware architecture consists of a set of nodes interconnected by a communication channel that uses the time-triggered protocol as the communication protocol. The software architecture depends on the triggering mechanisms for the start of communication and processing activities.

2.2.1 TIME VS. EVENTS

According to [Kop97a] a *trigger* is “an event that causes the start of some action, e.g., the execution of a task or the transmission of a message.” Different approaches to the design of real-time systems can be identified, based on the triggering mechanisms for the processing and communication: *event-triggered* or *time-triggered*.

In the event-triggered approach all the activities happen when a significant change of state occurs. The significant events are brought to the attention of the CPU by the interrupt mechanism. Event-triggered systems typically require preemptive priority-based scheduling, where the appropriate process is invoked to service the event.

In the time-triggered approach all the activities are initiated at predetermined points in time. Thus, there is only one interrupt in each node of a distributed time-triggered system, the time interrupt. In a distributed time-triggered system it is assumed that the clocks of all nodes are synchronized to provide a global notion of time. Time-triggered systems typically require non-preemptive static cyclic scheduling, where the process activation or message communication is done based on a schedule table built off-line.

We consider the time-triggered protocol for the communication infrastructure, and thus, the communication of messages is time-triggered. However, depending on the particular application, the activation of processes can be either time-triggered (Chapter 4) or event-triggered (Chapter 5).

2.2.2 HARDWARE ARCHITECTURE

We consider architectures consisting of nodes connected by a broadcast communication channel (Figure 2.2). Every node consists of a TTP controller [Kop97b], a CPU, a RAM, a ROM and an I/O interface to sensors and actuators. A node can also have an ASIC in order to accelerate parts of its functionality.

Time Triggered Protocol. Communication between nodes is based on the time-triggered protocol (TTP) [Kop94]. TTP was designed for distributed real-time applications that require predictability and reliability (e.g, drive-by-wire). It integrates all the services necessary for fault-tolerant real-time systems. TTP services of importance to our problems are: message transport with acknowledgment and predictable low latency, clock synchronization within the microsecond range and rapid mode changes.

The communication channel is a broadcast channel, so a message sent by a node is received by all the other nodes. The bus access scheme is time-division multiple-access (TDMA) (Figure

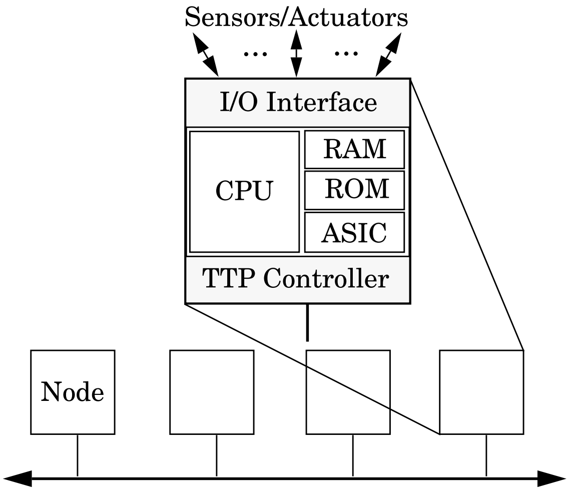

Figure 2.2: System Architecture

2.3). Each node N_i can transmit only during a predetermined time interval, the so called TDMA slot S_i. In such a slot, a node can send several messages packaged in a frame. We consider that a slot S_i is at least large enough to accommodate the largest message generated by any process assigned to node N_i, so the messages do not have to be split in order to be sent. A sequence of slots corresponding to all the nodes in the architecture is called a TDMA round. A node can have only one slot in a TDMA round. Several TDMA rounds can be combined together in a cycle that is repeated periodically. The sequence and length of the slots are the same for all the TDMA rounds. However, the length and contents of the frames may differ.

Every node has a TTP controller that implements the protocol services, and runs independently of the node's CPU. Communication with the CPU is performed through a so called message base interface (MBI) which is usually implemented as a dual ported RAM (Figure 2.4).

The TDMA access scheme is imposed by a so called message descriptor list (MEDL) that is located in every TTP controller.

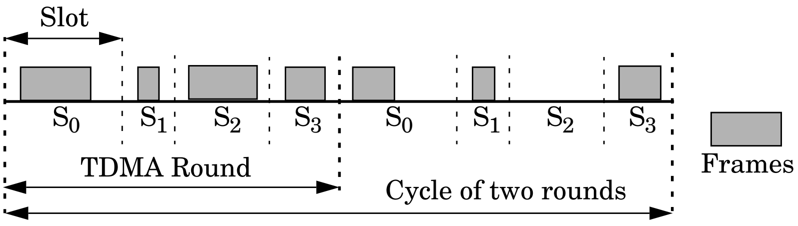

Figure 2.3: Buss Access Scheme

The MEDL basically contains: the time when a frame has to be sent or received, the address of the frame in the MBI and the length of the frame. MEDL serves as a schedule table for the TTP controller which has to know when to send or receive a frame to or from the communication channel.

The TTP controller provides each CPU with a timer interrupt based on a local clock, synchronized with the local clocks of the other nodes. The clock synchronization is done by comparing the a-priori known time of arrival of a frame with the observed arrival time. By applying a clock synchronization algorithm, TTP provides a global time-base of known precision, without any overhead on the communication.

Information transmitted on the bus has to be properly formatted in a frame. A TTP frame has the following fields: start of frame, control field, data field, and CRC field. The data field can contain one or more application messages.

2.2.3 SOFTWARE ARCHITECTURE

We have designed two distinct software architectures: one for time-triggered systems, and another for event-triggered systems. The main component of both software architectures is a real-time kernel that runs on top of each node of the architecture.

Time Triggered Systems. Each kernel in the software architecture for the time-triggered systems has a schedule table. This schedule table contains all the information needed to take decisions on activation of processes and transmission of messages, based on the values of conditions (Table 4.1).

In order to run a predictable hard real-time application the overhead of the kernel and the worst case administrative overhead (WCAO) of every system call has to be determined. Having a time-triggered system, all the activity is derived from the progression of time which means that there are no other interrupts except for the timer interrupt.

Several activities, like polling of the I/O or diagnostics, take place directly in the timer interrupt routine. The overhead due to this routine is expressed as the utilization factor U_t. U_t represents a fraction of the CPU power utilized by the timer interrupt routine, and has an influence on the execution times of the processes.

We also have to take into account the overheads for process activation and message passing. For process activation we con-

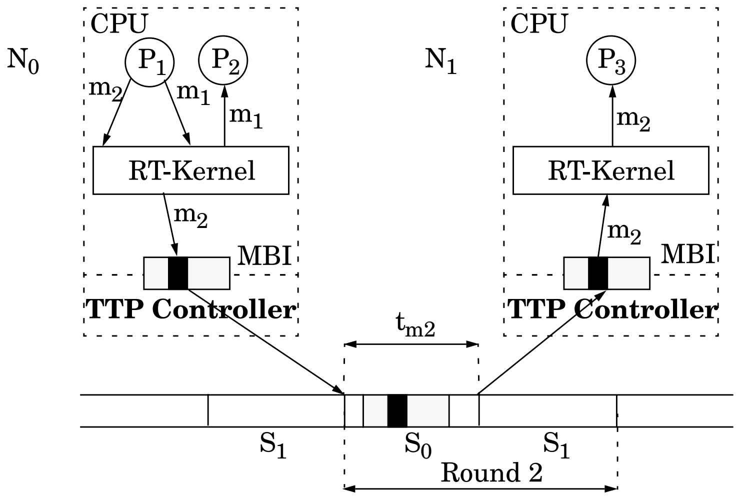

Figure 2.4: Message Passing, Time-Driven Systems

sider an overhead δ_{PA}. The message passing mechanism is illustrated in Figure 2.4, where we have three processes, P_1 to P_3. P_1 and P_2 are mapped to node N_0 that transmits in slot S_0, and P_3 is mapped to node N_1 that transmits in slot S_1. Message m_1 is transmitted between P_1 and P_2 that are on the same node, while message m_2 is transmitted from P_1 to P_3 between the two nodes. We consider that each process has its own memory locations for the messages it sends or receives and that the addresses of the memory locations are known to the kernel through the schedule table.

P_1 is activated according to the schedule table, and when it finishes it calls the send kernel function in order to send m_1, and then m_2. Based on the schedule table, the kernel copies m_1 from the corresponding memory location in P_1 to the memory location in P_2. The time needed for this operation represents the WCAO δ_S for sending a message between processes located on the same node[1]. When P_2 will be activated it finds the message in the right location. According to our scheduling policy, whenever a receiving process needs a message, the message is already placed in the corresponding memory location. Thus, there is no overhead on the receiving side, for messages exchanged on the same node.

Message m_2 has to be sent from node N_0 to node N_1. At a certain time, known from the schedule table, the kernel transfers m_2 to the TTP controller by packaging m_2 into a frame in the MBI. The WCAO of this function is δ_{KS}. Later on, the TTP controller knows from its MEDL when it has to take the frame from the MBI, in order to broadcast it on the bus. In our example the timing information in the schedule table of the kernel and the MEDL is determined in such a way that the broadcasting of the frame is done in the slot S_0 of *Round 2*. The TTP controller of

1. Overheads δ_S, δ_{KS} and δ_{KR} depend on the length of the transferred message; in order to simplify the presentation this aspect is not discussed further.

node N_1 knows from its MEDL that it has to read a frame from slot S_0 of *Round 2* and to transfer it into the MBI. The kernel in node N_1 will read the message m_2 from the MBI, with a corresponding WCAO of δ_{KR}. When P_3 will be activated based on the local schedule table of node N_1, it will already have m_2 in its right memory location.

Event Triggered Systems. Each kernel in the software architecture for the event-triggered systems has a so called tick scheduler. The tick scheduler is activated periodically by the timer interrupts and decides on activation of processes, based on their priorities. Several activities, like polling of the I/O or diagnostics, take also place in the timer interrupt routine.

As in the previous section, the overhead of the kernel and the worst case administrative overhead (WCAO) of every system call have to be determined. Our schedulability analysis takes into account these overheads, and also the overheads due to the message passing.

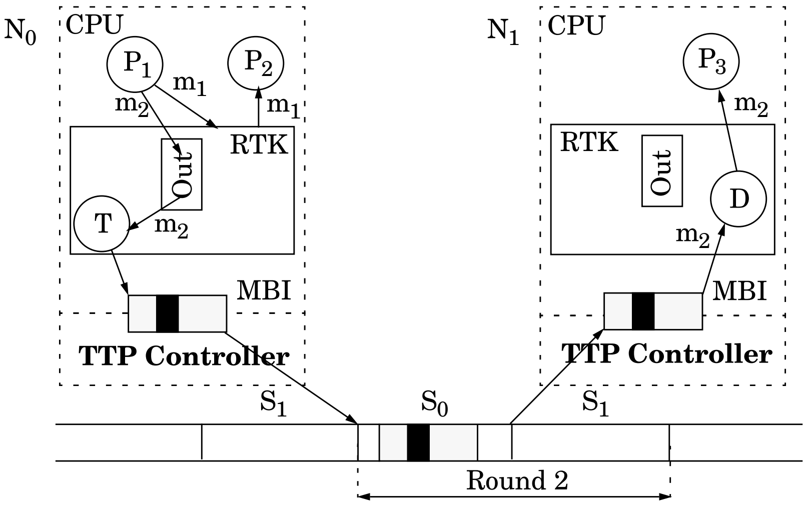

Figure 2.5: Message Passing, Event-Driven Systems

The message passing mechanism is illustrated in Figure 2.5, where we have three processes, P_1 to P_3. P_1 and P_2 are mapped to node N_0 that transmits in slot S_0, and P_3 is mapped to node N_1 that transmits in slot S_1. Message m_1 is transmitted between P_1 and P_2 that are on the same node, while message m_2 is transmitted from P_1 to P_3 between the two nodes.

Messages between processes located on the same processor are passed through shared protected objects. The overhead for their communication is accounted for by the blocking factor, computed according to the priority ceiling protocol [Sha90].

Message m_2 has to be sent from node N_0 to node N_1. Thus, after m_2 is produced by P_1, it will be placed into an outgoing message queue, called *Out*. The access to the queue is guarded by a priority-ceiling semaphore. A so called transfer process (denoted with T in Figure 2.5) moves the message from the *Out* queue into the MBI.

How the message queue is organized and how the message transfer process selects the particular messages and assembles them into a frame, depends on the particular approach chosen for message scheduling (see Section 5.1). The message transfer process is activated at certain a priori known moments, by the tick scheduler in order to perform the message transfer. These activation times are stored in a message handling time table (MHTT) available to the real-time kernel in each node. Both the MEDL and the MHTT are generated off-line as result of the schedulability analysis and optimization which will be discussed later. The MEDL imposes the times when the TTP controller of a certain node has to move frames from the MBI to the communication channel. The MHTT contains the times when messages have to be transferred by the message transfer process from the *Out* queue into the MBI, in order to further be broadcasted by the TTP controller. As result of this synchronization, the activation times in the MHTT are directly related to those in the MEDL and the first table results directly from the second one.

It is easy to observe that we have the most favourable situation when, at a certain activation, the message transfer process finds in the *Out* queue all the “expected” messages which then can be packed into the just following frame to be sent by the TTP controller. However, application processes are not statically scheduled and availability of messages in the *Out* queue can not be guaranteed at fixed times. Worst case situations have to be considered, as will be shown in Section 5.1.

Let us come back to Figure 2.5. There we assumed a context in which the broadcasting of the frame containing message m_2 is done in the slot S_0 of *Round 2*. The TTP controller of node N_1 knows from its MEDL that it has to read a frame from slot S_0 of *Round 2* and to transfer it into its MBI. In order to synchronize with the TTP controller and to read the frame from the MBI, the tick scheduler on node N_1 will activate, based on its local MHTT, a so called delivery process, denoted with D in Figure 2.5. The delivery process takes the frame from the MBI, and extracts the messages from it. For the case when a message is split into several packets, sent over several TDMA rounds, we consider that a message has arrived at the destination node after all its corresponding packets have arrived. When m_2 has arrived, the delivery process copies it to process P_3 which will be activated. Activation times for the delivery process are fixed in the MHTT just as explained earlier for the message transfer process.

The number of activations of the message transfer and delivery processes depends on the number of frames transferred, and they are taken into account in our analysis, as well as the delay implied by the propagation on the communication bus.

Chapter 3 Related Work

A LOT HAS BEEN published in the last years in the areas of hardware/software codesign and real-time systems research. The intent of this chapter is to give a brief overview of the previous research on codesign with an emphasis on scheduling and communication synthesis.

The aspects of our work that differ from the related research presented in this chapter are:

- we consider a more complex system model that is able to capture both the flow of data and that of control;
- our system architectures are heterogeneous and consider a realistic communication model based on the time-triggered protocol;
- we consider issues related to the interaction between scheduling of processes and communication scheduling; and
- we have provided system level communication synthesis strategies that lead to significant improvements on the performance of the system.

3.1 Hardware/Software Codesign

Section 1.1 has introduced hardware/software codesign (shorter, codesign) and presented a possible codesign flow. The intention of this section is to provide a short overview of this emerging research area. For more details, the reader is referred to several surveys on this topic [Mic96, Mic97, Ern98, Gaj95, Sta97, Wol94].

Codesign is a relatively new research area. The "First International Workshop on Hardware/Software Codesign" has taken place in 1992, and has been an yearly event since then. Around the same time hardware/software codesign tracks and sessions have started to appear at important Electronic Design Automation (EDA) conferences like DAC, DATE, ICCAD, ISSS, etc.

The initial assumptions of codesign were quite restrictive, and the goals modest. For example, several researchers have assumed a simple specification in form of a computer program, and the main goal was to obtain an as high as possible execution performance within a given cost (acceleration). The architecture considered consisted of a single processor together with an ASIC used to accelerate parts of the functionality [Cho95a, Gup95, Moo97]. In this context, the main problems were to divide the functionality between the ASIC and the CPU (hardware/software partitioning) [Ele97, Ern93, Gup93, Vah94], to automatically generate drivers and other components related to communication (communication synthesis) [Cho92, Wal94] and to simulate and verify the resulting system (cosimulation and coverification) [Val95, Val96]. However, today the initial assumptions are no longer valid and the goals are much broader [Bol97, Dav98, Dav99, Dic98, Lak99, Ver96]:

- The applications are heterogeneous, consisting of hardware and software components. Both hardware and software can be data or control dominated, and hardware can be both digital and analog.

- The specification for such applications is inherently heterogeneous and complex. Several languages as well as several models of computation can be found within a specification.
- The architectures are varied ranging from distributed embedded systems, in the automotive electronics area, to systems on a chip used in telecommunications.
- The goals include not only acceleration with minimal hardware cost, but also issues related to the reuse of legacy hardware and software subsystems, real-time constraints, quality of service, fault tolerance and dependability, power consumption, flexibility, time-to-market, etc.

3.2 Scheduling

Process scheduling for performance estimation and synthesis of real-time systems has been intensively researched in the last years. The existing approaches differ in the scheduling strategy adopted, system architectures considered, handling of the communication and process interaction aspects. However, our main distinction in this section will be made between non-preemptive static cyclic scheduling and preemptive fixed-priority scheduling. We have to mention that performance estimation and scheduling of processes typically requires, as an input, estimated execution times of single processes [Eng99, Ern97, Gon95, Hen95, Li95, Lun99, Mal97, Suz96].

Non-preemptive static cyclic scheduling. Static cyclic scheduling of a set of data dependent software processes on a multiprocessor architecture has been intensively researched [Kop97a, Xu00].

Several approaches are based on list scheduling heuristics using different priority criteria [Cof72, Deo98, Jor97, Kwo96, Wu90] or on branch-and-bound algorithms [Kas84]. These approaches are based on the assumption that a number of identical processors are available to which processes are progressively assigned as the static schedule is elaborated. Such an assumption is obviously not acceptable for distributed embedded systems which are heterogeneous by nature. In [Jor97] a list scheduling based approach is extended to handle heterogeneous architectures. Scheduling is performed by progressively assigning tasks to the allocated processors with the goal to minimize the length of the schedule. The proposed algorithm handles only processors which execute one single process at a time (not typical for hardware) and the resulting partitioning does not take into consideration any design constraints.

In [Ben96, Pra92] static scheduling and partitioning of processes, and allocation of system components, are formulated as a mixed integer linear programming (MILP) problem. A disadvantage of this approach is the complexity of solving the MILP model. The size of such a model grows quickly with the number of processes and allocated resources. In [Kuc97] a formulation using constraint logic programming has been proposed for similar problems.

In all the previous approaches process interaction is only in terms of dataflow. However, when including control dependencies significant improvements in the quality of the resulting schedules can be obtained [Ele98a]. Section 4.1 presents in more detail related research on the static scheduling for systems with control and data dependencies that is used as a starting point for our work.

It has been claimed [Xu93] that static cyclic scheduling approach is the only approach that can solve a certain class of problems. However, advances in the area of fixed priority preemptive scheduling show that such classes of problems can also be handled with other scheduling strategies [Aud93, Tin94b].

Fixed priority preemptive scheduling. Preemptive scheduling of independent processes with static priorities running on single processor architectures has its roots in [Liu73]. The approach has been later extended to accommodate more general computational models and has also been applied to distributed systems [Tin94a]. The reader is referred to [Aud95, Bal98, Sta93] for surveys on this topic.

In [Yen97] performance estimation is based on a preemptive scheduling strategy with static priorities using rate monotonic analysis. In [Lee99] an earlier deadline first strategy is used for non-preemptive scheduling of processes with possible data dependencies. Preemptive and non-preemptive static scheduling are combined in the cosynthesis environment described in [Dav98, Dav99].

In many of the previous scheduling approaches researchers have assumed that processes are scheduled independently. However, this is not the case in reality, where process sets can exhibit both data and control dependencies. Moreover, knowledge about these dependencies can be used in order to improve the accuracy of schedulability analyses and the quality of produced schedules.

One way of dealing with data dependencies between processes with static priority based scheduling has been indirectly addressed by the extensions proposed for the schedulability analysis of distributed systems through the use of the *release jitter* [Tin94b]. Release jitter is the worst case delay between the arrival of a process and its release (when it is placed in the run-queue for the processor) and can include the communication delay due to the transmission of a message on the communication channel.

Tindell et al. [Tin94b] and Yen et al. [Yen98] use time offset relationships and phases, respectively, in order to model data dependencies. Offset and phase are similar concepts that express the existence of a fixed interval in time between the arrivals of sets of processes. The authors show that by introducing such concepts into the computational model, the pessimism of the analysis is significantly reduced when bounding the time behaviour of the system. The work has been later extended with the concept of dynamic offsets [Pal98]. The works by [Tin94b] and [Yen98] are further detailed in Section 5.2 that introduces the schedulability analysis for the time-triggered protocol. Also, a brief introduction to schedulability analysis is presented in Section 5.1.

When control dependencies exist then, depending on conditions, only a subset of the set of processes is executed during an invocation of the system. *Modes* have been used to model a certain class of control dependencies [Foh93]. Such a model basically assumes that at the starting of an execution cycle, a particular functionality is known in advance and is fixed for one or several cycles until another mode change is performed. However, modes cannot handle fine grained control dependencies, or certain combinations of data and control dependencies. Careful modeling using the *periods* of processes (lower bound between subsequent re-arrivals of a process) can also be a solution for some cases of control dependencies [Ger96]. If, for example, we know that a certain set of processes will only execute every second cycle of the system, we can set their periods to the double of the period of the rest of the processes in the system. However, using the worst case assumption on periods leads very often to unnecessarily pessimistic schedulability evaluations. More refined process models can produce much better schedulability results, as will be later shown in the thesis. Recent works [Bar98a, Bar98b] aim at extending the existing models to handle control dependencies. In [Bar98b] Baruah introduces the *recurring real-time task model* that is able to capture lower level control dependencies, and presents an exponential-time analysis for uniprocessor systems.

3.3 Aspects Related to Communication

Currently, more and more real-time systems are used in physically distributed environments and have to be implemented on distributed architectures in order to meet reliability, functional, and performance constraints. However, researchers have often ignored or very much simplified aspects concerning the communication infrastructure.

One typical approach is to consider communication processes as processes with a given execution time (depending on the amount of information exchanged) and to schedule them as any other process, without considering issues like communication protocol, bus arbitration, packaging of messages, clock synchronization, etc. These aspects are, however, essential in the context of safety-critical distributed real-time applications and one of our objectives is to develop a strategy which takes them into consideration for process scheduling.

Many efforts dedicated to communication synthesis have concentrated on the synthesis support for the communication infrastructure but without considering hard real-time constraints and system level scheduling aspects [Cho95b, Dav95, Knu99, Nar94]. Lower level communication synthesis aspects under timing constraints have been addressed in [Ort98, Knu99].

We have to mention here some results obtained in extending real-time schedulability analysis so that network communication aspects can be handled. In [Tin95], for example, the CAN protocol is investigated while the work reported in [Erm97] considers systems based on the ATM protocol. Analysis for a simple TDMA protocol is provided in [Tin94a] that integrates processor and communication schedulability and provide a “holistic” schedulability analysis in the context of distributed real-time systems.

Chapter 4 Scheduling and Bus Access Optimization for Time Driven Systems

IN THIS CHAPTER we consider time-driven distributed real-time systems that use the time-triggered protocol for the communication infrastructure. Thus, both the activation of processes and the transmission of messages are done based on the progression of time.

The chapter starts by presenting an approach to static scheduling under control and data dependencies for distributed real-time systems [Dob98, Ele98a, Ele00]. The approach considers a simplified communication model in which the execution time of the communication processes depends only on the amount of data exchanged by the processes engaged in the communication. The communication processes are treated exactly as ordinary processes during scheduling, and the bus is modelled similar to a programmable processor that can "execute" one communication at a time as soon as the communication becomes "ready".

We propose in this chapter several extensions to this approach:

- scheduling of messages using a realistic communication model based on the time-triggered protocol (Section 4.2.1);
- a new priority function for list scheduling that uses knowledge about the bus access scheme in order to improve the schedule quality (Section 4.2.2); and
- optimization strategies for the synthesis of parameters of the communication protocol, aimed at improving the schedule quality (Section 4.2.3).

4.1 Scheduling with Control and Data Dependencies

In our approach, we consider distributed hard-real time systems modelled using conditional process graphs.

Optimal scheduling has been proven to be an NP complete problem [Ull75] in even simpler contexts than those characteristic to distributed systems represented as CPGs. Thus, it is essential to develop heuristics which produce good quality results in a reasonable time.

In [Dob98, Ele98a, Ele00] the authors concentrate on developing a scheduling algorithm for systems with both control and data dependencies, modelled using the conditional process graph. According to this model, some processes can only be activated if certain conditions, computed by previously executed processes, are fulfilled. Thus, process scheduling is complicated since at a given activation of the system, only a certain subset of the total amount of processes is executed and this subset differs from one activation to the other.

The output produced by their scheduling algorithm is a schedule table that contains all the information needed by a distributed run time scheduler to take decisions on activation of processes. It is considered that during execution a very simple non-preemptive scheduler located in each processing element decides on process and communication activation depending on the actual values of conditions. Only one part of the table has to be stored in each processor, namely the part concerning decisions which are taken by the corresponding scheduler.

Under these assumptions, Table 4.1 presents a possible schedule (produced by the algorithm in Figure 4.1) for the conditional process graph in Figure 2.1. In Table 4.1 there is one row for each "ordinary" or communication process, which contains activation times corresponding to different values of conditions. Each column in the table is headed by a logical expression constructed as a conjunction of condition values. Activation times in a given column represent starting times of the processes when the respective expression is true.

According to the schedule in Table 4.1 process P_1 is activated unconditionally at the time 0, given in the first column of the table. However, activation of some processes at a certain execution depends on the values of the conditions, which are unpredictable. For example, process P_{11} has to be activated at t=44 if $C \wedge D$ is true and t=52 if $\overline{C} \wedge D$ is true. At a certain moment during the execution, when the values of some conditions are already known, they have to be used in order to take the best possible decisions on when and which process to activate. Therefore, after the termination of a process that produces a condition (disjunction process), the value of the condition is broadcasted from the corresponding processor to all other processors. This broadcast is scheduled as soon as possible on the communication channel, and is considered together with the scheduling of the messages.

To produce a deterministic behaviour, which is correct for any combination of conditions, the table has to fulfill several requirements:

1. No process will be activated if, for a given execution, the con-

ditions required for its activation are not fulfilled.
2. Activation times have to be uniquely determined by the conditions.
3. Activation of a process P_i at a certain time t has to depend

Table 4.1: Schedule Table for Graph in Figure 2.1

process	**true**	**C**	**C∧D**	**C∧$\overline{\textbf{D}}$**	**$\overline{\textbf{C}}$**	**$\overline{\textbf{C}}$∧D**	**$\overline{\textbf{C}}$∧$\overline{\textbf{D}}$**
P_1	0						
P_2		5					
P_3			14	14			
P_4			45	45			
P_5			51	50		55	47
P_6		3			3		
P_7		7			7		
P_8			9			9	
P_9			11			11	
P_{10}			13			13	
P_{11}			44			52	
P_{12}			47	9		55	9
P_{13}			48	13		56	11
P_{14}						14	9
$P_{1,2}$		4					
$P_{4,5}$			48	47			
$P_{2,3}$			13	13			
$P_{3,4}$			44	44			
$P_{12,13}$			47	10		55	
$P_{8,10}$			12			12	
$P_{10,11}$			43			43	
C		3				11	9
D			11	9		11	9

only on condition values which are determined at the respective moment t and are known to the processing element which executes P_i.

4.1.1 LIST SCHEDULING BASED ALGORITHM

As the starting point for our improved scheduling technique that is tailored for time-triggered embedded systems we consider the list scheduling based algorithm in [Dob98, Ele00] presented, in a very simplified form, in Figure 4.1.

```
ListScheduling(CurrentTime, ReadyList, KnownConditions)
    repeat
        Update(ReadyList)
        for each processing element PE
            if PE is free at CurrentTime then
                $P_i$ = GetReadyProcess(ReadyList)
                if there exists a $P_i$ then
                    Insert($P_i$, ScheduleTable, CurrentTime, KnownConds)
                    if $P_i$ is a disjunction process then
                        $C_i$ = condition calculated by $P_i$
                        ListScheduling(CurrentTime,
                            ReadyList $\cup$ ready nodes from the true branch,
                            KnownConditions $\cup$ true $C_i$)
                        ListScheduling(CurrentTime,
                            ReadyList $\cup$ ready nodes from the false branch,
                            KnownConditions $\cup$ false $C_i$)
                    end if
                end if
            end if
        end for
        CurrentTime = time when a scheduled process terminates
    until all processes of this alternative path are scheduled
end ListScheduling
```

Figure 4.1: List Scheduling Based Algorithm

List scheduling heuristics [Ele98b] are based on priority lists from which processes are extracted in order to be scheduled at certain moments. In the algorithm presented in Figure 4.1, there is such a list, ReadyList, that contains the processes which are eligible to be activated on the corresponding processor at time CurrentTime. These are processes which have not been yet scheduled but have all predecessors already scheduled and terminated.

The ListScheduling function is recursive and calls itself for each disjunction node in order to separately schedule the nodes in the true branch, and those in the false branch respectively. Thus, the alternative paths are not activated simultaneously and resource sharing is correctly achieved (for details on how the algorithm fulfils the three requirements on the schedule table we refer to [Ele00]).

An essential component of a list scheduling heuristic is the priority function used to solve conflicts between ready processes. The highest priority process will be extracted by function GetReadyProcess from the ReadyList in order to be scheduled.

4.1.2 PCP Priority Function

Priorities for list scheduling very often are based on the critical path (CP) from the respective process to the sink node. Thus, for CP scheduling, the priority assigned to a process P_i will be the maximal execution time from the current node to the sink:

$$l_{Pi} = \max_k \sum_{P_j \in \pi_{ik}} t_{Pj} ,$$

where π_{ik} is the kth path from node.

Considering the concrete definition of the problem, significant improvements of the resulting schedule can be obtained, without any penalty in scheduling time, by making use of the available information on process allocation [Ele98b].

Let us consider the graph in Figure 4.2 and suppose that the list scheduling algorithm has to decide between scheduling proc-

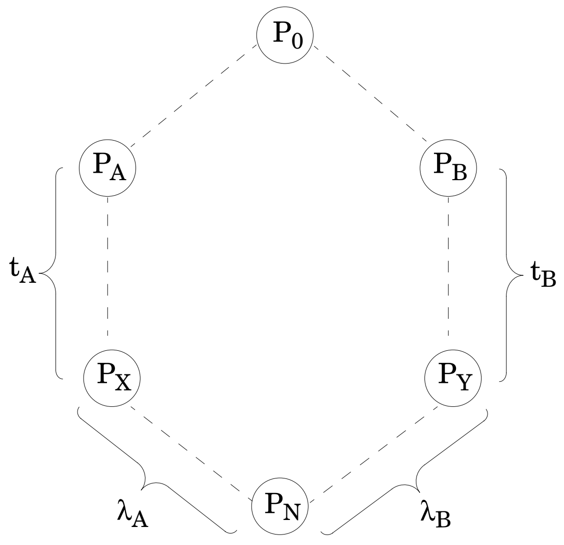

Figure 4.2: Delay estimation for PCP scheduling

ess P_A or P_B which are both ready to be scheduled on the same programmable processor or bus pe_i. In Figure 4.2 we depicted only the critical path from P_A and P_B to the sink node. Let us consider that P_X is the last successor of P_A on the critical path such that all processes from P_A to P_X are assigned to the same processing element pe_i. The same holds for P_Y relative to P_B. t_A and t_B are the total execution time of the chain of processes from P_A to P_X and from P_B to P_Y respectively, following the critical paths. λ_A and λ_B are the total execution times of the processes on the rest of the two critical paths. Thus, we have:

$l_{PA} = t_A + \lambda_A$, and $l_{PB} = t_B + \lambda_B$.

However, [Ele98b] does not use the length of these critical paths as a priority. The policy in [Ele98b] is based on the estimation of a lower bound L on the total delay, taking into consideration that the two chains of processes P_A-P_X and P_B-P_Y are executed on the same processor. L_{PA} and L_{PB} are the lower bounds if P_A and P_B respectively are scheduled first:

$L_{PA} = \max(T\_current + t_A + \lambda_A, T\_current + t_A + t_B + \lambda_B)$
$L_{PB} = \max(T\_current + t_B + \lambda_B, T\_current + t_B + t_A + \lambda_A)$

The alternative that offers the perspective of the shorter delay $L = \min(L_{PA}, L_{PB})$ is selected. It can be observed that if $\lambda_A > \lambda_B$ then $L_{PA} < L_{PB}$, which means that we have to schedule P_A first so that $L = L_{PA}$; similarly if $\lambda_B > \lambda_A$ then $L_{PB} < L_{PA}$, and we have to schedule P_B first in order to get $L = L_{PB}$.

4.2 Scheduling for Time Driven Systems

We propose several extensions to the scheduling algorithm briefly described in Section 4.1. The extensions consider a realistic communication and execution infrastructure, and include aspects of the communication protocol in the optimization process.

Thus, as an input to our problem we consider a safety-critical application that has several operating modes, and each mode is modelled by a conditional process graph. The architecture of the system is given as described in the Section 2.2. Each process of the process graph is mapped on a CPU or an ASIC of a node. The worst case execution time (WCET) for each process mapped on a processing element is known, as well as the length b_{mi} of each message.

We are interested to derive a worst case delay on the system execution time for each operating mode, so that this delay is as small as possible, and to synthesize the local schedule tables for each node, as well as the MEDL for the TTP controllers, which guarantee this delay.

Considering the concrete definition of our problem, the communication time is no longer dependent only on the length of the message, as assumed in the previous section. Thus, if the message is sent between two processes mapped onto different nodes, the message has to be scheduled according to the TTP protocol. Several messages can be packaged together in the data field of a

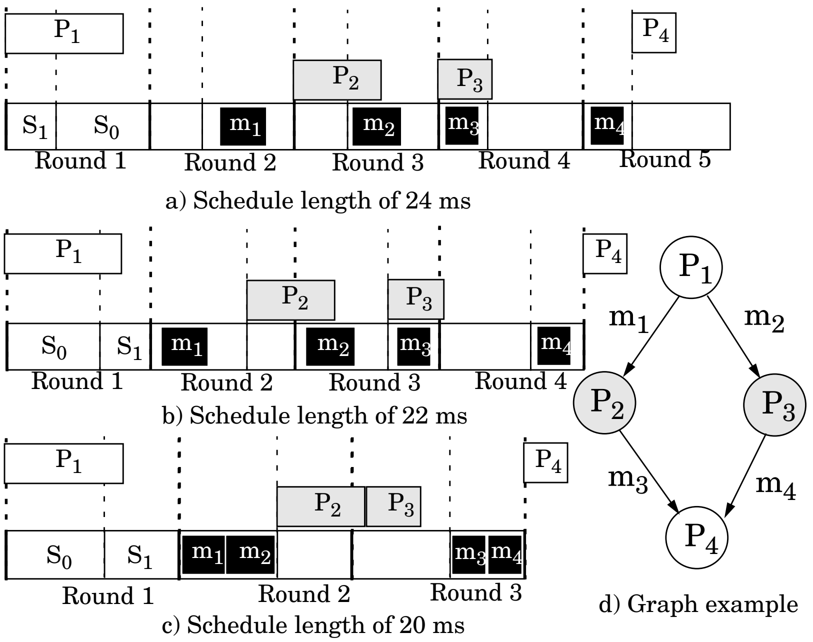
P1
P2
P3
P4
S1
S0
m1
m2
m3
m4
Round 1
Round 2
Round 3
Round 4
Round 5
a) Schedule length of 24 ms
b) Schedule length of 22 ms
c) Schedule length of 20 ms
d) Graph example

Figure 4.3: Scheduling Example

frame. The number of messages that can be packaged depends on the slot length corresponding to the node. The effective time spent by a message m_i *on the bus* is $t_{m_i} = b_{S_i}/T$, where b_{S_i} is the length of the slot S_i and T is the transmission speed of the channel. Therefore, the communication time t_{m_i} does not depend on the bit length b_{m_i} of the message m_i, but on the slot length corresponding to the node sending m_i.

The important impact of the communication parameters on the performance of the application is illustrated in Figure 4.3 by means of a simple example.

In Figure 4.3d we have a process graph consisting of four processes P_1 to P_4 and four messages m_1 to m_4. The architecture consists of two nodes interconnected by a TTP channel. The first node, N_0, transmits on the slot S_0 of the TDMA round and the second node, N_1, transmits on the slot S_1. Processes P_1 and P_4

are mapped on node N_0, while processes P_2 and P_3 are mapped on node N_1. With the TDMA configuration in Figure 4.3a, where the slot S_1 is scheduled first and slot S_0 is second, we have a resulting schedule length of 24 ms. However, if we swap the two slots inside the TDMA round without changing their lengths, we can improve the schedule by 2 ms, as seen on Figure 4.3b. Further more, if we have the TDMA configuration in Figure 4.3c where slot S_0 is first, slot S_1 is second and we increase the slot lengths so that the slots can accommodate both of the messages generated on the same node, we obtain a schedule length of 20 ms which is optimal. However, increasing the length of slots does not necessarily improve a schedule, as it delays the communication of messages generated by other nodes.

In the next two sections our goal is to synthesize the local schedule table of each node and the MEDL of the TTP controller for a given order of slots in the TDMA round and given slot lengths. The ordering of slots and the optimization of slot lengths will be discussed in Section 4.2.3.

4.2.1 Scheduling of Messages with the TTP

Given a certain bus access scheme, which means a given ordering of the slots in the TDMA round and fixed slot lengths, the CPG has to be scheduled with the goal to minimize the worst case execution delay. This can be performed using the algorithm ListScheduling (Figure 4.1) presented in Section 4.1.1. Two aspects have to be discussed here: the planning of messages in predetermined slots and the impact of this communication strategy on the priority assignment.

The function ScheduleMessage in Figure 4.4 is called in order to plan the communication of a message m, with length b_m, generated on $Node_m$ and which is ready to be transmitted at *TimeReady*. ScheduleMessage returns the first round and the corresponding slot (the slot corresponding to $Node_m$) which can host the message. In Figure 4.4 *RoundLength* is the length of a

```
ScheduleMessage ($TimeReady$, $b_m$, $Node_m$)
    -- the slot in which the message has to be sent
    $Slot$=the slot assigned to $Node_m$
    -- the first round which could be a candidate
    $Round = \lfloor TimeReady / RoundLength \rfloor$
    -- is the right slot in this round already gone?
    if $time\_ready - Round * RoundLength > start_{Slot}$ then
        -- if yes, take the next round
        $Round = Round + 1$
    end if
    -- is enough space left in the slot for the message?
    while $b_m > b_{Slot} - b_{occupied}$ do
        -- if not, take the next round
        $Round = Round + 1$
    end while
    -- return the right round and slot
    return ($Round$, $Slot$)
end ScheduleMessage
```

Figure 4.4: Message Scheduling

TDMA round expressed in time units (in Figure 4.5, for example, *RoundLength*=18 ms). The first round after *TimeReady* is the initial candidate to be considered. For this round, however, it can be too late to catch the right slot, in which case the next round is selected. When a candidate round is selected we have to check that there is enough space left in the slot for our message ($b_{occupied}$ represents the total number of bits occupied by messages already scheduled in the respective slot of that round). If no space is left, the communication has to be delayed for another round.

With this message scheduling scheme, the algorithm in Figure 4.1 will generate correct schedules for a TTP based architecture, with guaranteed worst case execution delays. However, the quality of the schedules can be much improved by adapting the priority assignment scheme so that particularities of the communication protocol are taken into consideration.

4.2.2 IMPROVED PRIORITY FUNCTION

For the scheduling algorithm outlined previously we initially used the Partial Critical Path (PCP) priority function [Dob98, Ele98b, Ele00]. PCP uses as a priority criterion the length of that part of the critical path corresponding to a process P_i which starts with the first successor of P_i that is assigned to a processor different from the processor running P_i. The PCP priority function is statically computed once at the beginning of the scheduling procedure.

However, considering the concrete definition of our problem, significant improvements of the resulting schedule can be obtained by including knowledge of the bus access scheme into the priority function. This new priority function will be used by the GetReadyProcess (Figure 4.1) in order to decide which process to select from the list of ready process.

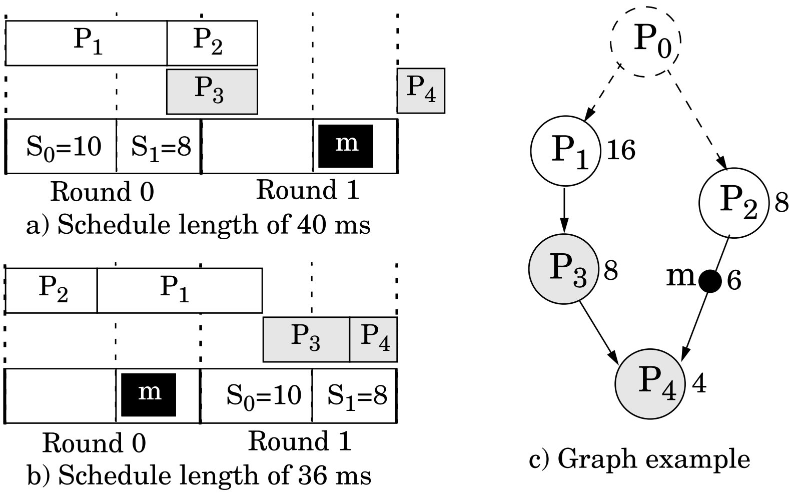

Figure 4.5: Priority Function Example

Let us consider the graph in Figure 4.5c, and suppose that the list scheduling algorithm has to decide between scheduling process P_1 or P_2 which are both ready to be scheduled on the same programmable processor. The worst case execution time of the processes is depicted on the right side of the respective node and is expressed in ms. The architecture consists of two nodes interconnected by a TTP channel. Processes P_1 and P_2 are mapped on node N_1, while processes P_3 and P_4 are mapped on node N_0. Node N_0 transmits on slot S_0 of the TDMA round and N_1 transmits on slot S_1. Slot S_0 has a length of 10 ms while slot S_1 has a length of 8 ms. For simplicity we suppose that there is no message transferred between P_1 and P_3. PCP (see Section 4.1.2) assigns a higher priority to P_1 because it has a partial critical path of 12, starting from P_3, longer than the partial critical path of P_2 which is 10 and starts from m. This results in a schedule length of 40 ms as depicted in Figure 4.5a. On the other hand, if we schedule P_2 first, the resulting schedule, depicted in Figure 4.5b, is of only 36 ms.

This apparent anomaly is due to the fact that the way we have computed PCP priorities, considering message communication as a simple activity of delay 6ms, is not realistic in the context of a TDMA protocol. Let us consider the particular TDMA configuration in Figure 4.5 and suppose that the scheduler has to decide at t=0, which one of the processes P_1 or P_2 to schedule. If P_2 is scheduled, the message is ready to be transmitted at t'=8. Based on a computation similar to that used in Figure 4.5, it follows that message m will be placed in round $\lfloor 8/18 \rfloor = 0$, and it arrives in time to get slot S_1 of that round ($TimeReady$=8 < $start_{S1}$=10). Thus, m arrives at t_{arr}=18, which means a delay relative to t'=8 (when the message was ready) of δ=10. This is the delay that should be considered for computing the partial critical path of P_2, which now results in $\delta + t_{P4}$=14 (longer than the one corresponding to P_1).

The obvious conclusion is that priority estimation has to be based on message planning with the TDMA scheme. Such an estimation, however, cannot be performed statically, before scheduling. If we take the same example in Figure 4.5, but consider that the priority based decision is taken by the scheduler at t=5, m will be ready at t'=13. This is too late for m to get into slot S_1 of round 0. The message arrives with round 1 at t_{arr}=36. This leads to a delay due to the message passing of δ=36-13=23, different from the one computed above.

We introduce a new priority function, the modified PCP (MPCP), which is computed during scheduling, whenever several processes are in competition to be scheduled on the same resource. Similar to PCP, the priority metric is the length of that portion of the critical path corresponding to a process P_i which

```
Lambda(lambda, CurrentProcess)
    if CurrentProcess is a message then
        slot = slot of node sending CurrentProcess
        round = lambda / RoundLength
        if lambda - RoundLength * round > start of slot in round
            round = next round
        end if
        while not message fits in the slot of round then
            round = next round
        end while
        lambda = round * RoundLength +
            start of slot in round + length of slot
    else
        lambda = lambda + WCET of CurrentProcess
    end if
    if lambda > MaxLambda
     MaxLambda = lambda
    end if
    for each successor of CurrentProcess
        Lambda(lambda, successor)
    end for
    return MaxLambda
end Lambda
```

Figure 4.6: The Lambda Function

starts with the first successor of P_i that is assigned to a processor different from $M(P_i)$. The critical path estimation starts with time t at which the processes in competition are ready to be scheduled on the available resource. During the partial traversal of the graph the delay introduced by a certain node P_j is estimated as follows:

$$\delta_{Pj} = \begin{cases} t_{Pj}, \text{ if } P_j \text{ is not a message passing} \\ t_{arr}\text{-}t', \text{ if } P_j \text{ is a message passing} \end{cases}$$

t' is the time when the node generating the message terminates (and the message is ready); t_{arr} is the time when the slot to which the message is supposed to be assigned has arrived. The slot is determined like in Figure 4.4, but without taking into consideration space limitations in slots.

Thus, the priority function MPCP has to be dynamically determined during the scheduling algorithm for each ready process, every time the GetReadyProcess function is activated in order to select a process from the ReadyList. The computation of MPCP is performed inside the GetReadyProcess function and involves a partial traversal of the graph, as presented in Figure 4.6.

As the experimental results (Section 4.3) show, using MPCP instead of PCP for the TTP based architecture results in an important improvement of the quality of generated schedules, with a slight increase in scheduling time.

4.2.3 COMMUNICATION SYNTHESIS

In the previous subsections we have shown how the algorithm ListScheduling can produce an efficient schedule for a CPG, given a certain TDMA bus access scheme. However, as shown in Figure 4.3, both the ordering of slots and the slot lengths strongly influence the worst case execution delay of the system.

We first present a heuristic which, based on a greedy approach, determines an ordering of slots and their lengths so that the worst case delay corresponding to a certain CPG is as small as possible.

Greedy Approaches. The initial solution, the so called "straightforward" one, assigns in order nodes to the slots ($Node_{Si}$=N_i) and fixes the slot length $length_{Si}$ to the minimal allowed value, which is equal to the length of the largest message generated by a process assigned to $Node_{Si}$. The algorithm

```
OptimizeAccess
    -- creates the initial, straightforward solution
    for i = 0 to NrSlot - 1 do
        Node_S = N_i
        length_S = MinLength_Si
    end for
    -- over all slots
    for i = 0 to NrSlot - 1 do
        -- over all slots which have not yet been allocated
        -- a node and slot length
        for j = i to NrSlot - 1 do
            swap values (Node_Si, length_Si) with (Node_Sj, length_Sj)
            -- initially, length_Si has the minimal allowed value
            for all slot lengths λ, larger than length_Si do
                length_S = λ
                ListScheduling( ... )
                remember BestSolution = (Node_Si, length_Si),
                    with the smallest δ_max produced by ListScheduling
            end for
            swap back values (Node_Si, length_Si) with (Node_Sj, length_Sj)
                to the state before entering the for cycle
        end for
        -- slot S_i gets a node allocated and a length fixed
        bind (Node_Si, length_Si) = BestSolution
    end for
end OptimizeAccess
```

Figure 4.7: Optimization of the Bus Access Scheme

starts with the first slot and tries to find the node which, when transmitting in this slot, will minimize the worst case delay of the system, as produced by ListScheduling. Simultaneously with searching for the right node to be assigned to the slot, the algorithm looks for the optimal slot length. Once a node was selected for the first slot and a slot length fixed, the algorithm continues with the next slots, trying to assign nodes (and to fix slot lengths) from those nodes which have not yet been assigned.

When calculating the length of a certain slot, a first alternative could be to try all the slot lengths λ allowed by the protocol. Such an approach starts with the minimum slot length determined by the largest message to be sent from the candidate node, and it continues incrementing with the smallest data unit (e.g. 2 bits) up to the largest slot length determined by the maximum allowed data field in a TTP frame (e.g., 32 bits, depending on the controller implementation). We call this alternative OptimizeAccess1. A second alternative, OptimizeAccess2, is based on a feedback from the scheduling algorithm which recommends slot sizes to be tried out. Before starting the actual optimization process for the bus access scheme, a scheduling of the straightforward solution is performed which generates the recommended slot lengths. These lengths are produced by the ScheduleMessage function (Figure 4.4), whenever a new round has to be selected because of lack of space in the current slot. In such a case the slot length which would be needed in order to accommodate the new message is added to the list of recommended lengths for the respective slot. With this alternative, the optimization algorithm in Figure 4.7 only selects among the recommended lengths when searching for the right dimension of a certain slot.

Simulated Annealing. A second algorithm we have developed is based on a simulated annealing (SA) strategy.

The greedy strategy constructs the solution by progressively selecting the best candidate in terms of the schedule length pro-

```
SimulatedAnnealing
    construct an initial TDMA round $x^{now}$
    temperature = initial temperature TI
    repeat
        for i = 1 to temperature length TL
            generate randomly a neighboring solution x' of $x^{now}$
            delta = schedule with x' - schedule with $x^{now}$
            if delta < 0 then $x^{now}$ = x'
            else
                generate q = random (0, 1)
                if $q < e^{-delta / temperature}$ then $x^{now}$ = x' end if
            end if
        end for
        temperature = $\alpha$ * temperature
    until stopping criterion is met
    return solution corresponding to the best schedule
end SimulatedAnnealing
```

Figure 4.8: The Simulated Annealing Strategy

duced by the function ListScheduling. Unlike the greedy strategy, SA tries to escape from a local optimum by randomly selecting a new solution from the neighbours of the current solution. The new solution is accepted if it is an improved solution. However, a worse solution can also be accepted with a certain probability that depends on the deterioration of the cost function and on a control parameter called temperature [Ree93].

In Figure 4.8 we give a short description of this algorithm. An essential component of the algorithm is the generation of a new solution *x*' starting from the current one x^{now}. The neighbours of the current solution x^{now} are obtained by a permutation of the slots in the TDMA round and/or by increasing/decreasing the slot lengths. We generate the new solution by either randomly swapping two slots (with a probability 0.3) and/or by increasing/

decreasing with the smallest data unit the length of a randomly selected slot (with a probability 0.7).

For the implementation of this algorithm, the parameters *TI* (initial temperature), *TL* (temperature length), α (cooling ratio), and the stopping criterion have to be determined. They define the so called cooling schedule and have a decisive impact on the quality of the solutions and the CPU time consumed. We were interested to obtain values for *TI*, *TL* and α that will guarantee the finding of good quality solutions in an acceptable time.

For graphs with 160 and less processes we were able to run an exhaustive search that found the optimal solutions. For the rest of the graph dimensions, we performed very long and expensive runs with the SA algorithm, and the best ever solution produced has been considered as the optimum for the further experiments. Based on further experiments we have determined the parameters of the SA algorithm so that the optimization time is reduced as much as possible but the optimal result is still produced. For example, for the graphs with 320 nodes, *TI* is 500, *TL* is 400 and α is 0.97. The algorithm stops if for three consecutive temperatures no new solution has been accepted.

4.3 Experimental Results

For evaluation of our scheduling algorithms we first used conditional process graphs generated for experimental purpose. We considered architectures consisting of 2, 4, 6, 8 and 10 nodes. 40 processes were assigned to each node, resulting in graphs of 80, 160, 240, 320 and 400 processes. 30 graphs were generated for each graph dimension, thus a total of 150 graphs were used for experimental evaluation. Execution times and message lengths were assigned randomly using both uniform and exponential distribution. For the communication channel we considered a transmission speed of 256 kbps and a length below 20 meters.

The maximum length of the data field was 8 bytes, and the frequency of the TTP controller was chosen to be 20 MHz. All experiments were run on a SPARCstation 20.

The first result concerns the quality of the schedules produced by the list scheduling based algorithm using PCP and MPCP priority functions. In order to compare the two priority functions we have calculated the average percentage deviations of the schedule length produced with PCP and MPCP from the length of the best schedule between the two. The results are depicted in Figure 4.9. In average the deviation with MPCP is 11.34 times smaller than with PCP. However, due to its dynamic nature, MPCP has in average a bigger execution time than PCP. The average execution times for the ListScheduling function using PCP and MPCP are depicted in Figure 4.10 and are under half a second for graphs with 400 processes.

In the next experiments we were interested to check the potential of the algorithms presented in Section 4.2.3 to improve

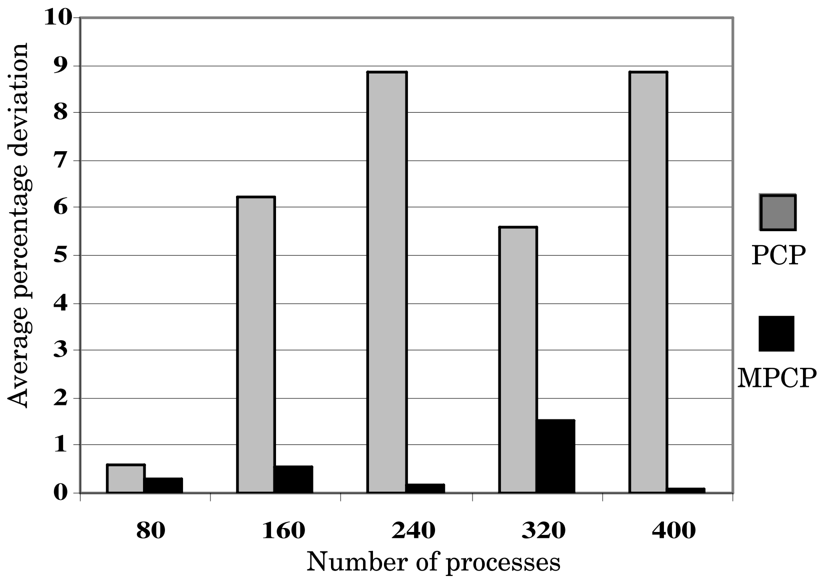

Figure 4.9: Quality of Schedules with PCP and MPCP

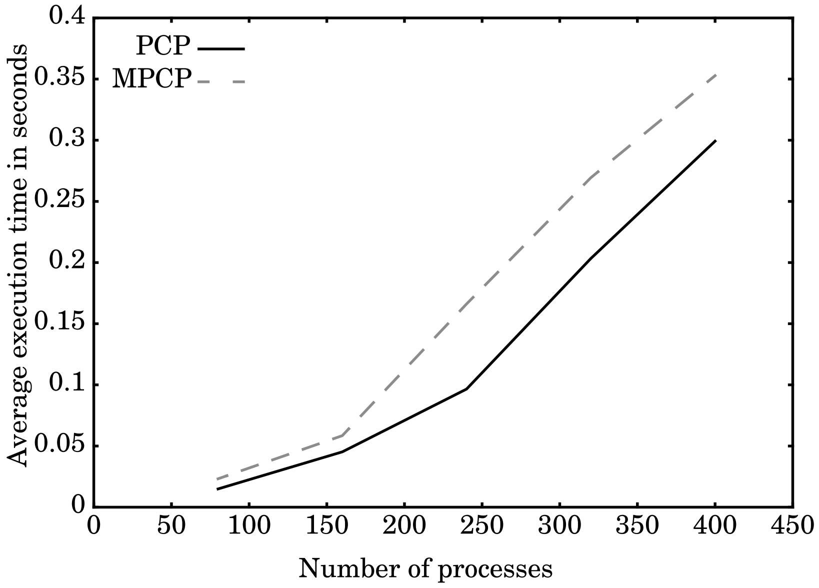

Figure 4.10: Average Exec. Times of PCP and MPCP

the generated schedules by optimizing the bus access scheme. We compared schedule lengths obtained for the 150 CPGs considering four different bus access schemes: the straightforward solution, the optimized schemes generated with the two alternatives of our greedy algorithm (OptimizeAccess1 and OptimizeAccess2) and the near-optimal scheme produced using a the simulated annealing (SA) based algorithm. Very long and extensive runs have been performed with the SA algorithm for each graph and the best ever solution produced has been considered as the near-optimum for that graph.

Table 4.1 presents the average and maximum percentage deviation of the schedule lengths obtained with the straightforward solution and with the two optimized schemes from the length obtained with the near-optimal scheme. For each of the graph dimensions, the average optimization time, expressed in seconds, is also given. The first conclusion is that by considering the optimization of the bus access scheme, the results improve

Table 4.1: Evaluation of the Bus Access Optimization Algorithm

Nr. of proc.	Straightforward solution		OptimizeAccess1			OptimizeAccess2		
	avg. dev.	max. dev.	avg. dev.	max. dev.	exec. time	avg. dev.	max. dev.	exec. time
80	3.16%	21%	0.02%	0.5%	0.25s	1.8%	19.7%	0.04s
160	14.4%	53.4%	2.5%	9.5%	2.07s	4.9%	26.3%	0.28s
240	37.6%	110%	7.4%	24.8%	10.46s	9.3%	31.4%	1.34s
320	51.5%	135%	8.5%	31.9%	34.69s	12.1%	37.1%	4.8s
400	48%	135%	10.5%	32.9%	56.04s	11.8%	31.6%	8.2s

significantly compared to the straightforward solution. The greedy heuristic performs well for all the graph dimensions. As expected, the alternative OptimizeAccess1 (which considers all allowed slot lengths) produces slightly better results, on average, than OptimizeAccess2. However, the execution times are much smaller for OptimizeAccess2. It is interesting to mention that the average execution times for the SA algorithm, needed to find the near-optimal solutions, are between 5 minutes for the CPGs with 80 processes and 275 minutes for 400 processes.

Finally, Chapter 6 presents a real-life example implementing a vehicle cruise controller.

Chapter 5 Schedulability Analysis and Communication Synthesis for Event Driven Systems

TTP HAS BEEN classically associated with non-preemptive static scheduling of processes, mainly because of fault tolerance reasons [Kop97a]. In the previous chapter we have addressed the issue of non-preemptive static process scheduling and communication synthesis using TTP.

However, considering preemptive priority based scheduling at the process level, with time triggered static scheduling at the communication level, can be the right solution under certain circumstances [Lon99]. A communication protocol like TTP, provides a global time base, improves fault-tolerance and predictability. At the same time, certain particularities of the application or of the underlying real-time operating system can impose a priority based scheduling policy at the process level.

In this chapter we consider event-driven distributed real-time systems where the activation of processes is event-triggered, while the communications are time-triggered, according to the TTP.

The chapter is structured as follows. The next section introduces some previous work on schedulability analysis that is needed for later discussions. We then show how the current state of the art schedulability analysis for distributed real-time systems can be extended to consider the time triggered protocol (Section 5.2). Section 5.3 presents the schedulability analysis we have developed for systems with both control and data dependencies modelled as a set of conditional process graphs. Once realistic communication aspects are captured by the schedulability analysis, this can be used to drive the communication synthesis process described in Section 5.4. Finally, Section 5.5 presents the experimental results obtained for the work presented in this chapter.

5.1 Schedulability Analysis

A set of processes is *schedulable* if there exists at least one scheduling algorithm that is able to produce a feasible schedule. A schedule is *feasible* if all tasks can be completed within the specified constraints.

The aim of the schedulability analysis is to determine sufficient and necessary conditions under which the system is schedulable. There are basically two approaches to this problem:

- produce exact (both sufficient and necessary) feasibility test of a set of processes within the constraints; and
- derive the worst-case response times (the longest time between the arrival of a process and its completion) and compare them to the deadlines, the so called *response time analysis*.

The schedulability analysis has its roots in [Liu73] that has provided a sufficient feasibility test (based on a utilization bound) for processes that have priorities assigned according to the rate monotonic priority assignment policy [Liu73]:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \le n(2^{1/n} - 1)$$

where n is the number of processes, C_i is the worst case execution time of process P_i, and T_i is its deadline.

There are several unrealistic assumptions in [Liu73]: the processes are considered independent, periodic, having a deadline equal to the period, etc.

The second approach, using the response time analysis to check the exact feasibility of a set of processes, has been discussed in [Aud91]. It compares the worst case response time of each process to its deadline, and if the inequalities hold, the system is schedulable:

$$r_i = C_i + \sum_{\forall j \in hp(P_i)} C_j \left\lceil \frac{r_i}{T_j} \right\rceil$$

where r_i is the response time of process P_i and $hp(P_i)$ is the set of processes that have higher priority than P_i. The authors note that the summation term increases monotonically in r_i, thus solutions can be found using a recurrence relation. This approach makes no assumptions regarding the priority assignment scheme, and allows the deadlines to be less than the periods.

A huge amount of research has been done in the last decades, that aims at relaxing the assumptions made by the previous works, and the reader is referred to the overview in [Aud95].

Of particular importance for our work are the research in [Yen98, Tin94b] that aims to relax assumptions made on the independence of processes, and the research presented in [Tin94a] that extends the existing analysis to distributed systems with a simple TDMA protocol. Thus, we will present these works in detail in the coming sections.

5.2 Schedulability Analysis with the Time Triggered Protocol

For the purpose of this section, we consider applications modelled as a set of processes. Each process P_i is allocated to a certain processor, has a known worst-case execution time C_i, a period T_i, a deadline D_i and a uniquely assigned priority. We consider a preemptive execution environment, which means that higher priority processes can interrupt the execution of lower priority processes. A lower priority process can block a higher priority process (e.g., it is in its critical section), and the blocking time is computed according to the priority ceiling protocol [Sha90]. Processes exchange messages, and for each message m_i we know its size S_{m_i}. A message is sent once in every n_m invocations of the sending process, and has a unique destination process. Each process is allocated to a node of our distributed architecture, and the messages are transmitted according to the TTP.

We are interested to synthesize the MEDL of the TTP controllers (and as a direct consequence, also the MHTTs -- see Section 2.2.3) so that the process set is schedulable on an as cheap (slow) as possible processor set.

Under these assumptions Tindell et al. [Tin94a] integrate processor and communication schedulability and provide a "holistic" schedulability analysis in the context of distributed real-time systems with communication based on a simple TDMA protocol. The basic idea is that the release jitter of a destination process depends on the communication delay between sending and receiving a message. The release jitter of a process is the worst case delay between the arrival of the process and its release (when it is placed in the run-queue for the processor). The communication delay is the worst case time spent between sending a message and the message arriving at the destination process.

Thus, for a process *d(m)* that receives a message *m* from a sender process *s(m)*, the release jitter is

$$J_{d(m)} = r_{s(m)} + a_m + r_{deliver} + T_{tick},$$

where $r_{s(m)}$ is the response time of the process sending the message, a_m (worst case arrival time) is the worst case time needed for message *m* to arrive at the communication controller of the destination node, $r_{deliver}$ is the response time of the delivery process (see Section 2.2.3), and T_{tick} is the jitter due to the operation of the tick scheduler. The communication delay for a message m is

$$C_m = a_m + r_{deliver}.$$

a_m itself is the sum of the access delay and the propagation delay. The access delay is the time a message queued at the sending processor spends waiting for the use of the communication channel. In a_m we also account for the execution time of the message transfer process (see Section 2.2.3). The propagation delay is the time taken for the message to reach the destination processor once physically sent by the corresponding TTP controller.

The worst case time message *m* takes to arrive at the communication controller of the destination node is determined in [Tin94a] using the arbitrary deadline analysis, and is given by:

$$a_m = \max_{q = 0, 1, 2, \ldots} (w_m(q) + X_m(q) - qT_m),$$

where the term $w_m(q) - qT_m$ is the access delay, $X_m(q)$ is the propagation delay, and T_m is the period of the message.

In [Tin94a] an analysis is given for the end-to-end delay of a message *m* in the case of a simple TDMA protocol. For this case,

$$w_m(q) = \left\lceil \frac{(q+1)p_m + I_m(w(q))}{S_p} \right\rceil T_{TDMA},$$

where p_m is the number of packets of message *m*, S_p is the size of the slot (in number of packets) corresponding to *m*, and I_m is the interference caused by packets belonging to messages of a higher priority than m. Although there are many similarities with the general TDMA protocol, the analysis in the case of TTP is different in several aspects and also differs to a large degree depending on the policy chosen for message scheduling.

Before going into details for each of the message scheduling approaches proposed by us, we analyze the propagation delay and the message transfer and delivery processes, as they do not depend on the particular message scheduling policy chosen. The propagation delay X_m of a message m sent as part of a slot S, with the TTP protocol, is equal to the time needed for the slot S to be transferred on the bus. This time depends on the slot size and on the features of the underlying bus.

The overhead produced by the communication activities must be accounted for not only as part of the access delay for a message, but also through its influence on the response time of processes running on the same processor. We consider this influence during the schedulability analysis of processes on each processor. We assume that the worst case computation time of the transfer process (T in Figure 2.5) is known, and that it is different for each of the four message scheduling approaches. Based on the respective MHTT, the transfer process is activated for each frame sent. Its worst case period is derived from the minimum time between successive frames.

The response time of the delivery process (D in Figure 2.5), r_{deliver}, is part of the communication delay. The influence due to the delivery process must be also included when analyzing the response time of the processes running on the respective processor. We consider the delivery process during the schedulability analysis in the same way as the message transfer process.

The response times of the communication and delivery processes are calculated, as for all other processes, using the arbitrary deadline analysis from [Tin94a].

The four approaches we have considered for scheduling of messages using TTP differ in the way the messages are allocated to the communication channel (either statically or dynamically) and whether they are split or not into packets for transmission. The next subsections present an analysis for these approaches as well as the degrees of liberty a designer has, in each of the cases, when synthesizing the MEDL.

5.2.1 STATIC SINGLE MESSAGE ALLOCATION (SM)

The first approach to scheduling of messages using TTP is to statically (off-line) schedule each of the messages into a slot of the TDMA cycle, corresponding to the node sending the message. We also consider that the slots can hold each at maximum one single message. This approach is well suited for application areas (like automotive electronics) where the messages are typically short and the ability to easily diagnose the system is critical.

As each slot carries only one fixed, predetermined message, there is no interference among messages. If a message m misses its slot it has to wait for the following slot assigned to m. The access delay for a message m in this approach is the maximum time between consecutive slots of the same node carrying the message m. We denote this time by $T_{m_{max}}$, illustrated in Figure 5.1.

In this case, the worst case arrival time a_m of a message m becomes $T_{m_{max}} + X_m$. Therefore, the main aspect influencing the schedulability analysis for the messages is the way the messages are statically allocated to slots, resulting different values for $T_{m_{max}}$. $T_{m_{max}}$, as well as X_m, depend on the slot sizes which in the case of SM are determined by the size of the largest message sent from the corresponding node, plus the bits for control and CRC, as imposed by the protocol.

During the synthesis of the MEDL, the designer has to allocate the messages to slots in such a way that the process set is schedulable. Since the schedulability of the process set can be influenced by the synthesis of the MEDL only through the $T_{m_{max}}$ parameters, these parameters have to be optimized.

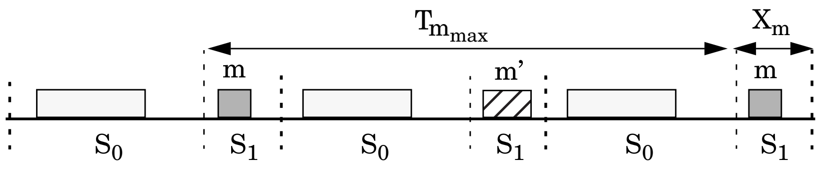

Figure 5.1: Worst case arrival time for SM

Let us consider the simple example depicted in Figure 5.2, where we have three processes, P_1, P_2, and P_3 running each on different processors. When process P_1 finishes executing it sends message m_1 to process P_3 and message m_2 to process P_2. In the TDMA configuration presented in Figure 5.2a, only the slot for the CPU running P_1 is important for our discussion and the other slots are represented with light gray. With this configuration, where the message m_1 is allocated to the rounds 1 and 4 and the message m_2 is allocated to rounds 2 and 3, process P_2 misses its deadline because of the release jitter due to the message m_2 in round 2. However, if we have the TDMA configuration depicted in Figure 5.2b, where m_1 is allocated to the rounds 2 and 4 and m_2 is allocated to the rounds 1 and 3, then all the processes meet their deadlines.

5.2.2 STATIC MULTIPLE MESSAGE ALLOCATION (MM)

This second approach is an extension of the first one. In this approach we allow more than one message to be statically assigned to a slot, and all the messages transmitted in the same slot are packaged together in a frame. In this case there is also no interference, so the access delay for a message m is the same as for the first approach, namely, the maximum time between consecutive slots of the same node carrying the message m, $T_{m_{max}}$.

However, this approach offers more freedom during the synthesis of the MEDL. We have now to decide also on how many and which messages should be put in a slot. This allows more

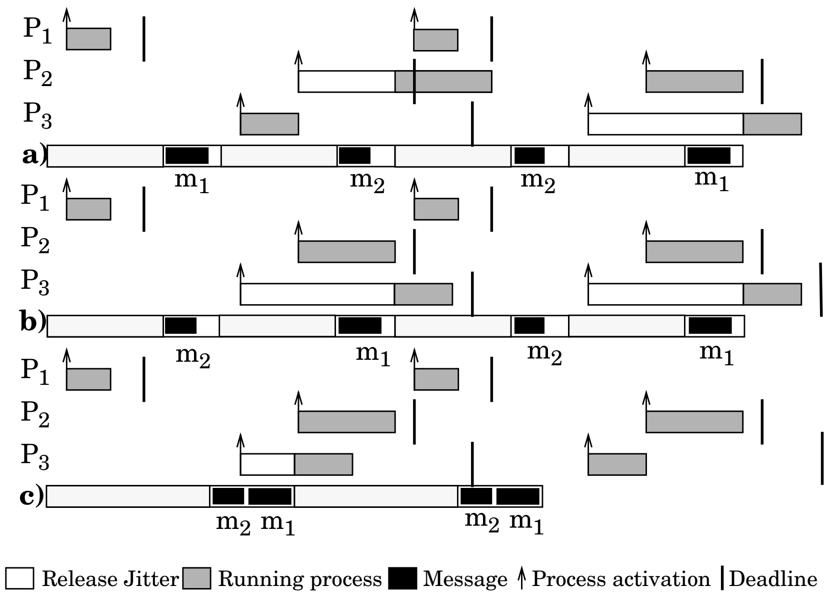

Figure 5.2: Optimizing the MEDL for SM and MM

flexibility in optimizing the $T_{m_{max}}$ parameter. To illustrate this, let us consider the same example depicted in Figure 5.2. With the MM approach, the TDMA configuration can be arranged as depicted in Figure 5.2c, where the messages m_1 and m_2 are put together in the same slot in the rounds 1 and 2. Thus, the deadline is met, and the release jitter is further reduced compared to the case presented in Figure 5.2b where the deadlines were also met but the process P_3 was experiencing large release jitter.

5.2.3 DYNAMIC MESSAGE ALLOCATION (DM)

The previous two approaches have statically allocated one or more messages to their corresponding slots. This third approach considers that the messages are dynamically allocated to frames, as they are produced.

Thus, when a message is produced by a sender process it is placed in the *Out* queue ordered according to the priorities of the messages. At its activation, the message transfer process takes a certain number of messages from the head of the *Out* queue and constructs the frame. The number of messages accepted is decided so that their total size does not exceed the length of the data field of the frame. This length is limited by the size of the slot corresponding to the respective processor. Since the messages are sent dynamically, we have to identify them in a certain way so that they are recognized when the frame arrives at the delivery process. We consider that each message has several identifier bits appended at the beginning of the message.

Since we dynamically package the messages into frames in the order they are sorted in the queue, the access delay to the communication channel for a message m depends on the number of messages queued ahead of it.

The analysis in [Tin94a] bounds the number of queued ahead *packets* of messages of higher priority than message m, as in their case it is considered that a message can be split into packets before it is transmitted on the communication channel. We use the same analysis, but we have to apply it for the number of *messages* instead that of packets. We have to consider that messages can be of different sizes as opposed to packets which always are of the same size.

Therefore, the total *size* of higher priority messages queued ahead of a message m in a window w is:

$$I_m(w) = \sum_{\forall j \in hp(m)} \left\lceil \frac{w + r_{s(j)}}{T_j} \right\rceil S_j,$$

where S_j is the size of the message m_j, $r_{s(j)}$ is the response time of the process sending message m_j, and T_j is the period of the message m_j.

Further, we calculate the worst case time that a message m spends in the *Out* queue. The number of TDMA rounds needed, in the worst case, for a message m placed in the queue to be removed from the queue for transmission is

$$\left\lceil \frac{S_m + I_m}{S_s} \right\rceil,$$

where S_m is the size of the message m and S_s is the size of the slot transmitting m (we assume, in the case of DM, that for any message x, $S_x \leq S_s$). This means that the worst case time a message m spends in the *Out* queue is given by

$$\left\lceil \frac{S_m + I_m}{S_s} \right\rceil T_{TDMA},$$

where T_{TDMA} is the time taken for a TDMA round.

To determine the term $w_m(q) - qT_m$ that gives the access delay (see Section 4), $w_m(q)$ is determined, using the arbitrary deadline analysis, as being:

$$w_m(q) = \left\lceil \frac{(q+1)S_m + I_m(w(q))}{S_s} \right\rceil T_{TDMA}.$$

Since the size of the messages is given with the application, the parameter that will be optimized during the synthesis of the MEDL is the slot size. To illustrate how the slot size influences the schedulability, let us consider the example in Figure 5.3a, where we have the same setting as for the example in Figure 5.2a. The difference is that we consider message m_1 having a higher priority than message m_2, and we schedule dynamically the messages as they are produced. With the TDMA configuration in Figure 5.3a message m_1 will be dynamically scheduled first in the slot of the first round, while message m_2 will wait in the *Out* queue until the next round comes, thus causing the process P_2 to miss its deadline. However, if we enlarge the slot so that it can accommodate both messages, message m_2 does not have to wait in the queue and it is transmitted in the same slot as m_1. Therefore P_2 will meet its deadline, as presented in Figure 5.3b. However, in general, increasing the length of slots does not necessarily improve the schedulability, as it delays the communication of messages generated by other nodes.

5.2.4 DYNAMIC PACKETS ALLOCATION (DP)

This approach is an extension of the previous one, as we allow the messages to be split into packets before they are transmitted on the communication channel. We consider that each slot has a size that accommodates a frame with the data field being a multiple of the packet size. This approach is well suited for the application areas that typically have large message sizes, and by splitting them into packets we can obtain a higher utilization of the bus and reduce the release jitter. However, since each packet has to be identified as belonging to a message, and messages have to be split at the sender and reconstructed at the destination, the overhead becomes higher than in the previous approaches.

For the analysis we use the formula from [Tin94a] which is based on similar assumptions as those for this approach:

$$w_m(q) = \left\lceil \frac{(q+1)p_m + I_m(w(q))}{S_p} \right\rceil T_{TDMA},$$

where p_m is the number of packets of message m, S_p is the size of the slot (in number of packets) corresponding to m, and

$$I_m(w) = \sum_{\forall j \in hp(m)} \left\lceil \frac{w + r_{s(j)}}{T_j} \right\rceil p_j,$$

where p_j is the number of packets of a message m_j.

In the previous approach (DM) the optimization parameter for the synthesis of the MEDL was the size of the slots. Within this approach we can also decide on the packet size, which becomes another optimization parameter. Consider the example in Figure 5.3c where messages m_1 and m_2 have a size of 6 bytes each. The packet size is considered to be 4 bytes and the slot corresponding to the messages has a size of 12 bytes (3 packets) in the TDMA configuration. Since message m_1 has a higher priority than m_2, it will be dynamically scheduled first in the slot of the first round, and it will need 2 packets. In the remaining packet, the first 4 bytes of m_2 are scheduled. Thus, the rest of 2 bytes from message m_2 have to wait for the next round, causing the process P_2 to miss its deadline. However, if we change the packet size to 3 bytes, and keep the same size of 12 bytes for the slot, we now have 4 packets in the slot corresponding to the CPU running P_1 (Figure 5.3d). Message m_1 will be dynamically scheduled first, and will take 2 packets from the slot of the first round. This will allow us to send m_2 in the same round, therefore meeting the deadline for P_2.

In this particular example, with one single sender processor and the particular message and slot sizes as given, the problem seems to be simple. This is, however, not the case in general. For example, the packet size which fits a particular node can be unsuitable in the context of the messages and slot size corre-

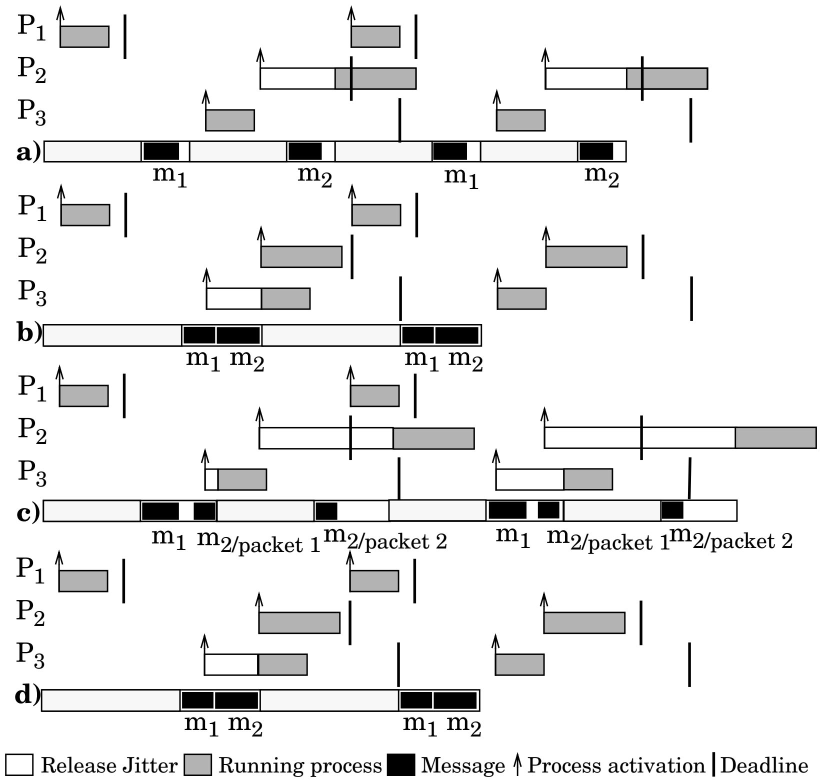

Figure 5.3: Optimizing the MEDL for DM and DP

sponding to another node. At the same time, reducing the packets size increases the overheads due to the transfer and delivery processes.

5.3 Schedulability Analysis under Control and Data Dependencies

In the previous section we were interested to extend the current response-time analysis to take into consideration the time-triggered protocol. For this we have modelled an application as a set of processes.

In this section we are interested to extend this model to handle both data and control dependencies. Thus, we consider applications modelled as a set ψ of n conditional process graphs Γ_i, $i = 1..n$. Every process P_i in such a graph is mapped to a certain processor, has a known worst-case execution time C_i, a deadline D_i, and a uniquely assigned priority. All processes belonging to the same CPG Γ_i have the same period $T_{\Gamma i}$ which is the period of the respective conditional process graph. Each CPG in the application has its own independent period. Typically, global deadlines $\delta_{\Gamma i}$ on the delay of each CPG are imposed and not individual deadlines on processes.

We consider a priority based preemptive execution environment, which means that higher priority processes will interrupt the execution of lower priority processes. A lower priority process can block a higher priority process (e.g., it is in its critical section), and the blocking time is computed according to the priority ceiling protocol [Sha90].

We are interested to develop a schedulability analysis for a system modelled as a set of conditional process graphs. For the rest of the section we will consider that global deadlines are imposed on each CPG. The approach can be easily extended if individual deadlines are imposed on processes.

To show the relevance of our problem, let us consider the example depicted in Figure 5.4, where we have a system modelled as two conditional process graphs Γ_1 and Γ_2 with a total of 9 processes (processes P_0, P_8, P_9 and P_{12} are dummy processes and are not counted), and one condition. The processes are mapped on three different processors as indicated by the shading in Figure 5.4, and the worst case execution time in milliseconds for each process on its respective processor is depicted to the left of each node. Γ_1 has a period of 200 ms, Γ_2 has a period of 150 ms. The deadlines are 100 ms on Γ_1 and 90 ms on Γ_2.

Table 5.1 presents the estimated worst case delay on the two graphs. In the column labelled “no conditions” we have the results for the case when the analysis is applied to the set of processes, ignoring control dependencies. This results in a worst case delay of 120 ms for Γ_1 and 82 ms for Γ_2. Thus, the system is considered to be not schedulable.

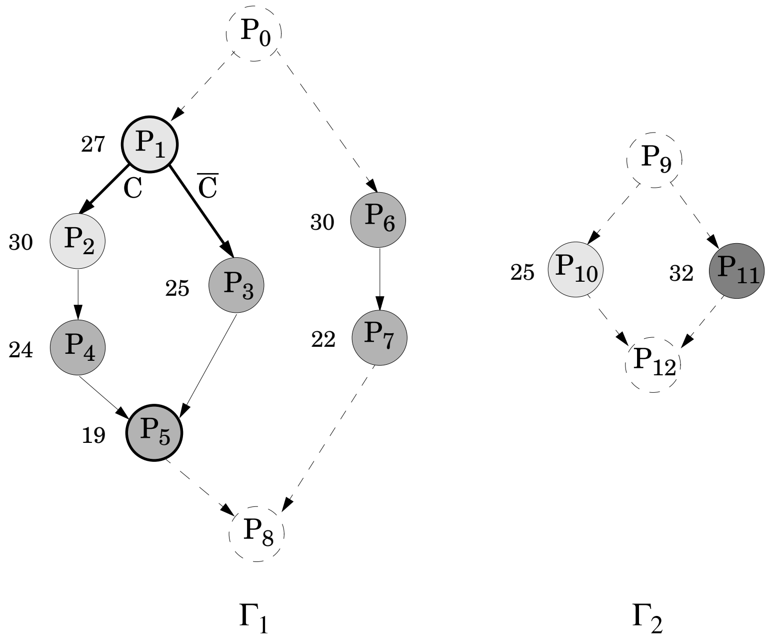

Figure 5.4: System with Control and Data Dependencies

Table 5.1: : Worst Case Delays for the System in Figure 5.4

CPG	Worst Case Delays	
	no conditions	conditions
Γ_1	120	100
Γ_2	82	82

However, this analysis assumes as a worst case scenario the possible activation of all nine processes for each execution of the system. This is the solution which will be obtained using a data-flow graph representation of the system. However, considering the CPG Γ_1 in Figure 5.4, it is easy to observe that process P_3 on the one side and processes P_2 and P_4 on the other side will not be activated during the same period of Γ_1.

Making use of this information for the analysis we obtain a worst case delay of 100 ms, for Γ_1, as shown in Table 5.1 in the column headed "conditions", which indicates that the system is schedulable.

5.3.1 TASKS WITH DATA DEPENDENCIES

Methods for schedulability analysis of data dependent processes with static priority preemptive scheduling have been proposed in [Yen98] and [Tin94b].

They use the concept of *offset* or *phase*, respectively, in order to handle data dependencies. [Tin94b] shows that the pessimism of the analysis is reduced through the introduction of offsets. The offsets have to be determined by the designer.

[Yen98] provides a framework that iteratively finds the phases (offsets) for all processes, and then feeds them back into the schedulability analysis which in turn is used again to derive better phases. Thus, the pessimism of the analysis is iteratively reduced.

We have used the framework provided by [Yen98] as a starting point for our analysis. The response time of a process P_i is:

$$r_i = C_i + \sum_{\forall j \in hp(P_i)} C_j \left\lceil \frac{r_i - O_{ij}}{T_j} \right\rceil \quad (1)$$

where $hp(P_i)$ is the set of processes that have higher priority than P_i, and O_{ij} is the phase of P_j relative to P_i.

As a first step, we have extended this analysis to real-time systems that use the time-triggered protocol as the underlying communication infrastructure. This aspect has been discussed in Section 5.2.

In [Yen98] a system is modelled as a set S of n *task graphs* G_i, $i = 1..n$. The system model assumed and the definition of a task graph are similar to our CPG, but without considering any conditions. The aim of the schedulability analysis in [Yen98] is to derive an as tight as possible worst case delay on the execution time of each of the task graphs in the system. This delay estimation is done using the algorithm DelayEstimate described in Figure 5.5.

At the core of this algorithm is a worst case response time calculation based on offsets, similar to the analysis in [Tin94b]. Thus, in the LatestTimes function worst case response times and upper bounds for the offsets are calculated, while the EarliestTimes function calculates the lower bounds of the offsets.

The LatestTimes function is a modified critical-path algorithm that calculates for each node of the graph the longest path to the sink node. Thus, during the topological traversal of the graph G within LatestTimes, for each process P_i, the worst case response time r_i is calculated according to the equation (1). This value is based on the values of the offsets known so far. Once an r_i is calculated, it can be used to determine and update offsets for other successor processes. Accordingly, the EarliestTimes function determines the lower bounds on the offsets. The influence on graph G from other graphs in the system is considered in both of the functions mentioned earlier.

These calculations can be improved by realizing that for a process P_i, there might exist a process P_j mapped on the same processor, with priority(P_i) < priority(P_j), such that their execution windows never overlap. In this case, the term in the equation (1) that expresses the influence of P_j on the execution of P_i can be dropped, resulting in a tighter worst case response time calculation. This situation is expressed through the so called *maxsep* table, computed by the MaxSeparations function, whose value *maxsep*[P_i, P_j] is less than or equal to 0 if the two processes never overlap during their execution. *maxsep* stands for *maximum separation*, an analysis modified from [Mc92] that builds

```
DelayEstimate(task graph G, system S)
--      derives the worst case delay of a task graph G considering
--      the influence from all other task graphs in the system S
        for each pair ($P_i$, $P_j$) in G
                maxsep[$P_i$, $P_j$] = $\infty$
        end for
        step = 0
        repeat
                LatestTimes(G)
                EarliestTimes(G)
                for each $P_i \in G$
                        MaxSeparations($P_i$)
                end for
        until maxsep is not changed or step < limit
        return the worst case delay $\delta_G$ of the graph G
end DelayEstimate

SchedulabilityTest(system S)
--      derives the worst case delay for each task graph in the system
--      and verifies if the deadlines are met
        for each task graph $G_i \in S$
                DelayEstimate($G_i$, S)
        end for
        if all task graphs meet their deadline system S is schedulable
end SchedulabilityTest
```

Figure 5.5: Delay Estimation and Schedulability Analysis for Task Graphs

the *maxsep* table based on the worst case execution times and offsets determined in EarliestTimes and LatestTimes.

Having a better view on the maximum separation between each pair of processes, tighter worst case execution times and offsets can be derived, which in turn contribute to the update of the *maxsep* table. This iterative tightening process is repeated until there is no modification to the *maxsep* table, or a certain imposed *limit* on the number of iterations is reached.

Finally, the DelayEstimate function returns the worst-case delay δ_G estimated for a task graph G, as the latest time when the sink node of G can finish its execution. Based on the delays produced by DelayEstimate, the function SchedulabilityTest in Figure 5.5 concludes on the schedulability of the system.

5.3.2 CONDITIONAL PROCESS GRAPHS

Section 2.1.1 has presented the conditional process graph. Before introducing our schedulability analysis for CPGs, we reinforce two concepts: the *unconditional subgraphs* and the *process guards*.

Depending on the values calculated for the conditions, different alternative paths through a conditional process graph are activated for a given activation of the system. To model this, a boolean expression X_{P_i}, called guard (introduced in Section 2.1.1), can be associated to each node P_i in the graph. It represents the necessary condition for the respective process to be activated. In Figure 5.8, for example, $X_{P4}=C \wedge \overline{D}$, $X_{P5}=\overline{C}$, $X_{P9}=true$, $X_{P11}=true$, and $X_{P12}=K$.

We call an alternative path through a conditional process graph, resulting from a combination of conditions, an *unconditional subgraph*, denoted by g. For example, the CPG Γ_1 in Figure 5.8 has three unconditional subgraphs, corresponding to the following three combinations of conditions: $C \wedge D$, $C \wedge \overline{D}$, and $\overline{C}$. The unconditional subgraph corresponding to the combina-

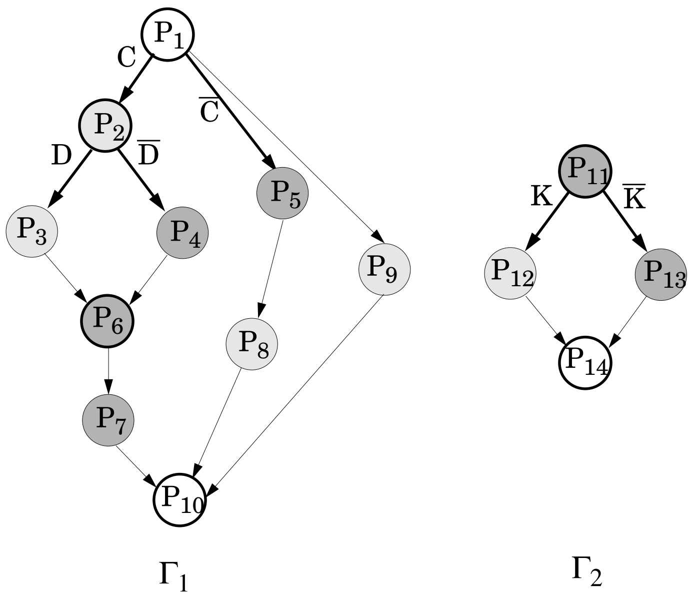

Figure 5.6: Example of two CPGs

tion $C \wedge \overline{D}$ in the CPG Γ_1 consists of processes P_1, P_2, P_4, P_6, P_7, P_9 and P_{10}.

The guards of each process, as well as the unconditional subgraphs resulting from a conditional process graph Γ can be determined through a simple recursive topological traversal of Γ.

Ignoring Conditions (IC). A straightforward approach to the schedulability analysis of systems represented as CPGs is to ignore control dependencies and to apply the schedulability analysis as described in Section 5.3.1 (the algorithm SchedulabilityTest in Figure 5.5).

This means that conditional edges in the CPGs are considered like simple edges and the conditions in the model are dropped. What results is a system S consisting of simple task graphs G_i, each one resulted from a CPG Γ_i of the given system ψ. The system S can then be analyzed using the algorithm in Figure 5.7. It

```
SA/IC(system $\psi$)
--      verifies the schedulability of a system consisting of a set of
--      conditional process graphs
        transform each $\Gamma_i \in \psi$ into the corresponding $G_i \in S$
        SchedulabilityTest($S$)
        if $S$ is schedulable, system $\psi$ is schedulable
end SA/IC
```

Figure 5.7: Schedulability Analysis Ignoring Conditions

is obvious that if the system S is schedulable, the system ψ is also schedulable.

This approach, which we call IC, is, of course, very pessimistic. However, this is the current practice when worst case arrival periods are considered and classical data flow graphs are used for modeling and scheduling.

Brute Force Solution (BF). The pessimism of the previous approach can be reduced by using a conditional process graph model. A simple, brute force solution is to apply the schedulability analysis presented in Section 5.3.1, after the CPGs have been decomposed into their constituent unconditional subgraphs.

Consider a system ψ which consists of n CPGs Γ_i, $i = 1..n$. Each CPG Γ_i can be decomposed into n_i unconditional subgraphs g_j^i, $j = 1..n_i$. In Figure 5.6, for example, we have 3 unconditional subgraphs g_1^1, g_2^1, g_3^1 derived from Γ_1 and two, g_1^2, g_2^2 derived from Γ_2.

At the same time, each CPG Γ_i can be transformed into a simple task graph G_i, by transforming conditional edges into ordinary ones and dropping the conditions. When deriving the worst case delay on Γ_i we apply the analysis from Section 5.3.1 (algorithm DelayEstimate in Figure 5.5) separately to each unconditional subgraph g_j^i in combination with the graphs (G_1, G_2, ... G_{i-1}, G_{i+1}, G_n). This means that we consider each alternative path from Γ_i in the context of the system, instead of the whole subgraph G_i as in the previous approach. This is described by the algorithm DE/CPG in Figure 5.8a. The schedulability analysis is then based on the delay estimation for each CPG as shown in the algorithm SA/BF in Figure 5.8b.

Such an approach, we call it BF, while producing tight bounds on the delays, can be expensive from the runtime point of view, because it is applied for each unconditional subgraph. In general, the number of unconditional subgraphs can grow exponentially. However, for many of the practical systems this is not the case, and the brute force method can be used. Alternatively, less expensive methods, like those presented below, should be applied.

```
DE/CPG(CPG $\Gamma$, system $S$)
--      derives the worst case delay of a CPG $\Gamma$ considering
--      the influence from all other task graphs in the system $S$
        extract all unconditional subgraphs $g_j$ from $\Gamma$
        for each $g_j$
            DelayEstimate($g_j$, $S$)
        end for
        return the largest of the delays, which is
           the worst case delay $\delta_\Gamma$ of CPG $\Gamma$
end DE/CPG

a) DE/CPG -- Delay Estimate for Conditional Process Graphs


SA/BF(system $\psi$)
--      verifies the schedulability of a system consisting of a set $\psi$ of
--      conditional process graphs
        transform each $\Gamma_i \in \psi$ into the corresponding $G_i \in S$
        for each $\Gamma_i \in \psi$
            DE/CPG($\Gamma_i$, {$G_1$, $G_2$, ...$G_{i-1}$, $G_{i+1}$, $G_n$})
        end for
        if all CPGs meet their deadline the system $\psi$ is schedulable
end SA/BF

b) SA/BF -- Schedulability Analysis: the Brute Force approach
```

Figure 5.8: Brute Force Schedulability Analysis

Condition Separation (CS) In some situations, the explosion of unconditional subgraphs makes the brute force method inapplicable. Thus, we need to find an analysis that is situated

```
SA/CS(system $\psi$)
--    verifies the schedulability of a system consisting of a set $\psi$ of
--    conditional process graphs
      transform each $\Gamma_i \in \psi$ into the corresponding $G_i \in S$
          and keep guard $X_{Pi}$ for each $P_i$
      for each $G_i \in S$
          -- derives the worst case delay of a task graph $G_i$
          -- considering the influence from all other task graphs
          -- in the system $S$
          for each pair ($P_i$, $P_j$) in $G_i$
              $maxsep[P_i, P_j] = \infty$
          end for
          $step = 0$
          repeat
              LatestTimes($G_i$)
              EarliestTimes($G_i$)
              for each $P_i \in G_i$
                  MaxSeparations($P_i$)
              end for
              for each pair ($P_i$, $P_j$) in $G_i$
                  if $\exists C, C \subset X_{Pi} \wedge \overline{C} \subset X_{Pj}$ then
                      $maxsep[P_i, P_j] = 0$
                  end if
              end for
          until maxsep is not changed or $step < limit$
      $\delta_{\Gamma i}$ is the worst case delay for $\Gamma_i$
      end for
      if all CPGs meet their deadline, the system $\psi$ is schedulable
end SA/CS
```

Figure 5.9: Schedulability Analysis using Condition Separation

somewhere between the two alternatives IC and BF, which means its should be not too pessimistic and should run in acceptable time.

A first idea is to go back to the DelayEstimate algorithm in Figure 5.5, and use the knowledge about conditions in order to update the *maxsep* table. Thus, if two processes P_i and P_j never overlap their execution because they execute under alternative values of conditions, then we can update *maxsep[*P_i*,* P_j*]* to 0, and thus improve the quality of the delay estimation. Two processes P_i and P_j never overlap their execution if there exists at least one condition C, so that $C \subset X_{Pi}$ (X_{Pi} is the guard of process P_i) and $\overline{C} \subset X_{Pj}$.

In this approach, called CS, we practically use the same algorithm as for ordinary task graphs and try to exploit the information captured by conditional dependencies in order to exclude certain influences during the analysis. In Figure 5.9 we show the algorithm SA/CS which performs the schedulability analysis based on this heuristic.

Relaxed Tightness Analysis (RT). The two approaches discussed here are similar to the brute force algorithm (Figure 5.8). However, they try to improve on the execution time of the analyses by reducing the complexity of the DelayEstimate algorithm (Figure 5.5) which is called from the DE/CPG function (Figure 5.8 a). This will reduce the execution time of the analysis, not by reducing the number of subgraphs which have to be visited (like in the CS approach), but by reducing the time needed to analyze each subgraph. As our experimental results show (Section 5.5), this approach can be very effective in practice. Of course, by the simplification applied to DelayEstimate the quality of the analysis is reduced in comparison to the brute force method.

We have considered two alternatives of which the first one is more drastic while the second one is trying a more refined tradeoff between execution time and quality of the analyses.

```
**DelayEstimateRT1**(task graph G, system S)
        LatestTimes(G)
end DelayEstimateRT1
```

a) Delay Estimation for RT1

```
DelayEstimateRT2(task graph G, system S)
        for each pair ($P_i$, $P_j$) in $G_i$
                maxsep[$P_i$, $P_j$] = $\infty$
        end for
        LatestTimes(G)
        EarliestTimes(G)
        for each $P_i \in G$
                MaxSeparations($P_i$)
        end for
        LatestTimes(G)
end DelayEstimateRT2
```

b) Delay Estimation for RT2

Figure 5.10: Delay Estimation for the RT Approaches

With both these approaches, the idea is not to run the iterative tightening loop in DelayEstimate that repeats until no changes are made to *maxsep* or until the *limit* is reached. While this tightening loop iteratively reduces the pessimism when calculating the worst case response times, the actual calculation of the worst case response times is done in LatestTimes, and the rest of the algorithm in Figure 5.5 just tries to improve on these values. For the first approach, called RT1 the function DelayEstimate has been transformed like in Figure 5.10a.

However, it might be worth using at least the MaxSeparations in order to obtain tighter values for the worst case response times. For the alternative RT2 in Figure 5.10b, DelayEstimateRT2 first calls LatestTimes and EarliestTimes, then MaxSeparations in order to build the *maxsep* table, and again LatestTimes to tighten the worst case response times.

5.4 Communication Synthesis

Once a schedulability analysis for event-driven distributed real-time systems is in place, our problem is to synthesize the communications. This means to synthesize the MEDL of the TTP controllers (and consequently the MHTTs) so that the process set is schedulable on an as cheap as possible architecture.

The MEDL is synthesized according to the optimization parameters available for each of the four approaches to message scheduling discussed in Section 5.2. In order to guide the optimization process, we need a cost function that captures the "degree of schedulability" for a certain MEDL implementation. Our cost function is a modified version of that in [Tin92]:

$$\textit{cost function} = \begin{cases} f_1 = \sum_{i=1}^{n} max(0, R_i - D_i)\text{, if } f_1 > 0 \\ f_2 = \sum_{i=1}^{n} R_i - D_i\text{, if } f_1 = 0 \end{cases}$$

where n is the number of processes in the application, R_i is the response time of a process P_i, and D_i is the deadline of a process P_i. If the process set is not schedulable, there exists at least one R_i that is greater than the deadline D_i, therefore the term f_1 of the function will be positive. In this case the cost function is equal to f_1. However, if the process set is schedulable, then all R_i are smaller than the corresponding deadlines D_i. In this case $f_1 = 0$ and we use f_2 as the cost function, as it is able to differentiate between two alternatives, both leading to a schedulable process set. For a given set of optimization parameters leading to a schedulable process set, a smaller f_2 means that we have improved the response times of the processes, so the application can be potentially implemented on a cheaper hardware architecture (with slower processors and/or bus). The release time R_i is calculated according to the arbitrary deadline analysis [Tin94a] based on the release jitter of the process (see section 4), its worst-case execution time, the blocking time, and the interference time due to higher priority processes.

For a given application, we are interested to synthesize a MEDL such that the cost function is minimized. We are also interested to evaluate in different contexts the four approaches to message scheduling, thus offering the designer a decision support for choosing the approach that best fits his application.

The MEDL synthesis problem belongs to the class of combinatorial problems, therefore we are interested to develop heuristics that are able to find accurate results in a reasonable time. We have developed optimization algorithms corresponding to each of the four approaches to message scheduling. A first set of algorithms is based on simple and fast greedy heuristics. A second class of heuristics aims at finding near-optimal solutions using the simulated annealing (SA) algorithm.

The greedy heuristic differs for each of the four approaches to message scheduling. The main idea is to improve the "degree of schedulability" of the process set by incrementally trying to reduce the release jitter of the processes.

The only way to reduce the release jitter in the SM and MM approaches is through the optimization of the $T_{m_{max}}$ parameters. This is achieved by a proper placement of messages into slots (see Figure 5.2).

The OptimizeSM algorithm presented in Figure 5.11, starts by deciding on a size *($size_{Si}$)* for each of the slots. There is nothing to be gained by enlarging the slot size, since in this approach a slot can carry at most one message. Thus, the slot sizes are set to the minimum size that can accommodate the largest message sent by the corresponding node.

Then, the algorithm has to decide on the number of rounds, thus determining the size of the MEDL. Since the size of the MEDL is physically limited, there is a limit to the number of rounds (e.g., 2, 4, 8, 16 depending on the particular TTP controller implementation). However, there is a minimum number of rounds *MinRounds* that is necessary for a certain application,

```
OptimizeSM
    -- set the slot sizes
    for each node $N_i$ do
        $size_{Si}$ = max(size of messages $m_j$ sent by node $N_i$)
    end for
    -- find the min. no. of rounds that can hold all the messages
    for each node $N_i$ do
        $nm_i$ = number of messages sent from $N_i$
    end for
    $MinRounds$ = max ($nm_i$)
    -- create a minimal complete MEDL
    for each message $m_i$
        find round in [1..MinRounds] that has an empty slot for $m_i$
        place $m_i$ into its slot in round
    end for
    for each RoundsNo in [MinRounds...MaxRounds] do
        -- insert messages in such a way that the cost is minimized
        repeat
            for each process $P_i$ that receives a message $m_i$ do
                if $D_i$ - $R_i$ is the smallest so far then $m$ = $m_{Pi}$ end if
            end for
            for each round in [1..RoundsNo] do
                place m into its corresponding slot in round
                calculate the CostFunction
                if the CostFunction is smallest so far then
                    BestRound = round
                end if
                remove m from its slot in round
            end for
            place m into its slot in BestRound if one was identified
        until the CostFunction is not improved
    end for
end OptimizeSM
```

Figure 5.11: Greedy Heuristic for SM

which depends on the number of messages transmitted. For example, if the processes mapped on node N_0 send in total 7 messages, then we have to decide on at least 7 rounds in order to accommodate all of them (in the SM approach there is at most one message per slot). Several numbers of rounds, *RoundsNo*, are tried out by the algorithm starting from *MinRounds* up to *MaxRounds*.

For a given number of rounds (that determine the size of the MEDL) the initially empty MEDL has to be populated with messages in such a way that the cost function is minimized. In order to apply the schedulability analysis that is the basis for the cost function, a *complete* MEDL has to be provided. A complete MEDL contains at least one instance of every message that has to be transmitted between the processes on different processors. A *minimal complete* MEDL is constructed from an empty MEDL by placing one instance of every message m_i into its corresponding empty slot of a *round*. In Figure 5.2a, for example, we have a MEDL composed of four rounds. We get a minimal complete MEDL, for example, by assigning m_2 and m_1 to the slots in rounds 3 and 4, and letting the slots in rounds 1 and 2 empty. However, such a MEDL might not lead to a schedulable system. The "degree of schedulability" can be improved by inserting instances of messages into the available places in the MEDL, thus minimizing the $T_{m_{max}}$ parameters. For example, in Figure 5.2a by inserting another instance of the message m_1 in the first round and m_2 in the second round leads to P_2 missing its deadline, while in Figure 5.2b inserting m_1 into the second round and m_2 into the first round leads to a schedulable system.

Our algorithm repeatedly adds a new instance of a message to the current MEDL in the hope that the cost function will be improved. In order to decide an instance of which message should be added to the current MEDL, a simple heuristic is used. We identify the process P_i which has the most "critical" situation, meaning that the difference between its deadline and response time, D_i - R_i, is minimal compared with all other processes. The message to be added to the MEDL is the message m=m_{Pi} received by the process P_i. Message m will be placed into that round *(BestRound)* which corresponds to the smallest value of the cost function. The algorithm stops if the cost function can not be further improved by adding more messages to the MEDL.

The OptimizeMM algorithm is similar to OptimizeSM. The main difference is that in the MM approach several messages can be placed into a slot (which also decides its size), while in the SM approach there can be at most one message per slot. Also, in the case of MM, we have to take additional care that the slots do not exceed the maximum allowed size for a slot.

The situation is simpler for the dynamic approaches, namely DM and DP, since we only have to decide on the slot sizes and, in the case of DP, on the packet size. For these two approaches, the placement of messages is dynamic and has no influence on the cost function. The OptimizeDM algorithm (see Figure 5.12) starts with the first slot $S_i = S_0$ of the TDMA round and tries to find that size *(BestSize*$_{Si}$*)* which corresponds to the smallest *CostFunction*. This slot size has to be large enough ($S_i \geq$

```
OptimizeDM
    for each node $N_i$ do
        $MinSize_{Si}$ = max(size of messages $m_j$ sent by node $N_i$)
    end for
    -- identifies the size that minimizes the cost function
    for each slot $S_i$
        $BestSize_{Si}$ = $MinSize_{Si}$
        for each SlotSize in [$MinSize_{Si}$...MaxSize] do
            calculate the CostFunction
            if the CostFunction is best so far then
                $BestSize_{Si}$ = $SlotSize_{Si}$
            end if
        end for
        $size_{Si}$ = $BestSize_{Si}$
    end for
end OptimizeDM
```

Figure 5.12: Greedy Heuristic for DM

$MinSize_{Si}$) to hold the largest message to be transmitted in this slot, and within bounds determined by the particular TTP controller implementation (e.g., from 2 bits up to $MaxSize$ = 32 bytes). Once the size of the first slot has been determined, the algorithm continues in the same manner with the next slots.

The OptimizeDP algorithm has, in addition, to determine the proper packet size. This is done by trying all the possible packet sizes given the particular TTP controller. For example, it can start from 2 bits and increment with the "smallest data unit" (typically 2 bits) up to 32 bytes. In the case of the OptimizeDP algorithm the slot size is a multiple of the packet size, and it is within certain bounds depending on the TTP controller.

We have also developed an SA based algorithm for bus access optimization corresponding to each of the four message scheduling approaches. In order to tune the parameters of the algorithm we have first performed very long and expensive runs on selected large examples, and the best ever solution, for each example, has been considered as the near-optimum. Based on further experiments we have determined the parameters of the SA algorithm, for different sizes of examples, so that the optimization time is reduced as much as possible but the near-optimal result is still produced. These parameters have then been used in the large scale experiments presented in the following section.

5.5 Experimental Results

We first present the experimental results for the schedulability analysis with the TTP (Section 5.2), comparing the four message scheduling approaches. Then, we present the results obtained for the communication synthesis problem outlined in Section 5.4. Finally, the results for the schedulability analysis of systems with data and control dependencies (Section 5.3) are presented.

Schedulability Analysis with TTP and Communication Synthesis. For the evaluation of our message scheduling approaches over TTP we used sets of processes generated for experimental purpose. We considered architectures consisting of 2, 4, 6, 8 and 10 nodes. 40 processes were assigned to each node, resulting in sets of 80, 160, 240, 320 and 400 processes. 30 sets were generated for each dimension, thus a total of 150 sets of processes were used for experimental evaluation. Worst case computation times, periods, deadlines, and message lengths were assigned randomly within certain intervals. For the communication channel we considered a transmission speed of 256 kbps. The maximum length of the data field in a slot was 32 bytes, and the frequency of the TTP controller was chosen to be 20 MHz. All experiments were run on a Sun Ultra 10 workstation.

For each of the 150 generated examples and each of the four scheduling approaches we have obtained, using our optimization strategy, the near-optimal values for the cost function (Section 5.4), produced by our SA based algorithm. These values, for a given example, might differ from one approach to another, as they depend on the optimization parameters and the schedulability analysis determined for each of the approaches. We were interested to compare the four approaches to message scheduling, based on the values obtained for the cost function.

Thus, Figure 5.13 presents the average percentage deviations of the cost function obtained in each of the four approaches, from the minimal value among them. The DP approach is generally the most performant, and the reason for this is that dynamic scheduling of messages is able to reduce release jitter because no space is waisted in the slots if the packet size is properly selected. However, by using the MM approach we can obtain almost the same result if the messages are carefully allocated to slots by our optimization strategy. Moreover, in the case of bigger sets of processes (e.g., 400) MM outperforms DP, as DP suffers form large overhead due to the handling of the packets. DM per-

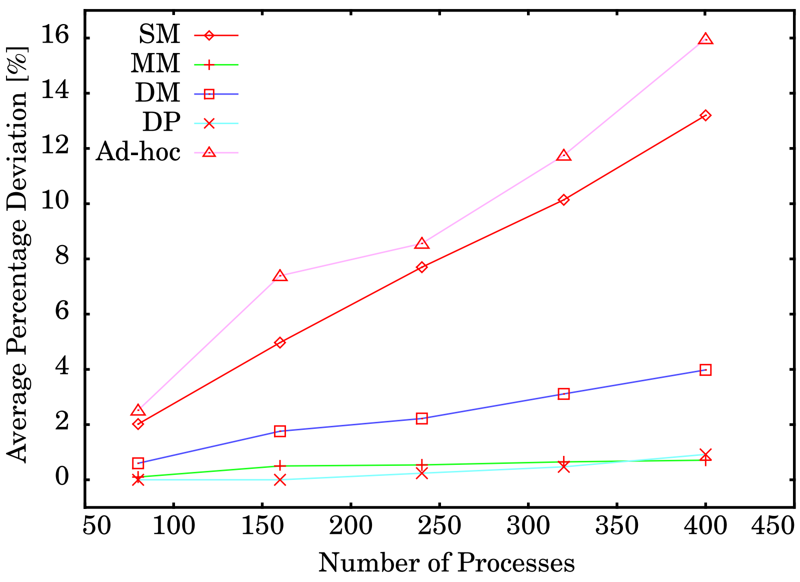

Figure 5.13: Comparison of the Four Approaches to Message Scheduling

forms worse than DP because it does not split the messages into packets, and this results in a mismatch between the size of the messages dynamically queued and the slot size, leading to unused slot space that increases the jitter. SM performs the worst as it does not permit much room for improvement, leading to large amounts of unused slot space. Also, DP has produced a MEDL that resulted in schedulable process sets for 1.33 times more cases than the MM and DM. MM, in its turn, produced two times more schedulable results than the SM approach.

Together with the four approaches to message scheduling, a so called ad-hoc approach is presented. The ad-hoc approach performs scheduling of messages without trying to optimize the access to the communication channel. The ad-hoc solutions are based on the MM approach and consider a design with the TDMA configuration consisting of a simple, straightforward, allocation of messages to slots. The lengths of the slots were

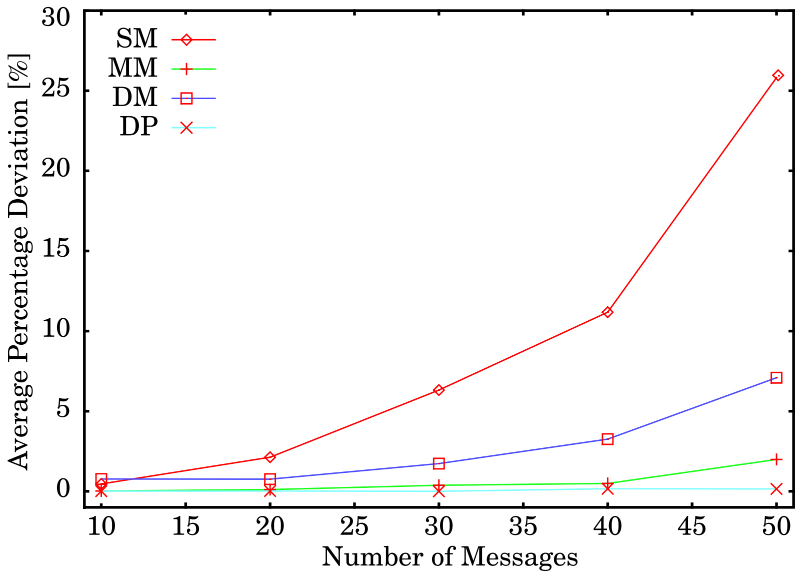

Figure 5.14: Four Approaches to Message Scheduling -- The Influence of the Messages Number

selected to accommodate the largest message sent from the respective node. Figure 5.13 shows that the ad-hoc alternative is constantly outperformed by any of the optimized solutions. This shows that by optimizing the access to the communication channel, significant improvements can be produced.

Next, we have compared the four approaches with respect to the number of messages exchanged between different nodes and the maximum message size allowed. For the results depicted in Figure 5.14 and 5.15 we have assumed sets of 80 processes allocated to 4 nodes. Figure 5.14 shows that as the number of messages increases, the difference between the approaches grows while the ranking among them remains the same. The same holds for the case when we increase the maximum allowed message size (Figure 5.15), with a notable exception: for large message sizes MM becomes better than DP, since DP suffers from the overhead due to packet handling.

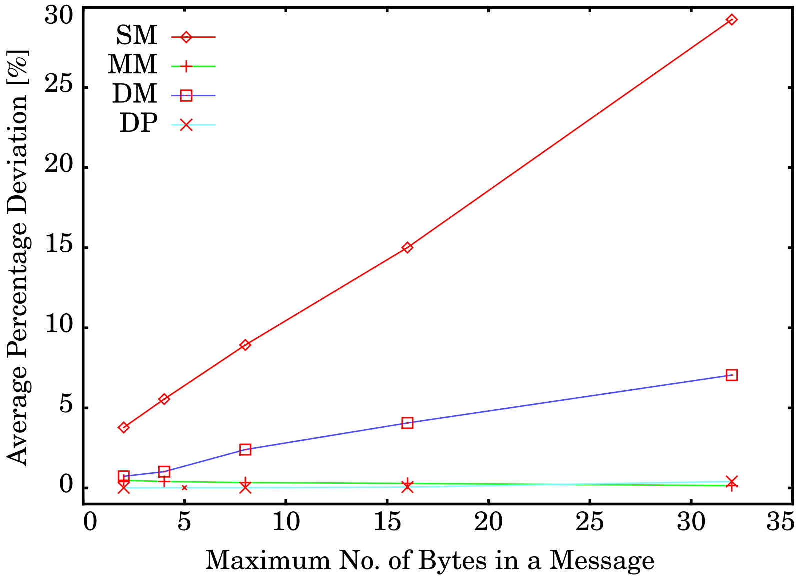

Figure 5.15: Four Approaches to Message Scheduling -- The Influnece of the Message Sizes

The above comparison between the four message scheduling alternatives is mainly based on the issue of schedulability. However, when choosing among the different policies, several other parameters can be of importance. Thus, a static allocation of messages can be beneficial from the point of view of testing and debugging and has the advantage of simplicity. Similar considerations can lead to the decision not to split messages. In any case, however, optimization of the bus access scheme is highly desirable.

In addition, we have considered a real-life example implementing an aircraft control system adapted from [Tin94b] where the ad-hoc solution and the SM approach failed to produce a schedulable solution. However, with the other two approaches schedulable solutions were produced, DP generating the smallest cost function followed in this order by MM and DM.

Table 5.1: Optimization Heuristics

Optimize		80 procs.	160 procs.	240 procs.	320 procs.	400 procs.
SM	avg.	0.12%	0.19%	0.50%	1.06%	1.63%
	max.	0.81%	2.28%	8.31%	31.05%	18.00%
MM	avg	0.05%	0.04%	0.08%	0.23%	0.36%
	max.	0.23%	0.55%	1.03%	8.15%	6.63%
DM	avg	0.02%	0.03%	0.05%	0.06%	0.07%
	max.	0.05%	0.22%	0.81%	1.67%	1.01%
DP	avg	0.01%	0.01%	0.05%	0.04%	0.03%
	max.	0.05%	0.13%	0.61%	1.42%	0.54%

We were also interested in the quality of our optimization heuristics for the communication synthesis problem. Thus, we have run all the examples presented above, using the four greedy heuristics (OptimizeSM, OptimizeMM, OptimizeDM, OptimizeDP) and compared the results with those produced by the SA based algorithm. The table below presents the average and maximum percentage deviations of the cost function for each of the graph dimensions.

All the four greedy heuristics perform very well, with less than 2% loss in quality compared to the results produced by the SA algorithms. The execution times for the greedy heuristics were more than two orders of magnitude smaller than those with SA.

Schedulability Analysis for Systems with Control and Data Dependencies. In this context, the two main aspects we were interested in are the quality of the schedulability analysis and the scalability of the algorithms for large examples. A set of massive experiments were performed on conditional process graphs generated for experimental purpose.

We considered architectures consisting of 2, 4, 6, 8 and 10 processors. 40 processes were assigned to each node, resulting in graphs of 80, 160, 240, 320 and 400 processes, having 2, 4, 6, 8 and 10 conditions, respectively. The number of unconditional subgraphs varied for each graph dimension depending on the number of conditions and the randomly generated structure of the CPGs. For example, for CPGs with 400 processes, the maximum number of unconditional subgraphs was 64.

30 graphs were generated for each graph dimension, thus a total of 150 graphs were used for experimental evaluation. Worst case execution times were assigned randomly using both uniform and exponential distribution. These experiments were also run on a Sun Ultra 10 workstation.

In order to compare the quality of the schedulability approaches, we need a cost function that captures, for a certain system, the difference in quality between the schedulability approaches proposed (Section 5.3). Our cost function is the difference between the deadline and the estimated worst case delay of a CPG, summed for all the CPGs in the system:

$$\textit{cost function} = \sum_{i=1}^{n} (D_{G_i} - \delta_{G_i})$$

where n is the number of CPGs in the system, $\delta_{\Gamma i}$ is the estimated worst case delay of the CPG Γ_i, and $D_{\Gamma i}$ is the deadline on Γ_i. A higher value for this cost function, for a given system, means that the corresponding approach produces better results (schedulability analysis is less pessimistic).

For each of the 150 generated example systems and each of the five approaches to schedulability analysis we have calculated the cost function mentioned previously, based on results produced with the algorithms described in Section 5.3. These values, for a given system, differ from one analysis to another, with the BF being the least pessimistic approach and therefore having the largest value for the cost function.

We are interested to compare the five approaches, based on the values obtained for the cost function. Thus, Figure 5.16

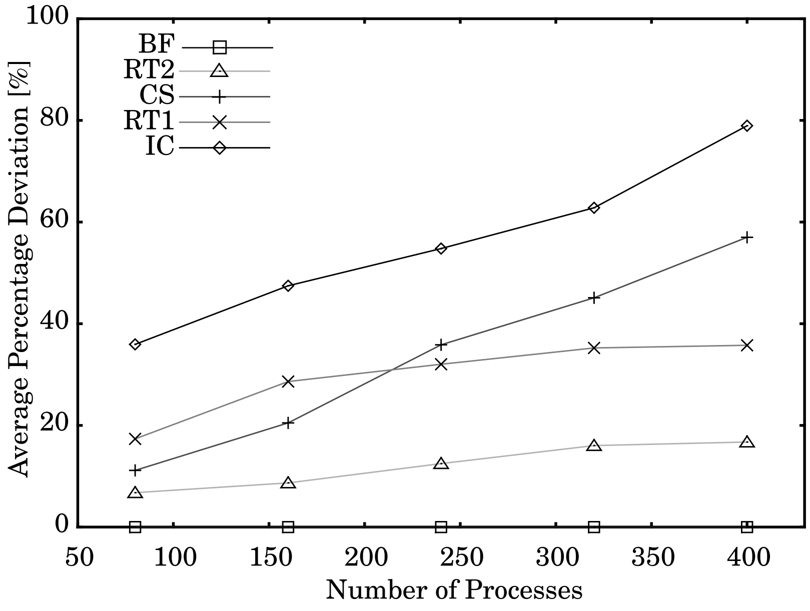

Figure 5.16: Average Percentage Deviations for Each of the Five Analyses

presents the average percentage deviations of the cost function obtained in each of the five approaches, compared to the value of the cost function obtained with the BF approach. A smaller value for the percentage deviation means a larger cost function, thus a better result. The percentage deviation is calculated according to the formula:

$$deviation = \frac{cost_{BF} - cost_{approach}}{cost_{BF}} 100 .$$

Figure 5.17 presents the average runtime of the algorithms, in seconds.

The brute force approach, BF, performs best in terms of quality and obtains the largest values for the cost function at the expense of a large execution time. The execution time can be up to 7 minutes for large graphs of 400 processes, 10 conditions, and 64 unconditional subgraphs. At the other end, the straightfor-

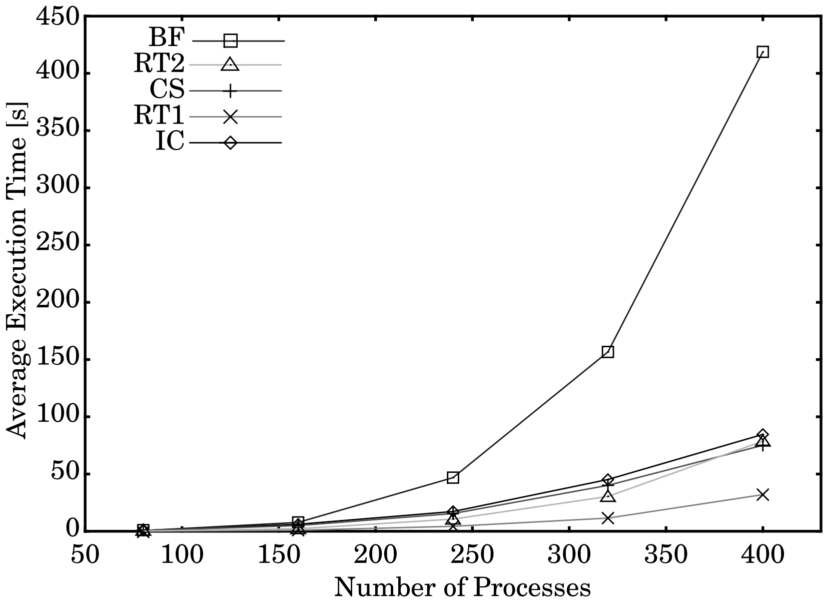

Figure 5.17: Average Execution Times for Each of the Five Analyses

ward approach IC that ignores the conditions, performs worst and becomes more and more pessimistic as the system size increases. As can be seen from Figure 5.16, IC has even for smaller systems of 160 processes (3 conditions, maximum 8 unconditional subgraphs) a 50% worse quality than the brute force approach, with almost 80% loss in quality, in average, for large systems of 400 processes. It is interesting to mention that the low quality IC approach has also an average execution time which is equal or comparable to the much better quality heuristics (except the BF, of course). This is because it tries to improve on the worst case delays through the iterative loop presented in DelayEstimate, Figure 5.5.

Let us turn our attention to the three approaches CS, RT1, and RT2 that, like the BF, consider conditions during the analysis but also try to perform a trade-off between quality and execu-

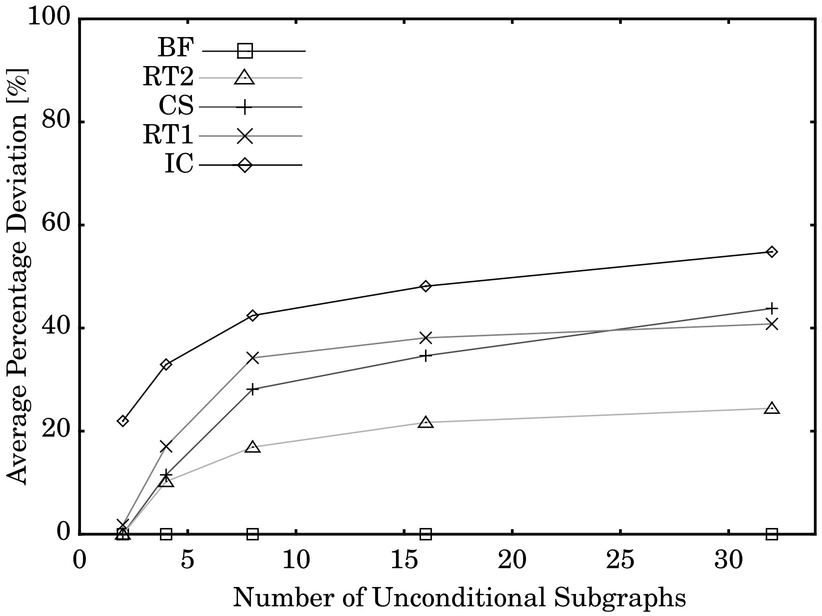

Figure 5.18: Comparison of the Schedulability Analysis Approaches Based on the No. of Unconditional Subgraphs

tion time. Figure 5.16 shows that the pessimism of the analysis is dramatically reduced by considering the conditions during the analysis. The RT1 and RT2 approaches, that visit each unconditional subgraph, perform in average better than the CS approach that considers condition separation for the whole graph. However, CS is comparable in quality with RT1, and even performs better for graphs of size smaller than 240 processes (4 conditions, maximum 16 subgraphs).

The RT2 analysis that tries to improve the worst case response times using the MaxSeparations, as opposed to RT1, performs best among the non-brute-force approaches. As can be seen from Figure 5.16, RT2 has less than 20% average deviation from the solutions obtained with the brute force approach. However, if faster runtimes are needed, RT1 can be used instead, as it is twice faster in execution time than RT2.

We were also interested to compare the five approaches with respect to the number of unconditional subgraphs and the number of conditional process graphs that form a system. For the results depicted in Figure 5.18 we have assumed CPGs consisting of 2, 4, 8, 16, and 32 unconditional subgraphs of maximum 50 processes each, allocated to 8 processors. Figure 5.18 shows that as the number of subgraphs increases, the differences between the approaches grow while the ranking among them remains the same, as resulted from Figure 5.16. The CS approach performs better than RT1 with a smaller number of subgraphs, but RT1 becomes better as the number of subgraphs in the CPGs increases.

Figure 5.19 presents on a logarithmic scale the average percentage deviations for systems consisting of 1, 2, 3, 4 and 5 conditional process graphs of 160 nodes each. As the number of conditional process graphs increases, the IC and CS approaches

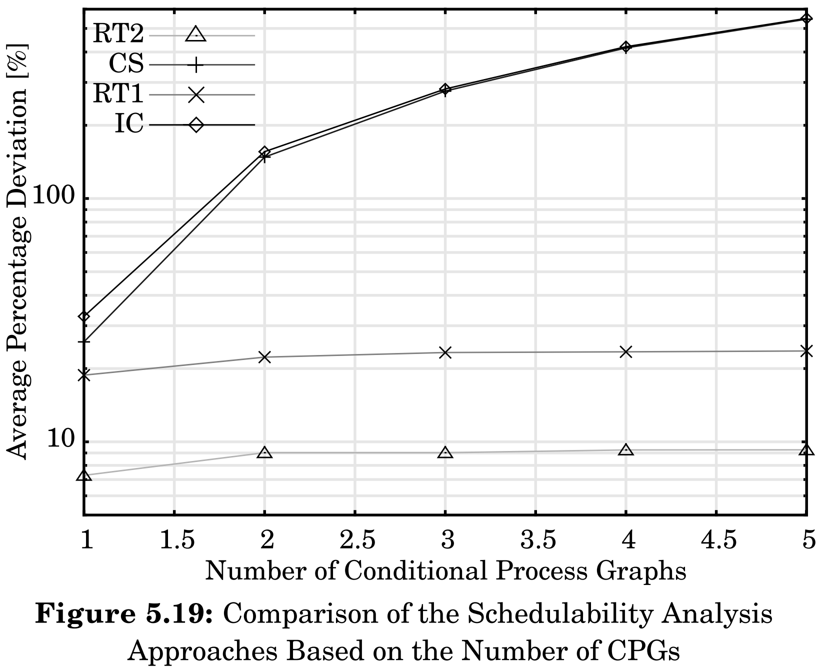

Figure 5.19: Comparison of the Schedulability Analysis Approaches Based on the Number of CPGs

become more pessimistic. However, RT1 and RT2 perform very well, with RT2 being the least pessimistic approach (except the BF approach, not depicted in Figure 5.19).

Finally, a real-life example implementing a vehicle cruise controller is presented in the next chapter.

Chapter 6 Application

THE RESEARCH PRESENTED in this thesis deals with aspects of scheduling and communication synthesis for distributed hard real-time systems. Such systems are today used in many application areas.

In this chapter we discuss the relevance of our results to the area of *automotive electronics*. The automotive electronics area deals with the electronically controlled functions onboard vehicles. Such electronic functionality is typically implemented in modern vehicles using distributed architectures consisting of several processors interconnected by a communication network. In this context, aspects of real-time communication are of extreme importance.

Electronic functionality in a vehicle might include: control of the displays in the dashboard, light control, mirror adjustment, seat position adjustment, climate control, window control, sunroof control, keyless vehicle entry systems, theft avoidance, control of the audio systems like radio or telephone, engine control, power train control, braking suspension, vehicle dynamics control, etc.

Most of these functions are not safety critical. However, as the electronic functionality replaces mechanical and hydraulic functions like braking and steering, the dependability aspect becomes of extreme importance. In this context, a communication protocol like TTP provides services that support fault-tolerance and predictability.

6.1 Cruise Controller

A typical safety critical application with hard real-time constraints, to be implemented on a TTP based architecture, is a vehicle cruise controller (CC). We have considered a CC system derived from a requirement specification provided by the industry.

The CC described in this specification delivers the following functionality: it maintains a constant speed for speeds over 35 km/h and under 200 km/h, offers an interface (buttons) to increase or decrease the reference speed, and is able to resume its operation at the previous reference speed. The CC operation is suspended when the driver presses the brake pedal.

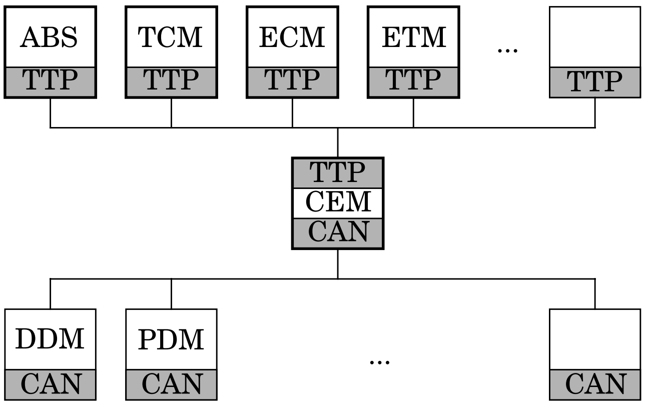

Figure 6.1: Hardware Architecture for the CC

The specification assumes that the CC will operate in an environment consisting of several nodes interconnected by a TTP channel (Figure 6.1). There are five nodes which functionally interact with the CC system: the Anti Blocking System (ABS), the Transmission Control Module (TCM), the Engine Control Module (ECM), the Electronic Throttle Module (ETM), and the Central Electronic Module (CEM). It has been decided to distribute the functionality (processes) of the CC over these five nodes. The transmission speed of the communication channel is 256 kbps and the frequency of the TTP controller was chosen to be 20 MHz.

We have modelled the specification of the CC system using a conditional process graph that consists of 32 processes, and includes two alternative tracks. The model is presented in Figure 6.2 where the worst case execution times are depicted to the right of each process and the message sizes to the left of each message.

6.2 Experimental Results

We have applied the scheduling algorithms and optimization heuristics presented in this thesis using as input the CPG corresponding to the CC system.

Let us first discuss the results obtained for the strategies presented in Chapter 4 that deals with time-driven systems. In Chapter 4 we were interested to compare the quality of the schedules produced by the list scheduling based algorithm using PCP and MPCP priority functions, and to check the potential of the algorithms presented in Section 4.2.3 to improve the generated schedules by optimizing the bus access scheme. For the implementation of the cruise controller as a time-driven system we have considered a deadline of 400 ms. Thus, for the CC example, the straightforward solution for bus access resulted in a schedule corresponding to a maximal delay of 429 ms (which does not meet the deadline) when PCP was used as a priority function, while using MPCP we obtained a schedule length of

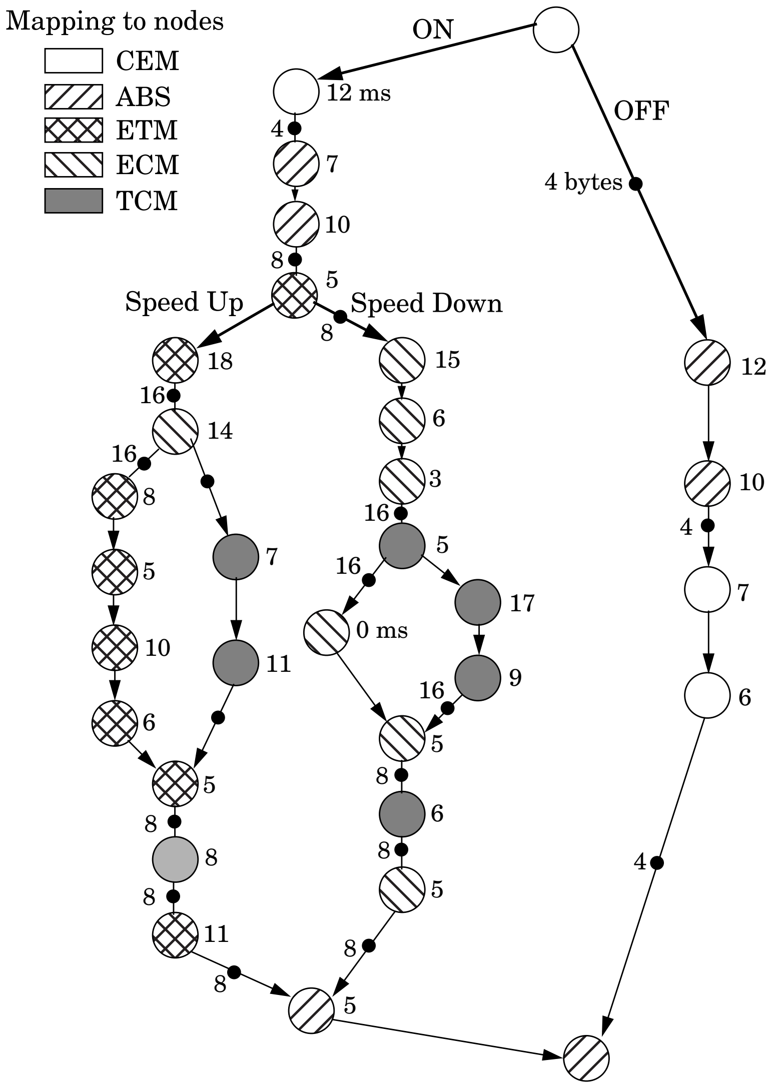

Figure 6.2: Cruise Controller Model

398 ms. The first and second greedy heuristics for bus access optimization produced solutions so that the worst case delay was reduced to 314 and 323 ms, respectively. The near-optimal solution (produced with the SA based approach) results in a delay of 302 ms.

Section 5.3 has presented several schedulability analysis techniques for systems with both control and data dependencies considering priority-based preemptive scheduling. For simplicity, the scheduling of messages has been treated separately, in Section 5.2. We have applied the analyses in Section 5.3 to the CPG, modelling the CC system without considering the messages (depicted with solid circles in Figure 6.2). The deadline has been set to 130 ms. In this context, we have obtained the following results. Without considering the conditions, IC obtained a worst case delay of 138 ms, thus the system resulted as being unschedulable. The system has also resulted as unschedulable with the Conditions Separation (CS) approach that has produced a result of 132 ms. However, the Brute Force approach (BF) produced a worst case delay of 124 ms which proves that the system implementing the vehicle cruise controller is, in fact, schedulable. Both Relaxed Tightness approaches (RT1 and RT2) produced the same worst case delay of 124 ms as the BF.

Chapter 7 Conclusions and Future Work

IN THIS THESIS we have presented aspects of scheduling and communication for distributed hard real-time systems. Special emphasis has been placed on the impact of the communication infrastructure and protocol on the overall system performance. The scheduling and communication strategies proposed are based on an abstract graph representation which captures, at process level, both dataflow and the flow of control.

The intention of this chapter is to summarize the work presented in the thesis and point out ideas for future work.

7.1 Conclusions

In this thesis we have considered hard-real time systems implemented on distributed architectures consisting of several nodes interconnected by a communication channel.

The systems are modelled using conditional process graphs that are able to capture data as well as control dependencies between processes.

We have considered both non-preemptive static cyclic scheduling and preemptive scheduling with static priorities for the scheduling of processes, while the communications are statically scheduled according to the time triggered protocol.

Time-Driven Systems. We have first proposed an extension to a static scheduling algorithm for CPGs. Thus, we have shown that the general scheduling algorithm for conditional process graphs can be successfully applied if the strategy for message planning is adapted to the requirements of the TDMA protocol. At the same time, the quality of generated schedules has been much improved by adjusting the priority function used by the scheduling algorithm to the particular communication protocol.

However, not only have particularities of the underlying architecture to be considered during scheduling, but the parameters of the communication protocol should also be adapted to fit the particular embedded application. We have shown that important performance gains can be obtained, without any additional cost, by optimizing the bus access scheme. The optimization algorithm, which now implies both process scheduling and optimization of the parameters related to the communication protocol, generates an efficient bus access scheme as well as the schedule tables for activation of processes and communications.

Event-Driven Systems. In the context of fixed priority preemptive scheduling we have proposed a schedulability analysis for the time-triggered protocol. We have considered four different approaches to message scheduling over TTP, that were compared based on the issue of schedulability. After this, we presented optimization strategies for the bus access scheme in order to fit the communication particularities of a certain application. We showed that by optimizing the bus access scheme, significant improvements in the "degree of schedulability" of a system can be produced. Our optimization heuristics are able to efficiently produce good quality results.

The schedulability analyses then have been extended in order to bound the response time of a hard real-time system with both control and data dependencies. Five approaches to the schedulability analysis of such systems are proposed, and we show that by considering the conditions during the analysis, the pessimism of the analysis can be drastically reduced.

7.2 Future Work

Once good performance analysis techniques are developed, they can be used to guide the partitioning and architecture selection tasks.

Mapping of processes. In [Pop98] we have addressed the problem of system level partitioning. Given an architecture consisting of several processors, ASICs and shared buses, the partitioning algorithm in [Pop98] finds the partitioning with the smallest hardware cost and is able to predict and guarantee the performance of the system in terms of worst case delay. Our intention is to extend the work in [Pop98] to consider a more realistic setting. Important issues that have to be considered are: reuse of functionality, physical constraints related to the placement of sensors and actuators, memory constraints, and a realistic communication infrastructure.

Architecture Selection. During the phase of mapping processes to architecture components, the mapping strategy could find out that there are not enough resources in order to guarantee the constraints imposed on the application. Such situations might include: lack of enough memory, lack of enough computing power to guarantee a certain imposed performance, etc. In such situations, several decisions have to be made related to architecture selection: how much memory to add and where, which processor should be replaced with a more powerful one, or if a new processor should be added to the architecture, etc. Nonetheless, before architecture selection can be performed, we also have to address the problem of architecture modelling.

References

[Aud91] N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings, "Hard Real-Time Scheduling: The Deadline Monotonic Approach", *Proceedings of 8th IEEE Workshop on Real-Time Operating Systems and* Soft-*ware*, 127-132, 1991.

[Aud93] N. C. Audsley, K. Tindell, A. Burns, "The End Of The Line For Static Cyclic Scheduling?", Proceedings of the 5th Euromicro Workshop on Real-Time Systems, 36 -41, 1993.

[Aud95] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, A. J. Wellings, "Fixed Priority Pre-emptive Scheduling: An Historical Perspective", *Real-Time Systems*, 8, 173-198, 1995.

[Axe96] J. Axelsson, "Hardware/software partitioning aiming at fulfilment of real-time constraints", *Journal of Systems Architecture*, 42, 449-464, 1996.

[Bal98] F. Balarin, L. Lavagno, P. Murthy, A. Sangiovanni-Vincentelli, "Scheduling for Embedded Real-Time Systems", *IEEE Design and Test of Computers*, 71-82, January-March, 1998.

[Bar98a] S. Baruah, “Feasibility Analysis of Recurring Branching Tasks”, *Proceedings of the 10th Euro*micro *Workshop on Real-Time Systems*, 138-145, 1998.

[Bar98b] S. Baruah, “A General Model for Recurring Real-Time Tasks”, *Proceedings of the IEEE Real-Time Symposium*, 114-122, 1998.

[Ben96] A. Bender, “Design of an Optimal Loosely Coupled Heterogeneous Multiprocessor System”, *Proc. ED&TC*, 275-281, 1996.

[Bol97] I. Bolsens, H. J. De Man, B. Lin, K. Van Rompaey, S. Vercauteren, D. Verkest, “Hardware/Software Co-Design of Digital Telecommunication Systems”, *Proceedings of the IEEE*, 85(3), 391-418, 1997.

[Cho92] P. Chou, R. Ortega, G. Borriello, “Synthesis of hardware/software interface in microcontroller-based systems”, *Proceedings of the International Conference on Computer Aided Design*, 1992.

[Cho95a] P. H. Chou, R. B. Ortega, G. Borriello, “The Chinook Hardware/Software Co-Synthesis System”, *Proc. Int. Symposium on System Synthesis*, 22-27, 1995.

[Cho95b] P. Chou, G. Borriello, “Interval Scheduling: Fine-Grained Code Scheduling for Embedded Systems”, *Proc. ACM/IEEE DAC*, 462-467, 1995.

[Cof72] E. G. Coffman Jr., R. L. Graham, “Optimal Scheduling for two Processor Systems”, *Acta Informatica*, 1, 200-213, 1972.[Dav95]

[Dav95] J. M. Daveau, T. Ben Ismail, A. A. Jerraya, “Synthesis of System-Level Communication by an Allocation-Based Approach”, *Proc. Int. Symposium on System Synthesis*, 150-155, 1995.

[Dav98] B. P. Dave, N. K. Jha, "COHRA: Hardware-Software Cosynthesis of Hierarchical Heterogeneous Distributed Systems", *IEEE Transactions on CAD*, 17(10), 900-919, 1998

[Dav99] B. P. Dave, G. Lakshminarayana, N. J. Jha, "COSYN: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded Systems", *IEEE Transactions on VLSI Systems*, 7(1), 92-104, 1999.

[Deo98] J. S. Deogun, R. M. Kieckhafer, A. W. Krings, "Stability and Performance of List Scheduling with External Process Delays", *Real Time Systems*, 15(1), 5-38, 1998.

[Dic98] R. P. Dick, N. K. Jha, "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems", *Proceedings of the International Conference on CAD*, 1998.

[Dob98] A. Doboli, P. Eles, "Scheduling under Control Dependencies for Heterogeneous Architectures", *International Conference on Computer Design (ICCD)*, 1998.

[Edw97] S. Edwards, L. Lavagno , E. A. Lee , A.Sangoivanni-Vincentelli , "Design of Embedded Systems: Formal Models, Validation and Synthesis", *Proceedings of the IEEE*, Vol. 85, No. 3, March 1997.

[Ele00] P. Eles, A. Doboli, P. Pop, Z. Peng, "Scheduling with Bus Access Optimization for Distributed Embedded Systems", *IEEE Transactions on VLSI Systems,* 2000 (to appear).

[Ele97] P. Eles, Z. Peng, K. Kuchcinski, A. Doboli, "System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search", *Design Automation for Embedded Systems*, 2(1), 5-32, 1997.

[Ele98] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, P. Pop, "Process Scheduling for Performance Estimation and Synthesis of Hardware/Software Systems", *Proceedings of the Euromicro Conference*, 168-175, 1998.

[Ele98a] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, P. Pop, "Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems", *Proceedings of Design Automation & Test in Europe (DATE)*, 1998.

[Ele98b] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, P. Pop, "Process Scheduling for Performance Estimation and Synthesis of Hardware/Software Systems", *Proceedings of 24th Euromicro Conference*, 1998.

[Eng99] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, H. Hansson, "Towards Industry Strength Worst-Case Execution Time Analysis", *Swedish National Real-Time Conference SNART'99*, 1999

[Erm97] H. Ermedahl, H. Hansson, M. Sjödin, "Response-Time Guarantees in ATM Networks", *Proc. IEEE Real-Time Systems Symposium*, 274-284, 1997.

[Ern93] R. Ernst, J. Henkel, T. Benner, "Hardware/software co-synthesis for microcontrollers", *IEEE Design & Test of Computers*, 64-75, September 1997.

[Ern97] R. Ernst, W. Ye, "Embedded Program Timing Analysis Based on Path Clustering and Architecture Classification", *Proc. Int. Conf. on CAD*, 598-604, 1997.

[Ern98] R. Ernst, "Codesign of Embedded Systems: Status and Trends", *IEEE Design and Test of Computers*, 45-54, April-June 1998.

[Ern99] L. Thiele, K. Strehl, D. Ziegengein, R. Ernst, J. Teich, "FunState-an internal design representation for codesign", *International Conference on Computer-Aided Design*, 558 -565, 1999.

[Foh93] G. Fohler, "Realizing Changes of Operational Modes with Pre Run-time Scheduled Hard Real-Time Systems", *Responsive Computer Systems,* H. Kopetz and Y. Kakuda, editors, 287-300, Springer Verlag, 1993.

[Gaj95] D. D. Gajski, F. Vahid, "Specification and Design of Embedded Hardware-Software Systems", *IEEE Design and Test of Computers*, 53-67, Spring 1995.

[Ger96] R. Gerber, D. Kang, S. Hong, M. Saksena, "End-to-End Design of Real-Time Systems", *Formal Methods in Real-Time Computing*, D. Mandrioli and C. Heitmeyer, editors, John Wiley & Sons, 1996.

[Gon95] J. Gong, D. D. Gajski, S. Narayan, "Software Estimation Using A Generic-Processor Model", *Proceedings of the European Design and Test Conf*, 498-502, 1995.

[Gup93] R. K. Gupta, G. De Micheli, "Hardware-software cosynthesis for digital systems", *IEEE Design & Test of Computers*, 29-41, September 1993.

[Gup95] R. K. Gupta, "Co-Synthesis of Hardware and Software for Digital Embedded Systems", *Kluwer Academic Publishers*, *Boston*, 1995.

[Hen95] J. Henkel, R. Ernst, "A Path-Based Technique for Estimating Hardware Run-time in Hardware/Software Cosynthesis", *Proceedings of the International Symposium on System Synthesis*, 116-121, 1995

[Jor97] P. B. Jorgensen, J. Madsen, "Critical Path Driven Cosynthesis for Heterogeneous Target Architectures", *Proceedings of the International Workshop on Hardware/Software Codesign*, 15-19, 1997.

[Kas84] H. Kasahara, S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing", *IEEE Transaction on Computers*, 33(11), 1023-1029, 1984.

[Knu99] P. V. Knudsen, J. Madsen, "Integrating Communication Protocol Selection with Hardware/Software Codesign", *IEEE Transactions on CAD*, 18(8), 1077-1095, 1999.

[Kop94] H. Kopetz, G. Grünsteidl, "TTP-A Protocol for Fault-Tolerant Real-Time Systems", *IEEE Computer*, 27(1), 14-23, 1994.

[Kop97a] H. Kopetz, "Real-Time Systems-Design Principles for Distributed Embedded Applications", *Kluwer Academic Publishers*, 1997.

[Kop97b] H. Kopetz et al, "A Prototype Implementation of a TTP/C, Controller", *SAE Congress and Exhibition*, 1997.

[Kuc97] K. Kuchcinski, "Embedded System Synthesis by Timing Constraint Solving", *Proceedings of the International Symposium on System Synthesis*, 50-57, 1997.

[Kwo96] Y. K. Kwok, I. Ahmad, "Dynamic Critical-Path Scheduling: an Effective Technique for Allocating Task Graphs to Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, 7(5), 506-521, 1996.

[Lak99] G. Lakshminarayana, K. S. Khouri, N. K. Jha, "Wawesched: A Novel Scheduling Technique for Control-Flow Intensive Designs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 18(5), 1999.

[Lee99] C. Lee, M. Potkonjak, W. Wolf, "Synthesis of Hard Real-Time Application Specific Systems", *Design Automation for Embedded Systems*, 4(4), 215-241, 1999.

[Li95] Y. S. Li, S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration", *Proc. ACM/IEEE DAC*, 456-461, 1995.

[Liu73] C. L. Liu, J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, V20, N1, 46-61, 1973.

[Lon99] H. Lonn, J. Axelsson, "A Comparison of Fixed-Priority and Static Cyclic Scheduling for Distributed Automotive Control Applications", *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, 142-149, 1999.

[Lun99] T. Lundqvist, P. Stenström, "An Integrated Path and Timing Analysis Method Based on Cycle-Level Symbolic Execution", *Real-Time Systems*, V17, N2/3, 183-207, 1999.

[Mal97] S. Malik, M. Martonosi, Y.S. Li, "Static Timing Analysis of Embedded Software", *Proc. ACM/IEEE DAC*, 147-152, 1997.

[Mc92] K. McMillan, D. Dill, "Algorithms for interface timing verification", *Proceedings of IEEE International Conference on Computer Design*, 48-51, 1992.

[Mic96] G. De Micheli, M.G. Sami eds., "Hardware/Software Co-Design", NATO ASI 1995, *Kluwer Academic Publishers*, 1996.

[Mic97] G. De Micheli, R.K. Gupta, "Hardware/Software Co-Design", *Proceedings of the IEEE*, 85(3), 349-365, 1997.

[Moo97] V. Mooney, T. Sakamoto, G. De Micheli, "Run-Time Scheduler Synthesis for Hardware-Software Systems and Application to Robot Control Design", *Proc. Int. Workshop on Hardware-Software Co-design*, 95-99, 1997.

[Nar94] S. Narayan, D. D. Gajski, “Synthesis of System-Level Bus Interfaces”, *Proceedings of the European Design and Test Conference*, 395-399, 1994.

[Ort98] R. B. Ortega, G. Borriello, “Communication Synthesis for Distributed Embedded Systems”, *Proceedings of the International Conference on CAD*, 437-444, 1998.

[Pal98] J. C. Palencia, M. González Harbour, “Schedulability Analysis for Tasks with Static and Dynamic Offsets”, *Proc. of the 19th IEEE Real-Time Systems Symposium*, 1998.

[Pop00a] P. Pop, P. Eles, Z. Peng, “Bus Access Optimization for Distributed Embedded Systems Based on Schedulability Analysis”, *Proceedings of the Design, Automation & Test In Europe Conference,* 567-574, 2000.

[Pop00b] P. Pop, P. Eles, Z. Peng, “Performance Estimation for Embedded Systems with Data and Control Dependencies”, *Proceedings of the 8th International Workshop on Hardware/Software Codesign,* 62-66, 2000.

[Pop00c] P. Pop, P. Eles, Z. Peng, “Schedulability Analysis for Systems with Data and Control Dependencies”, *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, 2000 (to appear).

[Pop98] P. Pop, P. Eles, Z. Peng, “Scheduling Driven Partitioning of Heterogeneous Embedded Systems”, *Swedish Workshop on Computer Systems Architecture,* 99-102, 1998.

[Pop99a] P. Pop, P. Eles, Z. Peng, “Scheduling with Optimized Communication for Time-Triggered Embedded Systems”, *7th International Workshop on Hardware/Software Codesign*, 178-182, 1999.

[Pop99b] P. Pop, P. Eles, Z. Peng, "Communication Scheduling for Time-Triggered Systems", *11th Euromicro Conference on Real-Time Systems (Work in Progress Proceedings),* 1999.

[Pop99c] P. Pop, P. Eles, Z. Peng, "An Improved Scheduling Technique for Time-Triggered Embedded Systems", *25th Euromicro Conference,* 303-310, 1999.

[Pop99d] P. Pop, P. Eles, Z. Peng, "Schedulability-Driven Communication Synthesis for Time Triggered Embedded Systems", *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications,* 287-294, 1999.

[Pra92] S. Prakash, A. Parker, "SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems", *Journal of Parallel and Distributed Computers*, 16, 338-351, 1992.

[Ree93] C. R. Reevs, *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific Publications, 1993.

[Sha90] L. Sha, R. Rajkumar, J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, 39(9), 1175-1185, 1990.

[Sta97] J. Staunstrup, W. Wolf eds., "Hardware/Software Co-Design: Principles and Practice", Kluwer Academic Publishers, 1997.

[Suz96] K. Suzuki, A. Sangiovanni-Vincentelli, "Efficient Software Performance Estimation Methods for Hardware/Software Codesign", *Proc. ACM/IEEE DAC*, 605-610, 1996.

[Sta93] J. A. Stankovic, K. Ramamritham, "Advances in Real-Time Systems", *IEEE Computer Society Press*, 1993.

[Tin92] K. Tindell, A. Burns, A. J. Wellings, "Allocating Real-Time Tasks (An NP-Hard Problem made Easy)," *Real-Time Systems*, 4(2), 145-165, 1992.

[Tin94a] K. Tindell, J. Clark, "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems", *Microprocessing and Microprogramming*, 40, 117-134, 1994.

[Tin94b] K. Tindell, "Adding Time-Offsets to Schedulability Analysis", *Department of Computer Science, Univer*sity *of York, Report Number YCS*-94-221, 1994.

[Tin95] K. Tindell, A. Burns, A. J. Wellings, "Calculating Controller Area Network (CAN) Message Response Times", *Control Engineering Practice*, 3(8), 1163-1169, 1995.

[Tur99] J. Turley, "Embedded Processors by the Numbers", *Embedded Systems Programming*, May 1999.

[Ull75] D. Ullman, "NP-Complete Scheduling Problems", *Journal of Computer Systems Science*, 10, 384-393, 1975.

[Vah94] F. Vahid, J. Gong, D. Gajski, "A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning", *Proceedings of European Design Automation Conference EURO-DAC/VHDL*, 214-219, 1994.

[Val95] C. A. Valderrama, A. Changuel, P. V. Raghavan, M. Abid, T. Ben Ismail, A. A. Jerraya, "A Unified Model for Co-simulation and Co-synthesis of Mixed Hardware/Software Systems", *Proceedings of the European Design and Test Conference*, 1995.

[Val96] C. A. Valderrama, F. Nacabal, P. Paulin, A. A. Jerraya, “Automatic Generation of Interfaces for Distributed C-VHDL Cosimulation of Embedded Systems: an Industrial Experience”, *Proceedings of the International Workshop on Rapid System Prototyping*, 72-77, 1996.

[Ver96] D. Verkest, K. Van Rompaey, I. Bolsens, H. De Man, “CoWare--A Design Environment for Heterogeneous Hardware/Software Systems”, *Design Automation for Embedded Systems,* 1, 357-386, 1996.

[Wal94] E. Walkup, G. Borriello, “Automatic Synthesis of Device Drivers for Hardware/Software Co-design”, *Technical Report 94-06-04, Dept. of Computer Science and Engineering, University of Washington,* 1994.

[Wir98] X-by-Wire Consortium, URL: http://www.vmars.tuwien.ac.at/projects/xbywire/, 1998

[Wol94] W. Wolf, “Hardware-Software Co-Design of Embedded Systems”, *Proceedings of the IEEE*, V82, N7, 967-989, 1994.

[Wu90] M. Y. Wu, D. D. Gajski, “Hypertool: A Programming Aid for Message-Passing Systems”, *IEEE Transactions on Parallel and Distributed Systems*, 1(3), 330-343, 1990.

[Xu00] J. Xu, D. L. Parnas, “Priority Scheduling Versus Pre-Run-Time Scheduling”, *Real Time Systems*, 18(1), 7-24, 2000.

[Xu93] J. Xu, D. L. Parnas, “On satisfying timing constraints in hard-real-time systems Software Engineering”, *IEEE Transactions* on Volume: 19(1), 70 -84, 1993.

[Yen97] T. Y. Yen, W. Wolf, “Hardware-Software Co-Synthesis of Distributed Embedded Systems”, *Kluwer Academic Publishers*, 1997.

[Yen98] T. Yen, W. Wolf, “Performance estimation for real-time distributed embedded systems”, *IEEE Transactions on Parallel and Distributed Systems*, Volume: 9(11), 1125 -1136, 1998