Scheduling with Bus Access Optimization for Distributed Embedded Systems

Petru Eles[1] (petel@ida.liu.se), Alex Doboli[2] (adoboli@ececs.uc.edu), Paul Pop[1] (paupo@ida.liu.se), and Zebo Peng[1] (zebpe@ida.liu.se)

[1] Dept. of Computer and Information Science
Linköping University
Sweden

[2] Dept. of Electr.&Comp. Eng. and Comp. Science
University of Cincinnati
USA

Abstract—**In this paper we concentrate on aspects related to the synthesis of distributed embedded systems consisting of programmable processors and application specific hardware components. The approach is based on an abstract graph representation which captures, at process level, both dataflow and the flow of control. Our goal is to derive a worst case delay by which the system completes execution, such that this delay is as small as possible, to generate a logically and temporally deterministic schedule, and optimize parameters of the communication protocol, such that this delay is guaranteed. We have further investigated the impact of particular communication infrastructures and protocols on the overall performance and, specially, 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 optimization algorithm, which 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.**

Index Terms—**System synthesis, distributed embedded systems, process scheduling, real-time systems, communication synthesis, time-triggered protocol.**

I. INTRODUCTION

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 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. 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 cosynthesis" is often used to denote this system-level synthesis process. Surveys on this topic can be found in [1, 2, 3, 4, 5, 6].

An important characteristic of an embedded system is its performance in terms of timing behavior. In this paper we concentrate on several aspects related to the 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. Thus, process scheduling has not only to be performed for the synthesis of the final system, but also as part of the performance estimation task.

Optimal scheduling, in even simpler contexts than that presented above, has been proven to be an NP complete problem [7]. Thus, it is essential to develop heuristics which produce good quality results in a reasonable time. In our approach, we assume that some processes can only be activated if certain conditions, computed by previously executed processes, are fulfilled [8, 9]. Thus, process scheduling is further 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. This is an important contribution of our approach because we capture both the flow of data and that of control at the process level, which allows a more accurate and direct modeling of a wide range of applications.

Performance estimation at the process level has been well studied in the last years. Papers like [10, 11, 12, 13, 14, 15, 16] provide a good background for derivation of execution time (or worst case execution time) for a single process. Starting from estimated execution times of single processes, performance estimation and scheduling of a system consisting of several processes can be performed. Preemptive scheduling of independent processes with static priorities running on single processor architectures has its roots in [17]. The approach has been later extended to accommodate more general computational models and has also been applied to distributed systems [18]. The reader is referred to [19, 20] for surveys on this topic. In [21] performance estimation is based on a preemptive scheduling strategy with static priorities using rate monotonic analysis. In [22] 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 [23, 24].

Several research groups have considered hardware/software architectures consisting of a single programmable processor and an ASIC acting as a hardware coprocessor. Under these circumstances deriving a static schedule for the software component is practically reduced to the linearization of a dataflow graph with nodes representing elementary operations or processes [25]. In the Vulcan system [26] software is implemented as a set of linear threads which are scheduled dynamically at execution. Linearization for thread generation can be performed both by exact, exponential complexity, algorithms as well as by faster urgency based heuristics. Given an application specified as a collection of tasks, the tool presented in [27] automatically generates a scheduler consisting of two parts: a static scheduler which is implemented in hardware and a dynamic scheduler for the software tasks running on a microprocessor.

Static cyclic scheduling of a set of data dependent software processes on a multiprocessor architecture has been intensively researched [28]. Several approaches are based on list scheduling heuristics using different priority criteria [29, 30, 31, 32] or on branch-and-bound algorithms [33, 34]. In [35, 36] 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 [37] a formulation using constraint logic programming has been proposed for similar problems.

It is important to mention that in all the approaches discussed above process interaction is only in terms of dataflow. This is the case also in [38] where a two level internal representation is introduced: control-dataflow graphs for operation level representation and pure dataflow graphs for representation at process level. The representation is used as a basis for derivation and validation of internal timing constraints for real-time embedded systems. In [39, 40] an internal design representation is presented which is able to capture mixed data/control flow specifications. It combines dataflow properties with finite state machine behavior. The scheduling algorithm discussed in [39] handles a subset of the proposed representation. Timing aspects are ignored and only software scheduling on a single processor system is considered.

In our approach, we consider embedded systems specified as interacting processes which have been mapped on an architecture consisting of several programmable processors and dedicated hardware components interconnected by shared buses. Process interaction in our model is not only in terms of dataflow but also captures the flow of control. Considering a non-preemptive execution environment we statically generate a schedule table and derive a guaranteed worst case delay.

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 [41, 42, 43, 44, 45].

We have to mention here some results obtained in extending real-time schedulability analysis so that network communication aspects can be handled. In [46], for example, the CAN protocol is investigated while the work reported in [47] considers systems based on the ATM protocol. These works, however, are restricted to software systems implemented with priority based preemptive scheduling.

In the first part of this paper we consider a communication model based on simple bus sharing. There we concentrate on the aspects of scheduling with data and control dependencies and such a simpler communication model allows us to focus on these issues. However, one of the goals of this paper is to highlight how communication and process scheduling strongly interact with each-other and how system-level optimization can only be performed by taking into consideration both aspects. Therefore, in the second part of the paper we introduce a particular 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. Moreover, our algorithm performs an optimization of parameters defining the communication protocol which is essential for reduction of the execution delay. Our system architecture is built on a communication model which is based on the time-triggered protocol (TTP) [48]. TTP is well suited for safety critical distributed real-time control systems and represents one of the emerging standards for several application areas, such as automotive electronics [28, 49].

The paper is divided into 8 sections. In section 2 we formulate our basic assumptions and set the specific goals of this work. Section 3 defines the formal graph based model which is used for system representation, introduces the schedule table, and creates the background needed for presentation of our scheduling technique. The scheduling algorithm for conditional process graphs is presented in section 4. In section 5 we introduce the hardware and software aspects of the TTP based system architecture. The mutual interaction between scheduling and the communication protocol as well as our strategy for scheduling with optimization of the bus access scheme are discussed in section 6. Section 7 describes the experimental evaluation and section 8 presents our conclusions.

II. PROBLEM FORMULATION

We consider a generic architecture consisting of *programmable processors* and application specific *hardware processors* (ASICs) connected through several *buses*. The buses can be shared by several communication channels connecting processes assigned to different processors. Only one process can be executed at a time by a programmable processor while a hardware processor can execute processes in parallel[1]. Processes on different processors can be executed in parallel. Only one data transfer can be performed by a bus at a given moment.

[1] In some designs certain processes implemented on the same hardware processor can share resources and, thus, cannot execute in parallel. This situation can easily be handled in our scheduling algorithm by considering such processes in a similar way with those allocated to programmable processors. For simplicity, in our presentation here we consider that processes allocated to ASICs do not share resources.

Data transfer on buses and computation can overlap.

Each process in the specification can be, potentially, assigned to several programmable or hardware processors which are able to execute that process. For each process estimated cost and execution time on each potential host processor are given [50]. We assume that the amount of data to be transferred during communication between two processes has been determined in advance. In [50] we presented algorithms for automatic hardware/software partitioning based on iterative improvement heuristics. The problem we are discussing in this paper concerns 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. Our goal is to derive a worst case delay by which the system completes execution, such that this delay is as small as possible, to generate the static schedule and optimize parameters of the communication protocol, such that this delay is guaranteed.

For the beginning we will consider an architecture based on a communication infrastructure in which communication tasks are scheduled on buses similar to the way processes are scheduled on programmable processors. The time needed for a given communication is estimated depending on the parameters of the bus to which the respective communication channel is assigned and the number of transferred bits. Communication time between processes assigned to the same processor is ignored. Based on this architectural model we introduce our approach to process scheduling *in the context of both control and data dependencies*.

In the second part of the paper we introduce an architectural model with a communication infrastructure suitable for safety critical hard real-time systems. This allows us to further investigate the scheduling problem and to explore the impact of the communication infrastructure on the overall system performance. The main goal is to determine the parameters of the communication protocol so that the overall system performance is optimized and, thus, the imposed time constraints can be satisfied. We show that system optimization and, in particular, scheduling cannot be efficiently performed without taking into consideration the underlying communication infrastructure.

III. PRELIMINARIES

A. The Conditional Process Graph

We consider that an application specified as a set of interacting processes is mapped to an abstract representation consisting of a directed, acyclic, polar graph $G(V, E_S, E_C)$ called *process graph*. Each node $P_i \in V$ represents one process. E_S and E_C are the sets of simple and conditional edges respectively. $E_S \cap E_C = \emptyset$ 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 task. 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 paper, 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 CPG depicted in Fig. 1, P_0 and P_{32} are the source and sink nodes respectively. For the rest of 31 nodes, 17, denoted $P_1, P_2, .., P_{17}$, are ordinary processes specified by the designer. They are assigned to one of the two programmable processors pe_1 and pe_2 or to the hardware component pe_3. The rest of 14 nodes are so called *communication processes* ($P_{18}, P_{19}, .., P_{31}$). They are represented in Fig. 1 as solid circles and are introduced during the mapping process for each connection which links processes assigned to different processors. These processes model inter-processor communication 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 Fig. 1 are performed on bus b_1. As discussed in the previous section, we treat, for the beginning, communication processes exactly as ordinary processes. Buses are similar to programmable processors in the sense that only one communication can take place on a bus at a given moment.

An edge $e_{ij} \in E_C$ is a *conditional edge* (represented with thick lines in Fig. 1) and it 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 Fig. 1 processes P_2, P_{11} and P_{12} have conditional edges at their output. Process P_2, for example, communicates alternatively with P_4 and P_5, or with P_6. Process P_{12}, if activated (which occurs only if condition D in P_{11} has value *true*), always communicates with P_{16} but alternatively with P_{14} or P_{15}, depending on the value of condition K.

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 Fig. 1 circles representing conjunction and

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

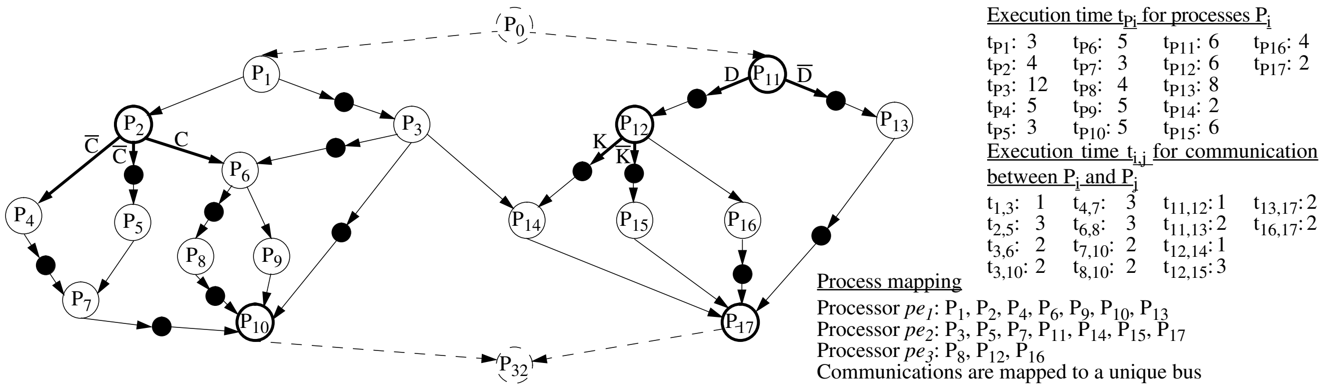


Fig. 1. Conditional Process Graph with execution times and mapping

disjunction nodes are depicted with thick borders. The alternative paths starting from disjunction node P_2, which computes condition C, meet in conjunction node P_{10}. Node P_{17} is the joining point for both the paths corresponding to condition K (starting from disjunction node P_{12}) and condition D (starting from disjunction node P_{11}). 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. In Fig. 1, process P_7 can be activated after it received messages sent by P_4 and P_5; process P_{10} waits for messages sent by P_3, P_8, and P_9 or by P_3 and P_7. 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 times of some processes as well as multiple deadlines can be easily modeled 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.

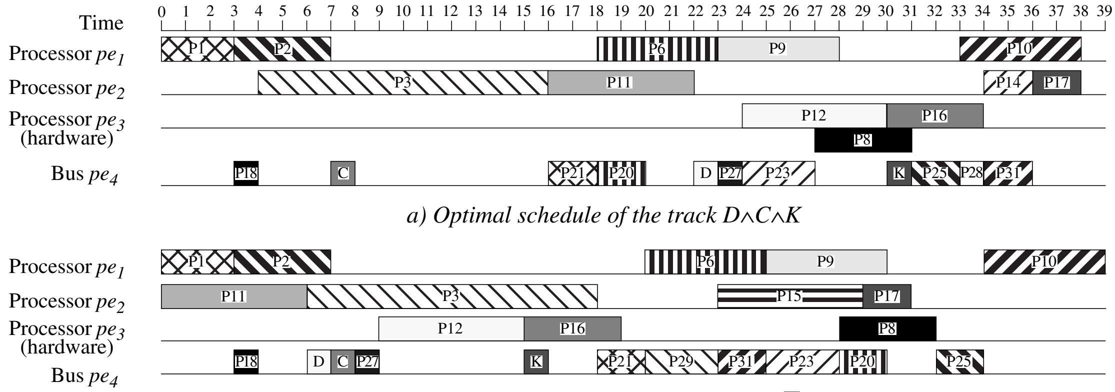
A boolean expression X_{Pi}, called a *guard*, can be associated to each node P_i in the graph. It represents the necessary conditions for the respective process to be activated. In Fig. 1, for example, $X_{P3}=true$, $X_{P14}=D \wedge K$, $X_{P17}=true$, $X_{P5}=\overline{C}$. X_{Pi} is not only necessary 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 case of hard real-time systems this will, typically, be worst case execution times and their estimation has been extensively discussed in the literature [13, 14]. 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.

B. The Schedule Table

For a given execution of the system, that subset of the processes is activated which corresponds to the actual track followed through the CPG. The actual track taken depends on the value of certain conditions. For each individual track there exists an optimal schedule of the processes which produces a minimal delay. Let us consider the CPG in Fig.1. If all three conditions, C, D, and K are true, the optimal schedule requires P_1 to be activated at time t=0 on processor pe_1, and processor pe_2 to be kept idle until t=4, in order to activate P_3 as soon as possible (see Fig. 2a). However, if C and D are true but K is false, the optimal schedule requires to start both P_1 on pe_1 and P_{11} on pe_2 at t=0; P_3 will be activated in this case at t=6, after P_{11} has terminated and, thus, pe_2 becomes free (see Fig. 2b). This example reveals one of the difficulties when generating a schedule for a system like that in Fig. 1. As the values of the conditions are unpredictable, the decision on which process to activate on pe_2 and at which time has to be taken without knowing which values the conditions will later get. On the other side, at a certain moment during 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. Heuristic algorithms have to be developed to produce a schedule of the processes such that the worst case delay is as small as possible. The output of such an algorithm is a so called s*chedule table*. In this table there is one row for each ordinary or communication process, which contains activation times for that process corresponding to different values of the conditions. Each column in the table is headed by a logical expression constructed as a conjunction of condition val-



a) Optimal schedule of the track $D \wedge C \wedge K$

b) Optimal schedule of the track $D \wedge C \wedge \overline{K}$

Fig. 2. Optimal schedules for two tracks extracted from the CPG in Fig. 1

ues. Activation times in a given column represent starting times of the processes when the respective expression is true.

Table 1 shows a possible schedule table corresponding to the system depicted in Fig. 1. According to this schedule processes P_1, P_2, P_{11} as well as the communication process P_{18} are activated unconditionally at the times given in the first column of the table. No condition value has yet been determined to select between alternative schedules. Process P_{14}, on the other hand, has to be activated at t=24 if $D \wedge \overline{C} \wedge K$=*true* and t=18 if $D \wedge C \wedge K$=*true*. To determine the worst case delay, δ_{max}, we have to observe the rows corresponding to processes P_{10} and P_{17}: δ_{max}= max{35 + t_{10}, 30 + t_{17}}=40.

The schedule table contains all information needed by a distributed run time scheduler to take decisions on activation of processes. We consider that, during execution, a very simple scheduler located on each processor decides on process and communication activation depending on actual values of conditions. Once activated, a process executes until it completes. Only one part of the table has to be stored on each processor, namely the part concerning decisions which are taken by the corresponding scheduler.

Our goal is to derive a minimal worst case delay and to generate the corresponding schedule table given a conditional process graph $\Gamma(V^*, E_S^*, E_C^*, M)$ and estimated worst case ex-

Table 1: Schedule Table corresponding to the CPG shown in Fig. 1

	true	D	$D \wedge C$	$D \wedge C \wedge \overline{K}$	$D \wedge C \wedge K$	$D \wedge \overline{C}$	$D \wedge \overline{C} \wedge \overline{K}$	$D \wedge \overline{C} \wedge K$	$\overline{D}$	$\overline{D} \wedge C$	$\overline{D} \wedge \overline{C}$
P_1	0										
P_2	3										
P_3		6							6		
P_4						7					7
P_5							18	18			18
P_6				21	20					20	
P_7							21	21			21
P_8				29	28					28	
P_9				26	25					25	
P_{10}				35	34		27	26		34	26
P_{11}	0										
P_{12}			9			9					
P_{13}										10	13
P_{14}					18			24			
P_{15}				19			24				
P_{16}				15	15		15	15			
P_{17}				25	24		30	26		24	24
P_{18} (1→3)	3										
P_{19} (2→5)						9					8
P_{20} (3→10)				21	20		21	20		20	20
P_{21} (3→6)				19	18					18	
P_{22} (4→7)						12					13
P_{23} (6→8)				26	25					25	
P_{24} (7→10)							25	24			24
P_{25} (8→10)				33	32					32	
P_{26} (11→13)										8	11
P_{27} (11→12)			8			8					
P_{28} (12→14)					16			16			
P_{29} (12→15)				16			16				
P_{30} (13→17)										22	22
P_{31} (16→17)				23	22		23	22			
D	6										
C		7							7		
K			15			15					

ecution times for each process $P_i \in V^*$.

At a certain execution of the system, one of the N_{alt} alternative tracks through the CPG will be executed. Each alternative track corresponds to one subgraph $G_k \in \Gamma$, k=1, 2, ..., N_{alt}. For each subgraph there is an associated logical expression L_k (the label of the track) which represents the necessary conditions for that subgraph to be executed. The subgraph G_k contains those nodes P_j of the conditional process graph Γ, for which $L_k \Rightarrow X_{Pj}$ (X_{Pj} is the guard of node P_j and it has to be true whenever L_k is true). For the CPG in Fig. 1 we have 6 subgraphs (alternative tracks) corresponding to the following labels: $C \wedge D \wedge K$, $C \wedge D \wedge \overline{K}$, $C \wedge \overline{D}$, $\overline{C} \wedge D \wedge K$, $\overline{C} \wedge D \wedge \overline{K}$, $\overline{C} \wedge \overline{D}$.

If at activation of the system all the condition values *would be* known in advance, the processes *could be* executed according to the (near)optimal schedule of the corresponding subgraph G_k. Under these circumstances the worst case delay δ_{max} would be:

$\delta_{max} = \delta_M$, with

$\delta_M = \max\{\delta_k, k=1, 2, ..., N_{alt}\}$, where δ_k is the delay corresponding to subgraph G_k.

However, this is not the case as we do not assume any prediction of the condition values at the start of the system. Thus, what we can say is only that[1]: $\delta_{max} \geq \delta_M$.

A scheduling heuristic has to produce a schedule table for which the difference $\delta_{max} - \delta_M$ is minimized. This means that the perturbation of the individual schedules, introduced by the fact that the actual track is not known in advance, should be as small as possible.

C. Conditions, Guards and Influences

We first introduce some notations. If E is a logical expression, we use the notation $/E/$ to denote the set of conditions used in E. Thus, $/C \wedge \overline{D} \wedge K/ = /C \wedge D \wedge \overline{K}/ = \{C, D, K\}$; $/true/ = /false/ = \emptyset$. Similarly, if M is a set of condition values, $/M/$ is the set of conditions used in M. For example, if $M = \{C, \overline{D}, K\}$, then $/M/ = \{C, D, K\}$. For a set M of condition values, we denote with $\wedge M$ the logical expression consisting of the conjunction of that values. If $M = \{C, \overline{D}, K\}$ then $\wedge M = C \wedge \overline{D} \wedge K$.

Activation times in the schedule table are such that a process is started only after its predecessors, corresponding to the actually executed track, have terminated. This is a direct consequence of the data-flow dependencies as captured in the CPG model. However, in the context of the CPG semantics, there are more subtle requirements which the schedule table has to fulfill in order to produce a deterministic behavior which is correct for any combination of condition values:

R1. If for a certain execution of the system the guard X_{Pi} becomes true then P_i has to be activated during that execution.

R2. If for a certain process P_i, with guard X_{Pi}, there exists an activation time in the column headed by the logical expression E_k, then $E_k \Rightarrow X_{Pi}$; this means that no process will be activated if the conditions required for its execution are not fulfilled.

R3. Activation times and the corresponding logical expressions have to be determined so that a process is activated not more then one single time for a given execution of the system. Thus, if for a certain execution, a process P_i is scheduled to be executed at τ_{Pi}^{Ej} (placed in column headed by E_j), there is no other execution time $\tau_{Pi}^{Ek} \neq \tau_{Pi}^{Ej}$ for P_i (placed in column headed by E_k) so that E_k becomes *true* during the same execution.

R4. Activation of a process P_i at a certain time t has to depend only on condition values which are determined at the respective moment t and are known to the processor which takes the decision on P_i to be executed.

The correct behavior, according to the semantics associated to conditions, is guaranteed by requirements R1 and R2. R3 guarantees the functionality as a single rate system. Requirement R4 states that decisions at any location of the distributed system are based on information which is available at the particular location and at the given time.

A scheduling algorithm has to traverse, in one way or another, all the alternative tracks in the CPG and to place activation times of processes and the corresponding logical expressions into the schedule table. For a given execution track, labelled L_k, and a process P_i which is included in that track, one single activation time τ_{Pi}^{Lk} has to be placed into the table (see R3 above); the corresponding column will be headed by an expression E. We say that $/E/$ is the set of conditions used for scheduling P_i. It is obvious that $/E/ \subseteq /L_k/$ and $L_k \Rightarrow E$.

Which are the conditions in the set $/E/$, to be used for scheduling a certain process? This is a question to be answered by the scheduling algorithm. A straightforward answer would be: those conditions which determine if the process has to be activated or which have an influence on the moment when this activation has to occur.

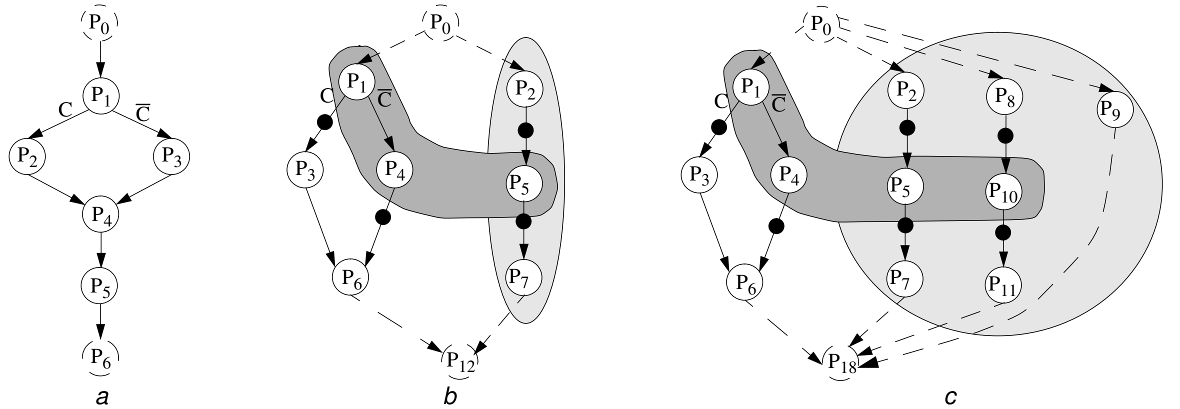
In the following section we will give a more exact answer to the question above. As a prerequisite we first define the so called *guard set* GS_{Pi} and *influence set* IS_{Pi} of a process P_i.

The *guard set* GS_{Pi} of a process P_i is the set of conditions which decide whether or not the process is activated during a certain execution of the system. $GS_{Pi} = /X_{Pi}/$, where X_{Pi} is the guard of process P_i. In Fig. 1, for example, $X_{P3}=true$, $X_{P14}=D \wedge K$, $X_{P15}=D \wedge \overline{K}$; thus, $GS_{P3}=\emptyset$, $GS_{P14}=GS_{P15}=\{D, K\}$.

However, not only the conditions in the guard set have an impact on the execution of a process. There are other conditions which do not decide *if* a process has or has not to be activated during a given execution, but which influence the moment *when* the process is scheduled to start. In Fig. 3a, for example, $GS_{P4}=\emptyset$; process P_4 is executed regardless if C is *true* or *false*. However, the activation time of P_4 depends on the value of condition C: P_4 will be activated at $\tau_{P4}^{C} = t_{P1}+t_{P2}$, if C=*true*, and at $\tau_{P4}^{\overline{C}}=t_{P1}+t_{P3}$, if C=*false*. As a consequence, the activation time of P_5 is also dependent on the value of C. We say that both P_4 and P_5 are influenced by C, and this influence is induced to them (from their predecessors) by succession in the CPG.

However, not only by succession can the influence of a condition be induced to a process. This is illustrated in Fig. 3b, where we assume that each communication is mapped to a distinct bus and that $t_{P1} < t_{P2} + t_{2,5} < t_{P1} + t_{P4}$. For C=*true* we have τ_{P5}^{C} $=\max(t_{P2}+t_{2,5}, t_{P1})=t_{P2}+t_{2,5}$. For C=*false*, P_4 becomes ready

[1] This formula to be rigorously correct, δ_M has to be the maximum of the *optimal* delays for each subgraph.



Process mapping to processors is illustrated by shading; all processors are programmable.

Fig. 3. Examples of Conditional Process Graphs

and will be scheduled before P_5; thus, $\tau_{P5}^{\overline{C}} = \max(t_{P2}+t_{2,5}, t_{P1}+t_{P4}) = t_{P1}+t_{P4}$. The activation time of P_5 depends on condition C, because P_5 shares the same programmable processor with P_4 and the activation of P_4 is conditioned by C. In this case the influence of condition C is induced on P_5 from P_4 by resource sharing. This influence is then further induced from P_5, by succession, to the communication process between P_5 and P_7 and to P_7. P_2 and the communication P_2 to P_5 as well as P_1 are not influenced by C.

Fig. 3c illustrates a more complex situation. We assume that communications are mapped to distinct buses and that, if there is a conflict for the processor, P_8 is scheduled first and, with decreasing priority, P_9, P_{11}, and P_2. Concerning execution times, we assume: $t_{P1}<t_{P8}+t_{8,10}<t_{P1}+t_{P4}$, $t_{P8}+t_{8,10}+t_{P10}+t_{10,11}<t_{P8}+t_9<t_{P1}+t_{P4}+t_{P10}+t_{10,11}$. Under these circumstances, if C=*true*, P_{11} is ready when the processor is released by P_9, and thus will be scheduled before P_2; we have $\tau_{P2}^{C}=t_{P8}+t_{P9}+t_{P11}$. For C=*false*, P_{10} is delayed because of the earlier activation of P_4, which makes that P_{11} will not be ready when P_9 has finished; thus, P_2 is scheduled before P_{11} and $\tau_{P2}^{\overline{C}}=t_{P8}+t_{P9}$. In this case, the influence of C on P_2 has been induced by resource sharing from P_{11} which is influenced from its predecessor P_{10}.

We can observe from the examples above that the influence of a certain condition on a process can only be determined dynamically, during the scheduling process. A particular scheduling policy, priorities and execution times have all an impact on whether at a certain moment the scheduling of a process is influenced or not by a condition. In Fig. 3c, for example, scheduling of P_2 is influenced by C only because the scheduler has considered P_8 with a higher priority and this is also why P_8 is not influenced by C.

The set of conditions to be used for scheduling a certain process P_i at a moment t, consists of the guard set GS_{Pi} corresponding to that process and of all the other conditions which influence the activation of P_i. An efficient heuristic has to be developed for the scheduler in order to determine these conditions so that a correct and close to optimal schedule table is produced. In order to solve this problem we define the so called influence set IS_{Pi} corresponding to a process P_i. It is important to notice that this set of conditions can be determined statically for a given CPG and it consists of all conditions which are not part of the guard set but *potentially could* influence the scheduling of process P_i.

A condition C is in the *influence set* IS_{Pi} of a process P_i, if C is not in the guard set GS_{Pi} of P_i and if the influence of C can be induced to P_i *by succession* or *by resource sharing*.

The influence of a condition C can be induced to a process P_i *by succession* if P_i is a successor of a process P_j and the following condition is satisfied: $C \in IS_{Pj}$ or $C \in GS_{Pj}$.

The influence of a condition C can be induced to a process P_i *by resource sharing* if all the following three conditions are satisfied:

1. P_i is mapped on a programmable processor or a bus.
2. P_i is neither the process that computes condition C nor a predecessor of that process.
3. There exists a process P_j mapped to the same resource as P_i and
   a. P_j is neither a predecessor nor a successor of P_i, and
   b. $C \in IS_{Pj}$ or $C \in GS_{Pj}$.

In Fig. 3b, for example, $IS_{P5}=IS_{P6}=IS_{P7}=IS_{5,7}=\{C\}$. For the other processes, the influence set is void. P_5 can be influenced by resource sharing while the other processes by succession. In Fig. 3c the influence of C can be induced by succession to P_6 and by resource sharing to P_5 and P_{10}. This is then transferred by succession to P_7, P_{11}, and to the communication processes preceding them. The influence can be induced by resource sharing to P_2, P_8, and P_9 and then by succession to the communications following P_2 and P_8. For both examples we considered that communications do not share any bus. If in Fig. 3b we consider that all communications are mapped to the same bus, then $IS_{2,5}=\{C\}$, as result of resource sharing.

In the next section we will see how the influence set is utilized by a particular scheduling algorithm in order to determine the actual set of conditions used for scheduling a process.

The activation of a process P_i at a certain moment t depends on the conditions specified in the expression heading the respective column of the schedule table (the conditions used to schedule P_i). According to requirement R4, the values of all these conditions have to be determined at the respective moment t and have to be known to the processor $Q(P_i)$. $Q(P_i)=M(P_i)$, the processor which executes P_i, if P_i is not a communication process; if P_i is a communication process, then $Q(P_i)=M(P_j)$, where P_j is the process that initiates the communication.

The value of a condition is determined at the moment τ at which the corresponding disjunction process terminates. This means that at any moment t, $t \geq \tau$, the value of the condition is available for scheduling decisions on the processor which has executed the corresponding disjunction process. However, in order to be available on any other processor, the value has to arrive at that processor. The scheduling algorithm has to consider both the time and the resource (bus) needed for this communication. Transmissions of condition values are scheduled as communications on buses. These communication processes are not represented in the CPG and they are the only activities which are not a priori mapped to a specific resource (bus). A condition communication will be mapped as part of the scheduling process to a bus. For broadcasting of condition values only buses are considered to which all processors are connected and we assume that at least one such bus exists[1]. The time τ_0 needed for such a communication is the same for all conditions and depends on the features of the employed buses. The transmitted condition value is available for scheduling decisions on all other processors τ_0 time units after initiation of the broadcast. For the example given in Table 1 communication time for condition values has been considered τ_0=1. In Table 1 the last three rows indicate the schedule for communication of the values of conditions C, D, and K.

IV. SCHEDULING OF CONDITIONAL PROCESS GRAPHS

A. The Scheduling Algorithm

Our algorithm for scheduling of conditional process graphs relies on a list scheduling heuristic. List scheduling heuristics [29] are based on ordered lists from which processes are extracted to be scheduled at certain moments. In the algorithm presented in Fig. 4 we have such a list, *Ls_Ready*, in which processes are placed in increasing order of the time when they become ready for execution (this time is stored as an attribute $p.t_{ready}$ of each process p in the list, and is the moment when all the predecessors of p have terminated). If p is mapped to a hardware processor, it is scheduled, without any restriction, at the moment when it is ready. On a programmable processor or bus, however, a new process p can be scheduled only after the respective processing element $M(p)$ becomes free ($free_{M(p)}$ indicates this moment). There can be several processes mapped on $M(p)$, so that, at a given moment, $p.t_{ready} \leq free_{M(p)}$. All of them will be ready when processing element $M(p)$ becomes free. From these processes function *Select* will pick out the one which has the highest priority assigned to it, in order to be scheduled. This priority function is one of the essential features which differentiate various list scheduling approaches. We will later discuss the particular priority function used by our algorithm.

The algorithm presented in Fig. 4 is able to schedule, based on a certain priority function, process graphs without conditional control dependencies. In Fig. 5 we show the algorithm for scheduling of conditional process graphs. It is based on the general principles of list scheduling, using the list *Ls_Ready* to store processes ordered by the time when they become ready for execution. *Cond_Set* is the set of those condition values for which the corresponding disjunction process (which computes the value) has been already scheduled. Each condition value $c \in$ *Cond_Set* has two attributes attached: $c.t_{comp}$ is the moment when the value c has been computed (the disjunction process has terminated) and $c.t_{arr}$ is the time when c is available on processors different from the one that executes the disjunction process. The recursive procedure *List_schedule_CPG* traverses the CPG analyzing each possible alternative track and considering for each track only the processes executed for the respective condition values. The algorithm, thus, proceeds along a binary decision tree corresponding to the alternative tracks, which is explored in depth first order.

List_schedule ()
  for each processing element pe_i, i=1,2, ..., N_{pe}, **do**
    $free_{pei}$=0
  end for;
  schedule P_0 at t=0; initialize ready process list *Ls_Ready*
  with direct successors of P_0;   -- P_0 is the source node
  while *Ls_Ready* is not empty **do**
    p=*Head*(*Ls_Ready*);
    if $M(p) \in HP$ **then**   -- hardware supports several
        schedule p at t=$p.t_{ready}$ -- processes at a time;
    else
        p=*Select*(*Ls_Ready*, $M(p)$);
        schedule p at t=max($p.t_{ready}$, $free_{M(p)}$);
        $free_{M(p)}$=t+t_p
    end if;
    delete p from *Ls_Ready*;
    Ls_Ready = *Ls_Ready* + processes which become
                ready after p is executed
  end while
end List_schedule;

Fig. 4. Basic List Scheduling Algorithm

Fig. 6 shows the decision tree explored during generation of Table 1 for the CPG in Fig. 1. The nodes of the tree correspond to the states reached during scheduling when a disjunction process has been scheduled and the algorithm branches for the alternative condition values. Whenever a process P_i has been scheduled, it is eliminated from the ready list *Ls_Ready* and all processes which become ready after P_i has been executed are added to *Ls_Ready*. If, however, P_i is a disjunction process computing condition C, the two possible alternatives are handled separately. First, the processes are added to *Ls_Ready* which become ready on the true alternative and scheduling is continued with the condition set *Cond_Set* $\cup \{C\}$. Then, those processes are added to *Ls_Ready* which become ready on the false alternative and scheduling is continued with the condition set *Cond_Set* $\cup \{\overline{C}\}$.

Let us suppose that a process P_i has become ready and has been added to the list *Ls_Ready* before the branching for a certain condition Z. If the process is scheduled before the branching, it will be eliminated from *Ls_Ready* and, thus, not handled any more during further exploration of the tree below that branching point. This scheduling decision is the one and only

[1] This assumption is made for simplification of the further discussion. If no bus is connected to all processors, communication tasks have to be scheduled on several buses according to the actual interconnection topology.

```
List_schedule_CPG (Ls_Ready, free_pei|i=1... N_pe, Cond_Set)
  p=empty_process;
  while Ls_Ready is not empty and p is not a disjunction process do
    p=Head(Ls_Ready);
    if M(p)∈HP then                 -- hardware processors support several processes at a time;
        τ=p.t_ready
    else                            -- programmable processors and buses support one process at a time
        p=Select(Ls_Ready, M(p));
        τ=max(p.t_ready, free_M(p))
    end if;
    Z={c | c∈Cond_Set and c.t_comp≤τ};                  -- set of condition values computed at τ
    CS={c | c∈Z and (/{c}/ ⊂ GS_p or /{c}/ ⊂ IS_p)};     -- set of condition values used to schedule p
    Exp = ∨CS;                                           -- the logical expression used to schedule p
    θ = maximum among c.t_arr, where c∈CS and c is computed on a processor different from Q(p);
          -- this is the earliest time when all condition values needed to schedule p are known on processor Q(p)
    schedule p at t=max(τ, θ), controlled by expression Exp;
    free_M(p)=t+t_p;
    delete p from Ls_Ready;
    if p is not a disjunction process then
        Ls_Ready = Ls_Ready + processes which become ready after p is executed
    end if
  end while
  if p is a disjunction process then        -- we consider that C is the condition computed by p
    C.t_comp=t+t_p;
    Schedule the communication of the condition value and fix C.t_arr accordingly;
    Ls_Ready_true = Ls_Ready + processes on the true alternative, which become ready after p is executed;
    List_schedule_CPG(Ls_Ready_true, free_pei|i=1... N_pe, Cond_Set ∪ {C});
    Ls_Ready_false = Ls_Ready + processes on the false alternative, which become ready after p is executed;
    List_schedule_CPG(Ls_Ready_false, free_pei|i=1... N_pe, Cond_Set ∪ {C̄});
end List_schedule_CPG;
```

Fig. 5. List Scheduling of Conditional Process Graphs

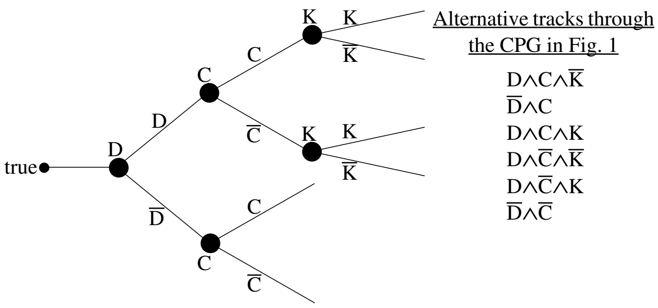


Fig. 6. The decision tree explored at generation of the schedule table for the CPG in Fig. 1

valid for P_i on all tracks including that point. If P_i has not been scheduled before branching on Z, it will be inherited by *Ls_Ready* on both branches (these branches belong to different execution tracks through the CPG) and will be eventually scheduled on both of them. There are two important conclusions to be drawn here:

1. If a process becomes ready on a certain execution track through the CPG, it will be scheduled for activation on that track. By this, requirement R1 is satisfied.
2. During exploration of a certain execution track, a given process which has to be activated on that track, is considered one single time for scheduling.

We will now discuss three additional aspects concerning the scheduling algorithm in Fig. 5: the set of conditions used to schedule a certain process, the communication of condition values, and the priority assignment.

B. Conditions Used for Process Scheduling

When a process is ready to be scheduled at a moment τ, the set of conditions to be used to schedule it has to be determined:

A condition C will be used at moment τ for scheduling a process P_i, if C is a member of the guard set or of the influence set of P_i and, according to the current schedule, the value of C has already been computed at time τ.

There are several aspects to be discussed in the context of this rule:

1. As discussed in section III, the influence set IS_{Pi} consists of all conditions, except those in the guard set, which *potentially could have* an influence on the execution of P_i. The above rule selects from the influence set those conditions which *actually have* an influence on the execution of P_i, considering the actual track which is scheduled and the particular moment τ. Those conditions are eliminated which have not been computed at moment τ, according to the current schedule, and, thus, are excluded to have any influence (by sharing or succession) on the scheduling of P_i in the current context.
2. Every condition C, which is a member of the guard set of a process P_i ($C \in GS_{Pi}$), is used to schedule P_i. By this, requirement R2 is satisfied.
   We use the notation c for the value of condition C corresponding to the current track. C is a member of the guard set of P_i, which means that it is computed by a predecessor of P_i. Thus, $c \in Cond\_Set$ and $c.t_{comp} \leq \tau$ (any predecessor of P_i has been scheduled and has terminated before P_i be-

comes ready). Consequently, $c \in Z$ and also $c \in CS$ (Fig. 5), which means that C will be used for scheduling P_i.

3. Consider a process P_i with the guard X_{Pi}. For a given execution track, labelled L_k ($L_k \Rightarrow X_{Pi}$), P_i is scheduled at τ^{E}_{Pi}, conditioned by expression E. Let us suppose that the same process P_i is scheduled also on another track, labelled L_q ($L_q \Rightarrow X_{Pi}$) at $\tau^{E'}_{Pi} \neq \tau^{E}_{Pi}$, conditioned by expression E'. We have shown in the previous subsection that P_i is scheduled one single time for a given execution track. In order to prove that requirement R3 is satisfied, we have to show that during execution of neither of the two tracks both expressions E and E' can become true.
We have $E = X_{Pi} \wedge Y$ and $E' = X_{Pi} \wedge Y'$, where Y and Y' are conjunctions of condition values selected from the influence set of P_i; we know that $L_k \Rightarrow Y$ and $L_q \Rightarrow Y'$. Let us consider $I_1, I_2, ...I_N$, the conditions which appear both in L_k and in L_q, but with complementary values. If we take, for example, the tracks labelled $D \wedge \overline{C} \wedge \overline{K}$ and $D \wedge C \wedge K$ in Fig. 6, then I_1 is C and I_2 is K. For any pair of tracks there has to be at least one such condition (the one on which the two tracks branch). If no one of the conditions $I_1, ...I_N$, is a member of the influence set of P_i, then P_i is scheduled at identical times on the two tracks (there is no influence on the process from any condition which differentiates the tracks) and, thus, there is no conflict ($\tau^{E'}_{Pi} = \tau^{E}_{Pi}$).
Let us suppose that I_1 is a member of the influence set of P_i. If P_i is considered for scheduling before the algorithm has branched on condition I_1, then the respective scheduling time is the one and only for both tracks L_k and L_q (see previous subsection) and there is no conflict.
If P_i is considered for scheduling after the algorithm has branched on condition I_1, but before $I_1.t_{calc}$, we have $\tau^{E'}_{Pi} = \tau^{E}_{Pi}$ and, thus, no conflict (before $I_1.t_{calc}$, that means before the disjunction process has terminated and I_1 has been computed, the two branches act identically). If P_i is scheduled after $I_1.t_{calc}$, then I_1 will be used, to schedule P_i on both tracks ($I_1 \in /E/$ and $I_1 \in /E'/$) but with complementary values. Consequently, E and E' cannot both become true during the same execution.

C. Communication of Condition Values

In subsection III/C we have shown that the values of all conditions needed to schedule a certain process P_i have to be known on processor $Q(P_i)$ at the activation moment of that process. In Fig. 5 we have used θ to denote the earliest moment when all these conditions are available, and process P_i will not be scheduled before θ. Hence, correctness requirement R4 is also satisfied by the algorithm in Fig. 5.

Transmission of a value for condition C is scheduled as soon as possible on the first bus that becomes available after termination of the disjunction process which computes C. At the same time, the moment when the respective value becomes available on processors different from the one running the disjunction process has to be determined: $C.t_{arr} = t + \tau_0$, where t is the time at which transmission of the condition value has been scheduled. We mention that communication of a condition value will only be scheduled if there exists at least one process P_j so that $C \in GS_{Pj}$ or $C \in IS_{Pj}$, and $Q(P_j)$ is different from the processor running the disjunction process.

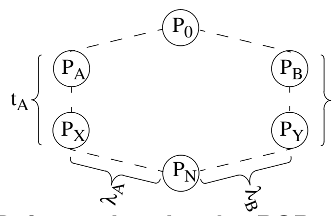


Fig. 7. Delay estimation for PCP scheduling

D. Priority Assignment

In this subsection we first introduce the priority function used by our list scheduling algorithm and then show how priorities are assigned in the particular context of a CPG.

List scheduling algorithms rely on priorities in order to solve conflicts between several processes which are ready to be executed on the same programmable processor or bus. Such priorities very often are based on the critical path (CP) from the respective process to the sink node. In this case (we call it CP scheduling), the priority assigned to a process P_i will be the maximal total execution time on a path from the current node to the sink:

$$l_{Pi} = \max_{k} \sum_{P_j \in \pi_{ik}} t_{Pj}$$, where π_{ik} is the k^{th} path from node P_i to the sink node.

Considering the concrete definition of our 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 mapping.

Let us consider the graph in Fig. 7 and suppose that the list scheduling algorithm has to decide between scheduling process P_A or P_B which are both ready to be scheduled on the same programmable processor or bus pe_i. In Fig. 7 we have 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 $\quad l_{PB} = t_B + \lambda_B$.

However, we will not use the length of these critical paths as a priority. Our policy, called partial critical path scheduling (PCP), 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:

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

We select the alternative that offers the perspective of the shorter delay $L = \min(L_{PA}, L_{PB})$. 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 $L = L_{PB}$.

As a conclusion, for PCP scheduling we use the value of λ_{Pi} as a priority criterion instead of the length l_{pi} of the whole critical path. Thus, we take into consideration only 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 $M(P_i)$.

For evaluation of the PCP policy we used 1250 graphs generated for experimental purpose. 250 graphs have been generated for each dimension of 20, 40, 75, 130, and 200 nodes. We considered architectures consisting of one ASIC and one to eleven processors and one to eight buses. We have evaluated the percentage deviations of the schedule lengths produced by CP, UB[1] and PCP based list scheduling from the lengths of the optimal schedules. The optimal schedules were produced running a branch-and-bound based algorithm. The average deviation for all graphs is 4.73% with UB, 4.69% with CP, and 2.35%, two times smaller, with PCP. Further details concerning the PCP policy and its experimental evaluation can be found in [33].

How to apply a PCP priority assignment to conditional process graphs? The problem is that in a CPG a process potentially belongs to several tracks and relative to each of these tracks it has a different PCP priority. Thus, a process is characterized not by a single priority but by a set of priorities, one for each track to which the process belongs. P_{11} in Fig 1, for example, has priority 14 corresponding to all tracks which imply condition value D, priority 15 corresponding to the tracks which imply $D \wedge K$, and 18 for those implying $D \wedge \overline{K}$.

When a certain process is considered for scheduling, the scheduler does not know the particular track to be followed and, thus, it is not obvious which priority to use for the given process. We have used a very simple and efficient heuristic for priority assignment to processes in a CPG: for each process P_i, that PCP priority is considered which corresponds to the most critical track to which P_i belongs. This means that P_i gets the maximal priority among those assigned for each particular track. In the case of P_{11} (Fig. 1), the chosen priority will be 18.

From the practical point of view the above heuristic means that priorities are assigned to processes by simply applying PCP priority assignment to the simple (unconditional) process graph which is obtained from the CPG by ignoring conditional dependencies and considering conditional edges as simple edges.

By this priority assignment policy, we try to enforce schedules so that the execution of longer tracks is as similar as possible to the best schedule we could generate if we would know in advance that the particular track is going to be selected. By introducing perturbations (possible delays) into the shorter tracks and let the longer ones proceed as similar as possible to their (near)optimal schedule, we hope to produce a schedule table with a minimized worst case execution time. This is in line with the objective formulated in section III/B.

V. A TTP BASED SYSTEM ARCHITECTURE

In the previous sections we have considered a general system architecture with a relatively simple communication infrastructure. Such a model is representative for a large class of applications and this is the reason why it has been used in order to develop our scheduling strategy for CPGs.

When designing a particular system, however, very specific architectural solutions and particular communication protocols have often to be used. In such a situation particularities of the communication infrastructure have to be considered for system scheduling. At the same time, several parameters of the communication protocol have to be optimized in order to meet time constraints at the lowest possible cost. These two aspects are strongly interrelated and have a strong impact on the quality of the final design.

[1] Urgency based (UB) scheduling uses the difference between the ALAP schedule of a process and the current time, as a priority.

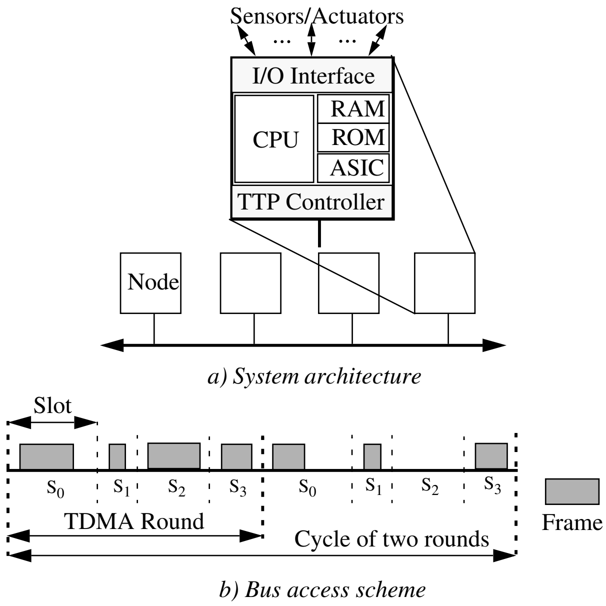


Fig. 8. TTP based system architecture

We consider a system architecture built around a communication model which is based on the time-triggered protocol (TTP) [48]. TTP is well suited for the category of systems targeted by our synthesis environment. These are systems which include both data-flow and control aspects and which are implemented on distributed architectures. They have to fulfill hard real-time constraints and are often safety-critical. TTP is also perfectly suited for systems implemented with static non-preemptive scheduling and, thus, represents an ideal target architecture for the scheduling approach presented in the previous sections.

While in the previous sections we emphasized the impact of conditional dependencies on system scheduling, in the rest of the paper we mainly concentrate on how the communication infrastructure and protocol have to be considered during scheduling and how they can be optimized in order to reduce the system delay. In this section we describe our hardware and software architecture based on the TTP. In section VI we then show how scheduling and optimization of the communication protocol interact with each other and how they can be considered together in order to improve the overall system performance.

A. Hardware Architecture

We consider architectures consisting of nodes connected by a broadcast communication channel (Fig. 8a). Every node consists of a TTP controller [48], 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.

Communication between nodes is based on the TTP. TTP was designed for distributed real-time applications that require predictability and reliability. The communication is performed on 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) (Fig. 8b). 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 (see Fig. 9).

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

B. Software Architecture

We have designed a software architecture which runs on the CPU in each node, and which has a real-time kernel as its main component. Each kernel has a schedule table that contains all the information needed to take decisions on activation of processes and transmission of messages, based on the values of conditions.

The message passing mechanism is illustrated in Fig. 9, 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. 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

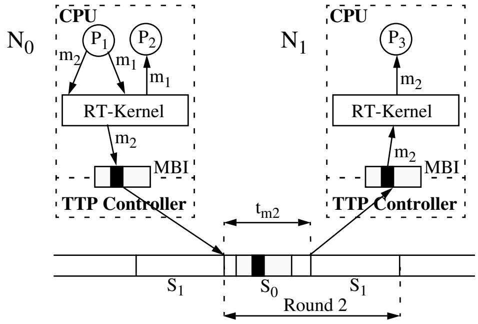


Fig. 9. Message passing mechanism

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 it into a frame in the MBI. 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 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. 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.

In [51] we presented a detailed discussion concerning the overheads due to the kernel and to every system call. We also presented formulas for derivation of the worst case execution delay of a process, taking into account the overhead of the timer interrupt, the worst case overhead of the process activation and message passing functions.

VI. SCHEDULING WITH BUS ACCESS OPTIMIZATION

A. Problem Formulation

We consider a system captured as a CPG. The target architecture is as described in section V. Each process is mapped on a CPU or an ASIC of a node. We are interested to derive a delay on the system execution time 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.

For each message its length b_{mi} is given. If the message is exchanged by two processes mapped on the same node, the message communication time is completely accounted for in the worst case execution delay of the two processes as shown in subsection V/B. Thus, from the scheduling point of view, communication of such messages is instantaneous. This is in line with our CPG representation where no communication processes are introduced between processes mapped to the same processor. However, 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 frame. The number of messages that can be packaged depends on the slot

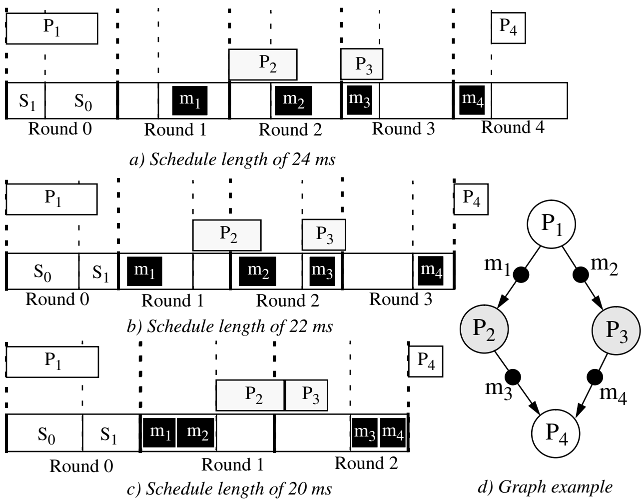


Fig. 10. Scheduling example

length corresponding to the node. The effective time spent by a message m_i on the bus is $t_{mi} = b_{Si}/T$, where b_{Si} is the length of slot S_i and T is the transmission speed of the channel. In Fig. 9, t_{m2} depicts the time spent by m_2 on the bus. The previous equation shows that the communication time t_{mi} does not depend on the bit length b_{mi} 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 Fig. 10. In Fig. 10d we have a 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 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 Fig. 10a, where slot S_1 is scheduled first and slot S_0 is second, we have a resulting schedule length of 24ms. However, if we swap the two slots inside the TDMA round without changing their lengths, we can improve the schedule by 2ms, as seen in Fig. 10b. Further more, if we have the TDMA configuration in Fig. 10c 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 20ms which is optimal. However, increasing the length of slots is not necessarily improving a schedule, as it delays the communication of messages generated by other nodes.

Our optimization strategy, described in the following subsections, determines the sequence and length of the slots in a TDMA round with the goal to reduce the delay on the execution time of the system. Before discussing this optimization of the bus access scheme, we first analyze how the particularities of the TTP protocol have to be taken into consideration at scheduling of CPGs.

B. Scheduling with a Given Bus Access Scheme

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 our algorithm *List_schedule_CPG* (Fig. 5) presented in section IV. 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 *Plan_message* in Fig. 11 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 *time_ready*. *Plan_message* returns the first round and corresponding slot (the slot corresponding to $Node_m$) which can host the message. In Fig. 11 *round_length* is the length of a TDMA round expressed in time units (in Fig. 12, for example, *round_length*=18ms). The first round after *time_ready* 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 planning scheme, the algorithm in Fig. 5 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.

Let us consider the graph in Fig. 12c, 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 subsection IV/D) 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 depicted in Fig. 12a. On the other hand, if we schedule P_2 first, the resulting schedule, depicted in Fig. 12b, 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 Fig. 12 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 Fig. 11, 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 (*time_ready*=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 com-

```
Plan_message (time_ready, b_m, Node_m)
  Slot=the slot assigned to Node_m;                      -- the slot in which the message has to be sent
  Round=⌊time_ready / round_length⌋;                     -- the first round which could be a candidate
  if time_ready - Round*round_length > start_Slot then       -- is the right slot in this round already gone?
     Round = Round+1                                         -- if yes, take the next round
  end if;
  while b_m > b_Slot - b_occupied do                     -- is enough space left in the slot for the message?
     Round = Round+1                                     -- if not, take the next round
  end while;
  return (Round, Slot)                                   -- return the right round and slot
end Plan_message;
```

Fig. 11. Message communication planning

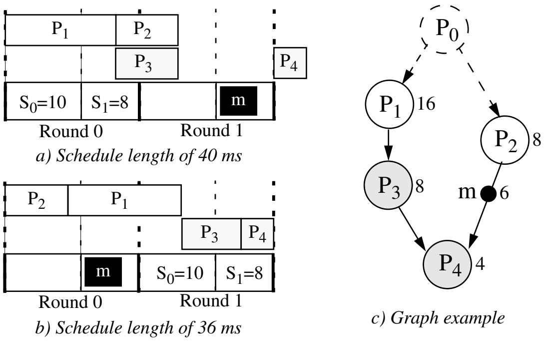


Fig. 12. Priority function example

puting the partial critical path of P_2, which now results in δ+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 Fig. 12, 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 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 Fig. 11, but without taking into consideration space limitations in slots. As the experimental results (section VII) 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.

C. Optimization of the Bus Access Scheme

In the previous subsection we have shown how our algorithm *List_schedule_CPG* (Fig. 5) can produce an efficient schedule for a CPG, given a certain TDMA bus access scheme. However, as discussed in subsection VI/A, both the ordering of slots and the slot lengths strongly influence the worst case execution delay of the system. In Fig. 13 we show 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.

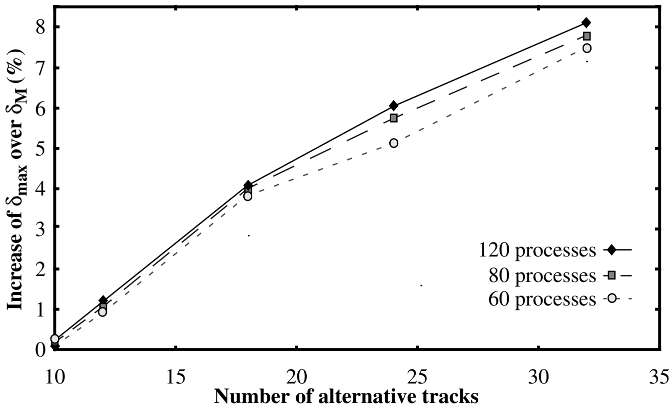
The initial solution, the so called "straightforward" one, assigns in order nodes to the slots ($Node_{Si}$=N_i) and fixes the slot length $length_{Si}$ to the minimal allowed value, which is equal to the length of the largest message generated by a process assigned to $Node_{Si}$. The algorithm starts 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 *List_Schedule_CPG*. 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 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 *Optimize_access_1*. A second alternative, *Optimize_access_2*, 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 straight-forward solution is performed which generates the recommended slot lengths. These lengths are produced by the *Plan_message* function (Fig. 11), 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 Fig. 13 only selects among the recommended lengths when searching for the right dimension of a certain slot.

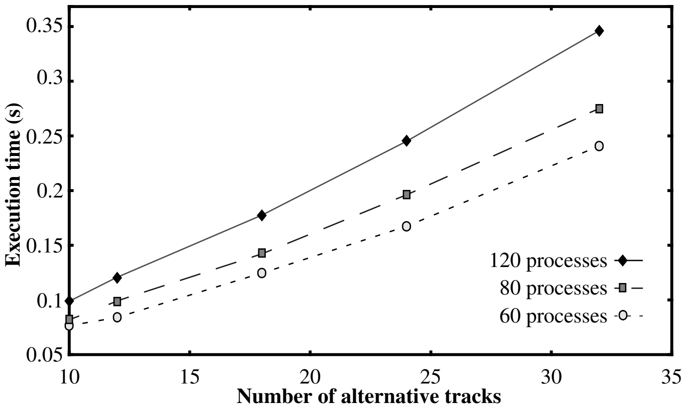
As the experimental results show (see section VII), optimi-

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

Fig. 13. Optimization of the bus access scheme



a) Increase of the worst case delay



b) Execution time of the scheduling algorithm

Fig. 14. evaluation of the scheduling algorithm for CPGs

zation of the bus access scheme can produce huge improvements in terms of performance. As expected, the *Optimize_access_2* alternative is much faster then the first one, while the quality of the results produced is only slightly lower.

VII. EXPERIMENTAL RESULTS

In the following two subsections we show a series of experiments which demonstrate the effectiveness of the proposed algorithms. The first set of results is related to the scheduling of conditional process graphs while the second group of experiments targets the problem of scheduling with optimization of the bus access scheme. As a general strategy, we have evaluated our algorithms performing experiments on a large number of test cases generated for experimental purpose. We then have validated the proposed approaches using real-life examples. All experiments were run on SPARCstation 20.

A. Evaluation of the Scheduling Algorithm

As discussed in subsection III/B, there are N_{alt} alternative tracks through a CPG, and a schedule could be generated for each one separately. Suppose δ_M is the longest of these schedules. This, however, does not mean that the worst case delay δ_{max}, corresponding to the CPG, is guaranteed to be δ_M. Such a delay cannot be guaranteed in theory, as the values of conditions, and thus the actual track to be followed, cannot be predicted. The objective of our scheduling heuristic is to generate a schedule table so that the difference $\delta_{max}-\delta_M$ is minimized.

For evaluation of the scheduling algorithm we used 1080 conditional process graphs generated for experimental purpose. 360 graphs have been generated for each dimension of 60, 80, and 120 processes. The number of alternative tracks through the graphs is 10, 12, 18, 24, or 32. Execution times were assigned randomly using both uniform and exponential distribution. We considered architectures consisting of one ASIC and one to eleven processors and one to eight buses.

Fig. 14a presents the percentage increase of the worst case delay δ_{max} over the delay δ_M of the longest track. The delay δ_M has been obtained by scheduling separately each alternative track trough the respective CPG, using PCP list scheduling, and selecting the delay which corresponds to the longest track. The average increase is between 0.1% and 8.1% and, practically, it does not depend on the number of processes in the graph but only on the number of alternative tracks. It is worth to be mentioned that a zero increase ($\delta_{max}=\delta_M$) was produced for 90% of the graphs with 10 alternative tracks, 82% with 12 tracks, 57% with 18 tracks, 46% with 24 tracks, and 33% with 32 tracks.

Concerning execution time, the interesting aspect is how the algorithm scales with the number of alternative tracks and that of processes. The worst case complexity of the scheduling algorithm depends on the number of tracks, which theoretically can grow exponentially. However, such an explosion is unlikely for practically significant applications. Fig. 14b shows the

Table 2: Worst case delays for the OAM block

Mode	System model		δ_{max} (ns)									
	nr. of processes	nr. of tracks	1P/1M		1P/2M		2P/1M			2P/2M		
			µp1	µp2	µp1	µp2	2×µp1	2×µp2	µp1+µp2	2×µp1	2×µp2	µp1+µp2
Mode 1	32	6	4471	2701	4471	2701	2932	2131	2532	2932	1932	2532
Mode 2	23	3	1732	1167	1732	1167	1732	1167	1167	1732	1167	1167
Mode 3	42	8	5852	3548	5852	3548	5033	3548	3548	5033	3548	3548

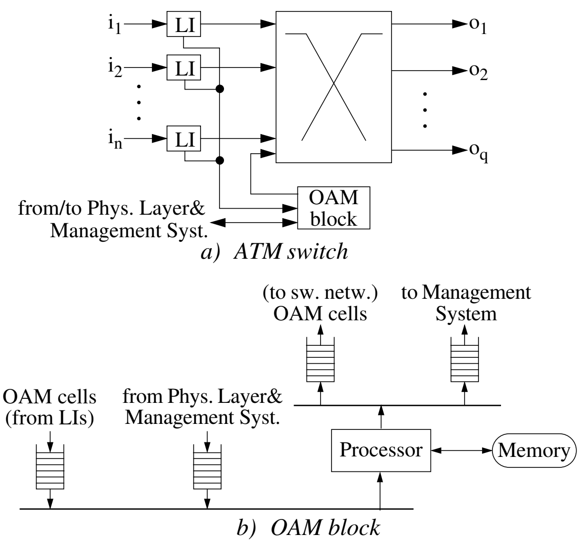


Fig. 15. ATM switch with OAM block

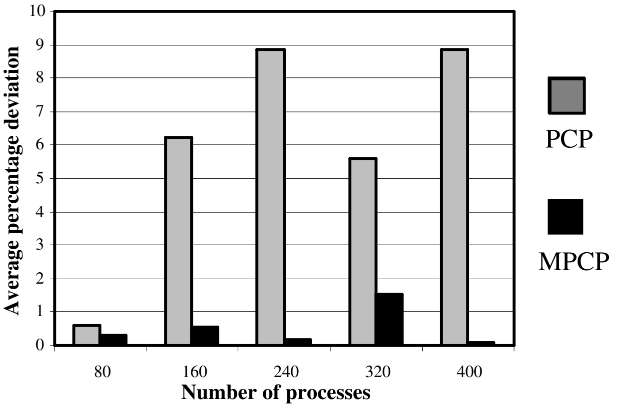
average execution time for the scheduling algorithm, as a function of the number of alternative tracks. We observe very small execution times for even large graphs and a very good scaling with the number of alternative tracks. The increase of execution time with the number of processes, for a given number of alternative tracks, is practically linear which corresponds to the theoretical complexity of list scheduling algorithms [52, 53].

One of the very important applications of our scheduling algorithm is for performance estimation during design space exploration. We have performed such an experiment as part of a project aiming to implement the operation and maintenance (OAM) functions corresponding to the F4 level of the ATM (Asynchronous Transfer Mode) protocol layer [54]. Fig. 15a shows an abstract model of the ATM switch. Through the switching network, cells are routed between the *n* input and *q* output lines. In addition, the ATM switch also performs several OAM related tasks.

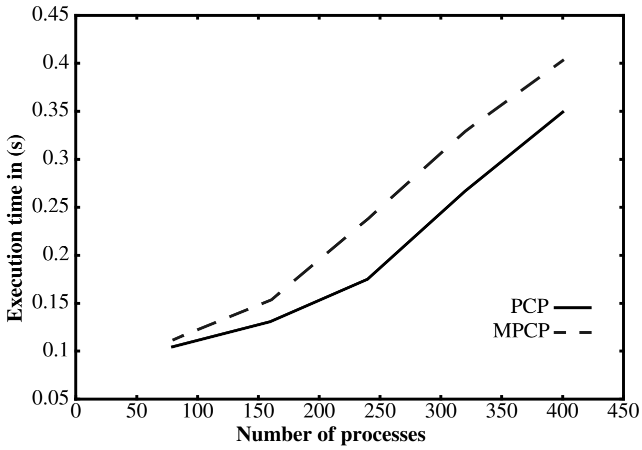
In [50] we discussed hardware/software partitioning of the OAM functions corresponding to the F4 level. We concluded that filtering of the input cells and redirecting of the OAM cells towards the OAM block have to be performed in hardware as part of the line interfaces (LI). The other functions are performed by the *OAM block* and can be implemented in software.

We have identified three independent modes in the functionality of the OAM block. Depending on the content of the input buffers (Fig. 15b), the OAM block switches between these three modes. Execution in each mode is controlled by a statically generated schedule table for the respective mode. We specified the functionality corresponding to each mode as a set of interacting VHDL processes. These specifications have then been translated to the corresponding CPGs. Table 2 shows the characteristics of the resulting CPGs. The main objective of this experiment was to estimate, using our scheduling algorithm, the worst case delays in each mode for different alternative architectures of the OAM block. Based on these estimations as well as on the particular features of the environment in which the switch will be used, an appropriate architecture can be selected and the dimensions of the buffers can be determined.

Fig. 15b shows a possible implementation architecture of the OAM block, using one processor and one memory module (1P/1M). Our experiments included also architecture models with two processors and one memory module (2P/1M), as well as structures consisting of one respectively two processors and two memory modules (1P/2M, 2P/2M). The inclusion of alternatives with two memory modules is justified by the fact that the information processed by the OAM block is organized in two main tables which potentially could be accessed in parallel. The target architectures are based on two types of processors: one (µp1) running at 80MHz and another (µp2) running at 120MHz. For each architecture, processes have been assigned to processors taking into consideration the potential parallelism of the CPGs and the amount of communication between processes. The worst case delays resulted after generation of the schedule table for each of the three modes, are given in Table 2. As expected, using a faster processor reduces the delay in each of the three modes. Introducing an additional processor, however, has no effect on the execution delay in *mode 2* which does not present any potential parallelism. In *mode 3* the delay is reduced by using two µp1 processors instead of one. For the faster, µp2 processor, however, the worst case delay cannot be improved by introducing an additional processor. Using two processors will always improve the worst case delay in *mode 1*. As for the additional memory module, only in *mode 1* the model contains memory accesses which are potentially executed in parallel. Table 2 shows that only for the architecture consisting of two µp2 processors, providing an additional memory module pays back by a reduction of the worst case delay in *mode 1*. The reason is that, with the process execution times corresponding to this processor and the 2P/1M architecture, the track containing parallel memory accesses is the one which dictates the worst case execution time. Thus, adding a second memory module results in a reduced worst case delay of the system in *mode 1*.



a) Percentage deviation of schedule lengths



b) Execution time of the scheduling algorithm

Fig. 16. Scheduling with PCP and MPCP for TTP based architectures

B. Evaluation of Scheduling with Bus Access Optimization

In this set of experiments we were interested to investigate the efficiency of our scheduling algorithm in the context of the TDMA based protocol and the potential of our optimization strategies for the bus access scheme. We considered TTP based 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 CPGs were generated for each graph dimension, thus a total of 150 CPGs 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.

The first results concern the improvement of the schedules produced by our algorithm when using the MPCP priority function instead of the one based on the general PCP priority. In order to compare the two priority functions the 150 CPGs were scheduled, considering the TTP based architectures presented above, using first PCP for priority assignment and then MPCP. We calculated the average percentage deviations of the schedule lengths produced with MPCP and PCP for each graph, from the length of the best schedule among the two. The results are depicted in Fig. 16a. The diagram shows an important improvement of the resulted schedules if the TDMA-specific priority function MPCP is used. In average the deviation with MPCP is 11.34 times smaller than with PCP. However, due to its dynamic nature, MPCP implies a slightly larger execution time, as shown in Fig. 16b.

In the following experiments we were interested to check the potential of the algorithm presented in subsection VI/C to improve 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 (*Optimize_access_1* and *Optimize_access_2*) and a so called near-optimal scheme. The near-optimal scheme was produced using a simulated annealing (SA) based algorithm for bus access optimization which is presented in [51]. 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 3 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 significantly compared to the straightforward solution. The greedy heuristic performs well for all the graph dimensions. As expected, the alternative *Optimize_access_1* (which considers all allowed slot lengths) produces slightly better results, on average, than *Optimize_access_2*. However, the execution times are much smaller for *Optimize_access_2*. 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 [51].

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.

The specification assumes that the CC will operate in an environment consisting of several nodes interconnected by a TTP channel. There are five nodes which functionally interact with the CC system: the Anti Blocking System, the Transmission Control Module, the Engine Control Module, the Electronic Throttle Module, and the Central Electronic Module. It has been decided to distribute the functionality (processes) of the CC over these five nodes.

The CPG corresponding to the CC system consists of 32 processes, and includes two alternative tracks. The maximum allowed delay is of 110 ms. For our model, the straightforward

Table 3: Evaluation of the bus access optimization algorithm

Nr. of proc.	Straightforward solution		Optimize_access_1			Optimize_access_2		
	average deviation	maximal deviation	average deviation	maximal deviation	optimization time	average deviation	maximal deviation	optimization 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

solution for bus access resulted in a schedule corresponding to a maximal delay of 114 ms (which does not meet the deadline) when PCP was used as a priority function, while using MPCP we obtained a schedule length of 109 ms. Both of the greedy heuristics for bus access optimization produced solutions so that the worst case delay was reduced to 103 ms. The near-optimal solution (produced with the SA based approach) results in a delay of 97 ms.

VIII. CONCLUSIONS

We have presented an approach to process scheduling for the synthesis of embedded systems implemented on architectures consisting of several programmable processors and application specific hardware components. The approach is based on an abstract graph representation which captures, at process level, both dataflow and the flow of control. The scheduling problem has been considered in strong interrelation with the problem of communication in distributed systems.

We first presented a general approach to process scheduling with control and data dependencies, considering a generic bus-based distributed architecture. The proposed algorithm is based on a list scheduling approach and statically generates a schedule table which contains activation times for processes and communications. The main problems which have been solved in this context are the minimization of the worst case delay and the generation of a logically and temporally deterministic table, taking into consideration communication times and the sharing of the communication support.

We have further investigated the impact of particular communication infrastructures and protocols on the overall performance and, specially, how the requirements of such an infrastructure have to be considered for process and communication scheduling. Considering a TTP-based system architecture 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 after 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.

The algorithms have been evaluated based on extensive experiments using a large number of graphs generated for experimental purpose as well as real-life examples.

There are several aspects which were omitted from the discussion in this paper. In [55] we have analyzed the optimization of a TDMA bus access scheme in the context of priority based preemptive scheduling. There we also considered the possibility of messages being split over several successive frames. We neither insist here on the relatively simple procedure for post-processing of the schedule table, during which the table can be simplified for certain situations in which identical activation times are scheduled for a given process on different columns. During post-processing the table is also split into sub-tables containing the particular activities to be performed by a certain processor.

REFERENCES

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

[2] G. De Micheli and R.K. Gupta, "Hardware/Software Co-Design", *Proceedings of the IEEE*, V85, N3, 1997, pp. 349-365.

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

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

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

[6] W. Wolf, "Hardware-Software Co-Design of Embedded Systems", *Proceedings of the IEEE*, V82, N7, 1994, pp. 967-989.

[7] J.D. Ullman, "NP-Complete Scheduling Problems", *Journal of Comput. Syst. Sci.*, 10, 1975, pp. 384-393.

[8] A. Doboli and P. Eles, "Scheduling under Control Dependencies for Heterogeneous Architectures", Proc. International Conference on Computer Design (ICCD), 1998, pp. 602-608.

[9] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli and P. Pop, "Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems". *Proc. Des. Aut. & Test in Europe*, 1998, pp. 132-138.

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

[11] J. Gong, D. D. Gajski and S. Narayan, “Software Estimation Using A Generic-Processor Model”, *Proc. Europ. Des. & Test Conf*, 1995, pp. 498-502.

[12] J. Henkel and R. Ernst, “A Path-Based Technique for Estimating

Hardware Run-time in Hw/Sw-Cosynthesis", *Proc. Int. Symp. on Syst. Synthesis*, 1995, pp. 116-121.

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

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

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

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

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

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

[19] N.C. Audsley, A. Burns, R.I. Davis, K. Tindell and A.J. Wellings, "Fixed Priority Pre-emptive Scheduling: An Historical Perspective", *Real-Time Systems*, V8, N2/3, 1995, pp. 173-198.

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

[21] T. Y. Yen and W. Wolf, "*Hardware-Software Co-Synthesis of Distributed Embedded Systems*", Kluwer Academic Publishers, 1997.

[22] C. Lee, M. Potkonjak and W. Wolf, "Synthesis of Hard Real-Time Application Specific Systems", *Design Automation for Embedded Systems*, V4, N4, 1999, pp. 215-241.

[23] B.P. Dave and N.K. Jha, "COHRA: Hardware-Software Cosynthesis of Hierarchical Heterogeneous Distributed Systems", *IEEE Trans. on CAD*, V17, N10, 1998, pp. 900-919.

[24] B.P. Dave, G. Lakshminarayana and N.J. Jha, "COSYN: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded Systems", *IEEE Trans. on VLSI Systems*, V7, N1, 1999, pp. 92-104.

[25] P. Chou and G. Borriello, "Interval Scheduling: Fine-Grained Code Scheduling for Embedded Systems", *Proc. ACM/IEEE DAC*, 1995, pp. 462-467.

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

[27] V. Mooney, T. Sakamoto and 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*, 1997, pp. 95-99.

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

[29] E.G. Coffman Jr. and R.L. Graham, "Optimal Scheduling for two Processor Systems", *Acta Informatica*, 1, 1972, pp. 200-213.

[30] P.B. Jorgensen and J. Madsen, "Critical Path Driven Cosynthesis for Heterogeneous Target Architectures", *Proc. Int. Worksh. on Hardw.-Softw. Co-design*, 1997, pp. 15-19.

[31] Y.K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: an Effective Technique for Allocating Task Graphs to Multiprocessors", *IEEE Trans. on Parallel and Distrib. Syst.*, V7, N5, 1996, pp. 506-521.

[32] M.Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems", *IEEE Trans. on Parallel and Distrib. Syst.*, V1, N3, 1990, pp. 330-343.

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

[34] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing", *IEEE Trans. on Comp.*, V33, N11, 1984, pp. 1023-1029.

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

[36] S. Prakash and A. Parker, "SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems", *Journal of Parallel and Distrib. Comp.*, V16, 1992, 338-351.

[37] K. Kuchcinski, "Embedded System Synthesis by Timing Constraint Solving", *Proc. Int. Symp. on Syst. Synth.*, 1997, pp. 50-57.

[38] A. Dasdan, D. Ramanathan and R.K. Gupta, "A Timing-Driven Design and Validation Methodology for Embedded Real-Time Systems", *ACM Trans. on Des. Aut. of Electronic Systems*, V3, N4, 1998, pp. 533-553.

[39] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst and J. Teich, "Scheduling Hardware/Software Systems Using Symbolic Techniques", *Proc. Int. Workshop on Hardware-Software Co-design*, 1999, 173-177.

[40] D. Ziegenbein, K. Richter, R. Ernst, J. Teich and L. Thiele, "Representation of Process Model Correlation for Scheduling", *Proc. Int. Conf. on CAD*, 1998, pp. 54-61.

[41] P. H. Chou, R.B. Ortega and G. Borriello, "The Chinook Hardware/Software Co-Synthesis System", *Proc. Int. Symp. on Syst. Synthesis*, 1995, pp. 22-27.

[42] J.M. Daveau, T. Ben Ismail and A.A. Jerraya, "Synthesis of System-Level Communication by an Allocation-Based Approach", *Proc. Int. Symp. on Syst. Synthesis*, 1995, pp. 150-155.

[43] P.V. Knudsen and J. Madsen, "Integrating Communication Protocol Selection with Hardware/Software Codesign", *IEEE Trans. on CAD*, V18, N8, 1999, pp. 1077-1095.

[44] S. Narayan and D.D. Gajski, "Synthesis of System-Level Bus Interfaces", *Proc. Europ. Des. & Test Conf*, 1994, pp. 395-399.

[45] R.B. Ortega and G. Borriello, "Communication Synthesis for Distributed Embedded Systems", *Proc. Int. Conf. on CAD*, 1998, pp. 437-444.

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

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

[48] H. Kopetz and G. Grünsteidl, "TTP-A Protocol for Fault-Tolerant Real-Time Systems", *IEEE Computer*, Vol27 N1, 1997, pp. 14-23.

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

[50] P. Eles, Z. Peng, K. Kuchcinski and A. Doboli, "System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search", *Des. Automation for Embedded Syst.*, V2, N1, 1997, pp. 5-32.

[51] P. Pop, P. Eles and Z. Peng, "Scheduling with Optimized Communication for Time-Triggered Embedded Systems", *Proc. Int. Worksh. on Hardware-Software Co-design*, 1999, pp. 178-182.

[52] G. De Micheli, "*Synthesis and Optimization of Digital Circuits*", McGrow-Hill, 1994.

[53] S.H. Gerez, "*Algorithms for VLSI Design Automation*", John Wiley&Sons, 1999.

[54] T.M. Chen and S.S. Liu, *ATM Switching Systems*, Artech House Books, 1995.

[55] P. Pop, P. Eles and Z. Peng, "Bus Access Optimization for Distributed Embedded Systems Based on Schedulability Analysis", *Proc. Des. Aut. & Test in Europe*, 2000.