An Improved Scheduling Technique for Time-Triggered Embedded Systems

Paul Pop, Petru Eles, and Zebo Peng
Dept. of Computer and Information Science
Linköping University
Sweden

Abstract

In this paper we present an improved scheduling technique for the synthesis of time-triggered embedded systems. Our system model captures both the flow of data and that of control. We have considered communication of data and conditions for a time-triggered protocol implementation that supports clock synchronization and mode changes. We have improved the quality of the schedules by introducing a new priority function that takes into consideration the communication protocol. Communication has been optimized through packaging messages into slots with a properly selected order and lengths. Several experiments and a real-life example demonstrate the efficiency of our approach.

1. Introduction

Depending on the particular application, an embedded system has certain requirements of performance, cost, dependability, size, etc. For hard real-time applications the timing requirements are extremely important. Thus, in order to function correctly, an embedded system implementing such an application has to meet its deadlines. One of the application areas for such systems is that of safety-critical automotive applications (e.g. drive-by-wire, brake-by-wire)[17].

Due to the complexity of the systems, co-synthesis environments are developed to assist the designer in finding the most cost effective solution that, at the same time, meets the design requirements. In this paper we concentrate on an improved scheduling technique for safety-critical systems consisting of multiple programmable processors and ASICs interconnected by a communication channel. In this context scheduling has a decisive influence on the correct behaviour of the system with respect to its timing requirements.

Process scheduling for performance estimation and synthesis of embedded systems has been intensively researched in the last years. The approaches differ in the scheduling strategy adopted, system architectures considered, handling of the communication, and process interaction aspects.

Preemptive scheduling with static priorities using rate monotonic scheduling analysis is performed in [16], static non-preemptive scheduling is done in [7, 8, 15], while in [12] scheduling is considered together with allocation and partitioning. In order to guarantee the timing requirements of safety-critical applications, non-preemptive static scheduling is preferred over the other scheduling strategies [9].

Several approaches consider spimple architectures consisting of a programmable processor and an ASIC.

For process interaction (if considered) the mentioned approaches are based on a dataflow model representation. Communication aspects have been treated in a very simplified way during process scheduling. One typical solution is to consider communication tasks as processes with a given execution time (depending on the amount of information transferred) and to schedule them as any other process [7, 12, 14], without considering issues like communication protocol, packaging of messages, clock synchronization, etc. These aspects are, however, essential in the context of safety-critical distributed applications and one of the objectives of this paper is to develop a strategy which takes them into consideration for process scheduling.

In our approach an embedded system is viewed as a set of interacting processes mapped on an architecture consisting of several programmable processors and ASICs interconnected by a communication channel. Process interaction captures both the flow of data and that of control, since some processes can be activated depending on conditions computed by previously executed processes. We consider a non-preemptive environment in which the execution of processes is triggered at certain points in time, and we generate a schedule table and a worst case delay that is guaranteed under any conditions. We also address the clock synchronization aspect in distributed embedded systems, in order to guarantee the correct behaviour of the system with respect to its timing constraints.

The way interprocess communication is modeled has a crucial impact on the accuracy of the timing. Thus, in our approach we consider the time-triggered protocol (TTP) [10] as the communication protocol on the communication channel. TTP is a communication protocol well suited for safety-critical distributed real-time control systems and it can be successfully applied to automotive applications.

The paper is divided into 9 sections. Section 2 presents our graph-based abstract system representation and section 3 introduces the list scheduling algorithm for conditional process graphs which is the starting point of our improved scheduling technique. The architectures considered for system implementation are presented in section 4. Section 5 formulates the problem and section 6 and 7 present the improved scheduling techniques proposed. The algorithms are evaluated in section 8, and section 9 presents our conclusions.

2. Conditional Process Graph

As an abstract model for system representation we use a directed, acyclic polar graph with conditional edges (Fig. 1) [3].

Each node in this graph represents a process which is assigned to a processing element. The graph is polar, which means that there are two nodes, called *source* and *sink*, that conventionally represent the first and the last task respectively. 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.

Each process P_i is assigned to a processor or bus and is characterized by an execution time t_{P_i} (denoted by the values on the right side of the nodes in Fig. 1). In the process graph depicted in Fig. 1, P_0 and P_{15} are the source and sink nodes respectively. The rest of the nodes denoted P_1, P_2, .., P_{14} are "ordinary" processes specified by the designer. They are assigned to one of the two programmable processors or to the hardware component (ASIC). The rest of the nodes are so called *communication processes* and they are represented in Fig. 1 as filled circles. They are introduced during the generation of the system representation for each connection which links processes mapped to different processors.

An edge from process P_i to P_j indicates that the output of P_i is the input of P_j. Unlike a simple edge, a conditional edge (depicted with thicker lines in Fig. 1) has an associated condition. Transmission of a message on a conditional edge will take place only if the associated condition is satisfied and not, like on simple edges, for each activation of the input process P_i.

We call a node with conditional edges at its output a *disjunction node* (and the corresponding process a *disjunction process*). Alternative paths starting from a disjunction node, which correspond to a certain condition, are disjoint and they meet in a so called *conjunction node* (with the corresponding process called *conjunction process*). In Fig. 1, conjunction and disjunction nodes are depicted with triangles. The alternative paths starting from disjunction node P_7, corresponding to the different values of condition D, meet in the conjunction node P_{12}.

According to our model, we assume that a process, which is not a conjunction process, can be activated after all its inputs have arrived. A conjunction process can be activated after messages coming on one of the alternative paths have arrived. Once activated, a process can not be preempted by other processes. All processes issue their outputs when they terminate.

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 resource.

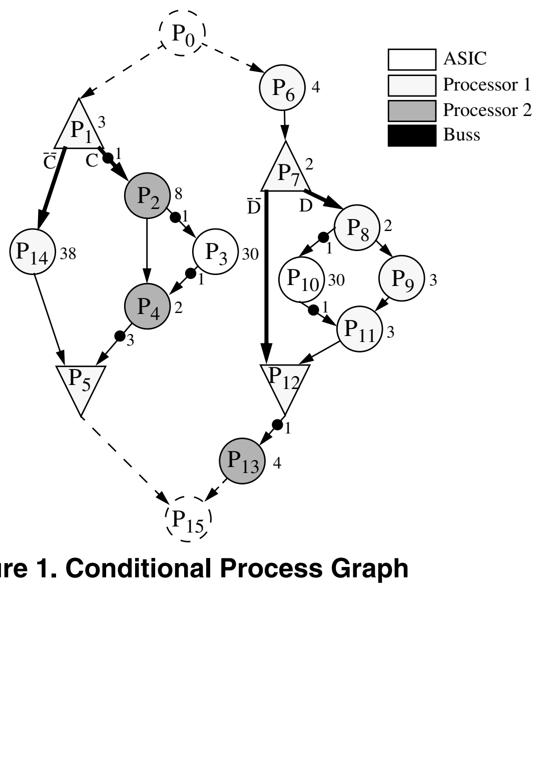

Figure 1. Conditional Process Graph

3. Scheduling of Conditional Process Graphs

In [2, 3] our goal was to derive a worst case delay by which the system completes execution, so that this delay is as small as possible, and to generate the schedule which guarantees this delay. For this, we have considered a simplified communication model in which the execution time of the communication processes was a function only of the amount of data exchanged by the processes engaged in the communication. Also, we have treated the communication processes exactly as ordinary processes during the scheduling of the conditional process graph. Thus, we have modeled the buss similar to a programmable processor that can "execute" one communication at a time as soon as the communication becomes "ready".

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

Under these assumptions, Table 1 presents a possible schedule (produced by the algorithm in Fig. 2) for the conditional process graph in Fig. 1. In Table 1 there is one row for each "ordinary" or communication process, which contains activation times corresponding to different values of conditions. Each column in the table is headed by a logical expression constructed as a conjunction of condition values. Activation times in a given column represent starting times of the processes when the respective expression is true.

According to the schedule in Table 1 process P_1 is activated unconditionally at the time 0, given in the first column of the table. However, activation of some processes at a certain execution depends on the values of the conditions, which are unpredictable. For example, process P_{11} has to be activated at t=44 if $C \wedge D$ is true and t=52 if $\overline{C} \wedge D$ is true. At a certain moment during the execution, when the values of some conditions are already known, they have to be used in order to take the best possible decisions on when and which process to activate. Therefore, after the termination of a process that produces a condition, the value of the condition is broadcasted from the corresponding processor to all other processors. This broadcast is scheduled as soon as possible on the communication channel, and is considered together with the scheduling of the messages.

Table 1: Schedule Table for Graph in Figure 1

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

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

1. No process will be activated if, for a given execution, the conditions required for its activation are not fulfilled.

2. Activation times have to be uniquely determined by the conditions.

3. Activation of a process P_i at a certain time t has to depend only on condition values which are determined at the respective moment t and are known to the processing element which executes P_i.

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

Figure 2. List Scheduling Based Algorithm

As the starting point for our improved scheduling technique that is tailored for time-triggered embedded systems we consider the list scheduling based algorithm in Fig. 2 [2].

List scheduling heuristics [4] are based on priority lists from which processes are extracted in order to be scheduled at certain moments. In our algorithm, presented in Fig. 2, we have such a list, ReadyList, that contains the processes which are eligible to be activated on the respective processor at time CurrentTime. These are processes which have not been yet scheduled but have all predecessors already scheduled and terminated.

The ListScheduling function is recursive and calls itself for each conjunction node in order to separately schedule the nodes in the true branch, and those in the false branch respectively. Thus, the alternative paths are not activated simultaneously and resource sharing is correctly achieved [2].

An essential component of a list scheduling heuristic is the priority function used to solve conflicts between ready processes. The highest priority process will be extracted by function GetReadyProcess from the ReadyList in order to be scheduled. The priority function will be further discussed in section 6.

4. Time-Triggered Systems

We consider architectures consisting of nodes connected by a broadcast communication channel. Every node consists of a TTP controller [10], 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.

On top of each node runs a real-time kernel that has a schedule table which contains all the information needed to take decisions on activation of processes and transmission of messages, based on the values of conditions.

Communication between nodes is based on the TTP [9]. TTP was designed for distributed real-time applications that require predictability and reliability (e.g, drive-by-wire). It integrates all the services necessary for fault-tolerant real-time systems. The TTP services of importance to our problem are: message transport with acknowledgment and predictable low latency, clock synchronization within the microsecond range and rapid mode changes.

The communication channel is a broadcast channel, so a message sent by a node is received by all the other nodes. The bus access scheme is time-division multiple-access (TDMA) (Fig. 3). Each node N_i can transmit only during a predetermined time interval, the so called TDMA slot S_i. In such a slot, a node can send several messages packaged in a frame. We consider that a slot S_i is at least large enough to accommodate the largest message generated by any process assigned to node N_i, so the messages do not have to be split in order to be sent. A sequence of slots corresponding to all the nodes in the architecture is called a TDMA round. A node can have only one slot in a TDMA round. Several TDMA rounds can be combined together in a cycle that is repeated periodically. The sequence and length of the slots are the same for all the TDMA rounds. However, the length and contents of the frames may differ.

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

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.

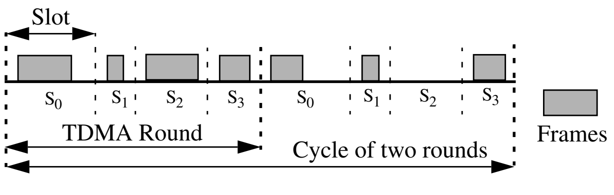

Figure 3. Buss Access Scheme

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

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

5. Scheduling for Time-Triggered Systems

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

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

Considering the concrete definition of our problem, the communication time is no longer dependent only on the length of the message, as assumed in our previous approaches [2, 3]. Thus, if the message is sent between two processes mapped onto different nodes, the message has to be scheduled according to the TTP protocol. Several messages can be packaged together in the data field of a frame. The number of messages that can be packaged depends on the slot length corresponding to the node. The effective time spent by a message m_i *on the bus* is $t_{m_i} = b_{S_i}/T$ where b_{Si} is the length of the slot S_i and T is the transmission speed of the channel. Therefore, 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. 4 by means of a simple example.

In Fig. 4 d) we have a process graph consisting of four processes P_1 to P_4 and four messages m_1 to m_4. The architecture consists of two nodes interconnected by a TTP channel. The first node, N_0, transmits on the slot S_0 of the TDMA round and the second node, N_1, transmits on the slot S_1. Processes P_1 and P_4 are mapped on node N_0, while processes P_2 and P_3 are mapped on node N_1. With the TDMA configuration in Fig. 4 a), where the slot S_1 is scheduled first and slot S_0 is second, we have a resulting schedule length of 24 ms. However, if we swap the two slots inside the TDMA round without changing their lengths, we can improve the

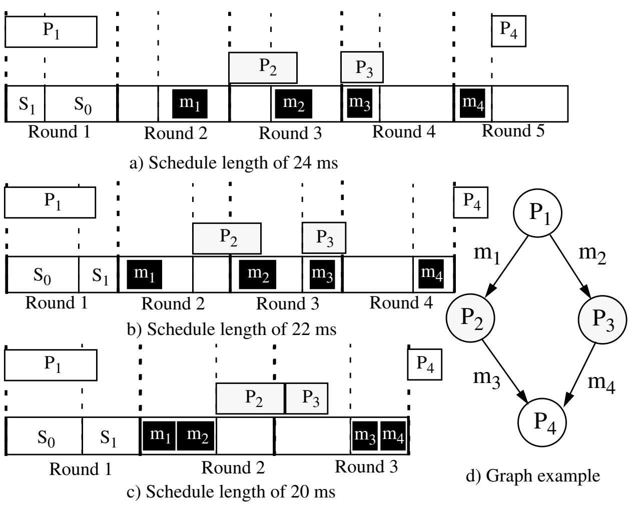
P1
P2
P3
P4
S1
S0
m1
m2
m3
m4
Round 1
Round 2
Round 3
Round 4
Round 5
a) Schedule length of 24 ms
b) Schedule length of 22 ms
c) Schedule length of 20 ms
d) Graph example

Figure 4. Scheduling Example

schedule by 2 ms, as seen on Fig. 4 b). Further more, if we have the TDMA configuration in Fig. 4 c) where slot S_0 is first, slot S_1 is second and we increase the slot lengths so that the slots can accommodate both of the messages generated on the same node, we obtain a schedule length of 20 ms which is optimal. However, increasing the length of slots is not necessarily improving a schedule, as it delays the communication of messages generated by other nodes.

6. Synthesis of local schedule tables and MEDL

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

We propose several extensions to our scheduling algorithm described in section 3. The extension considers a realistic communication and execution infrastructure, and includes aspects of the communication protocol in the optimization process.

Thus, if the ready process P_i returned by the GetReadyProcess function in Fig. 2 is a message, then instead of scheduling P_i like an "ordinary" process we will schedule P_i according to the TTP protocol using the

```
ScheduleMessage
    slot = slot of the node sending the message
    round = CurrentTme / RoundLength
    if CurrentTime - round * RoundLength > start of slot in round then
        round = next round
    end if
    if not message fits in the slot of round then
        round = next round
    end if
    Insert (message, round, slot, ScheduleTable)
end ScheduleMessage
```

Figure 5. The ScheduleMessage Function

ScheduleMessage function in Fig. 5.

Depending on the CurrentTime and the given TDMA configuration, the function determines the first TDMA round where the message can be scheduled in the slot corresponding to the sender node. If the slot is full in the first selected round because of previously scheduled messages, the message has to wait for the next round.

For the scheduling algorithm outlined in section 3 we initially used the Partial Critical Path (PCP) priority function [2, 4]. PCP uses as a priority criterion the length of that part of the critical path corresponding to a process P_i which starts with the first successor of P_i that is assigned to a processor different from the processor running P_i. The PCP priority function is statically computed once at the beginning of the scheduling procedure.

However, considering the concrete definition of our problem, significant improvements of the resulting schedule can be obtained by including knowledge of the buss access scheme into the priority function. This new priority function will be used by the GetReadyProcess (Fig. 2) in order to decide which process to select from the list of ready process.

Let us consider the process graph in Fig. 6, 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. Process P_0 is a dummy process, inserted in order for the graph to be polar. 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_0, while processes P_3 and P_4 are mapped on node N_1. The first node, N_0, transmits on the slot S_0 of the TDMA round and the second node, N_1, transmits on the slot S_1. Slot S_0 has a "length" of 10 ms while slot S_1 has a length of 8 ms.

PCP assigns a higher priority to P_1 because it has a partial critical path of 14 ms, starting from P_3, longer than the partial critical path of P_2 which is 10 ms and starts from m. This results in a schedule length of 40 ms depicted in Fig. 6 a). On the other hand, if we schedule P_2 first, the resulting schedule, depicted in Fig. 6 b), is only of 36 ms.

This apparent anomaly is due to the fact that the PCP

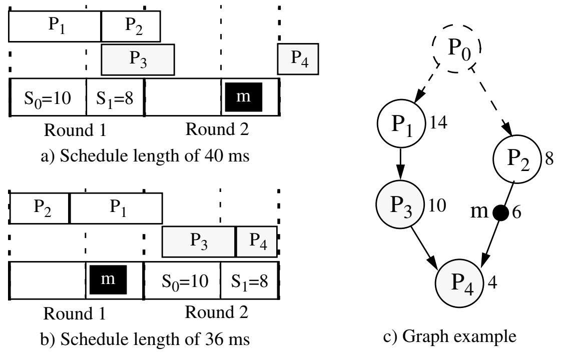

Figure 6. Priority Function Example

```
Lambda(lambda, CurrentProcess)
  if CurrentProcess is a message then
    slot = slot of node sending CurrentProcess
    round = lambda / RoundLength
    if lambda - RoundLength * round > start of slot in round
      round = next round;
    end if
    while slot in round is full
      round = next round
    end while
    lambda = round * RoundLength +
      start of slot in round + length of slot
  else
    lambda = lambda + WCET of CurrentProcess
  end if
  if lambda > MaxLambda
    MaxLambda = lambda
  end if
  for each successor of CurrentProcess
    Lambda(lambda, successor)
  end for
  return MaxLambda
end Lambda
```

Figure 7. The Lambda Function

function is not based on the realistic communication model, but considers communication as an "ordinary" process which is allocated to the bus.

However, if we consider the particular TDMA configuration in Fig. 6, we note that P_4 can only be started at the end of the slot S_1 in which m is scheduled. Thus, if we account for the TDMA configuration in the partial critical path of P_2 we get a partial critical path of $S_0+S_1=18$ ms. Based on this value, process P_2 should be selected first for scheduling resulting in a schedule of 36 ms.

We introduce a new priority function, namely PCP2, that includes knowledge about the particular TDMA configuration:

$$\lambda_{Pi} = \max_{k} \sum_{P_j \in \pi_{ik}} t_{Pj}$$

where π_{ik} is the kth path from the first successor of P_i, that is assigned to a processor different from the processor running P_i, to the sink node. In the previous equation t_{Pi} is:

$$t_{Pi} = \begin{cases} \text{WCET of } P_i\text{, if } P_i \text{ is not a message} \\ \text{end time of the slot in which } P_i \text{ is scheduled, if } P_i \text{ is a message} \end{cases}$$

The end time of the slot in which P_i is scheduled depends on the particular ordering of the slots and their lengths, on the current time, which determines the TDMA round in which P_i is scheduled, and on the scheduling decisions taken before scheduling P_i (for example, a slot might be full because of previously scheduled messages, thus P_i has to be scheduled in the next round).

Thus, priority function PCP2 has to be dynamically determined during the scheduling algorithm for each ready process, every time the GetReadyProcess function is activated in order to select a process from the ReadyList. The computation of PCP2 is performed inside the GetReadyProcess function and involves a partial traversal of the graph, as presented in Fig. 7.

As the experimental evaluation in section 8 shows, PCP2 performs better that PCP resulting in shorter schedule lengths. The cost is a slight increase in the execution time of the ListScheduling algorithm, since PCP2 is determined dynamically during the scheduling process.

7. Synthesis of TDMA Buss Access

In order to get an optimized schedule it is not enough to consider the communication protocol during the scheduling of the conditional process graph. As the example in Fig. 4 shows, the ordering of the slots and the slot lengths have a big impact on the schedule quality.

In this section, we propose a heuristic to determine an ordering of the slots and the slot lengths so that the execution delay of the application is as small as possible.

A short description of the algorithm is shown in Fig. 8. The algorithm starts with the first slot of the TDMA round and tries to find the node which, by transmitting in this slot, will produce the smallest delay on the system execution time. Once a node has been selected to transmit in the first slot, the algorithm continues in the same manner with the next slots.

The selection of a node for a certain slot is done by trying out all the nodes not yet allocated to a slot. Thus, for a candidate node, the schedule length is calculated considering the TDMA round given so far using the ListScheduling algorithm as described in sections 3 and 6.

All the possible slot lengths are considered for a slot bound to a given candidate node. In order to find the slot length that will minimize the delay, the algorithm starts with the minimum slot length determined by the largest message to be sent from the candidate node. Then, 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).

Since our heuristic is based on a greedy approach, it does not guarantee the finding of the optimal TDMA configura-

```
SythesizeTDMA
  for each slot
    for each node not yet allocated to a slot
      Bind(node, slot, minimum possible length for this slot)
      for every possible slot length
        do ListScheduling in the context of current TDMA round
        remember the best schedule for this slot
      end for
    end for
    Bind(node, slot and length corresponding to the best schedule)
  end for
  return TDMA configuration
end
```

Figure 8. Heuristic for Synthesis of TDMA

tion. However, as the experimental results in section 8 show, the heuristic produces good results in a very short time.

In [12] two other heuristics which synthesize the TDMA buss access were presented.

8. Experimental Evaluation

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

The first result concerns the quality of the schedules produced by the list scheduling based algorithm using PCP and PCP2 priority functions. In order to compare the two priority functions we have calculated the average percentage deviations of the schedule length produced with PCP and PCP2 from the length of the best schedule between the two. The results are depicted in Fig. 9. In average the deviation with PCP2 is 11.34 times smaller than with PCP. However, due to its dynamic nature, PCP2 has in average a bigger execution time than PCP. The average execution times for the ListScheduling function using PCP and PCP2 are depicted in Fig. 10 and are under half a second for graphs with 400 processes.

The next result concerns the quality of the heuristics for ordering slots and slot length selection. We compare the schedule length produced by the greedy algorithm SynthesizeTDMA (Fig. 7) with results obtained using a simulated annealing (SA) based algorithm which is presented in [12]. The SA algorithm has been tuned to produce (near)optimal schedules at the cost of relatively large execution times. Table 2 presents the average and maximum percentage deviation of the schedule lengths produced by the SynthesizeTDMA algorithm from the (near)optimal schedules. The table also shows the average execution times in seconds.

Table 2: Percentage Deviation and Exec. Times

No. of processes	Naive Designer		SynthesizeTDMA		
	average	max.	average	max.	time
80	3.16%	21%	0.02%	0.5%	0.25s
160	14.4%	53.4%	2.5%	9.5%	2.07s
240	37.6%	110%	7.4%	24.8%	0.46s
320	51.5%	135%	8.5%	31.9%	34.69s
400	48%	135%	10.5%	32.9%	56.04s

Together with the greedy heuristic, a naive designer's approach is presented. The naive designer performs scheduling without trying to optimize the access to the communication channel, namely the TDMA round and the slot lengths. For the naive designer's approach we considered a TDMA round consisting of a straightforward ascending order of allocation of the nodes to the TDMA slots; the slot lengths were selected to accommodate the largest message sent by the respective node.

Table 2 shows that when considering the optimization of the access to the communication channel, the results improve dramatically compared to the naive designer's approach. The greedy heuristic performs very well for all the graph dimensions in relatively short time.

Finally, we considered a real-life example implementing a vehicle cruise controller. The conditional process graph that models the cruise controller has 32 processes, and it was mapped on an architecture consisting of 4 nodes, namely: Anti Blocking System, Transmission Control Module, Engine Control Module and Electronic Throttle Module. We considered one mode of operation with a dead-

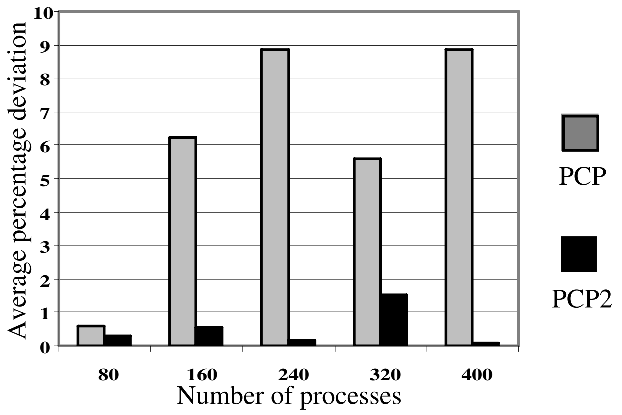

Figure 9. Quality of Schedules with PCP and PCP2

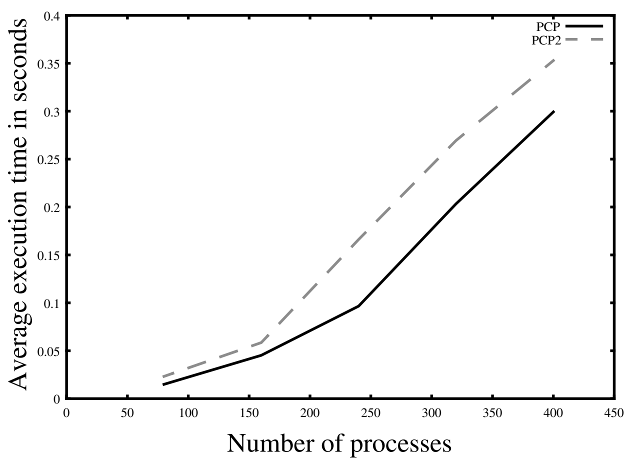

Figure 10. Average Exec. Times of PCP and PCP2

line of 110 ms. The naive designer's approach resulted in a schedule corresponding to a delay of 114 ms (which does not meet the deadline) using PCP as a priority function, while using PCP2 we obtained 109 ms. The greedy heuristic produced a delay of 103 ms on the worst case execution time of the system for both PCP and PCP2.

9. Conclusions

We have presented an improved scheduling technique for synthesis of safety-critical distributed embedded systems. Our system model captures both the flow of data and that of control. We have considered communication of data and conditions for a time-triggered protocol implementation that supports clock synchronization and mode changes. We have improved the quality of the schedule by introducing a new priority function that takes into consideration the communication protocol. Also, we have improved communications through packaging of messages into slots with a properly selected order and lengths.

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

Acknowledgments

The authors are grateful to Jakob Axelsson from Volvo Technological Development for his support and for providing the automotive electronics case study. The research has been partly supported by the Swedish Foundation for Strategic Research.

References

[1] Chou, P. , Boriello, G. Interval Scheduling: Fine-Grained Code Scheduling for Embedded Systems. Proc. DAC, 1995, 462-467.

[2] Doboli, A., Eles, P. Scheduling under Control Dependencies for Heterogeneous Architectures. International Conference on Computer Design, 1998

[3] Eles, P., Kuchcinski, K., Peng, Z., Doboli, A., Pop, P. Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems. Proc. Des. Aut. & Test in Europe, 1998.

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

[5] Gupta, R. K., De Micheli, G. A Co-Synthesis Approach to Embedded System Design Automation. Design Automation for Embedded Systems, V1, 1/2, 1996, 69-120.

[6] Jorgensen, P.B., Madsen, J. Critical Path Driven Cosynthesis for Heterogeneous Target Architectures. Proc. Int. Workshop on Hardware-Software Co-design, 1997, 15-19.

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

[8] Kopetz, H. Real-Time Systems-Design Principles for Distributed Embedded Applications. Kluwer Academic Publ., 1997

[9] Kopetz, H., Grünsteidl, G. TTP-A Protocol for Fault-Tolerant Real-Time Systems. IEEE Computer, Vol: 27/1, 14-23.

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

[11] Kuchcinski, K. Embedded System Synthesis by Timing Constraint Solving. Proc. Int. Symp. on System Synthesis, 1997.

[12] Pop, P., Eles, P., Peng, Z. Scheduling with Optimized Communication for Time-Triggered Embedded Systems. Proc. Int. Workshop on Hardware-Software Co-design, 1999.

[13] Prakash, S., Parker, A. SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems. Journal of Parallel and Distributed Computing, V16, 1992, 338-351.

[14] Wu, M.Y., Gajski, D.D. Hypertool: A Programming Aid for Message-Passing Systems. IEEE Trans. on Parallel and Distributed Systems, V. 1, N. 3, 1990, 330-343.

[15] Yen, T. Y., Wolf, W. Hardware-Software Co-Synthesis of Distributed Embedded Systems. Kluwer Academic Publisher, 1997.

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