Scheduling with Optimized Communication for Time-Triggered Embedded Systems

Paul Pop, Petru Eles, Zebo Peng
Dept. of Computer and Information Science, Linköping University, S-58183 Linköping, Sweden
{paupo, petel, zebpe}@ida.liu.se

ABSTRACT

We present an approach to process scheduling for synthesis of safety-critical distributed embedded systems. Our system model captures both the flow of data and that of control. The communication model is based on a time-triggered protocol. We take into consideration overheads due to communication and the execution environment. Communications have been optimized through packaging of messages into slots with a properly selected order and lengths. Several experiments demonstrate the efficiency of the approach.

1. INTRODUCTION

In this paper we concentrate on process scheduling for time-triggered systems consisting of multiple programmable processors and ASICs interconnected by a communication channel. For such systems, 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. Preemptive scheduling with static priorities using rate monotonic analysis is performed in [13]. In [11] the problem is formulated using mixed integer linear programming while the solution proposed in [10] is based on constraint logic programming. Static non-preemptive scheduling of a set of processes on a multiprocessor architecture has been discussed in [5, 6, 12]. Several approaches consider architectures consisting of a single programmable processor and an ASIC. Under such circumstances deriving a static schedule for the software component practically means the linearization of the dataflow graph [1, 4].

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 [5, 10, 11], 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 is not only in terms of dataflow but also captures the flow of control, since some processes can be activated depending on conditions computed by previously executed processes. We consider a non-preemptive execution environment in which the activation of processes and communications is triggered at certain points in time, and we generate a schedule table and derive a worst case delay which is guaranteed under any condition. Such a scheduling policy is well suited to a large class of safety-critical applications [7].

The scheduling strategy is based on a realistic communication model and execution environment. We take into consideration the overheads due to communications and to the execution environment and consider during the scheduling process the requirements of the communication protocol. Moreover, our scheduling algorithm performs an optimization of parameters defining the communication protocol which is essential for the reduction of the execution delay.

Our system architecture is built on a communication model which is based on the time-triggered protocol (TTP) [8]. TTP is well suited for safety critical distributed real-time control systems and represents one of the emerging standards for several application areas like, for example, automotive electronics [7, 14].

The paper is divided into 7 sections. In section 2 we present our graph-based abstract system representation. The architectures considered for system implementation are presented in section 3. Section 4 formulates the problem and section 5 presents the scheduling strategy proposed. The algorithms, one based on a greedy approach and the other on simulated annealing, are evaluated in section 6, and section 7 presents our conclusions.

2. CONDITIONAL PROCESS GRAPH

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

Each node in this graph represents a process which is assigned to a processing element. 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 Figure 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. A process can be activated only after all its inputs have arrived, and issues its outputs when it terminates. However, a conjunction process (where the alternative paths corresponding to different values of a condition meet, e.g., P_{10}) can be activated after messages coming on one of the alternative paths have arrived. Once activated, the process can not be preempted by other processes.

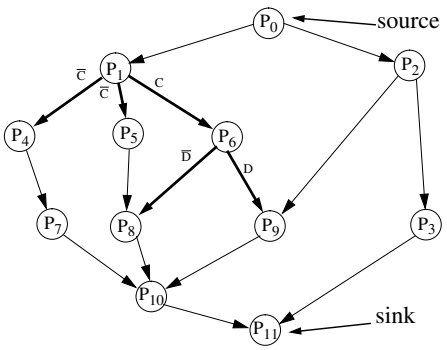

Figure 1. Conditional Process Graph

Activation of processes at a certain execution depends on the values of the conditions, which are unpredictable. 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. Thus, in the following of the paper we will not treat communication of the conditions explicitly when scheduling messages.

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.

3. SYSTEM ARCHITECTURE

3.1 Hardware architecture

We consider architectures consisting of nodes connected by a broadcast communication channel (Figure 2). Every node consists of a TTP controller [9], 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 [8]. 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) (Figure 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 (see Figure 4).

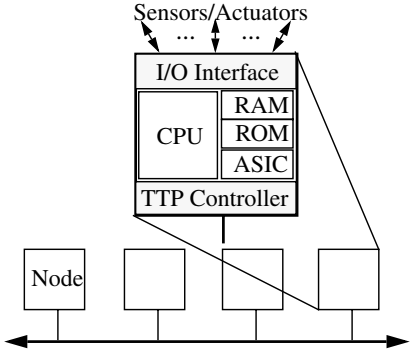

Figure 2. System Architecture

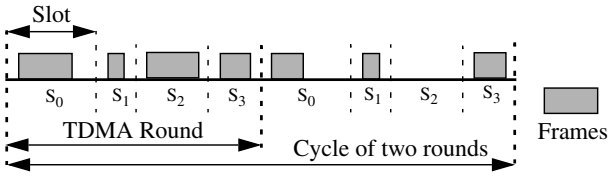

Figure 3. Buss Access Scheme

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.

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.

3.2 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.

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

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

We also have to take into account the overheads for process activation and message passing. For process activation we consider an overhead δ_{PA}. The message passing mechanism is illustrated in Figure 4, where we have three processes, P_1 to P_3. P_1 and P_2 are mapped to node N_0 that transmits in slot S_0, and P_3 is mapped to node N_1 that transmits in slot S_1. Message m_1 is transmitted

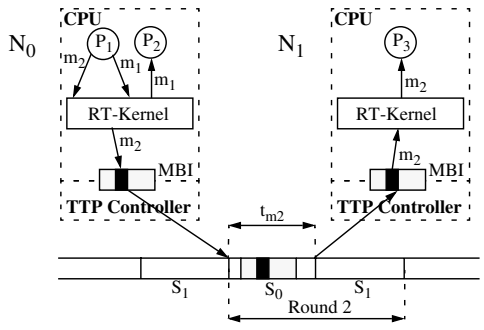

Figure 4. Message Passing Mechanism

between P_1 and P_2 that are on the same node, while message m_2 is transmitted from P_1 to P_3 between the two nodes. We consider that each process has its own memory locations for the messages it sends or receives and that the addresses of the memory locations are known to the kernel through the schedule table.

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

Message m_2 has to be sent from node N_0 to node N_1. At a certain time, known from the schedule table, the kernel transfers m_2 to the TTP controller by packaging m_2 into a frame in the MBI. The WCAO of this function is δ_{KS}. Later on, the TTP controller knows from its MEDL when it has to take the frame from the MBI, in order to broadcast it on the bus. In our example the timing information in the schedule table of the kernel and the MEDL is determined in such a way that the broadcasting of the frame is done in the slot S_0 of Round 2. The TTP controller of node N_1 knows from its MEDL that it has to read a frame from slot S_0 of Round 2 and to transfer it into the MBI. The kernel in node N_1 will read the message m_2 from the MBI, with a corresponding WCAO of δ_{KR}. When P_3 will be activated based on the local schedule table of node N_1, it will already have m_2 in its right memory location.

4. PROBLEM FORMULATION

As an input 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 section 3.1. The overhead U_t of each kernel and the WCAO of each system call are known. Each process of the process graph is mapped on a CPU or an ASIC of a node.

We are interested to derive a 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.

The worst case execution delay of a process is estimated taking into account the overhead of the timer interrupt, the WCAO of the process activation, and the WCAO of the message passing mechanism. Therefore, the worst case execution delay of a process P_i will be:

$$T_{P_i} = (\delta_{PA} + t_{P_i} + \theta_{C_1} + \theta_{C_2}) \cdot (1 + U_t)$$

where t_{Pi} is the worst case execution time of the code of process P_i, θ_{C1} is the overhead for communication from P_i to processes on the same node, and θ_{C2} is the overhead for communication between processes on different nodes:

$$\theta_{C_1} = \sum_{i=1}^{N_{out}^{loc}(P_i)} \delta_{S_i} \qquad \theta_{C_2} = \sum_{i=1}^{N_{out}^{rem}(P_i)} \delta_{KS_i} + \sum_{i=1}^{N_{in}^{rem}(P_i)} \delta_{KR_i}.$$

In the previous equations, $N_{out}^{loc}(P_i)$ is the number of messages to be sent by the process P_i to other processes on the same node. $N_{out}^{rem}(P_i)$ is the number of messages transferred to the MBI, and $N_{in}^{rem}(P_i)$ is the number of messages transferred from the MBI by the kernel, during the execution of process P_i. It has to be noticed that θ_{C1} refers to the overhead caused by sending the $N_{out}^{loc}(P_i)$ messages generated by process P_i which are directed to other processes on the *same node*. However, θ_{C2} considers the overhead due to the *remote communications* which not necessarily originate from P_i, but are scheduled to be performed by the kernel during the period P_i is active. Thus, for example, transferring the message m_2 (which is generated by P_1) in Figure 4, to the MBI can be scheduled during the time P_2 is active. This can be due to the fact that no place was available in the MBI before that moment.

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 above. Thus, from the scheduling point of view, communication of the message is instant. 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 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. In Figure 4, 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 Figure 5 by means of a simple example.

In Figure 5 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 Figure 5 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

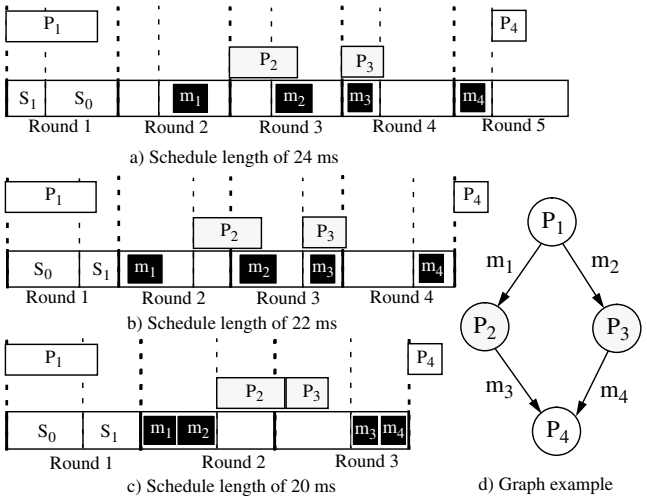

Figure 5. Scheduling Example

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

TDMA round without changing their lengths, we can improve the schedule by 2 ms, as seen on Figure 5 b). Further more, if we have the TDMA configuration in Figure 5 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.

The sequence and length of the slots in a TDMA round are determined by our scheduling strategy with the goal to reduce the delay on the execution time of the system.

5. THE SCHEDULING STRATEGY

The problem of conditional process graph scheduling has been addressed by us in [2, 3] without considering a specific communication protocol and execution environment. We do not re-discuss here those algorithms which are part of the function **schedule** mentioned below. That work has been largely extended by considering a realistic communication and execution infrastructure, and by including aspects of the communication protocol in the optimization process. For this reasons, the worst case execution delays T_{Pi} of the processes are computed according to the formula given in section 4. A major extension concerns the scheduling of the messages on the TTP bus, considering *a given* order of slots in the TDMA round and *given* slot lengths.

The **schedule_message** function in Figure 6 is called in order to schedule the highest priority message that is ready for transmission at the current_time on the TTP bus, and also offers important feedback information to the optimization heuristic concerning the selection of slot lengths.

Depending on the current_time 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. Our optimization heuristics have to consider enlarging this slot in the hope to reduce the delay on the execution time caused by scheduling of the message in the next round. A list of recommended slot lengths for a node is kept for this purpose, and updated by the **schedule_message** function. Function **schedule_message** is called by the function **schedule**, which generates the schedule and corresponding tables based on the given slot order and slot lengths.

In order to get an optimized schedule we have to determine an ordering of the slots and the slot lengths so that the execution delay is as small as possible. We first present two variants of an algorithm based on a greedy approach. A short description of the algorithm is shown in Figure 7.

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 was 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,

```
schedule_message
    slot = slot of the node sending the message
    round = current_time / round_length
    if current_time - round * round_length > start of slot in round then
        round = next round
    end if
    if not message fits in the slot of round then
        insert (needed slot length to fit, recommended slot lengths)
        round = next round
    end if
    put in schedule table (message, round, slot)
end
```

Figure 6. The schedule_message function

```
greedy
    for each slot
        for each node not yet allocated to a slot
            bind (node, slot, minimum possible length for this slot)
            for (1)every slot length or (2)recommended slot lengths
                schedule 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 solution
end
```

Figure 7. The Greedy Algorithm

the schedule length is calculated considering the TDMA round given so far. Several lengths are considered for a slot bound to a given candidate node. The first variant of this strategy, named Greedy 1, tries all the slot lengths. It 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). The second variant of the greedy strategy, named Greedy 2, tries only the slot lengths recommended by the **schedule_message** function.

A second algorithm we have developed is based on a simulated annealing (SA) strategy. The greedy strategy constructs the solution by progressively selecting the best candidate in terms of the schedule length produced by the function **schedule**. Unlike the greedy strategy, SA tries to escape from a local optimum by randomly selecting a new solution from the neighbors of the current solution. The new solution is accepted if it is an improved solution. However, a worse solution can also be accepted with a certain probability that depends on the deterioration of the cost function and on a control parameter called temperature.

In Figure 8 we give a short description of this algorithm. An essential component of the algorithm is the generation of a new solution x' starting from the current one x_{now}. The neighbors of the current solution x_{now} are obtained by a permutation of the slots in the TDMA round and/or by increasing/decreasing the slot lengths. We generate the new solution by either randomly swapping two slots (with a probability 0.3) or by increasing/decreasing with the smallest data unit the length of a randomly selected slot (with a probability 0.7).

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

For graphs with 160 and less processes we were able to run an exhaustive search that found the optimal solutions. For the rest of the graph dimensions, we performed very long and expensive runs

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

Figure 8. The Simulated Annealing Algorithm

with the SA algorithm, and the best ever solution produced has been considered as the optimum for the further experiments. Based on further experiments we have determined the parameters of the SA algorithm so that the optimization time is reduced as much as possible but the optimal result is still produced. For example, for the graphs with 320 nodes, TI is 500, TL is 400 and α is 0.97. The algorithm stops if for three consecutive temperatures no new solution has been accepted.

6. 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 two variants of the greedy algorithm. It is based on the percentage deviations of the schedule lengths produced by Greedy 1 and Greedy 2 from the lengths of the (near)optimal schedules obtained with the SA algorithm (see section 5). Table 1 presents the average and maximum percentage deviation for each of the graph dimensions, and the average execution time expressed in seconds.

No.of proc.	NaiveDesigner		Greedy 1			Greedy 2		
	aver.	max.	aver.	max.	time.	aver.	max.	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%	0.46s	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

Table 1: Percentage Deviation and Execution Times

Together with the greedy variants, 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 1 shows that 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, and the variant Greedy 1 (that considers all the possible slot lengths) performs slightly better than Greedy 2. However, the execution times are smaller for Greedy 2, than for Greedy 1. The average execution times for the SA algorithm to find the (near)optimal solutions are between 5 minutes for graphs with 80 processes and 275 minutes for 400 processes.

As a conclusion, the greedy approach is able to produce accurate results in a very short time. Therefore it can be also used for performance estimation as part of a larger design space exploration cycle. SA is able to find near-optimal results in reasonable time, and can be used for the synthesis of the final implementation of the system.

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 deadline of 110 ms. The naive designer's approach resulted in a schedule corresponding to a delay of 114 ms, that does not meet the deadline. Both of the greedy approaches produced a delay of 103 ms on the worst case execution time of the system, while the SA approach produced a schedule of 97 ms.

7. CONCLUSIONS

We have presented an approach to process scheduling 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 taking into consideration the overheads of the real-time kernel and the communication protocol. We have optimized communications through packaging of messages into slots with a properly selected order and lengths.

The scheduling algorithms proposed can be used both for accurate performance estimations and for system synthesis. 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 TD for his support and for providing the automotive electronics case study. The research has been partly supported by the Swedish Foundation for Strategic Research.

8. 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] 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.

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

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

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

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

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

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

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

[12] 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.

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

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