Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems

Petru Eles[1,2], Krzysztof Kuchcinski[1], Zebo Peng[1], Alexa Doboli[2], and Paul Pop[2]
[1] Dept. of Computer and Information Science
Linköping University
Sweden
[2] Computer Science and Engineering Department
Technical University of Timisoara
Romania

Abstract

We present an approach to process scheduling based on an abstract graph representation which captures both data-flow and the flow of control. Target architectures consist of several processors, ASICs and shared busses. We have developed a heuristic which generates a schedule table so that the worst case delay is minimized. Several experiments demonstrate the efficiency of the approach.

1. Introduction

In this paper we concentrate on process scheduling for systems consisting of communicating processes implemented on multiple processors and dedicated hardware components. In such a system in which several processes communicate with each other and share resources, scheduling 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 [13]. In our approach, we assume that some processes can be activated if certain conditions, computed by previously executed processes, are fulfilled. 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 an accurate and direct modeling of a wide range of applications.

Performance estimation at the process level has been well studied in the last years [10, 12]. Starting from estimated execution times of single processes, performance estimation and scheduling of a system containing several processes can be performed. In [14] performance estimation is based on a preemptive scheduling strategy with static priorities using rate-monotonic-analysis. In [11] scheduling and partitioning of processes, and allocation of system components are formulated as a mixed integer linear programming problem while the solution proposed in [8] is based on constraint logic programming. Several research groups consider hardware/software architectures consisting of a single programmable processor and an ASIC. Under these circumstances deriving a static schedule for the software component practically means the linearization of a dataflow graph [2, 6].

Static scheduling of a set of data-dependent software processes on a multiprocessor architecture has been intensively researched [3, 7, 9]. An essential assumption in these approaches is that a (fixed or unlimited) number of identical processors are available to which processes are progressively assigned as the static schedule is elaborated. Such an assumption is not acceptable for distributed embedded systems which are typically heterogeneous.

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

The paper is divided into 7 sections. In section 2 we formulate our basic assumptions and introduce the graph-based model which is used for system representation. The schedule table and the general scheduling strategy are presented in sections 3 and 4. The algorithm for generation of the schedule table is presented in section 5. Section 6 describes the experimental evaluation and section 7 presents our conclusions.

2. Problem Formulation and the Conditional Process Graph

We consider a generic architecture consisting of *programmable processors* and application specific *hardware processors* (ASICs) connected through several *busses*. These busses 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. Processes on different processors can be executed in parallel. Only one data transfer can be performed by a bus at a given moment. Computation and

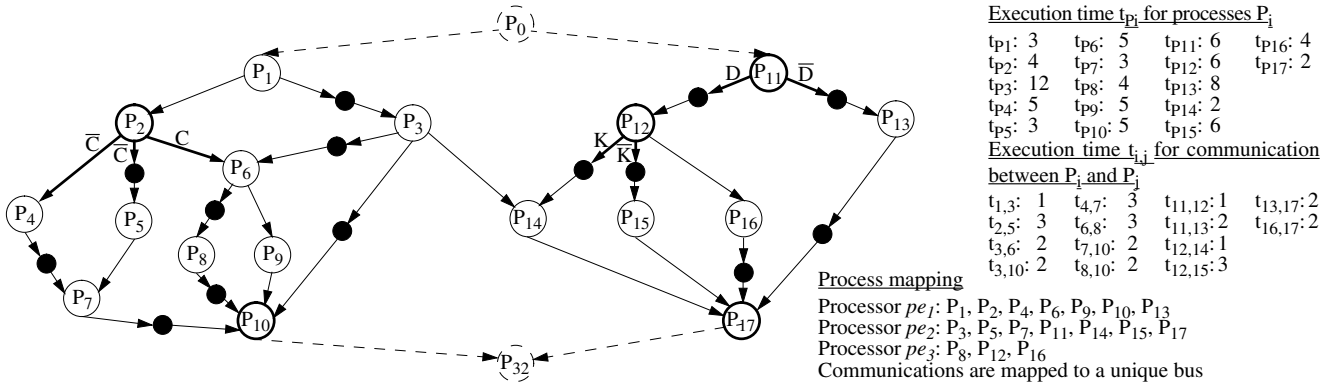


Fig. 1. Conditional Process Graph with execution times and mapping

data transfer on several busses can overlap.

In [4] 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 is assigned to a (programmable or hardware) processor and each communication channel which connects processes assigned to different processors is assigned to a bus. Our goal is 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.

As an abstract model for system representation we use a directed, acyclic, polar graph $\Gamma(V, E_S, E_C)$. Each node $P_i \in V$ represents one process. E_S and E_C are the sets of simple and conditional edges respectively. $E_S \cap E_C = \emptyset$ 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 process. These nodes are introduced as dummy processes so that all other nodes in the graph are successors of the source and predecessors of the sink respectively.

The mapping of processes to processors and busses is given by a function M: $V \rightarrow PE$, where $PE=\{pe_1, pe_2, .., pe_{Npe}\}$ is the set of processing elements. For any process P_i, $M(P_i)$ is the processing element to which P_i is assigned for execution.

Each process P_i, assigned to processor or bus $M(P_i)$, is characterized by an execution time t_{Pi}. In the process graph depicted in Fig. 1, P_0 and P_{32} are the source and sink nodes respectively. Nodes 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 are so called *communication processes* (P_{18}, P_{19}, .., P_{31}). They are represented in Fig. 1 as black dots and are introduced for each connection which links processes mapped to different processors. These processes model inter-processor communication and their execution time is equal to the corresponding communication time.

An edge $e_{ij} \in E_C$ is a *conditional edge* (thick lines in Fig. 1) and it has an associated condition. Transmission on such an edge takes place only if the associated condition is satisfied. 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 circles representing conjunction and disjunction nodes are depicted with thick borders. We assume that conditions are independent.

A boolean expression X_{Pi}, called guard, can be associated to each node P_i in the graph. It represents the necessary condition for the respective process to be activated. In Fig. 1, for example, $X_{P3}=true$, $X_{P14}=D \wedge K$, $X_{P17}=true$, $X_{P5}=\overline{C}$. Two nodes P_i and P_j, where P_j is not a conjunction node, can be 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 restriction avoids specifications in which a process is blocked 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 input paths.

According to our model, we assume that a process, which is not a conjunction process, can be activated only after all its inputs have arrived. A conjunction process can be activated after messages coming on one of the alternative paths have arrived. All processes issue their outputs when they terminate. If we consider the activation time of the source process as a reference, the activation time of the sink process is the delay of the system at a certain execution.

3. The Schedule Table

For a given execution of the system, a subset of the processes is activated which corresponds to the actual path through the process graph. This path depends on certain conditions. For each individual path there is an optimal schedule of the processes which produces a minimal delay. Let us consider the process graph 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. 4a). 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. 4b). 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. An algorithm has to be developed which produces a schedule of the processes so that the worst case delay is as small as possible. The output of this 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 values. Activation times in a given column represent starting times of the processes when the respective expression is true.

Table 1 shows part of the 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 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=35 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}: $\delta_{max}= \max\{34 + t_{10}, 37 + t_{17}\}=39$.

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 non-preemptive scheduler located on each programmable/ communication processor decides on process and communication activation depending on actual values of conditions. Once activated, a process executes until it completes. To produce a deterministic behavior which is correct for any combination of conditions, the table has to fulfill several requirements:

1. If for a certain process P_i, with guard X_{Pi}, there exists an activation time in the column headed by 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.
2. Activation times have to be uniquely determined by the conditions. Thus, if for a certain process P_i there are several alternative activation times then, for each pair of such times (τ_{Pi}^{Ej}, τ_{Pi}^{Ek}) placed in columns headed by expressions E_j and E_k, $E_j \wedge E_k$=*false*.
3. If for a certain execution of the system the guard X_{Pi} becomes true then P_i has to be activated during that execution. Thus, considering all expressions E_j corresponding to columns which contain an activation time for P_i, $\bigvee E_j = X_{Pi}$.
4. 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 $M(P_i)$ which executes P_i.

Table 1: Part of schedule table for the graph 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_{10}				34	34		26	26		34	26
P_{11}	0										
P_{14}					35			24			
P_{17}				29	37		30	26		22	24
P_{18} $1\rightarrow3$	3										
P_{19} $2\rightarrow5$						9					10
P_{20} $3\rightarrow10$				28	20		21	21		22	18
D	6										
C		7							7		
K			15			15					

The value of a condition is determined at the moment τ at which the corresponding disjunction process terminates. Thus, at any moment t, $t \geq \tau$, the condition is available for scheduling decisions on the processor which has executed the 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 needed for this communication.

The following strategy has been adopted for scheduling the communication of conditions: after termination of a disjunction process the value of the condition is broadcasted from the corresponding processor to all other processors; this broadcast is scheduled as soon as possible on the first bus which becomes available after termination of the disjunction process. For this task only busses 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 this communication is the same for all conditions and depends on the features of the employed buses. Given the minimal amount of transferred information, the time τ_0 is smaller than (at most equal to) any other communication time. The transmitted condition 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 conditions has been considered τ_0=1. The last three rows in Table 1 indicate the schedule for communication of conditions *C*, *D*, and *K*.

4. The Scheduling Strategy

Our goal is to derive a minimal worst case delay and to generate the corresponding schedule table for a process graph $\Gamma(V, E_S, E_C)$, a mapping function M: $V \rightarrow PE$, and execution times t_{Pi} for each process $P_i \in V$. At a certain

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 busses according to the actual interconnection topology.

execution of the system, one of the N_{alt} alternative paths through the process graph will be executed. Each alternative path 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 path) which represents the necessary conditions for that subgraph to be executed.

If at activation of the system all the conditions *would be* known, 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 conditions 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 path is not known in advance, should be as small as possible. We have developed a heuristic which, starting from the schedules corresponding to the alternative paths, produces the global schedule table, as result of a, so called, *schedule merging operation*. Hence, we perform scheduling of a process graph as a succession of the following two steps:

1. Scheduling of each individual alternative path;
2. Merging of the individual schedules and generation of the schedule table.

We present algorithms for scheduling of the individual paths in [5]. In this paper we concentrate on the generation mechanism of the global schedule table.

5. The Table Generation Algorithm

The input for the generation of the schedule table is a set of N_{alt} schedules, each corresponding to an alternative path, labeled L_k, through the process graph Γ. Each such schedule consists of a set of pairs (P_i, τ_{Pi}^{Lk}), where P_i is a process activated on path L_k and τ_{Pi}^{Lk} is the start time of process P_i according to the respective schedule. The schedule table generated as output fulfills the requirements presented in section 3.

The schedule merging algorithm is guided by the length of the schedules produced for each alternative path. While progressively constructing the schedule table, at each moment, priority is given to the requirements of the schedule corresponding to that path, among those which are still reachable, that produces the largest delay. Thus, we induce perturbations into the short delay paths and let the long ones proceed as similar as possible to their (near)optimal schedule.

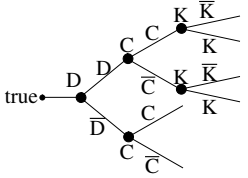
5. 1. Schedule Merging

The generation algorithm of the schedule table proceeds along a binary decision tree corresponding to all alternative paths, which is explored in a depth first order. Fig. 2 shows the decision tree explored during generation of Table 1. The nodes of the decision tree correspond to the states reached when, according to the actual schedule, a disjunction process has been terminated and the value of a new condition has been computed. The algorithm is guided by the following basic rules:

1. Start times of processes are fixed in the table according, with priority, to the schedule of that path which is reachable from the current state and produces the longest delay.
2. The start time τ_{Pi}^{Lk} of a process P_i is placed in a column headed by the conjunction of all condition values known at τ_{Pi}^{Lk} on the processing element $M(P_i)$, according to the current schedule. If such a column does not yet exist in the table, it will be generated.
3. After a new path has been selected, its schedule will be adjusted by enforcing the start times of certain processes according to their previously fixed values. This can be the case of a process P_i which is part of the current path labelled L_k ($L_k \Rightarrow X_{Pi}$), and of a previously handled path labelled L_q ($L_q \Rightarrow X_{Pi}$). When handling path L_q an activation time for process P_i has been fixed in a column headed by expression E. If E depends exclusively on conditions corresponding to tree nodes which are predecessors of the branching node between the two paths, then the schedule of the current path, L_k, has to be adjusted by taking into consideration the previously fixed activation time of P_i.
4. Further readjustments of the current schedule are performed in order to avoid violation of requirement 2 in section 3. This aspect will be discussed in subsection 5.2.

At the beginning, start times of processes are placed into Table 1 according to the schedule which corresponds to the path labeled $D \wedge C \wedge \overline{K}$. After the first back-step, to node K (Fig. 2), the schedule corresponding to path $D \wedge C \wedge K$ becomes the actual one. New start times will be fixed into the schedule table according to an adjusted version of this schedule. The next back-step is to node C. Two schedules are now reachable taking the branch $\overline{C}$, which are labelled $D \wedge \overline{C} \wedge \overline{K}$ and $D \wedge \overline{C} \wedge K$ respectively. $D \wedge \overline{C} \wedge \overline{K}$, which produces a larger delay, will be selected first as the actual schedule. It will be followed until the next beck-step has been performed.

The algorithm for generation of the schedule table is briefly described, as a recursive procedure, in Fig. 3. An essential aspect of this algorithm is that, after each back-step, a new schedule has to be selected as the current one. The selection rule gives priority to the path with the largest delay, among those which are reachable from the current node in the decision tree. Further start times of processes will be



Length of the optimal schedule for the alternative paths through the graph in Fig. 1

$D \wedge C \wedge \overline{K}$	39
$D \wedge C$	39
$D \wedge C \wedge K$	38
$D \wedge \overline{C} \wedge \overline{K}$	32
$D \wedge \overline{C} \wedge K$	31
$\overline{D} \wedge \overline{C}$	31

Fig. 2. Decision tree explored for the graph in Fig. 1

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

```
BuildScheduleTable(current_schedule, back_step)
  if back_step then
    Select new current_schedule
    Adjust current_schedule
    Check for conflicts and readjust current_schedule
  end if
  while not (EndOfSchedule or
        arrived at moment so that a disjunction process is terminated) do
    Take following process in current_schedule and
                                  place start time into ScheduleTable
  end while
  if EndOfSchedule then return end if
  BuildScheduleTable(current_schedule, false)
  BuildScheduleTable(current_schedule, true)
end BuildScheduleTable
```

Fig. 3. Algorithm for generation of a schedule

placed into the schedule table according to an *adjusted* version of the new current schedule. Processes which satisfy the condition of rule three presented above have to be moved to their previously fixed start time. They are considered as locked in this new position. As result of a simple rescheduling procedure the start times of the other, unlocked, processes are changed to the earliest moment which is allowed, taking in consideration data dependencies. Relative priorities of unlocked processes assigned to the same non-hardware processor are kept as in the original schedule.

In Fig. 4 we show the adjustment of the schedule labelled $D \wedge C \wedge K$ performed after the back-step to node K. At this moment start times of processes P_1, P_2, P_{11}, P_3, P_{12}, P_{18}, P_{27}, and of the communication processes for conditions D, C, and K have already been placed in the table according to the schedule of path $D \wedge C \wedge \overline{K}$ which is shown in Fig. 4b. The activation time of these processes has been placed in columns headed by expressions *true*, C or $D \wedge C$, and consequently they are mandatory also for path $D \wedge C \wedge K$ (both node C and D are predecessors of node K which is the branching node between the two paths). Under these circumstances some of the other processes have to be moved from their original position in this schedule, shown in Fig. 4a, to their position in the adjusted schedule of Fig 4c. This adjusted version is used in order to fix start times of further processes until the next back-step.

5. 2. Conflict Handling at Table Generation

Suppose we are currently handling a path labelled L_k. According to the adjusted schedule of this path we place an activation time τ_{Pi}^{Lk} of process P_i into the table, so that the respective column is headed by an expression E. The problem is how to preserve the coherence of the table in the sense introduced by requirement 2 defined in section 3. If there is no activation time previously introduced in the row corresponding to P_i no conflicts can be generated. If, however, the respective row contains activation times, there exists a potential of conflict between the column headed by E and columns which already include activation times of P_i. Let us consider that such a column is headed by an expression F. According to requirement 2, we have a conflict between columns E and F if there exists no condition C so that $E=q \wedge C$ and $F=q' \wedge \overline{C}$. Intuitively, such a conflict means that for two or more paths the same process P_i is scheduled at different times but the conditions known on the processing element $M(P_i)$ do not allow to the scheduler to identify the current path and to take a deterministic decision on activation of P_i.

If placement of an activation time for process P_i in a column headed by expression E produces a conflict, the current schedule has to be *readjusted* so that an expression E' will head the column that hosts the new activation time of P_i and no conflict is induced in the schedule table. As shown in the algorithm presented in Fig. 3, after adjustment of a schedule, unlocked processes are checked if their placement in the table produces any conflicts. If this is the case, the proc-

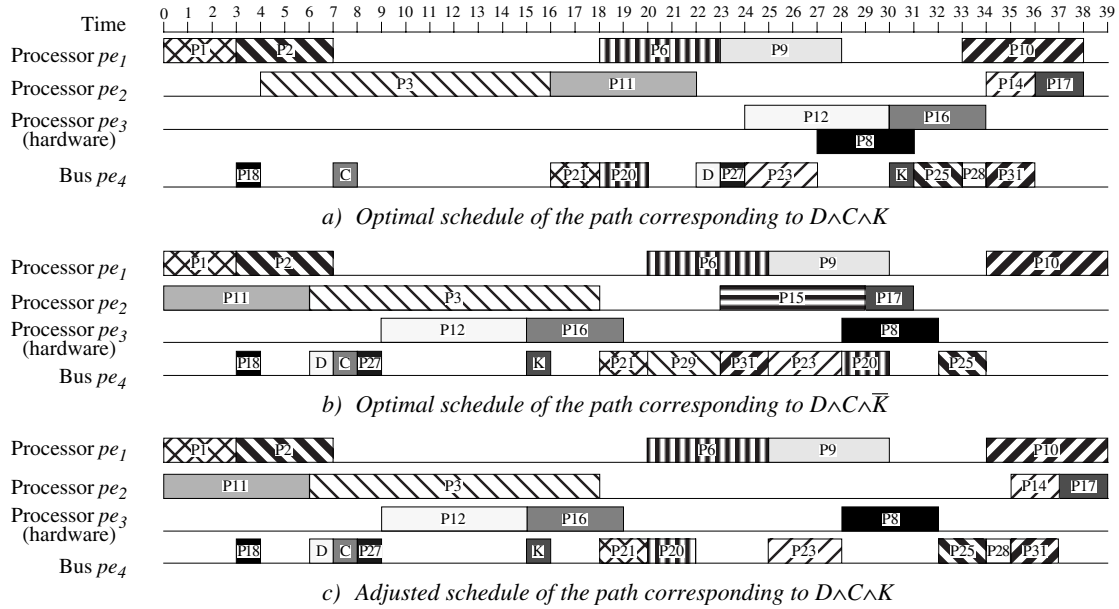


Fig. 4. Optimal and adjusted schedules for paths extracted from the process graph in Fig. 1

ess will be moved to a new activation time and the schedule is readjusted by changing the start time of some unlocked processes (similar to the operation performed at the initial adjustment). The main problem which has to be solved is to find the new activation time for P_i so that conflicts are avoided. In [5] we demonstrated the following two theorems in the context of our table generation algorithm:

Theorem 1

Consider a process P_i which is part of two paths, L_k and L_q, with activation times τ_{Pi}^{Lk} and τ_{Pi}^{Lq} respectively. If the set of predecessors of P_i is different in the two paths, then no conflict is possible between the columns corresponding to the two activation times.

As a consequence of this theorem readjustments for conflict handling can not impose an activation time of a process which is not feasible for the respective path.

Theorem 2

Consider a process P_i so that placement of its activation time τ_{Pi}^{L}, corresponding to the current path L, into the schedule table produces a conflict. There exists an activation time τ' of P_i, corresponding to one of the previously handled paths with which the current one is in conflict, so that τ' has the following property: if P_i is moved to activation time τ' in the current schedule, all conflicts are avoided.

Consider W the set of columns with which there exists a conflict at placement of the activation time for P_i. Based on Theorem 2 we know that one of the times τ_{Pi}^{F} placed in a column $F \in W$, represents a correct solution for conflict elimination. Thus, the following loop over the set W can produce the new activation time of a process P_i so that all conflicts are avoided:

for all columns $F \in W$ **do**
  if moving P_i to τ_{Pi}^{F} all conflicts are avoided **then**
    return τ_{Pi}^{F}
  end if

6. Experimental Evaluation

The strategy we have presented for generation of the schedule table guarantees that the path corresponding to the largest delay, δ_M, will be executed in exactly δ_M time. This, however, does not mean that the worst case delay δ_{max}, corresponding to the generated global schedule, is always guaranteed to be δ_M. Such a delay can not be guaranteed in theory. According to our scheduling strategy δ_{max} will be worse than δ_M if the schedule corresponding to an initially faster path is disturbed at adjustment or conflict handling so that its delay becomes larger than δ_M.

For evaluation of the schedule merging algorithm we used 1080 conditional process graphs generated for experimental purpose. 360 graphs have been generated for each dimension of 60, 80, and 120 nodes. The number of alternative paths 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 busses [5]. Experiments were run on a SPARCstation 20.

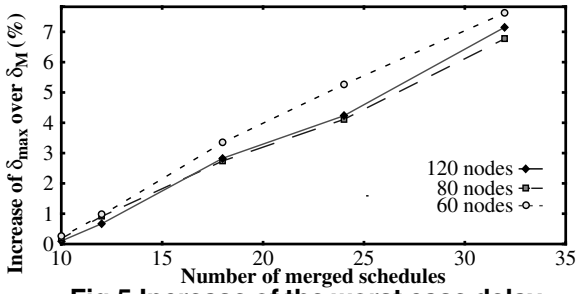


Fig.5.Increase of the worst case delay

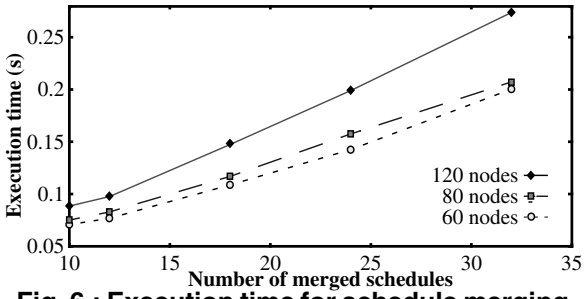


Fig. 6.: Execution time for schedule merging

Fig. 5 presents the percentage increase of the worst case delay δ_{max} over the delay δ_M of the longest path. The average increase is between 0.1% and 7.63% and, practically, it does not depend on the dimension of the graph but only on the number of merged schedules. It is worth to be mentioned that a zero increase ($\delta_{max}=\delta_M$) was produced for 90% of the graphs with 10 alternative paths, 82% with 12 paths, 57% with 18 paths, 46% with 24 paths, and 33% with 32 paths. In Fig. 6 we show the average execution time for the schedule merging algorithm, as a function of the number of merged schedules. The time needed for scheduling of the individual paths depends on the employed algorithm. As demonstrated in [5], good quality results can be obtained with a list scheduling based algorithm which needs less than 0.003 seconds for graphs having 120 nodes.

Finally, we present a real-life example which implements the operation and maintenance (OAM) functions corresponding to the F4 level of the ATM protocol layer [1]. Fig. 7a 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 [4] 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. 7b), the OAM block switches between these three modes. Execution in each mode is controlled by a

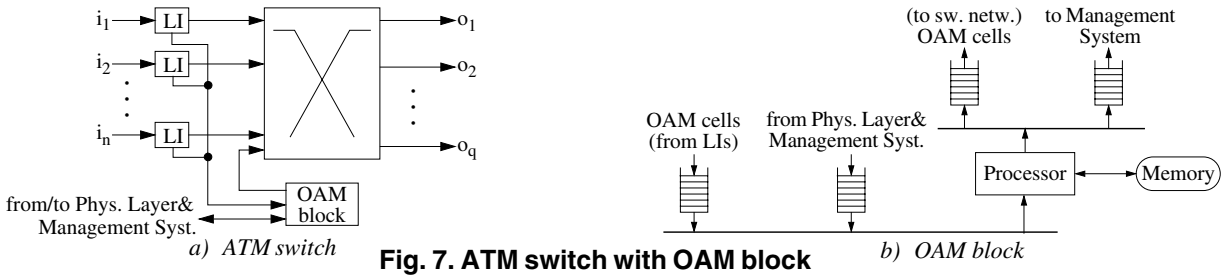


Fig. 7. ATM switch with OAM block

statically generated schedule table for the respective mode. We specified the functionality corresponding to each mode as a set of interacting VHDL processes. Table 2 shows the characteristics of the resulting process graphs. The main objective of this experiment was to estimate 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. 7b 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 target architectures are based on two types of processors: 486DX2/80MHz and Pentium/120MHz. For each architecture, processes have been assigned to processors taking into consideration the potential parallelism of the process graphs and the amount of communication between processes. The worst case delays resulting 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 486 processors instead of one. For the Pentium processor, however, the worst case delay can not 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 Pentium processors providing an additional memory module pays back by a reduction of the worst case delay in *mode 1*.

Table 2: Worst case delays for the OAM block

mode	Model		Worst case delay (ns)									
			1P/1M		1P/2M		2P/1M			2P/2M		
	nr. proc.	nr. paths	486	Pent.	486	Pent.	2× 486	2× Pent.	486+ Pent.	2× 486	2× Pent.	486+ Pent.
1	32	6	4471	2701	4471	2701	2932	2131	2532	2932	1932	2532
2	23	3	1732	1167	1732	1167	1732	1167	1167	1732	1167	1167
3	42	8	5852	3548	5852	3548	5033	3548	3548	5033	3548	3548

7. Conclusions

We have presented an approach to process scheduling for the synthesis of embedded systems. The approach is based on an abstract graph representation which captures, at process level, both dataflow and the flow of control. A schedule table is generated by a merging operation performed on the schedules of the alternative paths. 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 busses. The algorithms have been evaluated based on experiments using a large number of graphs generated for experimental purpose as well as real-life examples.

References

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

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

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

[4] P. Eles, Z. Peng, K. Kuchcinski, A. Doboli, “System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search”, *Des. Aut. for Emb. Syst.*, V2, N1, 1997, 5-32.

[5] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, P. Pop, “Process Scheduling for Performance Estimation and Synthesis of Embedded Systems”, Research Report, Department of Computer and Information Science, Linköping University, 1997.

[6] R. K. Gupta, G. De Micheli, “A Co-Synthesis Approach to Embedded System Design Automation”, *Des. Aut. for Emb. Syst.*, V1, N1/2, 1996, 69-120.

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

[8] K. Kuchcinski, "Embedded System Synthesis by Timing Constraint Solving", *Proc. Int. Symp. on Syst. Synth.*, 1997.

[9] Y.K. Kwok, I. Ahmad, "Dynamic Critical-Path Scheduling: an Effective Technique for Allocating Task Graphs to Multiprocessors", *IEEE Trans. on Par. & Distr. Syst.*, V7, N5, 1996, 506-521.

[10] Y. S. Li, S. Malik, “Performance Analysis of Embedded Software Using Implicit Path Enumeration”, *Proc. ACM/IEEE Design Automation Conference*, 1995, 456-461.

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

[12] K. Suzuki, A. Sangiovanni-Vincentelli, “Efficient Software Performance Estimation Methods for Hardware/Software Codesign”, *Proc. ACM/IEEE DAC*, ’96, 605-610.

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

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