Process Scheduling for Performance Estimation and Synthesis of Hardware/Software Systems

Petru Eles[1], Krzysztof Kuchcinski[1], Zebo Peng[1], Alexa Doboli[2], and Paul Pop[1]
[1] Dept. of Computer and Information Science
Linköping University
Sweden
[2] Dept. of Electr.&Comp. Eng. and Comp. Science
University of Cincinnati
USA

Abstract

The paper presents an approach to process scheduling for embedded systems. Target architectures consist of several processors and ASICs connected by shared busses. We have developed algorithms for process graph scheduling based on list-scheduling and branch-and-bound strategies. One essential contribution is in the manner in which information on process allocation is used in order to efficiently derive a good quality or optimal schedule. Experiments show the superiority of these algorithms compared to previous approaches like critical-path heuristics and ILP based optimal scheduling. An extension of our approach allows the scheduling of conditional process graphs capturing both data and control flow. In this case a schedule table has to be generated so that the worst case delay is minimized.

1. Introduction

During synthesis of an embedded system the designer maps the functionality captured by the input specification on different architectures, trying to find the most efficient solution which, at the same time, meets the design requirements. This design process implies the iterative execution of several allocation and partitioning steps before the hardware and software components of the final implementation can be generated. Both allocation and partitioning can be performed manually or automatically. Accurate performance estimation tools are essential components of such a system synthesis environment. They provide the adequate feedback on design decisions needed for efficient exploration of the design space.

Both for accurate performance estimation and for the final synthesis of an efficient system good quality scheduling algorithms are required. In this paper we concentrate on process scheduling for systems consisting of communicating processes implemented on multiple processors and dedicated hardware components. Optimal scheduling in such a context is an NP complete problem [13]. Thus, it is essential to develop heuristics which produce good results in a reasonable time.

In [15] performance estimation is based on a preemptive scheduling strategy with static priorities using rate-monotonic-analysis. In [12] scheduling and partitioning of processes, and allocation of system components are formulated as a mixed integer linear programming (MILP) problem while the solution proposed in [11] 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, 8].

Static scheduling of a set of data-dependent software processes on a multiprocessor architecture has been intensively researched [3, 10, 14]. 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 a set of interacting processes which have been mapped on an architecture consisting of several different processors and dedicated hardware components connected by dedicated and/or shared busses. Considering a non-preemptive execution environment we statically generate a process schedule and estimate the worst case delay. We have developed list scheduling and branch-and-bound based algorithms which can be used both for accurate and fast performance estimation and for optimal or close to optimal system synthesis. Based on extensive experiments we evaluated the performance of the algorithms which are superior to previous approaches like critical-path heuristics and ILP based optimal scheduling. Finally, we have extended our approach by assuming that some processes can be activated if certain conditions, computed by previously executed processes, are fulfilled. This is an important contribution because it allows to capture both data and control flow at the process level.

The paper is divided into 8 sections. In section 2 we formulate our basic assumptions and introduce the formal graph-based model which is used for system representation. The list scheduling and branch-and-bound based heuristics are presented in sections 3 and 4 and are evaluated in section 5.

The extension of our approach to accept conditional activation of processes is presented in section 6. Section 7 describes an example and section 8, finally, presents our conclusions.

2. Problem Formulation and the Process Graph

We consider a generic architecture consisting of general purpose or application oriented *programmable processors* and application specific *hardware processors* (ASICs) connected through several *busses*. Only one process can be executed at a time by a programmable processor while a hardware processor can execute processes in parallel. Only one data transfer can be performed by a bus at a given moment. Computation and data transfer can overlap.

In [5] 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 executed on different processors is assigned to a bus. Our goal is to derive a 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 $G(V,E)$. Each node $P_i \in V$ represents one process and 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 are successors of the source and predecessors of the sink respectively.

The mapping of processes is given by a function $M: V \rightarrow PE$, where $PE=\{pe_1, pe_2, .., pe_{Npe}\}$ is the set of processing elements. $PE=PP \cup HP \cup B$, where PP is the set of programmable processors, HP is the set of hardware components, and B is the set of busses. For a process P_i, $M(P_i)$ is the processing element to which P_i is assigned for execution. Each process P_i, assigned to $M(P_i)$, is characterized by an execution time t_{Pi}. *Communication processes*, assigned to $p_{ei} \in B$, 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. A process can be activated after all its inputs have arrived and it issues its outputs when it terminates. Once activated, a process executes until it completes.

In the following two sections we present algorithms for scheduling of a process graph. Depending on their speed and accuracy, they can be used at different stages of the design process for system synthesis and/or estimation. One of the essential contributions consists in the manner in which information on process allocation is used by these algorithms in order to efficiently derive a good quality or optimal schedule.

3. Partial Critical Path (PCP) Scheduling

List scheduling heuristics [3, 9, 14] are based on priority lists from which processes are extracted in order to be scheduled at certain moments. In our algorithm, presented in Fig. 1, we have such a list, $List_{pei}$, for each processing element pe_i. It contains the processes which are eligible to be activated on the respective processor at time *T_current*. These are processes which have not been yet scheduled but have all predecessors already scheduled and terminated. 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 *Select* from the list corresponding to a programmable processor or a bus in order to be scheduled. Processes assigned to a hardware processor pe_i are scheduled, without any restriction, immediately after they entered $List_{pei}$.

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

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

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

Let us consider the graph in Fig. 2 and suppose that the list scheduling algorithm has to decide between scheduling process P_A or P_B which are both ready to be scheduled on the same programmable processor or bus pe_i. In Fig. 2 we depicted only the critical path from P_A and P_B to the sink node. Let us consider that P_X is the last successor of P_A on the critical path such that all processes from P_A to P_X are assigned to the same processing element pe_i. The same holds for P_Y relative to P_B. t_A and t_B are the total execution time of the chain of processes from P_A to P_X and from P_B to P_Y respectively, following the critical paths. λ_A and λ_B are the total execution times of the processes on the rest of the two critical paths. Thus, we have:

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

Preliminary operations for priority assignment
for each processing element pe_i, i=1,2, ..., N_{pe}, **do** $free_{pei}$=0 **end for**;
T_current=0; schedule P_0 at *T_current*; -- P_0 is the source node
repeat
  Update ready process lists $List_{pei}$;
  for each processing element pe_i, i=1,2, ..., N_{pe}, **do**
    if $p_{ei} \in HP$ **then**
      -- hardw. processors support several processes at a time
      Schedule all processes in $List_{pei}$ at time *T_current*;
    elseif *T_current* ≥ $free_{pi}$ **then**
      -- programmable processors or buses support one process at a time
      p=*Select*($List_{pei}$); schedule p at time *T_current*;
      $free_{pei}$=*T_current*+t_p
    end if
  end for
  T_current=t_{next}, where t_{next} is the next time a scheduled process terminates;
until all direct predecessors of P_N are scheduled; -- P_N is the sink node

Fig. 1. List scheduling

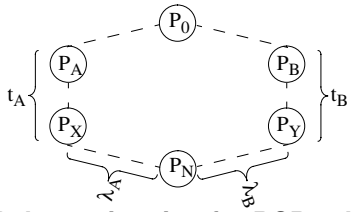


Fig. 2. Delay estimation for PCP scheduling

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

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

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

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

As a conclusion, for PCP scheduling we use the value of λ_{Pi} as a priority criterion instead of the length l_{pi} of the whole critical path. Thus, we take into consideration only that part of the critical path corresponding to a process P_i which starts with the first successor of P_i that is assigned to a processor different from $M(P_i)$. The complexity of PCP priority assignment is the same as for CP ($O(v+e)$). Experimental evaluation of PCP scheduling is presented in section 5.

4. A Branch-and-Bound (BB) Based Heuristic

As our experiments show, the PCP based algorithm is able to produce, with a short execution time, relatively good quality schedules. Nevertheless, due to the limited investigation capacity which is typical for list scheduling and the priorities employed, list scheduling algorithms are not always able to find very good quality or optimal results.

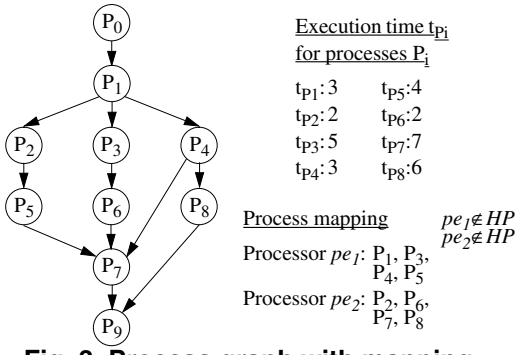


Fig. 3. Process graph with mapping

The branch-and-bound (BB) strategy is based on a more extensive search, visiting several (in the worst case all) alternative solutions in order to find the optimal one. In order to apply a BB strategy, the state space corresponding to the problem is organized as a *state tree*. Each node S_i corresponds to a certain state and the children of S_i are those states which can be reached from S_i as result of a scheduling decision. The number of children derived from a certain state depends on the number of different decisions which can be taken after the first process active in the respective state terminates. Among these decisions we have to consider the alternative to keep a certain (non-hardware) processor idle even if there exists a ready process to be executed on it. Each path from the root of the tree to a leaf node corresponds to a possible solution obtained after a sequence of decisions. We are interested in finding the leaf node S_k, such that the path from the root to S_k corresponds to the optimal solution.

Fig. 4 shows part of the state tree explored at scheduling of the process graph in Fig. 3. Each node in the tree corresponds to a state during the scheduling process. In each

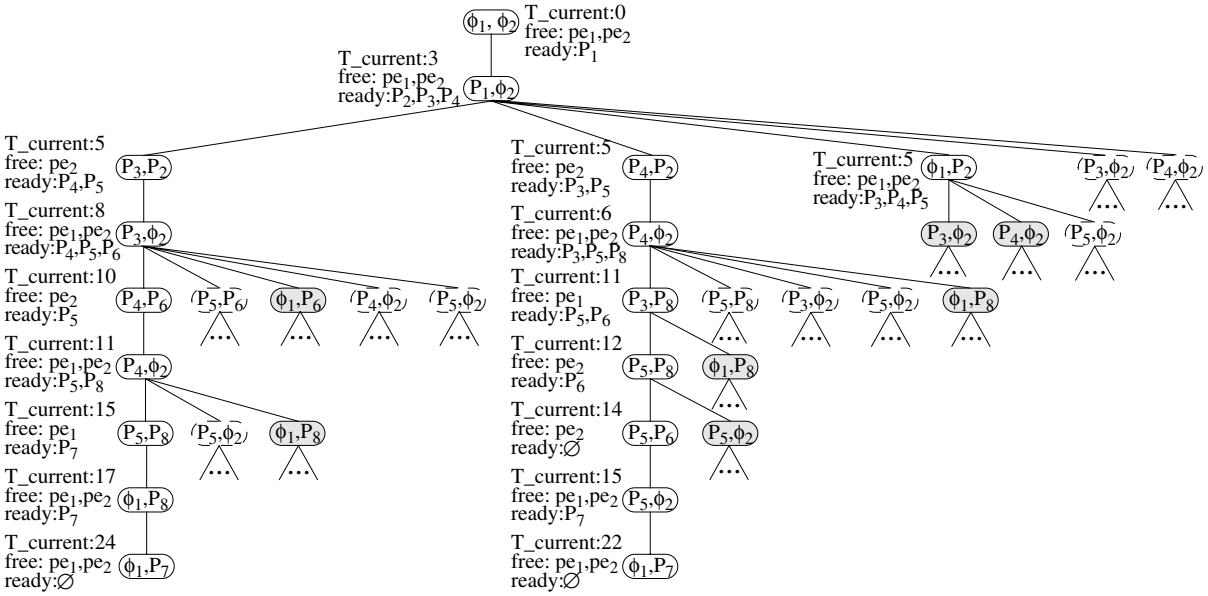


Fig. 4.: A part of the state tree explored at scheduling of the process graph in Fig. 3

node we inscribed the pair of processes which are active on the two processing elements pe_1 and pe_2. For some nodes we also gave the current time, the free processor(s) and the ready process(es) at the moment when the decision is taken to pass to a next state. For *T_current*=0, P_1 is activated as the only ready process. The event which allows to leave this new state is the termination of P_1 at *T_current*=3 when both processors are available and P_2, P_3, and P_4 are ready to be executed. Five different decisions are now possible which lead to five descendent nodes. Thus, processes P_3 and P_2, or P_4 and P_2 can be scheduled on processors pe_1 and pe_2 respectively. However, there are three more alternatives in which processor pe_1 is kept idle with pe_2 executing P_2, or pe_2 is kept idle with pe_1 executing P_3 or P_4 (we denote an idle processor pe_i by specifying on the respective position the empty process ϕ_i). The leaf node on the left extreme corresponds to a scheduling solution which can be produced by the PCP heuristic. The other leaf node represents the state corresponding to the optimal schedule.

BB is based on the idea to visit only a part of the state tree without missing any state which can lead to the optimal solution. The selection and branching rules used in our BB-based algorithm are presented in [6]. Here we concentrate on the estimation algorithms used for the bounding rule.

Before branching from a node *S*, a decision is taken if exploration has to continue on the respective subtree or the subtree can be cut. The decision is based on two values: the upper bound *U* (which is the length of the best schedule found so far) and the lower bound LB_S (which sets a lower limit on the length of the schedule corresponding to any leaf node in the subtree originating from *S*). Thus, whenever for a certain node *S* the inequality $LB_S \geq U$ holds, the corresponding subtree is cut. The estimation of such a lower bound is a critical aspect of any BB approach.

Let us consider a node *S* in the solution tree and Λ the set of active processes corresponding to the respective state. The state represented by node *S* is the result of a sequence of decisions captured by the path from the root to the node. From the point of view of the graph $G(V,E)$ to be scheduled, these decisions have determined the start times t^s_{Pi} assigned to processes P_i which are members of a set $\Phi \subset V$. If we consider a process P_i, such that $P_i \in \Phi$, then for each process P_j, such that P_j is a predecessor of P_i, it is true that $P_j \in \Phi$. All processes P_k, such that $P_k \in V-\Phi$, will get start times as result of future decisions. All these decisions produce states which are part of the subtree originating in node *S*. What we have to do is to estimate a value LB_S such that $LB_S \leq \min_k(\delta_k)$, where δ_k is the length of the schedule corresponding to the *k*th leaf node in the subtree derived from node *S*.

The lower bound estimation practically implies estimations of start times t^s_{Pk} and exit times t^f_{Pk} for all processes $P_k \in V-\Phi$. An initial bound ω_{Pk} on the start times can be set for each process $P_k \in V-\Phi$. This bound depends only on the processing element pe_l on which the process is executed. Thus, $\omega_{Pk}=b_{pel}$ if $M(P_k)=pe_l$, where

$$b_{pel} = t^f_{Pj}, \text{ if } \exists P_j \in \Lambda \text{ such that } P_j \neq \phi_l, M(P_j) = pe_l, pe_l \notin HP;$$

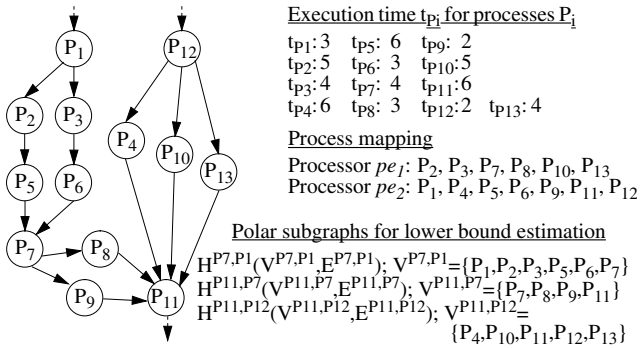


Fig. 5. Polar subgraphs for lower bound estimation

$$b_{pel} = \min_{\substack{P_i \in \Lambda \\ P_i \neq \phi_{M(Pi)}}} (t^f_{Pi}), \text{ if } (\phi_l \in \Lambda, pe_l \notin HP) \text{ or } (pe_l \in HP).$$

This initial bounding reflects the fact that no process $P_k \in V-\Phi$ is started before any already scheduled process assigned to the same non-hardware processing element ($pe_l \notin HP$) terminates. On a programmable processor or bus which has been kept idle in state *S* ($\phi_l \in \Lambda$, $pe_l \notin HP$), as well as on a hardware component ($pe_l \in HP$), no new process is started before any active process terminates.

Lower bound estimation is based on traversal of the paths linking anchor processes with the sink of the process graph. The set of *anchor processes* is defined as $\Delta=\{P_i | P_i \in V-\Phi, pred(P_i) \subset \Phi\}$; $pred(P_i)$ is the set of all direct predecessors of P_i. Nodes have to be visited in a topological order, starting from the anchor processes, and to each visited node P_x the following start time is assigned:

$t^s_{Px}=\max(\omega_{Px}, t^f_{pred(Px)})$, if P_x has only one direct predecessor;

$t^s_{Px}=\max(\omega_{Px}, \max_{P_i \in pred(P_x)}(t^f_{Pi}), \mu_{Px})$, if P_x has at least two direct predecessors.

The estimated lower bound LB_S *is the estimated start time* t^s *for the sink node*. A node with at least two direct predecessors is a *join* node. A *fork* node is a node with at least two direct successors. Estimation is based on the fact that no process can be started before all its predecessors have finished. If P_x is a join node, the value μ_{Px} will be obtained as result of a heuristic applied to polar subgraphs delimited by P_x and a corresponding fork node respectively. In [6] we present the heuristic for determination of μ_{Px}. Here we illustrate this algorithm by an example using the graph in Fig. 5. If the visited node is the join node P_7, μ_{P7} is derived considering the polar subgraph $H^{P7,P1}$ delimited by P_7 and the fork node P_1. We identify two sets, $\Pi_1^{P7,P1}=\{P_2,P_3\}$ and $\Pi_2^{P7,P1}=\{P_5,P_6\}$, containing processes assigned to processing elements $pe_1 \notin HP$ and $pe_2 \notin HP$ respectively (source and sink nodes are not included in these sets). We denote $Out_1^{P7,P1}$ the length of the shortest path from any process in $\Pi_1^{P7,P1}$ to the sink. Similarly $In_2^{P7,P1}$ is the length of the shortest path from the source to any process in $\Pi_2^{P7,P1}$. We can observe that $Out_1^{P7,P1}=3$, $In_2^{P7,P1}=4$, $Out_2^{P7,P1}=0$, and $In_1^{P7,P1}=0$. Determination of μ_{P7} is based on the fact that the time interval elapsed between termination of the source and activation of the sink has to be larger than the total execution time of the processes assigned to pe_1 plus the time intervals $In_1^{P7,P1}$ and

$Out_1^{P7,P1}$. The same applies to processes assigned to pe_2. Thus:

$$\mu_{P7} = t^f_{P1} + \max(In_1^{P7,P1} + \sum_{P_i \in \Pi_1^{P7,P1}} t_{Pi} + Out_1^{P7,P1},$$
$$In_2^{P7,P1} + \sum_{P_i \in \Pi_2^{P7,P1}} t_{Pi} + Out_2^{P7,P1}) = t^f_{P1} + 13.$$

For join node P_{11}, the same procedure has to be performed on the polar subgraphs $H^{P11,P12}$ and $H^{P11,P7}$. The maximum of the two resulting values, one for each subgraph, is the value for μ_{P11}. It is interesting to observe that the subgraph $H^{P11,P1}$ has been omitted from the discussion. It has the particularity that it contains a node, P_7, such that each path from the source to the sink goes through that node. Estimations based on this subgraph can not provide a better bound for μ_{P11} than already obtained from successive evaluations based on $H^{P7,P1}$ and $H^{P11,P7}$.

It is very important that for all join nodes P_x, the relative fork nodes Q_i and the corresponding values *In*, *Out* as well as the sums of process execution times, can be determined statically and stored before the start of the BB search. Thus, the complexity of the dynamic bounding process is reduced to that of the partial traversal of the process graph starting from the anchor processes. The static determination of the pairs (P_x,Q_i) and of the associated values runs in time $O(v(v+e)\ log\ v)$, which is the actual complexity of generating the connectivity matrix corresponding to the graph.

The evaluation of the lower bound LB_S needs, as discussed above, a partial traversal of the process graph. Before starting this evaluation, a very fast estimation of two weaker bounds $lb1_S$ and $lb2_S$ is performed:

$$lb1_S = \max_{pe_k \notin HP} (b_{pek} + \sum_{\substack{P_i \in V-\Phi \\ M(P_i)=pe_k}} t_{Pi}); lb2_S = \max_{P_i \in \Delta} (\omega_{Pi} + l_{pi}).$$

If bounding with one of these bounds succeeds, which means that $lb1_S \geq U$ or $lb2_S \geq U$, the evaluation of LB_S can be avoided. The value of $lb1_S$ is computed based on the total execution time of the yet unscheduled processes ($P_i \in V-\Phi$) which are assigned to the same non-hardware processor $pe_k \notin HP$, and on the earliest time, b_{pek}, when such a process could be activated. If needed, $lb2_S$ is derived using the critical paths l_{Pi} (see section 3) and the estimated start times ω_{Pi} of the anchor processes $P_i \in \Delta$.

In the state tree presented in Fig. 4 all states bordered by dashed lines are cut using the three lower bounds presented above. The state in the left subtree corresponding to active processes (P_5,P_6) is cut using the bound LB_S while all other states represented with dashed lines are cut by the bounds $lb1_S$ and $lb2_S$[1].

Unlike the list scheduling based approaches previously presented, BB scheduling, which always produces an optimal schedule, is of exponential complexity *in the worst case*. Due to a good branching, selection, and bounding strategy, however, the complexity is significantly reduced in practice. Experimental results presented in the next section show that even for large problems BB scheduling can be a viable solution.

1. Generation of the shaded states in Fig. 4 is avoided as result of our branching rule [6].

5. Experimental Evaluation

For evaluation of the algorithms presented in the previous sections we used 1250 graphs generated for experimental purpose. We considered architectures consisting of one ASIC and one to eleven processors and one to eight busses [6]. All experiments were run on a SPARCstation 20. We have evaluated the percentage deviations of the schedule lengths produced by CP, UB[2] and PCP scheduling from the lengths of the optimal schedules. The optimal schedules were produced running the BB based algorithm. The average deviation for all graphs is 4.73% for UB, 4.69% for CP, and 2.35%, two times smaller, for PCP. Deviations for the individual graphs are in the interval [0%, 44.74%] for all three heuristics. Execution times for the three list scheduling algorithms are very similar; scheduling of graphs up to 200 nodes is performed in less than 0.007 seconds.

Table 1 shows the percentage of the final results obtained with BB after certain time limits. It shows that for a very large majority of the graphs having 75 or less nodes the optimal schedule can be obtained in less than 0.3 seconds. After 3 seconds the optimal schedule for more than 50% of the 200-node graphs has already been generated. At the same time, in order to get a good schedule it is not necessary to let the BB algorithm run until termination. Table 2 shows the average and maximal deviation of the schedule lengths produced with PCP from the lengths of the intermediate schedules obtained with BB after certain time limits. For example, if we interrupt after 1 second the still running algorithms for the 130-node graphs, we get intermediate results which in average are 2.71% better than the PCP schedule but, for certain graphs, can be up to 10.31% better.

We have performed similar experiments using an ILP

Table 1: Percentage of final (optimal) results obtained with the BB algorithm

time lim. (s)	20 proc.	40 proc.	75 proc.	130 proc.	200 proc.
0.04	91.6%	0.0%	0.0%	0.0%	0.0%
0.08	95.6%	54.0%	0.0%	0.0%	0.0%
0.3	98.8%	83.6%	66.4%	0.0%	0.0%
1	99.6%	87.6%	77.6%	56.8%	0.0%
3	99.6%	89.6%	79.2%	70.8%	51.0%
5	100%	90.4%	79.2%	72.0%	62.1%
60	100%	92.4%	80.8%	76.8%	71.5%
1800	100%	96.8%	84.8%	80.4%	79.5%

Table 2: Percentage deviation of PCP schedule from intermediate results obtained with BB

time (s)	40 processes		75 processes		130 processes		200 processes	
	aver.	max.	aver.	max.	aver.	max.	aver.	max.
1	1.94%	17.65%	2.25%	16.11%	2.71%	10.31%	0%	0%
5	1.67%	5.50%	2.45%	16.11%	2.76%	8.11%	1.99%	21.10%
60	1.48%	4.92%	2.68%	19.01%	2.96%	10.73%	2.13%	10.58%
300	1.39%	4.26%	2.96%	19.01%	3.04%	13.73%	2.50%	12.75%

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

based formulation in order to derive optimal schedules [4]. Execution times for 40-node graphs were in this case more than an order of magnitude higher then with our BB algorithm. For 75-node graphs the execution times using ILP are already prohibitively large.

As a conclusion, PCP based list scheduling produces very quickly schedules of a good quality. This quality is superior to those produced by other list-scheduling heuristics, without any penalty in execution time. With our BB algorithm it is possible to get the optimal schedule for even large number of processes in a relatively short time. At the same time, the algorithm can very quickly produce schedules which are of very high, close to optimal, quality.

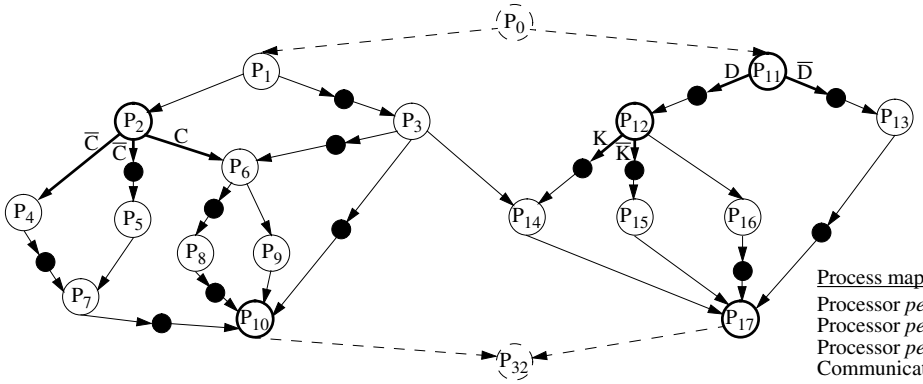
6. Scheduling of Conditional Process Graphs

An important extension of our system, presented in [6, 7], allows the scheduling of specifications which capture both data and control flow at the process level. In this section we present some of the basic ideas underlying this approach. We assume that some processes can be activated if certain conditions, computed by previously executed processes, are fulfilled. Thus, at a given activation of the system, only a certain subset of the total amount of processes is executed and this subset differs from one activation to the other.

The abstract model, presented in section 2, has been extended in order to capture conditional activation. A *conditional process graph* (Fig. 6) is a directed, acyclic, polar graph $\Gamma(V, E_S, E_C)$. 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. In Fig 6 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 communication processes (P_{18}, P_{19}, .., P_{31}) and are represented as black dots. An edge $e_{ij} \in E_C$ is a *conditional edge*. A conditional edge (thick lines in Fig. 6) has an associated condition. Transmission on a conditional edge e_{ij} will take place only if the associated condition is satisfied. We call a node with conditional edges at its output a *disjunction node*. Alternative paths starting from a disjunction node, which correspond to a certain condition, are disjoint and they meet in a so called *conjunction node*. In Fig. 6 circles representing conjunction and disjunction nodes are depicted with thick borders. We assume that conditions are independent and alternatives starting from different processes can not depend on the same condition. A conjunction process can be activated after messages coming on one of the alternative paths have arrived. 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. 6, for example, $X_{P3}=true$, $X_{P14}=D \wedge K$, $X_{P17}=true$, $X_{P5}=\overline{C}$.

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 conditional process graph in Fig. 6. 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. 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. This example reveals one of the difficulties when generating a schedule for a system like that in Fig. 6. 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.



Execution time t_{Pi} for processes P_i

t_{P1}: 3	t_{P6}: 5	t_{P11}: 6	t_{P16}: 4
t_{P2}: 4	t_{P7}: 3	t_{P12}: 6	t_{P17}: 2
t_{P3}: 12	t_{P8}: 4	t_{P13}: 8	
t_{P4}: 5	t_{P9}: 5	t_{P14}: 2	
t_{P5}: 3	t_{P10}: 5	t_{P15}: 6	

Execution time $t_{i,j}$ for communication between P_i and P_j

$t_{1,3}$: 1	$t_{4,7}$: 3	$t_{11,12}$: 1	$t_{13,17}$: 2
$t_{2,5}$: 3	$t_{6,8}$: 3	$t_{11,13}$: 2	$t_{16,17}$: 2
$t_{3,6}$: 2	$t_{7,10}$: 2	$t_{12,14}$: 1	
$t_{3,10}$: 2	$t_{8,10}$: 2	$t_{12,15}$: 3	

Process mapping

Processor pe_1: P_1, P_2, P_4, P_6, P_9, P_{10}, P_{13}
Processor pe_2: P_3, P_5, P_7, P_{11}, P_{14}, P_{15}, P_{17}
Processor pe_3: P_8, P_{12}, P_{16}
Communications are mapped to a unique bus

Fig. 6. Conditional Process Graph with execution times and mapping

Table 3: Part of schedule table for the graph in Fig. 6

	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					

Table 3 shows part of the schedule table corresponding to the system depicted in Fig. 6. 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 $E_k \Rightarrow X_{Pi}$ (X_{Pi} is true whenever is E_k true); 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.

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. For the example given in Table 3 communication time for conditions has been considered τ_0=1. The last three rows in Table 3 indicate the schedule for communication of conditions C, D, and K.

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 subgraph $G_k \in \Gamma$ which corresponds to the actual path through the process graph. 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 $G_k \in \Gamma$ (N_{alt} is the number of alternative paths).

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 (using one of the algorithms presented in the previous sections);
2. Merging of the individual schedules and generation of the schedule table.

The algorithm for merging individual schedules and table generation is presented in [6, 7].

7. An Example

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. 7 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 [5] 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 per-

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

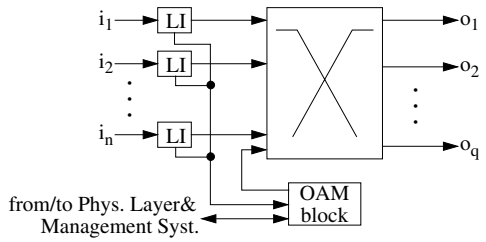


Fig. 7. ATM switch with OAM block

formed by the *OAM block* and can be implemented in software.

We have identified three independent modes in the functionality of the OAM block [6]. Execution in each mode is controlled by a statically generated schedule table for the respective mode. We specified the functionality corresponding to each mode as a set of interacting VHDL processes. Table 4 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.

Our experiments included architecture models with one processor and one memory module (1P/1M), 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 scheduling for each of the three modes are given in Table 4. 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 4 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 4: 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

8. Conclusions

We have presented an approach to process scheduling for the synthesis of embedded systems. For scheduling of process graphs we presented both a very fast list scheduling heuristic and a branch-and-bound based algorithm used successfully for generation of optimal schedules. One of the most interesting aspects concerning these algorithms is the manner in which information on process allocation is used in order to improve the quality of the result and to reduce execution time. The approach has been extended in order to capture at process level both dataflow and the flow of control. A schedule table is generated after scheduling of a conditional process graph, so that the worst case execution delay is minimized.

The scheduling approach we have presented can be used both for performance estimation and for system generation. 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 Books, '95.

[2] P. Chou, G. Boriello, "Interval Scheduling: Fine-Grained Code Scheduling for Embedded Systems", *Proc. 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] A. Doboli, "Contributions to the Design of Hardware/Software Multiprocessor Systems", PhD Thesis, Technical University Timisoara, 1997.

[5] P. Eles, Z. Peng, K. Kuchcinski, A. Doboli, "System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search", *Des. Autom. for Emb. Syst.*, V2, 1, 1997, 5-32.

[6] 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.

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

[8] R. K. Gupta, G. De Micheli, "A Co-Synthesis Approach to Embedded System Design Automation", *Des. Autom. for Emb. Syst.*, V1, 1/2, 1996, 69-120.

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

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

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

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

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

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

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