Scheduling with Start Time Related Deadlines

Přemysl Šůcha, Zdeněk Hanzálek
Department of Control Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
{suchap,hanzalek}@fel.cvut.cz

Abstract

This paper presents a scheduling problem for monoprocessor without preemption with timing constraints given by a task-on-node graph. The precedence relations are given by an oriented graph where edges are related either to the minimum time or to the maximum time elapsed between start times of the tasks. The processing time of a given task is associated to a given node in the oriented graph. The problem, finding an optimal schedule satisfying the timing constraints while minimizing makespan C_{max}, is solved by two solutions. The first is implemented as a B&B algorithm using a Critical Path estimation and estimation of Remaining Processing Time. Since the objective is to find a feasible schedule with minimal C_{max}, the bounding procedure uses the best known solution as a new dynamic timing constraint. It considers also scheduling anomaly while deciding feasibility of given solution. The second solution is based on ILP. Experimental results show comparison of the B&B and ILP solution.

1 Introduction

Many applications typically found in control and signal processing applications require quite complex algorithms usually given by set of recurrent equations (for example numeric filters). An eventuality to accelerate their computation speed is using of processors with pipelined arithmetic units. A goal of the monoprocessor scheduling is to assign tasks (i.e. the algorithm operations) to the processor in time.

Traditional scheduling algorithms (e.g., [1]) typically assume that deadlines are absolute, i.e., the deadlines are related to the schedule begin. In periodic real-time scheduling algorithms [4, 10], deadline for a given task instance is related to the begin of the corresponding period. However in many communication applications, realized as algorithms running on signal processors [6] or FPGAs, the deadline of the task T_j is related to the start time of the task T_i. The upper bound of start time of the task T_j is given by the start time of the task T_i plus some given time, since preemption is not assumed in this scheduling problem. Therefore, the absolute deadlines cannot be calculated aprior. Moreover we assume *precedence delays*, i.e. the precedence constraints that can be burden by some additional delays corresponding to hidden tasks not scheduled on a given processor (e.g., *pipeline scheduling problems* [8] or hidden tasks scheduled on infinite number of processors in FPGA [11]).

Formulation of the scheduling problem (monoprocessor, precedence delays, *start time related deadlines*, C_{max}) is based on [6], where it is solved by an heuristic algorithms. In this paper we propose an optimal scheduling algorithm based on branch and bound method and a solution using integer linear programming (ILP). In addition multiple deadlines are considered. Moreover, Brucker [3] has shown that complex scheduling problems like general shop problems, problems with multi–processor tasks, problems with multi–purpose machines and problems with changeover time can be reduced to our scheduling problem.

From the time complexity point of view, the scheduling problem presented in this article is NP-hard, since the scheduling problem $1|r_j, \widetilde{d}_j|C_{max}$ [2] is reducible to it. We remind that the problem $1|r_j, \widetilde{d}_j|C_{max}$ was proven to be NP-hard by reduction from the 3-PARTITION problem [9]. Moreover, NP-hard pipeline scheduling problem presented in [8] is also reducible to the scheduling problem presented in this article. On the other hand the presented scheduling problem is decidable since it can be solved by ILP [13].

This paper is organized as follows. Section 2 presents the scheduling problem formulation. Section 3 describes the optimal Branch and Bound algorithm using several bounding mechanisms. The next section presents alternative solution of the scheduling problem by ILP. Finally experimental results and comparison of B&B and ILP solutions is summarized in Section 5.

2 Formulation of the Scheduling Problem

The scheduling problem under assumption (originally defined in [6]) is given by a task-on-node graph G. Each task T_i is represented by the node T_i in the graph G and has a positive processing time p_i. Timing constraints between two nodes are represented by a set of directed edges. Each edge e_{ij} from the node T_i to the node T_j is labeled with an integer weight w_{ij}. There are two kinds of edges: the *forward edges* with positive weights and the *backward edges* with negative weights. The forward edge from the node T_i to the node T_j with the positive weight w_{ij} indicates that s_j, the start time of T_j, must be at least w_{ij} time units after s_i, the start time of T_i.

The backward edge from node T_j to node T_i with the negative weight w_{ji} indicates that s_j must be no more than $|w_{ji}|$ time units after s_i. Therefore, each negative weight w_{ji} represents d_j, the deadline of T_j, such that $d_j = s_i + |w_{ji}| + p_j$. So-called *limited graph* can be obtained by removing all backward edges from graph G. Limited graph is acyclic.

In this paper we are concerned with monoprocessor non-preemptive scheduling. Therefore if s_i is the start time of task T_i then no other task can be scheduled before $s_i + p_i$ time units. The precedence constraints represented in G by the forward edges allow computing of earliest start times of the tasks. Further we assume $w_{ij} \geq p_i$ for each couple of tasks T_i and T_j. This assumption is not restrictive at all in a monoprocessor context since the tasks must be serialized. Moreover it adds the possibility to specify the *precedence delays* when $w_{ij} > p_i$.

The scheduling problem is to find a *feasible schedule*, satisfying the timing constraints given by G, while minimizing makespan C_{max}. Let S be a schedule given as a vector $S = (s_1, s_2, \dots s_n)$ such that each couple of nodes T_i and T_j with nonzero edge w_{ij} has the start times satisfying equation

$$s_j - s_i \geq w_{ij}. \qquad (1)$$

Equation (1) holds for both, forward and backward edges. Let $\mathcal{T}$ be the set of n tasks to be scheduled. The processing time vector $\mathbf{p} = (p_1, p_2, \dots, p_n)$ and $n \times n$ dimensional matrix of weights $\mathbf{W}$ are input parameters of the addressed problem. Matrix $\mathbf{W}$ is composed of timing constraints w_{ij} related to edges e_{ij} (between tasks T_i and T_j). There is no edge from the node T_i to the node T_j when $w_{ij} = 0$. There is forward edge when $w_{ij} > 0$ and there is backward edge when $w_{ji} < 0$, and $w_{ii} = 0$ by definition.

Example: One instance of the scheduling problem containing five tasks T_1,T_2,. . .,T_5 is given in Figure 1. Execution times are $\mathbf{p} = (1, 3, 2, 4, 5)$ and delay between start times of tasks T_1 and T_5 have to be less then or equal to 10 ($w_{5,1} = -10$). Corresponding matrix $\mathbf{W}$ of weights is shown in next Figure.

$$\mathbf{W} = \begin{pmatrix} 0 & 2 & 1 & 3 & 0 \\ 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 4 \\ -10 & 0 & 0 & 0 & 0 \end{pmatrix}$$

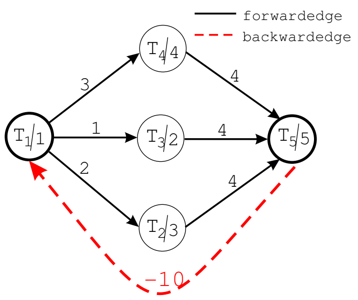

Figure 1: An example of problem given as matrix of weights $\mathbf{W}$ and corresponding graph G.

3 Solution by Branch and Bound Algorithm

Process of creating schedule S defines a partitioning of the set of tasks $\mathcal{T}$ into three disjoint subsets $\mathcal{T}_S(S)$, $\mathcal{T}_C(S)$ and $\mathcal{T}_R(S)$, where $\mathcal{T}_S(S)$ is the set of scheduled tasks, $\mathcal{T}_C(S)$ is the set of *candidate* tasks and $\mathcal{T}_R(S)$ is the set of remaining tasks.

A task T_k is a candidate to be scheduled into the partial schedule S if T_k has not been scheduled yet and all its predecessors belong to $\mathcal{T}_S$, the set of scheduled tasks.

$$T_k \in \mathcal{T}_C \Leftrightarrow T_k \notin \mathcal{T}_S \text{ and } T_i \in \mathcal{T}_S$$
$$\text{for all } i \text{ such that } w_{ik} > 0 \qquad (2)$$

If the node $T_k \in \mathcal{T}_C$ is chosen to be scheduled at time h, it is added to current partial schedule S with the start time equal to the maximum of the current time h and all its precedence timing constraints

$$s_k = max(h, \max_i(w_{ik} + s_i))$$
$$\text{for all } i \text{ such that } w_{ik} > 0. \qquad (3)$$

For a given partial schedule S an edge e_{ji} is called *activated backward edge* if and only if $w_{ji} < 0$, $T_i \in \mathcal{T}_S$ and $T_j \notin \mathcal{T}_S$. After scheduling task T_k sets $\mathcal{T}_S(S)$, $\mathcal{T}_C(S)$ and $\mathcal{T}_R(S)$ have to be actualized.

The scheduling problem can be solved by enumeration of a finite set F of *feasible solutions* respecting timing constraints and calculation of the criterion function $C_{max} : F \rightarrow N$ with intention to find a particular solution $S^* \in F$ such that

$$S^* = \arg\min_{S \in F} \left(C_{max}(S) \right). \qquad (4)$$

Branch and Bound (B&B) method is one of the enumeration methods considering certain solutions only indirectly, without actually evaluating them explicitly (e.g., when some partial solution does not lead to the optimal solution S^*). As its name implies, the B&B method consists of two fundamental procedures: branching and bounding. Branching is the procedure of partitioning a large problem into two or more mutually exclusive sub-problems. Furthermore the sub-problems can be partitioned in similar way, etc. Bounding calculates a lower bound on the optimal solution value C_{max} for each sub-problem generated in the branching process.

The branching procedure can be conveniently represented as a search tree. At level 0, the search tree consists of a single vertex representing the original problem. Each vertex in the n-th level represents one *final solution.* All n tasks are scheduled in final solution. All vertices in levels 1 to $n-1$ correspond to *partial solution* representing uncompleted schedule.

In order to implement the scheme of the branch and bound algorithm for our scheduling problem, one must first describe the branching procedure and the search strategy. Very simple recursive procedure creating the search tree of feasible solutions (see Equation (1)) can be stated as follows:

```
B&B algorithm:

1. [Initialisation]
   • Set $s_i = \infty \ \forall \ T_i \in \mathcal{T}$.
   • $\mathcal{T}_S = \emptyset$.
   • Find $\mathcal{T}_C$, the set of schedulable tasks (tasks without predecessors).
   • $S^B = S$, the best known final solution.
2. [Recursion] Call recursive procedure
   $vertex\_exploration(\mathcal{T}_S, \mathcal{T}_C, S)$.
```

```
Recursive procedure
$vertex\_exploration(\mathcal{T}_S, \mathcal{T}_C, S)$

1. [Bounding] Will be explained later in chapter "Bounding in the Search Tree". If the solution is feasible, then go to step 2, otherwise go to step 4.
2. [Test the solution]
   (a) If $\mathcal{T}_C = \emptyset$, the current solution $S$ is final solution. Assign the current solution $S$ to $S^B$ if $C_{max}(S) < C_{max}(S^B)$. Go to step 4.
   (b) If $\mathcal{T}_C \neq \emptyset$, the current solution is partial solution and then go to the step 3.
3. [Scheduling of candidates] For each candidate
   $T_k \in \mathcal{T}_C$ do:
   • Schedule the task $T_k$ by creating $S^N$ such that $s_i^N = s_i$ for all $i \neq k$ and $s_k$ is calculated using Equation (3).
   • Create new sets $\mathcal{T}_C^N$ and $\mathcal{T}_S^N$ such that $\mathcal{T}_S^N = \mathcal{T}_S \cup T_k$ and $\mathcal{T}_C^N$ contains tasks satisfying Equation (2).
   • call recursive procedure
     $vertex\_exploration(\mathcal{T}_S^N, \mathcal{T}_C^N, \mathcal{S}^N)$.
4. [Return]
```

Example: (continued) Complete set of solutions and partial solutions arranged in the search tree can be found in Figure 2. The leaves in the search tree are the final solutions. The non-feasible solutions are crossed out. In our case there is the only one feasible final solution $S = (0, 3, 1, 6, 10)$ and therefore this solution is optimal, $S = S^*$.

3.1 Bounding in the Search Tree

The aim of this chapter is to find a bounding mechanism reducing the size of the search tree (used in the step 1 of *vertex_exploration* procedure). At the same time we cannot eliminate any vertex on the unique path from the root to S^*, the optimal solution.

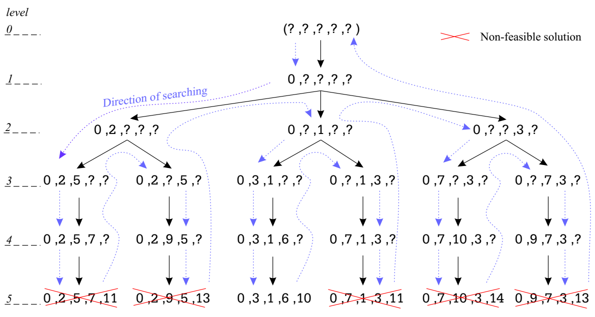

Figure 2: Illustration of the branching procedure.

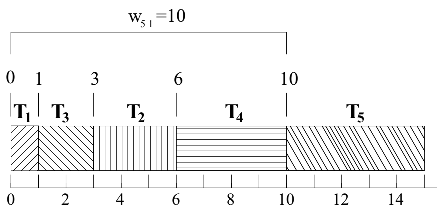

Figure 3: Optimal solution from example 1.

The *basic bounding*, tests only whether the solution is feasible i.e. whether Equation (1) is satisfied. In fact Equation (1) does not need to be checked for forward edges, since the scheduled task always satisfies the timing constraints related to forward edges (see step 3 of *vertex_exploration procedure*). On the other hand in the case of backward edges it is needed to check whether unscheduled tasks have not missed their latest possible starting time. Since the starting time of the unscheduled task is not known, the value of s_j in Equation (1) is substituted by h the current time. Therefore if $s_i - h \geq w_{ji}$ is satisfied for all activated backward edges, the corresponding solution is feasible. Otherwise it is needed to test, whether this order of tasks with shifted start times is feasible due to the scheduling anomaly as explained in 3.2.

Moreover, the basic bounding can be extended. The following text describes two bounding methods *Critical Path Bounding* and *Remaining Processing Time Bounding* reducing the number of search steps of B&B algorithm. These methods are based on $\widetilde{s}_j$, the estimated lower bound of s_j. This is useful when the solution cannot be eliminated by basic bounding but it can be eliminated since the value of the current time plus the time needed to complete some tasks is greater than the deadline. Both methods are implemented as extension of the step 1 in *vertex_exploration procedure*.

3.2 Scheduling Anomaly

The branching mechanism determines only order of tasks scheduled as soon as possible according to Equation (3). This approach satisfies feasibility only with respect to forward edges and the minimum C_{max} of the schedule. Unfortunately, this mechanism may result in a scheduling *anomaly* in constructed schedule with respect to feasibility given by backward edges. Let us consider a backward edge e_{ij} from task T_j to T_i. If *lateness* $L_j = s_i - s_j - w_{ji}$ of task T_j is greater then zero (tasks T_j missed its deadline by L_j ticks) then it is needed to test whether task T_i can be scheduled L_j ticks latter.

The test is performed by shifting T_i by L_j and by recalculation of start times of other scheduled tasks while Equation (3) is used to satisfy feasibility with respect to forward and backward edges. This "shifting" can cause increase of s_j by value in $< 0, L_j >$. If this value is 0, "shifting" does not increase the start time of T_j, the solution is feasible and branching procedure can continue with recalculated schedule S^N. Decision, whether for given unfeasible schedule S there exists feasible schedule S^N such that S^N has the same order of tasks as S, can be done either by procedure *shifting* outlined in Appendix A or it can be formulated as LP problem solvable in polynomial time.

3.3 Critical Path Bounding

In this method calculation of the estimation $\widetilde{s}_j$ is based on $n \times n$ matrix $\mathbf{F}$ of the longest paths in the limited graph. Floyd's algorithm is used to calculate $\mathbf{F}$ once at the initialization step of B&B algorithm. Therefore f_{ij} denotes the longest path by forward edges from node T_i to node T_j. If there is no path form node T_i to node T_j then $f_{ij} = \infty$.

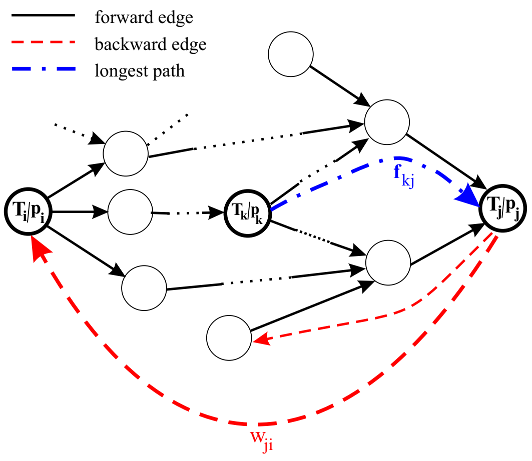

Figure 4: The critical path of an active backward edge e_{ji} with a weigth w_{ji}.

Figure 4 Illustrates the situation related to a partial schedule. When step 1 of the *vertex_exploration procedure* is performed for the last scheduled node T_k then we need to calculate $\widetilde{s}_j$. It is performed for each activated backward edge e_{ji} if there is a path by forward edges from T_k to T_j.

$$\widetilde{s}_j(S) = f_{kj} + s_k \quad \text{for each activated backward edge } e_{ji} \text{ such that } f_{kj} \neq \infty. \tag{5}$$

Then the feasibility of the current partial solution is checked while using adapted Equation (1) for all $\widetilde{s}_j(S)$ computed in Equation (5):

$$s_i - \widetilde{s}_j \geq w_{ji}. \tag{6}$$

The partial solution does not lead to any feasible solution if Equation (6) is not satisfied. Moreover also its predecessor in the search tree can be *eliminated indirectly* since it does not lead to any feasible solution. This follows from the fact that if any of the tasks exceeds its deadline related to the activated backward edge at level k (i.e., this task is at k-th position in the schedule), it will certainly exceed its deadline if it is scheduled later. This observation was already used in other scheduling algorithms [1, 2].

Nevertheless, presence of anomaly caused by backward edges must be also tested before solution elimination. Since task T_j is not scheduled yet, the value of its backward edge e_{ji} decreased by f_{kj} is used to impose a new constraint given by inequality

$$s_i - s_k \geq w_{ji} + f_{kj}. \tag{7}$$

3.4 Remaining Processing Time Bounding

The second method calculates the estimation $\widetilde{s}_j$ in a different way. It considers amount of work that must be done before the deadline. If there are many parallel edges in G, the longest path can be rather short in contrast to the processing time of the task needed to be executed. *Critical Path Bounding* method takes into consideration the depth of the graph G related to the weights of forward edges. On the other hand *Remaining Processing Time Bounding* method considers also the width of G, but due to the parallel branches it has to deal only with the processing time and not with the weighs of forward edges (please notice that p_i is less restrictive than w_{ij} - see chapter "Formulation of the Scheduling Problem"). In this sense the two methods have orthogonal influence on the branching process.

Computation of $\widetilde{s}_j$ begins with determination of $\mathcal{T}_P$, the set of nonscheduled tasks T_l such that $f_{lj} \neq \infty$ for all $l \neq j$. The set $\mathcal{T}_P$ and value of $\widetilde{s}_j$ are defined per each activated backward edge w_{ji}

$$T_l \in \mathcal{T}_P \Leftrightarrow f_{lj} \neq \infty \text{ for all } l \neq j \text{ and } T_l \notin \mathcal{T}_S. \tag{8}$$

$$\widetilde{s}_j(S) = h + \sum_{l \in \mathcal{T}_P} p(l) \quad \text{for all activated backward edges } e_{ji} \tag{9}$$

The node T_j is not included in $\mathcal{T}_P$ since the deadline given by the correspondent backward edge is related to the start time.

The estimation $\widetilde{s}_j$ is used in the same way like in chapter "Critical Path Bounding" but *Remaining Processing Time Bounding* method cannot be used to eliminate the predecessors indirectly in the search tree. This follows from the fact that the estimation $\widetilde{s}_j$ can be smaller for another solution at level k with the same predecessor in the search tree.

When both methods are applied in the B&B algorithm, the *Critical Path Bounding* method is applied before *Remaining Processing Time Bounding* method since the first one can eliminate indirectly the predecessor in the search tree and therefore its bounding has influence on more solutions.

Similarly to (7), presence of anomaly caused by a backward edge must be also tested before solution elimination. Since task T_j is not scheduled yet, the value of its backward edge e_{ji} decreased by $(p_k + \sum_{l \in \mathcal{T}_P} p(l))$ is used to impose a new constraint given by inequality

$$s_i - s_k \geq w_{ji} + p_k + \sum_{l \in \mathcal{T}_P} p(l). \tag{10}$$

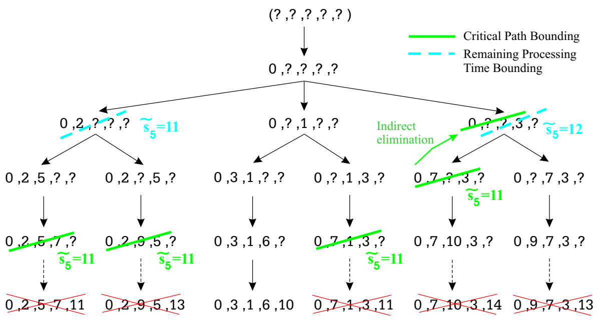

Figure 5: Illustration of the bounding procedure.

Example: (continued) The two bounding methods are demonstrated in Figure 5. The *Critical Path Bounding* method eliminates five partial solutions including $(0, ?, ?, 3, ?)$ eliminated indirectly and solution $S = (0, 7, 1, 3, ?)$ that cannot be eliminated by the second method. In this particular instance *Remaining Processing Time Bounding* method is very efficient. It eliminates two partial solutions at level 2 including $(0, 2, ?, ?, ?)$ that cannot be eliminated by the first method.

3.5 Using C_{max} for Bounding

Discussed bounding methods reduce state space by elimination of unfeasible solutions. But the final solution of the algorithm has to be not only feasible but in addition it has to minimize C_{max}. Hence searching in the tree can be bounded using C_{max} too. If current time h of a partial solution is smaller than $C_{max}(S^B)$ (the best known final solution) then this partial solution can lead to a final solution S_F with smaller $C_{max}(S_F)$. Otherwise the corresponding vertex can be eliminated. Formally the partial solution is not eliminated if

$$h < C_{max}(S^B). \tag{11}$$

Since the current time is equal to the start time plus processing time of the latest scheduled task T_j

$$s_j \leq C_{max}(S^B) - p_j. \tag{12}$$

The right side of this inequality can be considered as a new backward edge with $w_{ji} = C_{max}(S^B) - p_j$ from T_j to T_i. Under condition that $s_i = 0$ previous Equation can be convert to

$$s_i - s_j \geq w_{ji}. \tag{13}$$

Inequality (13) is identical to inequality (1). This reasoning is similar to the basic bounding mentioned in section "Bounding in the Search Tree". Moreover, it can be extended by estimation of $C_{max}(S_F)$, computed similarly as $\widetilde{s}_j$ estimation using *Critical Path Bounding* and *Remaining Processing Time Bounding*.

This way of bounding can by easily included into the current algorithm by modification of original graph G defined in chapter "Formulation of the Scheduling Problem". Since original graph G can have more source nodes and more sink nodes, it is needed to modify it to G' by adding two new nodes. The first one, called input node, is a unique source node in limited graph G' and the second one, called output node, is a unique sink node in limited graph G'. Consequently all source nodes in original limited graph G are successors of the input node, and all sink nodes in original limited graph G are predecessors of the output node. Thereafter, bounding by $C_{max}(S^B)$ is a kind of new dynamic backward edge from input node to output node.

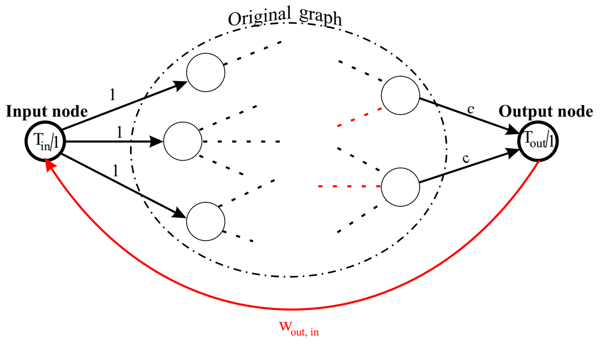

Figure 6: Construction of G' by adding dynamic backward edge weighted $-C_{max}$ into original graph.

Let the modified problem and all its related variables are marked prime. Then vector $\mathbf{p}$ and matrix $\mathbf{W}$ have to be modified as

$$\mathbf{p}' = [1, \mathbf{p}, 1] \qquad \mathbf{W}' = \begin{pmatrix} 0 & \mathbf{u} & 0 \\ 0 & \mathbf{W} & \mathbf{v} \\ w_{in,out} & 0 & 0 \end{pmatrix}. \tag{14}$$

Dimensions of $\mathbf{p}'$ and $\mathbf{W}'$ are increased by two $(n+2)$. u, the row vector of dimension n, represents connection of the input node to the source nodes of the original limited graph (u_m is equal to 1 if T_m is the source node in the original limited graph, otherwise it is equal to zero). Similarly v, the column vector of dimension n, represents connection of the sink nodes of the original limited graph to the output node (v_m has a constant value c if T_m is the sink node in original limited graph, otherwise it is equal to zero). Due to assumption made for w_{ij} in the section "Formulation of the Scheduling Problem" the constant value c is equal to maximum of the processing times of sink nodes in the original limited graph. Dynamic value of $w_{out,in}$ is initialized to the upper margin of C_{max} increased by one (processing time of source node is 1). The upper margin of C_{max} can by either given by user due to its expert knowledge or it can be calculated some heuristic [6].

Therefore original B&B algorithm has only two slight modifications:

- $\mathbf{p}'$ and $\mathbf{W}'$ are computed in the initialization step.
- The value of $w_{out,in}$ in matrix $\mathbf{W}'$ must be actualized in the step 2 of the *vertex_exploration procedure* if some new better feasible solution S' with $C_{max}(S')$ is found. The new $w_{out,in}$ is

$$w_{in,out} = C_{max}(S') - 1 - 1. \tag{15}$$

Value $C_{max}(S')$ is decreased two times because first we have to subtract the processing time of output node, and second decrement causes the B&B algorithm to look for a better solution (this transform of $\geq$ to $>$ is possible due to the integer values of the problem parameters).

When B&B algorithm finds the best solution $S^{*\prime}$, the S^* is extracted from $S^{*\prime}$ so that the first and the last tasks are removed from the schedule and each start time in the schedule is decremented (processing time of input node is one).

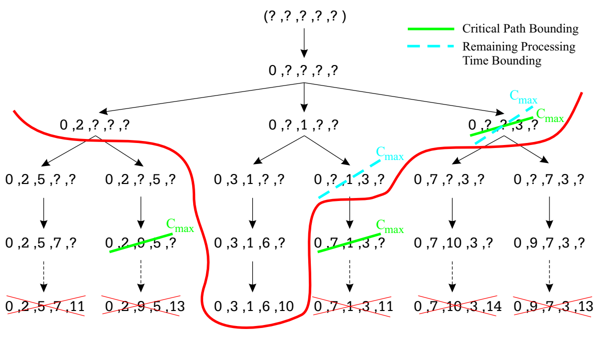

Figure 7: Illustration of the bounding procedure with C_{max} dynamic backward edge.

Example: (continued) In Figure 7 the example is extended by dynamic C_{max} backward edge. The "C_{max}" labels are new bounds issued by the modification of the original problem. The improvement is not so evident in this small example, but the reduction is very efficient in the case of more complex problems. Using of C_{max} bounding method is not very efficient at beginning of B&B algorithm. But when the optimal solution (or solution with C_{max} close C^*_{max}) is found, C_{max} bounding becomes very efficient.

4 Solution by ILP

Due to the problem NP–hardness it is meaningful to formulate our scheduling problem as problem of ILP, since various ILP algorithms solve instances of reasonable size in reasonable time. The schedule has to obey two constraints. The first is *precedence constraint* restriction corresponding to Inequality (1).

Each edge represents one precedence constraint. Hence, we have n_e inequalities (n_e is the number of edges in graph G).

The second kind of restrictions are *processor constraints.* They are related to the monoprocessor restriction, i.e., at maximum one task is executed at a given time. Two disjoint cases can occur. In the first case, we consider task T_i to be followed by task T_j (see Figure 8). Corresponding constraint is therefore

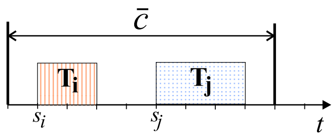

Figure 8: Processor constraint illustration example. T_i and T_j are tasks without precedence constraint ($p_i = 2, p_j = 3$) with start times $s_i = 1$, $s_j = 5$. An upper margin of C_{max} is $\overline{C} = 9$.

$$p_i \leq s_j - s_i. \tag{16}$$

In adition difference $s_j - s_i$ is certainly

$$s_j - s_i \leq \overline{C} - p_j \tag{17}$$

where $\overline{C}$ is a upper margin of C_{max}, calculated by some polynomial algorithm. The conjunction of (16) and (17) in to one double–inequality is

$$p_i \leq s_j - s_i \leq \overline{C} - p_j. \tag{18}$$

In the second case, we consider task T_j to be followed by task T_i. To derive constraints for the second case, it is enough to exchange index i with index j in the double–inequality (18)

$$\begin{aligned}
p_j \leq &\quad s_i - s_j &\leq \overline{C} - p_i, \\
p_j - \overline{C} \leq &\quad s_i - s_j - \overline{C} &\leq -p_i, \\
p_i \leq &\quad s_j - s_i + \overline{C} &\leq \overline{C} - p_j.
\end{aligned} \tag{19}$$

Exclusive OR relation between first case and second case, i.e., either (18) holds or (19) holds, disables to formulate the problem directly as an ILP program, since there is AND relation among all inequalities in ILP program.

The first case, constrained by (18), differs from the opposite second case, constrained by (19), only in $\overline{C}$ in the middle of double–inequality. This term signals whether T_i is before T_j or not. Therefore (18) and (19) can be reduced into one double–inequality, while using binary decision variable x_{ij} ($x_{ij} = 1$ when T_j is followed by T_i and $x_{ij} = 0$ when T_i is followed by T_j)

$$p_i \leq s_j - s_i + \overline{C} \cdot x_{ij} \leq \overline{C} - p_j. \tag{20}$$

To derive feasible monoprocessor schedule, double–inequality (20) must hold for each unordered couple of two distinct tasks. Therefore, there are $(n^2 - n)/2$ double–inequalities specifying the processor constraints at maximum. This restriction between tasks T_i and T_j is redundant when there is a path in limited graph from T_i to T_j or from T_j to T_i since order of tasks is determined by precedence constraints.

Makespawn minimization is realized by adding one variable C_{max} satisfying

5 backward edges										
#	# inspected vertices					CPU time [s]				
nodes	B&B1	B&B2	B&B3	B&B4	ILP	B&B1	B&B2	B&B3	B&B4	ILP
8	19.0	16.4	17.2	15.7	5.6	0.014	0.014	0.023	0.027	0.0016
10	99.0	17.1	75.4	36.8	28.2	0.086	0.060	0.099	0.071	0.0022
12	920.6	390.9	457.4	100.5	151.0	1.120	0.348	0.629	0.214	0.0112
14	11551.0	2307.0	3171.0	456.0	695.0	16.913	2.516	5.303	1.115	0.0604
16	65536.0	23418.0	25311.0	1595.0	5709.0	97.526	22.560	36.362	4.418	0.5930

10 backward edges										
#	# inspected vertices					CPU time [s]				
nodes	B&B1	B&B2	B&B3	B&B4	ILP	B&B1	B&B2	B&B3	B&B4	ILP
8	18.0	14.8	15.2	14.3	5.7	0.014	0.015	0.028	0.032	0.0025
10	50.1	32.9	36.3	25.4	19.4	0.048	0.036	0.069	0.063	0.0041
12	406.8	192.0	242.8	76.3	88.8	0.456	0.190	0.473	0.196	0.0091
14	4614.0	1222.8	1558.9	215.4	467.6	6.528	1.338	3.184	0.578	0.0437
16	30611.0	10077.0	12429.0	750.0	1366.0	42.611	10.737	24.175	2.278	0.1482

20 backward edges										
#	# inspected vertices					CPU time [s]				
nodes	B&B1	B&B2	B&B3	B&B4	ILP	B&B1	B&B2	B&B3	B&B4	ILP
8	16.4	13.4	13.8	13.3	4.32	0.016	0.017	0.038	0.042	0.0016
10	36.7	19.0	50.5	18.4	8.4	0.043	0.024	0.062	0.058	0.0022
12	299.9	62.3	94.8	45.6	48.2	0.421	0.088	0.295	0.154	0.0075
14	2917.8	208.7	310.6	96.7	88.6	4.927	0.312	0.993	0.341	0.0131
16	21074.0	494.0	760.0	241.0	337.0	40.515	0.788	2.515	0.918	0.0421

Table 1: Experimental results of scheduling algorithm complexity given by the number of inspected vertices and CPU time (mean value over fifty randomly generated set of input data). B&B1 - All feasible solutions. B&B2 - Critical Path Bounding, B&B3 - Remaining Processing Time Bounding, B&B4 - All methods together, ILP - Integer linear programming.

$$s_i + p_i \leq C_{max}, \quad \forall T_i \in \mathcal{T} \tag{21}$$

for all sink nodes of limited graph. Then the objective function is to minimize C_{max}. The summarized ILP program, using variables s_i, x_{ij}, C_{max}, is shown in Figure 9.

$$\min C_{max}$$

subject to

$$s_j - s_i \geq w_{ij}, \qquad \forall e_{ij} \in G$$

$$p_i \leq s_j - s_i + \overline{C} \cdot x_{ij} \leq \overline{C} - p_j,$$
$$\forall i \neq j \text{ and } T_i, T_j \in \mathcal{T} \text{ and } (f_{ij} < \infty \vee f_{ji} < \infty)$$

$$s_i - C_{max} \leq -p_i, \qquad \forall T_i \in \mathcal{T} \text{ sink node}$$

where

$s_i \in \langle 0, \overline{C} - p_i \rangle$, $x_i \in \langle 0, 1 \rangle$, $C_{max} \in \langle 0, \overline{C} \rangle$
s_i, x_{ij} are integers.

Figure 9: ILP program.

5 Experimental Results

Presented scheduling algorithms were implemented and tested in Matlab language (including procedure *shifting*). Integer linear program was solved by ILP solver tool LP_SOLVE 4.0 [7] called from Matlab as an external DLL library.

CPU times are hardly comparable, since Matlab is an interpreter and LP_SOLVE tool is compiled code. ILP uses also a Branch and Bound algorithm while solving linear programs in each vertex of searching tree. Therefore, comparison of number of processed vertices allows to examine results of both methods. Total number of processed vertices of ILP in Table 1 is given by value of variable *total_nodes* declared in lpkit.h of LP_SOLVE tool.

Figure 10 shows algorithms complexity in the terms of inspected solutions depending on the number of nodes in G. Fifty scheduling problems have been generated per each number of nodes (8,10,12,14 or 16) and the mean value of number of inspected solutions in each scheduling problem is shown in Figure 10. The scheduling problems were generated at random manner so that the number of forward edges was 3/2 of the number of nodes and there was fixed number of backward edges (5,10 or 20).

When all bounding methods are combined together (B&B4 in Table 1) the number of inspected states is approximately 81% of all feasible solutions for 8 nodes (with 20 backward edges) and it is 1.1% of all feasible states for 16 nodes. ILP inspects 26% of all feasible schedules in the case of 8 nodes and 20 backward edges and 1,6% in the case of 16 nodes with the same number of backward edges.

Implementation note: The computing time of scheduling algorithm in seconds depends on the programming language and the computer performance. The algorithm was tested on a PC Intel Pentium 4, 2.4GHz in Matlab 5.3 environment. Enabling of all bounding methods caused deceleration of the average vertex processing speed (approximately twice). But with respect to the experimental results (shown in Figure 10 - notice logarithmical scale of vertical scale) in the CPU computing time of B&B4 is much lower then one of B&B1.

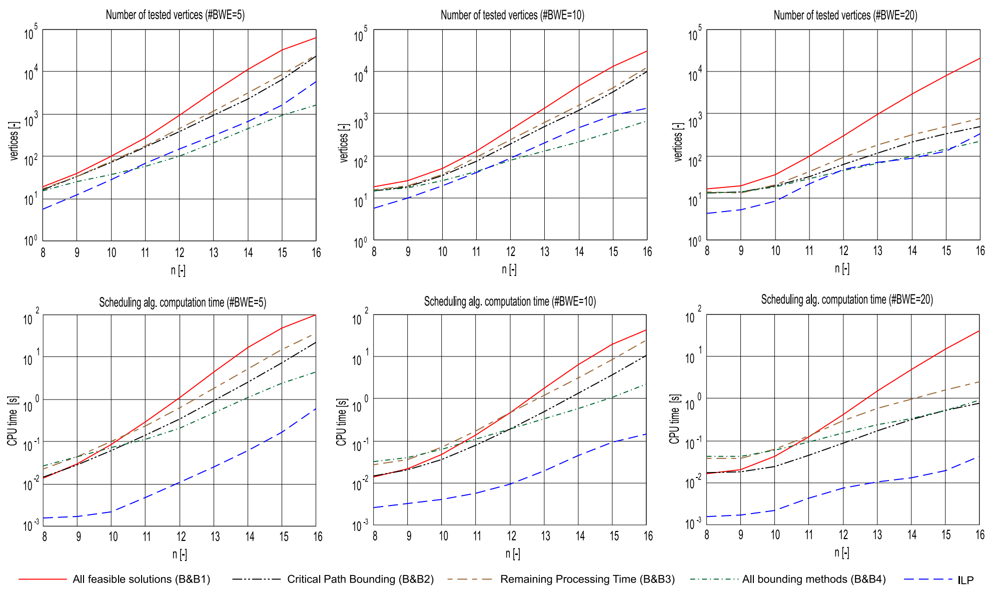

Figure 10: Diagram representation of experimental results (identical to Table 1).

6 Conclusion and Future Work

This paper presents two original solutions for the scheduling problem with start time related deadlines, which has been shown to be NP–hard. The first is implemented as the B&B algorithm using the Critical Path estimation and estimation of Remaining Processing Time. Since the objective is to find a feasible schedule with minimal C_{max}, the bounding procedure uses the best known solution as a new dynamic timing constraint. It considers also scheduling anomaly while deciding feasibility of given solution. The second, formulated as ILP, uses elegant way of processor constraint specification. Experimental results show impressive power of ILP.

It is an alternative to the scheduling problem presented in [6], which is based on heuristic algorithms. Our suggested solutions have not polynomial complexity but they are able to find the optimal solution. Further, we considered multiple backward edges. Our suggested solutions have higher efficiency if the backward edges are very restrictive. In a specific case, when the scheduling instance is too complex, a restrictive backward edge can be given by the user due to its expert knowledge in order to reduce the scheduling algorithm computing time.

In a mean time we have shown that the presented problem can be generalized for cyclic scheduling problems [13], used for FPGA design of RLS (Recursive Least Squares) filter.

In the future we would like to compare our solutions with [3].

References

[1] J. Błazewicz, K. Ecker, G. Schmidt and J. Węglarz. *Scheduling Computer and Manufacturing Processes.* Springer, second edition, 2001.

[2] P. Bratley, M. Florian and P. Robillard. Scheduling with earliest start and due date constraints. Naval Res. Logist. Quart. 18, 1971.

[3] P. Brucker, T. Hilbig and J. Hurink. A branch and bound algorithm for a single-machine scheduling problem with positive and negative time-lags. Discrete Applied Mathematics 94, 1999.

[4] G. C. Buttazzo. *Hard Real-Time Computing Systems.* Kluwer Academic Publishers, 1997.

[5] J. Hurink and J. Keuchel. Local search algorithms for a single-machine scheduling problem with positive and negative time-lags. Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science, 2001.

[6] M. Jacomino, D. Gutfreund and J. Pulou. Scheduling real-time processes with timing constraints and its applications to cyclic systems. Report internal, 1999.

[7] J. C. Kantor. *LP_SOLVE 4.0.* ftp://ftp.es.ele.tue.nl/pub/lp_solve/, 1995.

[8] A. M. Kordon. Minimizing the makespan for a UET bipartite graph on a single processor with an integer precedence delay. Rapport de Recherche LIP6, 2001.

[9] J. K. Lenstra, A. H. G. Rinnoy Kan and P. Brucker. Complexity of machine scheduling problems. Ann. Discrete Math. 1, 1977.

[10] J. W. S. Liu. *Real-Time Systems.* Prentice-Hall, Inc., 2002.

[11] R. Matoušek, Z. Pohl, J. Kadlec, M. Tichý and A. Heřmánek. Logarithmic arithmetic core based RLS LATTICE implementation. Design, Automation and Test in Europe DATE 02. (Sciuto D.,

Kloos C. D. eds.). IEEE, Los Alamitos, pp. 271., 2002.

[12] C.-S. Shih and J. W. S. Liu. State-dependant deadline scheduling. 23rd IEEE REAL-TIME SYSTEMS SYMPOSIUM, 2002.

[13] P. Šůcha and Z. Hanzálek. Scheduling of iterative algorithms on FPGA with pipelined arithmetic unit. Research Report, DCE, Czech Technical University in Prague, 2004.

Appendix

A Start time recalculation

$[feasible, S^N] = shifting(S)$

1. [Initialization] Create list $\mathcal{T}_U$ containing tasks from $\mathcal{T}_S$ ordered in non-decreasing order of start times. Assign $S^N = S$.

2. [Rescheduling]

   - Remove the last task T_k from list $\mathcal{T}_U$.
   - For each backward edge e_{ik} where $T_i \in \mathcal{T}_U$ calculate maximum lateness $L_k = \max_{e_{ij}}(s_k - s_i - w_{ik})$
   - if $L_k < 0$ then assign $s_k^N = s_k^N - L_k$. Recalculate start times of tasks belonging to $\mathcal{T}_S$ using Equation (3) and calculate new current time h^N. If $h^N > h$ then $feasible = FALSE$ and go to step 4. Else go to step 2.

3. [Test] If $S^N \neq S$ then $S = S^N$ and go to step 1. Else $feasible = TRUE$ and go to step 4.

4. [Return]