Scheduling of Tasks with Precedence Delays and Relative Deadlines - Framework for Time-optimal Dynamic Reconfiguration of FPGAs

Přemysl Šůcha, Zdeněk Hanzálek
Centre for Applied Cybernetics, Department of Control Engineering
Czech Technical University in Prague, {suchap,hanzalek}@fel.cvut.cz

***Abstract*— This paper is motivated by existing architectures of field programmable gate arrays (FPGAs) supporting runtime dynamic reconfiguration, further extended by on chip processor and internal SRAM memory. To facilitate the design process we present an optimal scheduling algorithm using very universal model, where tasks are constrained by precedence delays and relative deadlines. The precedence relations are given by an oriented graph, where tasks are represented by nodes. Edges in the graph are related either to the minimum time or to the maximum time elapsed between start times of the tasks. The NP-hard problem of finding an optimal schedule satisfying the timing and resource constraints while minimizing makespan C_{max}, is solved using two approaches. The first one is based on Integer Linear Programming and the second one is implemented as a Branch and Bound algorithm. Experimental results show the efficiency comparison of the ILP and Branch and Bound solutions. In addition our solutions are compared with existing heuristic approaches.**

***Index Terms* – Off-line scheduling, high–level synthesis, branch and bound, ILP, FPGA.**

I. Introduction

The studied problem is motivated by existing architectures of field programmable gate arrays (FPGAs) incorporating FPGA fabric, on chip processor, and internal SRAM memory. Some of these architectures support runtime dynamic reconfiguration allowing to change parts of the device, e.g. floating point units, while the rest operates at full speed. An example of such architecture is Xilinx Virtex2 (i.e. starter kit from Memec with XC2V1000-FG365-4) which contains the static part with the MicroBlaze 32-bit RISC soft processor and the dynamically reconfigurable part [1]. Another one, based on Atmel AT94K, contains the static part with the hard core of the AVR 8-bit microcontroller and the dynamic part with user defined designs [1], in our case different floating point units.

The *reconfigurable units* (i.e. corse grain modules computing some function) are usually pipelined. Therefore, they are characterized by the processing time p_i (associated with task T_i) representing the time to feed the unit (i.e. new data can be fed after p_i time units) and by the positive weight w_{ij} (associated with data produced by T_i) representing the input–output latency. The result of a computation, available after w_{ij} time units, is formalized as *precedence delay* in the scheduling theory.

For illustration of reconfigurable units, we may consider two types of arithmetic libraries operating with real numbers. The first type is the logarithmic number system arithmetic, namely the High-Speed Logarithmic Arithmetic (HSLA) library [2] implementing multiplication, division and square-root operations simply as fixed-point addition, subtraction and right shift. Addition and subtraction operations require more complicated evaluation, hence only one pipelined addition/subtraction unit is usually available for a given application. On the other hand, the number of multiplication, division and square root units can be nearly unlimited. For that reason, the scheduling of algorithms on HSLA (without reconfiguration) can be formalized as *scheduling of tasks with precedence delays and relative deadlines on one dedicated processor*. This problem will be called 1–DEDICATED in the rest of this article.

The second type is the floating-point library, namely FP32 by Celoxica [3]. It uses widely known IEEE format to store the data. In this case, each unit requires important number of hardware elements on the gate array, hence the only one unit of each kind is usually available for a given application. Scheduling of algorithms on FP32 can be formalized as *scheduling of tasks with precedence delays and relative deadlines on the set of dedicated processors*. This problem will be called m–DEDICATED and can be reduced to 1–DEDICATED problem.

The most important problem of dynamic reconfiguration, not seen in classical (i.e. static) design, is temporal interdependence of individual parts to be reconfigured. Scheduling algorithm have to distinguish, whether and when the reconfiguration is performed. When task T_j is processed immediately after task T_i and task T_j is processed by a different reconfigurable unit (i.e. currently not available in FPGA), processing of T_i is charged by *reconfiguration time*. This makes it almost impossible to manually produce efficient designs without proper methodology. For the simplicity, in this article we assume one unit to be present in reconfigurable part at a given time. The monoprocessor version of this problem can be formalized as *monoprocessor problem with changeover times*. We will show its reduction to m–DEDICATED problem and direct solution by ILP.

Not only the reconfigurable units, but also the memory units, become limited resources that have to be taken into consideration in many practical applications of FPGAs. In such case, some processing units and some memory units, may be required at one moment. This problem is formalized as *multiprocessor tasks scheduling*. We will also show its

reduction to m–DEDICATED problem and direct solution by ILP.

A. Related Work

Traditional off–line scheduling algorithms (e.g. [4]), typically assume that deadlines are absolute, i.e. the deadlines are related to the beginning of the schedule. On the other hand, when assuming, for example, interactions among on-chip processor, pipelined units and external environment, we may conveniently represent a given timing requirement by a deadline of the task T_j related to the start time of the task T_i. In such case deadlines cannot be calculated *a priory*, we use *relative deadlines*.

In connection with project planning, the concept of precedence delays and relative deadlines (called positive and negative time-lags) have been introduced by Roy [5] and further developed by Brucker et. all [6]. An heuristic solution [7] is based on local search algorithm. Another heuristic approach to the scheduling of signal processing algorithms with relative deadlines was studied in [8].

From the time complexity point of view, 1–DEDICATED problem is NP-hard, since the scheduling problem $1|r_j, \widetilde{d}_j|C_{max}$ [9] is reducible to it. Moreover, another NP-hard problems (e.g. pipeline scheduling problem presented in [10]) are also reducible to it. On the other hand, 1–DEDICATED problem is decidable, since it can be solved by ILP. Current works on monoprocessor scheduling with deadlines focus on minimization of number of tardy jobs [11], on minimization of penalties [12] and on minimization of weighted completion time [13].

In this paper we deal with time-optimal FPGA reconfiguration, i.e. with the scheduling issue. There are numerous works dealing with scheduling, placement issue or combining both of them. Heuristic algorithms for on–line scheduling and placement to partially reconfigurable devices have been proposed in [14]. An optimal algorithm and an approximation algorithm for the problem scheduling tasks without precedence relations on a fixed number of heterogeneous unrelated resources was shown in [15].

B. Contribution

This paper shows, how the time-optimal FPGA design with several relevant architectural features (pipelined units, FPGA fabric interaction with on–chip processor, arbitrary/restricted number of units, simultaneous memory access, dynamic reconfiguration) may be formalized as an off–line scheduling of non–preemptive tasks with precedence delays (minimum delay between two tasks) and relative deadlines (maximum delay between two tasks). It is shown that the problem is NP-hard by reduction of Bratley's scheduling problem $1|r_j, \widetilde{d}_j|C_{max}$. We adopt this simple but very universal model, we propose two original algorithms solving the scheduling problem and we evaluate their performance. The first algorithm is based on Branch and Bound (B&B) method and the second one on Integer Linear Programming (ILP).

The main contributions of this paper are:

- formalization of time-optimal FPGA design with pipelined units, on–chip processor, arbitrary/restricted number of dedicated units, simultaneous memory access and dynamic reconfiguration by the scheduling problem with precedence delays and relative deadlines
- original solution of m–DEDICATED problem by B&B algorithm
- original ILP based solution of m–DEDICATED problem
- direct ILP based solutions of multiprocessor tasks scheduling and multiprocessor problem with changeover times (i.e. without reduction to m–DEDICATED)
- performance evaluation of B&B solution, ILP solution and heuristic solution [8]

This paper is organized as follows. Section II presents formulation of m–DEDICATED problem and polynomial reductions of mentioned scheduling problems. Section III presents the solution of m–DEDICATED problem by ILP and direct solutions of multiprocessor tasks scheduling and multiprocessor problem with changeover times. The next section describes the B&B algorithm, solving m–DEDICATED problem, using several bounding mechanisms based on Floyd's algorithm. Finally experimental results and comparisons are summarized in Section V.

II. Problem Statement

A. Formulation of m–DEDICATED Problem

The set of n tasks $\mathcal{T} = \{T_1, \dots T_i, \dots T_n\}$ is constrained by the precedence relations, given by a task-on-node graph G. Each operation is represented by a task T_i corresponding to the node T_i in the graph G and has a non-negative 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 by 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 weight w_{ij}, associated to forward edge, specifies so called *precedence delays*. We use precedence delay to represent for example the input–output latency of the pipelined unit.

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 $\widetilde{d}_j(s_i)$, the deadline of T_j, such that $\widetilde{d}_j(s_i) = s_i + |w_{ji}| + p_j$. 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 non-preemptive scheduling on the set of m dedicated processors $\mathcal{P} = \{P_1, \dots P_d, \dots P_m\}$. The set of tasks $\mathcal{T}$ is a conjunction of disjoint sets $\mathcal{T}_1, \dots \mathcal{T}_d, \dots \mathcal{T}_m$. Both tasks T_i and T_j are assigned to dedicated processor P_d, if and only if, $T_i \in \mathcal{T}_d$ and $T_j \in \mathcal{T}_d$. Therefore if s_i is the start time of task T_i scheduled on processor P_d then no other task can be scheduled before $s_i + p_i$ time units on the same processor.

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

$$s_j - s_i \geq w_{ij}. \tag{1}$$

Equation (1) holds for both, forward and backward edges.

Let $\mathcal{T} = \{\mathcal{T}_1, \ldots \mathcal{T}_d, \ldots \mathcal{T}_m\}$ be the set of $n = n_1 + \ldots + n_d + \ldots + n_m$ tasks to be scheduled, where n_d is the number of tasks assigned to the dedicated processor P_d. The processing time vector $\mathbf{p} = (p_1, p_2, \ldots, p_n)$, $n \times n$ dimensional matrix of weights $\mathbf{W}$ and the sets of tasks running on dedicated processors $\mathcal{T}_1, \ldots \mathcal{T}_d, \ldots \mathcal{T}_m$ 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 $w_{ij} = -\infty$. There is forward edge when $w_{ij} > 0$ and there is backward edge when $w_{ji} < 0$, and $w_{ii} = -\infty$.

Example: One instance of the scheduling problem under consideration containing six tasks T_1,T_2,...,T_6 is given by the graph G and the corresponding matrix of weights $\mathbf{W}$ in Figure 1. The set of tasks $\mathcal{T}_1 = \{T_1, T_2, T_6\}$ is assigned to the first processor and $\mathcal{T}_2 = \{T_3, T_4, T_5\}$ to the second one. The processing times are $\mathbf{p} = (1, 3, 2, 3, 4, 5)$ and the precedence relations are given by the corresponding graph including backward edges (start time of task T_6 has to be at maximum 11 time units after start time of task T_1, and start time of task T_3 has to be at maximum 3 time units after start time of task T_1).

$$\mathbf{W} = \begin{pmatrix} -\infty & -\infty & 1 & 2 & 3 & -\infty \\ -\infty & -\infty & 4 & 4 & 4 & 13 \\ -3 & -\infty & -\infty & -\infty & -\infty & 4 \\ -\infty & -\infty & -\infty & -\infty & -\infty & 4 \\ -\infty & -\infty & -\infty & -\infty & -\infty & 4 \\ -11 & -\infty & -\infty & -\infty & -\infty & -\infty \end{pmatrix}$$

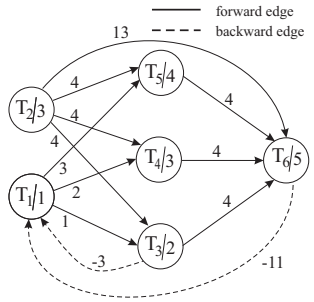

Fig. 1. An example of the scheduling problem.

B. Problem Complexity - Reduction of Bratley's problem to 1–DEDICATED

In order to facilitate a deduction, we will consider the monoprocessor case first. The monoprocessor scheduling problem with relative deadlines is NP–hard, since Bratley's scheduling problem $1|r_j, \widetilde{d}_j|C_{max}$ (monoprocessor scheduling with release dates, with deadlines and without precedence constraints) [9], [4] can be polynomially reduced to it. Therefore, each instance of Bratley's problem can be polynomially reduced to an instance of 1–DEDICATED problem.

The polynomial reduction is shown in Figure 2. The independent task set of Bratley's problem is represented by nodes $T_1, ..., T_n$ and their release dates and deadlines are represented

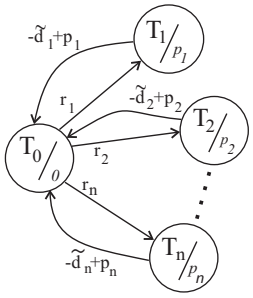

Fig. 2. Polynomial reduction of Bratley's problem $1|r_j, \widetilde{d}_j|C_{max}$ to 1–DEDICATED. Each independent task of the set $\mathcal{T} = \{T_1, T_2, ..., T_n\}$ is linked with a dummy task T_0 using precedence delays to specify the task's release date and deadline.

using precedence delays related to a dummy task T_0. The release date r_j of task T_j is represented by the forward edge e_{0j} from T_0 to T_j. Assuming $s_0 = s_i = 0$, Inequality (1) determines the restriction $s_j \geq r_j$, which is effectively the only restriction given by the release date.

Backward edges from T_i to T_0 with negative weight $w_{ji} = -\widetilde{d}_i + p_i$ represent deadlines. In the same way the deadline restriction, i.e. $s_i + p_i \leq \widetilde{d}_i$, is obtained from (1) for each of these edges assuming $s_0 = s_j = 0$.

We remind that Bratley's problem $1|r_j, \widetilde{d}_j|C_{max}$ was proven to be NP–hard by P–reduction from a 3–PARTITION problem [16]. The monoprocessor scheduling problem with precedence delays and relative deadlines (1–DEDICATED) is NP–hard, since each instance of Bratley's problem can be P–reduced to an instance of 1–DEDICATED as shown above. Further m–DEDICATED problem, scheduling of tasks with precedence delays and relative deadlines on the set of dedicated processors, is NP–hard, since the monoprocessor case (1–DEDICATED) is its subproblem. It is obvious that Bratley's problem extended by precedence relations, i.e. $1|r_j, prec, \widetilde{d}_j|C_{max}$, is reducible to 1–DEDICATED problem.

C. Reduction of m–DEDICATED to 1–DEDICATED

This subsection shows, how each task of m–DEDICATED problem can be transformed to an instance of 1–DEDICATED problem scheduled on one virtual processor. The most important property of problem formulation by tasks with precedence delays and relative deadlines is the universality. Brucker [6] has shown that many scheduling problems can be polynomially reduced to this problem and effectively solved as outlined in Subsections II-D and II-E.

The reduction from m–DEDICATED to 1–DEDICATED require an upper bound $\overline{C}$ of the C_{max}, that can be found e.g. as $\overline{C} = \sum_{i=1}^{n} \max\left(p_i, \max_{j \in \langle 1,n \rangle} l_{ij}\right)$. The main idea lies in assigment of all tasks on one virtual monoprocessor while using forward and backward edges to enclose $\mathcal{T}_d$ into d-th interval on the virtual processor. Let $T_i \in \mathcal{T}_{d_i}$ and $T_j \in \mathcal{T}_{d_j}$ then task T_i is assigned to the time interval $\langle (d_i - 1) \cdot \overline{C}, d_i \cdot \overline{C} \rangle$ on monoprocessor while recalculating its start time as $s'_i = s_i + (d_i - 1) \cdot \overline{C}$.

Then timing constraint (1) is translated to $s'_j - s'_i \geq w'_{ij}$, where $w'_{ij} = w_{ij} + (d_i - d_j) \cdot \overline{C}$. Further, enclosure of tasks $T_i \in \mathcal{T}_{d_i}$ into the interval $\langle (d_i - 1) \cdot \overline{C}, d_i \cdot \overline{C} \rangle$ is realized by introductin of a dummy task T_0 with $p_0 = 0$ and by introduction of the couple of new edges for each task T_i such as $s'_i - s'_0 \leq (d_i - 1) \cdot \overline{C}$ and $s'_0 - s'_i \leq d_i \cdot \overline{C}$.

Finally we add second dummy task T_{n+1} with $p_{n+1} = 0$ to formulate the C_{max} minimization of m–DEDICATED problem in terms of C_{max} minimization of 1–DEDICATED problem. Therefore for each task T_i we add one relation $s_i + p_i \leq s_{n+1}$.

D. Reduction of Problem with Multiprocessor Tasks to m–DEDICATED

The multiprocessor tasks scheduling problem is an extension of m–DEDICATED problem, where task T_i is associated with a set of processors $\mathcal{P}_i = \{P_i^1, \ldots, P_i^{m_i}\} \subset \mathcal{P}$. The reduction of this problem to m–DEDICATED problem is based on the propagation of each task T_i to the set of virtual tasks $T_i^1, \ldots T_i^v, \ldots T_i^{m_i}$. These tasks have the same processing time as T_i and each tasks T_i^v is assigned to virtual processor $P_i^v \in \mathcal{P}_i$. These tasks are forced to start at the same time by a couple of *synchronization relations*

$$s_i^v - s_i^1 \geq 0 \quad \text{and} \quad s_i^1 - s_i^v \geq 0. \tag{2}$$

Finally, we define a timing constraint (1) between tasks T_i^1 and T_j^1 if and only if there is a corresponding timing constraint between original tasks T_i and T_j.

E. Reduction of Monoprocessor Problem with Changeover Times to m–DEDICATED

This scheduling problem extends 1–DEDICATED problem by partitioning the set of tasks $\mathcal{T}$ into groups $G_1, \ldots, G_r$. If a task T_i from group G_k is scheduled immediately after a task T_j from different group G_l, there is a *reconfiguratin time* r_{lk} i.e. task T_i starts at the earliest r_{lk} time units after the finishing time of T_j, otherwise $r_{lk} = 0$.

This problem can be also polynomially reduced to m–DEDICATED scheduling problem. All tasks from $\mathcal{T}$ are processed on the first processor P_1. Further, for each pair T_i, T_j belonging to different groups G_k, G_l we introduce a virtual processor P_{ij}. To this virtual processor are dedicated two tasks T_{ij}^i and T_{ij}^j with processing time $p_{ij}^i = p_i + r_{kl}$ and $p_{ij}^j = p_j + r_{lk}$ respectively. By a couple of synchronization relations task T_{ij}^i and T_{ij}^j are forced to start at the same time as T_i and T_j on processor P_1, respectively.

Finally C_{max} minimization is given by C_{max} of processor P_1 that is reflected by introduction of a dummy tasks T_{n+1} in the similar way as in Subsection II-C.

Due to the above mentioned reductions we are able to formulate many practical scheduling problems on FPGA architectures with dynamic reconfiguration and limited resources as 1–DEDICATED problem. In the rest of the paper we show solution of this problem by ILP and B&B algorithm. In order to illustrate various features of our solutions, we develop the more complex one, i.e. m–DEDICATED, and in experimental section we compare performance of ILP and B&B solutions.

III. Solution of m–DEDICATED Problem by ILP

Due to NP–hardness of m–DEDICATED problem, it is meaningful to formulate it as problem of ILP, since various ILP algorithms solve instances of reasonable size in reasonable time. The schedule has to obey to two kinds of constraints. The first is *precedence constraint* restriction corresponding to Inequality (1). Since each edge represents one precedence constraint, we have n_e inequalities (n_e is the number of edges in graph G).

The second kind of restrictions are *processor constraints*, i.e. for each couple of tasks T_i and T_j assigned to the same processor, at maximum one is executed at a given time. Two disjoint cases can occur. In the first case, we consider task T_j to be followed by task T_i. In the second case, we consider task T_i to be followed by task T_j.

Exclusive OR relation between the first case and the second case disables to solve the problem directly by LP (Linear Programming), since there is AND relation among all inequalities in LP program. Therefore, we use a binary decision variable x_{ij} ($x_{ij} = 1$ when T_i is followed by T_j and $x_{ij} = 0$ when T_j is followed by T_i) and a very large positive number $\overline{C}$ ($\overline{C}$ is an upper bound of C_{max}).

Then the restrictions can be formulated as one double–inequality

$$p_j \leq s_i - s_j + \overline{C} \cdot x_{ij} \leq \overline{C} - p_i. \tag{3}$$

To derive a feasible schedule, double–inequality (3) must hold for each unordered couple of two tasks assigned to the same processor. Therefore, there are at maximum $\sum_{d=1}^{m} (n_d^2 - n_d)/2$ double–inequalities specifying processor constraints, where $n_d = |\mathcal{T}_d|$. The (3) is redundant for tasks T_i and T_j 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 as $w_{ij} \geq p_j$ for forward edges.

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

$$s_i + p_i \leq C_{max}, \quad \forall T_i \in \mathcal{T} \tag{4}$$

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 3.

$$\min C_{max}$$

subject to

$$\begin{array}{ll} s_j - s_i \geq w_{ij}, & \forall e_{ij} \in G \\ p_j \leq s_i - s_j + \overline{C} \cdot x_{ij} \leq \overline{C} - p_i, & \forall i < j \text{ and } T_i, T_j \in \mathcal{T}_d \\ s_i + p_i \leq C_{max}, & \forall T_i \in \mathcal{T} \text{ sink node} \end{array}$$

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.

Fig. 3. ILP program solving m–Dedicated scheduling problem.

A. Direct Solution of Problems with Multiprocessor Tasks and Changeover Times

The generality of ILP allows to formulate scheduling problems outlined in Subsections II-D and II-E directly without polynomial reduction that increases the number of nodes in both cases. Problem with multiprocessor tasks can be solved by ILP program in Figure 3 by a slight modification of the processor constraints (3). The processor constraint between T_i and T_j in m–DEDICATED problem is considered, when $T_i, T_j \in \mathcal{T}_d$, i.e. both tasks are dedicated to the same processor, otherwise it is omitted. Therefore, in problem with multiprocessor tasks, processor constraint between T_i and T_j is considered when $\mathcal{P}_i \cap \mathcal{P}_j \neq \emptyset$, i.e. tasks T_i and T_j can not overlap if they require the same processor.

The monoprocessor scheduling problem with changeover times can by also directly solved by a modification of the processor constraints (3). Since the processor constraint restricts the overlap only between a couple T_i and T_j then processing times p_i and p_j can be considered different for different couples of tasks, i.e. different processor constraints. Therefore, considering tasks T_i and T_j belonging to different groups G_k, G_l, the processing time used in processor constraint (3) for these tasks is considered $p^i_{ij} = p_i + r_{kl}$ and $p^j_{ij} = p_j + r_{lk}$, respectively. For tasks from the same group the processor constraint (3) stays the same.

IV. Solution of m–DEDICATED Problem by Branch and Bound Algorithm

The Branch and Bound algorithm creating schedule S defines a partitioning of $\mathcal{T}$, the set of tasks, 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 already 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.

$$T_k \in \mathcal{T}_C \Leftrightarrow (T_k \notin \mathcal{T}_S \text{ and } T_i \in \mathcal{T}_S \ \forall i \text{ such that } w_{ik} > 0) \quad (5)$$

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

$$s_k = max(h_d, \max_i(w_{ik} + s_i)) \quad \forall T_i \in \mathcal{T}_S \text{ such that } w_{ik} > 0. \quad (6)$$

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

$$S^* = \arg\min_{S \in F} (C_{max}(S)). \quad (7)$$

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 mechanism reduces the size of the search state space.

The branching procedure can be conveniently represented as a search tree (see Figure 4). 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 the final solution. All vertices in levels 1 to $n-1$ correspond to a *partial solution* representing an 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. A 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).
3) $S^* = S^B$

Recursive procedure: *vertex_exploration*($\mathcal{T}_S$, $\mathcal{T}_C$, S)

1) [Bounding] Will be explained later in Section IV-A. 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 a 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 a 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 $s^N_i = s_i$ for all $i \neq k$ and s_k is calculated using Equation (6).
   - Create new sets $\mathcal{T}^N_C$ and $\mathcal{T}^N_S$ such that $\mathcal{T}^N_S = \mathcal{T}_S \cup T_k$ and $\mathcal{T}^N_C$ contains tasks satisfying Equation (5).
   - call recursive procedure *vertex_exploration*($\mathcal{T}^N_S$,$\mathcal{T}^N_C$,$\mathcal{S}^N$).
4) [Return]

Example: (continued) Complete set of solutions and partial solutions arranged in the search tree can be found in Figure 4. The leaves in the 6-th level of the search tree are the final solutions. The non-feasible solutions are crossed out. In this example there are two feasible final solutions $S_1 = (3, 0, 4, 6, 9, 13)$ and $S_2 = (3, 0, 4, 10, 6, 14)$ therefore, the solution S_1 is optimal, i.e. $S^* = S_1$.

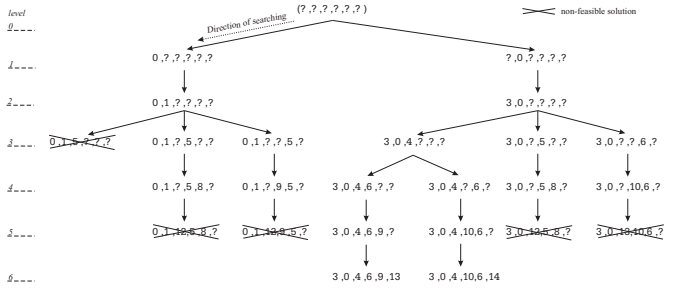

Fig. 4. Illustration of the branching procedure.

A. Bounding in the Search Tree

A bounding mechanism reduces the size of the search tree (used in the step 1 of *vertex_exploration* procedure). At the same time the bounding mechanism cannot eliminate any vertex on the unique path from the root to S^*, the optimal solution.

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_d, the current time on processor P_d. Therefore, if $s_i - h_d \geq w_{ji}$ 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 Section IV-B.

The basic bounding can be further extended. Our B&B algorithm uses 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 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. Due to the pagelimit, we have put this part in the Appendices A, B and C.

B. Scheduling Anomaly

The branching mechanism determines only order of tasks scheduled as soon as possible according to Equation (6). 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 than 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 (see example in Figure 5).

The test is performed via shifting T_i by L_j and by recalculation of start times of other scheduled tasks while Equation (6) is used to satisfy feasibility with respect to forward and backward edges. This "shifting" can cause increase of s_j by value in $\langle 0, L_j \rangle$. 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 the recalculated schedule S^N. Decision, whether for a given unfeasible schedule S there exists a 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 D or it can be formulated as LP problem solvable in polynomial time.

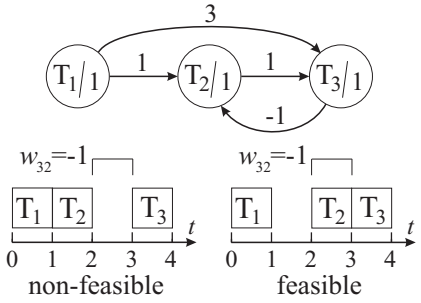

Fig. 5. An example of scheduling anomaly.

V. Experimental Results

Presented scheduling algorithms were implemented and tested in C language. Integer linear program was solved by non-commercial ILP solver tool LP_SOLVE 4.0 [17].

To compare ILP method with B&B algorithm we measure average CPU times for randomly generated instances. The weights of forward edges w_{ij} were chosen from a uniform distribution on the interval $\langle 1, 16 \rangle$ (and rounded towards nearest integer). The processing time p_i was generated form uniform distribution on the interval $\langle 1, \min_{T_j \in \mathcal{T}}(w_{ij}) \rangle$. And weights of backward edges w_{ji} were chosen from a uniform distribution on the interval $\langle f_{ij}, 2.2 \cdot f_{ij} \rangle$ where f_{ij} is the longest path by forward edges. A dedicated processor for a node was chosen from a uniform distribution on the interval $\langle 1, m \rangle$.

CPU times depends on efficiency of algorithm implementation, compilation and speed of computer. ILP also uses a branch and bound algorithm while solving linear programs in each vertex of its search tree. Therefore, comparison of average number of processed vertices allows to examine results of both methods. Total number of processed vertices of ILP in Table I is given by average value of variable $total\_nodes$ declared in lpkit.h of LP_SOLVE tool.

# tasks	CPU time [s]					# inspected vertices in search tree [-]				
	B&B0	B&B1	B&B2	B&B3	ILP	B&B0	B&B1	B&B2	B&B3	ILP
12	0.0016	0.0008	0.0013	0.0005	0.0244	516	302	338	63	137
14	0.0237	0.0081	0.0112	0.0022	0.1282	5605	2344	2696	264	868
16	0.2229	0.0443	0.0637	0.0116	0.8583	38490	11038	12805	1183	5355
18	5.3345	0.7934	1.0605	0.0835	7.4887	755950	182970	202190	6120	38740
20	133.8341	7.9255	12.0301	0.7664	51.0145	15991000	1396000	1669000	51000	221000

TABLE I

EXPERIMENTAL RESULTS OF SCHEDULING ALGORITHM COMPLEXITY GIVEN BY CPU TIME (MEAN VALUE OVER FIVE HUNDRED RANDOMLY GENERATED SET OF INPUT DATA) AND THE NUMBER OF INSPECTED VERTICES IN SEARCH TREE. B&B0 - ALL FEASIBLE SOLUTIONS, B&B1 - CRITICAL PATH BOUNDING, B&B2 - REMAINING PROCESSING TIME BOUNDING, B&B3 - ALL METHODS TOGETHER, ILP - INTEGER LINEAR PROGRAMMING.

# processors	CPU time [s]			# insp. vertices [-]		
	B&B3	B&B_R	ILP	B&B3	B&B_R	ILP
1	0.084	0.084	7.489	6120	6120	38740
2	0.610	0.266	0.228	50426	6620	1387
3	1.251	0.205	0.086	105420	4340	540
4	2.000	0.132	0.029	181230	2270	160
5	3.838	0.090	0.018	276620	1550	70

(a)

# backward edges	CPU time [s]		# insp. vertices [-]	
	B&B3	ILP	B&B3	ILP
8	0.0686	5.6610	5193	28765
12	0.0440	2.5917	2896	13343
16	0.0298	1.3459	1888	6763
20	0.0129	0.3298	703	669
24	0.0043	0.1041	210	478

(b)

TABLE II

(A) INFLUENCE OF THE NUMBER OF PROCESSORS (FOR 18 NODES WITH 27 FORWARD EDGES AND 9 BACKWARD EDGES) (B) INFLUENCE OF THE NUMBER OF BACKWARD EDGES (FOR 18 NODES WITH 27 FORWARD EDGES ON ONE PROCESSOR). ALL RESULTS ARE MEAN VALUES OVER TWO HUNDRED RANDOMLY GENERATED SET OF INPUT DATA.

Table I shows algorithm complexity for one dedicated processor ($m = 1$). Five hundreds of the scheduling instances have been generated per each number of nodes (12,14,16,18 or 20) and the mean value of CPU time and number of inspected solutions in each scheduling problem is shown in Table I. The scheduling problems were generated at random manner as explained above. The number of forward edges in Table I is equal to $3 \cdot n/2$ and the number of backward edges is equal to $n/2$.

When all bounding methods are combined together (B&B3 in Table I) the number of inspected states is approximately 12% of all feasible solutions for 12 nodes (with 18 forward edges and 6 backward edges) and it is 0.3% of all feasible states for 20 nodes (with 30 forward edges and 10 backward edges). ILP inspects 45% of all feasible schedules in the case of 12 nodes and 1.4% in the case of 20 nodes. Therefore, B&B3 solution is more efficient than ILP solution in the monoprocessor case. Up to our benchmarking experience the commercial ILP tools as CPLEX [18] soloves the scheduling problem about 100 times faster.

The influence of number of processors is illustrated in Table II(a), where column B&B3 indicates a performance of the B&B algorithm solving m–DEDICATED problem as described in Section IV. The column B&B_R indicates the case, when the instance of m–DEDICATED problem is firstly transformed to 1–DEDICATED problem and then solved by B&B algorithm. With growing number of processors, B&B_R behaves much better than B&B3, since it does not suffer from enumeration of multiple partial solutions. Average CPU times of ILP method decrease with number of processors since for fixed number of nodes the ILP model for problem with more dedicated processors has less decision variables ($\sum_{d=1}^{m}(n_d^2 - n_d)/2$). On the other hand B&B algorithm has the opposite behavior since the number of combinations increases with the number of processors.

Table II(b) shows the influence of number of backward edges on the average CPU time and the average number of inspected vertices in the search tree. For both methods holds that with increase of the number of backward edges the time complexity of the problem decreases since each relative deadline limits the state space of feasible solutions and optimum can be found faster. These three benchmarks were also used for the practical verification of both methods while the objective values of C_{max} were compared.

# backward edges	failure [%]		avg. rel. err. [%]	
	H0	H1	H0	H1
4	42,2	15,6	29,01	24,98
6	55,6	23,5	22,43	17,12
8	65,1	30,3	10,71	10,70
10	73,2	32,4	7,60	8,30
12	79,8	33,7	5,05	6,42
14	81,1	35,3	3,41	4,52
16	83,9	38,7	1,77	3,06

TABLE III

COMPARISON WITH HEURISTICS H0 AND H1. INFLUENCE OF THE NUMBER OF BACKWARD EDGES (FOR 18 NODES WITH 27 FORWARD EDGES ON ONE PROCESSOR).

The comparison with polynomial heuristics H0 and H1 [8] is shown in Table III. The first row (failure) shows the percentage of cases when H0, H1 respectively have not found any feasible solution, while both B&B and ILP give the feasible solutions. The second row shows the average relative error of $C_{max}(S^{H0})$ and $C_{max}(S^{H1})$ found by heuristics H0 and H1 respectively, with respect to optimal value of $C_{max}(S^*)$ in cases when H0, H1 respectively have found the feasible solution. The relative error was calculated as $(C_{max}(S^{H0}) - C_{max}(S^*))/C_{max}(S^*)$, $(C_{max}(S^{H1}) - C_{max}(S^*))/C_{max}(S^*)$ respectively. The results in Table III shows limited applicability of heuristics H0 and H1 for in-

stances with growing number of backward edges where B&B algorithm and ILP method have much lower computing time requirements (see Table II(b)).

Implementation note: The B&B algorithm was implemented in C language and tested on a PC Intel Pentium 4, 2.4GHz.

VI. Conclusions

This paper deals with problems related to optimal scheduling of FPGA design with pipelined function units, on–chip processor, arbitrary/restricted number of units, simultaneous memory access and dynamic reconfiguration. We have shown that all these problems can be polynomially reduced to the scheduling problem with precedence delays and relative deadlines.

Further, we have presented our original Branch and Bound algorithm and ILP based solution for the scheduling problem with precedence delays and relative deadlines on the set of dedicated processors. It was shown that ILP based solution can solve problems with simultaneous memory access and dynamic reconfiguration without adding virtual tasks (required by reduction to m–DEDICATED problem) and therefore is more effective. Experimental results show impressive power of rather simple and elegant ILP solution, namely in the case of increasing number of dedicated processors. By comparison with the available heuristics is obvious that the heuristics are not suitable for instances with more relative deadlines. On the other hand, the B&B and ILP solutions have lower computation requirements with increasing number of relative deadlines and therefore the optimal solution can be found faster.

References

[1] R. Bartosinski, M. Daněk, P. Honzík, and R. Matoušek, "Dynamic reconfiguration in FPGA-based SoC designs," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, Monterey, California, USA, 2005, pp. 274–274.

[2] R. Matoušek, M. Tichý, Z. Pohl, J. Kadlec, and C. Softley, "Logarithmic number system and floating–point arithmetics on FPGA," in *Field–Programmable Logic and Applications: Reconfigurable Computing is Going Mainstream*, ser. Lecture Notes in Computer Science, vol. 2438. Berlin: Springer, 2002, pp. 627–636.

[3] *Platform Developers Kit: Pipelined Floating-point Library Manual*, Celoxica Ltd., 2004, http://www.celoxica.com.

[4] J. Błazewicz, K. Ecker, G. Schmidt, and J. Węglarz, *Scheduling Computer and Manufacturing Processes*, 2nd ed. Springer, 2001.

[5] B. Roy, "Contribution de la théorie des graphes à l'étude de certains problèmes linéaires," *C. R. Acad. Sci. Paris*, vol. 248, pp. 2437–2439, 1959.

[6] 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*, vol. 94, no. 1-3, pp. 77–99, May 1999.

[7] J. Hurink and J. Keuchel, "Local search algorithms for a single-machine scheduling problem with positive and negative time-lags," *Discrete Applied Mathematics*, vol. 112, no. 1-3, pp. 179–197, 2001.

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

[9] P. Bratley, M. Florian, and P. Robillard, "Scheduling with earliest start and due date constraints," Naval Res. Logist. Quart. 18, pp. 511–517, 1971.

[10] A. M. Kordon, "Minimizing makespan for a bipartite graph on a single processor with an integer precedence delay," *Operations Research Letters*, vol. 32, no. 6, pp. 557–564, November 2004.

[11] S. Dauzère-Pérès and M. Sevaux, "An exact method to minimize the number of tardy jobs in single machine scheduling," *Journal of Scheduling*, vol. 7, no. 6, pp. 405–420, 2004.

[12] S. K.-S. F. Sourd, "The one-machine problem with earliness and tardiness penalties," *Journal of Scheduling*, vol. 6, no. 6, pp. 533–549, 2003.

[13] C. E. V. T'kindt, F. D. Croce, "Revisiting branch and bound search strategies for machine scheduling problems," *Journal of Scheduling*, vol. 7, no. 6, pp. 429–440, 2004.

[14] C. Steiger, H. Walder, M. Platzner, and L. Thiele, "Online scheduling and placement of real-time tasks to partially reconfigurable devices," in *24th IEEE International Real-Time Systems Symposium (RTSS '03), Cancun, Mexico*, 2003, pp. 224–235.

[15] A. Nahapetian, S. Ghiasi, and M. Sarrafzadeh, "Task scheduling on heterogeneous resources with heterogeneous reconfiguration costs," in *International Conference on Parallel and Distributed Computing and Systems*, November 2003, pp. 916–921.

[16] J. K. Lenstra, A. H. G. R. Kan, and P. Brucker, "Complexity of machine scheduling problems," *Annals of Discrete Mathematics*, vol. 1, pp. 343–362, 1977.

[17] J. C. Kantor, *Mixed Integer Linear Program solver LP_SOLVE 4.0*, ftp://ftp.es.ele.tue.nl/pub/lp_solve/, 1995.

[18] *CPLEX Version 8.0*, ILOG, Inc., 2002, http://www.ilog.com/products/cplex/.

Appendix

A. Critical Path Bounding

This method is used for calculating the start time estimation $\widetilde{s}_j$. It 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 $f_{ij} = \infty$.

When the step 1 of the *vertex_exploration* procedure is performed for the *last scheduled node* T_k in a partial schedule 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.

$$\widetilde{s}_j(S) = f_{kj} + s_k \qquad \forall \text{ activated backward edge } e_{ji} \text{ such that } f_{kj} \neq \infty \tag{8}$$

Then the feasibility of the current partial solution is checked while using adapted Equation (1) for all $\widetilde{s}_j(S)$ computed in Equation (8):

$$s_i - \widetilde{s}_j \geq w_{ji}. \tag{9}$$

The partial solution does not lead to any feasible solution if Equation (9) 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 [4], [9].

Nevertheless, presence of the 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

$$s_i - s_k \geq w_{ji} + f_{kj}. \quad (10)$$

B. Remaining Processing Time Bounding

The second method calculates the estimation $\widetilde{s}_j$ in a different way. It considers the amount of work that must be done on given processor 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 the consideration the depth of the graph G related to the weights of forward edges. On the other hand *Remaining Processing Time Bounding* method also considers 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 Section II). In this sense, the two methods have an 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 on P_d 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}

$$T_l \in \mathcal{T}_P \Leftrightarrow f_{lj} \neq \infty \quad \forall l \neq j : T_l \notin \mathcal{T}_S \text{ and } T_l \in \mathcal{T}_d. \quad (11)$$

$$\widetilde{s}_j(S) = h_d + \sum_{l \in \mathcal{T}_P} p(l) \quad \forall \text{ activated backward edges } e_{ji} \quad (12)$$

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

The estimation $\widetilde{s}_j$ is used in the same way like in Subsection A 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 statistically influence on more solutions.

Similarly to inequality (10), presence of the 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

$$s_i - s_k \geq w_{ji} + p_k + \sum_{l \in \mathcal{T}_P} p(l). \quad (13)$$

C. Using C_{max} for Bounding

Discussed bounding methods reduce the 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 the maximum of current time $max_{d \in <1,m>}(h_d)$ 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

$$\max_{d \in <1,m>} h_d < C_{max}(S^B). \quad (14)$$

Since $max_{d \in <1,m>}(h_d)$ is equal to the start time plus processing time of the latest task T_j on $\mathcal{T}_d$ so

$$s_j \leq C_{max}(S^B) - p_j. \quad (15)$$

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 the condition that $s_i = 0$, the previous equation can be converted to

$$s_i - s_j \geq w_{ji}. \quad (16)$$

Inequality (16) is identical to inequality (1). This reasoning is similar to the basic bounding mentioned in Section IV-A. 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 the original graph G defined in Section II. Since the 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 the original limited graph G are successors of the input node and all sink nodes in the 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 the input node to the output node.

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

$$\mathbf{p}' = [1, \mathbf{p}, 1] \qquad \mathbf{W}' = \begin{pmatrix} -\infty & \mathbf{u} & -\infty \\ -\infty & \mathbf{W} & \mathbf{v} \\ w_{in,out} & -\infty & -\infty \end{pmatrix}. \quad (17)$$

Dimensions of $\mathbf{p}'$ and $\mathbf{W}'$ are $(n+2)$ and $(n+2, n+2)$ respectively. The row vector $\mathbf{u}$ of dimension n, represents connection of the input node to the source nodes of the original limited graph (u_m is equal to 0 if T_m is the source node in the original limited graph, otherwise it is equal to $-\infty$). Similarly $\mathbf{v}$, the column vector of dimension n, represents connection of the sink nodes of the original limited graph to the output node (v_m is equal to p_m if T_m is the sink node in original limited graph, otherwise it is equal to $-\infty$). The dynamic value of $w_{out,in}$ is initialized to $\overline{C}$, the upper bound of C_{max}. $\overline{C}$ can be either given by user due to its expert knowledge or it can be calculated by heuristics [8].

Therefore, the 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

$$w_{in,out} = C_{max}(S') - 1. \tag{18}$$

Value $C_{max}(S')$ is decreased since B&B algorithm looks 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.

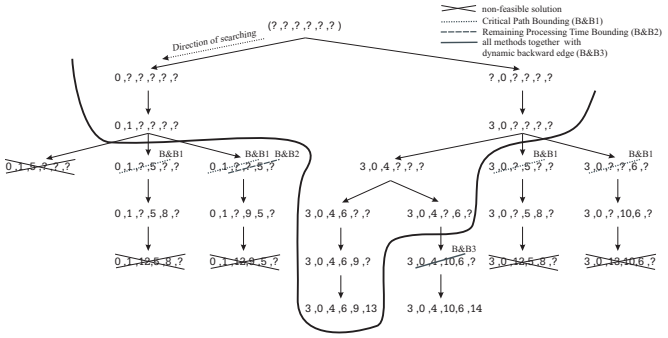

Fig. 6. Illustration of the bounding procedure with C_{max} dynamic backward edge.

Example: (continued) In Figure 6 the example is extended by dynamic C_{max} backward edge. The "B&B3" labels are new bounds issued by the dynamic backward edge. 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 the beginning of B&B algorithm. But when the optimal solution (or solution with C_{max} close to C^*_{max}) is found, C_{max} bounding becomes very efficient.

D. Start time recalculation

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

1) [Initialization] Create a 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 $s^N_k = s^N_k - L_k$. Recalculate start times of tasks belonging to $\mathcal{T}_S$ using Equation (6) and calculate new current time h^N_d. If $h^N_d > h_d$ 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]