Instruction Scheduling with Release Times and Deadlines on ILP Processors

Hui Wu
School of Computer Science and Engineering
The University of New South Wales
Email: huiw@cse.unsw.edu.au

Joxan Jaffar
School of Computing
National University of Singapore
Email: joxan@comp.nus.edu.sg

Jingling Xue
School of Computer Science and Engineering
The University of New South Wales
Email: jxue@cse.unsw.edu.au

Abstract

ILP (Instruction Level Parallelism) processors are being increasingly used in embedded systems. In embedded systems, instructions may be subject to timing constraints. An optimising compiler for ILP processors needs to find a feasible schedule for a set of time-constrained instructions. In this paper, we present a fast algorithm for scheduling instructions with precedence-latency constraints, individual integer release times and deadlines on an ILP processor with multiple functional units. The time complexity of our algorithm is $O(n^2 \log d) + min\{O(de), O(ne)\} + min\{O(ne), O(n^{2.376})\}$*, where* n *is the number of instructions,* e *is the number of edges in the precedence graph and* d *is the maximum latency. Our algorithm is guaranteed to find a feasible schedule whenever one exists in the following special cases: 1) one functional unit, arbitrary precedence constraints, latencies in* $\{0, 1\}$*, integer release times and deadlines; 2) two identical functional units, arbitrary precedence constraints, latencies of* 0*, integer release times and deadlines; 3) multiple identical functional units or multiple functional units of different types, monotone interval-ordered graph, integer release times and deadlines; 4) multiple identical functional units, in-forest, equal latencies, integer release times and deadlines. In case 1), our algorithm improves the existing fastest algorithm from* $O(n^2 \log n) + min\{O(ne), O(n^{2.376})\}$ *to* $min\{O(ne), O(n^{2.376})\}$*. In case 2), our algorithm improves the existing fastest algorithm from* $O(ne + n^2 \log n)$ *to* $min\{O(ne), O(n^{2.376})\}$*. In case 3), no polynomial time algorithm for multiple functional units of different types was known before.*

1 Introduction

ILP (Instruction Level Parallelism) processors are being increasingly used in embedded Systems. Examples include Texas Instruments' TI C6, StartCore's SC140, Philips' Trimedia and HP and STMicroelectronics' Lx [6].

A typical ILP processor consists of multiple parallel functional units of different types. An instruction can be executed only on a functional unit of the same type. Typically, the execution of an instruction takes one processor cycle. However, there are often delays of one or more processor cycles between instructions. These delays, called *latencies*, arise primarily because of off-chip communication and pipelining architecture. For example, if instruction v_i precedes instruction v_j and the latency between v_i and v_j is k (cycles), then instruction v_j can be executed only if k cycles has elapsed after the completion of v_i. Instruction scheduling is a key problem in an optimising compiler for ILP processors. In non-real-time applications, the objective of instruction scheduling is to find a shortest schedule for a set of instructions. This problem is NP-complete even if the target processor has only one functional unit and latencies can be arbitrarily large [12, 13].

In real-time systems, instructions are subject to timing constraints. Typical timing constraints include release times and deadlines. For example, in CNC systems [7], the output to motors must be sent at particular times to maintain high positioning accuracy of the machine tool. A number of researchers have studied the problem of scheduling time-constrained instructions [1–5]. Palem and Simon [4] studied the problem of scheduling instructions with individual deadlines on an ILP processor with multiple identical functional units. Their algorithm is guaranteed to find a feasible schedule in several special cases. Wu and Jaffar [2]

proposed an efficient algorithm for scheduling instructions with individual deadlines on an ILP processor with multiple functional units of different types. Their algorithm is guaranteed to find a feasible schedule in several special cases. Leung et al [1] proposed a polynomial-time algorithm for scheduling instructions with individual release times and deadlines on an ILP processor with multiple identical functional units.

In this paper, we propose a fast algorithm for scheduling instructions with individual release times and deadlines on an ILP processor with multiple functional units of different types. Our algorithm is guaranteed to find a feasible schedule whenever one exists in the following special cases: 1) one functional unit, arbitrary precedence constraints, latencies in $\{0, 1\}$, integer release times and deadlines; 2) two identical functional units, arbitrary precedence constraints, latencies of 0, integer release times and deadlines; 3) multiple identical functional units or multiple functional units of different types, monotone interval-ordered graph, integer release times and deadlines; 4) multiple identical functional units, in-forest, equal latencies, integer release times and deadlines. In case 1), our algorithm improves the existing fastest algorithm [3] from $O(n^2 \log n) + min\{O(ne), O(n^{2.376})\}$ to $min\{O(ne), O(n^{2.376})\}$, where n is the number of instructions and e is the number of edges in the precedence graph. In case 2), our algorithm improves the existing fastest algorithm [1] from $O(ne+n^2 \log n)$ to $min\{O(ne), O(n^{2.376})\}$. In case 3), no polynomial time algorithm for multiple functional units of different types was known before.

The main idea of our algorithm is computing a tighter deadline called the $l^{max}(v_i)$-*successor-tree-consistent deadline* for each instruction v_i, where $l^{max}(v_i)$ is the maximum latency between v_i and all its immediate successors. Given a problem instance P, the $l^{max}(v_i)$-successor-tree-consistent deadline of an instruction v_i is the upper bound on its latest completion time in any feasible schedule for the relaxed problem $P'(v_i)$ where the precedence-latency constraints are represented by the $l^{max}(v_i)$-*successor tree* which is a subset of the original precedence-latency constraints. To make it faster to compute the $l^{max}(v_i)$-successor-tree-consistent deadline for each instruction v_i, we use a number of techniques, namely, forward scheduling, backward scheduling, disjoint set union-find and binary search.

2 Model and Definitions

The target ILP processor M has m functional units F_1, F_2, $\cdots$, F_m of w different types R_1, R_2, $\cdots$, R_w. The type of F_j is denoted by $R(F_j)$. The number of the functional units of type R_i is m_i. An instruction of type R_i can be executed only on a functional unit of the same type. The execution of each instruction takes one processor cycle. A latency exists between two instructions with direct dependency. The precedence-latency constraints are represented by a weighted DAG $G = (V, E, W)$ where V denotes the set of all instructions, E the set of precedence constraints and W the set of all latencies. In addition, each instruction may have a pre-assigned release time and a pre-assigned deadline. If an instruction has no pre-assigned release time, its release time is set to 0. If an instruction has no pre-assigned deadline, its deadline is set to the largest pre-assigned deadline.

The problem of scheduling instructions with individual release times and deadlines on an ILP processor is described as follows. Given a problem instance P: a set $V = \{v_1, v_2, \cdots, v_n\}$ of n UET (Unit Execution Time) instructions, where each instruction has a type $R(v_i) \in \{R_1, R_2, \cdots, R_w\}$, a set of precedence-latency constraints in the form of a weighted DAG $G = (V, E, W)$, where $E = \{(v_i, v_j) : v_j$ is directly dependent on $v_i\}$, $W = \{l_{ij} : (v_i, v_j) \in E$ and $l_{ij} \in \{0, 1, \cdots, d\}$ is the latency between v_i and $v_j\}$, a set $RT = \{r_i : r_i$ is the release time of v_i and r_i is a non-negative integer$\}$ of release times, a set $D = \{d_i : d_i$ is the deadline of v_i and d_i is a positive integer$\}$ of deadlines and the ILP processor M, compute a *feasible schedule* whenever one exists. A schedule $\sigma : V \rightarrow \{0, 1, 2, \cdots, \}$ is called a feasible schedule if it satisfies the following constraints:

1. Precedence-latency constraints: $\forall (v_i, v_j) \in E$ $(\sigma(v_i) + 1 + l_{ij} \leq \sigma(v_j)$.
2. Release time and deadline constraints: $\forall v_i \in V$ $(r_i \leq \sigma(v_i) \leq d_i - 1)$.
3. Resource constraints: For each type R_i, 1) an instruction v_j of type R_i can be executed only on a functional unit of type R_i; 2) $\forall t \in [0, \infty)(|\{v_k \in V : R(v_k) = R_i$ and $\sigma(v_k) \leq t < \sigma(v_k) + 1\}| \leq m_i)$ i.e. the number of instructions of type R_i which are executed at the same time, cannot exceed the number of functional units of type R_i.

σ is called a *valid schedule* if it satisfies constraints 1 and 2.

Given two instructions v_i and v_j, if there is a directed path from v_i to v_j, then v_i is a *predecessor* of v_j and v_j is a *successor* of v_i. Especially, if $(v_i, v_j) \in E$, then v_i is an *immediate predecessor* of v_j and v_j is an *immediate successor* of v_i. If instruction v_i has no immediate successor, then v_i is a *sink instruction*; if v_i has no immediate predecessor, then v_i is a *source instruction*. The set of all successors of instruction v_i is denoted by $Succ(v_i)$. Throughout this paper, we use $l^{max}(v_i)$ to denote the maximum latency between v_i and its immediate successors.

In a weighted DAG G, if there is a directed path P_{ij} from

v_i to v_j, the *path length* of P_{ij} is the sum of its constituent edge weights and the number of instructions in P_{ij}, excluding two end instructions v_i and v_j. The *maximum path length* from v_i to v_j, denoted by l_{ij}^+, is the maximum path length of all paths from v_i to v_j. The maximum path length, also called *transitive latency*, between v_i and v_j specifies that the relative distance between v_i and v_j in any valid schedule must be at least l_{ij}^+ time units.

Definition 2.1. *Given a problem instance P, the* edge-consistent release time *of an instruction v_i, denoted by $r(v_i)$, is recursively defined as follows: $r(v_i) = max\{r_i, max\{r(v_j) + l_{ji} + 1 : v_j$ is an immediate predecessor of $v_i\}\}$. The* edge-consistent deadline *of an instruction v_i, denoted by $d(v_i)$, is recursively defined as follows: $d(v_i) = min\{d_i, min\{d(v_j) - l_{ij} - 1 : v_j$ is an immediate successor of $v_i\}\}$.*

Given a problem instance P, the edge-consistent release times and the edge-consistent deadlines of all instructions can be computed in $O(e)$ time by using breadth-first search, where e is the number of edges in the precedence graph.

Definition 2.2. *Given a non-negative integer k, a weighted DAG $G = (V, E, W)$ and an instruction v_i, the k-successor tree of v_i is a weighted directed tree $WST(G, v_i, k) = (V', E', W')$, where $V' = \{v_i\} \cup \{v_j : v_j$ is a successor of v_i in $G\}$, $E' = \{(v_i, v_j) : v_j$ is a successor of v_i in $G\}$ and $W' = \{l'_{ij} : (v_i, v_j) \in E'$ and $l'_{ij} = l_{ij}^+$ if $l_{ij}^+ < k$, $l'_{ij} = k$ otherwise$\}$.*

In this paper, all time points and the two endpoints of any time interval are non-negative integer.

Definition 2.3. *Given a problem instance P and a type R_i, a time interval $[t_1, t_2)$ is called a* forbidden interval *with respect to R_i if there are $m_i(t_2 - t_1)$ instructions such that their release times and deadlines are within $[t_1, t_2 - 1]$ and $[t_1 + 1, t_2]$, respectively. Given a forbidden interval $[t_1, t_2)$ with respect to R_i, all time points $t_1, t_1 + 1, \cdots, t_2 - 1$ are called* forbidden time points *with respect to R_i. A forbidden interval is called a* maximum forbidden interval *if no longer forbidden interval contains it.*

Intuitively, all instructions of type R_i in a forbidden interval with respect to R_i fully occupy the forbidden interval and cannot be scheduled outside the forbidden interval in any feasible schedule. As a result, no other instruction can be scheduled in this forbidden interval. Forbidden intervals are used to make it faster to compute the $l^{max}(v_i)$-successor-tree-consistent deadline for each non-sink instruction v_i. All maximum forbidden intervals can be computed in $O(n)$ time if we keep two lists of all instructions sorted in non-decreasing order of their release times and in non-decreasing order of their deadlines, respectively.

An *interval-ordered graph* [14] is a DAG $G = (V, E)$, where V is a set of intervals in the real line, $E = \{(v_i, v_j) : v_i, v_j \in V$ and $\forall(x \in v_i$ and $y \in v_j)\ x < y\}$. In an interval-ordered graph G, given any two nodes v_i and v_j, either all predecessors of v_i are also the predecessors of v_j or all predecessors of v_j are also the predecessors of v_i. A *monotone interval-ordered graph* [4] is a weighted interval-ordered graph where for any pair of edges (v_i, v_j) and (v_i, v_k), $l_{ij} \geq l_{ik}$ holds if the predecessors of v_k are also the predecessors of v_j. *In-forest* is a set of disjoint directed trees where each node has at most one immediate successor.

The key idea of our algorithm is computing the $l^{max}(v_i)$-*successor-tree-consistent deadline* for each instruction v_i, where $l^{max}(v_i)$ is the maximum latency between v_i and all its immediate successors. The $l^{max}(v_i)$-successor-tree-consistent deadline of instruction v_i, denoted by d'_i, is typically tighter than its pre-assigned deadline. Specifically, given a problem instance P and an instruction v_i, if v_i is a sink instruction, then d'_i is equal to its pre-assigned deadline; otherwise, d'_i is the upper bound on its latest completion time in any feasible schedule for the relaxed problem instance $P'(v_i)$ which has the same set of instructions as in P with the following constraints:

- Precedence-latency constraints: the $l^{max}(v_i)$-successor-tree $WST(G, v_i, l^{max}(v_i))$.
- Release time constraints: $RT = \{r(v_j)$: the release time of v_j is its edge-consistent release time $r(v_j)\}$.
- Deadline constraints: $D = \{\overline{d}_j$: if v_j is a successor of v_i or the edge-consistent release time of v_j is greater than that of v_i, then the deadline $\overline{d}_j$ of v_j is v_j's $l^{max}(v_j)$-successor-tree-consistent deadline; otherwise, it is v_j's edge-consistent deadline$\}$.
- Resource constraints: the same ILP processor as in P.

To compute the $l^{max}(v_i)$-successor-tree-consistent deadline for a non-sink instruction v_i, our algorithm first computes its *the successor-tree-consistent deadline*. The successor-tree-consistent deadline of v_i is the upper bound on its latest completion time in any feasible schedule for the relaxed problem instance $P(v_i)$. The only difference between $P(v_i)$ and $P'(v_i)$ is that there is no latency constraint in $P(v_i)$.

3 Forward Scheduling and Backward Scheduling

In our algorithm, both forward scheduling and backward scheduling are used to compute the $l^{max}(v_i)$-successor-tree-consistent deadline of each non-sink instruction v_i. Forward scheduling solves the following special instruction scheduling problem: Given a set A of n independent UET

instructions with integer release times and deadlines, find a feasible schedule σ_f on the ILP processor M such that the maximum completion time of all instructions is minimised. Forward scheduling is a greedy scheduling technique where each instruction is scheduled as early as possible. In forward scheduling, an instruction is *ready* at time t if t is not less than its release time. Forward scheduling works as follows. For each time point 0, 1, ..., choose a ready instruction v_k with the smallest deadline to run on an idle functional unit of type $R(v_k)$. Ties are broken arbitrarily. A schedule generated by forward scheduling is called *forward schedule*. A forward schedule can be constructed in $O(n)$ time by using Frederickson's linear time algorithm [9, 15] for scheduling a set of UET tasks with individual integer release times and deadlines on multiple identical processors as follows.

1. Let R_{s_1}, R_{s_2}, $\cdots$,R_{s_p} be the different types of all instructions in A. Partition all instructions in A into p disjoint sets $A_j = \{v_j : v_j \in A \text{ and } R(v_j) = R_{s_j}\}$ $(j = 1, 2, \cdots, p)$.

2. For each $A_j (j = 1, 2, \cdots, p)$, compute a forward schedule σ_{f_j} for A_j on m_{s_j} identical functional units by using Frederickson's algorithm, where m_{s_j} is the number of functional unit of type R_{s_j}.

3. The forward schedule σ_f for A is the union of $\sigma_{f_1}, \sigma_{f_2}$, $\cdots, \sigma_{f_p}$.

Backward scheduling solves the following special instruction scheduling problem: Given a set of n independent UET instructions with individual integer deadlines, find a feasible schedule on the ILP processor M such that the minimum start time of all instructions is maximised. Note that in backward scheduling release time constraints are ignored. In backward scheduling, each instruction is scheduled as late as possible. In backward scheduling, an instruction is *ready* at time t if t is less than its deadline. Backward scheduling works as follows. For each time point $t_{max} - 1$, $t_{max} - 2$, $\cdots$, where t_{max} is the largest deadline of all instructions, choose a ready instruction v_k which has the largest deadline among all ready instructions to run on an idle functional unit of type $R(v_k)$. Ties are broken arbitrarily. The schedule generated by backward scheduling is called *backward schedule*. A backward schedule can be trivially constructed in $O(n)$ time if we keep a sorted list of all instructions in non-increasing order of deadlines. It can be shown that forward scheduling and backward scheduling have the following properties.

Property 3.1. *Given a set V of independent instructions with individual integer release times and deadlines, forward scheduling will find a feasible schedule iff one exists. Furthermore, given a forward schedule σ_f for V, $t_{max}(\sigma_f) = min\{t_{max}(\sigma) : \sigma$ is a feasible schedule for $V\}$ holds, where $t_{max}(\sigma) = max\{\sigma(v_i) : v_i \in V\}$.*

Property 3.2. *Given a set V of independent instructions with individual integer deadlines, backward scheduling will find a feasible schedule iff one exists. Furthermore, given a backward schedule σ_b for V, $t_{min}(\sigma_b) = max\{t_{min}(\sigma) : \sigma$ is a feasible schedule for $V\}$ holds, where $t_{min}(\sigma) = min\{\sigma(v_i) : v_i \in V\}$.*

4 Scheduling Algorithm

In this section, we describe a fast algorithm for scheduling instructions with precedence-latency constraints, individual release times and deadlines on the ILP processor M.

Our algorithm consists of three main steps. The first step is preprocessing. The preprocessing includes computing edge-consistent release times and deadlines for all instructions and sorting 4 arrays which will be used in forward scheduling, backward scheduling and computing the $l^{max}(v_i)$-successor-tree-consistent deadline for each non-sink instruction v_i. The second step is computing the $l^{max}(v_i)$-successor-tree consistent deadline d_i' for each non-sink instruction v_i. The last step is constructing a schedule for P by using list scheduling.

Note that by the definition of the $l^{max}(v_j)$-successor-tree-consistent deadline, if an instruction v_j is a successor of v_i or the edge-consistent release time of v_j is greater than that of v_i, then the $l^{max}(v_j)$-successor-tree-consistent deadline of v_j must be computed before that of v_i. To satisfy this requirement, our algorithm uses an array L of all non-sink instructions which is sorted in non-ascending order of release times. The framework of our algorithm is shown in pseudo code as follows.

```
procedure Instruction_Scheduler(P)
 /* P is a problem instance */

var L_r^≤, L_d^≤, L_d^≥: array of all instructions in P;
var L: array of all non-sink instructions in P;

begin

/****** Preprocessing ******/

 compute the edge-consistent release times and deadlines
 for all instructions;
 for each instruction v_i do
     begin
         set its release time to its edge-consistent release time.
         set its deadline to its edge-consistent deadline.
     end
 sort L_r^≤ in non-decreasing order of release times;
 sort L_d^≤ in non-decreasing order of deadlines;
```

/ Both $L_r^{\leq}$ and $L_d^{\leq}$ are used in forward scheduling */*
sort $L_d^{\geq}$ in non-ascending order of deadlines;
/ $L_d^{\geq}$ is used in backward scheduling */*
sort L in non-ascending order of release times;

/ Compute the $l^{max}(L[i])$-successor-tree-consistent deadline for each non-sink instruction $L[i]$ */*
k= the number of instructions in L;
for $i = 0, 1 \cdots, k-1$ **do**
begin
compute the $l^{max}(L[i])$-successor-tree-consistent deadline of instruction $L[i]$;
if *the $l^{max}(L[i])$-successor-tree-consistent deadline of $L[i]$ is less than its edge-consistent deadline* **then**
begin
set $L[i]$'s deadline to its $l^{max}(L[i])$-successor -tree-consistent deadline;
sort $L_d^{\geq}$ in non-ascending order of deadlines;
sort $L_d^{\leq}$ in non-decreasing order of deadlines;
end
end

*/****** Compute a feasible schedule ******/*

compute a schedule σ for P by using list scheduling;

end

In list scheduling, the priority of each instruction v_i is its $l^{max}(v_i)$-successor-tree-consistent deadline and a smaller number implies a higher priority. List scheduling works as follows. At any time, among all ready instructions, an instruction with the highest priority is chosen and scheduled as early as possible on an idle functional unit of same type as the instruction. Ties are broken arbitrarily. An instruction v_i is ready at time t if 1) for each immediate predecessor v_j of v_i v_j has finished before $t - l_{ji}$, and 2) t is not less than its release time.

Our algorithm computes the $l^{max}(v_i)$-successor-tree-consistent deadline of each non-sink instruction v_i in two steps. In the first step, our algorithms computes the successor-tree-consistent deadline of v_i. In the second step, our algorithms uses binary search and the successor-tree-consistent deadline of v_i to compute its $l^{max}(v_i)$-successor-tree-consistent deadline. Next we describe these two steps in details.

STEP 1: Computing the successor-tree-consistent deadline of v_i.

Let $\sigma_i^{f_1}$ be a forward schedule for $V - \{v_i\} - Succ(v_i)$ and R_{s_1}, R_{s_2}, $\cdots$, R_{s_c} be c different types of all instructions in $Succ(v_i) \cup \{v_i\}$ and $R_{s_c} = R(v_i)$. Given a type $x \in \{R_{s_j} : j = 1, 2, \cdots, c\}$ and a time point t, an instruction set $A(x,t)$ is defined as follows.

- If $x \neq R(v_i)$, then $A(x,t) = \{v_k : v_k \in Succ(v_i)$ and $R(v_k) = x\} \cup \{v_k : v_k \in V - Succ(v_i)$ and $R(v_k) = x$ and $\sigma_i^{f_1}(v_k) \geq t\}$.
- If $x = R(v_i)$, then two cases are distinguished. If one functional unit of type $R(v_i)$ is idle during the time interval $[t-1, t)$ in the forward schedule $\sigma_i^{f_1}$, $A(x,t) = \{v_k : v_k \in Succ(v_i)$ and $R(v_k) = x\} \cup \{v_k : v_k \in V - Succ(v_i) - \{v_i\}$ and $R(v_k) = x$ and $\sigma_i^{f_1}(v_k) \geq t\}$. Otherwise, $A(x,t) = \{v_k : v_k \in Succ(v_i)$ and $R(v_k) = x\} \cup \{v_k : v_k \in V - Succ(v_i) - \{v_i\}$ and $R(v_k) = x$ and $\sigma_i^{f_1}(v_k) \geq t\} \cup \{v_j\}$, where v_j is the instruction of type $R(v_i)$ scheduled at time $t-1$ with the largest deadline in $\sigma_i^{f_1}$.

Let t_{max} be a time point satisfying the following constraint:

For each type $R_{s_j} (j = 1, 2, \cdots, c)$ $min\{\sigma_{b_j}(v_k) : v_k \in A(R_{s_j}, t_{max})\} \geq t_{max}$, where σ_{b_j} is a backward schedule for $A(R_{s_j}, t_{max})$.

By the properties of forward scheduling and backward scheduling, the successor-tree-consistent deadline of v_i is $\min\{d_i, t_{max}\}$.

Our algorithm for computing the successor-tree-consistent deadline of v_i is shown as follows:

1. For each type $R_{s_j} (j = 1, 2, \cdots, c-1)$ compute the maximum time point $t_{max}[j]$ satisfying $min\{\sigma_{b_j}(v_k) : v_k \in A(R_{s_j}, t_{max}[j]\} \geq t_{max}[j]$, where σ_{b_j} is a backward schedule for $A(R_{s_j}, t_{max}[j])$.
2. For the type R_{s_c} compute the maximum time point $t_{max}[c]$ satisfying 1) $t_{max}[c] \leq min\{t_{max}[1], t_{max}[2], \cdots, t_{max}[c-1]\}$, and 2) $min\{\sigma_{b_c}(v_k) : v_k \in A(R_{s_c}, t_{max}[c]\} \geq t_{max}[c]$, where σ_{b_c} is a backward schedule for $A(R_{s_c}, t_{max}[c])$.

It is not difficult to show that the successor-tree-consistent deadline of v_i is $min\{d_i, t_{max}[c]\}$. The maximum time point $t_{max}[j] (j = 1, 2, \cdot, c)$ can be computed by using disjoint set union-find algorithm as follows.

1. Let $S(r_i, R_{s_j}) = \{v_k : v_k \in Succ(v_i)$ and $R(v_k) = R_{s_j}\} \cup \{v_k : v_k \in V - \{v_i\} - Succ(v_i)$ and $\sigma_i^{f_1}(v_k) \geq r_i$, and $R(v_k) = R_{s_j}\}$, $d[j,0] = r_i$ and $d[j,1], d[j,2], \cdots, d[j,c_j]$ be c_j different deadlines of all instructions in $S(r_i, R_{s_j})$ with $d[j,1] < d[j,2] < \cdots < d[j,c_j]$, where r_i is the release time of v_i. Partition the time interval $[d[j,0], d[j,c_j])$ into c_j smaller disjoint intervals $\pi_1 = [d[j,0], d[j,1])$, $\pi_2 = [d[j,1], d[j,2]), \cdots, \pi_{c_j} = [d[j,c_j-1], d[j,c_j])$. An instruction v_k in $S(r_i, R_{s_j})$ belongs to an interval $[x,y)$ if its deadline d_k satisfies $x < d_k \leq y$. Each instruction $v_k \in S(r_i, R_{s_j})$ is assigned a rank, denoted by $rank(v_k)$. If v_k belongs to the interval π_b,

then v_k's rank is b. Each interval $\pi_b(b = 1, 2, \cdots, c_j)$ has two fields: $size$ and $limit$, where $\pi_b.size$ keeps the number of instructions currently scheduled in the interval π_b and $\pi_b.limit$ is the maximum number of instructions which can be scheduled in the interval π_b. Initially, $\pi_b.size$ is set to 0 and $\pi_b.limit$ is set to $m_{s_j}(d[j,b] - d[j,b-1])$, where m_{s_j} is the number of functional units of type R_{s_j}. In addition, a variable u is used to dynamically keep the interval number of the first non-empty interval ($\pi_u.size \neq 0$) from the left. The dynamic update on u is trivial and therefore omitted in the subsequent descriptions.

2. For each instruction $v_k \in \{v_t : v_t \in Succ(v_i)$ and $R(v_t) = R_{s_j}\}$ do the following.

   (a) Find the interval π_b to which v_k belongs by using $find(v_k)$.

   (b) Put v_k into interval π_b by doing 1) $\pi_b.size = \pi_b.size + 1$; 2) if $\pi_b.size = \pi_b.limit$ and $b > 1$, then merge π_b with its left interval π_{b-1} by using $union(\pi_{b-1}, \pi_b)$; 3) if $\pi_b.size = \pi_b.limit$ and $b = 1$, then no feasible schedule exists for $P(v_i)$. As a result, no feasible schedule exists for $P'(v_i)$.

3. Let $v_{w_1}, v_{w_2}, \cdots, v_{w_p}$ be all instructions satisfying the following constraints: 1) For each $v_{w_k}(k = 1, 2, \cdots, p)$, both $v_{w_k} \in V - Succ(v_i) - \{v_i\}$ and $R(v_{w_k}) = R_{s_j}$ hold. 2) For any $s, t \in [1 : p]$, if $s < t$, then either $\sigma_i^{f_1}(v_{w_s}) < \sigma_i^{f_1}(v_{w_t})$ or $(\sigma_i^{f_1}(v_{w_s}) = \sigma_i^{f_1}(v_{w_t})$ and $d_{w_s} \leq d_{w_t})$ holds, where d_{w_s} and d_{w_t} are the deadlines of v_{w_s} and v_{w_t}, respectively. Note that the sorted list of $v_{w_1} v_{w_2} \cdots v_{w_p}$ can be constructed in $O(n)$ time.
   Let TC be the condition defined as follows:

   - If $R_{s_j} \neq R(v_i)$, then TC is $\sigma_i^{f_1}(v_{w_k}) < d[j,u] - \lceil(\pi_u.size/m_{s_j})\rceil$ or $d[j,u] - \lceil(\pi_u.size/m_{s_j})\rceil < r_i$; otherwise, it is $(\sigma_i^{f_1}(v_{w_k}) < d[j,u] - \lceil(\pi_u.size/m_{s_j})\rceil$ and one functional unit of type $R(v_i)$ is idle during the time interval $[d[j,u] - \lceil(\pi_u.size/m_{s_j})\rceil - 1, d[j,u] - \lceil(\pi_u.size/m_{s_j})\rceil)$ in $\sigma_i^{f_1}$ or $d[j,u] - \lfloor(\pi_u.size/m_{s_j})\rfloor < r_i$.

   For each instruction $v_{w_k}(k = p, p-1, \cdots, 1)$, do the following: If the condition TC holds, jump out of the loop; otherwise, remove v_{w_k} from the forward schedule $\sigma_i^{f_1}$ and put it into the interval to which it belongs as follows.

   (a) Find the interval π_b to which v_{w_k} belongs by using $find(v_{w_k})$.

   (b) Put v_{w_k} into interval π_b by doing 1) $\pi_b.size = \pi_b.size + 1$; 2) if $\pi_b.size = \pi_b.limit$ and $b > 1$, then merge π_b with its left interval π_{b-1} by using $union(\pi_{b-1}, \pi_b)$; 3) if $\pi_b.size = \pi_b.limit$ and $b = 1$, then no feasible schedule exists for $P(v_i)$. As a result, no feasible schedule exists for $P'(v_i)$.

   When the loop terminates normally, $t_{max}[j] = d[j,u] - \lceil(\pi_u.size/m_{s_j})\rceil$.

STEP 2: Computing the $l^{max}(v_i)$-successor-tree-consistent deadline of v_i.

Let $a = max\{r_i, t_{max} - l^{max}(v_i) - 1\}$, and $b = min\{d_i - 1, t_{max} - 1\}]$. We first check if the $l^{max}(v_i)$-successor-tree-consistent deadline of v_i falls within the interval $[a, b]$ by using binary search. Binary search cannot be performed before all maximum forbidden intervals with respect to $R(v_i)$ have been removed. The procedure for computing d_i' is shown as follows.

1. Find all forbidden intervals with respect to v_i for the relaxed problem instance $P'(v_i)$ excluding v_i and let $B[0], B[1], \cdots, B[r]$ be $r + 1$ all different non-forbidden time points in the interval $[a, b]$ with $B[j - 1] < B[j](j = 1, 2, \cdots, r)$;

2. Perform binary search over the interval $[B[0], B[r]]$ to find the largest $B[j]$ such that instruction v_i can be scheduled at $B[j]$ in a feasible schedule for the relaxed problem instance $P'(v_i)$.

Notice that if v_i cannot be scheduled at $B[k]$, then it cannot be scheduled at any time point $B[t](r > t > k)$ in any feasible schedule for $P'(v_i)$. Therefore, binary search can be used to find the the largest $B[j]$ such that instruction v_i can be scheduled at $B[j]$ in a feasible schedule for the relaxed problem instance $P'(v_i)$. To check if v_i can be scheduled at a time point $B[j]$, we simply set the release time of v_i to $B[j]$ and find a forward schedule for $P'(v_i)$. If the forward schedule is feasible, then v_i can be scheduled at $B[j]$; otherwise, v_i cannot be scheduled at $B[j]$ in any feasible schedule for $P'(v_i)$.

If either the $l^{max}(v_i)$-successor-tree-consistent deadline of v_i is not found to be within $[a, b]$ or $a > b$ holds, find a time point t_{max} satisfying the following constraints:

1. $r_i \leq t_{max} < a$.

2. $[t_{max}, t_{max} + 1]$ is not a forbidden interval.

3. $[t_{max} + 1, a]$ is a forbidden interval with respect to v_i or $t_{max} = a - 1$.

If such a t_{max} exists, it is the $l^{max}(v_i)$-successor-tree-consistent deadline of v_i. Otherwise, no feasible schedule for the relaxed problem instance $P'(v_i)$. As a result, no

feasible exists for the original problem instance $P(v_i)$.

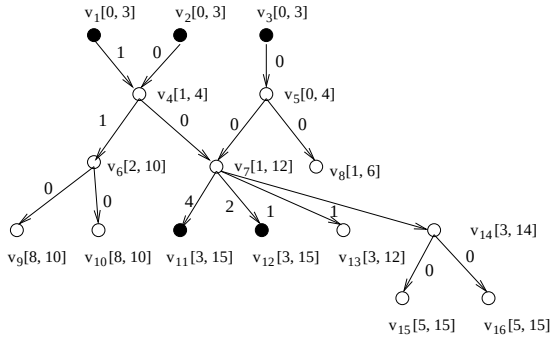

Figure 1. A problem instance P

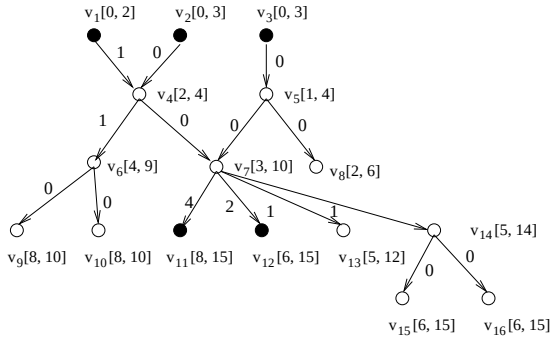

Figure 2. The edge-consistent release times and deadlines in P

Example 1 Consider a problem instance P with 14 instructions and an ILP processor with two heterogeneous functional units F_1 and F_2. The precedence-latency constraints, release times and deadlines are shown in Figure 1 where a filled node denotes an instruction which must be executed on F_1 and a non-filled node represents an instruction which must be executed on F_2. x and y in $[x, y]$ are the pre-assigned release time and deadline of the corresponding instruction, respectively.

First, our algorithm computes the edge-consistent release times and deadlines for all instructions. In Figure 2, x and y in $[x, y]$ are the edge-consistent release time and edge-consistent deadline of the corresponding instruction, respectively.

Next, our algorithm computes the $l_{max}(v_i)$-successor-tree-consistent deadline for each non-sink instruction v_i in non-increasing order of their edge-consistent release times. Suppose that our algorithm has computed the 0-successor-tree consistent deadline of v_6 which is 7, we show how our algorithm computes the 4-successor-tree-consistent deadline d'_7 of v_7. By the definition, d'_7 is the upper bound of the latest completion of v_7 in any schedule for the relaxed problem instance $P'(v_7)$ shown in Figure 3. Our algorithm computes d'_7 in two steps:

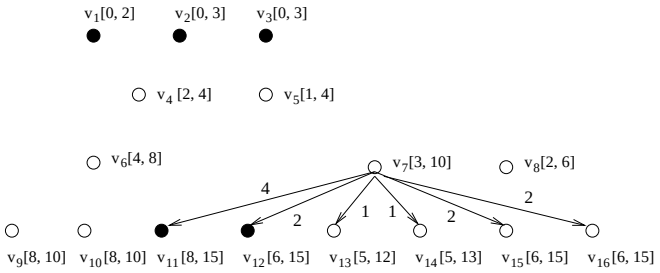

Figure 3. The relaxed problem instance $P'(v_7)$

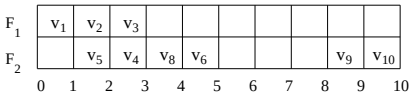

Figure 4. A forward schedule for $V - Succ(v_7) - \{v_7\}$

1. Compute the successor-tree-consistent deadline of v_7. First, our algorithm computes a forward schedule for $V - Succ(v_7) - \{v_7\}$ shown in Figure 4. Next it computes a backward schedule for $Succ(v_7)$ shown in Figure 5. Lastly it applies disjoint union-find algorithm to find the successor-tree-consistent deadline which is 11.

2. Compute d'_7. First, our algorithm finds all forbidden time points within $[a, b]$ and stores all the non-forbidden time points within $[a, b]$ in the array B, where $a = 6$, $b = 10$. In this case, two forbidden time points are 8 and 9. Therefore, we have $B[0] = 6$, $B[1] = 7$ and $B[2] = 10$. Next, it applies binary search to find the latest completion time of v_7 in the relaxed problem instance $P'(v_7)$ which is 7.

The $l_{max}(v_i)$-successor-tree-consistent deadline of each instruction v_i is shown in Figure 6 where y in $[x, y]$ beside each instruction v_i is the $l_{max}(v_i)$-successor-tree-consistent deadline of v_i.

Lastly, our algorithm uses list scheduling to compute a schedule for the original problem instance P. The schedule which is feasible is shown in Figure 7.

By using induction and the properties of forward scheduling and backward scheduling, we can prove the fol-

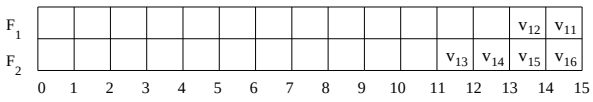

Figure 5. A backward schedule for $Succ(v_7)$

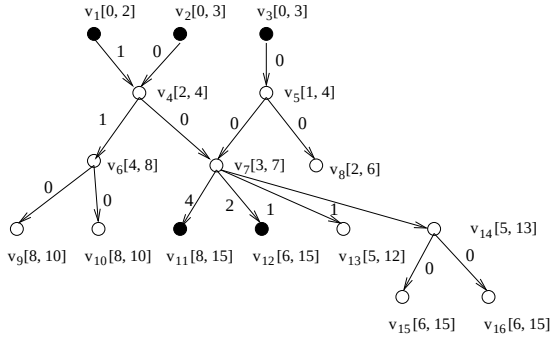

Figure 6. The $l_{max}(v_i)$-successor-tree-consistent deadlines

lowing lemma. The proof is omitted due to the space limitation.

Lemma 4.1. *Given a problem instance P, each instruction v_i must be completed before its $l^{max}(v_i)$-successor-tree-consistent deadline in any feasible schedule for P.*

Theorem 4.1. *Our scheduling algorithm computes a feasible schedule whenever one exists in the following special cases.*

1. *Arbitrary DAG, latencies in $\{0, 1\}$, individual integer release times and deadlines, and one functional unit.*
2. *Arbitrary DAG, latencies of 0, individual integer release times and deadlines, and two identical functional units.*
3. *Monotone interval-ordered graph, arbitrary latencies, individual integer release times and deadlines, and multiple functional units of different types or multiple identical functional units.*
4. *In-forest, equal latencies, individual integer release times and deadlines, and multiple identical functional units.*

Proof Suppose that there exists a feasible schedule σ', but a schedule σ computed by our algorithm is not feasible. Let v_k be the first late instruction and t the earliest integer time point satisfying 1) there are $m_k(\sigma(v_k) - t)$ instructions scheduled in the time interval $[t, \sigma(v_k))$ on m_k functional unit of type $R(v_k)$ in σ, where m_k is the number of functional units of type $R(v_k)$, and 2) for each instruction

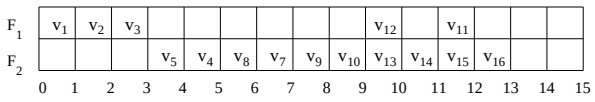

Figure 7. A forward schedule for the original problem instance P

v_j of type $R(v_k)$ scheduled in the time interval $[t, \sigma(v_k))$, $d'_j \leq d'_k$ holds. Let $S = \{v_k\} \cup \{v_j : t \leq \sigma(v_j) < \sigma(v_k)$ and $R(v_j) = R(v_k)\}$. If $t = 0$, then by pigeon hole principle, there must be a late instruction in any schedule for P, which contradicts the assumption. Otherwise, consider the following special cases.

1. Arbitrary DAG, latencies in $\{0, 1\}$, individual integer release times and deadlines and one functional unit. Let v_i be the instruction scheduled in time interval $[t-2, t-1)$. Consider all possible cases.
   (a) No instruction is scheduled in time interval $[t-2, t-1)$ or $d'_i > d'_k$. In this case, by the greediness of list scheduling, the release times of all instructions in S must be greater than or equal to t. Therefore, by pigeonhole principle, at least one instruction must be late in any feasible schedule for P, which contradicts the assumption.
   (b) $d'_i \leq d'_k$. Consider the two possible cases.
      i. The release times of all instructions in S are greater than or equal to t. By pigeonhole principle, there must be at least one late instruction in any feasible schedule for P, which contradicts the assumption.
      ii. There is at least one instruction whose release time is less than or equal to $t-1$. In this case, all instructions whose release times are less than or equal to $t-1$ must be the successors of v_i. By our algorithm for computing the $l^{max}(v_i)$-successor-tree-consistent deadline, v_i must be also late with respect to its $l^{max}(v_i)$-successor-tree-consistent deadline in the schedule σ, which contradicts the assumption that v_k is the first late instruction.
2. Arbitrary DAG, latencies of 0, individual integer release times and deadlines, and two identical functional units. Consider all possible cases.
   (a) No instruction is scheduled in the time interval $[t, t-1)$ or the $l^{max}(v_j)$-successor-tree-consistent deadline of each instruction v_j scheduled in $[t, t-1)$ is greater than d'_k. By the greediness of list scheduling, the release times of all

instructions in S must be greater than or equal to t. By pigeon hole principle, there must be at least one late instruction in any feasible schedule for P, which contradicts the assumption.

(b) There is an instruction v_i scheduled in the time interval $[t, t-1)$ with $d'_i \leq d'_k$. In this case, for each instruction $v_j \in S$, either v_j is a successor of v_i or $r_j \geq t$. If no instruction in S is the successor of v_i, by pigeon hole principle, there must be at least one late instruction in any feasible schedule for P, which contradicts the assumption. Otherwise, by our algorithm for computing successor-tree-consistent deadlines, v_i must be also late with respect to its $l^{max}(v_i)$-successor-tree-consistent in σ, which contradicts the assumption that v_k is the first late instruction.

3. Monotone interval-ordered graph, arbitrary latencies, individual integer release times and deadlines, and multiple functional units of different types or multiple identical functional units.
   Let S_1 be the set of all instructions in S whose release times are less than t and v_r an instruction in S_1 which has the minimum number of predecessors. Since v_r cannot start before t in σ, there must exist an immediate predecessor v_s of v_r such that v_s prevents v_r from starting before t in σ due to the latency l_{sr}. By the property of monotone interval-ordered graph, v_s is the predecessor of all instruction in S_1. Let $S_2 = \{v_j : v_j \in S_1$ and v_j is an immediate successor of $v_s\}$. By the definition of monotone interval-ordered graph, for each instruction $v_j \in S_2$, $l_{sj} \geq l_{sr}$ holds. Since each instruction in $S_1 - S_2$ must be a successor of some instruction in S_1, for each instruction $v_j \in S_1$, $l^+_{sj} \geq l_{sr}$ also holds. By our algorithm for computing successor-tree-consistent deadlines, v_s must be also late with respect to its $l^{max}(v_s)$-successor-tree-consistent deadline in the schedule σ, which contradicts the assumption that v_k is the first late instruction.

4. In-forest, equal latencies, individual integer release times and deadlines, and multiple identical functional units. The proof for this special case is essentially the same as in [11].

Our algorithm for computing the successor-tree-consistent deadlines uses disjoint set union-find algorithm. Since the union tree in this case is a chain, we can use Gabow's linear time union-find algorithm [15]. Therefore, for each non-sink instruction v_i, it takes $O(n)$ time to compute the successor-tree-consistent deadline for v_i, where n is the number of instructions. After the successor-tree-consistent deadline of each non-sink instruction v_i has been computed, our algorithm uses binary search to compute the $l^{max}(v_i)$-successor-tree-consistent deadline for v_i. The binary search takes $O(n \log d)$ time, where d is the maximum latency. In addition, maintaining two sorted arrays $L_d^{\geq}$ and $L_d^{\leq}$ takes $O(n)$ time because each time only $L[i]$'s deadline is changed. Therefore, it takes $O(n^2 * \log d)$ time to compute the $l^{max}(v_i)$-successor-tree-consistent deadlines for all non-sink instructions. The transitive closure can be computed in $O(ne)$ time by n depth-first searches, where e is the number of edges in the precedence graph. Alternatively, it can be reduced to matrix multiplication [17], which is $O(n^{2.376})$ [18]. Moreover, the $l^{max}(v_i)$-successor-tree for each instruction can be computed in $min\{O(ne), O(ed)\}$ time. Therefore, it is easy to show the following theorem.

Theorem 4.2. *The time complexity of our algorithm is* $min\{O(ne),\ O(n^{2.376})\} + min\{O(ed)\ ,O(ne)\} + O(n^2 \log d)$.

5 Conclusion

We proposed a fast algorithm for scheduling instructions in a basic block with precedence-latency constraints, timing constraints in the form of individual integer release times and deadlines on an ILP processor. The key idea of our scheduling algorithm is computing the $l^{max}(v_i)$-successor-tree-consistent deadline for each instruction. To make it faster to compute the $l^{max}(v_i)$-successor-tree-consistent deadline for each non-sink instruction v_i, we use a number of techniques, namely, forward scheduling, backward scheduling, disjoint set union-find and binary search. Our algorithm is guaranteed to find a feasible schedule whenever one exists in a number of special cases. In the first special case where the processor has only one functional unit and the maximum latency is 1, our algorithm improves the existing fastest algorithm [3] from $O(n^2 \log n) + min\{O(ne), O(n^{2.376})\}$ to $min\{O(ne), O(n^{2.376})\}$. In the second special case where the ILP processor has only two identical functional units, our algorithm improves the existing fastest algorithm [1] from $O(ne + n^2 \log n)$ to $min\{O(ne), O(n^{2.376})\}$. The first polynomial time algorithm for this special case proposed by Garey and Johnson [8] runs in $O(n^3)$ time. In the third special case where the precedence-latency constraints can be represented as a monotone interval-ordered graph and the ILP processor has multiple functional units of different types, our algorithm is the first polynomial time algorithm.

Further research on instruction scheduling with timing constraints is expected. One open problem is loop scheduling with individual release times and deadlines on an ILP processor. In non-real-time computing, software pipelining is an efficient approach to employ ILP. In real-time embedded systems, timing satisfaction is the primary consideration. It is interesting to see how release times and deadlines

are handled in software pipelining. Another open problem is scheduling instructions with timing constraints on clustered ILP processors. On a clustered ILP processor such as Lx, communication constraints exist. If two instructions with data dependency are assigned to different clusters, communication delay between these two instructions must be respected in any valid schedule. However, if these two instructions are assigned to the same cluster, there is not communication delay. It is not known if there is any consistency technique for handling communication constraints efficiently.

References

[1] Leung, Allen, Krishna V. Palem and Amir Pnueli. Scheduling Time-Constrained Instructions on Pipelined Processors. ACM Transactions on Programming Languages and Systems 23(1), 73-103, January 2001.

[2] Wu, Hui and Joxan Jaffar. An Efficient Algorithm for Scheduling Instructions with Deadline Constraints on ILP Processors. The Proceedings of the 22nd IEEE Real-Time Systems Symposium, London, UK, Dec. 2001, pp. 235-242.

[3] Wu, Hui, Joxan Jaffar and Roland Yap. Instruction Scheduling with Timing Constraints on A Single RISC Processor with 0/1 Latencies. Lecture Notes in Computer Science, Volume 1894. Springer Verlag. pp. 457-469.

[4] Palem, Krishna. V. and Barbara B. Simon. Scheduling Time-Critical Instructions on RISC machines. ACM Transactions on Programming Languages and Systems 15(4), 632-658, Sept. 1993.

[5] Bogong Su, Shiyuan Ding, Jian Wang, Jinshi Xia. Microcode compaction with timing constraints. Proceedings of MICRO 1987, pp. 59-68.

[6] Paolo Faraboschi et al. Lx: a technology platform for Customizable VLIW Embedded Processing. Proceedings of the 27th International Symposium on Computer Architecture, Vancouver, Canada, 2000, pp. 203-213.

[7] Namyun Kim et al. Visual Assessment of A Real-Time System Design: A Case Study on A CNC Controller. Proceedings of IEEE Real-Time Systems Symposium, Washington, USA, 1996, pp. 300-310.

[8] Garey, M.R. and D.S. Johnson. Two Processor Scheduling with Start-Times and Deadlines. SIAM J. Comput. 6, 1977, pp. 416-426.

[9] Greg N. Frederickson. Scheduling Unit-Time Tasks with Integer Release Times and Deadlines. Information Processing Letters, 16(4), 171-173, 1983.

[10] Bernstein, D. and I. Gertner. Scheduling Expressions on A RISC Processor with A Maximal Delay of One Cycle. ACM Transactions on Programming Languages and Systems, 11(1), 57-66, 1989.

[11] Bruno, J. J. Jones and K. So. Deterministic Scheduling with RISC Processors. IEEE Transactions on Computers, 29, 308-316, April 1980.

[12] Finta, L. and Z. Liu. Single Processor Scheduling Subject to Precedence Delays. Discrete Applied Mathematics 70, 247-266, 1996.

[13] Hennessy, J. and T. Gross. Postpass Code Optimisation of Pipeline Constraints. ACM Transactions on Programming Languages and Systems 5(3), 1983.

[14] Papadimitriou, C. and M. Yannakakis. Scheduling Interval-Ordered Instructions. SIAM Journal on Computing 8, 405-409, 1979.

[15] Gabow H.N and R. E. Tarjan. A Linear-Time Algorithm for A Special Case of Disjoint Set Union. Journal of Computer and System Sciences 30, 209-221, 1985.

[16] Garey, M.R., D.S. Johnson, B.B. Simon and R.E. Tarjan. Scheduling Unit-Time Jobs with Arbitrary Release Times and Deadlines. SIAM J. Comput 10, 256-269, 1981.

[17] Aho, A. V., J. E. Hopcroft and J. D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass., 1974.

[18] Coppersmith, D. and S. Winograd. Matrix Multiplication via Arithmetic Progressions. J. of Symbolic Computation, 9, 251-280, 1990.

[19] http://developer.intel.com/design/itanium.