# Scheduling Time-Constrained Instructions by the Predecessor-Successor-Tree Bound Consistency

**Paper 138**

**Abstract**

We study the following instruction scheduling problem: Given a set of instructions with individual release times, deadlines, precedence-latency constraints, find a feasible schedule whenever one exists on a VLIW processor. This problem is NP-complete even if the VLIW processor has only one functional unit. We propose a novel efficient heuristic for this scheduling problem. Our heuristic is based on a novel bound consistency notion, namely the predecessor-successor-tree bound consistency. The time complexity of our heuristic is $O(n^3 \log d)$, where $n$ is the number of instructions, $e$ is the number of edges in the precedence graph and $d$ is the maximum latency. Our simulation results show that our heuristic performs significantly better than the previous one.

## Introduction

A VLIW (Very Long Instruction Word) processor supports ILP (Instruction Level Parallelism). A typical VLIW processor consists of multiple units of different types. Each unit can execute a fixed set instructions. All units run in parallel so that multiple instructions can be executed at the same time. Examples of contemporary VLIW processors include the TriMedia media processors by NXP (formerly Philips Semiconductors), the SHARC DSP by Analog Devices, the C6000 DSP family by Texas Instruments, the STMicroelectronics ST200 family, and Intel's Itanium IA-64.

To make efficient use of all units of a VLIW processor, the compiler needs an efficient instruction scheduler. In non-real-time computing, the objective of the instruction scheduling is to find a valid schedule with the minimum length. In embedded systems, instructions may be subject to timing constraints. Typical timing constraints are release times and deadlines. The release time specifies the earliest start time of an instruction and the deadline is the latest completion time of an instruction in any feasible schedule. The objective of a compiler for embedded systems based on VLIW processors is to find a feasible schedule that satisfies all timing constraints.

In this paper, we study the problem of scheduling instructions with release times and deadlines on a VLIW processor. This problem is NP-complete even if there is only one functional unit and the maximum latency can be arbitrarily large (Hennessy & Gross 1983; Palem & Simon 1993). We propose a novel efficient scheduling heuristic. Our scheduling heuristic is based on a novel consistency notion, namely predecessor-successor-tree bound

consistency. The time complexity of our heuristic is $O(n^3 \log d)$, where $n$ is the number of instructions and $d$ is the maximum latency in the precedence-latency graph. Our heuristic is guaranteed to find a feasible schedule whenever one exists in a number of special cases. Our simulation results show that our heuristic performs significantly better than the one proposed by Wu et al (Wu, Jaffar, & Xue 2006).

## Problem and Definitions

We assume that 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 may exist between two instructions. 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. All release times and deadlines are integers.

The problem of scheduling instructions with individual release times and deadlines on a VLIW 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 $G = \langle V, E, W\rangle$, 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 non-negative integer$\}$ of deadlines and the processor $M$, compute a *feasible schedule* whenever one exists. A schedule $\sigma : V \to \{0, 1, 2, \cdots, \}$ 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 $R_i$.

$\sigma$ 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 an instruction $v_i$ is denoted by $Succ(v_i)$. The set of all predecessors of $v_i$ is denoted by $Pred(v_i)$. Throughout this paper, we use $l_{max}(v_i)$ to denote the maximum latency between $v_i$ and its immediate successors and $l^{max}(v_i)$ to denote the maximum latency between $v_i$ and its immediate predecessors.

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 1.** *Given a weighted DAG $G = \langle V, E, W \rangle$, an instruction $v_i \in V$ and a non-negative integer $k$, the $k$-successor tree of $v_i$ is a weighted directed tree $ST(G, v_i, k) = \langle V(v_i), E(v_i), W(v_i) \rangle$, where $V(v_i) = \{v_i\} \cup \{v_j : v_j \in Succ(v_i)\}$, $E(v_i) = \{(v_i, v_j) : v_j \in Succ(v_i)\ \}$, and each latency $l_{ij} \in W(v_i)$ is defined as follows: If the transitive latency $l^+_{ij}$ between $v_i$ and $v_j$ in $G$ is greater than $k$, $l_{ij} = k$; otherwise, $l_{ij} = l^+_{ij}$.*

**Definition 2.** *Given a weighted DAG $G = \langle V, E, W \rangle$, an instruction $v_i \in V$ and a non-negative integer $k$, the $k$-predecessor tree of $v_i$ is a weighted directed tree $PT(G, v_i, k) = \langle V(v_i), E(v_i), W(v_i) \rangle$, where $V(v_i) = \{v_i\} \cup \{v_j : v_j \in Pred(v_i)\ \}$, $E(v_i) = \{(v_j, v_i) : v_j \in Pred(v_i)\}$, and each latency $l_{ji} \in W(v_i)$ is defined as follows: If the transitive latency $l^+_{ji}$ between $v_j$ and $v_i$ in $G$ is greater than $k$, $l_{ij} = k$; otherwise, $l_{ji} = l^+_{ji}$.*

**Definition 3.** *Given a problem instance $P$ and an instruction $v_i$ , the edge-consistent release time of $v_i$, denoted by $r'_i$, is recursively defined to be $max\{r_i,\ max\{r'_j + l_{ji} + 1$: $v_j$ is an immediate predecessor of $v_i\}\}$.*

**Definition 4.** *Given a problem instance $P$ and an instruction $v_i$ , the edge-consistent deadline of $v_i$, denoted by $\overline{d}_i$, is recursively defined to be $min\{d_i,\ min\{\overline{d}_j - l_{ij} - 1$: $v_j$ is an immediate successor of $v_i\}\}$.*

It is easy to prove that for each instruction $v_i$ it cannot start before its edge-consistent release time and must be completed by its edge-consistent deadline in any feasible schedule for a problem instance $P$.

**Definition 5.** *Given a problem instance $P$, a time interval $[r, d)$ is a forbidden interval w.r.t. type $R_i$ if there are $m_i(d - r)$ instructions of type R in $P$ whose release times and deadlines are within $[r, d]$. A forbidden interval is a maximum forbidden interval if no forbidden interval is its superset. All time points in $[r, d)$ are called forbidden time points.*

Intuitively, a forbidden interval is an interval that is fully occupied by a set of instructions. We assume that all (time) points are integer.

**Definition 6.** *Given a problem instance $P$ and an instruction $v_i$ and a non-negative integer $k$, the relaxed problem instance $Pred(P, v_i, k)$ of $P$ consists of the same set of instructions as in $P$ with the following constraints:*

1. *Precedence-latency constraints represented by the $k$-predecessor tree $PT(G, v_i, k)$ of $v_i$.*
2. *Release times. The release time of each instruction is the same as in $P$.*
3. *Deadlines. The deadline of each instruction is the same as in $P$.*
4. *Resources constraints. The resource constraints are the same as in $P$.*

**Definition 7.** *Given a problem instance $P$, an instruction $v_i$ and a non-negative integer $k$, the relaxed problem instance $Succ(P, v_i, k)$ of $P$ consists of the same set of instructions in $P$ with the following constraints:*

1. *Precedence-latency constraints represented by the $k$-successor tree $ST(G, v_i, k)$ of $v_i$.*
2. *Release times. The release time of each instruction is the same as in $P$.*
3. *Deadlines. The deadline of each instruction is the same as in $P$.*
4. *Resources constraints. The resource constraints are the same as in $P$.*

**Definition 8.** *Consider a set $S$ of relaxed problem instances of a problem instance $P$ $S = \{Pred(P, v_i, l^{max}(v_i)) : v_i$ is a non-source instruction$\} \cup \{Succ(P, v_i, l_{max}(v_i)) : v_i$ is a non-sink instruction$\}$. Repeat the following procedure until neither the release time nor the deadline of any instruction can be changed:*

1. *For each relaxed problem instance $Pred(P, v_i, l^{max}(v_i))$ , compute the smallest start time $r'_i$ of $v_i$ in all feasible schedules for $Pred(P, v_i, l^{max}(v_i))$ and replace the release time of $v_i$ by $r'_i$ in $P$ and in each relaxed problem instance in $S$.*

2. *For each relaxed problem instance $Succ(P, v_i, l_{max}(v_i))$, compute the largest completion time $d'_i$ of $v_i$ in all feasible schedules for $Succ(P, v_i, l_{max}(v_i))$ and replace the deadline of $v_i$ by $d'_i$ in $P$ and in each relaxed problem instance in $S$.*

*When the above procedure terminates, $P$ is said to be predecessor-successor-tree-bound consistent. The release time of each instruction $v_i$ is called the predecessor-successor-tree-bound-consistent release time and deadline of each instruction $v_i$ is called the predecessor-tree-bound-consistent deadline.*

The predecessor-successor-tree-bound consistency is a key notion behind our instruction scheduling heuristic. It computes tighter release times and deadlines than the successor-tree-consistency proposed in (Wu, Jaffar, & Xue 2006). We will present a fast algorithm for achieving this consistency.

An *interval-ordered graph* (Papadimitriou & Yannakakis 1979) is a DAG $G = \langle V, E\rangle$, 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* (Palem & Simon 1993) 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 $v_j$. An *In-tree* is a directed tree where each node has at most one immediate successor.

## Forward Scheduling and Backward Scheduling

Our algorithm for achieving the predecessor–successor-tree-bound consistency uses both forward scheduling and backward scheduling. Our forward scheduling considers a set of individual instructions with individual release times and deadlines. Its objective is to find a feasible schedule such that the largest completion time of all instructions is minimised. We use the classical EDF (Earliest Deadline First) scheduling strategy as our forward scheduling strategy. Let $V = \{v_i : i = 1, 2, \cdots, p\}$ be a set of independent instructions such that the deadline of $v_i$ is not greater than that of $v_{i+1}(i = 1, 2, p-1)$. The forward scheduling works as follows. For each $v_i(i = 1, 2, \cdots, p)$ schedule it as early as possible on a functional unit that is of the same type as $v_i$. A schedule computed by the forward scheduling is called a *forward schedule*. Given two sorted lists $L_1$ and $L_2$ of all $p$ instructions where $L_1$ is sorted in non-descending order of release times and $L_2$ is sorted in non-descending order of deadlines, a forward schedule can be computed by the disjoint set union-find algorithm in $O(p)$ time (Wu, Jaffar, & Xue 2006).

Our backward scheduling considers a set of independent of instructions with individual release times and deadlines. The objective of backward scheduling is to find a feasible schedule such that the smallest start time of all instructions is maximised. Note that our backward scheduling is different from the one in (Wu, Jaffar, & Xue 2006) where the release times of all instructions are ignored. Let $V = \{v_i : i = 1, 2, \cdots, p\}$ be a set of independent instructions such that the the release time of $v_i$ is not less than that of $v_{i+1}(i = 1, 2, p-1)$. The backward scheduling works as follows. For each $v_i(i = 1, 2, \cdots, p-1)$ schedule it as late as possible on a functional unit that is of the same type as $v_i$. A schedule computed by the backward scheduling is called a *backward schedule*. Next we describe a linear time algorithm for computing a backward schedule for a set of independent instructions.

First we describe a linear time algorithm for computing a backward schedule for a set of instructions of the same type by using the disjoint set union-find algorithm. Let $V_j = \{v_{j_1}, v_{j_2}, \cdots, v_{j_q}\}$ be a set of $q$ instructions with the same type $R_j$, $d_{j_i}(j = 1, 2, \cdots, s)$ be $s$ different deadlines of all instructions in $V_j$ and $\pi_{j_i} = [d_{j_{i-1}}, d_{j_i})(i = 1, 2, \cdots, s)$ be $s$ intervals, where $d_{j_0} = 0$. Assume that there are $m_j$ functional units of type $R_j$. An instruction $v_{j_t}$ is said to belong to an interval $\pi_{j_i}$ if its deadline is equal to the right boundary $d_{j_i}$ of $\pi_{j_i}$. The number of instructions that can be scheduled in each interval $\pi_{j_i}$ is $m_j * (d_{j_i} - d_{j_{i-1}})$. An interval structure has four fields: $left$, $right$, $size$, and $limit$, where $left$ is the left boundary of the interval, $right$ the right boundary of the interval, $limit$ the maximum number of instructions that can be scheduled in the interval, and $size$ the number of instructions that have been scheduled in the interval. Our algorithm for computing a backward schedule for $V_j$ is shown as follows:

```
**algorithm** $BackwardScheduling(V)$
**input**: a set $V$ of $q$ instructions with individual release times
and deadlines
**output**: a backward schedule whenever one exists

**var** $L1$: array of all instructions in $V$ sorted in non-descending
order of deadlines
**var** $L2$: array of all instructions in $V$ sorted in non-increasing
order of release times
{
  let $d_{j_i}(i = 1, 2, \cdots, s)$ be $s$ different deadlines of all
  instructions in $V_j$ such that $d_{j_{i+1}} > d_{j_i}$;
  $d_{j_0} = 0$;
  Let $\pi_{j_i} = [d_{j_{i-1}}, d_{j_i})(i = 1, 2, \cdots, s)$ be $s$ disjoint intervals;
  **for** $i = s-1, s-2, \cdots, 1$ **do**
      $\pi_{j_i}.left = d_{j_{i-1}}$;
      $\pi_{j_i}.right = d_{j_i}$;
      $\pi_{j_i}.limit = m_j * (d_{j_i}, d_{j_{i-1}})$;
      $\pi_{j_i}.size = 0$;
  **for** $i = q, q-1, \cdots, 0$ **do**
      find an interval $\pi_{j_s}$ that the instruction $L2[i]$ belongs to;
      **while** ( $\pi_{j_s}.size = \pi_{j_s}.limit$ and $s \neq 1$) **do**
          // merge $\pi_{j_s}$ with $\pi_{j_{s-1}}$
          $\pi_{j_{s-1}}.right = d_{j_s}.right$;
          $\pi_{j_{s-1}}.limit = \pi_{j_{s-1}}.limit + d_{j_s}.limit$;
          $\pi_{j_{s-1}}.size = \pi_{j_{s-1}}.size + d_{j_s}.size$;
          $s = s-1$;
      **if** ( $\pi_{j_s}.size = \pi_{j_s}.limit$ and $s = 1$ )
          no feasible schedule exists;
      **else**
          $t = \pi_{j_s}.right - \lfloor \pi_{j_s}.size/m_j \rfloor$;
```

```
        schedule $L2[i]$ in the interval $[t-1, t)$;
        $\pi_{j_s}.size = \pi_{j_s}.size + 1$ ;
}
```

A backward schedule on the VLIW processor M can be computed as follows:

1. Partition all instructions into $w$ disjoint sets $S_i(i = 1, 2, \cdots, w)$, where $S_i = \{v_j : v_j \in V$ and the type of $v_j$ is $R_i$ $\}$.
2. Compute a backward schedule $\sigma_i$ for $S_i$ on the processor M.

The union of all backward schedules $\sigma_i(i = 1, 2, \cdots, w)$ is a backward schedule for $V$. Since the union tree is a chain (Gabow & Tarjan 1985), our backward scheduling algorithm takes $O(q)$ time for a set of $q$ instructions.

It can be shown that forward scheduling and backward scheduling have the following properties.

**Property 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 $\sigma_f$ 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 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 $\sigma_b$ for $V$, $t_{min}(\sigma_b) = max\{t_{min}(\sigma) : \sigma$ is a feasible schedule for $V\}$ holds, where $t_{min}(\sigma) = min\{\sigma(v_i) : v_i \in V\}$.*

## The Predecessor-Successor-Tree-Bound Consistency

In this section, we propose a fast algorithm for achieving the predecessor-successor-tree-bound consistency for a problem instance $P$.

Our algorithm is shown in pseudo code as follows.

```
algorithm $SuccTreeConsistentBoundConsistency(P)$

var $L1$: array of all non-source instructions in $P$
var $L2$: array of all non-sink instructions in $P$
var $L3$, $L4$: arrays of all instructions in $P$;
{

// Preprocessing

  for each instruction $v_i$ do
      compute its edge-consistent release time
      and edge-consistent deadline;
      change its release time to its edge-consistent release time;
      change its deadline to its edge-consistent deadline;
  for each non-source instruction $v_i$ do
      compute the $l^{max}(v_i)$-predecessor tree of $v_i$;
  for each non-sink instruction $v_i$ do
      compute the $l_{max}(v_i)$-successor tree of $v_i$;
  sort $L1$ in non-descending order of release times;
  sort $L2$ in non-ascending order of deadlines;
  sort $L3$ in non-descending order of release times;
  sort $L4$ in non-descending order of deadlines;

// Achieve the predecessor-successor-tree-bound consistency

  $quit = 1$;
  while $quit = 1$ do
      $quit = 0$;
      for $i = 0, 1, \cdots, |L1|$ do
          compute the smallest start time $s_i$ of $L1[i]$
          in all feasible schedules for $Pred(P, L1[i], l^{max}(L1[i]))$;
          if the current release time of $L1[i] < s_i$ then
              change the current release time of $L1[i]$ to $s_i$;
              sort $L3$ in non-descending order of release times;
              $quit = 1$;
      for $i = 0, 1 \cdots, |L2| - 1$ do
          compute the largest completion time $f_i$ of $L2[i]$
          in all feasible schedules for $Succ(P, L2[i], l_{max}(L2[i]))$;
          if $f_i$ is less than the current deadline of $L2[i]$ then
              change $L2[i]$'s deadline to $f_i$;
              sort $L4$ in non-descending order of deadlines
              $quit = 1$;

// Adjust release times and deadlines by using forbidden intervals

      Find all the maximum forbidden intervals $[b_j, c_j)$
      $(j = 1, \cdots, k)$ w.r.t. $P$ without precedence constraints;
      for each maximum interval $[b_j, c_j)$ do
          for each instruction $v_i$ in $P$ do
                  case $r_i < c_j$ and $r_i > b_j$ and $d_i > c_j$:
                      change the release time of $v_i$ to $c_j$;
                      $quit = 1$;
                  case $r_i < b_j$ and $d_i > b_j$ and $d_i < c_j$:
                      change the deadline of $v_i$ to $b_j$;
                      $quit = 1$;
      sort $L3$ in non-descending order of release times;
      sort $L4$ in non-descending order of deadlines;
}
```

Our algorithm uses the following two key procedures:

1. Computing the smallest start time of an instruction $v_i$ in all feasible schedules for the relaxed problem instance $Pred(P, v_i, l^{max}(v_i))$.
2. Computing the largest completion time of an instruction $v_i$ in all feasible schedules for the relaxed problem instance $Succ(P, v_i, l_{max}(v_i))$.

To make it faster to achieve the predecessor-successor-tree bound consistency, our algorithm computes the smallest start times of each non-source instruction in the relaxed problem instance $Pred(P, L1[i], l^{max}(v_i))$ in non-descending order of their edge-consistent release times and the largest completion time of each non-sink instruction in the relaxed problem instance $Succ(P, v_i, l_{max}(v_i))$ in non-ascending order of their edge-consistent deadlines.

Given two sorted lists $L3$ and $L4$ of all instructions, the largest completion time of an instruction $v_i$ in all feasible schedules for the relaxed problem instance $Succ(P, v_i, l_{max}(v_i))$ can be computed in $O(n * \log d)$ time as in (Wu, Jaffar, & Xue 2006), where $n$ is the number of instructions and $d$ is the maximum latency. Next, we show how to compute the smallest start time of $v_i$ in all feasible schedules for the relaxed problem instance $Pred(P, v_i, l^{max}(v_i))$ in $O(n * \log d)$ time.

The smallest start time of $v_i$ in all feasible schedules for the relaxed problem instance $Pred(P, v_i)$ in two steps.

**STEP 1: Computing the largest start time of $v_i$ in all feasible schedules for the relaxed problem instance $Pred(P, v_i, 0)$ where all latency constraints are ignored.**

Let $\sigma_i^{b_1}$ be a backward schedule for $V - \{v_i\} - Pred(v_i)$ and $R_{s_1}$, $R_{s_2}$, $\cdots$, $R_{s_c}$ be $c$ different types of all instructions in $Pred(v_i) \cup \{v_i\}$ 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 $B(x, t)$ is defined as follows.

- If $x \neq R(v_i)$, $A(x, t) = \{v_k : v_k \in Pred(v_i)$ and $R(v_k) = x\} \cup \{v_k : v_k \in V - Pred(v_i)$ and $R(v_k) = x$ and $\sigma_i^{b_1}(v_k) \leq t\}$.
- If $x = R(v_i)$, two cases are distinguished. If one functional unit of type $R(v_i)$ is idle during the time interval $[t-1, t)$ in the backward schedule $\sigma_i^{b_1}$, $B(x, t) = \{v_k : v_k \in Pred(v_i)$ and $R(v_k) = x\} \cup \{v_k : v_k \in V - Succ(v_i) - \{v_i\}$ and $R(v_k) = x$ and $\sigma_i^{b_1}(v_k) \leq t\}$. Otherwise, $B(x, t) = \{v_k : v_k \in Pred(v_i)$ and $R(v_k) = x\}$ $\cup$ $\{v_k : v_k \in V - Pred(v_i) - \{v_i\}$ and $R(v_k) = x$ and $\sigma_i^{b_1}(v_k) \leq t\} \cup \{v_j\}$, where $v_j$ is an instruction of type $R(v_i)$ scheduled in the interval $[t-1, t)$ with the largest deadline in $\sigma_i^{b_1}$.

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

For each type $R_{s_j}(j = 1, 2, \cdots, c)$ $min\{\sigma_{b_j}(v_k) : v_k \in B(R_{s_j}, t_{min})\} \leq t_{min}$, where $\sigma_{f_j}$ is a forward schedule for $B(R_{s_j}, t_{min})$.

By the properties of forward scheduling and backward scheduling, the smallest start time of $v_i$ in any feasible schedule for the relaxed problem instance $Pred(P, v_i, 0)$ is $\max\{r_i, t_{min}\}$.

Our algorithm for computing the smallest start time of $v_i$ in all feasible schedules for the relaxed problem instance $Pred(P, v_i, 0)$ is shown as follows:

1. For each type $R_{s_j}(j = 1, 2, \cdots, c-1)$ compute the minimum time point $t_{min}[j]$ satisfying $min\{\sigma_{f_j}(v_k) : v_k \in B(R_{s_j}, t_{min}[j]\} \leq t_{min}[j]$, where $\sigma_{f_j}$ is a forward schedule for $B(R_{s_j}, t_{min}[j])$.

It is not difficult to show that the smallest start time of $v_i$ in all feasible schedules for the relaxed problem instance $Pred(P, v_i, 0)$ is $max\{r_i, max\{t_{min}[j] : j = 1, 2, \cdots, c\}\}$. The minimum time point $t_{min}[j](j = 1, 2, \cdot, c)$ can be computed by using disjoint set union-find algorithm as follows.

1. Let $S(d_i, R_{s_j}) = \{v_k : v_k \in Pred(v_i)$ and $R(v_k) = R_{s_j}\} \cup \{v_k : v_k \in V - \{v_i\} - Pred(v_i)$ and $\sigma_i^{b_1}(v_k) \leq d_i - 1$, and $R(v_k) = R_{s_j}\}$, $r[j, c_j]$ be $d_i$ and $r[j, 0]$, $r[j, 1]$, $\cdots$, $r[j, c_j - 1]$ be $c_j$ different release times of all instructions in $S(d_i, R_{s_j})$ with $r[j, 0] < r[j, 1] < \cdots < r[j, c_j]$, where $d_i$ is the deadline of $v_i$. Partition the time interval $[r[j, 0], r[j, c_j])$ into $c_j$ smaller disjoint intervals $\pi_1 = [r[j, 0], r[j, 1])$, $\pi_2 = [r[j, 1], r[j, 2])$, $\cdots$, $\pi_{c_j} = [r[j, c_j - 1], r[j, c_j])$. An instruction $v_k$ in $S(d_i, R_{s_j})$ belongs to an interval $\pi_s$ if its release time is equal to $r[j, s]$. 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 $\pi_b$, 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 $\pi_b$ and $\pi_b.limit$ is the maximum number of instructions which can be scheduled in the interval $\pi_b$. 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 right. The dynamic update on $u$ is trivial and therefore omitted in the subsequent descriptions.
2. For each instruction $v_k \in \{v_t : v_t \in Pred(v_i)$ and $R(v_t) = R_{s_j}\}$ do the following.
   (a) Find the interval $\pi_b$ to which $v_k$ belongs by using $find(v_k)$.
   (b) Scheduling $v_k$ in the interval $\pi_b$ by doing 1) $\pi_b.size = \pi_b.size + 1$; 2) if $\pi_b.size = \pi_b.limit$ and $b > 1$, then merge $\pi_b$ with its right interval $\pi_{b+1}$ by using $union(\pi_b, \pi_{b+1})$; 3) if $\pi_b.size = \pi_b.limit$ and $b = c_j$, 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 $v_{w_k} \in V - Pred(v_i) - \{v_i\}$ and $R(v_{w_k}) = R_{s_j}$ hold. 2) For any $s, t \in [1 : p]$, if $s < t$, then either $\sigma_i^{b_1}(v_{w_s}) < \sigma_i^{b_1}(v_{w_t})$ or $(\sigma_i^{b_1}(v_{w_s}) = \sigma_i^{b_1}(v_{w_t})$ 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 $\sigma_i^{b_1}(v_{w_k}) < r[j, u] + \lceil(\pi_u.size/m_{s_j})\rceil$ or $r[j, u] + \lceil(\pi_u.size/m_{s_j})\rceil < d_i - 1$; otherwise, it is $(\sigma_i^{b_1}(v_{w_k}) < r[j, u] + \lceil(\pi_u.size/m_{s_j})\rceil$ and one functional unit of type $R(v_i)$ is idle during the time interval $[r[j, u] + \lceil(\pi_u.size/m_{s_j})\rceil - 1, r[j, u] - \lceil(\pi_u.size/m_{s_j})\rceil)$ in $\sigma_i^{b_1}$ or $r[j, u] - \lfloor(\pi_u.size/m_{s_j})\rfloor < d_i - 1$.

   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 backward schedule $\sigma_i^{b_1}$ and put it into the interval to which it belongs as follows:
   (a) Find the interval $\pi_b$ to which $v_{w_k}$ belongs by using $find(v_{w_k})$.
   (b) Schedule $v_{w_k}$ in interval $\pi_b$ by doing 1) $\pi_b.size = \pi_b.size + 1$; 2) if $\pi_b.size = \pi_b.limit$ and $b < c_j$, then merge $\pi_b$ with its right interval $\pi_{b+1}$ by using $union(\pi_b, \pi_{b+1})$; 3) if $\pi_b.size = \pi_b.limit$ and $b = c_j$, 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, $t_{min}[j] = r[j, u] + \lceil(\pi_u.size/m_{s_j})\rceil$.

**STEP 2: Computing the smallest start time of $v_i$ in all feasible schedules for the relaxed problem instance**

$Pred(P, v_i, l^{max}(v_i))$.

Let $a = max\{r_i, t_{min}\}$, and $b = min\{a + l^{max}(v_i), d_i - 1\}$. We first check if the smallest start time of $v_i$ in all feasible schedules for $Pred(P, v_i, l^{max}(v_i))$ 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 the smallest start time of $v_i$ in all feasible schedules for the relaxed problem instance $Pred(P, v_i, l^{max}(v_i))$ 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 smallest $B[j]$ such that instruction $v_i$ can be scheduled at $B[j]$ in a feasible schedule for the relaxed problem instance $Pred(P, v_i, l^{max}(v_i))$.

Notice that if $v_i$ cannot be scheduled at $B[k]$, then it cannot be scheduled at any time point $B[t] (t < k)$ in any feasible schedule for $Pred(P, v_i, l^{max}(v_i))$. Therefore, binary search can be used to find the the smallest $B[j]$ such that instruction $v_i$ can be scheduled at $B[j]$ in a feasible schedule for $Pred(P, v_i, l^{max}(v_i))$. 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 $Pred(P, v_i, l^{max}(v_i))$. 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 $Pred(P, v_i, l^{max}(v_i))$.

If either a start time of $v_i$ is not found to be within $[a, b]$ or $a > b$ holds, find a time point $t_{min}$ satisfying the following constraints:

1. $b < t_{min} \leq d_i - 1$.
2. $[t_{min}, t_{min} + 1]$ is not a forbidden interval.
3. $[b, t_{min} + 1]$ is a forbidden interval with respect to $v_i$.

If such a $t_{min}$ exists, it is the smallest start time of $v_i$ in all feasible schedule for the relaxed problem instance $Pred(P, v_i, l^{max}(v_i))$. Otherwise, no feasible schedule for $Pred(P, v_i, l^{max}(v_i))$. As a result, no feasible exists for the original problem instance $P(v_i)$.

**Theorem 1.** *Given a problem instance $P$, each instruction must be completed by its predecessor-successor-consistent deadline and cannot start before its predecessor-successor-consistent release time in any feasible schedule for $P$.*

**Proof** If an instruction $v_i$ cannot start at time $t$ in any feasible schedule for $Pred(P, v_i, l^{max}(v_i))$, it cannot start at $t$ in any feasible schedule for $P$ either. Similarly, if an instruction $v_i$ must be completed by time $t$ in any feasible schedule for $Succ(P, v_i, l_{max}(v_i))$, it must also be completed by $t$ in any feasible schedule for $P$. Therefore, our algorithm computes a valid bound for each instruction.

**Theorem 2.** *The time complexity of our algorithm is $O(n^3 \log d)$, where $n$ is the number of instructions and $d$ is the maximum latency in the precedence-latency graph.*

**Proof** The time complexity of the preprocessing part is dominated by computing the $l^{max}(v_i)$-predecessor tree of each non-source instruction $v_i$ and the $l_{max}(v_j)$-successor tree of each non-sink instruction $v_j$; It takes $O(e)$ time to compute the $l^{max}(v_i)$-predecessor tree of a non-source instruction $v_i$ by using breadth-first search, where $e$ is the number of edges in the precedence-latency graph. Similarly, it takes $O(e)$ time to compute the $l_{max}(v_i)$-successor tree of a non-sink instruction $v_j$. Therefore, the preprocessing part takes $O(ne)$ time, where $n$ is the number of instructions.

The loop body of the **while** loop consists the following parts:

1. Computing the smallest start times for all non-source instructions. Given the sorted lists $L3$ and $L4$, it takes $O(n \log d)$ time to compute the smallest start time of a non-source instruction $v_i$ in all feasible schedules for $Pred(P, v_i, l^{max}(v_i))$. Since only $L1[i]$'s release time is updated, it takes $O(n)$ time to sort the list $L3$. Therefore this part takes $O(n^2 \log d)$ time.
2. Computing the largest completion times for all non-sink instructions. Given the sorted lists $L3$ and $L4$, it takes $O(n \log d)$ time to compute the largest completion time of a non-sink instruction $v_i$ in all feasible schedules for $Succ(P, v_i, l_{max}(v_i))$ as explained in (Wu, Jaffar, & Xue 2006), where $d$ is the maximum latency. Since only $L1[i]$'s deadline is updated, it takes $O(n)$ time to sort the list $L4$. Therefore this part also takes $O(n^2 \log d)$ time.
3. Adjusting release times and deadlines. Given the two sorted lists $L3$ and $L4$, it takes $O(n)$ time to find all maximum forbidden intervals (Quimper *et al.* 2005). Therefore, this part takes $O(n^2)$ time.
4. Sorting $L3$ and $L4$, which takes $O(n \log n)$.

Notice that after the first iteration of the **while** loop a release time or a deadline is changed only if a new maximum forbidden interval is formed. Since there are only $n$ instructions, the number of new maximum forbidden intervals is at most $n$. Therefore, the number of iterations of the **while** loop is at most $n$. Therefore, the time complexity of our algorithm is $O(n^3 \log d)$. Since $d$ is very small, the time complexity of our algorithm for achieving the predecessor-successor-tree-bound consistency is actually $O(n^3)$.

**Example 1** A problem instance $P$ consists of a set of $14$ instructions with the following constraints:

1. Precedence-latency constraints shown in Figure 1.
2. Individual release times and deadlines shown in Figure 1 where the first element in each bracket denotes the release time and the second element denotes the deadline of the corresponding instruction.
3. The VLIW processor has four functional units F1, F2, F3 and F4. F1 and F2 are of the same type. F3 and F4 are of the same type. Instructions $v_1$ to $v_6$ can only be executed on F1 or F2. Other instructions must be executed on F3 or F4.

The edge-consistent release times and deadlines of all instructions are shown in Figure 2 where the first element in each bracket denotes the edge-consistent release time and the second element denotes the edge-consistent deadline of the corresponding instruction. The predecessor-successor-tree-bound-consistent release times and deadlines of all instructions are shown in Figure 3 where the first element in each bracket denotes the predecessor-successor-tree-bound-consistent release time and the second element denotes the predecessor-successor-tree-bound-consistent deadline of the corresponding instruction.

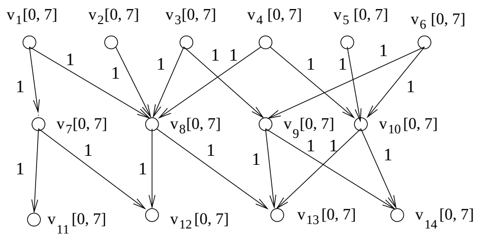


Figure 1: A problem instance $P$

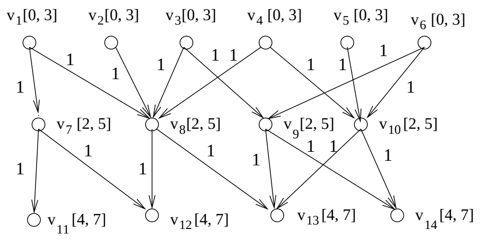


Figure 2: Edge-consistent release times and deadlines in $P$

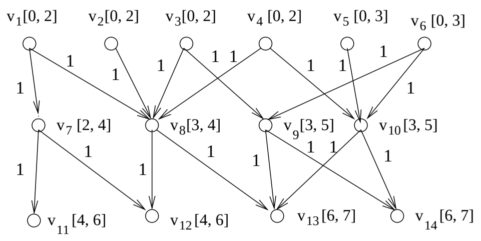
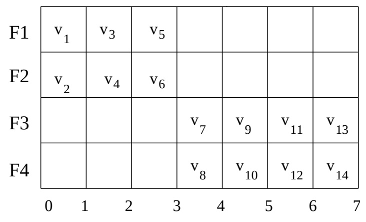


Figure 3: the predecessor-successor-tree-bound-consistent release times and deadlines in $P$

## Instruction Scheduling Heuristic

In this section, we propose a novel heuristic for scheduling instructions with individual release times and deadlines on pipelined processors. Our heuristic uses EDF (Earliest Deadline First) scheduling strategy where the deadline of each instruction is its predecessor-successor-tree-consistent deadline. Given a problem instance $P$, our heuristic works as follows:

1. Run our algorithm for achieving the predecessor-successor-tree consistency for $P$.



| | 0–1 | 1–2 | 2–3 | 3–4 | 4–5 | 5–6 | 6–7 |
|---|---|---|---|---|---|---|---|
| F1 | $v_1$ | $v_3$ | $v_5$ | | | | |
| F2 | $v_2$ | $v_4$ | $v_6$ | | | | |
| F3 | | | | $v_7$ | $v_9$ | $v_{11}$ | $v_{13}$ |
| F4 | | | | $v_8$ | $v_{10}$ | $v_{12}$ | $v_{14}$ |

Figure 4: A feasible schedule for $P$

2. For each instruction set its deadline to its predecessor-successor-tree-bound consistent deadline.
3. At any time choose an instruction with the smallest deadline among all ready instructions and schedule it on an idle pipeline as early as possible. An instruction is ready if all its predecessors have been completed and the maximum latency between this instruction and all its immediate predecessors has elapsed.

As we will show in next section, our heuristic performs significantly better than the heuristic proposed by Wu et al (Wu, Jaffar, & Xue 2006). For all the special cases in which the heuristic proposed by Wu et al is guaranteed to find a feasible schedule, our heuristic is also guaranteed to find one.

**Theorem 3.** *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.*

The proof of this theorem is the same as in (Wu, Jaffar, & Xue 2006) and therefore omitted.

Consider the example given in the previous section. A feasible schedule computed by our scheduling heuristic is shown in Figure 4. Note that for this problem instance the modified deadlines of instructions $v_i(1, 2, \cdots, 6)$ computed by the heuristic proposed by Wu et al (Wu, Jaffar, & Xue 2006) are all equal to $3$. Therefore, their heuristic is not guaranteed to find a feasible schedule.

## Related Work and Simulation Results

The problem of scheduling a set of instructions with precedence-latency constraints, individual release times and deadlines on a VLIW processor was originally studied by Wu et al (Wu, Jaffar, & Xue 2006). The same problem with a simpler processor model where all functional units are identical was studied by Leung et al (Leung, Palem, & Pnueli 2001). The main idea of the heuristic proposed by Wu et al is to compute the modified release time and deadline for each instruction. The modified release time of each instruction is its edge-consistent release time. The modified deadline of each instruction $v_i$ is its largest completion time in all feasible schedules for the relaxed problem instance $Succ(P, v_i)$. Their heuristic is guaranteed to find a feasible schedule in several special cases. Since their heuristic only uses the edge-consistent release times to compute the modified deadlines, the modified deadlines computed by their algorithm are usually larger than the predecessor-successor-tree-bound-consistent deadlines computed by our heuristic.

Our heuristic solves all the special cases their heuristic solves. To compare the performance of our heuristic with that of theirs, we generated $1000$ general instruction scheduling problem instances with tight deadline constraints. Our simulation aimed at generating hard problem instances for which it is difficult to find a feasible schedule. To do so, we took two steps to generate a problem instance as follows:

1. First we generated a problem instance without any release time and deadline by using graph density. The density of a precedence-latency graph is defined as $4e/(n(n-1))$, where $n$ is the number of instructions and $e$ is the number edges in the precedence-latency graph. The densities of all the problem instances ranged from $0.2$ to $0.6$.
2. Next we assigned a release time and a deadline to each instruction in the problem instance as follows:
   (a) Compute a valid schedule for the problem instance without any release time and deadline by using the algorithm proposed by Wu et al. (Wu & Jaffar 2001).
   (b) Let $l$ be the length of the schedule. Assign a release time and a deadline to each instruction such that its release time is less than $l$ and its deadline is less than $l$.

In our simulation, the number of instructions ranged from $50$ to $200$, the number of types ranged from $2$ to $4$ and the number of functional units of each type ranged from $1$ to $4$ and the maximum latency ranged from $0$ to $8$. Among the $1000$ problem instances generated, our heuristic found a feasible schedule for $956$ problem instances while the heuristic proposed by Wu el al found a feasible schedule for $685$ problem instances. We also noticed that whenever their heuristic found a feasible schedule for a problem instance, so did our heuristic. The main reason why our heuristic performs significantly better is that the predecessor-successor-tree-consistent deadline of each instruction is tighter than the deadline computed by their heuristic and better represents its relative degree of urgency.

## Conclusion

In this paper, we have proposed a novel heuristic for scheduling a set of instructions with individual release times and deadlines on a VLIW processor. Our heuristic is underpinned by a novel consistency notion named the predecessor-successor-tree bound consistency. We proposed a fast algorithm for achieving this bound consistency by using a number of techniques such as forward scheduling, backward scheduling and disjoint set union-find algorithm. Our simulation results show that the heuristic significantly outperforms the one proposed by Wu et al.

## References

Bruno, J. J. J., and So, K. 1980. Deterministic scheduling with risc processors. *IEEE Transactions on Computers* 29:308–316.

Gabow, H. N., and Tarjan, R. E. 1985. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences* 30:209–221.

Garey, M. R., and Johnson, D. S. 1976. Scheduling instructions with nonuniform deadlines on two processors. *Journal of ACM* 23(3):461–467.

Garey, M. R., and Johnson, D. S. 1977. Two processor scheduling with start-times and deadlines. *SIAM J. Comput.* 6:416–426.

Garey, M. R.; Johnson, D.; Simon, B.; and Tarjan, R. 1981. Scheduling unit-time jobs with arbitrary release times and deadlines. *SIAM Journal on Computing* 10:256–269.

Hennessy, J., and Gross, T. 1983. Postpass code optimisation of pipeline constraints. *ACM Transactions on Programming Languages and Systems* 5(3):422–448.

Leung, A.; Palem, K. V.; and Pnueli, A. 2001. Scheduling time-constrained instructions on pipelined processors. *ACM Transactions on Programming Languages and Systems* 23(1):73–103.

Malik, A. M.; McInnes, J.; and van Beek, P. 2006. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. In *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence*, 278–287.

Palem, K. V., and Simon, B. B. 1993. Scheduling time-critical instructions on risc machines. *ACM Transactions on Programming Languages and Systems* 15(4):632–658.

Papadimitriou, C., and Yannakakis, M. 1979. Scheduling interval-ordered instructions. *SIAM Journal on Computing* 8:405–409.

Quimper, C.-G.; Golynski, A.; Lpez-Ortiz, A.; and van Beek, P. 2005. An efficient bounds consistency algorithm for the global cardinality constraint. *Constraints* 10(2):115–135.

Simon, B. B., and Warmuth, M. August 1989. A fast algorithm for multiprocessor scheduling of unit length jobs. *SIAM Journal on Computing* 18:690–710.

Wu, H., and Jaffar, J. 2001. An efficient algorithm for scheduling instructions with deadline constraints on ILP processors. In *The Proceedings of the 22nd IEEE Real-Time Systems Symposium*, 235–242.

Wu, H.; Jaffar, J.; and Xue, J. 2006. Instruction scheduling with release times and deadlines on ILP processors. In *Proceedings of The 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 51–60.

Wu, H.; Jaffar, J.; and Yap, R. 2000. Instruction scheduling with timing constraints on a single RISC processor with $0/1$ latencies. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, 457–469.